Computer Networks: A Systems Approach

Release Version 6.1

Peterson and Davie

Nov 26, 2019

TABLE OF CONTENTS

1 Foundation **5**
- 1.1 Applications . . . 6
- 1.2 Requirements . . . 9
- 1.3 Architecture . . . 21
- 1.4 Software . . . 31
- 1.5 Performance . . . 37

2 Direct Links **47**
- 2.1 Technology Landscape . . . 48
- 2.2 Encoding . . . 51
- 2.3 Framing . . . 54
- 2.4 Error Detection . . . 59
- 2.5 Reliable Transmission . . . 65
- 2.6 Multi-Access Networks . . . 77
- 2.7 Wireless Networks . . . 83
- 2.8 Access Networks . . . 95

3 Internetworking **103**
- 3.1 Switching Basics . . . 104
- 3.2 Switched Ethernet . . . 116
- 3.3 Internet (IP) . . . 125
- 3.4 Routing . . . 151
- 3.5 Implementation . . . 169

4 Advanced Internetworking **179**
- 4.1 Global Internet . . . 180
- 4.2 IP Version 6 . . . 190
- 4.3 Multicast . . . 200
- 4.4 Multiprotocol Label Switching . . . 210
- 4.5 Routing Among Mobile Devices . . . 221

5 End-to-End Protocols **229**
- 5.1 Simple Demultiplexor (UDP) . . . 230
- 5.2 Reliable Byte Stream (TCP) . . . 233
- 5.3 Remote Procedure Call . . . 260
- 5.4 Transport for Real-Time (RTP) . . . 276

6 Congestion Control **287**
6.1 Issues in Resource Allocation . . . 288
6.2 Queuing Disciplines . . . 296
6.3 TCP Congestion Control . . . 301
6.4 Advanced Congestion Control . . . 311
6.5 Quality of Service . . . 323

7 End-to-End Data **341**
7.1 Presentation Formatting . . . 342
7.2 Multimedia Data . . . 355

8 Network Security **373**
8.1 Trust and Threats . . . 374
8.2 Cryptographic Building Blocks . . . 375
8.3 Key Predistribution . . . 383
8.4 Authentication Protocols . . . 389
8.5 Example Systems . . . 395

9 Applications **415**
9.1 Traditional Applications . . . 416
9.2 Multimedia Applications . . . 437
9.3 Infrastructure Applications . . . 449
9.4 Overlay Networks . . . 461

PREFACE

It has been nearly ten years since the 5th Edition of *Computer Networks: A Systems Approach* was published. Much has changed in that time, most notably, the explosion of the cloud and smartphone apps onto the scene. In many ways, this is reminiscent of the dramatic affect the Web was having on the Internet when we published the 1st Edition of the book in 1996.

The 6th Edition adapts to the times, but keeps the *Systems Approach* as its north star. In broad strokes, we update and improve this new edition in four main ways:

- We refresh the examples to reflect the current state of the world. This includes deleting anachronisms (e.g., dial-up modem), using popular applications (e.g., Netflix, Spotify) to motivate the problems being addressed, and updating the numbers to represent the state-of-the-art technology (e.g., 10-Gbps Ethernet).
- We connect the dots between the original research that led to the development of technologies like multicast, real-time video streaming, and quality-of-service, and the now-familar cloud applications like GoToMeeting, Netflix, and Spotify. This is in keeping with our emphasis on the design process and not just the end result, which is especially important today since so much the Internet is primarily available in proprietary commercial services.
- We place the Internet in the broader context of the Cloud, and just as importantly, in the context of the commerial forces that are shaping the Cloud. This has minimal impact on the technical details presented throughout the book, but it is discussed in a new *Perspective* section at the end of each chapter. We hope one side-effect of this discussion is to foster an appreciation for the Internet's continuous evolution, and the opportunity for innovation this represents.
- We distill the important principles of network design in a series of *Key Takeaways* throughout the book. Each takeaway is a concise statement of either a general system design rule or a fundamental networking concept, drawing on the examples presented in the surrounding text. Pedagogically, these takeaways correspond to the high-level *learning objectives* for the book.

More specifically, the 6th Edition includes the following major changes:

- New Perspective Section in Chapter 1 introduces the recurring *Cloudification* theme.
- New Section 2.8 describes the *Access Network*, including Passive Optical Networks (PON) and 5G's Radio Access Networks (RAN).
- Refactored topics across Sections 3.1 (*Switching Basics*) and 3.2 (*Switched Ethernet*), including expanded coverage of VLANs.

- Section 3.5 updated to include descriptions of *White-Box Switches* and *Software-Defined Networks* (SDN).
- New Perspective Section in Chapter 3 describes *VXLANs* and the role of overlays in the Cloud.
- Refactored topics across Sections 4.1 (*Global Internet*) and 4.2 (*IP Version 6*).
- New Perspective Section in Chapter 4 describes how the *Cloud* impacts the Internet's structure.
- Section 5.2 expanded to include a discussion of *QUIC*.
- Section 5.3 expanded to include a description of *gRPC*.
- Sections 6.3 and 6.4 updated to include descriptions of *TCP CUBIC, DCTCP,* and *BBR*.
- Section 6.4 expanded to include a description of *Active Queue Management* (AQM).
- Section 7.1 expanded to include a desciption of *Protocol Buffers*.
- Section 7.2 expanded to include a desciption of *HTTP Adaptive Streaming*.
- New Section 8.1 introduces the duality of *Threats* and *Trust*.
- Refactored topics across Sections 8.3 (*Key Predistribution*) and 8.2 (*Authentication Protocols*).
- New Perspective Section in Chapter 8 describes *Decentralized Identity Management* and the role of *Blockchains*.
- Section 9.1 updated to include a description of *HTTP/2*, along with a discussion of *REST*, *gRPC*, and *Cloud Services*.
- Section 9.3 expaned to include a description of modern *Network Management Systems* including the use of *OpenConfig* and *gNMI*.

Organization

To construct a networking course around the material in this book, it can be helpful to understand the overall organization, which can be characterized as having three major parts:

- Conceptual and foundational material, that is, the big ideas at the heart of networking.
- Core protocols and algorithms that illustrate how the foundational ideas are put to practice.
- Advanced material that might or might not fit in any single semester course.

This characterization can be applied at the Chapter level: Chapter 1 is foundational, Chapters 2, 3, 5, and 9 are core, and Chapters 4, 6, 7, and 8 cover more advanced topics.

This characterization can also be applied at the Section level, where roughly speaking, each Chapter advances from basic concepts to specific technology to advanced techniques. For example, Chapter 3 starts by introducing the basics of switched networks (3.1), then covers the specifics of Switched Ethernet and the IP Internet (3.2-3.4), and concludes with an optional discusion of SDN (3.5). Similarly, Chapter 6 starts with foundational ideas (6.1-6.2), then explores TCP congestion control (6.3), and concludes with optional advanced material (6.4-6.5).

Acknowledgements

We would like to acknowledge the following people for their help with new content:

- Larry Brakmo: TCP Congestion Control
- Carmelo Cascone: White-Box Switches
- Charles Chan: White-Box Switches
- Jude Nelson: Decentralized Identity
- Oguz Sunay: Cellular Networks
- Thomas Vachuska: Network Management

Along with the following individuals (github users) for their various contributions and bug fixes:

- Mohammed Al-Ameen
- Andy Bavier
- Manuel Berfelde
- Chris Goldsworthy
- John Hartman
- Diego López León
- Matteo Scandolo
- Mike Wawrzoniak
- (spacewander)
- Arnaud (arvdrpoo)
- Desmond (kingdido999)
- Guo (ZJUGuoShuai)
- Hellman (eshellman)
- Xtao (vertextao)
- Mike Appelman
- Seth (springbov)

Finally, we would like to thank the following reviewers for their many helpful comments and suggestions. Their impact was significant.

- Mark J. Indelicato, Rochester Institute of Technology
- Michael Yonshik Choi, Illinois Institute of Technology
- Sarvesh Kulkarni, Villanova University
- Alexander L. Wijesinha, Towson University

Larry & Bruce
November 2019

CHAPTER

ONE

FOUNDATION

> I must create a System, or be enslav'd by another Man's; I will not Reason and Compare: my business is to Create.
>
> —*William Blake*

Problem: Building a Network

Suppose you want to build a computer network, one that has the potential to grow to global proportions and to support applications as diverse as teleconferencing, video on demand, electronic commerce, distributed computing, and digital libraries. What available technologies would serve as the underlying building blocks, and what kind of software architecture would you design to integrate these building blocks into an effective communication service? Answering this question is the overriding goal of this book—to describe the available building materials and then to show how they can be used to construct a network from the ground up.

Before we can understand how to design a computer network, we should first agree on exactly what a computer network is. At one time, the term *network* meant the set of serial lines used to attach dumb terminals to mainframe computers. Other important networks include the voice telephone network and the cable TV network used to disseminate video signals. The main things these networks have in common are that they are specialized to handle one particular kind of data (keystrokes, voice, or video) and they typically connect to special-purpose devices (terminals, hand receivers, and television sets).

What distinguishes a computer network from these other types of networks? Probably the most important characteristic of a computer network is its generality. Computer networks are built primarily from general-purpose programmable hardware, and they are not optimized for a particular application like making phone calls or delivering television signals. Instead, they are able to carry many different types of data, and they support a wide, and ever growing, range of applications. Today's computer networks have pretty much taken over the functions previously performed by single-use networks. This chapter looks at some typical applications of computer networks and discusses the requirements that a network designer who wishes to support such applications must be aware of.

Once we understand the requirements, how do we proceed? Fortunately, we will not be building the first network. Others, most notably the community of researchers responsible for the Internet, have gone before us. We will use the wealth of experience generated from the Internet to guide our design. This experience is embodied in a *network architecture* that identifies the available hardware and software components and shows how they can be arranged to form a complete network system.

In addition to understanding how networks are built, it is increasingly important to understand how they are operated or managed and how network applications are developed. Almost all of us now have computer networks in our homes, offices, and in some cases in our cars, so operating networks is no longer a matter only for a few specialists. And with the proliferation of smartphones, many more of this generation are developing networked applications than in the past. So we need to consider networks from these multiple perspectives: builders, operators, application developers.

To start us on the road toward understanding how to build, operate, and program a network, this chapter does four things. First, it explores the requirements that different applications and different communities of people place on the network. Second, it introduces the idea of a network architecture, which lays the foundation for the rest of the book. Third, it introduces some of the key elements in the implementation of computer networks. Finally, it identifies the key metrics that are used to evaluate the performance of computer networks.

1.1 Applications

Most people know the Internet through its applications: the World Wide Web, email, social media, streaming music or movies, videoconferencing, instant messaging, file-sharing, to name just a few examples. That is to say, we interact with the Internet as *users* of the network. Internet users represent the largest class of people who interact with the Internet in some way, but there are several other important constituencies.

There is the group of people who *create* the applications—a group that has greatly expanded in recent years as powerful programming platforms and new devices such as smartphones have created new opportunities to develop applications quickly and to bring them to a large market.

Then there are those who *operate* or *manage* networks—mostly a behind-the-scenes job, but a critical one and often a very complex one. With the prevalence of home networks, more and more people are also becoming, if only in a small way, network operators.

Finally, there are those who *design* and *build* the devices and protocols that collectively make up the Internet. That final constituency is the traditional target of networking textbooks such as this one and will continue to be our main focus. However, throughout this book we will also consider the perspectives of application developers and network operators.

Considering these perspectives will enable us to better understand the diverse requirements that a network must meet. Application developers will also be able to make applications that work better if they understand how the underlying technology works and interacts with the applications. So, before we start figuring out how to build a network, let's look more closely at the types of applications that today's networks support.

1.1.1 Classes of Applications

The World Wide Web is the Internet application that catapulted the Internet from a somewhat obscure tool used mostly by scientists and engineers to the mainstream phenomenon that it is today. The Web itself has become such a powerful platform that many people confuse it with the Internet, and it's a bit of a stretch to say that the Web is a single application.

In its basic form, the Web presents an intuitively simple interface. Users view pages full of textual and graphical objects and click on objects that they want to learn more about, and a corresponding new page appears. Most people are also aware that just under the covers each selectable object on a page is bound to

an identifier for the next page or object to be viewed. This identifier, called a Uniform Resource Locator (URL), provides a way of identifying all the possible objects that can be viewed from your web browser. For example,

```
http://www.cs.princeton.edu/llp/index.html
```

is the URL for a page providing information about one of this book's authors: the string `http` indicates that the Hypertext Transfer Protocol (HTTP) should be used to download the page, `www.cs.princeton.edu` is the name of the machine that serves the page, and `/llp/index.html` uniquely identifies Larry's home page at this site.

What most web users are not aware of, however, is that by clicking on just one such URL over a dozen messages may be exchanged over the Internet, and many more than that if the web page is complicated with lots of embedded objects. This message exchange includes up to six messages to translate the server name (`www.cs.princeton.edu`) into its Internet Protocol (IP) address (`128.112.136.35`), three messages to set up a Transmission Control Protocol (TCP) connection between your browser and this server, four messages for your browser to send the HTTP "GET" request and the server to respond with the requested page (and for each side to acknowledge receipt of that message), and four messages to tear down the TCP connection. Of course, this does not include the millions of messages exchanged by Internet nodes throughout the day, just to let each other know that they exist and are ready to serve web pages, translate names to addresses, and forward messages toward their ultimate destination.

Another widespread application class of the Internet is the delivery of "streaming" audio and video. Services such as video on demand and Internet radio use this technology. While we frequently start at a website to initiate a streaming session, the delivery of audio and video has some important differences from fetching a simple web page of text and images. For example, you often don't want to download an entire video file—a process that might take a few minutes—before watching the first scene. Streaming audio and video implies a more timely transfer of messages from sender to receiver, and the receiver displays the video or plays the audio pretty much as it arrives.

Note that the difference between streaming applications and the more traditional delivery of text, graphics, and images is that humans consume audio and video streams in a continuous manner, and discontinuity—in the form of skipped sounds or stalled video—is not acceptable. By contrast, a regular (non-streaming) page can be delivered and read in bits and pieces. This difference affects how the network supports these different classes of applications.

A subtly different application class is *real-time* audio and video. These applications have considerably tighter timing constraints than streaming applications. When using a voice-over-IP application such as Skype or a videoconferencing application, the interactions among the participants must be timely. When a person at one end gestures, then that action must be displayed at the other end as quickly as possible.[1]

When one person tries to interrupt another, the interrupted person needs to hear that as soon as possible and decide whether to allow the interruption or to keep talking over the interrupter. Too much delay in this sort of environment makes the system unusable. Contrast this with video on demand where, if it takes several seconds from the time the user starts the video until the first image is displayed, the service is still deemed satisfactory. Also, interactive applications usually entail audio and/or video flows in both directions, while a streaming application is most likely sending video or audio in only one direction.

Videoconferencing tools that run over the Internet have been around now since the early 1990s but have

[1] Not quite "as soon as possible"... Human factors research indicates 300 ms is a reasonable upper bound for how much round-trip delay can be tolerated in a telephone call before humans complain, and a 100-ms delay sounds very good.

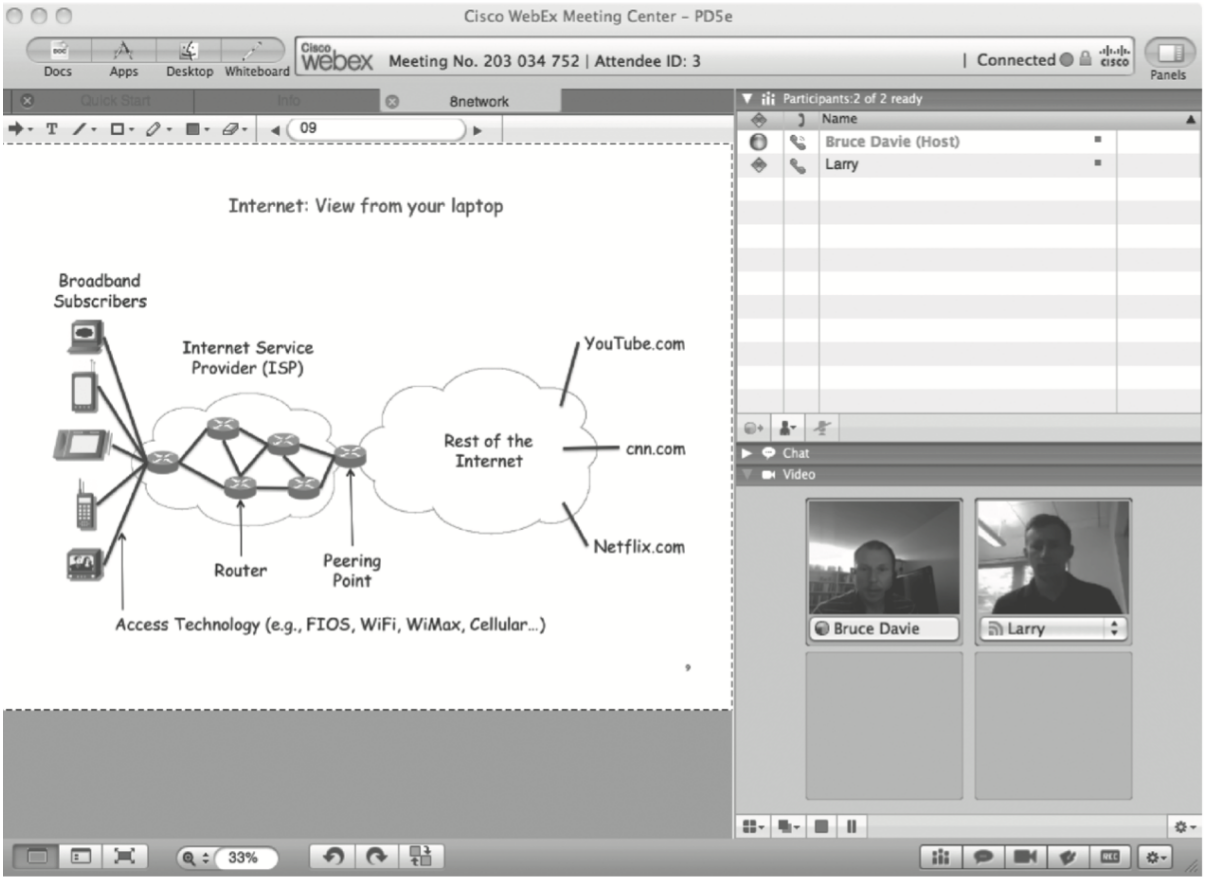


Figure 1.1.: A multimedia application including videoconferencing.

achieved widespread use in the last few years, with several commercial products on the market. An example of one such system is shown in Figure 1.1. Just as downloading a web page involves a bit more than meets the eye, so too with video applications. Fitting the video content into a relatively low bandwidth network, for example, or making sure that the video and audio remain in sync and arrive in time for a good user experience are all problems that network and protocol designers have to worry about. We'll look at these and many other issues related to multimedia applications later in the book.

Although they are just two examples, downloading pages from the web and participating in a videoconference demonstrate the diversity of applications that can be built on top of the Internet and hint at the complexity of the Internet's design. Later in the book we will develop a more complete taxonomy of application types to help guide our discussion of key design decisions as we seek to build, operate, and use networks that such a wide range of applications. The book concludes by revisiting these two specific applications, as well as several others that illustrate the breadth of what is possible on today's Internet.

For now, this quick look at a few typical applications will suffice to enable us to start looking at the problems that must be addressed if we are to build a network that supports such application diversity.

1.2 Requirements

We have established an ambitious goal for ourselves: to understand how to build a computer network from the ground up. Our approach to accomplishing this goal will be to start from first principles and then ask the kinds of questions we would naturally ask if building an actual network. At each step, we will use today's protocols to illustrate various design choices available to us, but we will not accept these existing artifacts as gospel. Instead, we will be asking (and answering) the question of *why* networks are designed the way they are. While it is tempting to settle for just understanding the way it's done today, it is important to recognize the underlying concepts because networks are constantly changing as technology evolves and new applications are invented. It is our experience that once you understand the fundamental ideas, any new protocol that you are confronted with will be relatively easy to digest.

1.2.1 Stakeholders

As we noted above, a student of networks can take several perspectives. When we wrote the first edition of this book, the majority of the population had no Internet access at all, and those who did obtained it while at work, at a university, or by a dial-up modem at home. The set of popular applications could be counted on one's fingers. Thus, like most books at the time, ours focused on the perspective of someone who would design networking equipment and protocols. We continue to focus on this perspective, and our hope is that after reading this book you will know how to design the networking equipment and protocols of the future.

However, we also want to cover the perspectives of two additional stakeholders: those who develop networked applications and those who manage or operate networks. Let's consider how these three stakeholders might list their requirements for a network:

- An *application programmer* would list the services that his or her application needs: for example, a guarantee that each message the application sends will be delivered without error within a certain amount of time or the ability to switch gracefully among different connections to the network as the user moves around.

- A *network operator* would list the characteristics of a system that is easy to administer and manage: for example, in which faults can be easily isolated, new devices can be added to the network and configured correctly, and it is easy to account for usage.
- A *network designer* would list the properties of a cost-effective design: for example, that network resources are efficiently utilized and fairly allocated to different users. Issues of performance are also likely to be important.

This section attempts to distill the requirements of different stakeholders into a high-level introduction to the major considerations that drive network design and, in doing so, identify the challenges addressed throughout the rest of this book.

1.2.2 Scalable Connectivity

Starting with the obvious, a network must provide connectivity among a set of computers. Sometimes it is enough to build a limited network that connects only a few select machines. In fact, for reasons of privacy and security, many private (corporate) networks have the explicit goal of limiting the set of machines that are connected. In contrast, other networks (of which the Internet is the prime example) are designed to grow in a way that allows them the potential to connect all the computers in the world. A system that is designed to support growth to an arbitrarily large size is said to *scale*. Using the Internet as a model, this book addresses the challenge of scalability.

To understand the requirements of connectivity more fully, we need to take a closer look at how computers are connected in a network. Connectivity occurs at many different levels. At the lowest level, a network can consist of two or more computers directly connected by some physical medium, such as a coaxial cable or an optical fiber. We call such a physical medium a *link*, and we often refer to the computers it connects as *nodes*. (Sometimes a node is a more specialized piece of hardware rather than a computer, but we overlook that distinction for the purposes of this discussion.) As illustrated in Figure 1.2, physical links are sometimes limited to a pair of nodes (such a link is said to be *point-to-point*), while in other cases more than two nodes may share a single physical link (such a link is said to be *multiple-access*). Wireless links, such as those provided by cellular networks and Wi-Fi networks, are an important class of multiple-access links. It is always the case that multiple-access links are limited in size, in terms of both the geographical distance they can cover and the number of nodes they can connect. For this reason, they often implement the so-called *last mile*, connecting end users to the rest of the network.

If computer networks were limited to situations in which all nodes are directly connected to each other over a common physical medium, then either networks would be very limited in the number of computers they could connect, or the number of wires coming out of the back of each node would quickly become both unmanageable and very expensive. Fortunately, connectivity between two nodes does not necessarily imply a direct physical connection between them—indirect connectivity may be achieved among a set of cooperating nodes. Consider the following two examples of how a collection of computers can be indirectly connected.

Figure 1.3 shows a pair of shows a set of nodes, each of which is attached to one or more point-to-point links. Those nodes that are attached to at least two links run software that forwards data received on one link out on another. If organized in a systematic way, these forwarding nodes form a *switched network*. There are numerous types of switched networks, of which the two most common are *circuit switched* and *packet switched*. The former is most notably employed by the telephone system, while the latter is used for the overwhelming majority of computer networks and will be the focus of this book. (Circuit switching is,

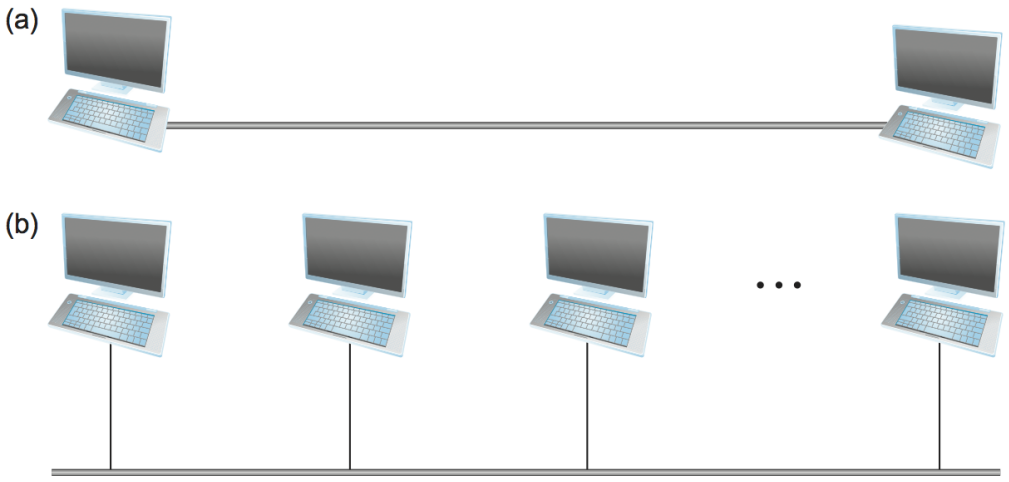


Figure 1.2.: Direct links: (a) point-to-point; (b) multiple-access.

however, making a bit of a comeback in the optical networking realm, which turns out to be important as demand for network capacity constantly grows.) The important feature of packet-switched networks is that the nodes in such a network send discrete blocks of data to each other. Think of these blocks of data as corresponding to some piece of application data such as a file, a piece of email, or an image. We call each block of data either a *packet* or a *message*, and for now we use these terms interchangeably.

Packet-switched networks typically use a strategy called *store-and-forward*. As the name suggests, each node in a store-and-forward network first receives a complete packet over some link, stores the packet in its internal memory, and then forwards the complete packet to the next node. In contrast, a circuit-switched network first establishes a dedicated circuit across a sequence of links and then allows the source node to send a stream of bits across this circuit to a destination node. The major reason for using packet switching rather than circuit switching in a computer network is efficiency, discussed in the next subsection.

The cloud in Figure 1.3 distinguishes between the nodes on the inside that *implement* the network (they are commonly called *switches*, and their primary function is to store and forward packets) and the nodes on the outside of the cloud that *use* the network (they are traditionally called *hosts*, and they support users and run application programs). Also note that the cloud is one of the most important icons of computer networking. In general, we use a cloud to denote any type of network, whether it is a single point-to-point link, a multiple-access link, or a switched network. Thus, whenever you see a cloud used in a figure, you can think of it as a placeholder for any of the networking technologies covered in this book.[1]

A second way in which a set of computers can be indirectly connected is shown in Figure 1.4. In this situation, a set of independent networks (clouds) are interconnected to form an *internetwork*, or internet for short. We adopt the Internet's convention of referring to a generic internetwork of networks as a lowercase *i* internet, and the TCP/IP Internet we all use every day as the capital *I* Internet. A node that is connected to two or more networks is commonly called a *router* or *gateway*, and it plays much the same role as a switch—it forwards messages from one network to another. Note that an internet can itself be viewed as another kind of network, which means that an internet can be built from a set of internets. Thus, we can recursively build arbitrarily large networks by interconnecting clouds to form larger clouds. It can reasonably be argued that this idea of interconnecting widely differing networks was the fundamental innovation of the

[1] The use of clouds to represent networks predates the term *cloud computing* by at least a couple of decades, but there an increasingly rich connection between these two usages, which we explore in the *Perspective* discussion at the end of each chapter.

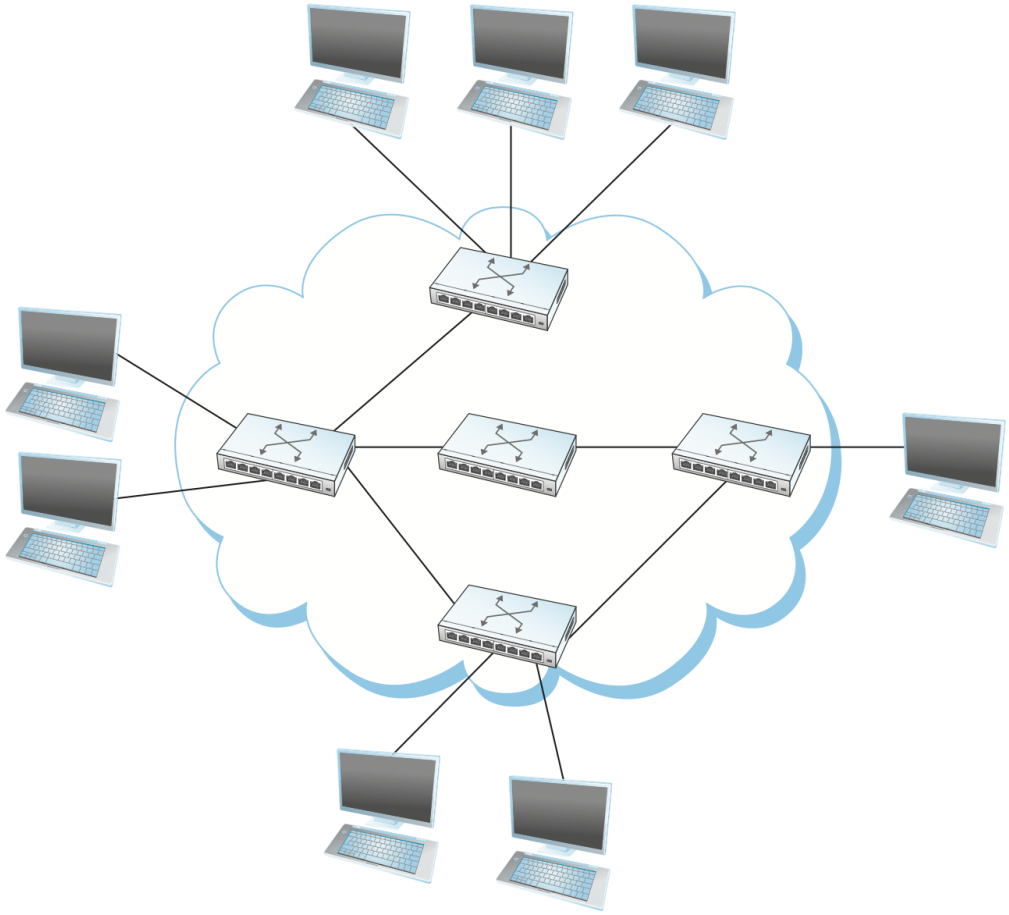


Figure 1.3.: Switched network.

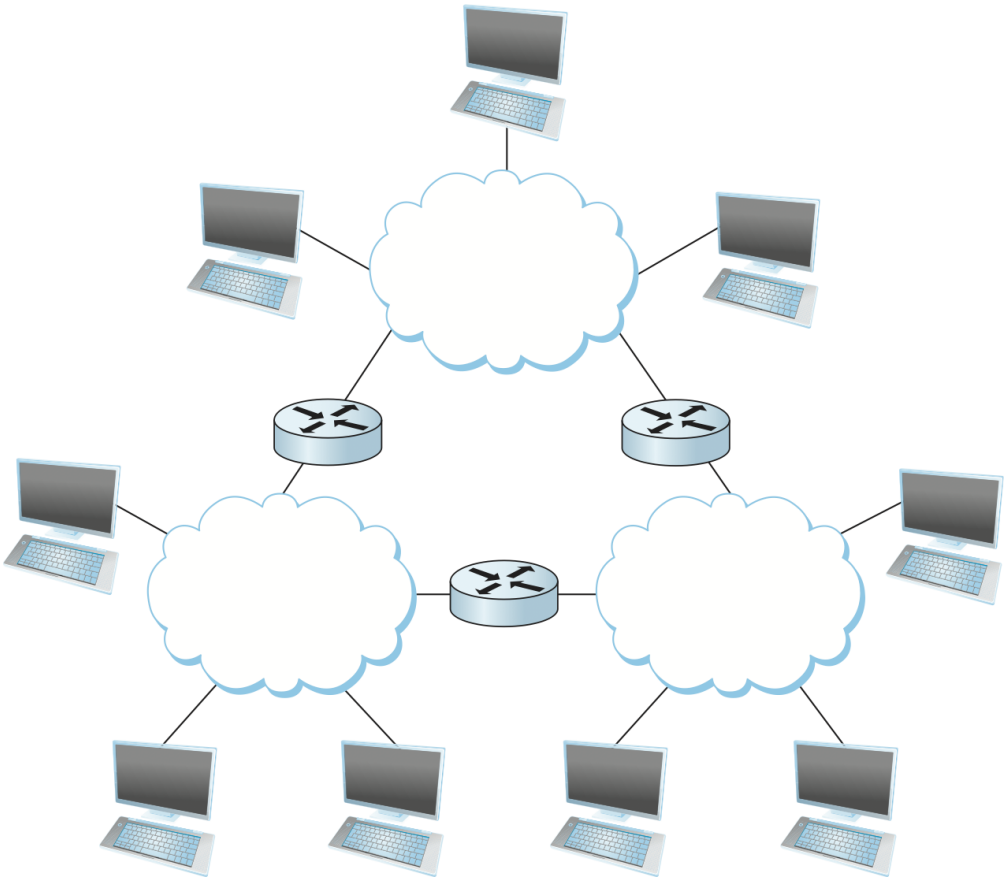


Figure 1.4.: Interconnection of networks.

Internet and that the successful growth of the Internet to global size and billions of nodes was the result of some very good design decisions by the early Internet architects, which we will discuss later.

Just because a set of hosts are directly or indirectly connected to each other does not mean that we have succeeded in providing host-to-host connectivity. The final requirement is that each node must be able to say which of the other nodes on the network it wants to communicate with. This is done by assigning an *address* to each node. An address is a byte string that identifies a node; that is, the network can use a node's address to distinguish it from the other nodes connected to the network. When a source node wants the network to deliver a message to a certain destination node, it specifies the address of the destination node. If the sending and receiving nodes are not directly connected, then the switches and routers of the network use this address to decide how to forward the message toward the destination. The process of determining systematically how to forward messages toward the destination node based on its address is called *routing*.

This brief introduction to addressing and routing has presumed that the source node wants to send a message to a single destination node (*unicast*). While this is the most common scenario, it is also possible that the source node might want to *broadcast* a message to all the nodes on the network. Or, a source node might want to send a message to some subset of the other nodes but not all of them, a situation called *multicast*. Thus, in addition to node-specific addresses, another requirement of a network is that it supports multicast and broadcast addresses.

Key Takeaway

The main idea to take away from this discussion is that we can define a *network* recursively as consisting of two or more nodes connected by a physical link, or as two or more networks connected by a node. In other words, a network can be constructed from a nesting of networks, where at the bottom level, the network is implemented by some physical medium. Among the key challenges in providing network connectivity are the definition of an address for each node that is reachable on the network (be it logical or physical), and the use of such addresses to forward messages toward the appropriate destination node(s). *[Next]*

1.2.3 Cost-Effective Resource Sharing

As stated above, this book focuses on packet-switched networks. This section explains the key requirement of computer networks—efficiency—that leads us to packet switching as the strategy of choice.

Given a collection of nodes indirectly connected by a nesting of networks, it is possible for any pair of hosts to send messages to each other across a sequence of links and nodes. Of course, we want to do more than support just one pair of communicating hosts—we want to provide all pairs of hosts with the ability to exchange messages. The question, then, is how do all the hosts that want to communicate share the network, especially if they want to use it at the same time? And, as if that problem isn't hard enough, how do several hosts share the same *link* when they all want to use it at the same time?

To understand how hosts share a network, we need to introduce a fundamental concept, *multiplexing*, which means that a system resource is shared among multiple users. At an intuitive level, multiplexing can be explained by analogy to a timesharing computer system, where a single physical processor is shared (multiplexed) among multiple jobs, each of which believes it has its own private processor. Similarly, data being sent by multiple users can be multiplexed over the physical links that make up a network.

To see how this might work, consider the simple network illustrated in Figure 1.5, where the three hosts on

the left side of the network (senders S1-S3) are sending data to the three hosts on the right (receivers R1-R3) by sharing a switched network that contains only one physical link. (For simplicity, assume that host S1 is sending data to host R1, and so on.) In this situation, three flows of data—corresponding to the three pairs of hosts—are multiplexed onto a single physical link by switch 1 and then *demultiplexed* back into separate flows by switch 2. Note that we are being intentionally vague about exactly what a "flow of data" corresponds to. For the purposes of this discussion, assume that each host on the left has a large supply of data that it wants to send to its counterpart on the right.

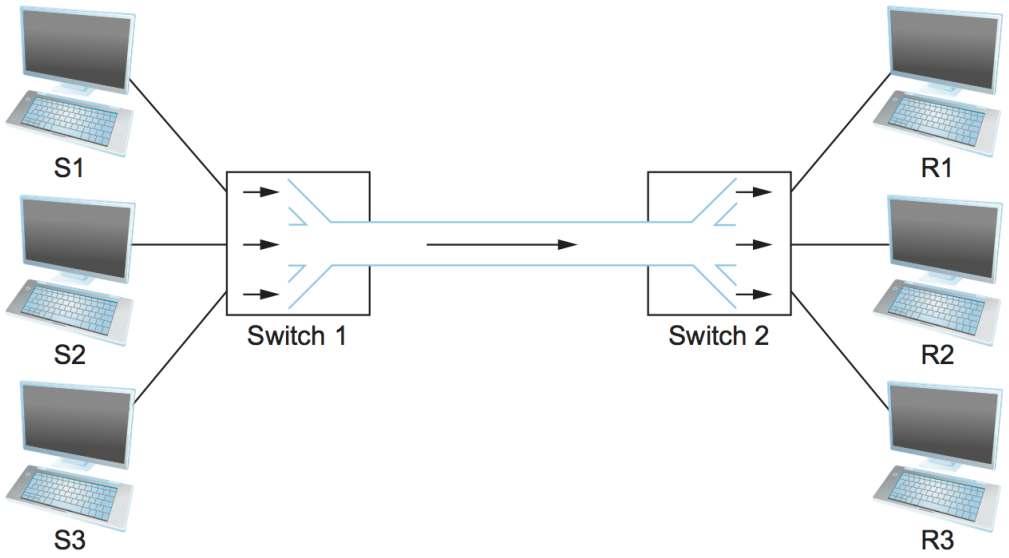


Figure 1.5.: Multiplexing multiple logical flows over a single physical link.

There are several different methods for multiplexing multiple flows onto one physical link. One common method is *synchronous time-division multiplexing* (STDM). The idea of STDM is to divide time into equal-sized quanta and, in a round-robin fashion, give each flow a chance to send its data over the physical link. In other words, during time quantum 1, data from S1 to R1 is transmitted; during time quantum 2, data from S2 to R2 is transmitted; in quantum 3, S3 sends data to R3. At this point, the first flow (S1 to R1) gets to go again, and the process repeats. Another method is *frequency-division multiplexing* (FDM). The idea of FDM is to transmit each flow over the physical link at a different frequency, much the same way that the signals for different TV stations are transmitted at a different frequency over the airwaves or on a coaxial cable TV link.

Although simple to understand, both STDM and FDM are limited in two ways. First, if one of the flows (host pairs) does not have any data to send, its share of the physical link—that is, its time quantum or its frequency—remains idle, even if one of the other flows has data to transmit. For example, S3 had to wait its turn behind S1 and S2 in the previous paragraph, even if S1 and S2 had nothing to send. For computer communication, the amount of time that a link is idle can be very large—for example, consider the amount of time you spend reading a web page (leaving the link idle) compared to the time you spend fetching the page. Second, both STDM and FDM are limited to situations in which the maximum number of flows is fixed and known ahead of time. It is not practical to resize the quantum or to add additional quanta in the case of STDM or to add new frequencies in the case of FDM.

The form of multiplexing that addresses these shortcomings, and of which we make most use in this book, is called *statistical multiplexing*. Although the name is not all that helpful for understanding the concept, statistical multiplexing is really quite simple, with two key ideas. First, it is like STDM in that the physical

link is shared over time—first data from one flow is transmitted over the physical link, then data from another flow is transmitted, and so on. Unlike STDM, however, data is transmitted from each flow on demand rather than during a predetermined time slot. Thus, if only one flow has data to send, it gets to transmit that data without waiting for its quantum to come around and thus without having to watch the quanta assigned to the other flows go by unused. It is this avoidance of idle time that gives packet switching its efficiency.

As defined so far, however, statistical multiplexing has no mechanism to ensure that all the flows eventually get their turn to transmit over the physical link. That is, once a flow begins sending data, we need some way to limit the transmission, so that the other flows can have a turn. To account for this need, statistical multiplexing defines an upper bound on the size of the block of data that each flow is permitted to transmit at a given time. This limited-size block of data is typically referred to as a *packet*, to distinguish it from the arbitrarily large *message* that an application program might want to transmit. Because a packet-switched network limits the maximum size of packets, a host may not be able to send a complete message in one packet. The source may need to fragment the message into several packets, with the receiver reassembling the packets back into the original message.

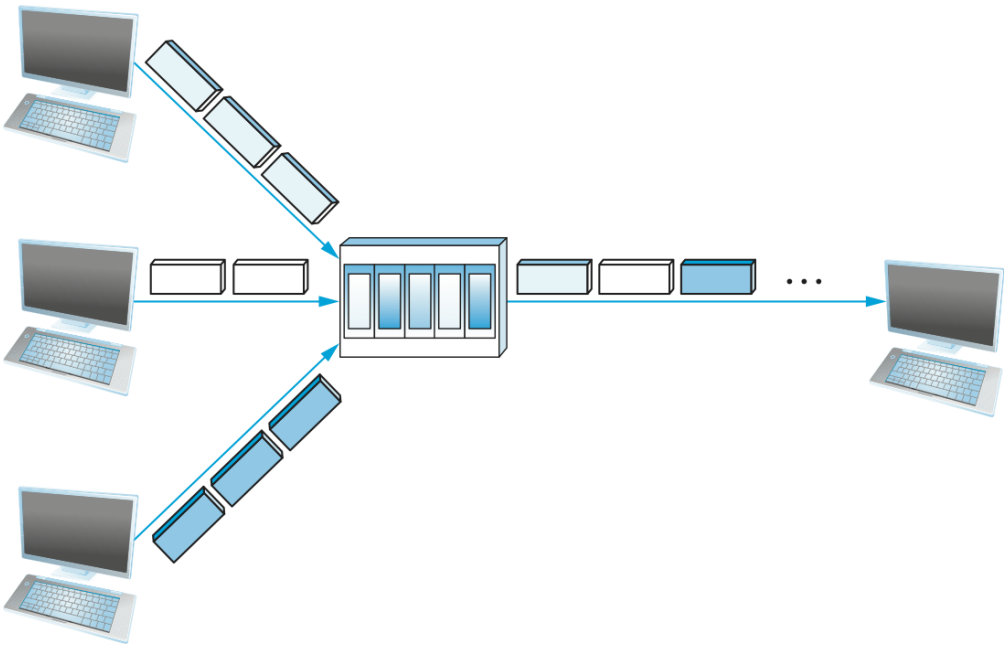


Figure 1.6.: A switch multiplexing packets from multiple sources onto one shared link.

In other words, each flow sends a sequence of packets over the physical link, with a decision made on a packet-by-packet basis as to which flow's packet to send next. Notice that, if only one flow has data to send, then it can send a sequence of packets back-to-back; however, should more than one of the flows have data to send, then their packets are interleaved on the link. Figure 1.6 depicts a switch multiplexing packets from multiple sources onto a single shared link.

The decision as to which packet to send next on a shared link can be made in a number of different ways. For example, in a network consisting of switches interconnected by links such as the one in Figure 1.5, the decision would be made by the switch that transmits packets onto the shared link. (As we will see later, not all packet-switched networks actually involve switches, and they may use other mechanisms to determine whose packet goes onto the link next.) Each switch in a packet-switched network makes this decision independently, on a packet-by-packet basis. One of the issues that faces a network designer is how to make

this decision in a fair manner. For example, a switch could be designed to service packets on a first-in, first-out (FIFO) basis. Another approach would be to transmit the packets from each of the different flows that are currently sending data through the switch in a round-robin manner. This might be done to ensure that certain flows receive a particular share of the link's bandwidth or that they never have their packets delayed in the switch for more than a certain length of time. A network that attempts to allocate bandwidth to particular flows is sometimes said to support *quality of service* (QoS).

Also, notice in Figure 1.6 that since the switch has to multiplex three incoming packet streams onto one outgoing link, it is possible that the switch will receive packets faster than the shared link can accommodate. In this case, the switch is forced to buffer these packets in its memory. Should a switch receive packets faster than it can send them for an extended period of time, then the switch will eventually run out of buffer space, and some packets will have to be dropped. When a switch is operating in this state, it is said to be *congested*.

Key Takeaway

The bottom line is that statistical multiplexing defines a cost-effective way for multiple users (e.g., host-to-host flows of data) to share network resources (links and nodes) in a fine-grained manner. It defines the packet as the granularity with which the links of the network are allocated to different flows, with each switch able to schedule the use of the physical links it is connected to on a per-packet basis. Fairly allocating link capacity to different flows and dealing with congestion when it occurs are the key challenges of statistical multiplexing. *[Next]*

1.2.4 Support for Common Services

The previous discussion focused on the challenges involved in providing cost-effective connectivity among a group of hosts, but it is overly simplistic to view a computer network as simply delivering packets among a collection of computers. It is more accurate to think of a network as providing the means for a set of application processes that are distributed over those computers to communicate. In other words, the next requirement of a computer network is that the application programs running on the hosts connected to the network must be able to communicate in a meaningful way. From the application developer's perspective, the network needs to make his or her life easier.

When two application programs need to communicate with each other, a lot of complicated things must happen beyond simply sending a message from one host to another. One option would be for application designers to build all that complicated functionality into each application program. However, since many applications need common services, it is much more logical to implement those common services once and then to let the application designer build the application using those services. The challenge for a network designer is to identify the right set of common services. The goal is to hide the complexity of the network from the application without overly constraining the application designer.

Intuitively, we view the network as providing logical *channels* over which application-level processes can communicate with each other; each channel provides the set of services required by that application. In other words, just as we use a cloud to abstractly represent connectivity among a set of computers, we now think of a channel as connecting one process to another. Figure 1.7 shows a pair of application-level processes communicating over a logical channel that is, in turn, implemented on top of a cloud that connects a set of hosts. We can think of the channel as being like a pipe connecting two applications, so that a sending application can put data in one end and expect that data to be delivered by the network to the application at

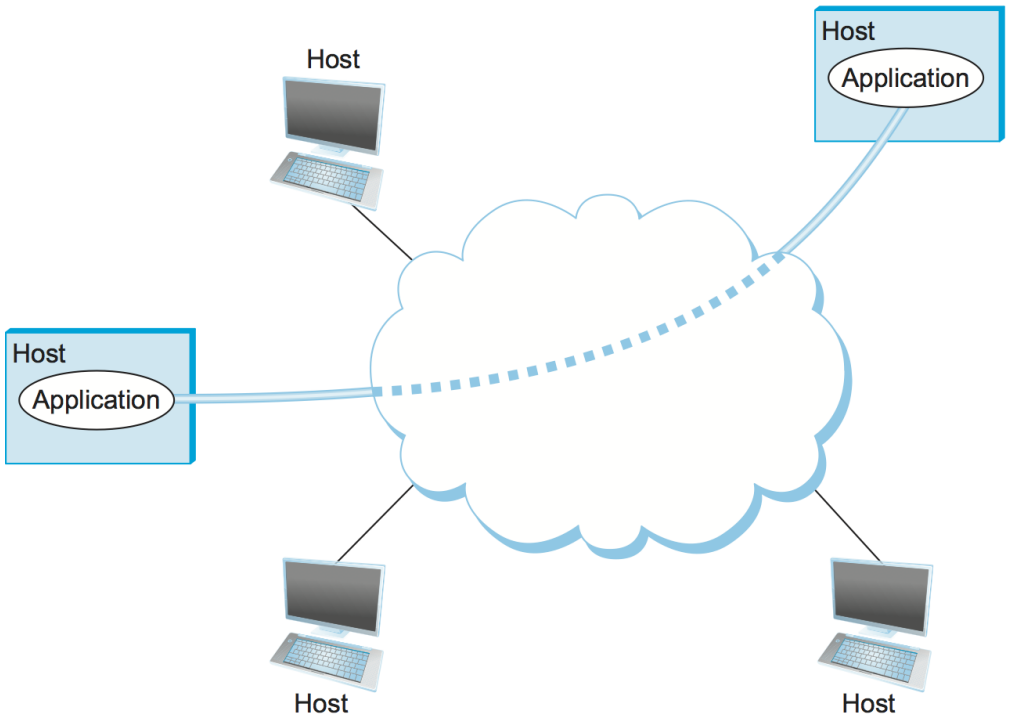


Figure 1.7.: Processes communicating over an abstract channel.

the other end of the pipe.

Like any abstraction, logical process-to-process channels are implemented on top of a collection of physical host-to-host channels. This is the essense of layering, the cornerstone of network architectures discussed in the next section.

The challenge is to recognize what functionality the channels should provide to application programs. For example, does the application require a guarantee that messages sent over the channel are delivered, or is it acceptable if some messages fail to arrive? Is it necessary that messages arrive at the recipient process in the same order in which they are sent, or does the recipient not care about the order in which messages arrive? Does the network need to ensure that no third parties are able to eavesdrop on the channel, or is privacy not a concern? In general, a network provides a variety of different types of channels, with each application selecting the type that best meets its needs. The rest of this section illustrates the thinking involved in defining useful channels.

Identify Common Communication Patterns

Designing abstract channels involves first understanding the communication needs of a representative collection of applications, then extracting their common communication requirements, and finally incorporating the functionality that meets these requirements in the network.

One of the earliest applications supported on any network is a file access program like the File Transfer Protocol (FTP) or Network File System (NFS). Although many details vary—for example, whether whole files are transferred across the network or only single blocks of the file are read/written at a given time—the communication component of remote file access is characterized by a pair of processes, one that requests

that a file be read or written and a second process that honors this request. The process that requests access to the file is called the *client*, and the process that supports access to the file is called the *server*.

Reading a file involves the client sending a small request message to a server and the server responding with a large message that contains the data in the file. Writing works in the opposite way—the client sends a large message containing the data to be written to the server, and the server responds with a small message confirming that the write to disk has taken place.

A digital library is a more sophisticated application than file transfer, but it requires similar communication services. For example, the *Association for Computing Machinery* (ACM) operates a large digital library of computer science literature at

```
http://portal.acm.org/dl.cfm
```

This library has a wide range of searching and browsing features to help users find the articles they want, but ultimately much of what it does is respond to user requests for files, such as electronic copies of journal articles.

Using file access, a digital library, and the two video applications described in the introduction (videoconferencing and video on demand) as a representative sample, we might decide to provide the following two types of channels: *request/reply* channels and *message stream* channels. The request/reply channel would be used by the file transfer and digital library applications. It would guarantee that every message sent by one side is received by the other side and that only one copy of each message is delivered. The request/reply channel might also protect the privacy and integrity of the data that flows over it, so that unauthorized parties cannot read or modify the data being exchanged between the client and server processes.

The message stream channel could be used by both the video on demand and videoconferencing applications, provided it is parameterized to support both one-way and two-way traffic and to support different delay properties. The message stream channel might not need to guarantee that all messages are delivered, since a video application can operate adequately even if some video frames are not received. It would, however, need to ensure that those messages that are delivered arrive in the same order in which they were sent, to avoid displaying frames out of sequence. Like the request/reply channel, the message stream channel might want to ensure the privacy and integrity of the video data. Finally, the message stream channel might need to support multicast, so that multiple parties can participate in the teleconference or view the video.

While it is common for a network designer to strive for the smallest number of abstract channel types that can serve the largest number of applications, there is a danger in trying to get away with too few channel abstractions. Simply stated, if you have a hammer, then everything looks like a nail. For example, if all you have are message stream and request/reply channels, then it is tempting to use them for the next application that comes along, even if neither type provides exactly the semantics needed by the application. Thus, network designers will probably be inventing new types of channels—and adding options to existing channels—for as long as application programmers are inventing new applications.

Also note that independent of exactly *what* functionality a given channel provides, there is the question of *where* that functionality is implemented. In many cases, it is easiest to view the host-to-host connectivity of the underlying network as simply providing a *bit pipe*, with any high-level communication semantics provided at the end hosts. The advantage of this approach is that it keeps the switches in the middle of the network as simple as possible—they simply forward packets—but it requires the end hosts to take on much of the burden of supporting semantically rich process-to-process channels. The alternative is to push additional functionality onto the switches, thereby allowing the end hosts to be "dumb" devices (e.g., telephone handsets). We will see this question of how various network services are partitioned between the packet

switches and the end hosts (devices) as a recurring issue in network design.

Reliable Message Delivery

As suggested by the examples just considered, reliable message delivery is one of the most important functions that a network can provide. It is difficult to determine how to provide this reliability, however, without first understanding how networks can fail. The first thing to recognize is that computer networks do not exist in a perfect world. Machines crash and later are rebooted, fibers are cut, electrical interference corrupts bits in the data being transmitted, switches run out of buffer space, and, as if these sorts of physical problems aren't enough to worry about, the software that manages the hardware may contain bugs and sometimes forwards packets into oblivion. Thus, a major requirement of a network is to recover from certain kinds of failures, so that application programs don't have to deal with them or even be aware of them.

There are three general classes of failure that network designers have to worry about. First, as a packet is transmitted over a physical link, *bit errors* may be introduced into the data; that is, a 1 is turned into a 0 or *vice versa*. Sometimes single bits are corrupted, but more often than not a *burst error* occurs—several consecutive bits are corrupted. Bit errors typically occur because outside forces, such as lightning strikes, power surges, and microwave ovens, interfere with the transmission of data. The good news is that such bit errors are fairly rare, affecting on average only one out of every 10^6 to 10^7 bits on a typical copper-based cable and one out of every 10^{12} to 10^{14} bits on a typical optical fiber. As we will see, there are techniques that detect these bit errors with high probability. Once detected, it is sometimes possible to correct for such errors—if we know which bit or bits are corrupted, we can simply flip them—while in other cases the damage is so bad that it is necessary to discard the entire packet. In such a case, the sender may be expected to retransmit the packet.

The second class of failure is at the packet, rather than the bit, level; that is, a complete packet is lost by the network. One reason this can happen is that the packet contains an uncorrectable bit error and therefore has to be discarded. A more likely reason, however, is that one of the nodes that has to handle the packet—for example, a switch that is forwarding it from one link to another—is so overloaded that it has no place to store the packet and therefore is forced to drop it. This is the problem of congestion just discussed. Less commonly, the software running on one of the nodes that handles the packet makes a mistake. For example, it might incorrectly forward a packet out on the wrong link, so that the packet never finds its way to the ultimate destination. As we will see, one of the main difficulties in dealing with lost packets is distinguishing between a packet that is indeed lost and one that is merely late in arriving at the destination.

The third class of failure is at the node and link level; that is, a physical link is cut, or the computer it is connected to crashes. This can be caused by software that crashes, a power failure, or a reckless backhoe operator. Failures due to misconfiguration of a network device are also common. While any of these failures can eventually be corrected, they can have a dramatic effect on the network for an extended period of time. However, they need not totally disable the network. In a packet-switched network, for example, it is sometimes possible to route around a failed node or link. One of the difficulties in dealing with this third class of failure is distinguishing between a failed computer and one that is merely slow or, in the case of a link, between one that has been cut and one that is very flaky and therefore introducing a high number of bit errors.

Key Takeaway

The key idea to take away from this discussion is that defining useful channels involves both understanding

the applications' requirements and recognizing the limitations of the underlying technology. The challenge is to fill in the gap between what the application expects and what the underlying technology can provide. This is sometimes called the *semantic gap.* *[Next]*

1.2.5 Manageability

A final requirement, which seems to be neglected or left till last all too often (as we do here), is that networks need to be managed. Managing a network includes upgrading equipment as the network grows to carry more traffic or reach more users, troubleshooting the network when things go wrong or performance isn't as desired, and adding new features in support of new applications. Network management has historically been a human-intensive aspect of networking, and while it is ulikely we'll get people entirely out of the loop, it is increasingly being addressed by automation and self-healing designs.

This requirement is partly related to the issue of scalability discussed above—as the Internet has scaled up to support billions of users and at least hundreds of millions of hosts, the challenges of keeping the whole thing running correctly and correctly configuring new devices as they are added have become increasingly problematic. Configuring a single router in a network is often a task for a trained expert; configuring thousands of routers and figuring out why a network of such a size is not behaving as expected can become a task beyond any single human. This is why automation is becoming so important.

One way to make a network easier to manage is to avoid change. Once the network is working, simply *do not touch it!* This mindset exposes the fundamental tension between *stability* and *feature velocity*: the rate at which new capabilities are introduced into the network. Favoring stability is the approach the telecommunications industry (not to mention University system administrators and corporate IT departments) adopted for many years, making it one of the most slow moving and risk averse industries you will find anywhere. But the recent explosion of the cloud has changed that dynamic, making it necessary to bring stability and feature velocity more into balance. The impact of the cloud on the network is a topic that comes up over and over throughout the book, and one we pay particular attention to in the *Perspectives* section at the end of each chapter. For now, suffice it to say that managing a rapidly evolving network is arguably *the* central challenge in networking today.

1.3 Architecture

The previous section established a pretty substantial set of requirements for network design—a computer network must provide general, cost-effective, fair, and robust connectivity among a large number of computers. As if this weren't enough, networks do not remain fixed at any single point in time but must evolve to accommodate changes in both the underlying technologies upon which they are based as well as changes in the demands placed on them by application programs. Furthermore, networks must be manageable by humans of varying levels of skill. Designing a network to meet these requirements is no small task.

To help deal with this complexity, network designers have developed general blueprints—usually called *network architectures*—that guide the design and implementation of networks. This section defines more carefully what we mean by a network architecture by introducing the central ideas that are common to all network architectures. It also introduces two of the most widely referenced architectures—the OSI (or 7-layer) architecture and the Internet architecture.

1.3.1 Layering and Protocols

Abstraction—the hiding of implementation details behind a well-defined interface—is the fundamental tool used by system designers to manage complexity. The idea of an abstraction is to define a model that can capture some important aspect of the system, encapsulate this model in an object that provides an interface that can be manipulated by other components of the system, and hide the details of how the object is implemented from the users of the object. The challenge is to identify abstractions that simultaneously provide a service that proves useful in a large number of situations and that can be efficiently implemented in the underlying system. This is exactly what we were doing when we introduced the idea of a channel in the previous section: we were providing an abstraction for applications that hides the complexity of the network from application writers.

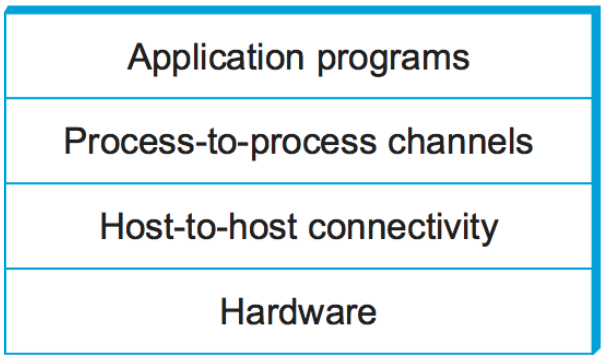


Figure 1.8.: Example of a layered network system.

Abstractions naturally lead to layering, especially in network systems. The general idea is that you start with the services offered by the underlying hardware and then add a sequence of layers, each providing a higher (more abstract) level of service. The services provided at the high layers are implemented in terms of the services provided by the low layers. Drawing on the discussion of requirements given in the previous section, for example, we might imagine a simple network as having two layers of abstraction sandwiched between the application program and the underlying hardware, as illustrated in Figure 1.8. The layer immediately above the hardware in this case might provide host-to-host connectivity, abstracting away the fact that there may be an arbitrarily complex network topology between any two hosts. The next layer up builds on the available host-to-host communication service and provides support for process-to-process channels, abstracting away the fact that the network occasionally loses messages, for example.

Layering provides two useful features. First, it decomposes the problem of building a network into more manageable components. Rather than implementing a monolithic piece of software that does everything you will ever want, you can implement several layers, each of which solves one part of the problem. Second, it provides a more modular design. If you decide that you want to add some new service, you may only need to modify the functionality at one layer, reusing the functions provided at all the other layers.

Thinking of a system as a linear sequence of layers is an oversimplification, however. Many times there are multiple abstractions provided at any given level of the system, each providing a different service to the higher layers but building on the same low-level abstractions. To see this, consider the two types of channels discussed in the previous section. One provides a request/reply service and one supports a message stream service. These two channels might be alternative offerings at some level of a multilevel networking system, as illustrated in Figure 1.9.

Using this discussion of layering as a foundation, we are now ready to discuss the architecture of a net-

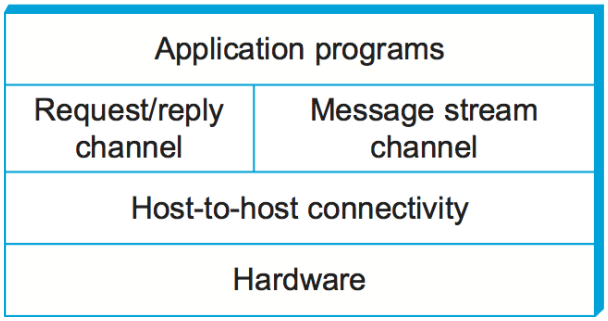


Figure 1.9.: Layered system with alternative abstractions available at a given layer.

work more precisely. For starters, the abstract objects that make up the layers of a network system are called *protocols*. That is, a protocol provides a communication service that higher-level objects (such as application processes, or perhaps higher-level protocols) use to exchange messages. For example, we could imagine a network that supports a request/reply protocol and a message stream protocol, corresponding to the request/reply and message stream channels discussed above.

Each protocol defines two different interfaces. First, it defines a *service interface* to the other objects on the same computer that want to use its communication services. This service interface defines the operations that local objects can perform on the protocol. For example, a request/reply protocol would support operations by which an application can send and receive messages. An implementation of the HTTP protocol could support an operation to fetch a page of hypertext from a remote server. An application such as a web browser would invoke such an operation whenever the browser needs to obtain a new page (e.g., when the user clicks on a link in the currently displayed page).

Second, a protocol defines a *peer interface* to its counterpart (peer) on another machine. This second interface defines the form and meaning of messages exchanged between protocol peers to implement the communication service. This would determine the way in which a request/reply protocol on one machine communicates with its peer on another machine. In the case of HTTP, for example, the protocol specification defines in detail how a *GET* command is formatted, what arguments can be used with the command, and how a web server should respond when it receives such a command.

To summarize, a protocol defines a communication service that it exports locally (the service interface), along with a set of rules governing the messages that the protocol exchanges with its peer(s) to implement this service (the peer interface). This situation is illustrated in Figure 1.10.

Except at the hardware level, where peers directly communicate with each other over a physical medium, peer-to-peer communication is indirect—each protocol communicates with its peer by passing messages to some lower-level protocol, which in turn delivers the message to *its* peer. In addition, there are potentially more than one protocol at any given level, each providing a different communication service. We therefore represent the suite of protocols that make up a network system with a *protocol graph*. The nodes of the graph correspond to protocols, and the edges represent a *depends on* relation. For example, Figure 1.11 illustrates a protocol graph for the hypothetical layered system we have been discussing—protocols RRP (Request/Reply Protocol) and MSP (Message Stream Protocol) implement two different types of process-to-process channels, and both depend on the Host-to-Host Protocol (HHP) which provides a host-to-host connectivity service.

In this example, suppose that the file access program on host 1 wants to send a message to its peer on host 2 using the communication service offered by RRP. In this case, the file application asks RRP to send

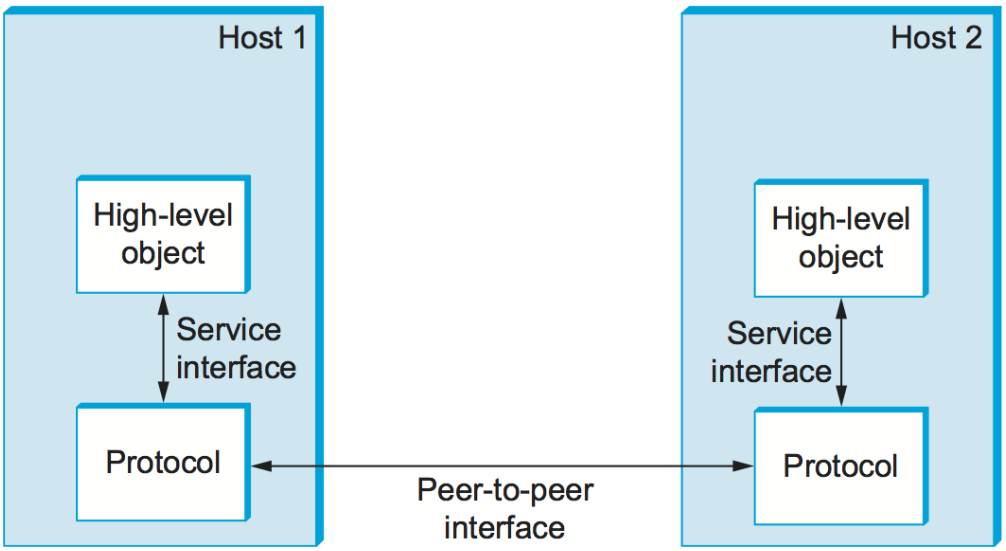


Figure 1.10.: Service interfaces and peer interfaces.

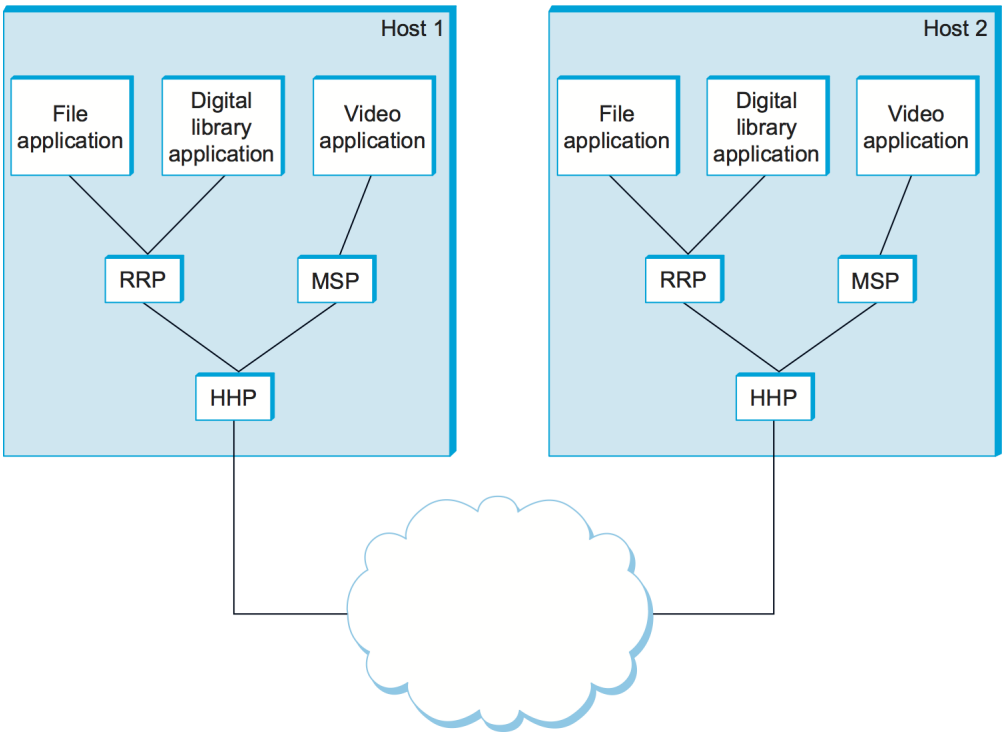


Figure 1.11.: Example of a protocol graph.

the message on its behalf. To communicate with its peer, RRP invokes the services of HHP, which in turn transmits the message to its peer on the other machine. Once the message has arrived at the instance of HHP on host 2, HHP passes the message up to RRP, which in turn delivers the message to the file application. In this particular case, the application is said to employ the services of the *protocol stack* RRP/HHP.

Note that the term *protocol* is used in two different ways. Sometimes it refers to the abstract interfaces—that is, the operations defined by the service interface and the form and meaning of messages exchanged between peers, and sometimes it refers to the module that actually implements these two interfaces. To distinguish between the interfaces and the module that implements these interfaces, we generally refer to the former as a *protocol specification*. Specifications are generally expressed using a combination of prose, pseudocode, state transition diagrams, pictures of packet formats, and other abstract notations. It should be the case that a given protocol can be implemented in different ways by different programmers, as long as each adheres to the specification. The challenge is ensuring that two different implementations of the same specification can successfully exchange messages. Two or more protocol modules that do accurately implement a protocol specification are said to *interoperate* with each other.

We can imagine many different protocols and protocol graphs that satisfy the communication requirements of a collection of applications. Fortunately, there exist standardization bodies, such as the Internet Engineering Task Force (IETF) and the International Standards Organization (ISO), that establish policies for a particular protocol graph. We call the set of rules governing the form and content of a protocol graph a *network architecture*. Although beyond the scope of this book, standardization bodies have established well-defined procedures for introducing, validating, and finally approving protocols in their respective architectures. We briefly describe the architectures defined by the IETF and ISO shortly, but first there are two additional things we need to explain about the mechanics of protocol layering.

1.3.2 Encapsulation

Consider what happens in when one of the application programs sends a message to its peer by passing the message to RRP. From RRP's perspective, the message it is given by the application is an uninterpreted string of bytes. RRP does not care that these bytes represent an array of integers, an email message, a digital image, or whatever; it is simply charged with sending them to its peer. However, RRP must communicate control information to its peer, instructing it how to handle the message when it is received. RRP does this by attaching a *header* to the message. Generally speaking, a header is a small data structure—from a few bytes to a few dozen bytes—that is used among peers to communicate with each other. As the name suggests, headers are usually attached to the front of a message. In some cases, however, this peer-to-peer control information is sent at the end of the message, in which case it is called a *trailer*. The exact format for the header attached by RRP is defined by its protocol specification. The rest of the message—that is, the data being transmitted on behalf of the application—is called the message's *body* or *payload*. We say that the application's data is *encapsulated* in the new message created by RRP.

This process of encapsulation is then repeated at each level of the protocol graph; for example, HHP encapsulates RRP's message by attaching a header of its own. If we now assume that HHP sends the message to its peer over some network, then when the message arrives at the destination host, it is processed in the opposite order: HHP first interprets the HHP header at the front of the message (i.e., takes whatever action is appropriate given the contents of the header) and passes the body of the message (but not the HHP header) up to RRP, which takes whatever action is indicated by the RRP header that its peer attached and passes the body of the message (but not the RRP header) up to the application program. The message passed up from RRP to the application on host 2 is exactly the same message as the application passed down to RRP on

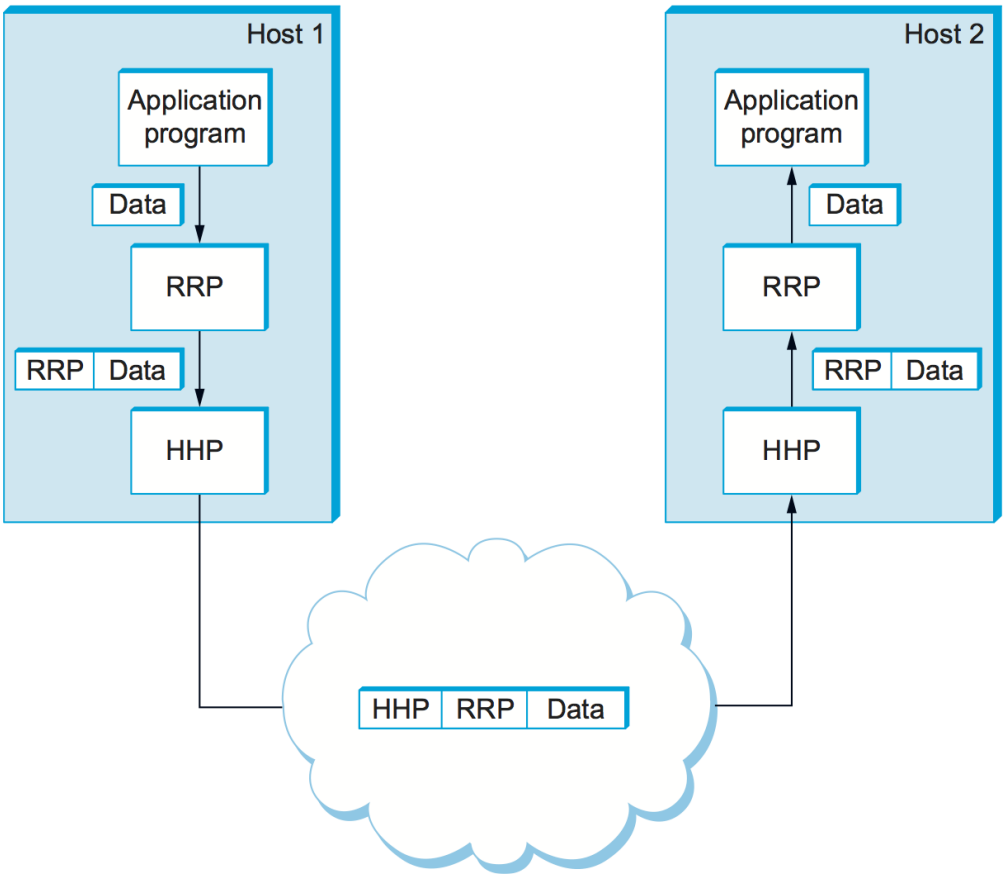


Figure 1.12.: High-level messages are encapsulated inside of low-level messages.

host 1; the application does not see any of the headers that have been attached to it to implement the lower-level communication services. This whole process is illustrated in Figure 1.12. Note that in this example, nodes in the network (e.g., switches and routers) may inspect the HHP header at the front of the message.

Note that when we say a low-level protocol does not interpret the message it is given by some high-level protocol, we mean that it does not know how to extract any meaning from the data contained in the message. It is sometimes the case, however, that the low-level protocol applies some simple transformation to the data it is given, such as to compress or encrypt it. In this case, the protocol is transforming the entire body of the message, including both the original application's data and all the headers attached to that data by higher-level protocols.

1.3.3 Multiplexing and Demultiplexing

Recall that a fundamental idea of packet switching is to multiplex multiple flows of data over a single physical link. This same idea applies up and down the protocol graph, not just to switching nodes. In Figure 1.11, for example, we can think of RRP as implementing a logical communication channel, with messages from two different applications multiplexed over this channel at the source host and then demultiplexed back to the appropriate application at the destination host.

Practically speaking, this simply means that the header that RRP attaches to its messages contains an identifier that records the application to which the message belongs. We call this identifier RRP's *demultiplexing key*, or *demux key* for short. At the source host, RRP includes the appropriate demux key in its header. When the message is delivered to RRP on the destination host, it strips its header, examines the demux key, and demultiplexes the message to the correct application.

RRP is not unique in its support for multiplexing; nearly every protocol implements this mechanism. For example, HHP has its own demux key to determine which messages to pass up to RRP and which to pass up to MSP. However, there is no uniform agreement among protocols—even those within a single network architecture—on exactly what constitutes a demux key. Some protocols use an 8-bit field (meaning they can support only 256 high-level protocols), and others use 16- or 32-bit fields. Also, some protocols have a single demultiplexing field in their header, while others have a pair of demultiplexing fields. In the former case, the same demux key is used on both sides of the communication, while in the latter case each side uses a different key to identify the high-level protocol (or application program) to which the message is to be delivered.

1.3.4 7-Layer OSI Model

The ISO was one of the first organizations to formally define a common way to connect computers. Their architecture, called the *Open Systems Interconnection* (OSI) architecture and illustrated in Figure 1.13, defines a partitioning of network functionality into seven layers, where one or more protocols implement the functionality assigned to a given layer. In this sense, the schematic given in is not a protocol graph, *per se*, but rather a *reference model* for a protocol graph. It is often referred to as the 7-layer model. While there is no OSI-based network running today, the terminology it defined is still widely used, so it is still worth a cursory look.

Starting at the bottom and working up, the *physical* layer handles the transmission of raw bits over a communications link. The *data link* layer then collects a stream of bits into a larger aggregate called a *frame*. Network adaptors, along with device drivers running in the node's operating system, typically implement

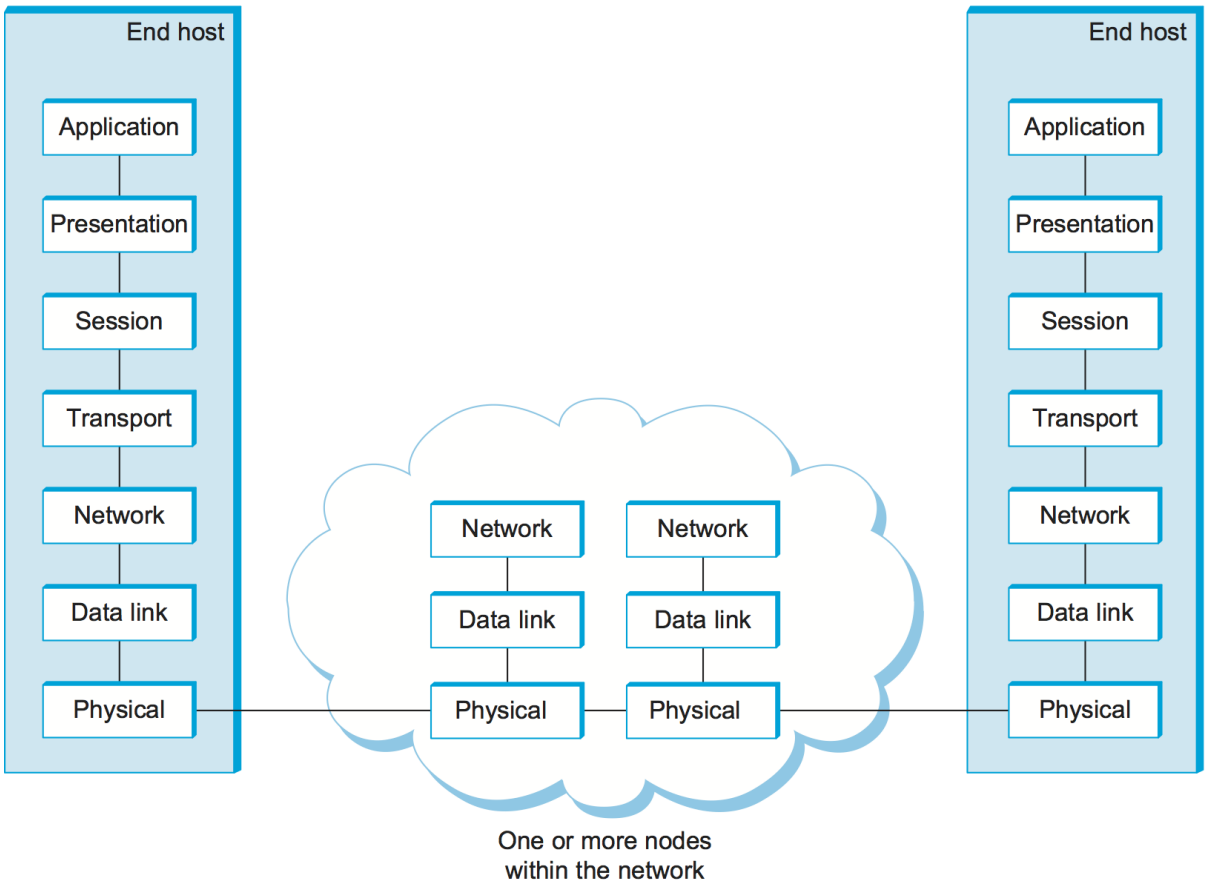


Figure 1.13.: The OSI 7-layer model.

the data link level. This means that frames, not raw bits, are actually delivered to hosts. The *network* layer handles routing among nodes within a packet-switched network. At this layer, the unit of data exchanged among nodes is typically called a *packet* rather than a frame, although they are fundamentally the same thing. The lower three layers are implemented on all network nodes, including switches within the network and hosts connected to the exterior of the network. The *transport* layer then implements what we have up to this point been calling a *process-to-process channel*. Here, the unit of data exchanged is commonly called a *message* rather than a packet or a frame. The transport layer and higher layers typically run only on the end hosts and not on the intermediate switches or routers.

Skipping ahead to the top (seventh) layer and working our way back down, we find the *application* layer. Application layer protocols include things like the Hypertext Transfer Protocol (HTTP), which is the basis of the World Wide Web and is what enables web browsers to request pages from web servers. Below that, the *presentation* layer is concerned with the format of data exchanged between peers—for example, whether an integer is 16, 32, or 64 bits long, whether the most significant byte is transmitted first or last, or how a video stream is formatted. Finally, the *session* layer provides a name space that is used to tie together the potentially different transport streams that are part of a single application. For example, it might manage an audio stream and a video stream that are being combined in a teleconferencing application.

1.3.5 Internet Architecture

The Internet architecture, which is also sometimes called the TCP/IP architecture after its two main protocols, is depicted in Figure 1.14. An alternative representation is given in Figure 1.15. The Internet architecture evolved out of experiences with an earlier packet-switched network called the ARPANET. Both the Internet and the ARPANET were funded by the Advanced Research Projects Agency (ARPA), one of the research and development funding agencies of the U.S. Department of Defense. The Internet and ARPANET were around before the OSI architecture, and the experience gained from building them was a major influence on the OSI reference model.

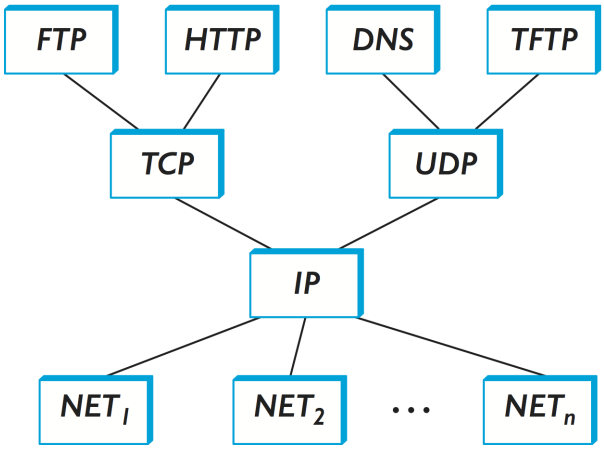


Figure 1.14.: Internet protocol graph.

While the 7-layer OSI model can, with some imagination, be applied to the Internet, a simpler stack is often used instead. At the lowest level is a wide variety of network protocols, denoted NET_1, NET_2, and so on. In practice, these protocols are implemented by a combination of hardware (e.g., a network adaptor) and software (e.g., a network device driver). For example, you might find Ethernet or wireless protocols (such as the 802.11 Wi-Fi standards) at this layer. (These protocols in turn may actually involve several

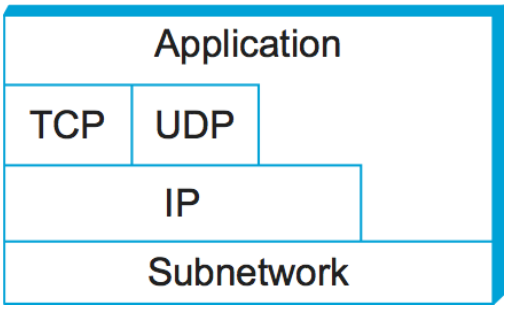


Figure 1.15.: Alternative view of the Internet architecture. The "subnetwork" layer was historically referred to as the "network" layer and is now often referred to as "Layer 2" (influenced by the OSI model).

sublayers, but the Internet architecture does not presume anything about them.) The next layer consists of a single protocol—the *Internet Protocol* (IP). This is the protocol that supports the interconnection of multiple networking technologies into a single, logical internetwork. The layer on top of IP contains two main protocols—the *Transmission Control Protocol* (TCP) and the *User Datagram Protocol* (UDP). TCP and UDP provide alternative logical channels to application programs: TCP provides a reliable byte-stream channel, and UDP provides an unreliable datagram delivery channel (*datagram* may be thought of as a synonym for message). In the language of the Internet, TCP and UDP are sometimes called *end-to-end* protocols, although it is equally correct to refer to them as *transport* protocols.

Running above the transport layer is a range of application protocols, such as HTTP, FTP, Telnet (remote login), and the Simple Mail Transfer Protocol (SMTP), that enable the interoperation of popular applications. To understand the difference between an application layer protocol and an application, think of all the different World Wide Web browsers that are or have been available (e.g., Firefox, Chrome, Safari, Netscape, Mosaic, Internet Explorer). There is a similarly large number of different implementations of web servers. The reason that you can use any one of these application programs to access a particular site on the Web is that they all conform to the same application layer protocol: HTTP. Confusingly, the same term sometimes applies to both an application and the application layer protocol that it uses (e.g., FTP is often used as the name of an application that implements the FTP protocol).

Most people who work actively in the networking field are familiar with both the Internet architecture and the 7-layer OSI architecture, and there is general agreement on how the layers map between architectures. The Internet's application layer is considered to be at layer 7, its transport layer is layer 4, the IP (internetworking or just network) layer is layer 3, and the link or subnet layer below IP is layer 2.

IETF and Standardization

Although we call it the "Internet architecture" rather than the "IETF architecture," it's fair to say that the IETF is the primary standardization body responsible for its definition, as well as the specification of many of its protocols, such as TCP, UDP, IP, DNS, and BGP. But the Internet architecture also embraces many protocols defined by other organizations, including IEEE's 802.11 ethernet and Wi-Fi standards, W3C's HTTP/HTML web specifications, 3GPP's 4G and 5G cellular networks standards, and ITU-T's H.232 video encoding standards, to name a few.

In addition to defining architectures and specifying protocols, there are yet other organizations that support the larger goal of interoperability. One example is the IANA (Internet Assigned Numbers Authority),

which as its name impies, is responsible for handing out the unique identifiers needed to make the protocols work. IANA, in turn, is a department within the ICANN (Internt Corporation for Assigned Names and Numbers), a non-profit organization that's responsible for the overall stewardship of the Internet.

The Internet architecture has three features that are worth highlighting. First, as best illustrated by Figure 1.15, the Internet architecture does not imply strict layering. The application is free to bypass the defined transport layers and to directly use IP or one of the underlying networks. In fact, programmers are free to define new channel abstractions or applications that run on top of any of the existing protocols.

Second, if you look closely at the protocol graph in Figure 1.14, you will notice an hourglass shape—wide at the top, narrow in the middle, and wide at the bottom. This shape actually reflects the central philosophy of the architecture. That is, IP serves as the focal point for the architecture—it defines a common method for exchanging packets among a wide collection of networks. Above IP there can be arbitrarily many transport protocols, each offering a different channel abstraction to application programs. Thus, the issue of delivering messages from host to host is completely separated from the issue of providing a useful process-to-process communication service. Below IP, the architecture allows for arbitrarily many different network technologies, ranging from Ethernet to wireless to single point-to-point links.

A final attribute of the Internet architecture (or more accurately, of the IETF culture) is that in order for a new protocol to be officially included in the architecture, there must be both a protocol specification and at least one (and preferably two) representative implementations of the specification. The existence of working implementations is required for standards to be adopted by the IETF. This cultural assumption of the design community helps to ensure that the architecture's protocols can be efficiently implemented. Perhaps the value the Internet culture places on working software is best exemplified by a quote on T-shirts commonly worn at IETF meetings:

> *We reject kings, presidents, and voting. We believe in rough consensus and running code.*
> **(David Clark)**

Key Takeaway

Of these three attributes of the Internet architecture, the hourglass design philosophy is important enough to bear repeating. The hourglass's narrow waist represents a minimal and carefully chosen set of global capabilities that allows both higher-level applications and lower-level communication technologies to coexist, share capabilities, and evolve rapidly. The narrow-waisted model is critical to the Internet's ability to adapt to new user demands and changing technologies. *[Next]*

1.4 Software

Network architectures and protocol specifications are essential things, but a good blueprint is not enough to explain the phenomenal success of the Internet: The number of computers connected to the Internet has grown exponentially for over three decades (although precise numbers are hard to come by). The number of users of the Internet was estimated to be around 4.1 billion by the end of 2018—roughly half of the world's population.

What explains the success of the Internet? There are certainly many contributing factors (including a good architecture), but one thing that has made the Internet such a runaway success is the fact that so much of

its functionality is provided by software running on general-purpose computers. The significance of this is that new functionality can be added readily with "just a small matter of programming." As a result, new applications and services have been showing up at an incredible pace.

A related factor is the massive increase in computing power available in commodity machines. Although computer networks have always been capable in principle of transporting any kind of information, such as digital voice samples, digitized images, and so on, this potential was not particularly interesting if the computers sending and receiving that data were too slow to do anything useful with the information. Virtually all of today's computers are capable of playing back digitized audio and video at a speed and resolution that are quite usable.

In the years since the first edition of this book appeared, the writing of networked applications has become a mainstream activity and not a job just for a few specialists. Many factors have played into this, including better tools to make the job easier and the opening up of new markets such as applications for smartphones.

The point to note is that knowing how to implement network software is an essential part of understanding computer networks, and while the odds are you will not be tasked to implement a low-level protocol like IP, there is a good chance you will find reason to implement an application-level protocol—the elusive "killer app" that will lead to unimaginable fame and fortune. To get you started, this section introduces some of the issues involved in implementing a network application on top of the Internet. Typically, such programs are simultaneously an application (i.e., designed to interact with users) and a protocol (i.e., communicates with peers across the network).

1.4.1 Application Programming Interface (Sockets)

The place to start when implementing a network application is the interface exported by the network. Since most network protocols are in software (especially those high in the protocol stack), and nearly all computer systems implement their network protocols as part of the operating system, when we refer to the interface "exported by the network," we are generally referring to the interface that the OS provides to its networking subsystem. This interface is often called the network *application programming interface* (API).

Although each operating system is free to define its own network API (and most have), over time certain of these APIs have become widely supported; that is, they have been ported to operating systems other than their native system. This is what has happened with the *socket interface* originally provided by the Berkeley distribution of Unix, which is now supported in virtually all popular operating systems, and is the foundation of language-specific interfaces, such as the Java or Python socket library. We use Linux and C for all code examples in this book, Linux because it is open source and C because it remains the language of choice for network internals. (C also has the advantage of exposing all the low-level details, which is helpful in understanding the underlying ideas.)

Sockets Enabled Application Explosion

It is hard to overstate the importance of the Socket API. It defines the demarcation point between the applications running on top of the Internet, and the details of how the Internet is implemented. As a consequence of Sockets providing a well-defined and stable interface, writing Internet applications exploded into a multi-billion dollar industry. Starting from the humble beginnings of the client/server paradigm and a handful of simple application programs like email, file transfer, and remote login, everyone now has access to an never-ending supply of cloud applications from their smartphones.

This section lays the foundation by revisiting the simplicity of a client program opening a socket so it can exchange messages with a server program, but today a rich software ecosystem is layered on top of the Socket API. This layer includes a plethora of cloud-based tools that lower the barrier for implementing scalable applications. We return to the interplay between the cloud and the network in every chapter, starting with the *Perspective* section at the end of Chapter 1.

Before describing the socket interface, it is important to keep two concerns separate in your mind. Each protocol provides a certain set of *services*, and the API provides a *syntax* by which those services can be invoked on a particular computer system. The implementation is then responsible for mapping the tangible set of operations and objects defined by the API onto the abstract set of services defined by the protocol. If you have done a good job of defining the interface, then it will be possible to use the syntax of the interface to invoke the services of many different protocols. Such generality was certainly a goal of the socket interface, although it's far from perfect.

The main abstraction of the socket interface, not surprisingly, is the *socket*. A good way to think of a socket is as the point where a local application process attaches to the network. The interface defines operations for creating a socket, attaching the socket to the network, sending/receiving messages through the socket, and closing the socket. To simplify the discussion, we will limit ourselves to showing how sockets are used with TCP.

The first step is to create a socket, which is done with the following operation:

```
int socket(int domain, int type, int protocol);
```

The reason that this operation takes three arguments is that the socket interface was designed to be general enough to support any underlying protocol suite. Specifically, the `domain` argument specifies the protocol *family* that is going to be used: `PF_INET` denotes the Internet family, `PF_UNIX` denotes the Unix pipe facility, and `PF_PACKET` denotes direct access to the network interface (i.e., it bypasses the TCP/IP protocol stack). The `type` argument indicates the semantics of the communication. `SOCK_STREAM` is used to denote a byte stream. `SOCK_DGRAM` is an alternative that denotes a message-oriented service, such as that provided by UDP. The `protocol` argument identifies the specific protocol that is going to be used. In our case, this argument is `UNSPEC` because the combination of `PF_INET` and `SOCK_STREAM` implies TCP. Finally, the return value from `socket` is a *handle* for the newly created socket—that is, an identifier by which we can refer to the socket in the future. It is given as an argument to subsequent operations on this socket.

The next step depends on whether you are a client or a server. On a server machine, the application process performs a *passive* open—the server says that it is prepared to accept connections, but it does not actually establish a connection. The server does this by invoking the following three operations:

```
int bind(int socket, struct sockaddr *address, int addr_len);
int listen(int socket, int backlog);
int accept(int socket, struct sockaddr *address, int *addr_len);
```

The `bind` operation, as its name suggests, binds the newly created `socket` to the specified `address`. This is the network address of the *local* participant—the server. Note that, when used with the Internet protocols, `address` is a data structure that includes both the IP address of the server and a TCP port number. Ports are used to indirectly identify processes. They are a form of *demux keys*. The port number is usually some well-known number specific to the service being offered; for example, web servers commonly accept connections on port 80.

The `listen` operation then defines how many connections can be pending on the specified `socket`. Finally, the `accept` operation carries out the passive open. It is a blocking operation that does not return until a remote participant has established a connection, and when it does complete it returns a *new* socket that corresponds to this just-established connection, and the `address` argument contains the *remote* participant's address. Note that when `accept` returns, the original socket that was given as an argument still exists and still corresponds to the passive open; it is used in future invocations of `accept`.

On the client machine, the application process performs an *active* open; that is, it says who it wants to communicate with by invoking the following single operation:

```
int connect(int socket, struct sockaddr *address, int addr_len);
```

This operation does not return until TCP has successfully established a connection, at which time the application is free to begin sending data. In this case, `address` contains the remote participant's address. In practice, the client usually specifies only the remote participant's address and lets the system fill in the local information. Whereas a server usually listens for messages on a well-known port, a client typically does not care which port it uses for itself; the OS simply selects an unused one.

Once a connection is established, the application processes invoke the following two operations to send and receive data:

```
int send(int socket, char *message, int msg_len, int flags);
int recv(int socket, char *buffer, int buf_len, int flags);
```

The first operation sends the given `message` over the specified `socket`, while the second operation receives a message from the specified `socket` into the given `buffer`. Both operations take a set of `flags` that control certain details of the operation.

1.4.2 Example Application

We now show the implementation of a simple client/server program that uses the socket interface to send messages over a TCP connection. The program also uses other Linux networking utilities, which we introduce as we go. Our application allows a user on one machine to type in and send text to a user on another machine. It is a simplified version of the Linux `talk` program, which is similar to the program at the core of instant messaging applications.

Client

We start with the client side, which takes the name of the remote machine as an argument. It calls the Linux utility to translate this name into the remote host's IP address. The next step is to construct the address data structure (`sin`) expected by the socket interface. Notice that this data structure specifies that we'll be using the socket to connect to the Internet (`AF_INET`). In our example, we use TCP port 5432 as the well-known server port; this happens to be a port that has not been assigned to any other Internet service. The final step in setting up the connection is to call `socket` and `connect`. Once the operation returns, the connection is established and the client program enters its main loop, which reads text from standard input and sends it over the socket.

```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

#define SERVER_PORT 5432
#define MAX_LINE 256

int
main(int argc, char * argv[])
{
  FILE *fp;
  struct hostent *hp;
  struct sockaddr_in sin;
  char *host;
  char buf[MAX_LINE];
  int s;
  int len;

  if (argc==2) {
    host = argv[1];
  }
  else {
    fprintf(stderr, "usage: simplex-talk host\n");
    exit(1);
  }

  /* translate host name into peer's IP address */
  hp = gethostbyname(host);
  if (!hp) {
    fprintf(stderr, "simplex-talk: unknown host: %s\n", host);
    exit(1);
  }

  /* build address data structure */
  bzero((char *)&sin, sizeof(sin));
  sin.sin_family = AF_INET;
  bcopy(hp->h_addr, (char *)&sin.sin_addr, hp->h_length);
  sin.sin_port = htons(SERVER_PORT);

  /* active open */
  if ((s = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
    perror("simplex-talk: socket");
    exit(1);
  }
  if (connect(s, (struct sockaddr *)&sin, sizeof(sin)) < 0)
  {
    perror("simplex-talk: connect");
    close(s);
    exit(1);
  }
```

(continues on next page)

(continued from previous page)

```
  /* main loop: get and send lines of text */
  while (fgets(buf, sizeof(buf), stdin)) {
    buf[MAX_LINE-1] = '\0';
    len = strlen(buf) + 1;
    send(s, buf, len, 0);
  }
}
```

Server

The server is equally simple. It first constructs the address data structure by filling in its own port number (SERVER_PORT). By not specifying an IP address, the application program is willing to accept connections on any of the local host's IP addresses. Next, the server performs the preliminary steps involved in a passive open; it creates the socket, binds it to the local address, and sets the maximum number of pending connections to be allowed. Finally, the main loop waits for a remote host to try to connect, and when one does, it receives and prints out the characters that arrive on the connection.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

#define SERVER_PORT  5432
#define MAX_PENDING  5
#define MAX_LINE     256

int
main()
{
  struct sockaddr_in sin;
  char buf[MAX_LINE];
  int buf_len, addr_len;
  int s, new_s;

  /* build address data structure */
  bzero((char *)&sin, sizeof(sin));
  sin.sin_family = AF_INET;
  sin.sin_addr.s_addr = INADDR_ANY;
  sin.sin_port = htons(SERVER_PORT);

  /* setup passive open */
  if ((s = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
    perror("simplex-talk: socket");
    exit(1);
  }
  if ((bind(s, (struct sockaddr *)&sin, sizeof(sin))) < 0) {
    perror("simplex-talk: bind");
    exit(1);
```

(continues on next page)

(continued from previous page)

```
  }
  listen(s, MAX_PENDING);

 /* wait for connection, then receive and print text */
  while(1) {
    if ((new_s = accept(s, (struct sockaddr *)&sin, &addr_len)) < 0) {
      perror("simplex-talk: accept");
      exit(1);
    }
    while (buf_len = recv(new_s, buf, sizeof(buf), 0))
      fputs(buf, stdout);
    close(new_s);
  }
}
```

1.5 Performance

Up to this point, we have focused primarily on the functional aspects of networks. Like any computer system, however, computer networks are also expected to perform well. This is because the effectiveness of computations distributed over the network often depends directly on the efficiency with which the network delivers the computation's data. While the old programming adage "first get it right and then make it fast" remains true, in networking it is often necessary to "design for performance." It is therefore important to understand the various factors that impact network performance.

1.5.1 Bandwidth and Latency

Network performance is measured in two fundamental ways: *bandwidth* (also called *throughput*) and *latency* (also called *delay*). The bandwidth of a network is given by the number of bits that can be transmitted over the network in a certain period of time. For example, a network might have a bandwidth of 10 million bits/second (Mbps), meaning that it is able to deliver 10 million bits every second. It is sometimes useful to think of bandwidth in terms of how long it takes to transmit each bit of data. On a 10-Mbps network, for example, it takes 0.1 microsecond (μs) to transmit each bit.

Bandwidth and throughput are subtly different terms. First of all, bandwidth is literally a measure of the width of a frequency band. For example, legacy voice-grade telephone lines supported a frequency band ranging from 300 to 3300 Hz; it was said to have a bandwidth of 3300 Hz - 300 Hz = 3000 Hz. If you see the word *bandwidth* used in a situation in which it is being measured in hertz, then it probably refers to the range of signals that can be accommodated.

When we talk about the bandwidth of a communication link, we normally refer to the number of bits per second that can be transmitted on the link. This is also sometimes called the *data rate.* We might say that the bandwidth of an Ethernet link is 10 Mbps. A useful distinction can also be made, however, between the maximum data rate that is available on the link and the number of bits per second that we can actually transmit over the link in practice. We tend to use the word *throughput* to refer to the *measured performance* of a system. Thus, because of various inefficiencies of implementation, a pair of nodes connected by a link with a bandwidth of 10 Mbps might achieve a throughput of only 2 Mbps. This would mean that an application on one host could send data to the other host at 2 Mbps.

Finally, we often talk about the bandwidth *requirements* of an application. This is the number of bits per second that it needs to transmit over the network to perform acceptably. For some applications, this might be "whatever I can get"; for others, it might be some fixed number (preferably not more than the available link bandwidth); and for others, it might be a number that varies with time. We will provide more on this topic later in this section.

While you can talk about the bandwidth of the network as a whole, sometimes you want to be more precise, focusing, for example, on the bandwidth of a single physical link or of a logical process-to-process channel. At the physical level, bandwidth is constantly improving, with no end in sight. Intuitively, if you think of a second of time as a distance you could measure with a ruler and bandwidth as how many bits fit in that distance, then you can think of each bit as a pulse of some width. For example, each bit on a 1-Mbps link is 1 μs wide, while each bit on a 2-Mbps link is 0.5 μs wide, as illustrated in Figure 1.16. The more sophisticated the transmitting and receiving technology, the narrower each bit can become and, thus, the higher the bandwidth. For logical process-to-process channels, bandwidth is also influenced by other factors, including how many times the software that implements the channel has to handle, and possibly transform, each bit of data.

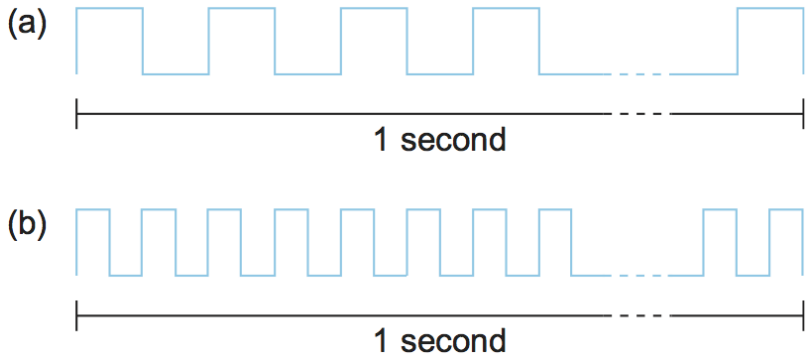


Figure 1.16.: Bits transmitted at a particular bandwidth can be regarded as having some width: (a) bits transmitted at 1 Mbps (each bit is 1 microsecond wide); (b) bits transmitted at 2 Mbps (each bit is 0.5 microseconds wide).

The second performance metric, latency, corresponds to how long it takes a message to travel from one end of a network to the other. (As with bandwidth, we could be focused on the latency of a single link or an end-to-end channel.) Latency is measured strictly in terms of time. For example, a transcontinental network might have a latency of 24 milliseconds (ms); that is, it takes a message 24 ms to travel from one coast of North America to the other. There are many situations in which it is more important to know how long it takes to send a message from one end of a network to the other and back, rather than the one-way latency. We call this the *round-trip time* (RTT) of the network.

We often think of latency as having three components. First, there is the speed-of-light propagation delay. This delay occurs because nothing, including a bit on a wire, can travel faster than the speed of light. If you know the distance between two points, you can calculate the speed-of-light latency, although you have to be careful because light travels across different media at different speeds: It travels at 3.0×10^8 m/s in a vacuum, 2.3×10^8 m/s in a copper cable, and 2.0×10^8 m/s in an optical fiber. Second, there is the amount of time it takes to transmit a unit of data. This is a function of the network bandwidth and the size of the packet in which the data is carried. Third, there may be queuing delays inside the network, since packet switches generally need to store packets for some time before forwarding them on an outbound link. So, we could define the total latency as

```
Latency = Propagation + Transmit + Queue
Propagation =  Distance/SpeedOfLight
Transmit = Size/Bandwidth
```

where `Distance` is the length of the wire over which the data will travel, `SpeedOfLight` is the effective speed of light over that wire, `Size` is the size of the packet, and `Bandwidth` is the bandwidth at which the packet is transmitted. Note that if the message contains only one bit and we are talking about a single link (as opposed to a whole network), then the `Transmit` and `Queue` terms are not relevant, and latency corresponds to the propagation delay only.

Bandwidth and latency combine to define the performance characteristics of a given link or channel. Their relative importance, however, depends on the application. For some applications, latency dominates bandwidth. For example, a client that sends a 1-byte message to a server and receives a 1-byte message in return is latency bound. Assuming that no serious computation is involved in preparing the response, the application will perform much differently on a transcontinental channel with a 100-ms RTT than it will on an across-the-room channel with a 1-ms RTT. Whether the channel is 1 Mbps or 100 Mbps is relatively insignificant, however, since the former implies that the time to transmit a byte (`Transimt`) is 8 μs and the latter implies `Transmit` = 0.08 μs.

In contrast, consider a digital library program that is being asked to fetch a 25-megabyte (MB) image—the more bandwidth that is available, the faster it will be able to return the image to the user. Here, the bandwidth of the channel dominates performance. To see this, suppose that the channel has a bandwidth of 10 Mbps. It will take 20 seconds to transmit the image ($25 \times 10^6 \times 8$-bits / (10×10^6 Mbps = 20 seconds), making it relatively unimportant if the image is on the other side of a 1-ms channel or a 100-ms channel; the difference between a 20.001-second response time and a 20.1-second response time is negligible.

Figure 1.17 gives you a sense of how latency or bandwidth can dominate performance in different circumstances. The graph shows how long it takes to move objects of various sizes (1 byte, 2 KB, 1 MB) across networks with RTTs ranging from 1 to 100 ms and link speeds of either 1.5 or 10 Mbps. We use logarithmic scales to show relative performance. For a 1-byte object (say, a keystroke), latency remains almost exactly equal to the RTT, so that you cannot distinguish between a 1.5-Mbps network and a 10-Mbps network. For a 2-KB object (say, an email message), the link speed makes quite a difference on a 1-ms RTT network but a negligible difference on a 100-ms RTT network. And for a 1-MB object (say, a digital image), the RTT makes no difference—it is the link speed that dominates performance across the full range of RTT.

Note that throughout this book we use the terms *latency* and *delay* in a generic way to denote how long it takes to perform a particular function, such as delivering a message or moving an object. When we are referring to the specific amount of time it takes a signal to propagate from one end of a link to another, we use the term *propagation delay*. Also, we make it clear in the context of the discussion whether we are referring to the one-way latency or the round-trip time.

As an aside, computers are becoming so fast that when we connect them to networks, it is sometimes useful to think, at least figuratively, in terms of *instructions per mile*. Consider what happens when a computer that is able to execute 100 billion instructions per second sends a message out on a channel with a 100-ms RTT. (To make the math easier, assume that the message covers a distance of 5000 miles.) If that computer sits idle the full 100 ms waiting for a reply message, then it has forfeited the ability to execute 10 billion instructions, or 2 million instructions per mile. It had better have been worth going over the network to justify this waste.

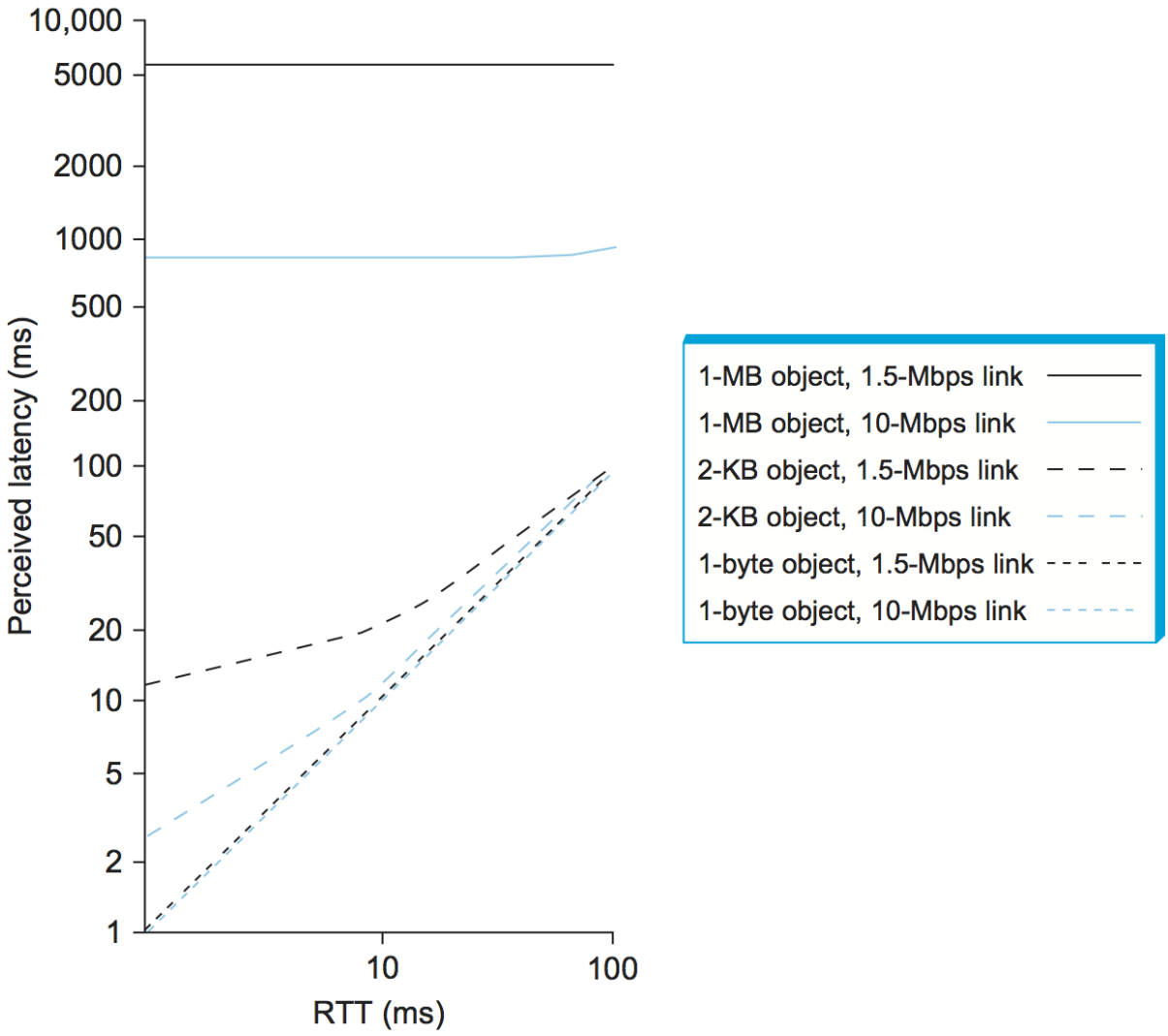


Figure 1.17.: Perceived latency (response time) versus round-trip time for various object sizes and link speeds.

1.5.2 Delay × Bandwidth Product

It is also useful to talk about the product of these two metrics, often called the *delay* × *bandwidth product*. Intuitively, if we think of a channel between a pair of processes as a hollow pipe (see Figure 1.18), where the latency corresponds to the length of the pipe and the bandwidth gives the diameter of the pipe, then the delay × bandwidth product gives the volume of the pipe—the maximum number of bits that could be in transit through the pipe at any given instant. Said another way, if latency (measured in time) corresponds to the length of the pipe, then given the width of each bit (also measured in time) you can calculate how many bits fit in the pipe. For example, a transcontinental channel with a one-way latency of 50 ms and a bandwidth of 45 Mbps is able to hold

$$50 \times 10^{-3} sec \times 45 \times 10^{6}\ bits/sec = 2.25 \times 10^{6}\ bits$$

or approximately 280 KB of data. In other words, this example channel (pipe) holds as many bytes as the memory of a personal computer from the early 1980s could hold.

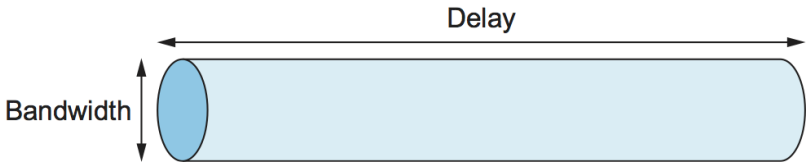


Figure 1.18.: Network as a pipe.

The delay × bandwidth product is important to know when constructing high-performance networks because it corresponds to how many bits the sender must transmit before the first bit arrives at the receiver. If the sender is expecting the receiver to somehow signal that bits are starting to arrive, and it takes another channel latency for this signal to propagate back to the sender, then the sender can send up one *RTT* × *bandwidth* worth of data before hearing from the receiver that all is well. The bits in the pipe are said to be "in flight," which means that if the receiver tells the sender to stop transmitting it might receive up to one RTT × bandwidth's worth of data before the sender manages to respond. In our example above, that amount corresponds to 5.5×10^6 bits (671 KB) of data. On the other hand, if the sender does not fill the pipe—i.e., does not send a whole RTT × bandwidth product's worth of data before it stops to wait for a signal—the sender will not fully utilize the network.

Note that most of the time we are interested in the RTT scenario, which we simply refer to as the delay × bandwidth product, without explicitly saying that "delay" is the RTT (i.e., multiply the one-way delay by two). Usually, whether the "delay" in delay × bandwidth means one-way latency or RTT is made clear by the context. Table 1.1 shows some examples of RTT × bandwidth products for some typical network links.

Table 1.1.: Example delay × bandwidth products.

Link Type	Bandwidth	One-Way Distance	RTT	RTT x Bandwidth
Wireless LAN	54 Mbps	50 m	0.33 μs	18 bits
Satellite	1 Gbps	35,000 km	230 ms	230 Mb
Cross-country fiber	10 Gbps	4,000 km	40 ms	400 Mb

1.5.3 High-Speed Networks

The seeming continual increase in bandwidth causes network designers to start thinking about what happens in the limit or, stated another way, what is the impact on network design of having infinite bandwidth available.

Although high-speed networks bring a dramatic change in the bandwidth available to applications, in many respects their impact on how we think about networking comes in what does *not* change as bandwidth increases: the speed of light. To quote Scotty from *Star Trek,* "Ye cannae change the laws of physics." In other words, "high speed" does not mean that latency improves at the same rate as bandwidth; the transcontinental RTT of a 1-Gbps link is the same 100 ms as it is for a 1-Mbps link.

To appreciate the significance of ever-increasing bandwidth in the face of fixed latency, consider what is required to transmit a 1-MB file over a 1-Mbps network versus over a 1-Gbps network, both of which have an RTT of 100 ms. In the case of the 1-Mbps network, it takes 80 round-trip times to transmit the file; during each RTT, 1.25% of the file is sent. In contrast, the same 1-MB file doesn't even come close to filling 1 RTT's worth of the 1-Gbps link, which has a delay × bandwidth product of 12.5 MB.

Figure 1.19 illustrates the difference between the two networks. In effect, the 1-MB file looks like a stream of data that needs to be transmitted across a 1-Mbps network, while it looks like a single packet on a 1-Gbps network. To help drive this point home, consider that a 1-MB file is to a 1-Gbps network what a 1-KB *packet* is to a 1-Mbps network.

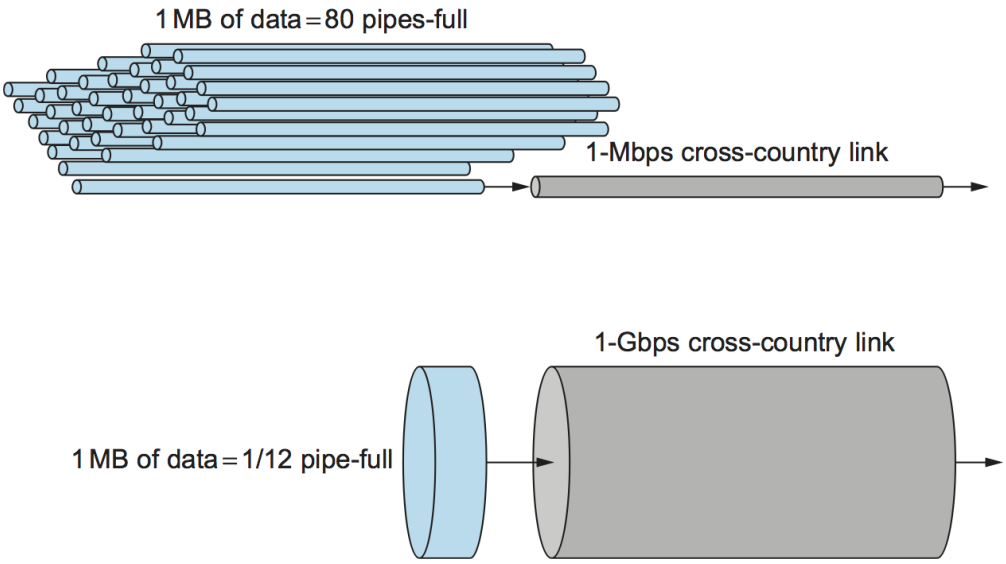


Figure 1.19.: Relationship between bandwidth and latency. A 1-MB file would fill the 1-Mbps link 80 times but only fill 1/12th of a 1-Gbps link.

Another way to think about the situation is that more data can be transmitted during each RTT on a high-speed network, so much so that a single RTT becomes a significant amount of time. Thus, while you wouldn't think twice about the difference between a file transfer taking 101 RTTs rather than 100 RTTs (a relative difference of only 1%), suddenly the difference between 1 RTT and 2 RTTs is significant—a 100% increase. In other words, latency, rather than throughput, starts to dominate our thinking about network design.

Perhaps the best way to understand the relationship between throughput and latency is to return to basics.

The effective end-to-end throughput that can be achieved over a network is given by the simple relationship

Throughput = TransferSize / TransferTime

where TransferTime includes not only the elements of one-way identified earlier in this section, but also any additional time spent requesting or setting up the transfer. Generally, we represent this relationship as

TransferTime = RTT + 1/Bandwidth x TransferSize

We use in this calculation to account for a request message being sent across the network and the data being sent back. For example, consider a situation where a user wants to fetch a 1-MB file across a 1-Gbps with a round-trip time of 100 ms. This includes both the transmit time for 1 MB (1 / 1 Gbps × 1 MB = 8 ms) and the 100-ms RTT, for a total transfer time of 108 ms. This means that the effective throughput will be

1 MB / 108 ms = 74.1 Mbps

not 1 Gbps. Clearly, transferring a larger amount of data will help improve the effective throughput, where in the limit an infinitely large transfer size will cause the effective throughput to approach the network bandwidth. On the other hand, having to endure more than 1 RTT—for example, to retransmit missing packets—will hurt the effective throughput for any transfer of finite size and will be most noticeable for small transfers.

1.5.4 Application Performance Needs

The discussion in this section has taken a network-centric view of performance; that is, we have talked in terms of what a given link or channel will support. The unstated assumption has been that application programs have simple needs—they want as much bandwidth as the network can provide. This is certainly true of the aforementioned digital library program that is retrieving a 250-MB image; the more bandwidth that is available, the faster the program will be able to return the image to the user.

However, some applications are able to state an upper limit on how much bandwidth they need. Video applications are a prime example. Suppose one wants to stream a video that is one quarter the size of a standard TV screen; that is, it has a resolution of 352 by 240 pixels. If each pixel is represented by 24 bits of information, as would be the case for 24-bit color, then the size of each frame would be (352 × 240 × 24) / 8 = 247.5 KB If the application needs to support a frame rate of 30 frames per second, then it might request a throughput rate of 75 Mbps. The ability of the network to provide more bandwidth is of no interest to such an application because it has only so much data to transmit in a given period of time.

Unfortunately, the situation is not as simple as this example suggests. Because the difference between any two adjacent frames in a video stream is often small, it is possible to compress the video by transmitting only the differences between adjacent frames. Each frame can also be compressed because not all the detail in a picture is readily perceived by a human eye. The compressed video does not flow at a constant rate, but varies with time according to factors such as the amount of action and detail in the picture and the compression algorithm being used. Therefore, it is possible to say what the average bandwidth requirement will be, but the instantaneous rate may be more or less.

The key issue is the time interval over which the average is computed. Suppose that this example video application can be compressed down to the point that it needs only 2 Mbps, on average. If it transmits 1 megabit in a 1-second interval and 3 megabits in the following 1-second interval, then over the 2-second interval it is transmitting at an average rate of 2 Mbps; however, this will be of little consolation to a channel

that was engineered to support no more than 2 megabits in any one second. Clearly, just knowing the average bandwidth needs of an application will not always suffice.

Generally, however, it is possible to put an upper bound on how large a burst an application like this is likely to transmit. A burst might be described by some peak rate that is maintained for some period of time. Alternatively, it could be described as the number of bytes that can be sent at the peak rate before reverting to the average rate or some lower rate. If this peak rate is higher than the available channel capacity, then the excess data will have to be buffered somewhere, to be transmitted later. Knowing how big of a burst might be sent allows the network designer to allocate sufficient buffer capacity to hold the burst.

Analogous to the way an application's bandwidth needs can be something other than "all it can get," an application's delay requirements may be more complex than simply "as little delay as possible." In the case of delay, it sometimes doesn't matter so much whether the one-way latency of the network is 100 ms or 500 ms as how much the latency varies from packet to packet. The variation in latency is called *jitter*.

Consider the situation in which the source sends a packet once every 33 ms, as would be the case for a video application transmitting frames 30 times a second. If the packets arrive at the destination spaced out exactly 33 ms apart, then we can deduce that the delay experienced by each packet in the network was exactly the same. If the spacing between when packets arrive at the destination—sometimes called the *inter-packet gap*—is variable, however, then the delay experienced by the sequence of packets must have also been variable, and the network is said to have introduced jitter into the packet stream, as shown in Figure 1.20. Such variation is generally not introduced in a single physical link, but it can happen when packets experience different queuing delays in a multihop packet-switched network. This queuing delay corresponds to the component of latency defined earlier in this section, which varies with time.

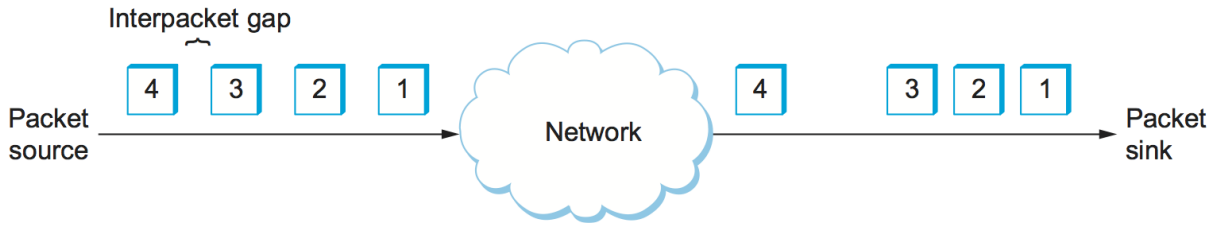


Figure 1.20.: Network-induced jitter.

To understand the relevance of jitter, suppose that the packets being transmitted over the network contain video frames, and in order to display these frames on the screen the receiver needs to receive a new one every 33 ms. If a frame arrives early, then it can simply be saved by the receiver until it is time to display it. Unfortunately, if a frame arrives late, then the receiver will not have the frame it needs in time to update the screen, and the video quality will suffer; it will not be smooth. Note that it is not necessary to eliminate jitter, only to know how bad it is. The reason for this is that if the receiver knows the upper and lower bounds on the latency that a packet can experience, it can delay the time at which it starts playing back the video (i.e., displays the first frame) long enough to ensure that in the future it will always have a frame to display when it needs it. The receiver delays the frame, effectively smoothing out the jitter, by storing it in a buffer.

Perspective: Feature Velocity

This chapter introduces some of the stakeholders in computer networks—network designers, application developers, end users, and network operators—to help motivate the technical requirements that shape how

networks are designed and built. This presumes all design decisions are purely technical, but of course, that's usually not the case. Many other factors, from market forces, to government policy, to ethical considerations, also influence how networks are designed and built.

Of these, the marketplace is the most influential, and corresponds to the interplay between network operators (e.g., AT&T, Comcast, Verizon, DT, NTT, China Unicom), network equipment venders (e.g., Cisco, Juniper, Ericsson, Nokia, Huawei, NEC), application and service providers (e.g., Facebook, Google, Amazon, Microsoft, Apple, Netflix, Spotify), and of course, subscribers and customers (i.e., individuals, but also enterprises and businesses). The lines between these players are not always crisp, with many companies playing multiple roles. The most notable example of this are the large cloud providers, who (a) build their own networking equipment using commodity components, (b) deploy and operate their own networks, and (c) provide end-user services and applications on top of their networks.

When you account for these other factors in the technical design process, you realize there are a couple of implicit assumptions in the textbook version of the story that need to be reevaluated. One is that designing a network is a one-time activity. Build it once and use it forever (modulo hardware upgrades so users can enjoy the benefits of the latest performance improvements). A second is that the job of building the network is largely divorced from the job of operating the network. Neither of these assumptions is quite right.

The network's design is clearly evolving, and we have documented these changes with each new edition of the textbook over the years. Doing that on a timeline measured in years has historically been good enough, but anyone that has downloaded and used the latest smartphone app knows how glacially slow anything measured in years is by today's standards. Designing for evolution has to be part of the decision making process.

On the second point, the companies that build networks are almost always the same ones that operate them. They are collectively known as *network operators*, and they include the companies listed above. But if we again look to the cloud for inspiration, we see that develop-and-operate isn't true just at the company level, but it is also how the fastest moving cloud companies organize their engineering teams: around the *DevOps* model. (If you are unfamiliar with DevOps, we recommend you read *Site Reliability Engineering: How Google Runs Production Systems* to see how it is practiced.)

What this all means is that computer networks are now in the midst of a major transformation, with network operators trying to simultaneously accelerate the pace of innovation (sometimes known as feature velocity) and yet continue to offer a reliable service (preserve stability). And they are increasingly doing this by adopting the best practices of cloud providers, which can be summarized as having two major themes: (1) take advantage of commodity hardware and move all intelligence into software, and (2) adopt agile engineering processes that break down barriers between development and operations.

This transformation is sometimes called the "cloudification" or "softwarization" of the network, and while the Internet has always had a robust software ecosystem, it has historically been limited to the applications running *on top of* the network (e.g., using the Socket API described in *Section 1.4*). What's changed is that today these same cloud-inspired engineering practices are being applied to the *internals* of the network. This new approach, known as *Software Defined Networks* (SDN), is a game changer, not so much in terms of how we address the fundamental technical challenges of framing, routing, fragmentation/reassembly, packet scheduling, congestion control, security, and so on, but in terms of how rapidly the network evolves to support new features.

This transformation is so important that we take it up again in the *Perspective* section at the end of each chapter. As these discussions will explore, what happens in the networking industry is partly about technology, but also partly about many other non-technical factors, all of which is a testament to how deeply

embedded the Internet is in our lives.

Broader Perspective

To continue reading about the cloudification of the Internet, see *Perspective: Race to the Edge*.

To learn more about DevOps, we recommend: Site Reliability Engineering: How Google Runs Production Systems, 2016.

CHAPTER

TWO

DIRECT LINKS

> It is a mistake to look too far ahead. Only one link in the chain of destiny can be handled at a time.
>
> —*Winston Churchill*

Problem: Connecting to a Network

In Chapter 1 we saw that networks consist of links interconnecting nodes. One of the fundamental problems we face is how to connect two nodes together. We also introduced the "cloud" abstraction to represent a network without revealing all of its internal complexities. So we also need to address the similar problem of connecting a host to a cloud. This, in effect, is the problem every Internet Service Provider (ISP) faces when it wants to connect a new customer to its network.

Whether we want to construct a trivial two-node network with one link or connect the one-billionth host to an existing network like the Internet, we need to address a common set of issues. First, we need some physical medium over which to make the connection. The medium may be a length of wire, a piece of optical fiber, or some less tangible medium (such as air) through which electromagnetic radiation (e.g., radio waves) can be transmitted. It may cover a small area (e.g., an office building) or a wide area (e.g., transcontinental).

Connecting two nodes with a suitable medium is only the first step, however. Five additional problems must be addressed before the nodes can successfully exchange packets, and once addressed, we will have provided *Layer 2* (L2) connectivity (using terminology from the OSI architecture).

The first is *encoding* bits onto the transmission medium so that they can be understood by a receiving node. Second is the matter of delineating the sequence of bits transmitted over the link into complete messages that can be delivered to the end node. This is the *framing* problem, and the messages delivered to the end hosts are often called *frames* (or sometimes *packets*). Third, because frames are sometimes corrupted during transmission, it is necessary to detect these errors and take the appropriate action; this is the *error detection* problem. The fourth issue is making a link appear reliable in spite of the fact that it corrupts frames from time to time. Finally, in those cases where the link is shared by multiple hosts—as is often the case with wireless links, for example—it is necessary to mediate access to this link. This is the *media access control* problem.

Although these five issues—encoding, framing, error detection, reliable delivery, and access mediation—can be discussed in the abstract, they are very real problems that are addressed in different ways by different networking technologies. This chapter considers these issues in the context of specific network technologies: point-to-point fiber links (for which SONET is the prevalent example); Carrier Sense Multiple Access

(CSMA) networks (of which classical Ethernet and Wi-Fi are the most famous examples); fiber-to-the home (for which PON is the dominant standard); and mobile wireless (where 4G is rapidly morphing into 5G).

The goal of this chapter is simultaneously to survey the available link-level technology and to explore these five fundamental issues. We will examine what it takes to make a wide variety of different physical media and link technologies useful as building blocks for the construction of robust, scalable networks.

2.1 Technology Landscape

Before diving into the challenges outlined in the problem statement at the beginning of this chapter, it is helpful to first get a lay of the land, which includes a wide array of link technologies. This is due, in part, to the diverse circumstances under which users are trying to connect their devices.

At one end of the spectrum, network operators that build global networks must deal with links that span hundreds or thousands of kilometers connecting refrigerator-sized routers. At the other end of the spectrum, a typical user encounters links mostly as a way to connect a computer to the existing Internet. Sometimes this link will be a wireless (Wi-Fi) link in a coffee shop; sometimes it's an Ethernet link in an office building or university; sometimes it is a smartphone connected to a cellular network; for an increasingly large slice of the population it is a fiber optic link provided by an ISP; and many others use some sort of copper wire or cable to connect. Fortunately, there are many common strategies used on these seemingly disparate types of links so that they can all be made reliable and useful to higher layers in the protocol stack. This chapter examines those strategies.

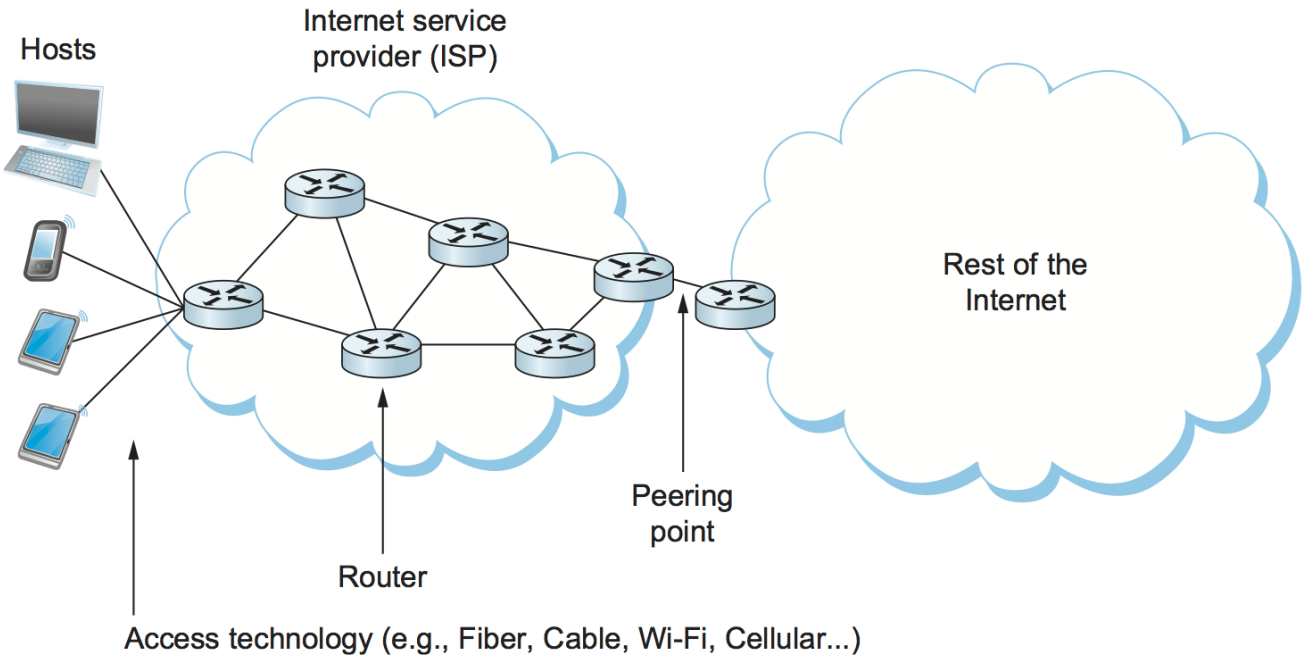


Figure 2.1.: An end-user's view of the Internet.

Figure 2.1 illustrates various types of links that might be found in today's Internet. On the left, we see a variety of end-user devices ranging from smartphones to tablets to full-fledged computers connected by various means to an ISP. While those links might use different technologies, they all look the same in this picture—a straight line connecting a device to a router. There are links that connect routers together inside

the ISP, as well as links that connect the ISP to the "rest of the Internet," which consists of lots of other ISPs and the hosts to which they connect.

These links all look alike not just because we're not very good artists but because part of the role of a network architecture is to provide a common abstraction of something as complex and diverse as a link. The idea is that your laptop or smartphone doesn't have to care what sort of link it is connected to—the only thing that matters is that it has a link to the Internet. Similarly, a router doesn't have to care what sort of link connects it to other routers—it can send a packet on the link with a pretty good expectation that the packet will reach the other end of the link.

How do we make all these different types of links look sufficiently alike to end users and routers? Essentially, we have to deal with all the physical limitations and shortcomings of links that exist in the real world. We sketched out some of these issues in the opening problem statement for this chapter, but before we can discuss these, we need to first introduce some simple physics. All of these links are made of some physical material that can propagate signals, such as radio waves or other sorts of electromagnetic radiation, but what we really want to do is send *bits*. In the later sections of this chapter, we'll look at how to encode bits for transmission on a physical medium, followed by the other issues mentioned above. By the end of this chapter, we'll understand how to send complete packets over just about any sort of link, no matter what physical medium is involved.

One way to characterize links, then, is by the medium they use—typically, copper wire in some form, such as twisted pair (some Ethernets and landline phones) and coaxial (cable); optical fiber, which is used for both fiber-to-the-home and many long-distance links in the Internet's backbone; or air/free space for wireless links.

Another important link characteristic is the *frequency*, measured in hertz, with which the electromagnetic waves oscillate. The distance between a pair of adjacent maxima or minima of a wave, typically measured in meters, is called the wave's *wavelength*. Since all electromagnetic waves travel at the speed of light (which in turn depends on the medium), that speed divided by the wave's frequency is equal to its wavelength. We have already seen the example of a voice-grade telephone line, which carries continuous electromagnetic signals ranging between 300 Hz and 3300 Hz; a 300-Hz wave traveling through copper would have a wavelength of

$$\text{SpeedOfLightInCopper / Frequency}$$

$$= 2/3 \times 3 \times 10^8/300$$

$$= 667 \times 10^3 \; meters$$

Generally, electromagnetic waves span a much wider range of frequencies, ranging from radio waves, to infrared light, to visible light, to x-rays and gamma rays. Figure 2.2 depicts the electromagnetic spectrum and shows which media are commonly used to carry which frequency bands.

What Figure 2.2 doesn't show is where the cellular network fits in. This is a bit complicated because the specific frequency bands that are licensed for cellular networks vary around the world, and even further complicated by the fact that network operators often simultaneously support both old/legacy technologies and new/next-generation technologies, each of which occupies a different frequency band. The high-level summary is that traditional cellular technologies range from 700-MHz to 2400-MHz, with new mid-spectrum allocations now happening at 6-GHz, and millimeter-wave (mmWave) allocations opening above 24-GHz. This mmWave band is likely to become an important part of the 5G mobile network.

So far we understand a link to be a physical medium carrying signals in the form of electromagnetic waves. Such links provide the foundation for transmitting all sorts of information, including the kind of data we

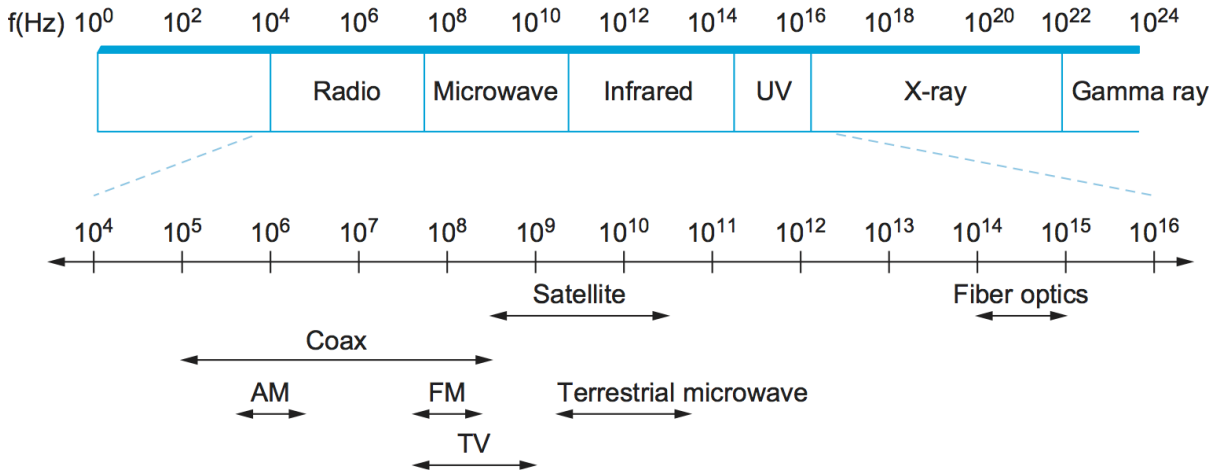


Figure 2.2.: Electromagnetic spectrum.

are interested in transmitting—binary data (1s and 0s). We say that the binary data is *encoded* in the signal. The problem of encoding binary data onto electromagnetic signals is a complex topic. To help make the topic more manageable, we can think of it as being divided into two layers. The lower layer is concerned with *modulation*—varying the frequency, amplitude, or phase of the signal to effect the transmission of information. A simple example of modulation is to vary the power (amplitude) of a single wavelength. Intuitively, this is equivalent to turning a light on and off. Because the issue of modulation is secondary to our discussion of links as a building block for computer networks, we simply assume that it is possible to transmit a pair of distinguishable signals—think of them as a "high" signal and a "low" signal—and we consider only the upper layer, which is concerned with the much simpler problem of encoding binary data onto these two signals. The next section discusses such encodings.

Another way to classify links is in terms of how they are used. Various economic and deployment issues tend to influence where different link types are found. Most consumers interact with the Internet either through wireless networks (which they encounter in coffee shops, airports, universities, etc.) or through so-called *last-mile* links (or alternatively, *access networks*) provided by an ISP, as illustrated in Figure 2.1. These link types are summarized in Table 2.1. They typically are chosen because they are cost-effective ways of reaching millions of consumers. DSL (Digital Subscriber Line), for example, is an older technology that was deployed over the existing twisted pair copper wires that already existed for plain old telephone services; G.Fast is a copper-based technology typically used within multi-dwelling apartment buildings, and PON (Passive Optical Network) is a newer technology that is commonly used to connect homes and businesses over recently deployed fiber.

Table 2.1.: Common services available for the last-mile connection to your home.

Service	Bandwidth
DSL (copper)	up to 100 Mbps
G.Fast (copper)	up to 1 Gbps
PON (optical)	up to 10 Gbps

And of course there is also the *mobile* or *cellular* network (also referred to as 4G, but which is rapidly evolving into 5G) that connects our mobile devices to the Internet. This technology can also serve as the

sole Internet connection into our homes or offices, but comes with the added benefit of allowing us to maintain Internet connectivity while moving from place to place.

These example technologies are common options for the last-mile connection to your home or business, but they are not sufficient for building a complete network from scratch. To do that, you'll also need some long-distance *backbone* links to interconnect cities. Modern backbone links are almost exclusively fiber today, and they typically use a technology called SONET (Synchronous Optical Network), which was originally developed to meet the demanding management requirements of telephone carriers.

Finally, in addition to last-mile, backbone, and mobile links, there are the links that you find inside a building or a campus—generally referred to as *local area networks* (LANs). Ethernet, and its wireless cousin Wi-Fi, are the dominant technologies in this space.

This survey of link types is by no means exhaustive, but it should have given you a taste of the diversity of link types that exist and some of the reasons for that diversity. In the coming sections, we will see how networking protocols can take advantage of that diversity and present a consistent view of the network to higher layers in spite of all the low-level complexity and economic factors.

2.2 Encoding

The first step in turning nodes and links into usable building blocks is to understand how to connect them in such a way that bits can be transmitted from one node to the other. As mentioned in the preceding section, signals propagate over physical links. The task, therefore, is to encode the binary data that the source node wants to send into the signals that the links are able to carry and then to decode the signal back into the corresponding binary data at the receiving node. We ignore the details of modulation and assume we are working with two discrete signals: high and low. In practice, these signals might correspond to two different voltages on a copper-based link, two different power levels on an optical link, or two different amplitudes on a radio transmission.

Most of the functions discussed in this chapter are performed by a *network adaptor*—a piece of hardware that connects a node to a link. The network adaptor contains a signalling component that actually encodes bits into signals at the sending node and decodes signals into bits at the receiving node. Thus, as illustrated in Figure 2.3, signals travel over a link between two signalling components, and bits flow between network adaptors.

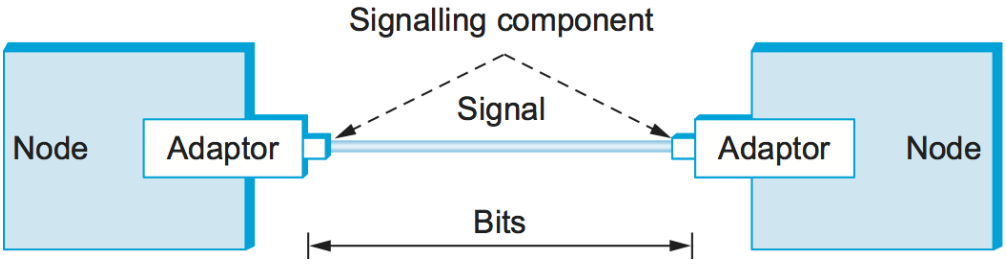


Figure 2.3.: Signals travel between signalling components; bits flow between adaptors.

Let's return to the problem of encoding bits onto signals. The obvious thing to do is to map the data value 1 onto the high signal and the data value 0 onto the low signal. This is exactly the mapping used by an encoding scheme called, cryptically enough, *non-return to zero* (NRZ). For example, Figure 2.4 schematically depicts the NRZ-encoded signal (bottom) that corresponds to the transmission of a particular sequence of bits (top).

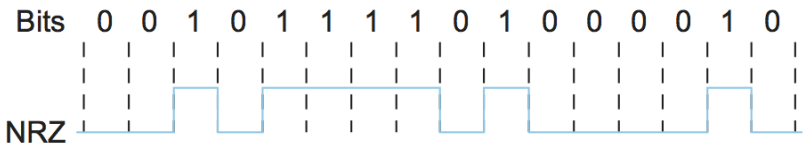


Figure 2.4.: NRZ encoding of a bit stream.

The problem with NRZ is that a sequence of several consecutive 1s means that the signal stays high on the link for an extended period of time; similarly, several consecutive 0s means that the signal stays low for a long time. There are two fundamental problems caused by long strings of 1s or 0s. The first is that it leads to a situation known as *baseline wander*. Specifically, the receiver keeps an average of the signal it has seen so far and then uses this average to distinguish between low and high signals. Whenever the signal is significantly lower than this average, the receiver concludes that it has just seen a 0; likewise, a signal that is significantly higher than the average is interpreted to be a 1. The problem, of course, is that too many consecutive 1s or 0s cause this average to change, making it more difficult to detect a significant change in the signal.

The second problem is that frequent transitions from high to low and *vice versa* are necessary to enable *clock recovery*. Intuitively, the clock recovery problem is that both the encoding and decoding processes are driven by a clock—every clock cycle the sender transmits a bit and the receiver recovers a bit. The sender's and the receiver's clocks have to be precisely synchronized in order for the receiver to recover the same bits the sender transmits. If the receiver's clock is even slightly faster or slower than the sender's clock, then it does not correctly decode the signal. You could imagine sending the clock to the receiver over a separate wire, but this is typically avoided because it makes the cost of cabling twice as high. So, instead, the receiver derives the clock from the received signal—the clock recovery process. Whenever the signal changes, such as on a transition from 1 to 0 or from 0 to 1, then the receiver knows it is at a clock cycle boundary, and it can resynchronize itself. However, a long period of time without such a transition leads to clock drift. Thus, clock recovery depends on having lots of transitions in the signal, no matter what data is being sent.

One approach that addresses this problem, called *non-return to zero inverted* (NRZI), has the sender make a transition from the current signal to encode a 1 and stay at the current signal to encode a 0. This solves the problem of consecutive 1s, but obviously does nothing for consecutive 0s. NRZI is illustrated in Figure 2.5. An alternative, called *Manchester encoding*, does a more explicit job of merging the clock with the signal by transmitting the exclusive OR of the NRZ-encoded data and the clock. (Think of the local clock as an internal signal that alternates from low to high; a low/high pair is considered one clock cycle.) The Manchester encoding is also illustrated in Figure 2.5. Observe that the Manchester encoding results in 0 being encoded as a low-to-high transition and 1 being encoded as a high-to-low transition. Because both 0s and 1s result in a transition to the signal, the clock can be effectively recovered at the receiver. (There is also a variant of the Manchester encoding, called *Differential Manchester*, in which a 1 is encoded with the first half of the signal equal to the last half of the previous bit's signal and a 0 is encoded with the first half of the signal opposite to the last half of the previous bit's signal.)

The problem with the Manchester encoding scheme is that it doubles the rate at which signal transitions are made on the link, which means that the receiver has half the time to detect each pulse of the signal. The rate at which the signal changes is called the link's *baud rate*. In the case of the Manchester encoding, the bit rate is half the baud rate, so the encoding is considered only 50% efficient. Keep in mind that if the receiver had been able to keep up with the faster baud rate required by the Manchester encoding in Figure 2.5, then both NRZ and NRZI could have been able to transmit twice as many bits in the same time period.

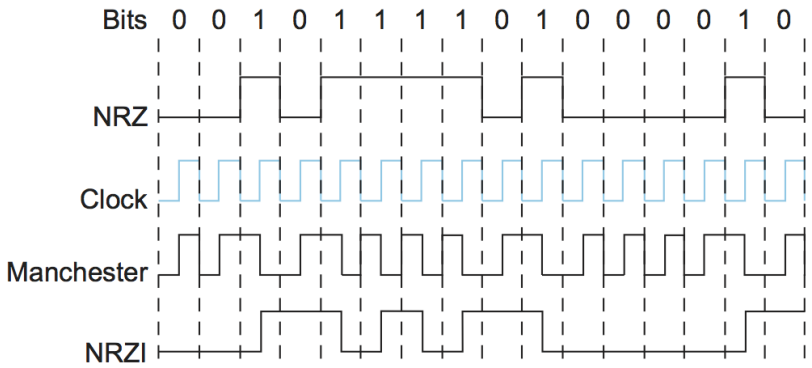


Figure 2.5.: Different encoding strategies.

Note that bit rate isn't necessarily less than or equal to the baud rate, as the Manchester encoding suggests. If the modulation scheme is able to utilize (and recognize) four different signals, as opposed to just two (e.g., "high" and "low"), then it is possible to encode two bits into each clock interval, resulting in a bit rate that is twice the baud rate. Similarly, being able to modulate among eight different signals means being able to transmit three bits per clock interval. In general, it is important to keep in mind we have over-simplified modulation, which is much more sophisticated than transmitting "high" and "low" signals. It is not uncommon to vary a combination of a signal's phase and amplitude, making it possible to encode 16 or even 64 different patterns (often dalled *symbols*) during each clock interval. *QAM (Quadrature Amplitude Modulation)* is widely used example of such a modulation scheme.

A final encoding that we consider, called *4B/5B*, attempts to address the inefficiency of the Manchester encoding without suffering from the problem of having extended durations of high or low signals. The idea of 4B/5B is to insert extra bits into the bit stream so as to break up long sequences of 0s or 1s. Specifically, every 4 bits of actual data are encoded in a 5-bit code that is then transmitted to the receiver; hence, the name 4B/5B. The 5-bit codes are selected in such a way that each one has no more than one leading 0 and no more than two trailing 0s. Thus, when sent back-to-back, no pair of 5-bit codes results in more than three consecutive 0s being transmitted. The resulting 5-bit codes are then transmitted using the NRZI encoding, which explains why the code is only concerned about consecutive 0s—NRZI already solves the problem of consecutive 1s. Note that the 4B/5B encoding results in 80% efficiency.

Table 2.2.: 4B/5B encoding.

4-bit Data Symbol	5-bit Code
0000	11110
0001	01001
0010	10100
0011	10101
0100	01010
0101	01011
0110	01110
0111	01111
1000	10010
1001	10011
1010	10110
1011	10111
1100	11010
1101	11011
1110	11100
1111	11101

Table 2.2 gives the 5-bit codes that correspond to each of the 16 possible 4-bit data symbols. Notice that since 5 bits are enough to encode 32 different codes, and we are using only 16 of these for data, there are 16 codes left over that we can use for other purposes. Of these, code `11111` is used when the line is idle, code `00000` corresponds to when the line is dead, and `00100` is interpreted to mean halt. Of the remaining 13 codes, 7 of them are not valid because they violate the "one leading 0, two trailing 0s," rule, and the other 6 represent various control symbols. Some of the framing protocols described later in this chapter make use of these control symbols.

2.3 Framing

Now that we have seen how to transmit a sequence of bits over a point-to-point link—from adaptor to adaptor—let's consider the scenario in Figure 2.6. Recall from Chapter 1 that we are focusing on packet-switched networks, which means that blocks of data (called *frames* at this level), not bit streams, are exchanged between nodes. It is the network adaptor that enables the nodes to exchange frames. When node A wishes to transmit a frame to node B, it tells its adaptor to transmit a frame from the node's memory. This results in a sequence of bits being sent over the link. The adaptor on node B then collects together the sequence of bits arriving on the link and deposits the corresponding frame in B's memory. Recognizing exactly what set of bits constitutes a frame—that is, determining where the frame begins and ends—is the central challenge faced by the adaptor.

There are several ways to address the framing problem. This section uses three different protocols to illustrate the various points in the design space. Note that while we discuss framing in the context of point-to-point links, the problem is a fundamental one that must also be addressed in multiple-access networks like Ethernet and Wi-Fi.

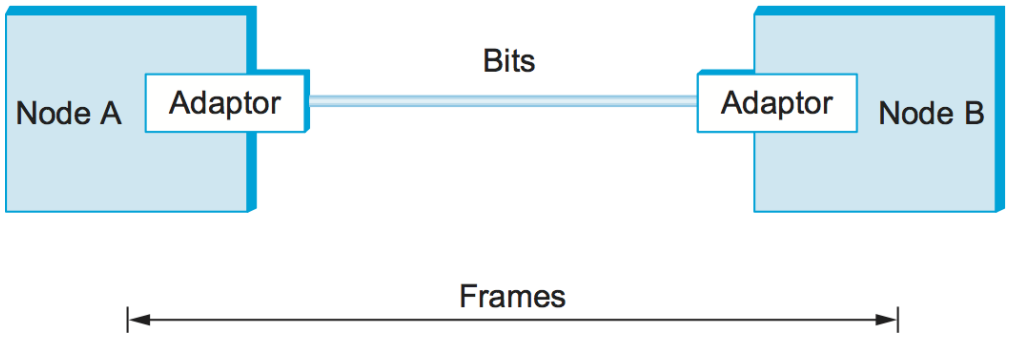


Figure 2.6.: Bits flow between adaptors, frames between hosts.

2.3.1 Byte-Oriented Protocols (PPP)

One of the oldest approaches to framing—it has its roots in connecting terminals to mainframes—is to view each frame as a collection of bytes (characters) rather than a collection of bits. Early examples of such *byte-oriented* protocols are the Binary Synchronous Communication (BISYNC) protocol developed by IBM in the late 1960s, and the Digital Data Communication Message Protocol (DDCMP) used in Digital Equipment Corporation's DECNET. (Once upon a time, large computer companies like IBM and DEC also built private networks for their customers.) The widely used Point-to-Point Protocol (PPP) is a recent example of this approach.

At a high level, there are two approaches to byte-oriented framing. The first is to use special characters known as *sentinel characters* to indicate where frames start and end. The idea is to denote the beginning of a frame by sending a special SYN (synchronization) character. The data portion of the frame is then sometimes contained between two more special characters: STX (start of text) and ETX (end of text). BISYNC used this approach. The problem with the sentinel approach, of course, is that one of the special characters might appear in the data portion of the frame. The standard way to overcome this problem by "escaping" the character by preceding it with a DLE (data-link-escape) character whenever it appears in the body of a frame; the DLE character is also escaped (by preceding it with an extra DLE) in the frame body. (C programmers may notice that this is analogous to the way a quotation mark is escaped by the backslash when it occurs inside a string.) This approach is often called *character stuffing* because extra characters are inserted in the data portion of the frame.

The alternative to detecting the end of a frame with a sentinel value is to include the number of bytes in the frame at the beginning of the frame, in the frame header. DDCMP used this approach. One danger with this approach is that a transmission error could corrupt the count field, in which case the end of the frame would not be correctly detected. (A similar problem exists with the sentinel-based approach if the ETX field becomes corrupted.) Should this happen, the receiver will accumulate as many bytes as the bad count field indicates and then use the error detection field to determine that the frame is bad. This is sometimes called a *framing error*. The receiver will then wait until it sees the next SYN character to start collecting the bytes that make up the next frame. It is therefore possible that a framing error will cause back-to-back frames to be incorrectly received.

The Point-to-Point Protocol (PPP), which is commonly used to carry Internet Protocol packets over various sorts of point-to-point links, uses sentinels and character stuffing. The format for a PPP frame is given in Figure 2.7.

This figure is the first of many that you will see in this book that are used to illustrate frame or packet

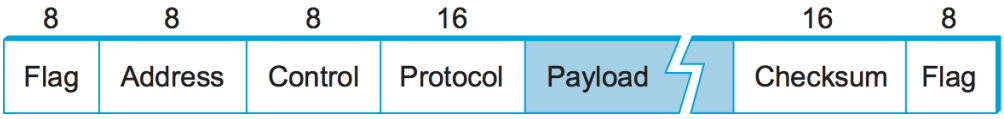


Figure 2.7.: PPP frame format.

formats, so a few words of explanation are in order. We show a packet as a sequence of labeled fields. Above each field is a number indicating the length of that field in bits. Note that the packets are transmitted beginning with the leftmost field.

The special start-of-text character, denoted as the `Flag` field is `01111110`. The `Address` and `Control` fields usually contain default values and so are uninteresting. The (Protocol) field is used for demultiplexing; it identifies the high-level protocol, such as IP. The frame payload size can be negotiated, but it is 1500 bytes by default. The `Checksum` field is either 2 (by default) or 4 bytes long. Note that despite its common name, this field is actually a CRC and not a checksum (as described in the next section).

The PPP frame format is unusual in that several of the field sizes are negotiated rather than fixed. This negotiation is conducted by a protocol called the Link Control Protocol (LCP). PPP and LCP work in tandem: LCP sends control messages encapsulated in PPP frames—such messages are denoted by an LCP identifier in the PPP (Protocol) field—and then turns around and changes PPP's frame format based on the information contained in those control messages. LCP is also involved in establishing a link between two peers when both sides detect that communication over the link is possible (e.g., when each optical receiver detects an incoming signal from the fiber to which it connects).

2.3.2 Bit-Oriented Protocols (HDLC)

Unlike byte-oriented protocols, a bit-oriented protocol is not concerned with byte boundaries—it simply views the frame as a collection of bits. These bits might come from some character set, such as ASCII; they might be pixel values in an image; or they could be instructions and operands from an executable file. The Synchronous Data Link Control (SDLC) protocol developed by IBM is an example of a bit-oriented protocol; SDLC was later standardized by the ISO as the High-Level Data Link Control (HDLC) protocol. In the following discussion, we use HDLC as an example; its frame format is given in Figure 2.8.

HDLC denotes both the beginning and the end of a frame with the distinguished bit sequence `01111110`. This sequence is also transmitted during any times that the link is idle so that the sender and receiver can keep their clocks synchronized. In this way, both protocols essentially use the sentinel approach. Because this sequence might appear anywhere in the body of the frame—in fact, the bits `01111110` might cross byte boundaries—bit-oriented protocols use the analog of the DLE character, a technique known as *bit stuffing*.

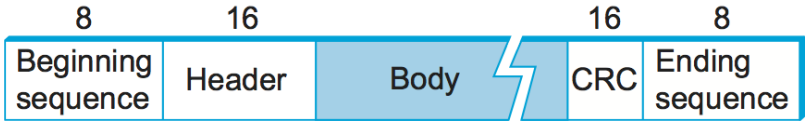


Figure 2.8.: HDLC frame format.

Bit stuffing in the HDLC protocol works as follows. On the sending side, any time five consecutive 1s have been transmitted from the body of the message (i.e., excluding when the sender is trying to transmit the distinguished `01111110` sequence), the sender inserts a 0 before transmitting the next bit. On the receiving

side, should five consecutive 1s arrive, the receiver makes its decision based on the next bit it sees (i.e., the bit following the five 1s). If the next bit is a 0, it must have been stuffed, and so the receiver removes it. If the next bit is a 1, then one of two things is true: Either this is the end-of-frame marker or an error has been introduced into the bit stream. By looking at the *next* bit, the receiver can distinguish between these two cases. If it sees a 0 (i.e., the last 8 bits it has looked at are `01111110`), then it is the end-of-frame marker; if it sees a 1 (i.e., the last 8 bits it has looked at are `01111111`), then there must have been an error and the whole frame is discarded. In the latter case, the receiver has to wait for the next `01111110` before it can start receiving again, and, as a consequence, there is the potential that the receiver will fail to receive two consecutive frames. Obviously, there are still ways that framing errors can go undetected, such as when an entire spurious end-of-frame pattern is generated by errors, but these failures are relatively unlikely. Robust ways of detecting errors are discussed in a later section.

An interesting characteristic of bit stuffing, as well as character stuffing, is that the size of a frame is dependent on the data that is being sent in the payload of the frame. It is in fact not possible to make all frames exactly the same size, given that the data that might be carried in any frame is arbitrary. (To convince yourself of this, consider what happens if the last byte of a frame's body is the ETX character.) A form of framing that ensures that all frames are the same size is described in the next subsection.

2.3.3 Clock-Based Framing (SONET)

A third approach to framing is exemplified by the Synchronous Optical Network (SONET) standard. For lack of a widely accepted generic term, we refer to this approach simply as *clock-based framing*. SONET was first proposed by Bell Communications Research (Bellcore), and then developed under the American National Standards Institute (ANSI) for digital transmission over optical fiber; it has since been adopted by the ITU-T. SONET has been for many years the dominant standard for long-distance transmission of data over optical networks.

An important point to make about SONET before we go any further is that the full specification is substantially larger than this book. Thus, the following discussion will necessarily cover only the high points of the standard. Also, SONET addresses both the framing problem and the encoding problem. It also addresses a problem that is very important for phone companies—the multiplexing of several low-speed links onto one high-speed link. (In fact, much of SONET's design reflects the fact that phone companies have to be concerned with multiplexing large numbers of the 64-kbps channels that traditionally are used for telephone calls.) We begin with SONET's approach to framing and discuss the other issues following.

As with the previously discussed framing schemes, a SONET frame has some special information that tells the receiver where the frame starts and ends; however, that is about as far as the similarities go. Notably, no bit stuffing is used, so that a frame's length does not depend on the data being sent. So the question to ask is "How does the receiver know where each frame starts and ends?" We consider this question for the lowest-speed SONET link, which is known as STS-1 and runs at 51.84 Mbps. An STS-1 frame is shown in Figure 2.9. It is arranged as 9 rows of 90 bytes each, and the first 3 bytes of each row are overhead, with the rest being available for data that is being transmitted over the link. The first 2 bytes of the frame contain a special bit pattern, and it is these bytes that enable the receiver to determine where the frame starts. However, since bit stuffing is not used, there is no reason why this pattern will not occasionally turn up in the payload portion of the frame. To guard against this, the receiver looks for the special bit pattern consistently, hoping to see it appearing once every 810 bytes, since each frame is $9 \times 90 = 810$ bytes long. When the special pattern turns up in the right place enough times, the receiver concludes that it is in sync and can then interpret the frame correctly.

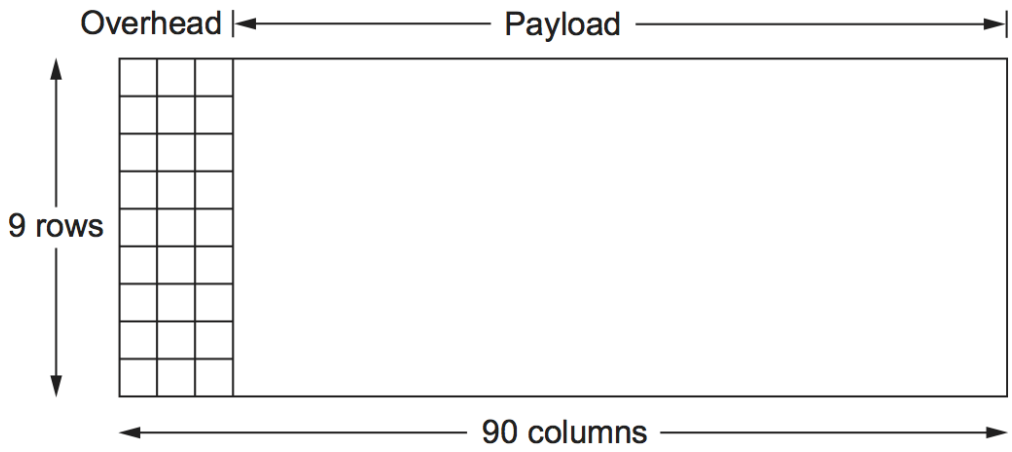


Figure 2.9.: A SONET STS-1 frame.

One of the things we are not describing due to the complexity of SONET is the detailed use of all the other overhead bytes. Part of this complexity can be attributed to the fact that SONET runs across the carrier's optical network, not just over a single link. (Recall that we are glossing over the fact that the carriers implement a network, and we are instead focusing on the fact that we can lease a SONET link from them and then use this link to build our own packet-switched network.) Additional complexity comes from the fact that SONET provides a considerably richer set of services than just data transfer. For example, 64 kbps of a SONET link's capacity is set aside for a voice channel that is used for maintenance.

The overhead bytes of a SONET frame are encoded using NRZ, the simple encoding described in the previous section where 1s are high and 0s are low. However, to ensure that there are plenty of transitions to allow the receiver to recover the sender's clock, the payload bytes are *scrambled*. This is done by calculating the exclusive OR (XOR) of the data to be transmitted and by the use of a well-known bit pattern. The bit pattern, which is 127 bits long, has plenty of transitions from 1 to 0, so that XORing it with the transmitted data is likely to yield a signal with enough transitions to enable clock recovery.

SONET supports the multiplexing of multiple low-speed links in the following way. A given SONET link runs at one of a finite set of possible rates, ranging from 51.84 Mbps (STS-1) to 39,813,120 Mbps (STS-768).[1] Note that all of these rates are integer multiples of STS-1. The significance for framing is that a single SONET frame can contain subframes for multiple lower-rate channels. A second related feature is that each frame is 125 μs long. This means that at STS-1 rates, a SONET frame is 810 bytes long, while at STS-3 rates, each SONET frame is 2430 bytes long. Notice the synergy between these two features: 3 × 810 = 2430, meaning that three STS-1 frames fit exactly in a single STS-3 frame.

Intuitively, the STS-N frame can be thought of as consisting of N STS-1 frames, where the bytes from these frames are interleaved; that is, a byte from the first frame is transmitted, then a byte from the second frame is transmitted, and so on. The reason for interleaving the bytes from each STS-N frame is to ensure that the bytes in each STS-1 frame are evenly paced; that is, bytes show up at the receiver at a smooth 51 Mbps, rather than all bunched up during one particular $1/N^{th}$ of the 125-μs interval.

Although it is accurate to view an STS-N signal as being used to multiplex N STS-1 frames, the payload

[1] STS stands for *Synchronous Transport Signal*, which is how SONET talks about frames. There is a parallel term—*Optical Carrier* (OC)—that is used to talk about the underlying optical signal that carries SONET frames. We say these two terms are parallel because STS-3 and OC-3, to use a concrete example, both imply a transmission rate of 155.52 Mbps. Since we're focused on framing here, we will stick with STS, but it is more likely that you will hear someone refer to an optical link by its "OC" name.

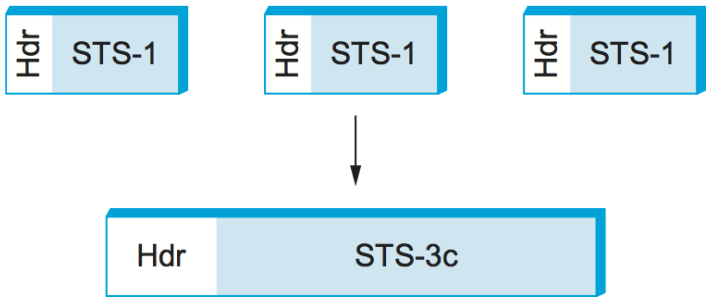


Figure 2.10.: Three STS-1 frames multiplexed onto one STS-3c frame.

from these STS-1 frames can be linked together to form a larger STS-N payload; such a link is denoted STS-Nc (for *concatenated*). One of the fields in the overhead is used for this purpose. Figure 2.10 schematically depicts concatenation in the case of three STS-1 frames being concatenated into a single STS-3c frame. The significance of a SONET link being designated as STS-3c rather than STS-3 is that, in the former case, the user of the link can view it as a single 155.25-Mbps pipe, whereas an STS-3 should really be viewed as three 51.84-Mbps links that happen to share a fiber.

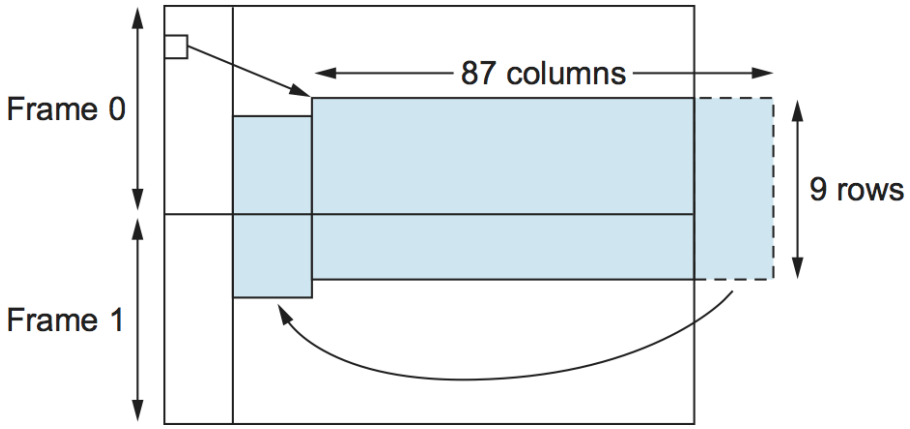


Figure 2.11.: SONET frames out of phase.

Finally, the preceding description of SONET is overly simplistic in that it assumes that the payload for each frame is completely contained within the frame. (Why wouldn't it be?) In fact, we should view the STS-1 frame just described as simply a placeholder for the frame, where the actual payload may *float* across frame boundaries. This situation is illustrated in Figure 2.11. Here we see both the STS-1 payload floating across two STS-1 frames and the payload shifted some number of bytes to the right and, therefore, wrapped around. One of the fields in the frame overhead points to the beginning of the payload. The value of this capability is that it simplifies the task of synchronizing the clocks used throughout the carriers' networks, which is something that carriers spend a lot of their time worrying about.

2.4 Error Detection

As discussed in Chapter 1, bit errors are sometimes introduced into frames. This happens, for example, because of electrical interference or thermal noise. Although errors are rare, especially on optical links, some mechanism is needed to detect these errors so that corrective action can be taken. Otherwise, the end

user is left wondering why the C program that successfully compiled just a moment ago now suddenly has a syntax error in it, when all that happened in the interim is that it was copied across a network file system.

There is a long history of techniques for dealing with bit errors in computer systems, dating back to at least the 1940s. Hamming and Reed-Solomon codes are two notable examples that were developed for use in punch card readers, when storing data on magnetic disks, and in early core memories. This section describes some of the error detection techniques most commonly used in networking.

Detecting errors is only one part of the problem. The other part is correcting errors once detected. Two basic approaches can be taken when the recipient of a message detects an error. One is to notify the sender that the message was corrupted so that the sender can retransmit a copy of the message. If bit errors are rare, then in all probability the retransmitted copy will be error free. Alternatively, some types of error detection algorithms allow the recipient to reconstruct the correct message even after it has been corrupted; such algorithms rely on *error-correcting codes*, discussed below.

One of the most common techniques for detecting transmission errors is a technique known as the *cyclic redundancy check* (CRC). It is used in nearly all the link-level protocols discussed in this chapter. This section outlines the basic CRC algorithm, but before discussing that approach, we first describe the simpler *checksum* scheme used by several Internet protocols.

The basic idea behind any error detection scheme is to add redundant information to a frame that can be used to determine if errors have been introduced. In the extreme, we could imagine transmitting two complete copies of the data. If the two copies are identical at the receiver, then it is probably the case that both are correct. If they differ, then an error was introduced into one (or both) of them, and they must be discarded. This is a rather poor error detection scheme for two reasons. First, it sends n redundant bits for an n-bit message. Second, many errors will go undetected—any error that happens to corrupt the same bit positions in the first and second copies of the message. In general, the goal of error detecting codes is to provide a high probability of detecting errors combined with a relatively low number of redundant bits.

Fortunately, we can do a lot better than this simple scheme. In general, we can provide quite strong error detection capability while sending only k redundant bits for an n-bit message, where k is much smaller than n. On an Ethernet, for example, a frame carrying up to 12,000 bits (1500 bytes) of data requires only a 32-bit CRC code, or as it is commonly expressed, uses CRC-32. Such a code will catch the overwhelming majority of errors, as we will see below.

We say that the extra bits we send are redundant because they add no new information to the message. Instead, they are derived directly from the original message using some well-defined algorithm. Both the sender and the receiver know exactly what that algorithm is. The sender applies the algorithm to the message to generate the redundant bits. It then transmits both the message and those few extra bits. When the receiver applies the same algorithm to the received message, it should (in the absence of errors) come up with the same result as the sender. It compares the result with the one sent to it by the sender. If they match, it can conclude (with high likelihood) that no errors were introduced in the message during transmission. If they do not match, it can be sure that either the message or the redundant bits were corrupted, and it must take appropriate action—that is, discarding the message or correcting it if that is possible.

One note on the terminology for these extra bits. In general, they are referred to as *error-detecting codes*. In specific cases, when the algorithm to create the code is based on addition, they may be called a *checksum*. We will see that the Internet checksum is appropriately named: It is an error check that uses a summing algorithm. Unfortunately, the word *checksum* is often used imprecisely to mean any form of error-detecting code, including CRCs. This can be confusing, so we urge you to use the word *checksum* only to apply to codes that actually do use addition and to use *error-detecting code* to refer to the general class of codes

described in this section.

2.4.1 Internet Checksum Algorithm

Our first approach to error detection is exemplified by the Internet checksum. Although it is not used at the link level, it nevertheless provides the same sort of functionality as CRCs, so we discuss it here.

The idea behind the Internet checksum is very simple—you add up all the words that are transmitted and then transmit the result of that sum. The result is the checksum. The receiver performs the same calculation on the received data and compares the result with the received checksum. If any transmitted data, including the checksum itself, is corrupted, then the results will not match, so the receiver knows that an error occurred.

You can imagine many different variations on the basic idea of a checksum. The exact scheme used by the Internet protocols works as follows. Consider the data being checksummed as a sequence of 16-bit integers. Add them together using 16-bit ones' complement arithmetic (explained below) and then take the ones' complement of the result. That 16-bit number is the checksum.

In ones' complement arithmetic, a negative integer (-x) is represented as the complement of x; that is, each bit of x is inverted. When adding numbers in ones' complement arithmetic, a carryout from the most significant bit needs to be added to the result. Consider, for example, the addition of -5 and -3 in ones' complement arithmetic on 4-bit integers: +5 is 0101, so -5 is 1010; +3 is 0011, so -3 is 1100. If we add 1010 and 1100, ignoring the carry, we get 0110. In ones' complement arithmetic, the fact that this operation caused a carry from the most significant bit causes us to increment the result, giving 0111, which is the ones' complement representation of -8 (obtained by inverting the bits in 1000), as we would expect.

The following routine gives a straightforward implementation of the Internet's checksum algorithm. The `count` argument gives the length of `buf` measured in 16-bit units. The routine assumes that `buf` has already been padded with 0s to a 16-bit boundary.

```
u_short
cksum(u_short *buf, int count)
{
    register u_long sum = 0;

    while (count--)
    {
        sum += *buf++;
        if (sum & 0xFFFF0000)
        {
            /* carry occurred, so wrap around */
            sum &= 0xFFFF;
            sum++;
        }
    }
    return ~(sum & 0xFFFF);
}
```

This code ensures that the calculation uses ones' complement arithmetic rather than the twos' complement that is used in most machines. Note the `if` statement inside the `while` loop. If there is a carry into the top 16 bits of `sum`, then we increment `sum` just as in the previous example.

Compared to our repetition code, this algorithm scores well for using a small number of redundant bits—only 16 for a message of any length—but it does not score extremely well for strength of error detection. For example, a pair of single-bit errors, one of which increments a word and one of which decrements another word by the same amount, will go undetected. The reason for using an algorithm like this in spite of its relatively weak protection against errors (compared to a CRC, for example) is simple: This algorithm is much easier to implement in software. Experience has suggested that a checksum of this form was adequate, but one reason it is adequate is that this checksum is the last line of defense in an end-to-end protocol. The majority of errors are picked up by stronger error detection algorithms, such as CRCs, at the link level.

2.4.2 Cyclic Redundancy Check

It should be clear by now that a major goal in designing error detection algorithms is to maximize the probability of detecting errors using only a small number of redundant bits. Cyclic redundancy checks use some fairly powerful mathematics to achieve this goal. For example, a 32-bit CRC gives strong protection against common bit errors in messages that are thousands of bytes long. The theoretical foundation of the cyclic redundancy check is rooted in a branch of mathematics called *finite fields*. While this may sound daunting, the basic ideas can be easily understood.

To start, think of an (n+1)-bit message as being represented by an n degree polynomial, that is, a polynomial whose highest-order term is x^n. The message is represented by a polynomial by using the value of each bit in the message as the coefficient for each term in the polynomial, starting with the most significant bit to represent the highest-order term. For example, an 8-bit message consisting of the bits 10011010 corresponds to the polynomial

$$M(x) = (1 \times x^7) + (0 \times x^6) + (0 \times x^5) + (1 \times x^4) + (1 \times x^3) + (0 \times x^2) + (1 \times x^1) + (0 \times x^0)$$

$$M(x) = x^7 + x^4 + x^3 + x^1$$

We can thus think of a sender and a receiver as exchanging polynomials with each other.

For the purposes of calculating a CRC, a sender and receiver have to agree on a *divisor* polynomial, $C(x)$. $C(x)$ is a polynomial of degree k. For example, suppose $C(x) = x^3+x^2+1$. In this case, $k = 3$. The answer to the question "Where did $C(x)$ come from?" is, in most practical cases, "You look it up in a book." In fact, the choice of $C(x)$ has a significant impact on what types of errors can be reliably detected, as we discuss below. There are a handful of divisor polynomials that are very good choices for various environments, and the exact choice is normally made as part of the protocol design. For example, the Ethernet standard uses a well-known polynomial of degree 32.

When a sender wishes to transmit a message $M(x)$ that is n+1 bits long, what is actually sent is the (n+1)-bit message plus k bits. We call the complete transmitted message, including the redundant bits, $P(x)$. What we are going to do is contrive to make the polynomial representing $P(x)$ exactly divisible by $C(x)$; we explain how this is achieved below. If $P(x)$ is transmitted over a link and there are no errors introduced during transmission, then the receiver should be able to divide $P(x)$ by $C(x)$ exactly, leaving a remainder of zero. On the other hand, if some error is introduced into $P(x)$ during transmission, then in all likelihood the received polynomial will no longer be exactly divisible by $C(x)$, and thus the receiver will obtain a nonzero remainder implying that an error has occurred.

It will help to understand the following if you know a little about polynomial arithmetic; it is just slightly different from normal integer arithmetic. We are dealing with a special class of polynomial arithmetic here, where coefficients may be only one or zero, and operations on the coefficients are performed using modulo 2

arithmetic. This is referred to as "polynomial arithmetic modulo 2." Since this is a networking book, not a mathematics text, let's focus on the key properties of this type of arithmetic for our purposes (which we ask you to accept on faith):

- Any polynomial $B(x)$ can be divided by a divisor polynomial $C(x)$ if $B(x)$ is of higher degree than $C(x)$.
- Any polynomial $B(x)$ can be divided once by a divisor polynomial $C(x)$ if $B(x)$ is of the same degree as $C(x)$.
- The remainder obtained when $B(x)$ is divided by $C(x)$ is obtained by performing the exclusive OR (XOR) operation on each pair of matching coefficients.

For example, the polynomial $x^3 + 1$ can be divided by $x^3 + x^2 + 1$ (because they are both of degree 3) and the remainder would be $0 \times x^3 + 1 \times x^2 + 0 \times x^1 + 0 \times x^0 = x^2$ (obtained by XORing the coefficients of each term). In terms of messages, we could say that 1001 can be divided by 1101 and leaves a remainder of 0100. You should be able to see that the remainder is just the bitwise exclusive OR of the two messages.

Now that we know the basic rules for dividing polynomials, we are able to do long division, which is necessary to deal with longer messages. An example appears below.

Recall that we wanted to create a polynomial for transmission that is derived from the original message $M(x)$, is k bits longer than $M(x)$, and is exactly divisible by $C(x)$. We can do this in the following way:

1. Multiply $M(x)$ by x^k; that is, add k zeros at the end of the message. Call this zero-extended message $T(x)$.
2. Divide $T(x)$ by $C(x)$ and find the remainder.
3. Subtract the remainder from $T(x)$.

It should be obvious that what is left at this point is a message that is exactly divisible by $C(x)$. We may also note that the resulting message consists of $M(x)$ followed by the remainder obtained in step 2, because when we subtracted the remainder (which can be no more than k bits long), we were just XORing it with the k zeros added in step 1. This part will become clearer with an example.

Consider the message $x^7 + x^4 + x^3 + x^1$, or 10011010. We begin by multiplying by x^3, since our divisor polynomial is of degree 3. This gives 10011010000. We divide this by $C(x)$, which corresponds to 1101 in this case. Figure 2.12 shows the polynomial long-division operation. Given the rules of polynomial arithmetic described above, the long-division operation proceeds much as it would if we were dividing integers. Thus, in the first step of our example, we see that the divisor 1101 divides once into the first four bits of the message (1001), since they are of the same degree, and leaves a remainder of 100 (1101 XOR 1001). The next step is to bring down a digit from the message polynomial until we get another polynomial with the same degree as $C(x)$, in this case 1001. We calculate the remainder again (100) and continue until the calculation is complete. Note that the "result" of the long division, which appears at the top of the calculation, is not really of much interest—it is the remainder at the end that matters.

You can see from the very bottom of Figure 2.12 that the remainder of the example calculation is 101. So we know that 10011010000 minus 101 would be exactly divisible by $C(x)$, and this is what we send. The minus operation in polynomial arithmetic is the logical XOR operation, so we actually send 10011010101. As noted above, this turns out to be just the original message with the remainder from the long division calculation appended to it. The recipient divides the received polynomial by $C(x)$ and, if the result is 0, concludes that there were no errors. If the result is nonzero, it may be necessary to discard the corrupted

message; with some codes, it may be possible to *correct* a small error (e.g., if the error affected only one bit). A code that enables error correction is called an *error-correcting code* (ECC).

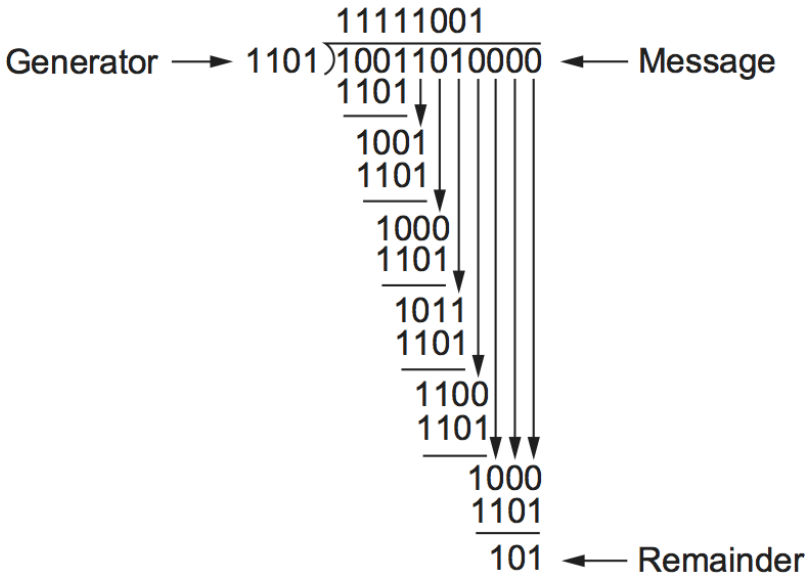


Figure 2.12.: CRC calculation using polynomial long division.

Now we will consider the question of where the polynomial $C(x)$ comes from. Intuitively, the idea is to select this polynomial so that it is very unlikely to divide evenly into a message that has errors introduced into it. If the transmitted message is $P(x)$, we may think of the introduction of errors as the addition of another polynomial $E(x)$, so the recipient sees $P(x) + E(x)$. The only way that an error could slip by undetected would be if the received message could be evenly divided by $C(x)$, and since we know that $P(x)$ can be evenly divided by $C(x)$, this could only happen if $E(x)$ can be divided evenly by $C(x)$. The trick is to pick $C(x)$ so that this is very unlikely for common types of errors.

One common type of error is a single-bit error, which can be expressed as $E(x) = x^i$ when it affects bit position *i*. If we select $C(x)$ such that thefirst and the last term (that is, the x^k and x^0 terms) are nonzero, then we already have a two-term polynomial that cannot divide evenly into the one term $E(x)$. Such a $C(x)$ can, therefore, detect all single-bit errors. In general, it is possible to prove that the following types of errors can be detected by a $C(x)$ with the stated properties:

- All single-bit errors, as long as the x^k and x^0 terms have nonzero coefficients
- All double-bit errors, as long as $C(x)$ has a factor with at least three terms
- Any odd number of errors, as long as $C(x)$ contains the factor $(x + 1)$

We have mentioned that it is possible to use codes that not only detect the presence of errors but also enable errors to be corrected. Since the details of such codes require yet more complex mathematics than that required to understand CRCs, we will not dwell on them here. However, it is worth considering the merits of correction versus detection.

At first glance, it would seem that correction is always better, since with detection we are forced to throw away the message and, in general, ask for another copy to be transmitted. This uses up bandwidth and may introduce latency while waiting for the retransmission. However, there is a downside to correction, as it generally requires a greater number of redundant bits to send an error-correcting code that is as strong (that

is, able to cope with the same range of errors) as a code that only detects errors. Thus, while error detection requires more bits to be sent when errors occur, error correction requires more bits to be sent *all the time*. As a result, error correction tends to be most useful when (1) errors are quite probable, as they may be, for example, in a wireless environment, or (2) the cost of retransmission is too high, for example, because of the latency involved retransmitting a packet over a satellite link.

The use of error-correcting codes in networking is sometimes referred to as *forward error correction* (FEC) because the correction of errors is handled "in advance" by sending extra information, rather than waiting for errors to happen and dealing with them later by retransmission. FEC is commonly used in wireless networks such as 802.11.

- Any "burst" error (i.e., sequence of consecutive errored bits) for which the length of the burst is less than k bits (Most burst errors of length greater than k bits can also be detected.)

Six versions of $C(x)$ are widely used in link-level protocols. For example, Ethernet uses CRC-32, which is defined as follows:

- CRC-32 = $x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$

Finally, we note that the CRC algorithm, while seemingly complex, is easily implemented in hardware using a k-bit shift register and XOR gates. The number of bits in the shift register equals the degree of the generator polynomial (k). Figure 2.13 shows the hardware that would be used for the generator $x^3 + x^2 + 1$ from our previous example. The message is shifted in from the left, beginning with the most significant bit and ending with the string of k zeros that is attached to the message, just as in the long division example. When all the bits have been shifted in and appropriately XORed, the register contains the remainder—that is, the CRC (most significant bit on the right). The position of the XOR gates is determined as follows: If the bits in the shift register are labeled 0 through $k - 1$, left to right, then put an XOR gate in front of bit n if there is a term x^n in the generator polynomial. Thus, we see an XOR gate in front of positions 0 and 2 for the generator $x^3 + x^2 + x^0$.

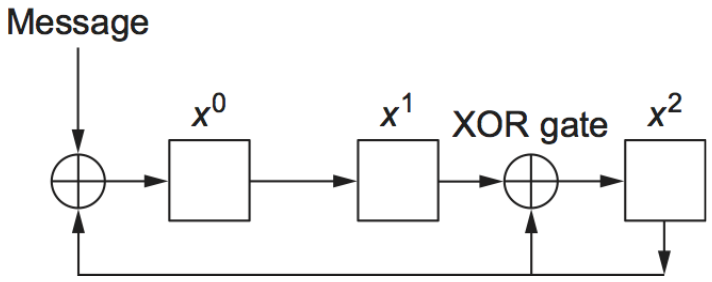


Figure 2.13.: CRC calculation using shift register.

2.5 Reliable Transmission

As we saw in the previous section, frames are sometimes corrupted while in transit, with an error code like CRC used to detect such errors. While some error codes are strong enough also to correct errors, in practice the overhead is typically too large to handle the range of bit and burst errors that can be introduced on a network link. Even when error-correcting codes are used (e.g., on wireless links) some errors will be too severe to be corrected. As a result, some corrupt frames must be discarded. A link-level protocol that wants to deliver frames reliably must somehow recover from these discarded (lost) frames.

It's worth noting that reliability is a function that *may* be provided at the link level, but many modern link technologies omit this function. Furthermore, reliable delivery is frequently provided at higher levels, including both transport and sometimes, the application layer. Exactly where it should be provided is a matter of some debate and depends on many factors. We describe the basics of reliable delivery here, since the principles are common across layers, but you should be aware that we're not just talking about a link-layer function.

Reliable delivery is usually accomplished using a combination of two fundamental mechanisms—*acknowledgments* and *timeouts*. An acknowledgment (ACK for short) is a small control frame that a protocol sends back to its peer saying that it has received an earlier frame. By control frame we mean a header without any data, although a protocol can *piggyback* an ACK on a data frame it just happens to be sending in the opposite direction. The receipt of an acknowledgment indicates to the sender of the original frame that its frame was successfully delivered. If the sender does not receive an acknowledgment after a reasonable amount of time, then it *retransmits* the original frame. This action of waiting a reasonable amount of time is called a *timeout*.

The general strategy of using acknowledgments and timeouts to implement reliable delivery is sometimes called *automatic repeat request* (abbreviated ARQ). This section describes three different ARQ algorithms using generic language; that is, we do not give detailed information about a particular protocol's header fields.

2.5.1 Stop-and-Wait

The simplest ARQ scheme is the *stop-and-wait* algorithm. The idea of stop-and-wait is straightforward: After transmitting one frame, the sender waits for an acknowledgment before transmitting the next frame. If the acknowledgment does not arrive after a certain period of time, the sender times out and retransmits the original frame.

Figure 2.14 illustrates timelines for four different scenarios that result from this basic algorithm. The sending side is represented on the left, the receiving side is depicted on the right, and time flows from top to bottom. Figure 2.14(a) shows the situation in which the ACK is received before the timer expires; (b) and (c) show the situation in which the original frame and the ACK, respectively, are lost; and (d) shows the situation in which the timeout fires too soon. Recall that by "lost" we mean that the frame was corrupted while in transit, that this corruption was detected by an error code on the receiver, and that the frame was subsequently discarded.

The packet timelines shown in this section are examples of a frequently used tool in teaching, explaining, and designing protocols. They are useful because they capture visually the behavior over time of a distributed system—something that can be quite hard to analyze. When designing a protocol, you often have to be prepared for the unexpected—a system crashes, a message gets lost, or something that you expected to happen quickly turns out to take a long time. These sorts of diagrams can often help us understand what might go wrong in such cases and thus help a protocol designer be prepared for every eventuality.

There is one important subtlety in the stop-and-wait algorithm. Suppose the sender sends a frame and the receiver acknowledges it, but the acknowledgment is either lost or delayed in arriving. This situation is illustrated in timelines (c) and (d) of Figure 2.14. In both cases, the sender times out and retransmits the original frame, but the receiver will think that it is the next frame, since it correctly received and acknowledged the first frame. This has the potential to cause duplicate copies of a frame to be delivered. To address this problem, the header for a stop-and-wait protocol usually includes a 1-bit sequence number—that is, the

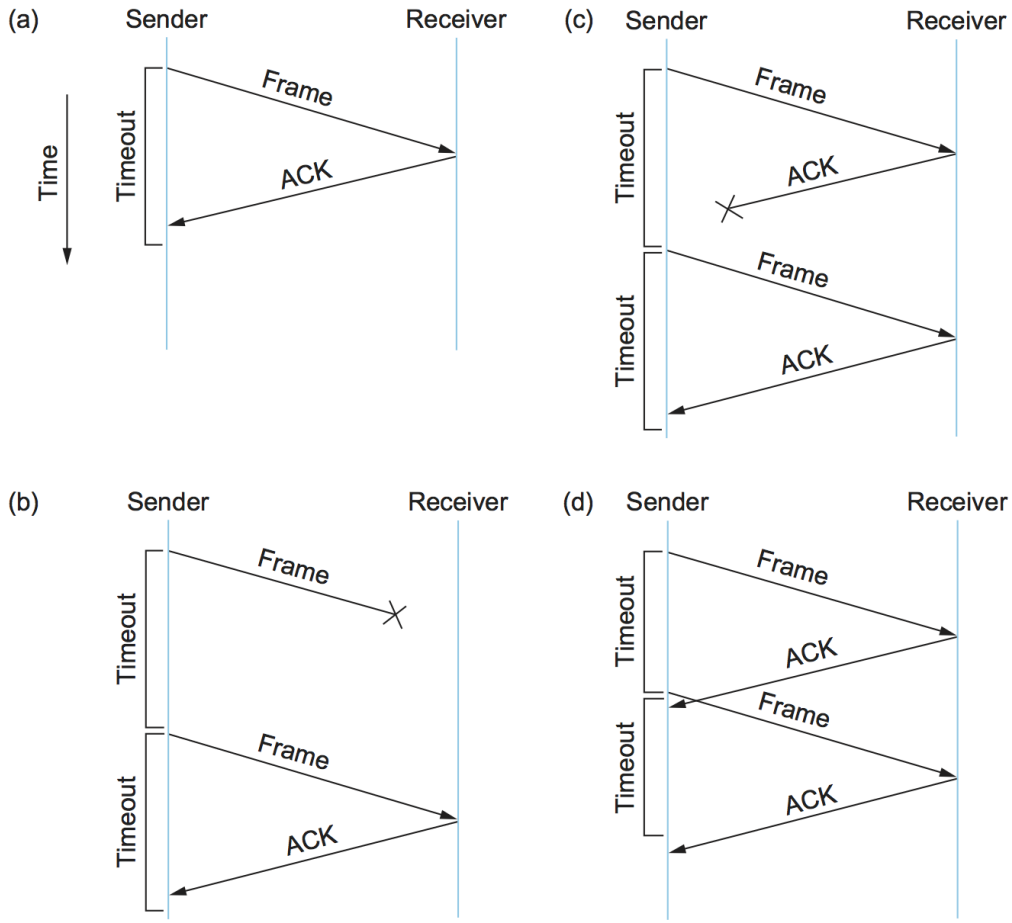


Figure 2.14.: Timeline showing four different scenarios for the stop-and-wait algorithm. (a) The ACK is received before the timer expires; (b) the original frame is lost; (c) the ACK is lost; (d) the timeout fires too soon.

sequence number can take on the values 0 and 1—and the sequence numbers used for each frame alternate, as illustrated in Figure 2.15. Thus, when the sender retransmits frame 0, the receiver can determine that it is seeing a second copy of frame 0 rather than the first copy of frame 1 and therefore can ignore it (the receiver still acknowledges it, in case the first ACK was lost).

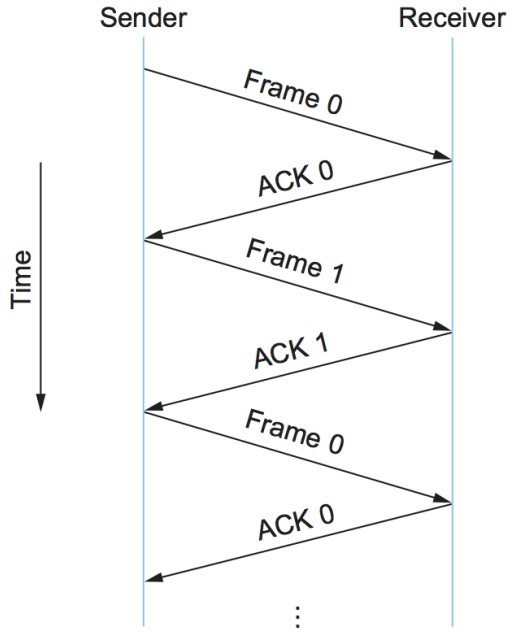


Figure 2.15.: Timeline for stop-and-wait with 1-bit sequence number.

The main shortcoming of the stop-and-wait algorithm is that it allows the sender to have only one outstanding frame on the link at a time, and this may be far below the link's capacity. Consider, for example, a 1.5-Mbps link with a 45-ms round-trip time. This link has a delay × bandwidth product of 67.5 Kb, or approximately 8 KB. Since the sender can send only one frame per RTT, and assuming a frame size of 1 KB, this implies a maximum sending rate of

Bits-Per-Frame / Time-Per-Frame = 1024 x 8 / 0.045 = 182 kbps

or about one-eighth of the link's capacity. To use the link fully, then, we'd like the sender to be able to transmit up to eight frames before having to wait for an acknowledgment.

Key Takeaway

The significance of the delay × bandwidth product is that it represents the amount of data that could be in transit. We would like to be able to send this much data without waiting for the first acknowledgment. The principle at work here is often referred to as *keeping the pipe full*. The algorithms presented in the following two subsections do exactly this. *[Next]*

2.5.2 Sliding Window

Consider again the scenario in which the link has a delay × bandwidth product of 8 KB and frames are 1 KB in size. We would like the sender to be ready to transmit the ninth frame at pretty much the same moment

that the ACK for the first frame arrives. The algorithm that allows us to do this is called *sliding window*, and an illustrative timeline is given in Figure 2.16.

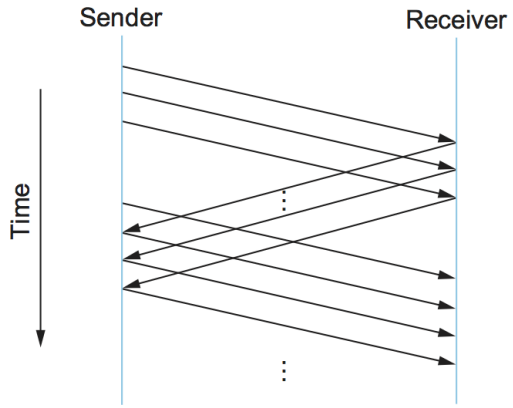


Figure 2.16.: Timeline for the sliding window algorithm.

The Sliding Window Algorithm

The sliding window algorithm works as follows. First, the sender assigns a *sequence number*, denoted `SeqNum`, to each frame. For now, let's ignore the fact that `SeqNum` is implemented by a finite-size header field and instead assume that it can grow infinitely large. The sender maintains three variables: The *send window size*, denoted `SWS`, gives the upper bound on the number of outstanding (unacknowledged) frames that the sender can transmit; `LAR` denotes the sequence number of the *last acknowledgment received*; and `LFS` denotes the sequence number of the *last frame sent*. The sender also maintains the following invariant:

```
LFS - LAR <= SWS
```

This situation is illustrated in Figure 2.17.

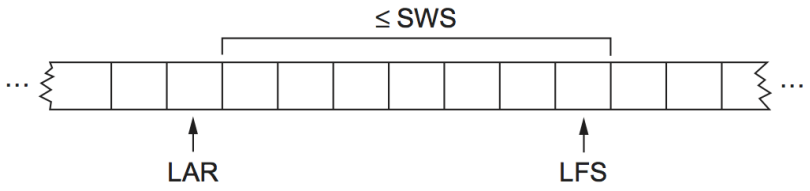


Figure 2.17.: Sliding window on sender.

When an acknowledgment arrives, the sender moves `LAR` to the right, thereby allowing the sender to transmit another frame. Also, the sender associates a timer with each frame it transmits, and it retransmits the frame should the timer expire before an ACK is received. Notice that the sender has to be willing to buffer up to `SWS` frames since it must be prepared to retransmit them until they are acknowledged.

The receiver maintains the following three variables: The *receive window size*, denoted `RWS`, gives the upper bound on the number of out-of-order frames that the receiver is willing to accept; `LAF` denotes the sequence number of the *largest acceptable frame*; and `LFR` denotes the sequence number of the *last frame received*. The receiver also maintains the following invariant:

```
LAF - LFR <= RWS
```

This situation is illustrated in Figure 2.18.

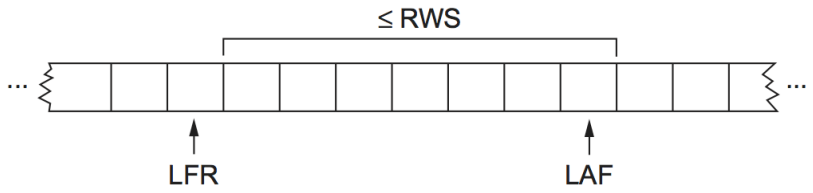


Figure 2.18.: Sliding window on receiver.

When a frame with sequence number `SeqNum` arrives, the receiver takes the following action. If `SeqNum <= LFR` or `SeqNum > LAF`, then the frame is outside the receiver's window and it is discarded. If `LFR < SeqNum <= LAF`, then the frame is within the receiver's window and it is accepted. Now the receiver needs to decide whether or not to send an ACK. Let `SeqNumToAck` denote the largest sequence number not yet acknowledged, such that all frames with sequence numbers less than or equal to `SeqNumToAck` have been received. The receiver acknowledges the receipt of `SeqNumToAck`, even if higher numbered packets have been received. This acknowledgment is said to be cumulative. It then sets `LFR = SeqNumToAck` and adjusts `LAF = LFR + RWS`.

For example, suppose `LFR = 5` (i.e., the last ACK the receiver sent was for sequence number 5), and `RWS = 4`. This implies that `LAF = 9`. Should frames 7 and 8 arrive, they will be buffered because they are within the receiver's window. However, no ACK needs to be sent since frame 6 has yet to arrive. Frames 7 and 8 are said to have arrived out of order. (Technically, the receiver could resend an ACK for frame 5 when frames 7 and 8 arrive.) Should frame 6 then arrive—perhaps it is late because it was lost the first time and had to be retransmitted, or perhaps it was simply delayed—the receiver acknowledges frame 8, bumps `LFR` to 8, and sets `LAF` to 12.[1] If frame 6 was in fact lost, then a timeout will have occurred at the sender, causing it to retransmit frame 6.

We observe that when a timeout occurs, the amount of data in transit decreases, since the sender is unable to advance its window until frame 6 is acknowledged. This means that when packet losses occur, this scheme is no longer keeping the pipe full. The longer it takes to notice that a packet loss has occurred, the more severe this problem becomes.

Notice that, in this example, the receiver could have sent a *negative acknowledgment* (NAK) for frame 6 as soon as frame 7 arrived. However, this is unnecessary since the sender's timeout mechanism is sufficient to catch this situation, and sending NAKs adds additional complexity to the receiver. Also, as we mentioned, it would have been legitimate to send additional acknowledgments of frame 5 when frames 7 and 8 arrived; in some cases, a sender can use duplicate ACKs as a clue that a frame was lost. Both approaches help to improve performance by allowing early detection of packet losses.

Yet another variation on this scheme would be to use *selective acknowledgments*. That is, the receiver could acknowledge exactly those frames it has received rather than just the highest numbered frame received in order. So, in the above example, the receiver could acknowledge the receipt of frames 7 and 8. Giving more information to the sender makes it potentially easier for the sender to keep the pipe full but adds complexity to the implementation.

[1] While it's unlikely that a packet could be delayed or arrive out-of-order on a point-to-point link, this same algorithm is used on multi-hop connections where such delays are possible.

The sending window size is selected according to how many frames we want to have outstanding on the link at a given time; SWS is easy to compute for a given delay × bandwidth product. On the other hand, the receiver can set RWS to whatever it wants. Two common settings are RWS = 1, which implies that the receiver will not buffer any frames that arrive out of order, and RWS = SWS, which implies that the receiver can buffer any of the frames the sender transmits. It makes no sense to set RWS > SWS since it's impossible for more than SWS frames to arrive out of order.

Finite Sequence Numbers and Sliding Window

We now return to the one simplification we introduced into the algorithm—our assumption that sequence numbers can grow infinitely large. In practice, of course, a frame's sequence number is specified in a header field of some finite size. For example, a 3-bit field means that there are eight possible sequence numbers, 0..7. This makes it necessary to reuse sequence numbers or, stated another way, sequence numbers wrap around. This introduces the problem of being able to distinguish between different incarnations of the same sequence numbers, which implies that the number of possible sequence numbers must be larger than the number of outstanding frames allowed. For example, stop-and-wait allowed one outstanding frame at a time and had two distinct sequence numbers.

Suppose we have one more number in our space of sequence numbers than we have potentially outstanding frames; that is, SWS <= MaxSeqNum - 1, where MaxSeqNum is the number of available sequence numbers. Is this sufficient? The answer depends on RWS. If RWS = 1, then MaxSeqNum >= SWS + 1 is sufficient. If RWS is equal to SWS, then having a MaxSeqNum just one greater than the sending window size is not good enough. To see this, consider the situation in which we have the eight sequence numbers 0 through 7, and SWS = RWS = 7. Suppose the sender transmits frames 0..6, they are successfully received, but the ACKs are lost. The receiver is now expecting frames 7, 0..5, but the sender times out and sends frames 0..6. Unfortunately, the receiver is expecting the second incarnation of frames 0..5 but gets the first incarnation of these frames. This is exactly the situation we wanted to avoid.

It turns out that the sending window size can be no more than half as big as the number of available sequence numbers when RWS = SWS, or stated more precisely,

```
SWS < (MaxSeqNum + 1)/ 2
```

Intuitively, what this is saying is that the sliding window protocol alternates between the two halves of the sequence number space, just as stop-and-wait alternates between sequence numbers 0 and 1. The only difference is that it continually slides between the two halves rather than discretely alternating between them.

Note that this rule is specific to the situation where RWS = SWS. We leave it as an exercise to determine the more general rule that works for arbitrary values of RWS and SWS. Also note that the relationship between the window size and the sequence number space depends on an assumption that is so obvious that it is easy to overlook, namely that frames are not reordered in transit. This cannot happen on a direct point-to-point link since there is no way for one frame to overtake another during transmission. However, we will see the sliding window algorithm used in a different environments, and we will need to devise another rule.

Implementation of Sliding Window

The following routines illustrate how we might implement the sending and receiving sides of the sliding window algorithm. The routines are taken from a working protocol named, appropriately enough, Sliding Window Protocol (SWP). So as not to concern ourselves with the adjacent protocols in the protocol graph, we denote the protocol sitting above SWP as the high-level protocol (HLP) and the protocol sitting below SWP as the link-level protocol (LLP).

We start by defining a pair of data structures. First, the frame header is very simple: It contains a sequence number (`SeqNum`) and an acknowledgment number (`AckNum`). It also contains a `Flags` field that indicates whether the frame is an ACK or carries data.

```
typedef u_char SwpSeqno;

typedef struct {
    SwpSeqno    SeqNum;    /* sequence number of this frame */
    SwpSeqno    AckNum;    /* ack of received frame */
    u_char      Flags;             /* up to 8 bits worth of flags */
} SwpHdr;
```

Next, the state of the sliding window algorithm has the following structure. For the sending side of the protocol, this state includes variables `LAR` and `LFS`, as described earlier in this section, as well as a queue that holds frames that have been transmitted but not yet acknowledged (`sendQ`). The sending state also includes a *counting semaphore* called `sendWindowNotFull`. We will see how this is used below, but generally a semaphore is a synchronization primitive that supports `semWait` and `semSignal` operations. Every invocation of `semSignal` increments the semaphore by 1, and every invocation of `semWait` decrements `s` by 1, with the calling process blocked (suspended) should decrementing the semaphore cause its value to become less than 0. A process that is blocked during its call to `semWait` will be allowed to resume as soon as enough `semSignal` operations have been performed to raise the value of the semaphore above 0.

For the receiving side of the protocol, the state includes the variable `NFE`. This is the *next frame expected*, the frame with a sequence number one more that the last frame received (LFR), described earlier in this section. There is also a queue that holds frames that have been received out of order (`recvQ`). Finally, although not shown, the sender and receiver sliding window sizes are defined by constants `SWS` and `RWS`, respectively.

```
typedef struct {
    /* sender side state: */
    SwpSeqno    LAR;        /* seqno of last ACK received */
    SwpSeqno    LFS;        /* last frame sent */
    Semaphore   sendWindowNotFull;
    SwpHdr      hdr;        /* pre-initialized header */
    struct sendQ_slot {
        Event   timeout;    /* event associated with send-timeout */
        Msg     msg;
    }   sendQ[SWS];

    /* receiver side state: */
    SwpSeqno    NFE;       /* seqno of next frame expected */
    struct recvQ_slot {
        int     received;  /* is msg valid? */
```

(continues on next page)

(continued from previous page)

```
        Msg     msg;
    }   recvQ[RWS];
} SwpState;
```

The sending side of SWP is implemented by procedure `sendSWP`. This routine is rather simple. First, `semWait` causes this process to block on a semaphore until it is OK to send another frame. Once allowed to proceed, `sendSWP` sets the sequence number in the frame's header, saves a copy of the frame in the transmit queue (`sendQ`), schedules a timeout event to handle the case in which the frame is not acknowledged, and sends the frame to the next-lower-level protocol, which we denote as `LINK`.

One detail worth noting is the call to `store_swp_hdr` just before the call to `msgAddHdr`. This routine translates the C structure that holds the SWP header (`state->hdr`) into a byte string that can be safely attached to the front of the message (`hbuf`). This routine (not shown) must translate each integer field in the header into network byte order and remove any padding that the compiler has added to the C structure. The issue of byte order is a non-trivial issue, but for now it is enough to assume that this routine places the most significant bit of a multiword integer in the byte with the highest address.

Another piece of complexity in this routine is the use of `semWait` and the `sendWindowNotFull` semaphore. `sendWindowNotFull` is initialized to the size of the sender's sliding window, `SWS` (this initialization is not shown). Each time the sender transmits a frame, the `semWait` operation decrements this count and blocks the sender should the count go to 0. Each time an ACK is received, the `semSignal` operation invoked in `deliverSWP` (see below) increments this count, thus unblocking any waiting sender.

```
static int
sendSWP(SwpState *state, Msg *frame)
{
    struct sendQ_slot *slot;
    hbuf[HLEN];

    /* wait for send window to open */
    semWait(&state->sendWindowNotFull);
    state->hdr.SeqNum = ++state->LFS;
    slot = &state->sendQ[state->hdr.SeqNum % SWS];
    store_swp_hdr(state->hdr, hbuf);
    msgAddHdr(frame, hbuf, HLEN);
    msgSaveCopy(&slot->msg, frame);
    slot->timeout = evSchedule(swpTimeout, slot, SWP_SEND_TIMEOUT);
    return send(LINK, frame);
}
```

Before continuing to the receive side of SWP, we need to reconcile a seeming inconsistency. On the one hand, we have been saying that a high-level protocol invokes the services of a low-level protocol by calling the `send` operation, so we would expect that a protocol that wants to send a message via SWP would call `send(SWP, packet)`. On the other hand, the procedure that implements SWP's send operation is called `sendSWP`, and its first argument is a state variable (`SwpState`). What gives? The answer is that the operating system provides glue code that translates the generic call to `send` into a protocol-specific call to `sendSWP`. This glue code maps the first argument to `send` (the magic protocol variable `SWP`) into both a function pointer to `sendSWP` and a pointer to the protocol state that SWP needs to do its job. The reason we have the high-level protocol indirectly invoke the protocol-specific function through the generic function call is that we want to limit how much information the high-level protocol has coded in it about the low-level

protocol. This makes it easier to change the protocol graph configuration at some time in the future.

Now we move on to SWP's protocol-specific implementation of the `deliver` operation, which is given in procedure `deliverSWP`. This routine actually handles two different kinds of incoming messages: ACKs for frames sent earlier from this node and data frames arriving at this node. In a sense, the ACK half of this routine is the counterpart to the sender side of the algorithm given in `sendSWP`. A decision as to whether the incoming message is an ACK or a data frame is made by checking the `Flags` field in the header. Note that this particular implementation does not support piggybacking ACKs on data frames.

When the incoming frame is an ACK, `deliverSWP` simply finds the slot in the transmit queue (`sendQ`) that corresponds to the ACK, cancels the timeout event, and frees the frame saved in that slot. This work is actually done in a loop since the ACK may be cumulative. The only other thing to notice about this case is the call to subroutine `swpInWindow`. This subroutine, which is given below, ensures that the sequence number for the frame being acknowledged is within the range of ACKs that the sender currently expects to receive.

When the incoming frame contains data, `deliverSWP` first calls `msgStripHdr` and `load_swp_hdr` to extract the header from the frame. Routine `load_swp_hdr` is the counterpart to `store_swp_hdr` discussed earlier; it translates a byte string into the C data structure that holds the SWP header. `deliverSWP` then calls `swpInWindow` to make sure the sequence number of the frame is within the range of sequence numbers that it expects. If it is, the routine loops over the set of consecutive frames it has received and passes them up to the higher-level protocol by invoking the `deliverHLP` routine. It also sends a cumulative ACK back to the sender, but does so by looping over the receive queue (it does not use the `SeqNumToAck` variable used in the prose description given earlier in this section).

```
static int
deliverSWP(SwpState state, Msg *frame)
{
    SwpHdr   hdr;
    char     *hbuf;

    hbuf = msgStripHdr(frame, HLEN);
    load_swp_hdr(&hdr, hbuf)
    if (hdr->Flags & FLAG_ACK_VALID)
    {
        /* received an acknowledgment--do SENDER side */
        if (swpInWindow(hdr.AckNum, state->LAR + 1, state->LFS))
        {
            do
            {
                struct sendQ_slot *slot;

                slot = &state->sendQ[++state->LAR % SWS];
                evCancel(slot->timeout);
                msgDestroy(&slot->msg);
                semSignal(&state->sendWindowNotFull);
            } while (state->LAR != hdr.AckNum);
        }
    }

    if (hdr.Flags & FLAG_HAS_DATA)
    {
```

(continues on next page)

(continued from previous page)

```
        struct recvQ_slot *slot;

        /* received data packet--do RECEIVER side */
        slot = &state->recvQ[hdr.SeqNum % RWS];
        if (!swpInWindow(hdr.SeqNum, state->NFE, state->NFE + RWS - 1))
        {
            /* drop the message */
            return SUCCESS;
        }
        msgSaveCopy(&slot->msg, frame);
        slot->received = TRUE;
        if (hdr.SeqNum == state->NFE)
        {
            Msg m;

            while (slot->received)
            {
                deliver(HLP, &slot->msg);
                msgDestroy(&slot->msg);
                slot->received = FALSE;
                slot = &state->recvQ[++state->NFE % RWS];
            }
            /* send ACK: */
            prepare_ack(&m, state->NFE - 1);
            send(LINK, &m);
            msgDestroy(&m);
        }
    }
    return SUCCESS;
}
```

Finally,`swpInWindow` is a simple subroutine that checks to see if a given sequence number falls between some minimum and maximum sequence number.

```
static bool
swpInWindow(SwpSeqno seqno, SwpSeqno min, SwpSeqno max)
{
    SwpSeqno pos, maxpos;

    pos    = seqno - min;       /* pos *should* be in range [0..MAX) */
    maxpos = max - min + 1;     /* maxpos is in range [0..MAX] */
    return pos < maxpos;
}
```

Frame Order and Flow Control

The sliding window protocol is perhaps the best known algorithm in computer networking. What is easily confused about the algorithm, however, is that it can be used to serve three different roles. The first role is the one we have been concentrating on in this section—to reliably deliver frames across an unreliable link. (In general, the algorithm can be used to reliably deliver messages across an unreliable network.) This is the

core function of the algorithm.

The second role that the sliding window algorithm can serve is to preserve the order in which frames are transmitted. This is easy to do at the receiver—since each frame has a sequence number, the receiver just makes sure that it does not pass a frame up to the next-higher-level protocol until it has already passed up all frames with a smaller sequence number. That is, the receiver buffers (i.e., does not pass along) out-of-order frames. The version of the sliding window algorithm described in this section does preserve frame order, although we could imagine a variation in which the receiver passes frames to the next protocol without waiting for all earlier frames to be delivered. A question we should ask ourselves is whether we really need the sliding window protocol to keep the frames in order at the link level, or whether, instead, this functionality should be implemented by a protocol higher in the stack.

The third role that the sliding window algorithm sometimes plays is to support *flow control*—a feedback mechanism by which the receiver is able to throttle the sender. Such a mechanism is used to keep the sender from over-running the receiver—that is, from transmitting more data than the receiver is able to process. This is usually accomplished by augmenting the sliding window protocol so that the receiver not only acknowledges frames it has received but also informs the sender of how many frames it has room to receive. The number of frames that the receiver is capable of receiving corresponds to how much free buffer space it has. As in the case of ordered delivery, we need to make sure that flow control is necessary at the link level before incorporating it into the sliding window protocol.

Key Takeaway

One important concept to take away from this discussion is the system design principle we call *separation of concerns*. That is, you must be careful to distinguish between different functions that are sometimes rolled together in one mechanism, and you must make sure that each function is necessary and being supported in the most effective way. In this particular case, reliable delivery, ordered delivery, and flow control are sometimes combined in a single sliding window protocol, and we should ask ourselves if this is the right thing to do at the link level. *[Next]*

2.5.3 Concurrent Logical Channels

The data link protocol used in the original ARPANET provides an interesting alternative to the sliding window protocol, in that it is able to keep the pipe full while still using the simple stop-and-wait algorithm. One important consequence of this approach is that the frames sent over a given link are not kept in any particular order. The protocol also implies nothing about flow control.

The idea underlying the ARPANET protocol, which we refer to as *concurrent logical channels*, is to multiplex several logical channels onto a single point-to-point link and to run the stop-and-wait algorithm on each of these logical channels. There is no relationship maintained among the frames sent on any of the logical channels, yet because a different frame can be outstanding on each of the several logical channels the sender can keep the link full.

More precisely, the sender keeps 3 bits of state for each channel: a boolean, saying whether the channel is currently busy; the 1-bit sequence number to use the next time a frame is sent on this logical channel; and the next sequence number to expect on a frame that arrives on this channel. When the node has a frame to send, it uses the lowest idle channel, and otherwise it behaves just like stop-and-wait.

In practice, the ARPANET supported 8 logical channels over each ground link and 16 over each satellite link. In the ground-link case, the header for each frame included a 3-bit channel number and a 1-bit sequence number, for a total of 4 bits. This is exactly the number of bits the sliding window protocol requires to support up to 8 outstanding frames on the link when `RWS = SWS`.

2.6 Multi-Access Networks

Developed in the mid-1970s by researchers at the Xerox Palo Alto Research Center (PARC), the Ethernet eventually became the dominant local area networking technology, emerging from a pack of competing technologies. Today, it competes mainly with 802.11 wireless networks but remains extremely popular in campus networks and data centers. The more general name for the technology behind the Ethernet is Carrier Sense, Multiple Access with Collision Detect (CSMA/CD).

As indicated by the CSMA name, the Ethernet is a multiple-access network, meaning that a set of nodes sends and receives frames over a shared link. You can, therefore, think of an Ethernet as being like a bus that has multiple stations plugged into it. The "carrier sense" in CSMA/CD means that all the nodes can distinguish between an idle and a busy link, and "collision detect" means that a node listens as it transmits and can therefore detect when a frame it is transmitting has interfered (collided) with a frame transmitted by another node.

The Ethernet has its roots in an early packet radio network, called Aloha, developed at the University of Hawaii to support computer communication across the Hawaiian Islands. Like the Aloha network, the fundamental problem faced by the Ethernet is how to mediate access to a shared medium fairly and efficiently (in Aloha, the medium was the atmosphere, while in the Ethernet the medium was originally a coax cable). The core idea in both Aloha and the Ethernet is an algorithm that controls when each node can transmit.

Modern Ethernet links are now largely point to point; that is, they connect one host to an Ethernet *switch*, or they interconnect switches. As a consequence, the "multiple access" algorithm is not used much in today's wired Ethernets, but a variant is now used in wireless networks, such as 802.11 networks (also known as Wi-Fi). Due to the enormous influence of Ethernet, we chose to describe its classic algorithm here, and then explain how it has been adapted to Wi-Fi in the next section. We will also discuss Ethernet switches elsewhere. For now, we'll focus on how a single Ethernet link works.

Digital Equipment Corporation and Intel Corporation joined Xerox to define a 10-Mbps Ethernet standard in 1978. This standard then formed the basis for IEEE standard 802.3, which additionally defines a much wider collection of physical media over which an Ethernet can operate, including 100-Mbps, 1-Gbps, 10-Gbps, 40-Gbps, and 100-Gbps versions.

2.6.1 Physical Properties

Ethernet segments were originally implemented using coaxial cable of length up to 500 m. (Modern Ethernets use twisted copper pairs, usually a particular type known as "Category 5," or optical fibers, and in some cases can be quite a lot longer than 500 m.) This cable was similar to the type used for cable TV. Hosts connected to an Ethernet segment by tapping into it. A *transceiver*, a small device directly attached to the tap, detected when the line was idle and drove the signal when the host was transmitting. It also received incoming signals. The transceiver, in turn, connected to an Ethernet adaptor, which was plugged into the host. This configuration is shown in Figure 2.19.

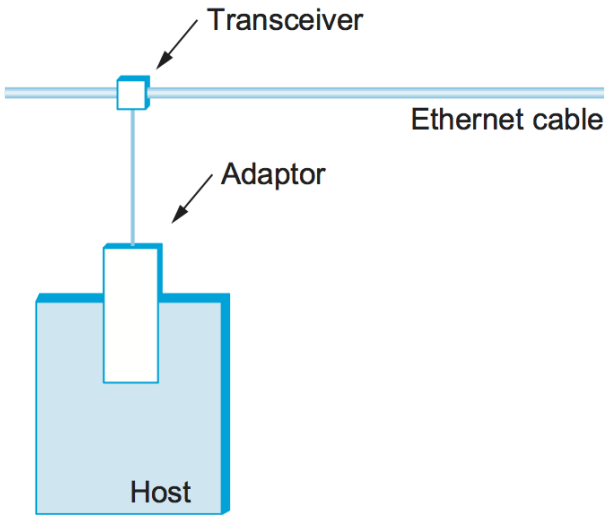


Figure 2.19.: Ethernet transceiver and adaptor.

Multiple Ethernet segments can be joined together by *repeaters* (or a multi-port variant of a repeater, called a *hub*). A repeater is a device that forwards digital signals, much like an amplifier forwards analog signals; repeaters do not understand bits or frames. No more than four repeaters could be positioned between any pair of hosts, meaning that a classical Ethernet had a total reach of only 2500 m. For example, using just two repeaters between any pair of hosts supports a configuration similar to the one illustrated in Figure 2.20; that is, a segment running down the spine of a building with a segment on each floor.

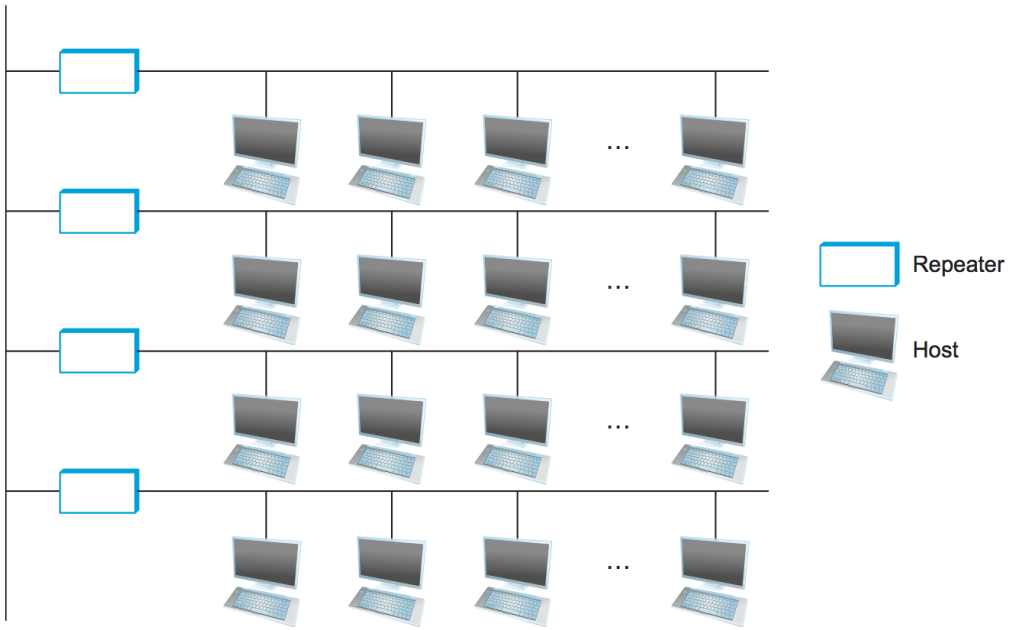


Figure 2.20.: Ethernet repeater, interconnecting segments to form a larger collision domain.

Any signal placed on the Ethernet by a host is broadcast over the entire network; that is, the signal is propagated in both directions, and repeaters and hubs forward the signal on all outgoing segments. Terminators attached to the end of each segment absorb the signal and keep it from bouncing back and interfering with

trailing signals. The original Ethernet specifications used the Manchester encoding scheme described in an earlier section, while 4B/5B encoding (or the similar 8B/10B) scheme is used today on higher speed Ethernets.

It is important to understand that whether a given Ethernet spans a single segment, a linear sequence of segments connected by repeaters, or multiple segments connected in a star configuration, data transmitted by any one host on that Ethernet reaches all the other hosts. This is the good news. The bad news is that all these hosts are competing for access to the same link, and, as a consequence, they are said to be in the same *collision domain*. The multi-access part of the Ethernet is all about dealing with the competition for the link that arises in a collision domain.

2.6.2 Access Protocol

We now turn our attention to the algorithm that controls access to a shared Ethernet link. This algorithm is commonly called the Ethernet's *media access control* (MAC). It is typically implemented in hardware on the network adaptor. We will not describe the hardware *per se*, but instead focus on the algorithm it implements. First, however, we describe the Ethernet's frame format and addresses.

Frame Format

Each Ethernet frame is defined by the format given in Figure 2.21. The 64-bit preamble allows the receiver to synchronize with the signal; it is a sequence of alternating 0s and 1s. Both the source and destination hosts are identified with a 48-bit address. The packet type field serves as the demultiplexing key; it identifies to which of possibly many higher-level protocols this frame should be delivered. Each frame contains up to 1500 bytes of data. Minimally, a frame must contain at least 46 bytes of data, even if this means the host has to pad the frame before transmitting it. The reason for this minimum frame size is that the frame must be long enough to detect a collision; we discuss this more below. Finally, each frame includes a 32-bit CRC. Like the HDLC protocol described in an earlier section, the Ethernet is a bit-oriented framing protocol. Note that from the host's perspective, an Ethernet frame has a 14-byte header: two 6-byte addresses and a 2-byte type field. The sending adaptor attaches the preamble and CRC before transmitting, and the receiving adaptor removes them.

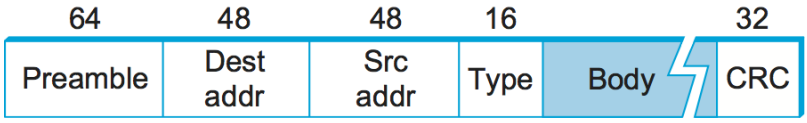


Figure 2.21.: Ethernet frame format.

Addresses

Each host on an Ethernet—in fact, every Ethernet host in the world—has a unique Ethernet address. Technically, the address belongs to the adaptor, not the host; it is usually burned into ROM. Ethernet addresses are typically printed in a form humans can read as a sequence of six numbers separated by colons. Each number corresponds to 1 byte of the 6-byte address and is given by a pair of hexadecimal digits, one for each of the 4-bit nibbles in the byte; leading 0s are dropped. For example, `8:0:2b:e4:b1:2` is the human-readable representation of Ethernet address

```
00001000  00000000  00101011  11100100  10110001  00000010
```

To ensure that every adaptor gets a unique address, each manufacturer of Ethernet devices is allocated a different prefix that must be prepended to the address on every adaptor they build. For example, Advanced Micro Devices has been assigned the 24-bit prefix `080020` (or `8:0:20`). A given manufacturer then makes sure the address suffixes it produces are unique.

Each frame transmitted on an Ethernet is received by every adaptor connected to that Ethernet. Each adaptor recognizes those frames addressed to its address and passes only those frames on to the host. (An adaptor can also be programmed to run in *promiscuous* mode, in which case it delivers all received frames to the host, but this is not the normal mode.) In addition to these *unicast* addresses, an Ethernet address consisting of all 1s is treated as a *broadcast* address; all adaptors pass frames addressed to the broadcast address up to the host. Similarly, an address that has the first bit set to 1 but is not the broadcast address is called a *multicast* address. A given host can program its adaptor to accept some set of multicast addresses. Multicast addresses are used to send messages to some subset of the hosts on an Ethernet (e.g., all file servers). To summarize, an Ethernet adaptor receives all frames and accepts

- Frames addressed to its own address
- Frames addressed to the broadcast address
- Frames addressed to a multicast address, if it has been instructed to listen to that address
- All frames, if it has been placed in promiscuous mode

It passes to the host only the frames that it accepts.

Transmitter Algorithm

As we have just seen, the receiver side of the Ethernet protocol is simple; the real smarts are implemented at the sender's side. The transmitter algorithm is defined as follows.

When the adaptor has a frame to send and the line is idle, it transmits the frame immediately; there is no negotiation with the other adaptors. The upper bound of 1500 bytes in the message means that the adaptor can occupy the line for only a fixed length of time.

When an adaptor has a frame to send and the line is busy, it waits for the line to go idle and then transmits immediately. (To be more precise, all adaptors wait 9.6 μs after the end of one frame before beginning to transmit the next frame. This is true for both the sender of the first frame as well as those nodes listening for the line to become idle.) The Ethernet is said to be a *1-persistent* protocol because an adaptor with a frame to send transmits with probability 1 whenever a busy line goes idle. In general, a *p-persistent* algorithm transmits with probability $0 \leq p \leq 1$ after a line becomes idle and defers with probability $q = 1 - p$. The reasoning behind choosing a $p<1$ is that there might be multiple adaptors waiting for the busy line to become idle, and we don't want all of them to begin transmitting at the same time. If each adaptor transmits immediately with a probability of, say, 33%, then up to three adaptors can be waiting to transmit and the odds are that only one will begin transmitting when the line becomes idle. Despite this reasoning, an Ethernet adaptor always transmits immediately after noticing that the network has become idle and has been very effective in doing so.

To complete the story about *p*-persistent protocols for the case when $p<1$, you might wonder how long a sender that loses the coin flip (i.e., decides to defer) has to wait before it can transmit. The answer for

the Aloha network, which originally developed this style of protocol, was to divide time into discrete slots, with each slot corresponding to the length of time it takes to transmit a full frame. Whenever a node has a frame to send and it senses an empty (idle) slot, it transmits with probability *p* and defers until the next slot with probability *q = 1 - p*. If that next slot is also empty, the node again decides to transmit or defer, with probabilities *p* and *q*, respectively. If that next slot is not empty—that is, some other station has decided to transmit—then the node simply waits for the next idle slot and the algorithm repeats.

Returning to our discussion of the Ethernet, because there is no centralized control it is possible for two (or more) adaptors to begin transmitting at the same time, either because both found the line to be idle or because both had been waiting for a busy line to become idle. When this happens, the two (or more) frames are said to *collide* on the network. Each sender, because the Ethernet supports collision detection, is able to determine that a collision is in progress. At the moment an adaptor detects that its frame is colliding with another, it first makes sure to transmit a 32-bit jamming sequence and then stops the transmission. Thus, a transmitter will minimally send 96 bits in the case of a collision: 64-bit preamble plus 32-bit jamming sequence.

One way that an adaptor will send only 96 bits—which is sometimes called a *runt frame*—is if the two hosts are close to each other. Had the two hosts been farther apart, they would have had to transmit longer, and thus send more bits, before detecting the collision. In fact, the worst-case scenario happens when the two hosts are at opposite ends of the Ethernet. To know for sure that the frame it just sent did not collide with another frame, the transmitter may need to send as many as 512 bits. Not coincidentally, every Ethernet frame must be at least 512 bits (64 bytes) long: 14 bytes of header plus 46 bytes of data plus 4 bytes of CRC.

Why 512 bits? The answer is related to another question you might ask about an Ethernet: Why is its length limited to only 2500 m? Why not 10 or 1000 km? The answer to both questions has to do with the fact that the farther apart two nodes are, the longer it takes for a frame sent by one to reach the other, and the network is vulnerable to a collision during this time.

Figure 2.22 illustrates the worst-case scenario, where hosts A and B are at opposite ends of the network. Suppose host A begins transmitting a frame at time t, as shown in (a). It takes it one link latency (let's denote the latency as d) for the frame to reach host B. Thus, the first bit of A's frame arrives at B at time t+d, as shown in (b). Suppose an instant before host A's frame arrives (i.e., B still sees an idle line), host B begins to transmit its own frame. B's frame will immediately collide with A's frame, and this collision will be detected by host B (c). Host B will send the 32-bit jamming sequence, as described above. (B's frame will be a runt.) Unfortunately, host A will not know that the collision occurred until B's frame reaches it, which will happen one link latency later, at time $t+2\times d$, as shown in (d). Host A must continue to transmit until this time in order to detect the collision. In other words, host A must transmit for $2\times d$ to be sure that it detects all possible collisions. Considering that a maximally configured Ethernet is 2500 m long, and that there may be up to four repeaters between any two hosts, the round-trip delay has been determined to be 51.2 μs, which on a 10-Mbps Ethernet corresponds to 512 bits. The other way to look at this situation is that we need to limit the Ethernet's maximum latency to a fairly small value (e.g., 51.2 μs) for the access algorithm to work; hence, an Ethernet's maximum length must be something on the order of 2500 m.

Once an adaptor has detected a collision and stopped its transmission, it waits a certain amount of time and tries again. Each time it tries to transmit but fails, the adaptor doubles the amount of time it waits before trying again. This strategy of doubling the delay interval between each retransmission attempt is a general technique known as *exponential backoff*. More precisely, the adaptor first delays either 0 or 51.2 μs, selected at random. If this effort fails, it then waits 0, 51.2, 102.4, or 153.6 μs (selected randomly) before trying again; this is k × 51.2 for k=0..3. After the third collision, it waits $k \times 51.2$ for k = $0..2^3$ - 1, again

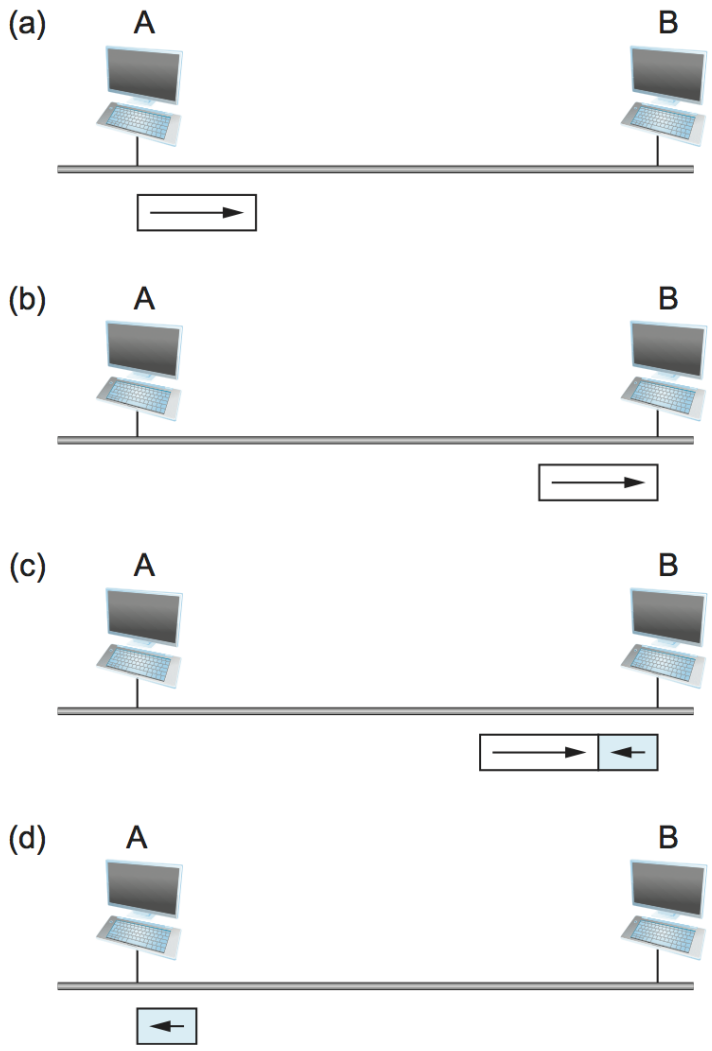


Figure 2.22.: Worst-case scenario: (a) A sends a frame at time t; (b) A's frame arrives at B at time t+d; (c) B begins transmitting at time t+d and collides with A's frame; (d) B's runt (32-bit) frame arrives at A at time t+2×d.

selected at random. In general, the algorithm randomly selects a *k* between 0 and 2^n - 1 and waits k × 51.2 μs, where *n* is the number of collisions experienced so far. The adaptor gives up after a given number of tries and reports a transmit error to the host. Adaptors typically retry up to 16 times, although the backoff algorithm caps *n* in the above formula at 10.

2.6.3 Longevity of Ethernet

Ethernet has been the dominant local area network technology for over 30 years. Today it is typically deployed point-to-point rather than tapping into a coax cable, it often runs at speeds of 1 or 10 Gbps rather than 10 Mbps, and it allows jumbo packets with up to 9000 bytes of data rather than 1500 bytes. But, it remains backwards compatible with the original standard. This makes it worth saying a few words about why Ethernets have been so successful, so that we can understand the properties we should emulate with any technology that tries to replace it.

First, an Ethernet is extremely easy to administer and maintain: There is no routing or configuration tables to be kept up-to-date, and it is easy to add a new host to the network. It is hard to imagine a simpler network to administer. Second, it is inexpensive: cable/fiber is relatively cheap, and the only other cost is the network adaptor on each host. Ethernet became deeply entrenched for these reasons, and any switch-based approach that aspired to displace it required additional investment in infrastructure (the switches), on top of the cost of each adaptor. The switch-based variant of Ethernet did eventually succeed in replacing multi-access Ethernet, but this is primarily because it could be *deployed incrementally*—with some hosts connected by point-to-point links to switches while others remained tapped into coax and connected to repeaters or hubs—all the while retaining the simplicity of network administration.

2.7 Wireless Networks

Wireless technologies differ from wired links in some important ways, while at the same time sharing many common properties. Like wired links, issues of bit errors are of great concern—typically even more so due to the unpredictable noise environment of most wireless links. Framing and reliability also have to be addressed. Unlike wired links, power is a big issue for wireless, especially because wireless links are often used by small mobile devices (like phones and sensors) that have limited access to power (e.g., a small battery). Furthermore, you can't go blasting away at arbitrarily high power with a radio transmitter—there are concerns about interference with other devices and usually regulations about how much power a device may emit at any given frequency.

Wireless media are also inherently multi-access; it's difficult to direct your radio transmission to just a single receiver or to avoid receiving radio signals from any transmitter with enough power in your neighborhood. Hence, media access control is a central issue for wireless links. And, because it's hard to control who receives your signal when you transmit over the air, issues of eavesdropping may also have to be addressed.

There is a baffling assortment of different wireless technologies, each of which makes different tradeoffs in various dimensions. One simple way to categorize the different technologies is by the data rates they provide and how far apart communicating nodes can be. Other important differences include which part of the electromagnetic spectrum they use (including whether it requires a license) and how much power they consume. In this section, we discuss two prominent wireless technologies: Wi-Fi (more formally known as 802.11), and Bluetooth. The next section discusses cellular networks in the context of ISP access services. Table 2.3 gives an overview of these technologies and how they compare to each other.

Table 2.3.: Overview of Leading Wireless Technologies.

	Bluetooth (802.15.1)	Wi-Fi (802.11)	4G Cellular
Typical link length	10 m	100 m	Tens of kilometers
Typical data rate	2 Mbps (shared)	150-450 Mbps	1-5 Mbps
Typical use	Link a peripheral to a computer	Link a computer to a wired base	Link mobile phone to a wired tower
Wired technology analogy	USB	Ethernet	PON

You may recall that bandwidth sometimes means the width of a frequency band in hertz and sometimes the data rate of a link. Because both these concepts come up in discussions of wireless networks, we're going to use *bandwidth* here in its stricter sense—width of a frequency band—and use the term *data rate* to describe the number of bits per second that can be sent over the link, as in Table 2.3.

2.7.1 Basic Issues

Because wireless links all share the same medium, the challenge is to share that medium efficiently, without unduly interfering with each other. Most of this sharing is accomplished by dividing it up along the dimensions of frequency and space. Exclusive use of a particular frequency in a particular geographic area may be allocated to an individual entity such as a corporation. It is feasible to limit the area covered by an electromagnetic signal because such signals weaken, or *attenuate*, with the distance from their origin. To reduce the area covered by your signal, reduce the power of your transmitter.

These allocations are typically determined by government agencies, such as the Federal Communications Commission (FCC) in the United States. Specific bands (frequency ranges) are allocated to certain uses. Some bands are reserved for government use. Other bands are reserved for uses such as AM radio, FM radio, television, satellite communication, and cellular phones. Specific frequencies within these bands are then licensed to individual organizations for use within certain geographical areas. Finally, several frequency bands are set aside for license-exempt usage—bands in which a license is not needed.

Devices that use license-exempt frequencies are still subject to certain restrictions to make that otherwise unconstrained sharing work. Most important of these is a limit on transmission power. This limits the range of a signal, making it less likely to interfere with another signal. For example, a cordless phone (a common unlicensed device) might have a range of about 100 feet.

One idea that shows up a lot when spectrum is shared among many devices and applications is *spread spectrum*. The idea behind spread spectrum is to spread the signal over a wider frequency band, so as to minimize the impact of interference from other devices. (Spread spectrum was originally designed for military use, so these "other devices" were often attempting to jam the signal.) For example, *frequency hopping* is a spread spectrum technique that involves transmitting the signal over a random sequence of frequencies; that is, first transmitting at one frequency, then a second, then a third, and so on. The sequence of frequencies is not truly random but is instead computed algorithmically by a pseudorandom number generator. The receiver uses the same algorithm as the sender and initializes it with the same seed; hence, it is able to hop frequencies in sync with the transmitter to correctly receive the frame. This scheme reduces interference by making it unlikely that two signals would be using the same frequency for more than the infrequent isolated bit.

A second spread spectrum technique, called *direct sequence*, adds redundancy for greater tolerance of interference. Each bit of data is represented by multiple bits in the transmitted signal so that, if some of the transmitted bits are damaged by interference, there is usually enough redundancy to recover the original bit. For each bit the sender wants to transmit, it actually sends the exclusive-OR of that bit and n random bits. As with frequency hopping, the sequence of random bits is generated by a pseudorandom number generator known to both the sender and the receiver. The transmitted values, known as an n-bit *chipping code*, spread the signal across a frequency band that is n times wider than the frame would have otherwise required. Figure 2.23 gives an example of a 4-bit chipping sequence.

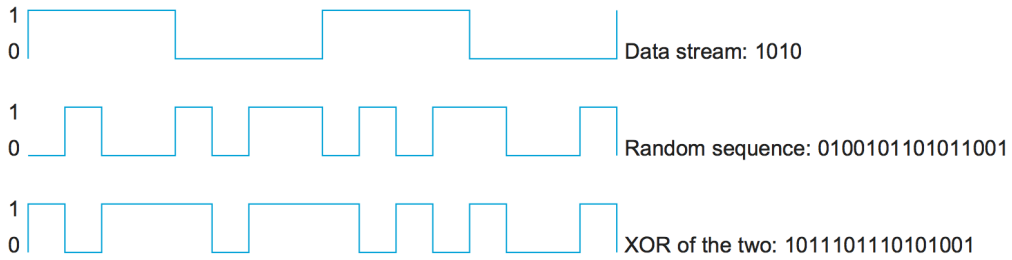


Figure 2.23.: Example 4-bit chipping sequence.

Different parts of the electromagnetic spectrum have different properties, making some better suited to communication, and some less so. For example, some can penetrate buildings and some cannot. Governments regulate only the prime communication portion: the radio and microwave ranges. As demand for prime spectrum increases, there is great interest in the spectrum that is becoming available as analog television is phased out in favor of digital.

In many wireless networks today we observe that there are two different classes of endpoints. One endpoint, sometimes described as the *base station*, usually has no mobility but has a wired (or at least high-bandwidth) connection to the Internet or other networks, as shown in Figure 2.24. The node at the other end of the link—shown here as a client node—is often mobile and relies on its link to the base station for all of its communication with other nodes.

Observe that in Figure 2.24 we have used a wavy pair of lines to represent the wireless "link" abstraction provided between two devices (e.g., between a base station and one of its client nodes). One of the interesting aspects of wireless communication is that it naturally supports point-to-multipoint communication, because radio waves sent by one device can be simultaneously received by many devices. However, it is often useful to create a point-to-point link abstraction for higher layer protocols, and we will see examples of how this works later in this section.

Note that in Figure 2.24 communication between non-base (client) nodes is routed via the base station. This is in spite of the fact that radio waves emitted by one client node may well be received by other client nodes—the common base station model does not permit direct communication between the client nodes.

This topology implies three qualitatively different levels of mobility. The first level is no mobility, such as when a receiver must be in a fixed location to receive a directional transmission from the base station. The second level is mobility within the range of a base, as is the case with Bluetooth. The third level is mobility between bases, as is the case with cell phones and Wi-Fi.

An alternative topology that is seeing increasing interest is the *mesh* or *ad hoc* network. In a wireless mesh, nodes are peers; that is, there is no special base station node. Messages may be forwarded via a chain of peer nodes as long as each node is within range of the preceding node. This is illustrated in Figure 2.25. This allows the wireless portion of a network to extend beyond the limited range of a single radio. From the point

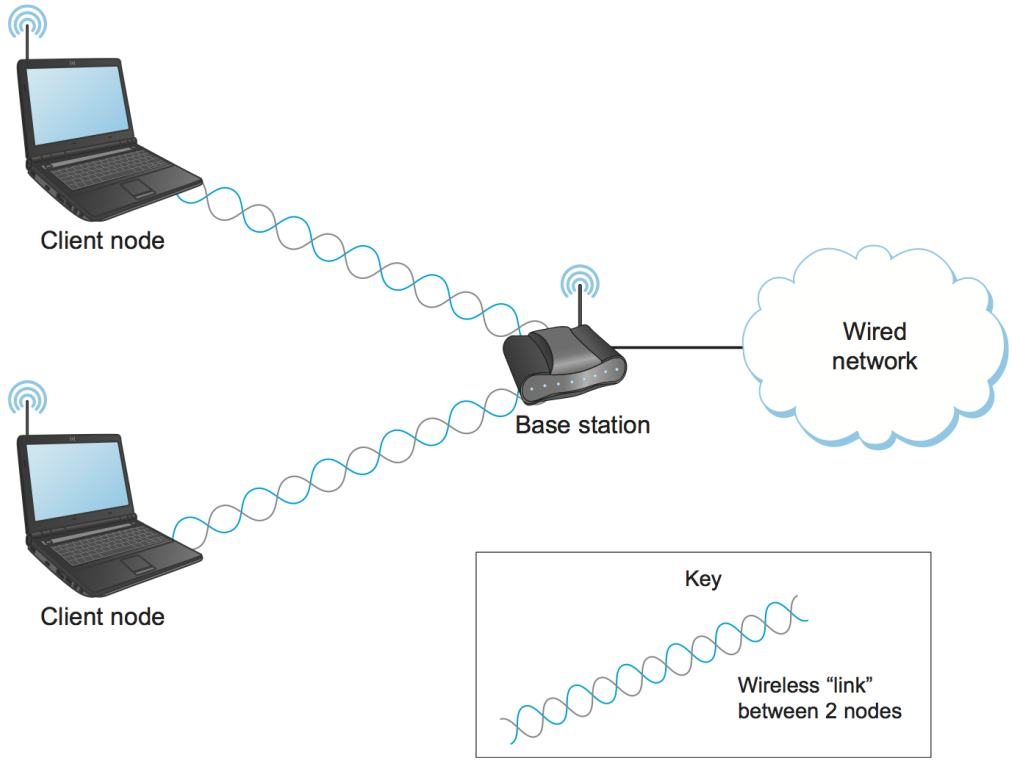


Figure 2.24.: A wireless network using a base station.

of view of competition between technologies, this allows a shorter-range technology to extend its range and potentially compete with a longer-range technology. Meshes also offer fault tolerance by providing multiple routes for a message to get from point A to point B. A mesh network can be extended incrementally, with incremental costs. On the other hand, a mesh network requires non-base nodes to have a certain level of sophistication in their hardware and software, potentially increasing per-unit costs and power consumption, a critical consideration for battery-powered devices. Wireless mesh networks are of considerable research interest, but they are still in their relative infancy compared to networks with base stations. Wireless sensor networks, another hot emerging technology, often form wireless meshes.

Now that we have covered some of the common wireless issues, let's take a look at the details of two common wireless technologies.

2.7.2 802.11/Wi-Fi

Most readers will have used a wireless network based on the IEEE 802.11 standards, often referred to as *Wi-Fi*. Wi-Fi is technically a trademark, owned by a trade group called the Wi-Fi Alliance, which certifies product compliance with 802.11. Like Ethernet, 802.11 is designed for use in a limited geographical area (homes, office buildings, campuses), and its primary challenge is to mediate access to a shared communication medium—in this case, signals propagating through space.

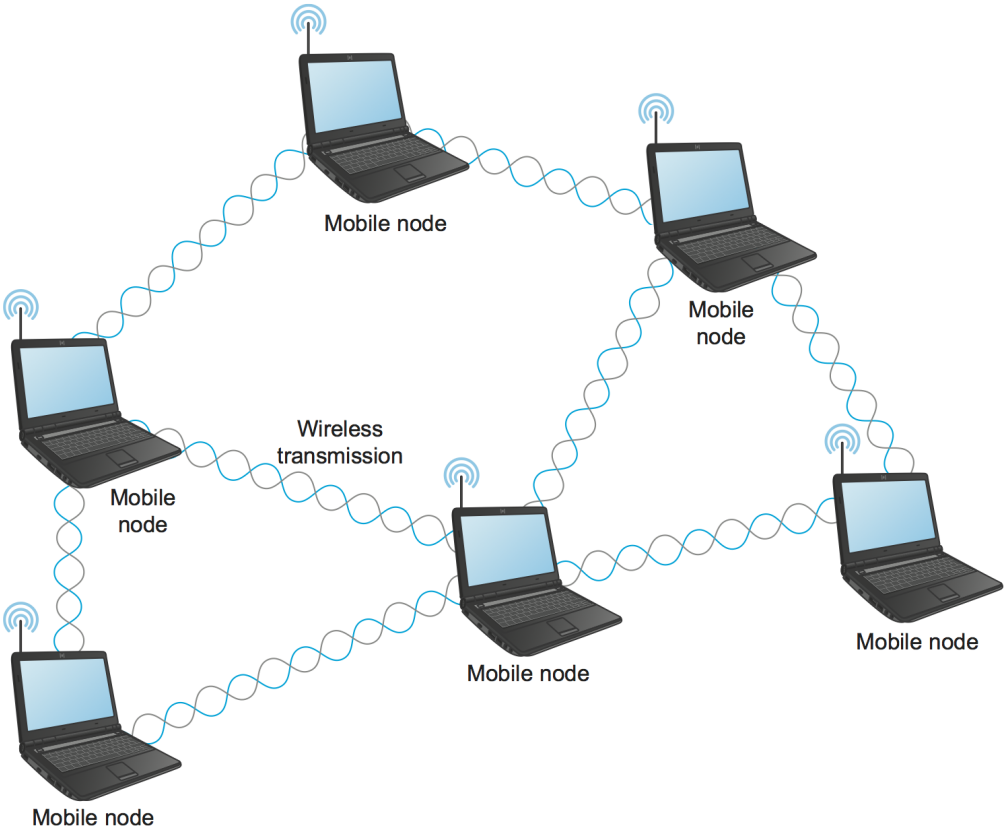


Figure 2.25.: A wireless ad hoc or mesh network.

Physical Properties

802.11 defines a number of different physical layers that operate in various frequency bands and provide a range of different data rates.

The original 802.11 standard defined two radio-based physical layers standards, one using frequency hopping (over 79 1-MHz-wide frequency bandwidths) and the other using direct sequence spread spectrum (with an 11-bit chipping sequence). Both provided data rates in the 2 Mbps range. Subsequently, the physical layer standard 802.11b was added, and using a variant of direct sequence, supported up to 11 Mbps. These three standards all operated in the license-exempt 2.4-GHz frequency band of the electromagnetic spectrum. Then came 802.11a, which delivered up to 54 Mbps using a variant of frequency division multiplexing called *orthogonal frequency division multiplexing (OFDM)*. 802.11a runs in the license-exempt 5-GHz band. 802.11g followed, and also using OFDM, delivered up to 54 Mbps. 802.11g is backward compatible with 802.11b (and returns to the 2.4-GHz band).

At the time of writing, many devices support 802.11n or 802.11ac, which typically achieve per-device data rates of 150 Mbps to 450 Mbps, respectively. This improvement is partly due to the use of multiple antennas and allowing greater wireless channel bandwidths. The use of multiple antennas is often called *MIMO* for multiple-input, multiple-output. The latest emerging standard, 802.11ax, promises another substantial improvement in throughput, in part by adopting many of the coding and modulation techniques used in the 4G/5G cellular network, which we describe in the next section.

It is common for commercial products to support more than one flavor of 802.11; many base stations support all five variants (a,b, g, n, and ac). This not only ensures compatibility with any device that supports any one of the standards but also makes it possible for two such products to choose the highest bandwidth option for a particular environment.

It is worth noting that while all the 802.11 standards define a *maximum* bit rate that can be supported, they mostly support lower bit rates as well (e.g., 802.11a allows for bit rates of 6, 9, 12, 18, 24, 36, 48, and 54 Mbps). At lower bit rates, it is easier to decode transmitted signals in the presence of noise. Different modulation schemes are used to achieve the various bit rates. In addition, the amount of redundant information in the form of error-correcting codes is varied. More redundant information means higher resilience to bit errors at the cost of lowering the effective data rate (since more of the transmitted bits are redundant).

The systems try to pick an optimal bit rate based on the noise environment in which they find themselves; the algorithms for bit rate selection can be quite complex. Interestingly, the 802.11 standards do not specify a particular approach but leave the algorithms to the various vendors. The basic approach to picking a bit rate is to estimate the bit error rate either by directly measuring the signal-to-noise ratio (SNR) at the physical layer or by estimating the SNR by measuring how often packets are successfully transmitted and acknowledged. In some approaches, a sender will occasionally probe a higher bit rate by sending one or more packets at that rate to see if it succeeds.

Collision Avoidance

At first glance, it might seem that a wireless protocol would follow the same algorithm as the Ethernet—wait until the link becomes idle before transmitting and back off should a collision occur—and, to a first approximation, this is what 802.11 does. The additional complication for wireless is that, while a node on an Ethernet receives every other node's transmissions and can transmit and receive at the same time, neither of these conditions holds for wireless nodes. This makes detection of collisions rather more complex. The

reason why wireless nodes cannot usually transmit and receive at the same time (on the same frequency) is that the power generated by the transmitter is much higher than any received is likely to be and so swamps the receiving circuitry. The reason why a node may not receive transmissions from another node is because that node may be too far away or blocked by an obstacle. This situation is a bit more complex than it first appears, as the following discussion will illustrate.

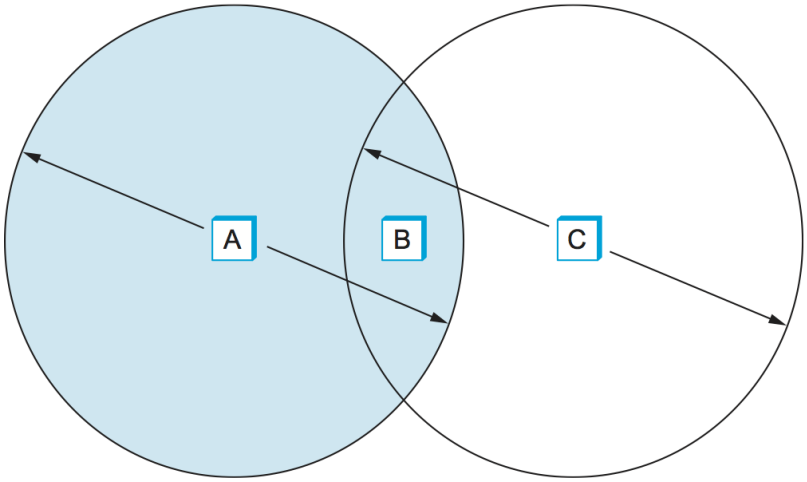


Figure 2.26.: The hidden node problem. Although A and C are hidden from each other, their signals can collide at B. (B's reach is not shown.)

Consider the situation depicted in Figure 2.26, where A and C are both within range of B but not each other. Suppose both A and C want to communicate with B and so they each send it a frame. A and C are unaware of each other since their signals do not carry that far. These two frames collide with each other at B, but unlike an Ethernet, neither A nor C is aware of this collision. A and C are said to be *hidden nodes* with respect to each other.

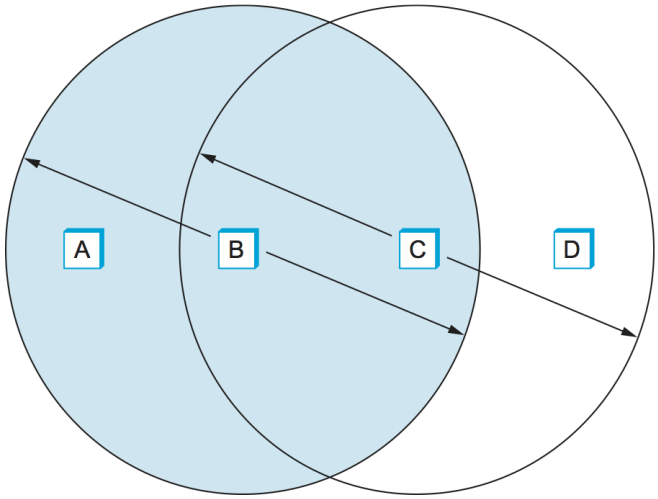


Figure 2.27.: The exposed node problem. Although B and C are exposed to each other's signals, there is no interference if B transmits to A while C transmits to D. (A and D's reaches are not shown.)

A related problem, called the *exposed node problem*, occurs under the circumstances illustrated in Figure 2.27, where each of the four nodes is able to send and receive signals that reach just the nodes to its imme-

diate left and right. For example, B can exchange frames with A and C but it cannot reach D, while C can reach B and D but not A. Suppose B is sending to A. Node C is aware of this communication because it hears B's transmission. It would be a mistake, however, for C to conclude that it cannot transmit to anyone just because it can hear B's transmission. For example, suppose C wants to transmit to node D. This is not a problem since C's transmission to D will not interfere with A's ability to receive from B. (It would interfere with A sending to B, but B is transmitting in our example.)

802.11 addresses these problems by using CSMA/CA, where the CA stands for collision *avoidance*, in contrast to the collision *detection* of CSMA/CD used on Ethernets. There are a few pieces to make this work.

The Carrier Sense part seems simple enough: Before sending a packet, the transmitter checks if it can hear any other transmissions; if not, it sends. However, because of the hidden node problem, just waiting for the absence of signals from other transmitters does not guarantee that a collision will not occur from the perspective of the receiver. For this reason, one part of CSMA/CA is an explicit ACK from the receiver to the sender. If the packet was successfully decoded and passed its CRC at the receiver, the receiver sends an ACK back to the sender.

Note that if a collision does occur, it will render the entire packet useless. For this reason, 802.11 adds an optional mechanism called RTS-CTS (Ready to Send-Clear to Send). This goes some way toward addressing the hidden node problem. The sender sends an RTS—a short packet—to the intended receiver, and if that packet is received successfully the receiver responds with another short packet, the CTS. Even though the RTS may not have been heard by a hidden node, the CTS probably will be. This effectively tells the nodes within range of the receiver that they should not send anything for a while—the amount of time of the intended transmission is included in the RTS and CTS packets. After that time plus a small interval has passed, the carrier can be assumed to be available again, and another node is free to try to send.

Of course, two nodes might detect an idle link and try to transmit an RTS frame at the same time, causing their RTS frames to collide with each other. The senders realize the collision has happened when they do not receive the CTS frame after a period of time, in which case they each wait a random amount of time before trying again. The amount of time a given node delays is defined by an exponential backoff algorithm very much like that used on the Ethernet.

After a successful RTS-CTS exchange, the sender sends its data packet and, if all goes well, receives an ACK for that packet. In the absence of a timely ACK, the sender will try again to request usage of the channel again, using the same process described above. By this time, of course, other nodes may again be trying to get access to the channel as well.

Distribution System

As described so far, 802.11 would be suitable for a network with a mesh (*ad hoc*) topology, and development of an 802.11s standard for mesh networks is nearing completion. At the current time, however, nearly all 802.11 networks use a base-station-oriented topology.

Instead of all nodes being created equal, some nodes are allowed to roam (e.g., your laptop) and some are connected to a wired network infrastructure. 802.11 calls these base stations *access points* (APs), and they are connected to each other by a so-called *distribution system*. Figure 2.28 illustrates a distribution system that connects three access points, each of which services the nodes in some region. Each access point operates on some channel in the appropriate frequency range, and each AP will typically be on a different channel than its neighbors.

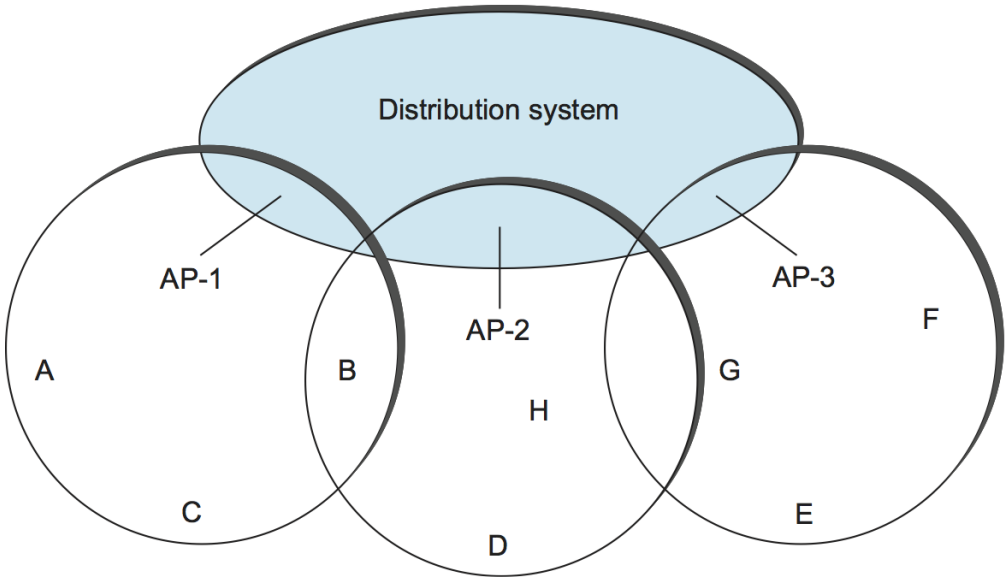


Figure 2.28.: Access points connected to a distribution system.

The details of the distribution system are not important to this discussion—it could be an Ethernet, for example. The only important point is that the distribution network operates at the link layer, the same protocol layer as the wireless links. In other words, it does not depend on any higher-level protocols (such as the network layer).

Although two nodes can communicate directly with each other if they are within reach of each other, the idea behind this configuration is that each node associates itself with one access point. For node A to communicate with node E, for example, A first sends a frame to its access point (AP-1), which forwards the frame across the distribution system to AP-3, which finally transmits the frame to E. How AP-1 knew to forward the message to AP-3 is beyond the scope of 802.11; it may have used a bridging protocol. What 802.11 does specify is how nodes select their access points and, more interestingly, how this algorithm works in light of nodes moving from one cell to another.

The technique for selecting an AP is called *scanning* and involves the following four steps:

1. The node sends a `Probe` frame.
2. All APs within reach reply with a `Probe Response` frame.
3. The node selects one of the access points and sends that AP an `Association Request` frame.
4. The AP replies with an `Association Response` frame.

A node engages this protocol whenever it joins the network, as well as when it becomes unhappy with its current AP. This might happen, for example, because the signal from its current AP has weakened due to the node moving away from it. Whenever a node acquires a new AP, the new AP notifies the old AP of the change (this happens in step 4) via the distribution system.

Consider the situation shown in Figure 2.29, where node C moves from the cell serviced by AP-1 to the cell serviced by AP-2. As it moves, it sends `Probe` frames, which eventually result in `Probe Response` frames from AP-2. At some point, C prefers AP-2 over AP-1, and so it associates itself with that access point.

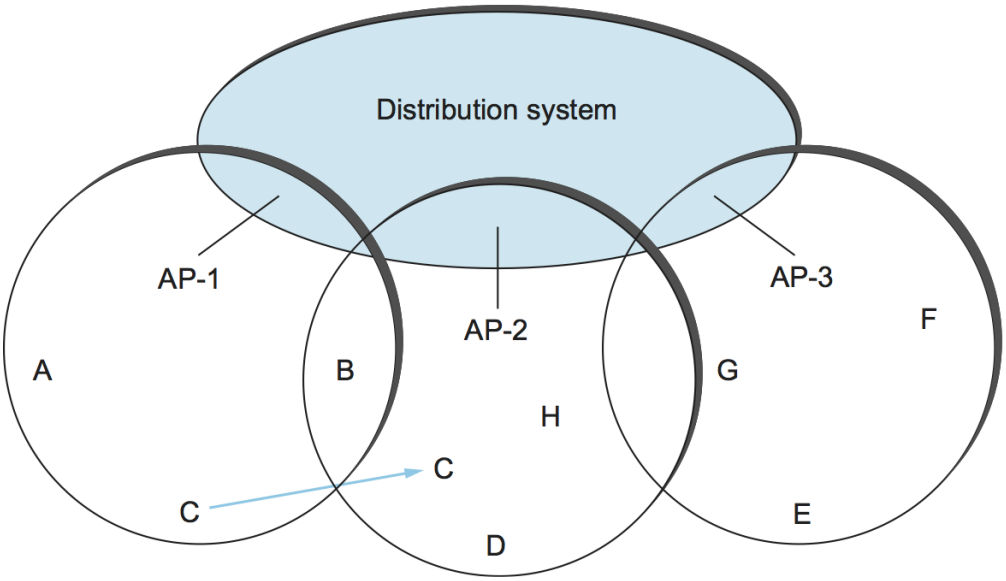


Figure 2.29.: Node mobility.

The mechanism just described is called *active scanning* since the node is actively searching for an access point. APs also periodically send a `Beacon` frame that advertises the capabilities of the access point; these include the transmission rates supported by the AP. This is called *passive scanning*, and a node can change to this AP based on the `Beacon` frame simply by sending an `Association Request` frame back to the access point.

Frame Format

Most of the 802.11 frame format, which is depicted in Figure 2.30, is exactly what we would expect. The frame contains the source and destination node addresses, each of which is 48 bits long; up to 2312 bytes of data; and a 32-bit CRC. The `Control` field contains three subfields of interest (not shown): a 6-bit `Type` field that indicates whether the frame carries data, is an RTS or CTS frame, or is being used by the scanning algorithm, and a pair of 1-bit fields—called `ToDS` and `FromDS`—that are described below.

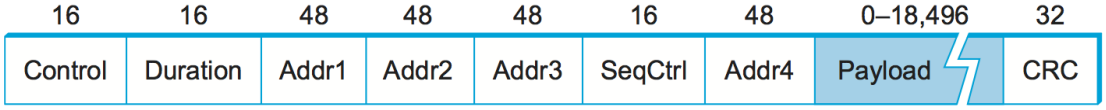


Figure 2.30.: 802.11 frame format.

The peculiar thing about the 802.11 frame format is that it contains four, rather than two, addresses. How these addresses are interpreted depends on the settings of the `ToDS` and `FromDS` bits in the frame's `Control` field. This is to account for the possibility that the frame had to be forwarded across the distribution system, which would mean that the original sender is not necessarily the same as the most recent transmitting node. Similar reasoning applies to the destination address. In the simplest case, when one node is sending directly to another, both the `DS` bits are 0, `Addr1` identifies the target node, and `Addr2` identifies the source node. In the most complex case, both `DS` bits are set to 1, indicating that the message went from a wireless node onto the distribution system, and then from the distribution system to another wireless node.

With both bits set, `Addr1` identifies the ultimate destination, `Addr2` identifies the immediate sender (the one that forwarded the frame from the distribution system to the ultimate destination), `Addr3` identifies the intermediate destination (the one that accepted the frame from a wireless node and forwarded it across the distribution system), and `Addr4` identifies the original source. In terms of the example given in Figure 2.28, `Addr1` corresponds to E, `Addr2` identifies AP-3, `Addr3` corresponds to AP-1, and `Addr4` identifies A.

Security of Wireless Links

One of the fairly obvious problems of wireless links compared to wires or fibers is that you can't be too sure where your data has gone. You can probably figure out if it was received by the intended receiver, but there is no telling how many other receivers might have also picked up your transmission. So, if you are concerned about the privacy of your data, wireless networks present a challenge.

Even if you are not concerned about data privacy—or perhaps have taken care of it in some other way—you may be concerned about an unauthorized user injecting data into your network. If nothing else, such a user might be able to consume resources that you would prefer to consume yourself, such as the finite bandwidth between your house and your ISP.

For these reasons, wireless networks typically come with some sort of mechanism to control access to both the link itself and the transmitted data. These mechanisms are often categorized as *wireless security.* The widely adopted WPA2 is described in Chapter 8.

2.7.3 Bluetooth (802.15.1)

Bluetooth fills the niche of very short range communication between mobile phones, PDAs, notebook computers, and other personal or peripheral devices. For example, Bluetooth can be used to connect a mobile phone to a headset or a notebook computer to a keyboard. Roughly speaking, Bluetooth is a more convenient alternative to connecting two devices with a wire. In such applications, it is not necessary to provide much range or bandwidth. This means that Bluetooth radios can use quite low power transmission, since transmission power is one of the main factors affecting bandwidth and range of wireless links. This matches the target applications for Bluetooth-enabled devices—most of them are battery powered (such as the ubiquitous phone headset) and hence it is important that they not consume much power.

Bluetooth operates in the license-exempt band at 2.45 GHz. Bluetooth links have typical bandwidths around 1 to 3 Mbps and a range of about 10 m. For this reason, and because the communicating devices typically belong to one individual or group, Bluetooth is sometimes categorized as a Personal Area Network (PAN).

Bluetooth is specified by an industry consortium called the *Bluetooth Special Interest Group*. It specifies an entire suite of protocols, going beyond the link layer to define application protocols, which it calls *profiles*, for a range of applications. For example, there is a profile for synchronizing a PDA with a personal computer. Another profile gives a mobile computer access to a wired LAN in the manner of 802.11, although this was not Bluetooth's original goal. The IEEE 802.15.1 standard is based on Bluetooth but excludes the application protocols.

The basic Bluetooth network configuration, called a *piconet*, consists of a master device and up to seven slave devices, as shown in Figure 2.31. Any communication is between the master and a slave; the slaves do not communicate directly with each other. Because slaves have a simpler role, their Bluetooth hardware and software can be simpler and cheaper.

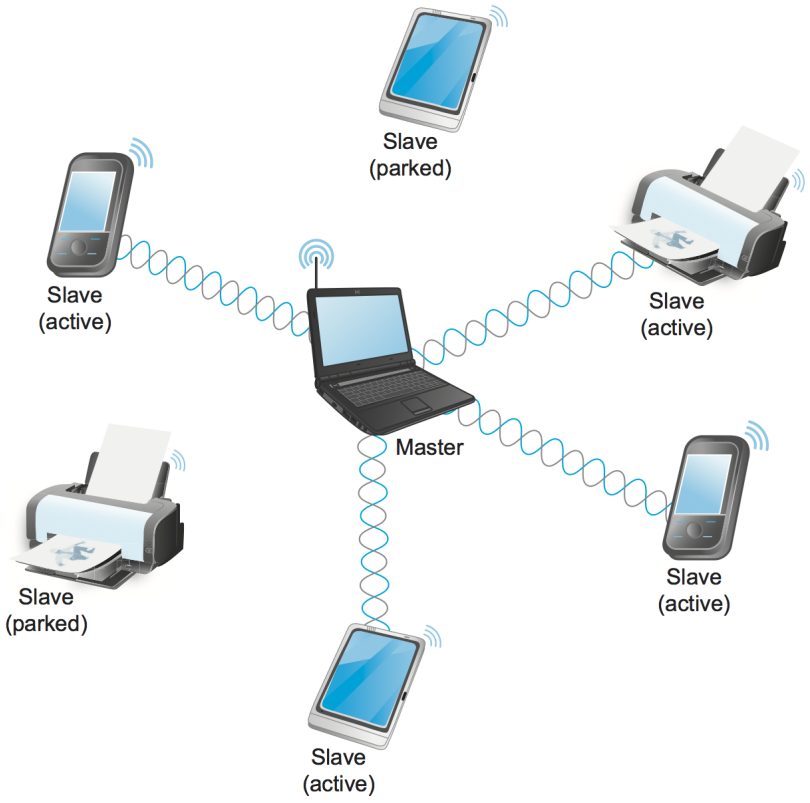


Figure 2.31.: A Bluetooth piconet.

Since Bluetooth operates in an license-exempt band, it is required to use a spread spectrum technique to deal with possible interference in the band. It uses frequency-hopping with 79 *channels* (frequencies), using each for 625 μs at a time. This provides a natural time slot for Bluetooth to use for synchronous time division multiplexing. A frame takes up 1, 3, or 5 consecutive time slots. Only the master can start to transmit in odd-numbered slots. A slave can start to transmit in an even-numbered slot—but only in response to a request from the master during the previous slot, thereby preventing any contention between the slave devices.

A slave device can be *parked*; that is, it is set to an inactive, low-power state. A parked device cannot communicate on the piconet; it can only be reactivated by the master. A piconet can have up to 255 parked devices in addition to its active slave devices.

In the realm of very low-power, short-range communication there are a few other technologies besides Bluetooth. One of these is ZigBee, devised by the ZigBee alliance and standardized as IEEE 802.15.4. It is designed for situations where the bandwidth requirements are low and power consumption must be very low to give very long battery life. It is also intended to be simpler and cheaper than Bluetooth, making it feasible to incorporate in cheaper devices such as *sensors*. Sensors are becoming an increasingly important class of networked device, as technology advances to the point where very cheap small devices can be deployed in large quantities to monitor things like temperature, humidity, and energy consumption in a building.

2.8 Access Networks

In addition to the Ethernet and Wi-Fi connections we typically use to connect to the Internet at home, at work, at school, and in many public spaces, most of us connect to the Internet over an *access* or *broadband* service that we buy from an ISP. This section describes two such technologies: *Passive Optical Networks* (PON), commonly referred to as fiber-to-the-home, and *Cellular Networks* that connect our mobile devices. In both cases, the networks are multi-access (like Ethernet and Wi-Fi), but as we will see, their approach to mediating access is quite different.

To set a little more context, ISPs (e.g., Telco or Cable companies) often operate a national backbone, and connected to the periphery of that backbone are hundreds or thousands of edge sites, each of which serves a city or neighborhood. These edge sites are commonly called *Central Offices* in the Telco world and *Head Ends* in the cable world, but despite their names implying "centralized" and "root of the hierarchy" these sites are at the very edge of the ISP's network; the ISP-side of the last-mile that directly connects to customers. PON and Cellular access networks are anchored in these facilities.[1]

2.8.1 Passive Optical Network

PON is the technology most commonly used to deliver fiber-based broadband to homes and businesses. PON adopts a point-to-multipoint design, which means the network is structured as a tree, with a single point starting in the ISP's network and then fanning out to reach up to 1024 homes. PON gets its name from the fact that the splitters are passive: they forward optical signals downstream and upstream without actively storing-and-forwarding frames. In this way, they are the optical variant of repeaters used in the classic Ethernet. Framing then happens at the source in the ISP's premises, in a device called an *Optical Line Terminal* (OLT), and at the end-points in individual homes, in a device called an *Optical Network Unit* (ONU).

Figure 2.32 shows an example PON, simplified to depict just one ONU and one OLT. In practice, a Central Office would include multiple OLTs connecting to thousands of customer homes. For completeness, Figure 2.32 also includes two other details about how the PON is connected to the ISP's backbone (and hence, to the rest of the Internet). The *Agg Switch* aggregates traffic from a set of OLTs, and the *BNG* (Broadband Network Gateway) is a piece of Telco equipment that, among many other things, meters Internet traffic for the sake of billing. As its name implies, the BNG is effectively the gateway between the access network (everything to the left of the BNG) and the Internet (everything to the right of the BNG).

Because the splitters are passive, PON has to implement some form of multi-access protocol. The approach it adopts can be summarized as follows. First, upstream and downstream traffic are transmitted on two different optical wavelengths, so they are completely independent of each other. Downstream traffic starts at the OLT and the signal is propagated down every link in the PON. As a consequence, every frame reaches every ONU. This device then looks at a unique identifier in the individual frames sent over the wavelength, and either keeps the frame (if the identifier is for it) or drops it (if not). Encryption is used to keep ONUs from eavesdropping on their neighbors' traffic.

Upstream traffic is then time-division multiplexed on the upstream wavelength, with each ONU periodically getting a turn to transmit. Because the ONUs are distributed over a fairly wide area (measured in kilometers) and at different distances from the OLT, it is not practical for them to transmit based on synchronized clocks,

[1] DSL is the legacy, copper-based counterpart to PON. DSL links are also terminated in Telco Central Offices, but we do not describe this technology since it is being phased out.

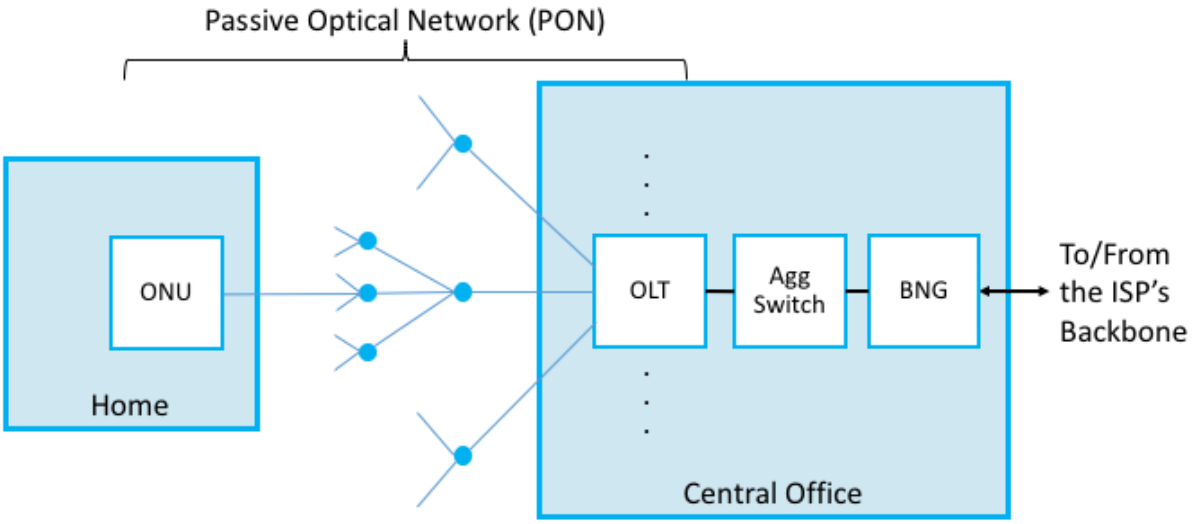


Figure 2.32.: An example PON that connects OLTs in the Central Office to ONUs in homes and businesses.

as in SONET. Instead, the ONT transmits *grants* to the individual ONUs, giving them a time interval during which they can transmit. In other words, the single OLT is responsible for centrally implementing the round-robin sharing of the shared PON. This includes the possibility that the OLT can grant each ONU a different share of time, effectively implementing different levels of service.

PON is similar to Ethernet in the sense that it defines a sharing algorithm that has evolved over time to accommodate higher and higher bandwidths. G-PON (Gigabit-PON) is the most widely deployed today, supporting a bandwidth of 2.25-Gbps. XGS-PON (10 Gigabit-PON) is just now starting to be deployed.

2.8.2 Cellular Network

While cellular telephone technology had its roots in analog voice communication, data services based on cellular standards are now the norm. Like Wi-Fi, cellular networks transmit data at certain bandwidths in the radio spectrum. Unlike Wi-Fi, which permits anyone to use a channel at either 2.4 or 5 GHz (all you have to do is set up a base station, as many of us do in our homes), exclusive use of various frequency bands have been auctioned off and licensed to service providers, who in turn sell mobile access service to their subscribers.

The frequency bands that are used for cellular networks vary around the world, and are complicated by the fact that ISPs often simultaneously support both old/legacy technologies and new/next-generation technologies, each of which occupies a different frequency band. The high-level summary is that traditional cellular technologies range from 700-MHz to 2400-MHz, with new mid-spectrum allocations now happening at 6-GHz and millimeter-wave (mmWave) allocations opening above 24-GHz.

Citizens Broadband Radio Service (CBRS)

In addition to the licensed bands, there is also an unlicensed band at 3.5-GHz set aside in North America, called *Citizens Broadband Radio Service* (CBRS), that anyone with a cellular radio can use. Similar

unlicensed bands are being set up in other countries, as well. This opens the door for setting up private cellular networks, for example, within a University campus, an enterprise, or a manufacturing plant.

To be more precise, the CBRS band allows three tiers of users to share the spectrum: first right of use goes to the original owners of this spectrum, naval radars and satellite ground stations; followed by priority users who receive this right over 10MHz bands for three years via regional auctions; and finally the rest of the population, who can access and utilize a portion of this band as long as they first check with a central database of registered users.

Like 802.11, cellular technology relies on the use of base stations that are connected to a wired network. In the case of the cellular network, the base stations are often called *Broadband Base Units* (BBU), the mobile devices that connect to them are usually referred to as *User Equipment* (UE), and the set of BBUs are anchored at an *Evolved Packet Core* (EPC) hosted in a Central Office. The wireless network served by the EPC is often called a *Radio Access Network* (RAN).

BBUs officially go by another name—Evolved NodeB, often abbreviated eNodeB or eNB—where NodeB is what the radio unit was called in an early incarnation of cellular networks (and has since evolved). Given that the cellular world continues to evolve at a rapid pace and eNB's are soon to be upgraded to gNB's, we have decided to use the more generic and less cryptic BBU.

Figure 2.33 depicts one possible configuration of the end-to-end scenario, with a few additional bits of detail. The EPC has multiple subcomponents, including an MME (Mobility Management Entity), an HSS (Home Subscriber Server), and an S/PGW (Session/Packet Gateway) pair; the first tracks and manages the movement of UEs throughout the RAN, the second is a database that contains subscriber-related information, and the Gateway pair processes and forwards packets between the RAN and the Internet (it forms the EPC's *user plane*). We say "one possible configuration" because the cellular standards allow wide variability in how many S/PGWs a given MME is responsible for, making is possible for a single MME to manage mobility across a wide geographic area that is served by multiple Central Offices. Finally, while not explicitly spelled out in Figure 2.33, it is sometimes the case that the ISP's PON network is used to connect the remote BBUs back to the Central Office.

The geographic area served by a BBU's antenna is called a *cell*. A BBU could serve a single cell or use multiple directional antennas to serve multiple cells. Cells don't have crisp boundaries, and they overlap. Where they overlap, an UE could potentially communicate with multiple BBUs. At any time, however, the UE is in communication with, and under the control of, just one BBU. As the device begins to leave a cell, it moves into an area of overlap with one or more other cells. The current BBU senses the weakening signal from the phone and gives control of the device to whichever base station is receiving the strongest signal from it. If the device is involved in a call or other network session at the time, the session must be transferred to the new base station in what is called a *handoff*. The decision making process for handoffs is under the purview of the MME, which has historically been a proprietary aspect of the cellular equipment vendors (although open source MME implementations are now starting to be available).

There have been multiple generations of protocols implementing the cellular network, colloquially known as 1G, 2G, 3G, and so on. The first two generations supported only voice, with 3G defining the transition to broadband access, supporting data rates measured in hundreds of kilobits-per-second. Today, the industry is at 4G (supporting data rates typically measured in the few megabits-per-second) and is in the process of transitioning to 5G (with the promise of a tenfold increase in data rates).

As of 3G, the generational designation actually corresponds to a standard defined by the 3GPP (3rd Generation Partnership Project). Even though its name has "3G" in it, the 3GPP continues to define the standard for

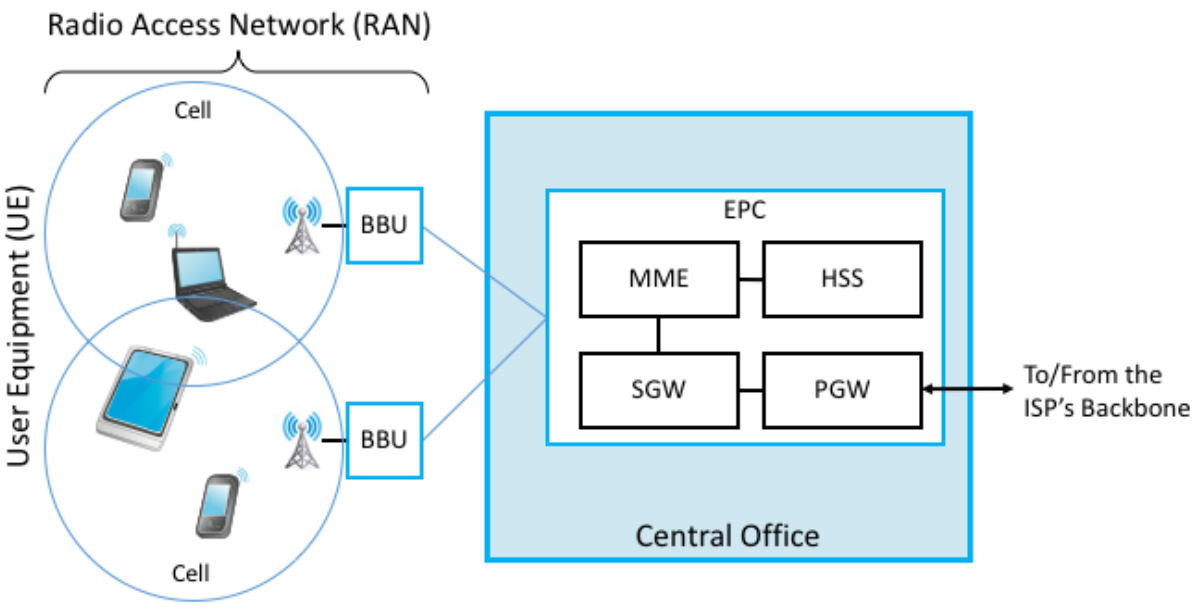


Figure 2.33.: A Radio Access Network (RAN) connecting a set of cellular devices (UEs) to an Evolved Packet Core (EPC) hosted in a Central Office.

4G and 5G, each of which corresponds to a release of the standard. Release 15, which is now published, is considered the demarcation point between 4G and 5G. By another name, this sequence of releases and generations is called LTE, which stands for *Long-Term Evolution*. The main takeaway is that while standards are published as a sequence of discrete releases, the industry as a whole has been on a fairly well-defined evolutionary path known as LTE. This section uses LTE terminology, but highlights the changes coming with 5G when appropriate.

The main innovation of LTE's air interface is how it allocates the available radio spectrum to UEs. Unlike Wi-Fi, which is contention-based, LTE uses a reservation-based strategy. This difference is rooted in each system's fundamental assumption about utilization: Wi-Fi assumes a lightly loaded network (and hence optimistically transmits when the wireless link is idle and backs off if contention is detected), while cellular networks assume (and strive for) high utilization (and hence explicitly assign different users to different "shares" of the available radio spectrum).

The state-of-the-art media access mechanism for LTE is called *Orthogonal Frequency-Division Multiple Access (OFDMA)*. The idea is to multiplex data over a set of 12 orthogonal subcarrier frequencies, each of which is modulated independently. The "Multiple Access" in OFDMA implies that data can simultaneously be sent on behalf of multiple users, each on a different subcarrier frequency and for a different duration of time. The subbands are narrow (e.g., 15kHz), but the coding of user data into OFDMA symbols is designed to minimize the risk of data loss due to interference between adjacent bands.

The use of OFDMA naturally leads to conceptualizing the radio spectrum as a two-dimensional resource, as shown in Figure 2.34. The minimal schedulable unit, called a *Resource Element (RE)*, corresponds to a 15kHz-wide band around one subcarrier frequency and the time it takes to transmit one OFDMA symbol. The number of bits that can be encoded in each symbol depends on the modulation rate, so for example using Quadrature Amplitude Modulation (QAM), 16-QAM yields 4 bits per symbol and 64-QAM yields 6 bits per symbol.

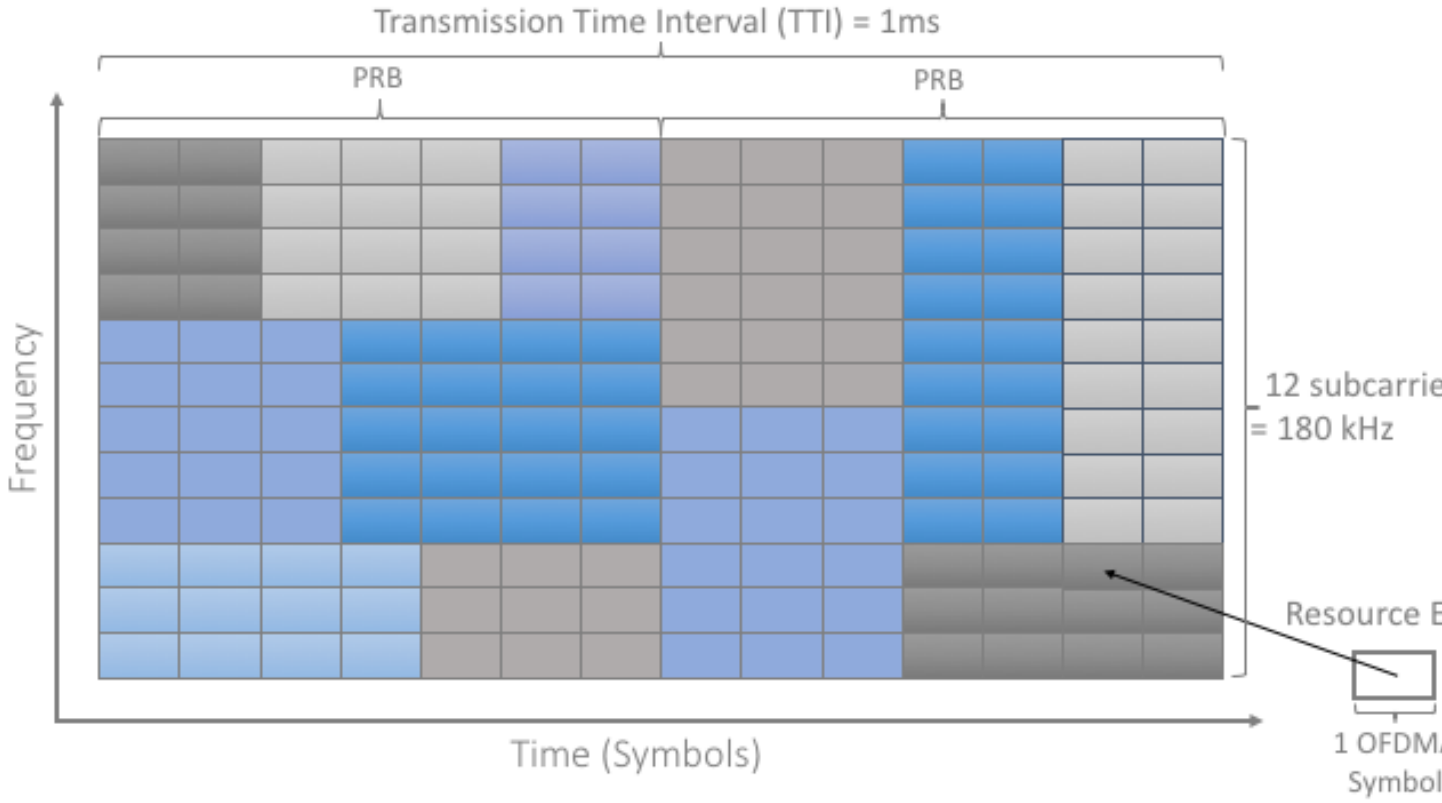


Figure 2.34.: The available radio spectrum abstractly represented by a 2-D grid of schedulable Resource Elements.

A scheduler makes allocation decisions at the granularity of blocks of 7x12=84 resource elements, called a *Physical Resource Block (PRB)*. Figure 2.34 shows two back-to-back PRBs, where UEs are depicted by different colored blocks. Of course time continues to flow along one axis, and depending on the size of the licensed frequency band, there may be many more subcarrier slots (and hence PRBs) available along the other axis, so the scheduler is essentially scheduling a sequence of PRBs for transmission.

The 1ms *Transmission Time Interval (TTI)* shown in Figure 2.34 corresponds to the time frame in which the BBU receives feedback from UEs about the quality of the signal they are experiencing. This feedback, called a *Channel Quality Indicator (CQI)*, essentially reports the observed signal-to-noise ratio, which impacts the UE's ability to recover the data bits. The base station then uses this information to adapt how it allocates the available radio spectrum to the UEs it is serving.

Up to this point, the description of how we schedule the radio spectrum is specific to 4G. The transition from 4G to 5G introduces additional degrees-of-freedom in how the radio spectrum is scheduled, making it possible to adapt the cellular network to a more diverse set of devices and applications domains.

Fundamentally, 5G defines a family of waveforms—unlike 4G, which specified only one waveform—each optimized for a different band in the radio spectrum.[2] The bands with carrier frequencies below 1GHz are designed to deliver mobile broadband and massive IoT services with a primary focus on range. Carrier frequencies between 1GHz-6GHz are designed to offer wider bandwidths, focusing on mobile broadband and mission-critical applications. Carrier frequencies above 24GHz (mmWaves) are designed to provide super wide bandwidths over short, line-of-sight coverage.

These different waveforms affect the scheduling and subcarrier intervals (i.e., the "size" of the Resource Elements just described).

- For sub-1GHz bands, 5G allows maximum 50MHz bandwidths. In this case, there are two waveforms: one with subcarrier spacing of 15kHz and another of 30kHz. (We used 15kHz in the example shown in Figure 2.34. The corresponding scheduling intervals are 0.5ms and 0.25ms, respectively. (We used 0.5ms in the example shown in Figure 2.34.)
- For 1GHz-6GHz bands, maximum bandwidths go up to 100MHz. Correspondingly, there are three waveforms with subcarrier spacings of 15kHz, 30kHz and 60kHz, corresponding to scheduling intervals of 0.5ms, 0.25ms and 0.125ms, respectively.
- For millimeter bands, bandwidths may go up to 400MHz. There are two waveforms, with subcarrier spacings of 60kHz and 120kHz. Both have scheduling intervals of 0.125ms.

This range of options is important because it adds another degree of freedom to the scheduler. In addition to allocating resource blocks to users, it has the ability to dynamically adjust the size of the resource blocks by changing the wave form being used in the band it is responsible for scheduling.

Whether 4G or 5G, the scheduling algorithm is a challenging optimization problem, with the objective of simultaneously (a) maximizing utilization of the available frequency band, and (b) ensuring that every UE receives the level of service it requires. This algorithm is not specified by 3GPP, but rather, is the proprietary intellectual property of the BBU vendors.

[2] A waveform is the frequency, amplitude, and phase-shift independent property (shape) of a signal. A sine wave is an example waveform.

Perspective: Race to the Edge

As we start to explore how softwarization is transforming the network, we should recognize that it is the access network that connects homes, businesses, and mobile users to the Internet that is undergoing the most radical change. The fiber-to-the-home and cellular networks described in *Section 2.8* are currently constructed from complex hardware appliances (e.g., OLTs, BNGs, BBUs, EPCs). Not only have these devices historically been closed and proprietary, but the vendors that sell them have typically bundled a broad and diverse collection of functionality in each. As a consequence, they have become expensive to build, complicated to operate, and slow to change.

In response, network operators are actively transitioning from these purpose-built appliances to open software running on commodity servers, switches, and access devices. This initiative is often called *CORD*, which is an acronym for **C**entral **O**ffice **R**e-architected as a **D**atacenter, and as the name suggests, the idea is to build the Telco Central Office (or the Cable Head End, resulting in the acronym *HERD*) using exactly the same technologies as in the large datacenters that make up the cloud.

The motivation for operators to do this is in part to benefit from the cost savings that come from replacing purpose-built appliances with commodity hardware, but it is mostly driven by the need to accelerate the pace of innovation. Their goal is to enable new classes of edge services—e.g., Public Safety, Autonomous Vehicles, Automated Factories, Internet-of-Things (IoT), Immersive User Interfaces—that benefit from low latency connectivity to end users, and more importantly, to the increasing number of devices those users surround themselves with. This results in a multi-tier cloud similar to the one shown in Figure 2.35.

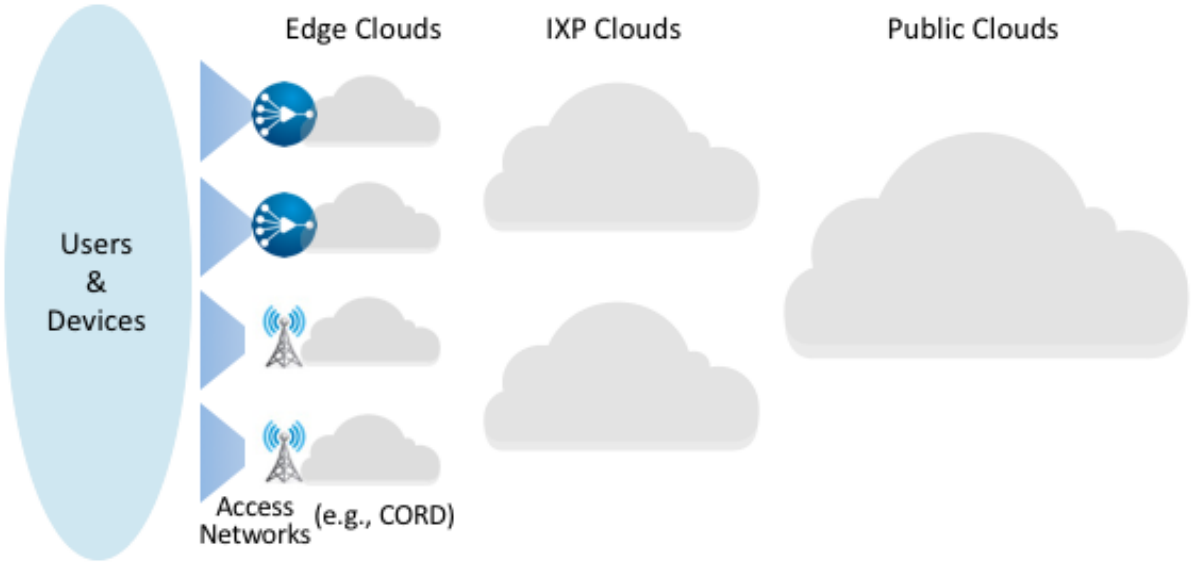


Figure 2.35.: Emerging multi-tier cloud includes datacenter-based public clouds, IXP-hosted distributed clouds, and access-based edge clouds, such as CORD. While there are on the order of 150 IXP-hosted clouds worldwide, we can expect there to be thousands or even tens of thousands of edge clouds.

This is all part of the growing trend to move functionality out of the datacenter and closer to the network edge, a trend that puts cloud providers and network operators on a collision course. Cloud providers, in pursuit of low-latency/high-bandwidth applications, are moving out of the datacenter and towards the edge at the same time network operators are adopting the best practices and technologies of the cloud to the edge that already exists and implements the access network. It's impossible to say how this will all play out over

time; both industries have their particular advantages.

On the one hand, cloud providers believe that by saturating metro areas with edge clusters and abstracting away the access network, they can build an edge presence with low enough latency and high enough bandwidth to serve the next generation of edge applications. In this scenario, the access network remains a dumb bit-pipe, allowing cloud providers to excel at what they do best: run scalable cloud services on commodity hardware.

On the other hand, network operators believe that by building the next generation access network using cloud technology, they will be able to co-locate edge applications in the access network. This scenario comes with built-in advantages: an existing and widely distributed physical footprint, existing operational support, and native support for both mobility and guaranteed service.

While acknowledging both of these possibilities, there is a third outcome that is not only worth considering, but also worth working towards: the *democratization of the network edge.* The idea is to make the access-edge cloud accessible to anyone, and not strictly the domain of incumbent cloud providers or network operators. There are three reasons to be optimistic about this possibility:

1. Hardware and software for the access network is becoming commoditized and open. This is a key enabler that we were just talking about. If it helps Telcos and CableCos be agile, then it can provide the same value to anyone.
2. There is demand. Enterprises in the automotive, factory, and warehouse space increasingly want to deploy private 5G networks for a variety of physical automation use cases (e.g., a garage where a remote valet parks your car or a factory floor making use of automation robots).
3. Spectrum is becoming available. 5G is opening up for use in an unlicensed or lightly licensed model in the US and Germany as two prime examples, with other countries soon to follow. This means 5G should have around 100-200 MHz of spectrum available for private use.

In short, the access network has historically been the purview of the Telcos, CableCos, and the vendors that sell them proprietary boxes, but the softwarization and virtualization of the access network opens the door for anyone (from smart cities to underserved rural areas to apartment complexes to manufacturing plants) to establish an access-edge cloud and connect it to the public Internet. We expect it to become as easy to do this as it is today to deploy a WiFi router. Doing so not only brings the access-edge into new (edgier) environments, but also has the potential to open the access network to developers that instinctively go where there are opportunities to innovate.

Broader Perspective

To continue reading about the cloudification of the Internet, see *Perspective: Virtual Networks All the Way Down*.

To learn more about the transformation taking place in access networks, we recommend: CORD: Central Office Re-architected as a Datacenter, IEEE Communications, October 2016 and Democratizing the Network Edge SIGCOMM CCR, April 2019.

CHAPTER

THREE

INTERNETWORKING

> Nature seems . . . to reach many of her ends by long circuitous routes.
>
> —*Rudolph Lotze*

Problem: Not All Networks are Directly Connected

As we have seen, there are many technologies that can be used to build last-mile links or to connect a modest number of nodes together, but how do we build networks of global scale? A single Ethernet can interconnect no more than 1024 hosts; a point-to-point link connects only two. Wireless networks are limited by the range of their radios. To build a global network, we need a way to interconnect these different types of links and multi-access networks. The concept of interconnecting different types of networks to build a large, global network is the core idea of the Internet and is often referred to as *internetworking*.

We can divide the internetworking problem up into a few subproblems. First of all, we need a way to interconnect links. Devices that interconnect links of the same type are often called *switches*, or sometimes *Layer 2* (L2) switches. These devices are the first topic of this chapter. A particularly important class of L2 switches in use today are those used to interconnect Ethernet segments. These switches are also sometimes called *bridges*.

The core job of a switch is to take packets that arrive on an input and *forward* (or *switch*) them to the right output so that they will reach their appropriate destination. There are a variety of ways that the switch can determine the "right" output for a packet, which can be broadly categorized as connectionless and connection-oriented approaches. These two approaches have both found important application areas over the years.

Given the enormous diversity of network types, we also need a way to interconnect disparate networks and links (i.e., deal with *heterogeneity*). Devices that perform this task, once called *gateways*, are now mostly known as *routers*, or alternatively, *Layer 3* (L3) switches. The protocol that was invented to deal with interconnection of disparate network types, the Internet Protocol (IP), is the topic of our second section.

Once we interconnect a whole lot of links and networks with switches and routers, there are likely to be many different possible ways to get from one point to another. Finding a suitable path or *route* through a network is one of the fundamental problems of networking. Such paths should be efficient (e.g., no longer than necessary), loop free, and able to respond to the fact that networks are not static—nodes may fail or reboot, links may break, and new nodes or links may be added. Our third section looks at some of the algorithms and protocols that have been developed to address these issues.

Once we understand the problems of switching and routing, we need some devices to perform those functions. This chapter concludes with some discussion of the ways switches and routers are implemented. While many packet switches and routers are quite similar to a general-purpose computer, there are many situations where more specialized designs are used. This is particularly the case at the high end, where there seems to be a never-ending need for more switching capacity that can handle the ever-increasing traffic load in the Internet's core.

3.1 Switching Basics

In the simplest terms, a switch is a mechanism that allows us to interconnect links to form a larger network. A switch is a multi-input, multi-output device that transfers packets from an input to one or more outputs. Thus, a switch adds the star topology (see Figure 3.1) to the set of possible network structures. A star topology has several attractive properties:

- Even though a switch has a fixed number of inputs and outputs, which limits the number of hosts that can be connected to a single switch, large networks can be built by interconnecting a number of switches.
- We can connect switches to each other and to hosts using point-to-point links, which typically means that we can build networks of large geographic scope.
- Adding a new host to the network by connecting it to a switch does not necessarily reduce the performance of the network for other hosts already connected.

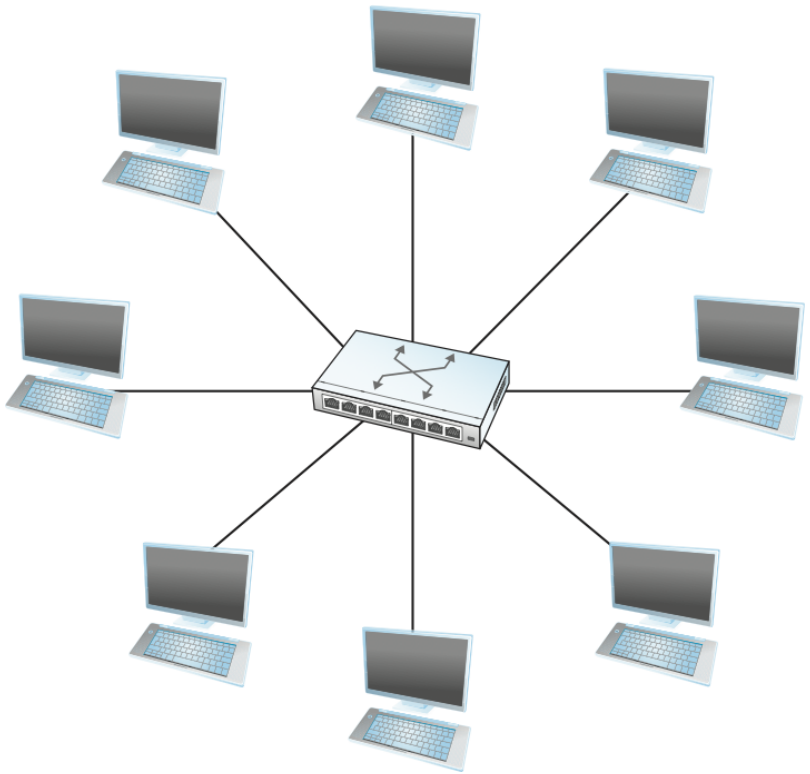


Figure 3.1.: A switch provides a star topology.

This last claim cannot be made for the shared-media networks discussed in the last chapter. For example, it is impossible for two hosts on the same 10-Mbps Ethernet segment to transmit continuously at 10 Mbps because they share the same transmission medium. Every host on a switched network has its own link to the switch, so it may be entirely possible for many hosts to transmit at the full link speed (bandwidth), provided that the switch is designed with enough aggregate capacity. Providing high aggregate throughput is one of the design goals for a switch; we return to this topic later. In general, switched networks are considered more *scalable* (i.e., more capable of growing to large numbers of nodes) than shared-media networks because of this ability to support many hosts at full speed.

Dense Wavelength Division Multiplexing

Our focus on packet-switched networks obsures the fact that, especially in wide-area networks, the underlying physical transport is all-optical: there are no packets. At this layer, commercially available DWDM (*Dense Wavelength Division Multiplexing*) equipment is able to transmit a large numbers of optical wavelengths (colors) down a single fiber. For example, one might send data on 100 or more different wavelengths, and each wavelength might carry as much as 100 Gbps of data.

Connecting these fibers is an optical device called a ROADM (*Reconfigurable Optical Add/Drop Multiplexers*). A collection of ROADMs (nodes) and fibers (links) form an optical transport network, where each ROADM is able to forward individual wavelengths along a multi-hop path, creating a logical end-to-end circuit. From the perspective of a packet-switched network that might be constructed on top of this optical transport, one wavelength, even it it crosses multiple ROADMs, appears to be a single point-to-point link between two switches, over which one might elect to run SONET or 100-Gbps Ethernet as the framing protocol. The reconfigurability feature of ROADMs means that it is possible to change these underlying end-to-end wavelengths, effectively creating a new topology at the packet-switching layer.

A switch is connected to a set of links and, for each of these links, runs the appropriate data link protocol to communicate with the node at the other end of the link. A switch's primary job is to receive incoming packets on one of its links and to transmit them on some other link. This function is sometimes referred to as either *switching* or *forwarding,* and in terms of the Open Systems Interconnection (OSI) architecture, it is the main function of the network layer, otherwise known as *Layer 2*.

The question, then, is how does the switch decide which output link to place each packet on? The general answer is that it looks at the header of the packet for an identifier that it uses to make the decision. The details of how it uses this identifier vary, but there are two common approaches. The first is the *datagram* or *connectionless* approach. The second is the *virtual circuit* or *connection-oriented* approach. A third approach, *source routing*, is less common than these other two, but it does have some useful applications.

One thing that is common to all networks is that we need to have a way to identify the end nodes. Such identifiers are usually called *addresses*. We have already seen examples of addresses, such as the 48-bit address used for Ethernet. The only requirement for Ethernet addresses is that no two nodes on a network have the same address. This is accomplished by making sure that all Ethernet cards are assigned a *globally unique* identifier. For the following discussion, we assume that each host has a globally unique address. Later on, we consider other useful properties that an address might have, but global uniqueness is adequate to get us started.

Another assumption that we need to make is that there is some way to identify the input and output ports of each switch. There are at least two sensible ways to identify ports: One is to number each port, and the other is to identify the port by the name of the node (switch or host) to which it leads. For now, we use numbering

of the ports.

3.1.1 Datagrams

The idea behind datagrams is incredibly simple: You just include in every packet enough information to enable any switch to decide how to get it to its destination. That is, every packet contains the complete destination address. Consider the example network illustrated in Figure 3.2, in which the hosts have addresses A, B, C, and so on. To decide how to forward a packet, a switch consults a *forwarding table* (sometimes called a *routing table*), an example of which is depicted in Table 3.1. This particular table shows the forwarding information that switch 2 needs to forward datagrams in the example network. It is pretty easy to figure out such a table when you have a complete map of a simple network like that depicted here; we could imagine a network operator configuring the tables statically. It is a lot harder to create the forwarding tables in large, complex networks with dynamically changing topologies and multiple paths between destinations. That harder problem is known as *routing* and is the topic of a later section. We can think of routing as a process that takes place in the background so that, when a data packet turns up, we will have the right information in the forwarding table to be able to forward, or switch, the packet.

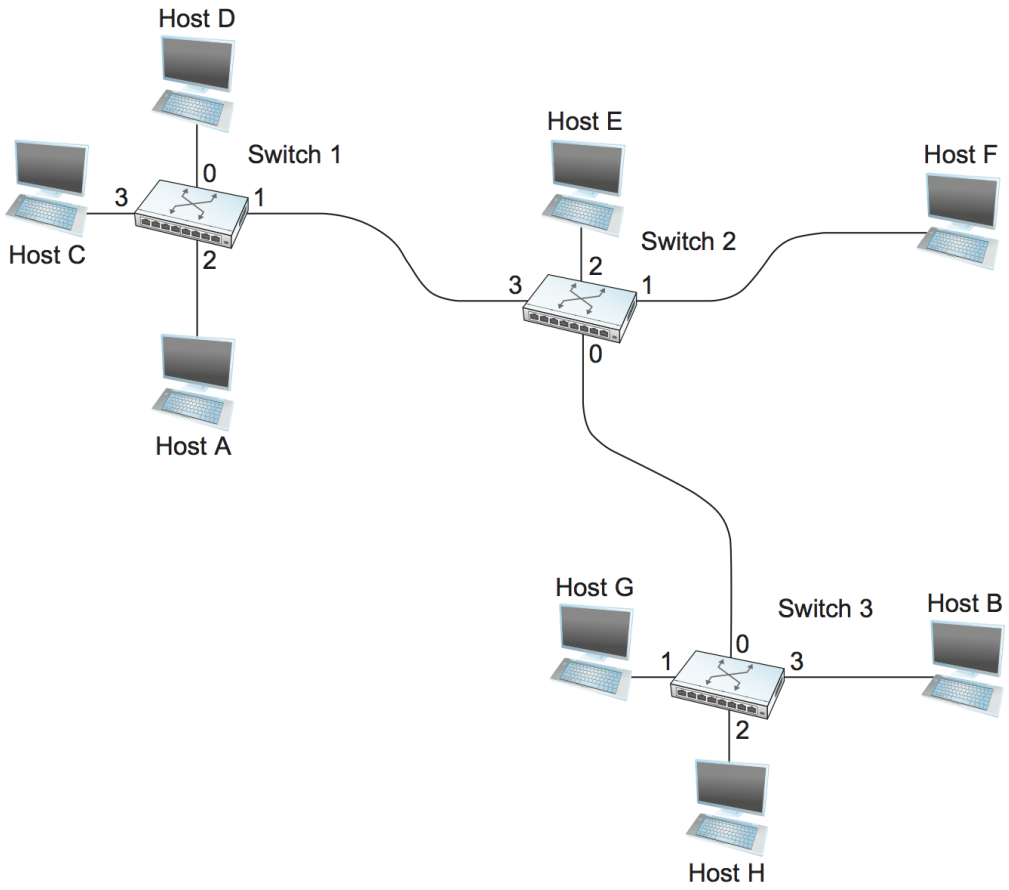


Figure 3.2.: Datagram forwarding: an example network.

Table 3.1.: Forwarding Table for Switch 2.

Destination	Port
A	3
B	0
C	3
D	3
E	2
F	1
G	0
H	0

Datagram networks have the following characteristics:

- A host can send a packet anywhere at any time, since any packet that turns up at a switch can be immediately forwarded (assuming a correctly populated forwarding table). For this reason, datagram networks are often called *connectionless*; this contrasts with the *connection-oriented* networks described below, in which some *connection state* needs to be established before the first data packet is sent.
- When a host sends a packet, it has no way of knowing if the network is capable of delivering it or if the destination host is even up and running.
- Each packet is forwarded independently of previous packets that might have been sent to the same destination. Thus, two successive packets from host A to host B may follow completely different paths (perhaps because of a change in the forwarding table at some switch in the network).
- A switch or link failure might not have any serious effect on communication if it is possible to find an alternate route around the failure and to update the forwarding table accordingly.

This last fact is particularly important to the history of datagram networks. One of the important design goals of the Internet is robustness to failures, and history has shown it to be quite effective at meeting this goal. Since datagram-based networks are the dominant technology discussed in this book, we postpone illustrative examples for the following sections, and move on to the two main alternatives.

3.1.2 Virtual Circuit Switching

A second technique for packet switching uses the concept of a *virtual circuit* (VC). This approach, which is also referred to as a *connection-oriented model*, requires setting up a virtual connection from the source host to the destination host before any data is sent. To understand how this works, consider Figure 3.3, where host A again wants to send packets to host B. We can think of this as a two-stage process. The first stage is "connection setup." The second is data transfer. We consider each in turn.

In the connection setup phase, it is necessary to establish a "connection state" in each of the switches between the source and destination hosts. The connection state for a single connection consists of an entry in a "VC table" in each switch through which the connection passes. One entry in the VC table on a single switch contains:

- A *virtual circuit identifier* (VCI) that uniquely identifies the connection at this switch and which will be carried inside the header of the packets that belong to this connection

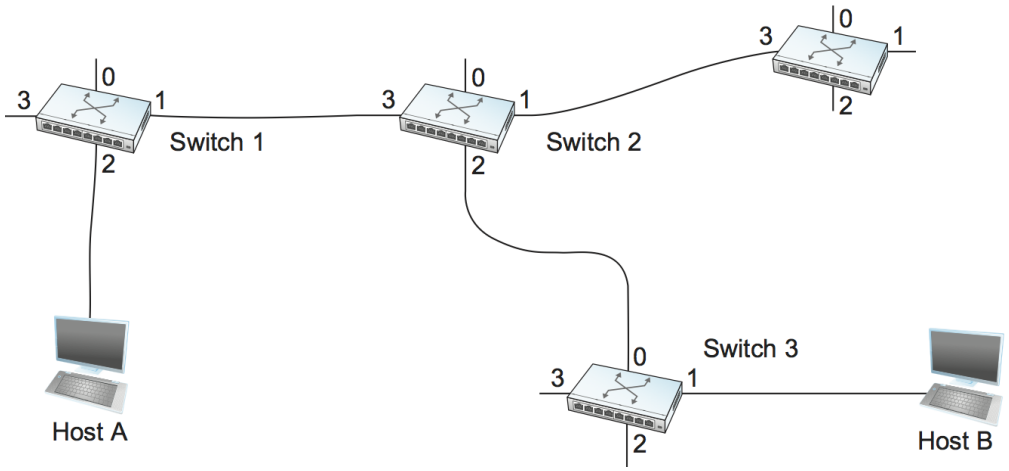


Figure 3.3.: An example of a virtual circuit network.

- An incoming interface on which packets for this VC arrive at the switch
- An outgoing interface in which packets for this VC leave the switch
- A potentially different VCI that will be used for outgoing packets

The semantics of one such entry is as follows: If a packet arrives on the designated incoming interface and that packet contains the designated VCI value in its header, then that packet should be sent out the specified outgoing interface with the specified outgoing VCI value having been first placed in its header.

Note that the combination of the VCI of packets as they are received at the switch *and* the interface on which they are received uniquely identifies the virtual connection. There may of course be many virtual connections established in the switch at one time. Also, we observe that the incoming and outgoing VCI values are generally not the same. Thus, the VCI is not a globally significant identifier for the connection; rather, it has significance only on a given link (i.e., it has *link-local scope*).

Whenever a new connection is created, we need to assign a new VCI for that connection on each link that the connection will traverse. We also need to ensure that the chosen VCI on a given link is not currently in use on that link by some existing connection.

There are two broad approaches to establishing connection state. One is to have a network administrator configure the state, in which case the virtual circuit is "permanent." Of course, it can also be deleted by the administrator, so a permanent virtual circuit (PVC) might best be thought of as a long-lived or administratively configured VC. Alternatively, a host can send messages into the network to cause the state to be established. This is referred to as *signalling*, and the resulting virtual circuits are said to be *switched*. The salient characteristic of a switched virtual circuit (SVC) is that a host may set up and delete such a VC dynamically without the involvement of a network administrator. Note that an SVC should more accurately be called a *signalled* VC, since it is the use of signalling (not switching) that distinguishes an SVC from a PVC.

Let's assume that a network administrator wants to manually create a new virtual connection from host A to host B. First, the administrator needs to identify a path through the network from A to B. In the example network of Figure 3.3, there is only one such path, but in general, this may not be the case. The administrator then picks a VCI value that is currently unused on each link for the connection. For the purposes of our example, let's suppose that the VCI value 5 is chosen for the link from host A to switch 1, and that 11 is

chosen for the link from switch 1 to switch 2. In that case, switch 1 needs to have an entry in its VC table configured as shown in Table 3.2.

Table 3.2.: Example Virtual Circuit Table Entry for Switch 1.

Incoming Interface	Incoming VCI	Outgoing Interface	Outgoing VCI
2	5	1	11

Similarly, suppose that the VCI of 7 is chosen to identify this connection on the link from switch 2 to switch 3 and that a VCI of 4 is chosen for the link from switch 3 to host B. In that case, switches 2 and 3 need to be configured with VC table entries as shown in Table 3.3 and Table 3.4, respectively. Note that the "outgoing" VCI value at one switch is the "incoming" VCI value at the next switch.

Table 3.3.: Virtual Circuit Table Entry at Switch 2.

Incoming Interface	Incoming VCI	Outgoing Interface	Outgoing VCI
3	11	2	7

Table 3.4.: Virtual Circuit Table Entry at Switch 3.

Incoming Interface	Incoming VCI	Outgoing Interface	Outgoing VCI
0	7	1	4

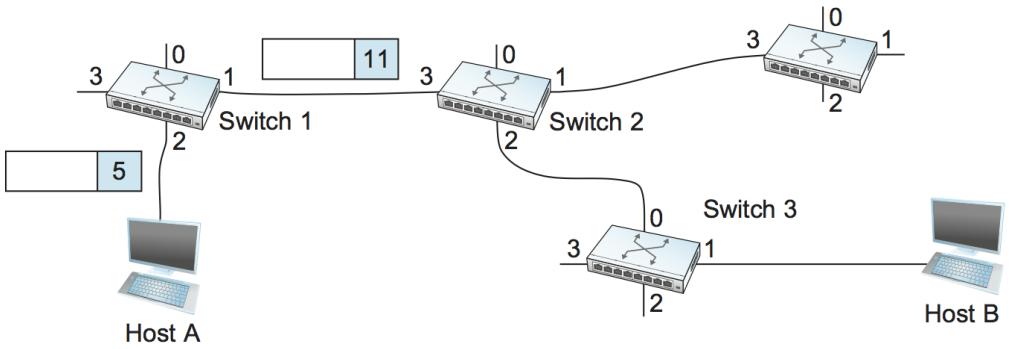


Figure 3.4.: A packet is sent into a virtual circuit network.

Once the VC tables have been set up, the data transfer phase can proceed, as illustrated in Figure 3.4. For any packet that it wants to send to host B, A puts the VCI value of 5 in the header of the packet and sends it to switch 1. Switch 1 receives any such packet on interface 2, and it uses the combination of the interface and the VCI in the packet header to find the appropriate VC table entry. As shown in Table 3.2, the table entry in this case tells switch 1 to forward the packet out of interface 1 and to put the VCI value 11 in the header when the packet is sent. Thus, the packet will arrive at switch 2 on interface 3 bearing VCI 11. Switch 2 looks up interface 3 and VCI 11 in its VC table (as shown in Table 3.3) and sends the packet on to switch 3 after updating the VCI value in the packet header appropriately, as shown in Figure 3.5. This process continues until it arrives at host B with the VCI value of 4 in the packet. To host B, this identifies the packet as having come from host A.

In real networks of reasonable size, the burden of configuring VC tables correctly in a large number of switches would quickly become excessive using the above procedures. Thus, either a network management

tool or some sort of signalling (or both) is almost always used, even when setting up "permanent" VCs. In the case of PVCs, signalling is initiated by the network administrator, while SVCs are usually set up using signalling by one of the hosts. We consider now how the same VC just described could be set up by signalling from the host.

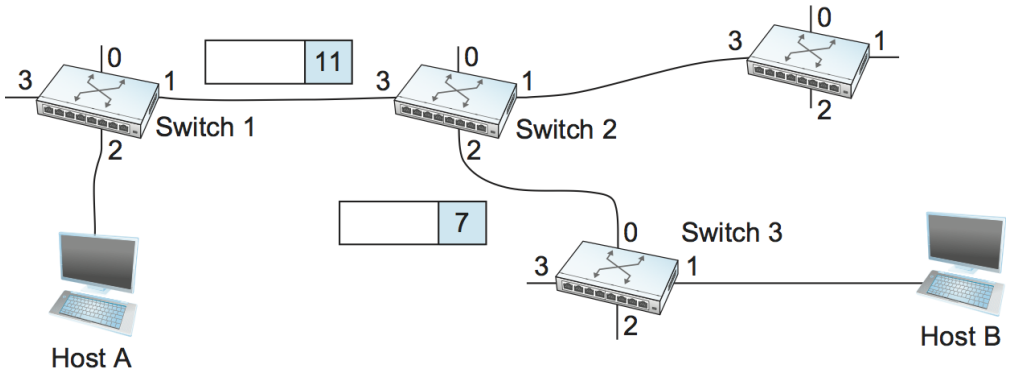


Figure 3.5.: A packet makes its way through a virtual circuit network.

To start the signalling process, host A sends a setup message into the network—that is, to switch 1. The setup message contains, among other things, the complete destination address of host B. The setup message needs to get all the way to B to create the necessary connection state in every switch along the way. We can see that getting the setup message to B is a lot like getting a datagram to B, in that the switches have to know which output to send the setup message to so that it eventually reaches B. For now, let's just assume that the switches know enough about the network topology to figure out how to do that, so that the setup message flows on to switches 2 and 3 before finally reaching host B.

When switch 1 receives the connection request, in addition to sending it on to switch 2, it creates a new entry in its virtual circuit table for this new connection. This entry is exactly the same as shown previously in Table 3.2. The main difference is that now the task of assigning an unused VCI value on the interface is performed by the switch for that port. In this example, the switch picks the value 5. The virtual circuit table now has the following information: "When packets arrive on port 2 with identifier 5, send them out on port 1." Another issue is that, somehow, host A will need to learn that it should put the VCI value of 5 in packets that it wants to send to B; we will see how that happens below.

When switch 2 receives the setup message, it performs a similar process; in this example, it picks the value 11 as the incoming VCI value. Similarly, switch 3 picks 7 as the value for its incoming VCI. Each switch can pick any number it likes, as long as that number is not currently in use for some other connection on that port of that switch. As noted above, VCIs have link-local scope; that is, they have no global significance.

Finally, the setup message arrives as host B. Assuming that B is healthy and willing to accept a connection from host A, it too allocates an incoming VCI value, in this case 4. This VCI value can be used by B to identify all packets coming from host A.

Now, to complete the connection, everyone needs to be told what their downstream neighbor is using as the VCI for this connection. Host B sends an acknowledgment of the connection setup to switch 3 and includes in that message the VCI that it chose (4). Now switch 3 can complete the virtual circuit table entry for this connection, since it knows the outgoing value must be 4. Switch 3 sends the acknowledgment on to switch 2, specifying a VCI of 7. Switch 2 sends the message on to switch 1, specifying a VCI of 11. Finally, switch 1 passes the acknowledgment on to host A, telling it to use the VCI of 5 for this connection.

At this point, everyone knows all that is necessary to allow traffic to flow from host A to host B. Each switch has a complete virtual circuit table entry for the connection. Furthermore, host A has a firm acknowledgment that everything is in place all the way to host B. At this point, the connection table entries are in place in all three switches just as in the administratively configured example above, but the whole process happened automatically in response to the signalling message sent from A. The data transfer phase can now begin and is identical to that used in the PVC case.

When host A no longer wants to send data to host B, it tears down the connection by sending a teardown message to switch 1. The switch removes the relevant entry from its table and forwards the message on to the other switches in the path, which similarly delete the appropriate table entries. At this point, if host A were to send a packet with a VCI of 5 to switch 1, it would be dropped as if the connection had never existed.

There are several things to note about virtual circuit switching:

- Since host A has to wait for the connection request to reach the far side of the network and return before it can send its first data packet, there is at least one round-trip time (RTT) of delay before data is sent.
- While the connection request contains the full address for host B (which might be quite large, being a global identifier on the network), each data packet contains only a small identifier, which is only unique on one link. Thus, the per-packet overhead caused by the header is reduced relative to the datagram model. More importantly, the lookup is fast because the virtual circuit number can be treated as an index into a table rather than as a key that has to be looked up.
- If a switch or a link in a connection fails, the connection is broken and a new one will need to be established. Also, the old one needs to be torn down to free up table storage space in the switches.
- The issue of how a switch decides which link to forward the connection request on has been glossed over. In essence, this is the same problem as building up the forwarding table for datagram forwarding, which requires some sort of *routing algorithm*. Routing is described in a later section, and the algorithms described there are generally applicable to routing setup requests as well as datagrams.

One of the nice aspects of virtual circuits is that by the time the host gets the go-ahead to send data, it knows quite a lot about the network—for example, that there really is a route to the receiver and that the receiver is willing and able to receive data. It is also possible to allocate resources to the virtual circuit at the time it is established. For example, X.25 (an early and now largely obsolete virtual-circuit-based networking technology) employed the following three-part strategy:

1. Buffers are allocated to each virtual circuit when the circuit is initialized.
2. The sliding window protocol is run between each pair of nodes along the virtual circuit, and this protocol is augmented with flow control to keep the sending node from over-running the buffers allocated at the receiving node.
3. The circuit is rejected by a given node if not enough buffers are available at that node when the connection request message is processed.

In doing these three things, each node is ensured of having the buffers it needs to queue the packets that arrive on that circuit. This basic strategy is usually called *hop-by-hop flow control.*

By comparison, a datagram network has no connection establishment phase, and each switch processes each packet independently, making it less obvious how a datagram network would allocate resources in a meaningful way. Instead, each arriving packet competes with all other packets for buffer space. If there are no free buffers, the incoming packet must be discarded. We observe, however, that even in a datagram-based

network a source host often sends a sequence of packets to the same destination host. It is possible for each switch to distinguish among the set of packets it currently has queued, based on the source/destination pair, and thus for the switch to ensure that the packets belonging to each source/destination pair are receiving a fair share of the switch's buffers.

In the virtual circuit model, we could imagine providing each circuit with a different *quality of service* (QoS). In this setting, the term *quality of service* is usually taken to mean that the network gives the user some kind of performance-related guarantee, which in turn implies that switches set aside the resources they need to meet this guarantee. For example, the switches along a given virtual circuit might allocate a percentage of each outgoing link's bandwidth to that circuit. As another example, a sequence of switches might ensure that packets belonging to a particular circuit not be delayed (queued) for more than a certain amount of time.

There have been a number of successful examples of virtual circuit technologies over the years, notably X.25, Frame Relay, and Asynchronous Transfer Mode (ATM). With the success of the Internet's connectionless model, however, none of them enjoys great popularity today. One of the most common applications of virtual circuits for many years was the construction of *virtual private networks* (VPNs), a subject discussed in a later section. Even that application is now mostly supported using Internet-based technologies today.

Asynchronous Transfer Mode (ATM)

Asynchronous Transfer Mode (ATM) is probably the most well-known virtual circuit-based networking technology, although it is now well past its peak in terms of deployment. ATM became an important technology in the 1980s and early 1990s for a variety of reasons, not the least of which is that it was embraced by the telephone industry, which at that point in time was less active in computer networks (other than as a supplier of links from which other people built networks). ATM also happened to be in the right place at the right time, as a high-speed switching technology that appeared on the scene just when shared media like Ethernet and token rings were starting to look a bit too slow for many users of computer networks. In some ways ATM was a competing technology with Ethernet switching, and it was seen by many as a competitor to IP as well.

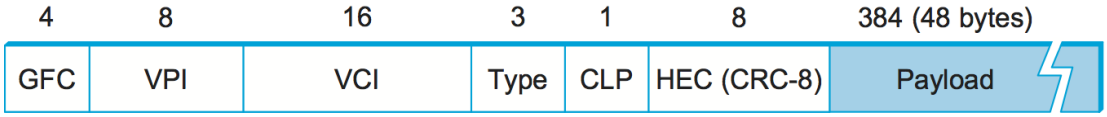


Figure 3.6.: ATM cell format at the UNI.

The approach ATM takes has some interesting properties, which makes it worth examining a bit further. The picture of the ATM packet format—more commonly called an ATM *cell*—in Figure 3.6 will illustrate the main points. We'll skip the generic flow control (GFC) bits, which never saw much use, and start with the 24 bits that are labelled VPI (virtual path identifier—8 bits) and VCI (virtual circuit identifier—16 bits). If you consider these bits together as a single 24-bit field, they correspond to the virtual circuit identifier introduced above. The reason for breaking the field into two parts was to allow for a level of hierarchy: All the circuits with the same VPI could, in some cases, be treated as a group (a virtual path) and could all be switched together looking only at the VPI, simplifying the work of a switch that could ignore all the VCI bits and reducing the size of the VC table considerably.

Skipping to the last header byte we find an 8-bit cyclic redundancy check (CRC), known as the *header error*

check (HEC). It uses CRC-8 and provides error detection and single-bit error correction capability on the cell header only. Protecting the cell header is particularly important because an error in the VCI will cause the cell to be misdelivered.

Probably the most significant thing to notice about the ATM cell, and the reason it is called a cell and not a packet, is that it comes in only one size: 53 bytes. What was the reason for this? One big reason was to facilitate the implementation of hardware switches. When ATM was being created in the mid- and late 1980s, 10-Mbps Ethernet was the cutting-edge technology in terms of speed. To go much faster, most people thought in terms of hardware. Also, in the telephone world, people think big when they think of switches—telephone switches often serve tens of thousands of customers. Fixed-length packets turn out to be a very helpful thing if you want to build fast, highly scalable switches. There are two main reasons for this:

1. It is easier to build hardware to do simple jobs, and the job of processing packets is simpler when you already know how long each one will be.
2. If all packets are the same length, then you can have lots of switching elements all doing much the same thing in parallel, each of them taking the same time to do its job.

This second reason, the enabling of parallelism, greatly improves the scalability of switch designs. It would be overstating the case to say that fast parallel hardware switches can only be built using fixed-length cells. However, it is certainly true that cells ease the task of building such hardware and that there was a lot of knowledge available about how to build cell switches in hardware at the time the ATM standards were being defined. As it turns out, this same principle is still applied in many switches and routers today, even if they deal in variable length packets—they cut those packets into some sort of cell in order to forward them from input port to output port, but this is all internal to the switch.

There is another good argument in favor of small ATM cells, having to do with end-to-end latency. ATM was designed to carry both voice phone calls (the dominant use case at the time) and data. Because voice is low-bandwidth but has strict delay requirements, the last thing you want is for a small voice packet queued behind a large data packet at a switch. If you force all packets to be small (i.e., cell-sized), then large data packets can still be supported by reassembling a set of cells into a packet, and you get the benefit of being able to interleave the forwarding of voice cells and data cells at every switch along the path from source to destination. This idea of using small cells to improve end-to-end latency is alive and well today in cellular access networks.

Having decided to use small, fixed-length packets, the next question was what is the right length to fix them at? If you make them too short, then the amount of header information that needs to be carried around relative to the amount of data that fits in one cell gets larger, so the percentage of link bandwidth that is actually used to carry data goes down. Even more seriously, if you build a device that processes cells at some maximum number of cells per second, then as cells get shorter the total data rate drops in direct proportion to cell size. An example of such a device might be a network adaptor that reassembles cells into larger units before handing them up to the host. The performance of such a device depends directly on cell size. On the other hand, if you make the cells too big, then there is a problem of wasted bandwidth caused by the need to pad transmitted data to fill a complete cell. If the cell payload size is 48 bytes and you want to send 1 byte, you'll need to send 47 bytes of padding. If this happens a lot, then the utilization of the link will be very low. The combination of relatively high header-to-payload ratio plus the frequency of sending partially filled cells did actually lead to some noticeable inefficiency in ATM networks that some detractors called the *cell tax*.

As it turns out, 48 bytes was picked for the ATM cell payload as a compromise. There were good arguments

for both larger and smaller cells, and 48 made almost no one happy—a power of two would certainly have been better for computers to process.

3.1.3 Source Routing

A third approach to switching that uses neither virtual circuits nor conventional datagrams is known as *source routing*. The name derives from the fact that all the information about network topology that is required to switch a packet across the network is provided by the source host.

There are various ways to implement source routing. One would be to assign a number to each output of each switch and to place that number in the header of the packet. The switching function is then very simple: For each packet that arrives on an input, the switch would read the port number in the header and transmit the packet on that output. However, since there will in general be more than one switch in the path between the sending and the receiving host, the header for the packet needs to contain enough information to allow every switch in the path to determine which output the packet needs to be placed on. One way to do this would be to put an ordered list of switch ports in the header and to rotate the list so that the next switch in the path is always at the front of the list. Figure 3.7 illustrates this idea.

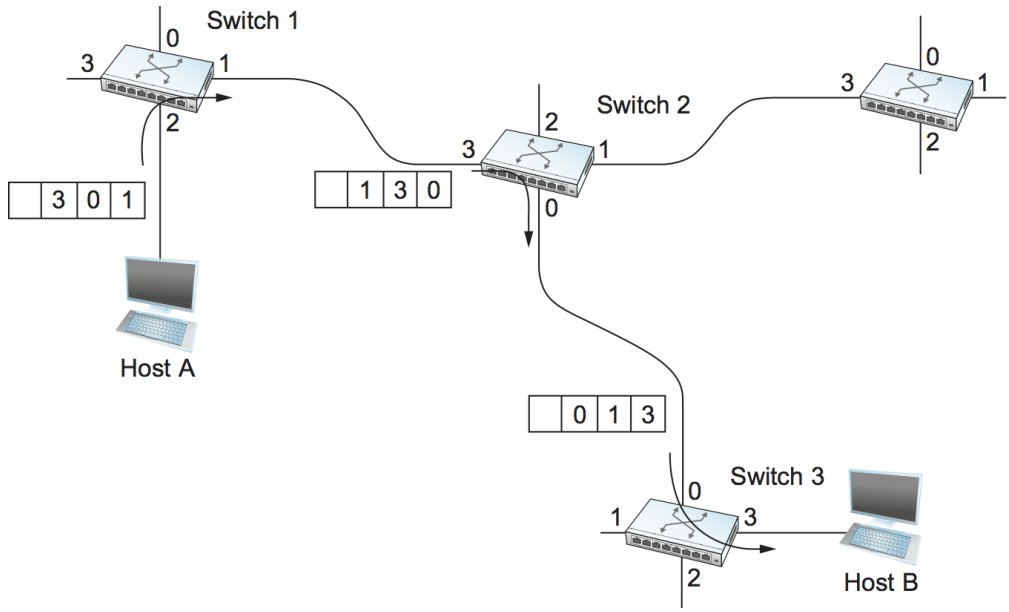


Figure 3.7.: Source routing in a switched network (where the switch reads the rightmost number).

In this example, the packet needs to traverse three switches to get from host A to host B. At switch 1, it needs to exit on port 1, at the next switch it needs to exit at port 0, and at the third switch it needs to exit at port 3. Thus, the original header when the packet leaves host A contains the list of ports (3, 0, 1), where we assume that each switch reads the rightmost element of the list. To make sure that the next switch gets the appropriate information, each switch rotates the list after it has read its own entry. Thus, the packet header as it leaves switch 1 enroute to switch 2 is now (1, 3, 0); switch 2 performs another rotation and sends out a packet with (0, 1, 3) in the header. Although not shown, switch 3 performs yet another rotation, restoring the header to what it was when host A sent it.

There are several things to note about this approach. First, it assumes that host A knows enough about the topology of the network to form a header that has all the right directions in it for every switch in the

path. This is somewhat analogous to the problem of building the forwarding tables in a datagram network or figuring out where to send a setup packet in a virtual circuit network. In practice, however, it is the first switch at the ingress to the network (as opposed to the end host connected to that switch) that appends the source route.

Second, observe that we cannot predict how big the header needs to be, since it must be able to hold one word of information for every switch on the path. This implies that headers are probably of variable length with no upper bound, unless we can predict with absolute certainty the maximum number of switches through which a packet will ever need to pass.

Third, there are some variations on this approach. For example, rather than rotate the header, each switch could just strip the first element as it uses it. Rotation has an advantage over stripping, however: Host B gets a copy of the complete header, which may help it figure out how to get back to host A. Yet another alternative is to have the header carry a pointer to the current "next port" entry, so that each switch just updates the pointer rather than rotating the header; this may be more efficient to implement. We show these three approaches in Figure 3.8. In each case, the entry that this switch needs to read is `A`, and the entry that the next switch needs to read is `B`.

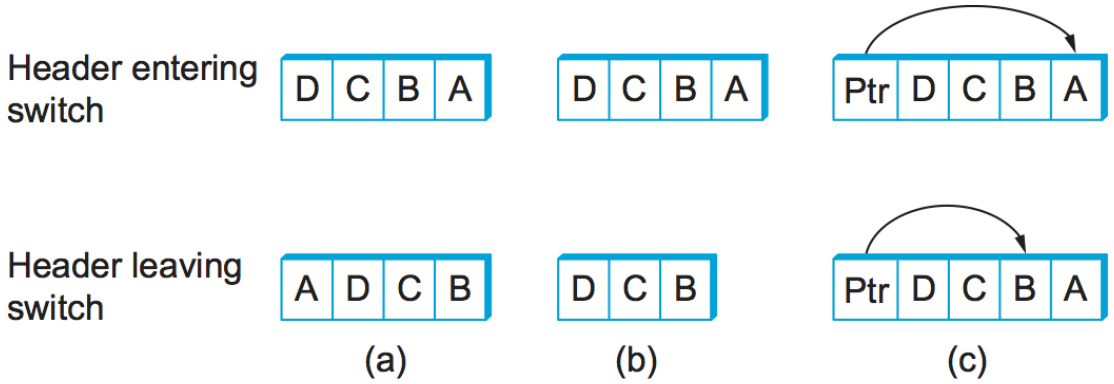


Figure 3.8.: Three ways to handle headers for source routing: (a) rotation; (b) stripping; (c) pointer. The labels are read right to left.

Source routing can be used in both datagram networks and virtual circuit networks. For example, the Internet Protocol, which is a datagram protocol, includes a source route option that allows selected packets to be source routed, while the majority are switched as conventional datagrams. Source routing is also used in some virtual circuit networks as the means to get the initial setup request along the path from source to destination.

Source routes are sometimes categorized as *strict* or *loose*. In a strict source route, every node along the path must be specified, whereas a loose source route only specifies a set of nodes to be traversed, without saying exactly how to get from one node to the next. A loose source route can be thought of as a set of waypoints rather than a completely specified route. The loose option can be helpful to limit the amount of information that a source must obtain to create a source route. In any reasonably large network, it is likely to be hard for a host to get the complete path information it needs to construct correct a strict source route to any destination. But both types of source routes do find application in certain scenarios, as we will see in later chapters.

3.2 Switched Ethernet

Having discussed some of the basic ideas behind switching, we now focus more closely on a specific switching technology: *Switched Ethernet*. The switches used to build such networks, which are often referred to as *L2 switches*, are widely used in campus and enterprise networks. Historically, they were more commonly referred to as *bridges* because they were used to "bridge" ethernet segments to build an *extended LAN*. But today most networks deploy Ethernet in a point-to-point configuration, with these links interconneted by L2 switches to form a switched Ethernet.

The following starts with the historical perspective (using bridges to connect a set of Ethernet segments), and then shifts to the perspective in wide-spread use today (using L2 switches to connect a set of point-to-point links). But whether we call the device a bridge or a switch—and the network you build an extended LAN or a switched Ethernet—the two behave in *exactly* the same way.

To begin, suppose you have a pair of Ethernets that you want to interconnect. One approach you might try is to put a repeater between them. This would not be a workable solution, however, if doing so exceeded the physical limitations of the Ethernet. (Recall that no more than two repeaters between any pair of hosts and no more than a total of 2500 m in length are allowed.) An alternative would be to put a node with a pair of Ethernet adaptors between the two Ethernets and have the node forward frames from one Ethernet to the other. This node would differ from a repeater, which operates on bits, not frames, and just blindly copies the bits received on one interface to another. Instead, this node would fully implement the Ethernet's collision detection and media access protocols on each interface. Hence, the length and number-of-host restrictions of the Ethernet, which are all about managing collisions, would not apply to the combined pair of Ethernets connected in this way. This device operates in promiscuous mode, accepting all frames transmitted on either of the Ethernets, and forwarding them to the other.

In their simplest variants, bridges simply accept LAN frames on their inputs and forward them out on all other outputs. This simple strategy was used by early bridges but has some pretty serious limitations as we'll see below. A number of refinements were added over the years to make bridges an effective mechanism for interconnecting a set of LANs. The rest of this section fills in the more interesting details.

3.2.1 Learning Bridges

The first optimization we can make to a bridge is to observe that it need not forward all frames that it receives. Consider the bridge in Figure 3.9. Whenever a frame from host A that is addressed to host B arrives on port 1, there is no need for the bridge to forward the frame out over port 2. The question, then, is how does a bridge come to learn on which port the various hosts reside?

One option would be to have a human download a table into the bridge similar to the one given in Table 3.5. Then, whenever the bridge receives a frame on port 1 that is addressed to host A, it would not forward the frame out on port 2; there would be no need because host A would have already directly received the frame on the LAN connected to port 1. Anytime a frame addressed to host A was received on port 2, the bridge would forward the frame out on port 1.

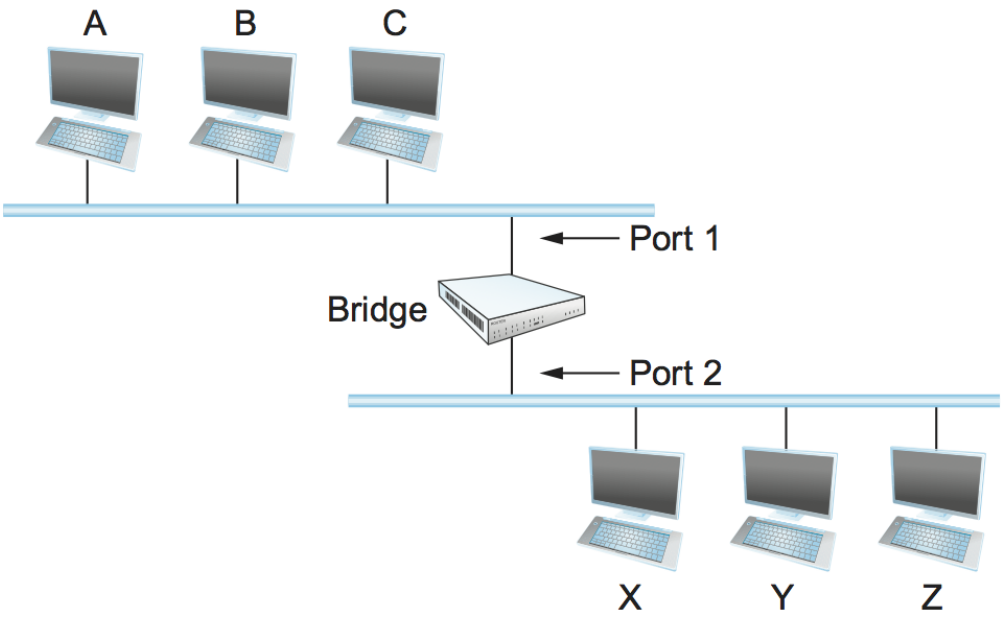


Figure 3.9.: Illustration of a learning bridge.

Table 3.5.: Forwarding Table Maintained by a Bridge.

Host	Port
A	1
B	1
C	1
X	2
Y	2
Z	2

Having a human maintain this table is too burdensome, and there is a simple trick by which a bridge can learn this information for itself. The idea is for each bridge to inspect the *source* address in all the frames it receives. Thus, when host A sends a frame to a host on either side of the bridge, the bridge receives this frame and records the fact that a frame from host A was just received on port 1. In this way, the bridge can build a table just like Table 3.5.

Note that a bridge using such a table implements a version of the datagram (or connectionless) model of forwarding described earlier. Each packet carries a global address, and the bridge decides which output to send a packet on by looking up that address in a table.

When a bridge first boots, this table is empty; entries are added over time. Also, a timeout is associated with each entry, and the bridge discards the entry after a specified period of time. This is to protect against the situation in which a host—and, as a consequence, its LAN address—is moved from one network to another. Thus, this table is not necessarily complete. Should the bridge receive a frame that is addressed to a host not currently in the table, it goes ahead and forwards the frame out on all the other ports. In other words, this table is simply an optimization that filters out some frames; it is not required for correctness.

3.2.2 Implementation

The code that implements the learning bridge algorithm is quite simple, and we sketch it here. Structure `BridgeEntry` defines a single entry in the bridge's forwarding table; these are stored in a `Map` structure (which supports `mapCreate`, `mapBind`, and `mapResolve` operations) to enable entries to be efficiently located when packets arrive from sources already in the table. The constant `MAX_TTL` specifies how long an entry is kept in the table before it is discarded.

```
#define BRIDGE_TAB_SIZE   1024  /* max size of bridging table */
#define MAX_TTL           120   /* time (in seconds) before an entry is␣
↪flushed */

typedef struct {
    MacAddr     destination;    /* MAC address of a node */
    int         ifnumber;       /* interface to reach it */
    u_short     TTL;            /* time to live */
    Binding     binding;        /* binding in the Map */
} BridgeEntry;

int     numEntries = 0;
Map     bridgeMap = mapCreate(BRIDGE_TAB_SIZE, sizeof(BridgeEntry));
```

The routine that updates the forwarding table when a new packet arrives is given by `updateTable`. The arguments passed are the source media access control (MAC) address contained in the packet and the interface number on which it was received. Another routine, not shown here, is invoked at regular intervals, scans the entries in the forwarding table, and decrements the `TTL` (time to live) field of each entry, discarding any entries whose `TTL` has reached 0. Note that the `TTL` is reset to `MAX_TTL` every time a packet arrives to refresh an existing table entry and that the interface on which the destination can be reached is updated to reflect the most recently received packet.

```
void
updateTable (MacAddr src, int inif)
{
    BridgeEntry       *b;

    if (mapResolve(bridgeMap, &src, (void **)&b) == FALSE )
    {
        /* this address is not in the table, so try to add it */
        if (numEntries < BRIDGE_TAB_SIZE)
        {
            b = NEW(BridgeEntry);
            b->binding = mapBind( bridgeMap, &src, b);
            /* use source address of packet as dest. address in table */
            b->destination = src;
            numEntries++;
        }
        else
        {
            /* can't fit this address in the table now, so give up */
            return;
        }
```

(continues on next page)

(continued from previous page)

```
    }
    /* reset TTL and use most recent input interface */
    b->TTL = MAX_TTL;
    b->ifnumber = inif;
}
```

Note that this implementation adopts a simple strategy in the case where the bridge table has become full to capacity—it simply fails to add the new address. Recall that completeness of the bridge table is not necessary for correct forwarding; it just optimizes performance. If there is some entry in the table that is not currently being used, it will eventually time out and be removed, creating space for a new entry. An alternative approach would be to invoke some sort of cache replacement algorithm on finding the table full; for example, we might locate and remove the entry with the smallest TTL to accommodate the new entry.

3.2.3 Spanning Tree Algorithm

The preceding strategy works just fine until the network has a loop in it, in which case it fails in a horrible way—frames potentially get forwarded forever. This is easy to see in the example depicted in Figure 3.10, where switches S1, S4, and S6 form a loop.

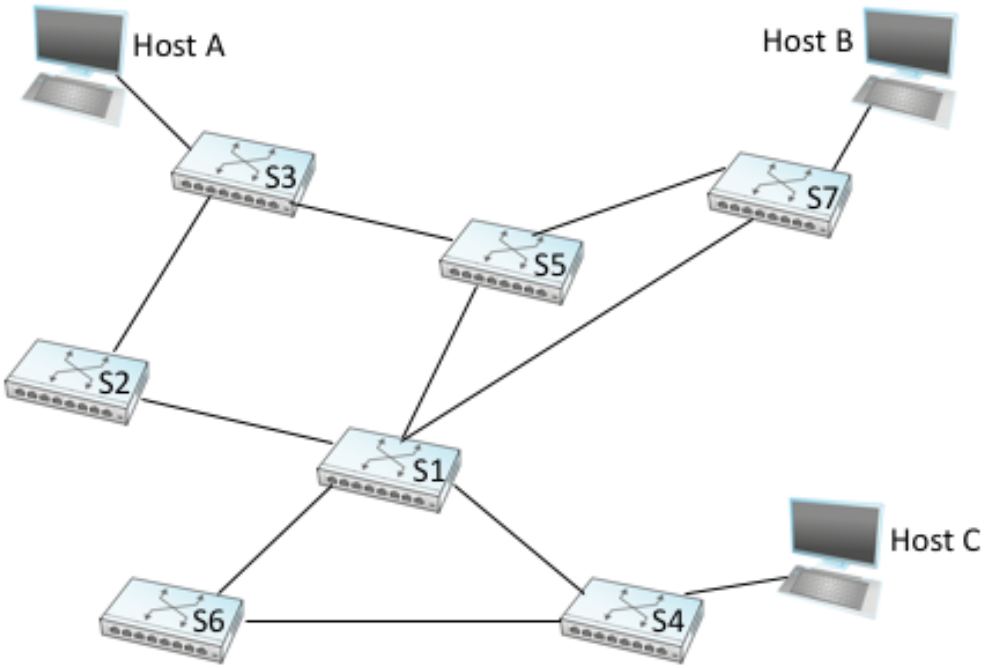


Figure 3.10.: Switched Ethernet with loops.

Note that we are now making the shift from calling the each forwarding device a bridge (connecting segments that might reach multiple other devices) to instead calling them L2 switches (connecting point-to-point links that reach just one other device). To keep the example managable, we include just three hosts. In practice, switches typically have 16, 24, or 48 ports, meaning they are able to connect to that many hosts (and other swiches).

In our example switched network, suppose that a packet enters switch S4 from Host C and that the destination address is one not yet in any switches's forwarding table: S4 sends a copy of the packet out its two other ports: to switches S1 and S6. Switch S6 forwards the packet onto S1 (and meanwhile, S1 forwards the packet onto S6), both of which in turn forward their packets back to S4. Switch S4 still doesn't have this destination in its table, so it forwards the packet out its two other ports. There is nothing to stop this cycle from repeating endlessly, with packets looping in both directions among S1, S4, and S6.

Why would a switched Ethernet (or extended LAN) come to have a loop in it? One possibility is that the network is managed by more than one administrator, for example, because it spans multiple departments in an organization. In such a setting, it is possible that no single person knows the entire configuration of the network, meaning that a switch that closes a loop might be added without anyone knowing. A second, more likely scenario is that loops are built into the network on purpose—to provide redundancy in case of failure. After all, a network with no loops needs only one link failure to become split into two separate partitions.

Whatever the cause, switches must be able to correctly handle loops. This problem is addressed by having the switches run a distributed *spanning tree* algorithm. If you think of the network as being represented by a graph that possibly has loops (cycles), then a spanning tree is a subgraph of this graph that covers (spans) all the vertices but contains no cycles. That is, a spanning tree keeps all of the vertices of the original graph but throws out some of the edges. For example, Figure 3.11 shows a cyclic graph on the left and one of possibly many spanning trees on the right.

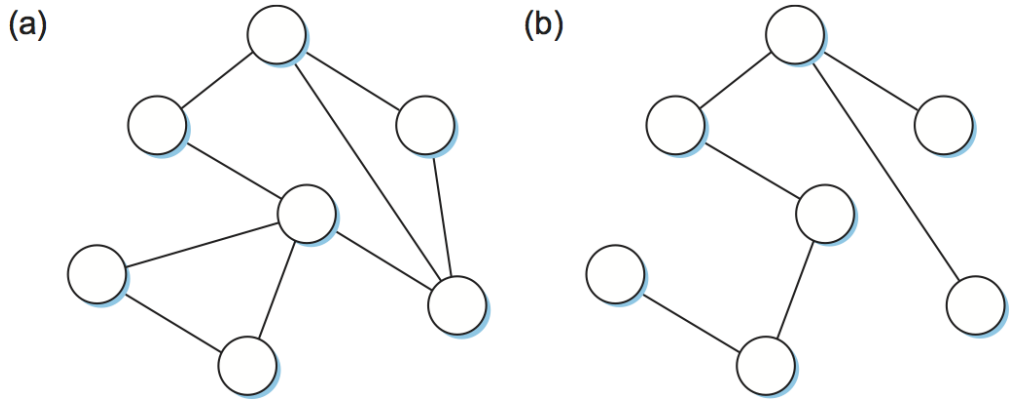


Figure 3.11.: Example of (a) a cyclic graph; (b) a corresponding spanning tree.

The idea of a spanning tree is simple enough: It's a subset of the actual network topology that has no loops and that reaches all the devices in the network. The hard part is how all of the switches coordinate their decisions to arrive at a single view of the spanning tree. After all, one topology is typically able to be covered by multiple spanning trees. The answer lies in the spanning tree protocol, which we'll describe now.

The spanning tree algorithm, which was developed by Radia Perlman, then at the Digital Equipment Corporation, is a protocol used by a set of switches to agree upon a spanning tree for a particular network. (The IEEE 802.1 specification is based on this algorithm.) In practice, this means that each switch decides the ports over which it is and is not willing to forward frames. In a sense, it is by removing ports from the topology that the network is reduced to an acyclic tree. It is even possible that an entire switch will not participate in forwarding frames, which seems kind of strange at first glance. The algorithm is dynamic, however, meaning that the switches are always prepared to reconfigure themselves into a new spanning tree should some switch fail, and so those unused ports and switches provide the redundant capacity needed to recover from failures.

The main idea of the spanning tree is for the switches to select the ports over which they will forward frames. The algorithm selects ports as follows. Each switch has a unique identifier; for our purposes, we use the labels S1, S2, S3, and so on. The algorithm first elects the switch with the smallest ID as the root of the spanning tree; exactly how this election takes place is described below. The root switch always forwards frames out over all of its ports. Next, each switch computes the shortest path to the root and notes which of its ports is on this path. This port is also selected as the switch's preferred path to the root. Finally, to account for the possibility there could be another switch connected to its ports, the switch elect a single *designated* switch that will be responsible for forwarding frames toward the root. Each designated switch is the one that is closest to the root. If two or more switches are equally close to the root, then the switches' identifiers are used to break ties, and the smallest ID wins. Of course, each switch might be connected to more than one other switch, so it participates in the election of a designated switch for each such port. In effect, this means that each switch decides if it is the designated switch relative to each of its ports. The switch forwards frames over those ports for which it is the designated switch.

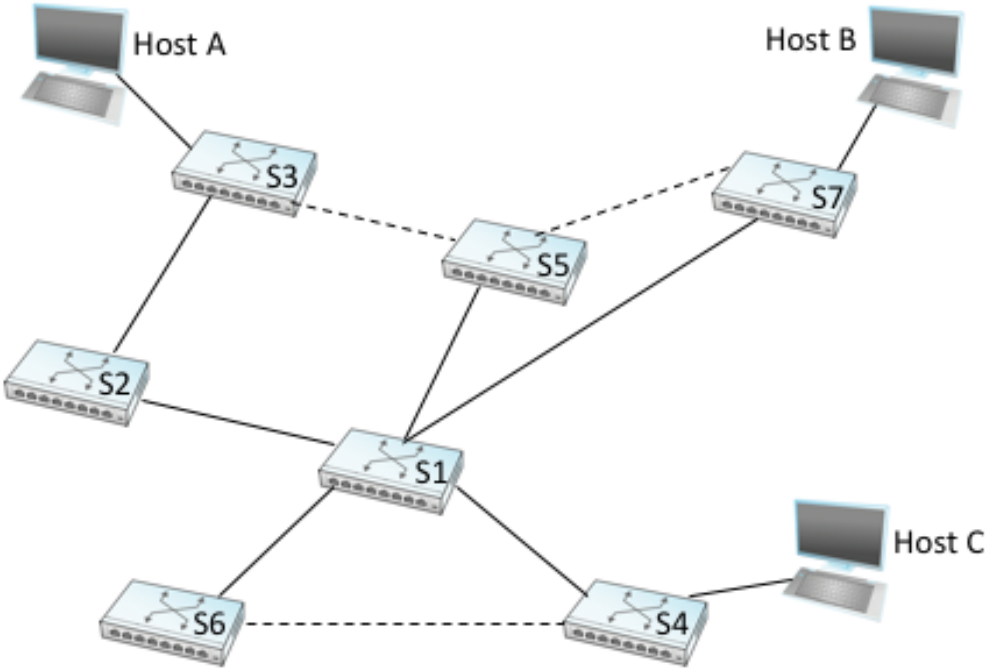


Figure 3.12.: Spanning tree with some ports not selected.

Figure 3.12 shows the spanning tree that corresponds to the network shown in Figure 3.10. In this example, S1 is the root, since it has the smallest ID. Notice that S3 and S5 are connected to each other, but S5 is the designated switch since it is closer to the root. Similarly, S5 and S7 are connected to each other, but in this case S5 is the designated switch since it has the smaller ID; both are an equal distance from S1.

While it is possible for a human to look at the network given in Figure 3.10 and to compute the spanning tree given in the Figure 3.12 according to the rules given above, the switches do not have the luxury of being able to see the topology of the entire network, let alone peek inside other switches to see their ID. Instead, they have to exchange configuration messages with each other and then decide whether or not they are the root or a designated switch based on these messages.

Specifically, the configuration messages contain three pieces of information:

1. The ID for the switch that is sending the message.

2. The ID for what the sending switch believes to be the root switch.
3. The distance, measured in hops, from the sending switch to the root switch.

Each switch records the current *best* configuration message it has seen on each of its ports ("best" is defined below), including both messages it has received from other switches and messages that it has itself transmitted.

Initially, each switch thinks it is the root, and so it sends a configuration message out on each of its ports identifying itself as the root and giving a distance to the root of 0. Upon receiving a configuration message over a particular port, the switch checks to see if that new message is better than the current best configuration message recorded for that port. The new configuration message is considered *better* than the currently recorded information if any of the following is true:

- It identifies a root with a smaller ID.
- It identifies a root with an equal ID but with a shorter distance.
- The root ID and distance are equal, but the sending switch has a smaller ID

If the new message is better than the currently recorded information, the switch discards the old information and saves the new information. However, it first adds 1 to the distance-to-root field since the switch is one hop farther away from the root than the switch that sent the message.

When a switch receives a configuration message indicating that it is not the root—that is, a message from a switch with a smaller ID—the switch stops generating configuration messages on its own and instead only forwards configuration messages from other switches, after first adding 1 to the distance field. Likewise, when a switch receives a configuration message that indicates it is not the designated switch for that port—that is, a message from a switch that is closer to the root or equally far from the root but with a smaller ID—the switch stops sending configuration messages over that port. Thus, when the system stabilizes, only the root switch is still generating configuration messages, and the other switches are forwarding these messages only over ports for which they are the designated switch. At this point, a spanning tree has been built, and all the switches are in agreement on which ports are in use for the spanning tree. Only those ports may be used for forwarding data packets.

Let's see how this works with an example. Consider what would happen in Figure 3.12 if the power had just been restored to a campus, so that all the switches boot at about the same time. All the switches would start off by claiming to be the root. We denote a configuration message from node X in which it claims to be distance d from root node Y as (Y,d,X). Focusing on the activity at S3, a sequence of events would unfold as follows:

1. S3 receives (S2, 0, S2).
2. Since 2 < 3, S3 accepts S2 as root.
3. S3 adds one to the distance advertised by S2 (0) and thus sends (S2, 1, S3) toward S5.
4. Meanwhile, S2 accepts S1 as root because it has the lower ID, and it sends (S1, 1, S2) toward S3.
5. S5 accepts S1 as root and sends (S1, 1, S5) toward S3.
6. S3 accepts S1 as root, and it notes that both S2 and S5 are closer to the root than it is, but S2 has the smaller id, so it remains on S3's path to the root.

This leaves S3 with active ports as shown in Figure 3.12. Note that Hosts A an B are not able to communication over the shortest path (via S5) because frames have to "flow up the tree and back down," but that's the price you pay to avoid loops.

Even after the system has stabilized, the root switch continues to send configuration messages periodically, and the other switches continue to forward these messages as just described. Should a particular switch fail, the downstream switches will not receive these configuration messages, and after waiting a specified period of time they will once again claim to be the root, and the algorithm will kick in again to elect a new root and new designated switches.

One important thing to notice is that although the algorithm is able to reconfigure the spanning tree whenever a switch fails, it is not able to forward frames over alternative paths for the sake of routing around a congested switch.

3.2.4 Broadcast and Multicast

The preceding discussion focuses on how switches forward unicast frames from one port to another. Since the goal of a switch is to transparently extend a LAN across multiple networks, and since most LANs support both broadcast and multicast, then switches must also support these two features. Broadcast is simple—each switch forwards a frame with a destination broadcast address out on each active (selected) port other than the one on which the frame was received.

Multicast can be implemented in exactly the same way, with each host deciding for itself whether or not to accept the message. This is exactly what is done in practice. Notice, however, that since not all hosts are a member of any particular multicast group, it is possible to do better. Specifically, the spanning tree algorithm can be extended to prune networks over which multicast frames need not be forwarded. Consider a frame sent to group M by a host A in Figure 3.12. If host C does not belong to group M, then there is no need for switch S4 to forward the frames over that network.

How would a given switch learn whether it should forward a multicast frame over a given port? It learns exactly the same way that a switch learns whether it should forward a unicast frame over a particular port—by observing the *source* addresses that it receives over that port. Of course, groups are not typically the source of frames, so we have to cheat a little. In particular, each host that is a member of group M must periodically send a frame with the address for group M in the source field of the frame header. This frame would have as its destination address the multicast address for the switches.

Although the multicast extension just described was once proposed, it was not widely adopted. Instead, multicast is implemented in exactly the same way as broadcast.

3.2.5 Virtual LANs (VLANs)

One limitation of switches is that they do not scale. It is not realistic to connect more than a few switches, where in practice *few* typically means "tens of." One reason for this is that the spanning tree algorithm scales linearly; that is, there is no provision for imposing a hierarchy on the set of switches. A second reason is that switches forward all broadcast frames. While it is reasonable for all hosts within a limited setting (say, a department) to see each other's broadcast messages, it is unlikely that all the hosts in a larger environment (say, a large company or university) would want to have to be bothered by each other's broadcast messages. Said another way, broadcast does not scale, and as a consequence L2-based networks do not scale.

One approach to increasing the scalability is the *virtual LAN* (VLAN). VLANs allow a single extended LAN to be partitioned into several seemingly separate LANs. Each virtual LAN is assigned an identifier (sometimes called a *color*), and packets can only travel from one segment to another if both segments have the same identifier. This has the effect of limiting the number of segments in an extended LAN that will receive any given broadcast packet.

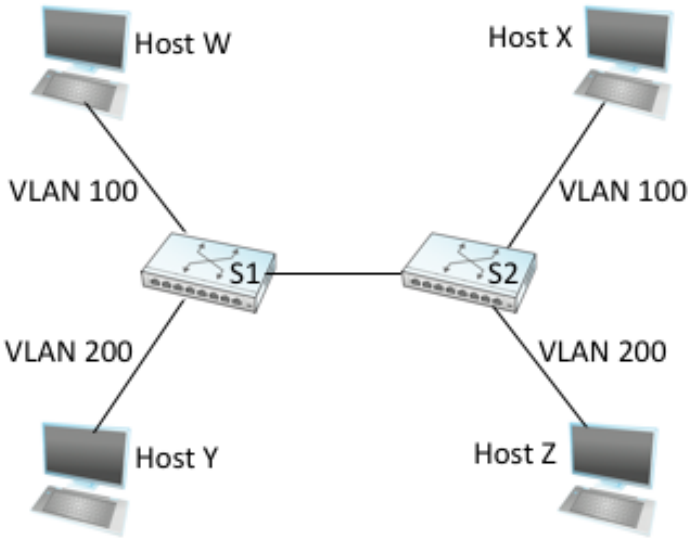


Figure 3.13.: Two virtual LANs share a common backbone.

We can see how VLANs work with an example. Figure 3.13 shows four hosts and two switches. In the absence of VLANs, any broadcast packet from any host will reach all the other hosts. Now let's suppose that we define the segments connected to hosts W and X as being in one VLAN, which we'll call VLAN 100. We also define the segments that connect to hosts Y and Z as being in VLAN 200. To do this, we need to configure a VLAN ID on each port of switches S1 and S2. The link between S1 and S2 is considered to be in both VLANs.

When a packet sent by host X arrives at switch S2, the switch observes that it came in a port that was configured as being in VLAN 100. It inserts a VLAN header between the Ethernet header and its payload. The interesting part of the VLAN header is the VLAN ID; in this case, that ID is set to 100. The switch now applies its normal rules for forwarding to the packet, with the extra restriction that the packet may not be sent out an interface that is not part of VLAN 100. Thus, under no circumstances will the packet—even a broadcast packet—be sent out the interface to host Z, which is in VLAN 200. The packet, however, is forwarded on to switch S1, which follows the same rules and thus may forward the packet to host W but not to host Y.

An attractive feature of VLANs is that it is possible to change the logical topology without moving any wires or changing any addresses. For example, if we wanted to make the link that connects to host Z be part of VLAN 100 and thus enable X, W, and Z to be on the same virtual LAN, then we would just need to change one piece of configuration on switch S2.

Supporting VLANs requires a fairly simple extension to the original 802.1 header specification, inserting a 12-bit VLAN ID (`VID`) field between the `SrcAddr` and `Type` fields, as shown in Figure 3.14. (This VID is typically referred to as a *VLAN Tag*.) There are actually 32-bits inserted in the middle of the header, but the first 16-bits are used to preserve backwards compatibility with the original specification (they use

`Type = 0x8100` to indicate that this frame includes the VLAN extension); the other four bits hold control information used to prioritize frames. This means it is possible to map $2^{12} = 4096$ virtual networks onto a single physical LAN.

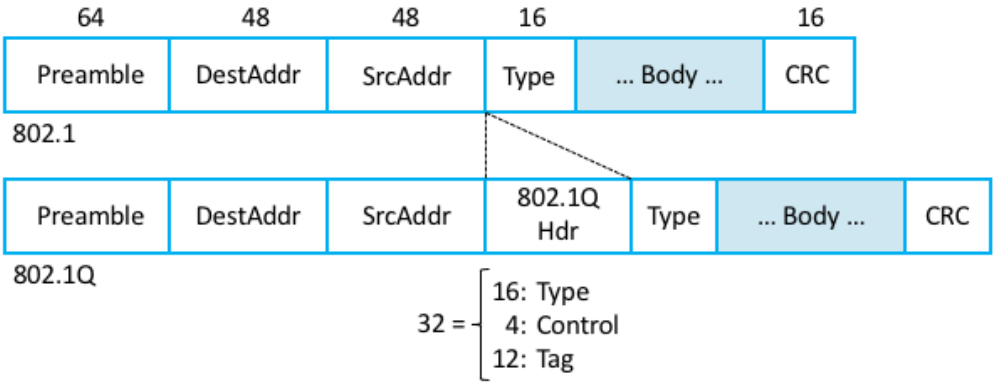


Figure 3.14.: 802.1Q VLAN tag embedded within an Ethernet (802.1) header.

We conclude this discussion by observing there is another limitation of networks built by interconnecting L2 switches: lack of support for heterogeneity. That is, switches are limited in the kinds of networks they can interconnect. In particular, switches make use of the network's frame header and so can support only networks that have exactly the same format for addresses. For example, switches can be used to connect Ethernet and 802.11 based networks to another, since they share a common header format, but switches do not readily generalize to other kinds of networks with different addressing formats, such as ATM, SONET, PON, or the cellular network. The next section explains how to address this limitation, as well as to scale switched networks to even larger sizes.

3.3 Internet (IP)

In the previous section, we saw that it was possible to build reasonably large LANs using bridges and LAN switches, but that such approaches were limited in their ability to scale and to handle heterogeneity. In this section, we explore some ways to go beyond the limitations of bridged networks, enabling us to build large, highly heterogeneous networks with reasonably efficient routing. We refer to such networks as *internetworks*. We'll continue the discussion of how to build a truly global internetwork in the next chapter, but for now we'll explore the basics. We start by considering more carefully what the word *internetwork* means.

3.3.1 What Is an Internetwork?

We use the term *internetwork*, or sometimes just *internet* with a lowercase *i*, to refer to an arbitrary collection of networks interconnected to provide some sort of host-to-host packet delivery service. For example, a corporation with many sites might construct a private internetwork by interconnecting the LANs at their different sites with point-to-point links leased from the phone company. When we are talking about the widely used global internetwork to which a large percentage of networks are now connected, we call it the *Internet* with a capital *I*. In keeping with the first-principles approach of this book, we mainly want you to learn about the principles of "lowercase *i*" internetworking, but we illustrate these ideas with real-world examples from the "big *I*" Internet.

Another piece of terminology that can be confusing is the difference between networks, subnetworks, and internetworks. We are going to avoid subnetworks (or subnets) altogether until a later section. For now, we use *network* to mean either a directly connected or a switched network of the kind described in the previous section and the previous chapter. Such a network uses one technology, such as 802.11 or Ethernet. An *internetwork* is an interconnected collection of such networks. Sometimes, to avoid ambiguity, we refer to the underlying networks that we are interconnecting as *physical* networks. An internet is a *logical* network built out of a collection of physical networks. In this context, a collection of Ethernet segments connected by bridges or switches would still be viewed as a single network.

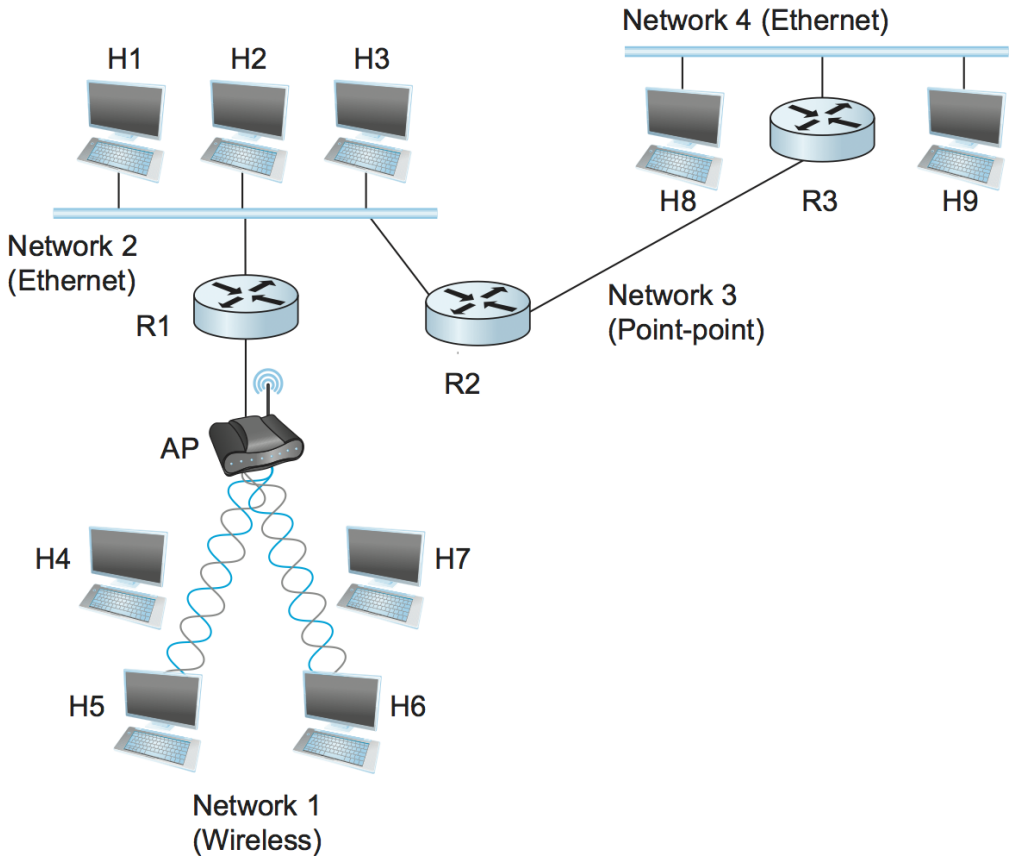


Figure 3.15.: A simple internetwork. H denotes a host and R denotes a router.

Figure 3.15 shows an example internetwork. An internetwork is often referred to as a "network of networks" because it is made up of lots of smaller networks. In this figure, we see Ethernets, a wireless network, and a point-to-point link. Each of these is a single-technology network. The nodes that interconnect the networks are called *routers*. They are also sometimes called *gateways*, but since this term has several other connotations, we restrict our usage to router.

The *Internet Protocol* is the key tool used today to build scalable, heterogeneous internetworks. It was originally known as the Kahn-Cerf protocol after its inventors. One way to think of IP is that it runs on all the nodes (both hosts and routers) in a collection of networks and defines the infrastructure that allows these nodes and networks to function as a single logical internetwork. For example, Figure 3.16 shows how hosts H5 and H8 are logically connected by the internet in Figure 3.15, including the protocol graph running on each node. Note that higher-level protocols, such as TCP and UDP, typically run on top of IP on the hosts.

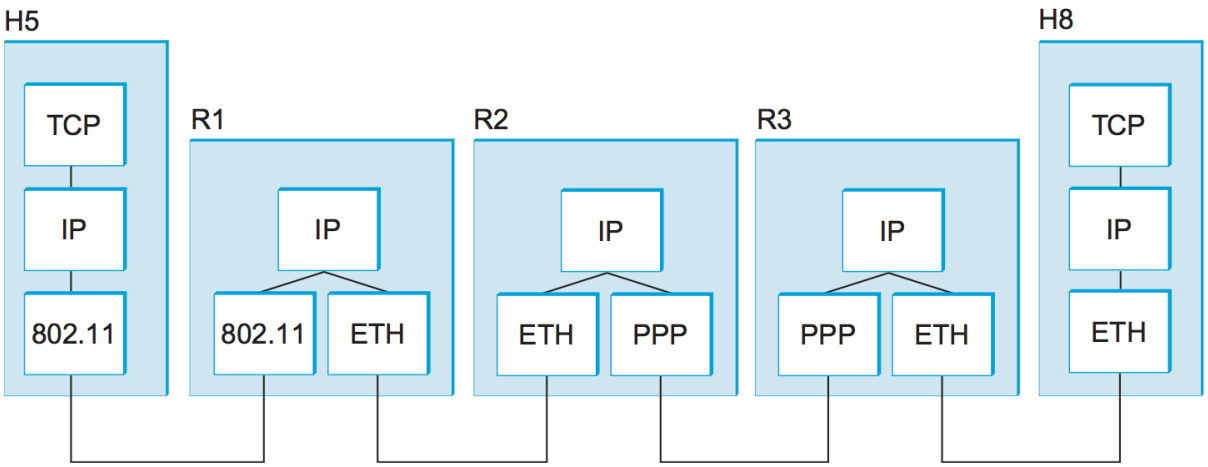


Figure 3.16.: A simple internetwork, showing the protocol layers used to connect H5 to H8 in the above figure. ETH is the protocol that runs over the Ethernet.

The rest of this and the next chapter are about various aspects of IP. While it is certainly possible to build an internetwork that does not use IP—and in fact, in the early days of the Internet there were alternative solutions—IP is the most interesting case to study simply because of the size of the Internet. Said another way, it is only the IP Internet that has really faced the issue of scale. Thus, it provides the best case study of a scalable internetworking protocol.

L2 vs L3 Networks

As seen in the previous section, an Ethernet can be treated as a point-to-point *link* interconnecting a pair of switches, with a mesh of interconnected switches forming a *Switched Ethernet*. This configuration is also known as an *L2 Network*.

But as we'll discover in this section, an Ethernet (even when arranged in a point-to-point configuration rather than a shared CSMA/CD network) can be treated as a *network* interconnecting a pair of routers, with a mesh of such routers forming an Internet. This configuration is also known as an *L3 Network*.

Confusingly, this is because a point-to-point Ethernet is both a link and a network (albeit a trivial two-node network in the second case), depending on whether it's connected to a pair of L2 switches running the spanning tree algorithm, or to a pair of L3 routers running IP (plus the routing protocols described later in this chapter). Why pick one configuration over the other? It partly depends on whether you want the network to be a single broadcast domain (if yes, pick L2), and whether you want the hosts connected to the network to be on different networks (if yes, select L3).

The good news is that when you fully understand the implications of this duality, you will have cleared a major hurdle in mastering modern packet-switched networks.

3.3.2 Service Model

A good place to start when you build an internetwork is to define its *service model*, that is, the host-to-host services you want to provide. The main concern in defining a service model for an internetwork is

that we can provide a host-to-host service only if this service can somehow be provided over each of the underlying physical networks. For example, it would be no good deciding that our internetwork service model was going to provide guaranteed delivery of every packet in 1 ms or less if there were underlying network technologies that could arbitrarily delay packets. The philosophy used in defining the IP service model, therefore, was to make it undemanding enough that just about any network technology that might turn up in an internetwork would be able to provide the necessary service.

The IP service model can be thought of as having two parts: an addressing scheme, which provides a way to identify all hosts in the internetwork, and a datagram (connectionless) model of data delivery. This service model is sometimes called *best effort* because, although IP makes every effort to deliver datagrams, it makes no guarantees. We postpone a discussion of the addressing scheme for now and look first at the data delivery model.

Datagram Delivery

The IP datagram is fundamental to the Internet Protocol. Recall an earlier section that a datagram is a type of packet that happens to be sent in a connectionless manner over a network. Every datagram carries enough information to let the network forward the packet to its correct destination; there is no need for any advance setup mechanism to tell the network what to do when the packet arrives. You just send it, and the network makes its best effort to get it to the desired destination. The "best-effort" part means that if something goes wrong and the packet gets lost, corrupted, misdelivered, or in any way fails to reach its intended destination, the network does nothing—it made its best effort, and that is all it has to do. It does not make any attempt to recover from the failure. This is sometimes called an *unreliable* service.

Best-effort, connectionless service is about the simplest service you could ask for from an internetwork, and this is a great strength. For example, if you provide best-effort service over a network that provides a reliable service, then that's fine—you end up with a best-effort service that just happens to always deliver the packets. If, on the other hand, you had a reliable service model over an unreliable network, you would have to put lots of extra functionality into the routers to make up for the deficiencies of the underlying network. Keeping the routers as simple as possible was one of the original design goals of IP.

The ability of IP to "run over anything" is frequently cited as one of its most important characteristics. It is noteworthy that many of the technologies over which IP runs today did not exist when IP was invented. So far, no networking technology has been invented that has proven too bizarre for IP; in principle, IP can run over a network that transports messages using carrier pigeons.

Best-effort delivery does not just mean that packets can get lost. Sometimes they can get delivered out of order, and sometimes the same packet can get delivered more than once. The higher-level protocols or applications that run above IP need to be aware of all these possible failure modes.

Packet Format

Clearly, a key part of the IP service model is the type of packets that can be carried. The IP datagram, like most packets, consists of a header followed by a number of bytes of data. The format of the header is shown in Figure 3.17. Note that we have adopted a different style of representing packets than the one we used in previous chapters. This is because packet formats at the internetworking layer and above, where we will be focusing our attention for the next few chapters, are almost invariably designed to align on 32-bit boundaries to simplify the task of processing them in software. Thus, the common way of representing them (used in

Internet Requests for Comments, for example) is to draw them as a succession of 32-bit words. The top word is the one transmitted first, and the leftmost byte of each word is the one transmitted first. In this representation, you can easily recognize fields that are a multiple of 8 bits long. On the odd occasion when fields are not an even multiple of 8 bits, you can determine the field lengths by looking at the bit positions marked at the top of the packet.

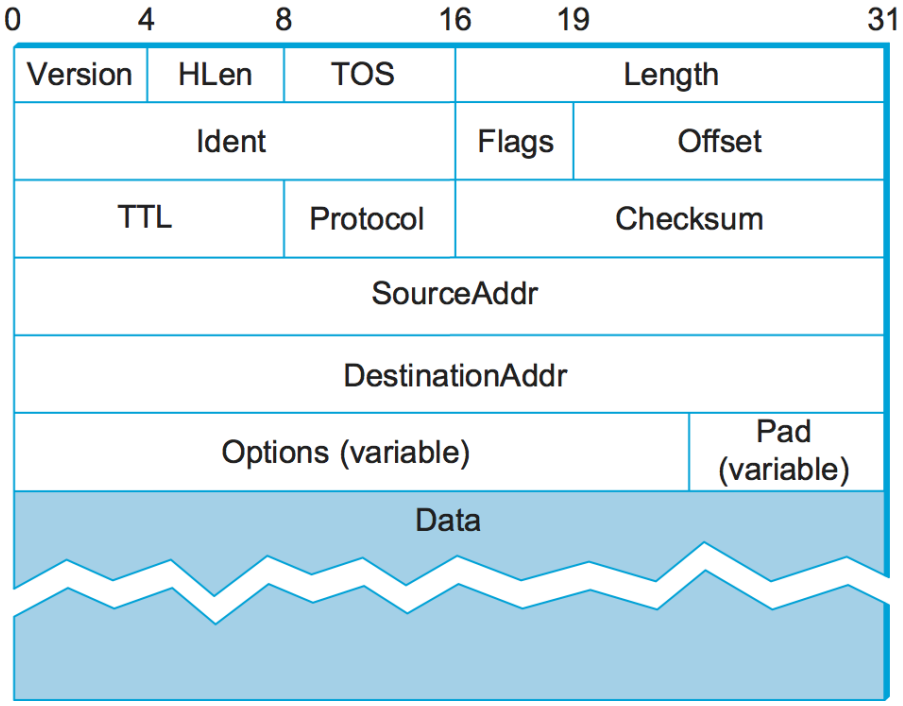


Figure 3.17.: IPv4 packet header.

Looking at each field in the IP header, we see that the "simple" model of best-effort datagram delivery still has some subtle features. The `Version` field specifies the version of IP. The still-assumed version of IP is 4, which is typically called *IPv4*. Observe that putting this field right at the start of the datagram makes it easy for everything else in the packet format to be redefined in subsequent versions; the header processing software starts off by looking at the version and then branches off to process the rest of the packet according to the appropriate format. The next field, `HLen`, specifies the length of the header in 32-bit words. When there are no options, which is most of the time, the header is 5 words (20 bytes) long. The 8-bit `TOS` (type of service) field has had a number of different definitions over the years, but its basic function is to allow packets to be treated differently based on application needs. For example, the `TOS` value might determine whether or not a packet should be placed in a special queue that receives low delay.

The next 16 bits of the header contain the `Length` of the datagram, including the header. Unlike the `HLen` field, the `Length` field counts bytes rather than words. Thus, the maximum size of an IP datagram is 65,535 bytes. The physical network over which IP is running, however, may not support such long packets. For this reason, IP supports a fragmentation and reassembly process. The second word of the header contains information about fragmentation, and the details of its use are presented in the following section entitled "Fragmentation and Reassembly."

Moving on to the third word of the header, the next byte is the `TTL` (time to live) field. Its name reflects its historical meaning rather than the way it is commonly used today. The intent of the field is to catch packets that have been going around in routing loops and discard them, rather than let them consume resources

indefinitely. Originally, `TTL` was set to a specific number of seconds that the packet would be allowed to live, and routers along the path would decrement this field until it reached 0. However, since it was rare for a packet to sit for as long as 1 second in a router, and routers did not all have access to a common clock, most routers just decremented the `TTL` by 1 as they forwarded the packet. Thus, it became more of a hop count than a timer, which is still a perfectly good way to catch packets that are stuck in routing loops. One subtlety is in the initial setting of this field by the sending host: Set it too high and packets could circulate rather a lot before getting dropped; set it too low and they may not reach their destination. The value 64 is the current default.

The `Protocol` field is simply a demultiplexing key that identifies the higher-level protocol to which this IP packet should be passed. There are values defined for the TCP (Transmission Control Protocol—6), UDP (User Datagram Protocol—17), and many other protocols that may sit above IP in the protocol graph.

The `Checksum` is calculated by considering the entire IP header as a sequence of 16-bit words, adding them up using ones' complement arithmetic, and taking the ones' complement of the result. Thus, if any bit in the header is corrupted in transit, the checksum will not contain the correct value upon receipt of the packet. Since a corrupted header may contain an error in the destination address—and, as a result, may have been misdelivered—it makes sense to discard any packet that fails the checksum. It should be noted that this type of checksum does not have the same strong error detection properties as a CRC, but it is much easier to calculate in software.

The last two required fields in the header are the `SourceAddr` and the `DestinationAddr` for the packet. The latter is the key to datagram delivery: Every packet contains a full address for its intended destination so that forwarding decisions can be made at each router. The source address is required to allow recipients to decide if they want to accept the packet and to enable them to reply. IP addresses are discussed in a later section—for now, the important thing to know is that IP defines its own global address space, independent of whatever physical networks it runs over. As we will see, this is one of the keys to supporting heterogeneity.

Finally, there may be a number of options at the end of the header. The presence or absence of options may be determined by examining the header length (`HLen`) field. While options are used fairly rarely, a complete IP implementation must handle them all.

Fragmentation and Reassembly

One of the problems of providing a uniform host-to-host service model over a heterogeneous collection of networks is that each network technology tends to have its own idea of how large a packet can be. For example, classic Ethernet can accept packets up to 1500 bytes long, but modern-day variants can deliver larger (jumbo) packets that carry up to 9000 bytes of payload. This leaves two choices for the IP service model: Make sure that all IP datagrams are small enough to fit inside one packet on any network technology, or provide a means by which packets can be fragmented and reassembled when they are too big to go over a given network technology. The latter turns out to be a good choice, especially when you consider the fact that new network technologies are always turning up, and IP needs to run over all of them; this would make it hard to pick a suitably small bound on datagram size. This also means that a host will not send needlessly small packets, which wastes bandwidth and consumes processing resources by requiring more headers per byte of data sent.

The central idea here is that every network type has a *maximum transmission unit* (MTU), which is the

largest IP datagram that it can carry in a frame.[1] Note that this value is smaller than the largest packet size on that network because the IP datagram needs to fit in the *payload* of the link-layer frame.

When a host sends an IP datagram, therefore, it can choose any size that it wants. A reasonable choice is the MTU of the network to which the host is directly attached. Then, fragmentation will only be necessary if the path to the destination includes a network with a smaller MTU. Should the transport protocol that sits on top of IP give IP a packet larger than the local MTU, however, then the source host must fragment it.

Fragmentation typically occurs in a router when it receives a datagram that it wants to forward over a network that has an MTU that is smaller than the received datagram. To enable these fragments to be reassembled at the receiving host, they all carry the same identifier in the `Ident` field. This identifier is chosen by the sending host and is intended to be unique among all the datagrams that might arrive at the destination from this source over some reasonable time period. Since all fragments of the original datagram contain this identifier, the reassembling host will be able to recognize those fragments that go together. Should all the fragments not arrive at the receiving host, the host gives up on the reassembly process and discards the fragments that did arrive. IP does not attempt to recover from missing fragments.

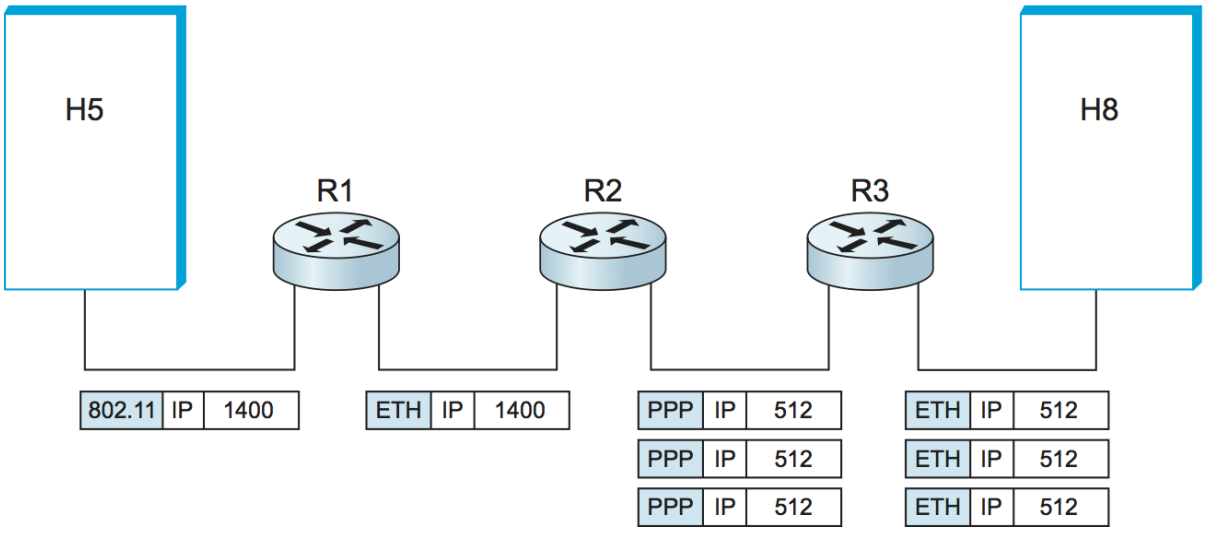


Figure 3.18.: IP datagrams traversing the sequence of physical networks graphed in the earlier figure.

To see what this all means, consider what happens when host H5 sends a datagram to host H8 in the example internet shown in Figure 3.15. Assuming that the MTU is 1500 bytes for the two Ethernets and the 802.11 network, and 532 bytes for the point-to-point network, then a 1420-byte datagram (20-byte IP header plus 1400 bytes of data) sent from H5 makes it across the 802.11 network and the first Ethernet without fragmentation but must be fragmented into three datagrams at router R2. These three fragments are then forwarded by router R3 across the second Ethernet to the destination host. This situation is illustrated in Figure 3.18. This figure also serves to reinforce two important points:

1. Each fragment is itself a self-contained IP datagram that is transmitted over a sequence of physical networks, independent of the other fragments.

2. Each IP datagram is re-encapsulated for each physical network over which it travels.

The fragmentation process can be understood in detail by looking at the header fields of each datagram, as

[1] In ATM networks, the MTU is, fortunately, much larger than a single cell, as ATM has its own fragmentation and reassembly mechanism. The link-layer frame in ATM is called a *convergence-sublayer protocol data unit* (CS-PDU).

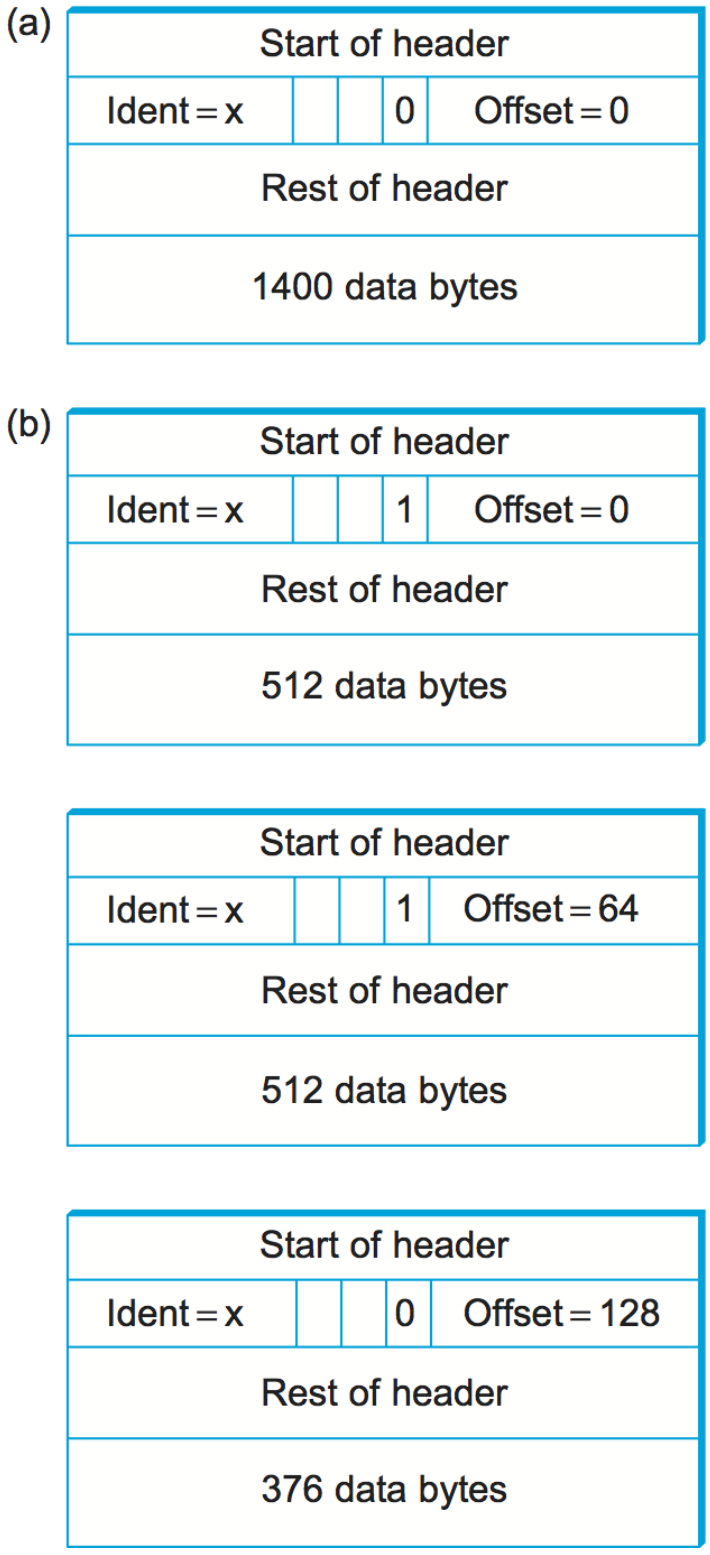


Figure 3.19.: Header fields used in IP fragmentation: (a) unfragmented packet; (b) fragmented packets.

is done in Figure 3.19. The unfragmented packet, shown at the top, has 1400 bytes of data and a 20-byte IP header. When the packet arrives at router R2, which has an MTU of 532 bytes, it has to be fragmented. A 532-byte MTU leaves 512 bytes for data after the 20-byte IP header, so the first fragment contains 512 bytes of data. The router sets the M bit in the `Flags` field (see Figure 3.17), meaning that there are more fragments to follow, and it sets the `Offset` to 0, since this fragment contains the first part of the original datagram. The data carried in the second fragment starts with the 513th byte of the original data, so the `Offset` field in this header is set to 64, which is 512/8. Why the division by 8? Because the designers of IP decided that fragmentation should always happen on 8-byte boundaries, which means that the `Offset` field counts 8-byte chunks, not bytes. (We leave it as an exercise for you to figure out why this design decision was made.) The third fragment contains the last 376 bytes of data, and the offset is now $2 \times 512/8 = 128$. Since this is the last fragment, the M bit is not set.

Observe that the fragmentation process is done in such a way that it could be repeated if a fragment arrived at another network with an even smaller MTU. Fragmentation produces smaller, valid IP datagrams that can be readily reassembled into the original datagram upon receipt, independent of the order of their arrival. Reassembly is done at the receiving host and not at each router.

IP reassembly is far from a simple process. For example, if a single fragment is lost, the receiver will still attempt to reassemble the datagram, and it will eventually give up and have to garbage-collect the resources that were used to perform the failed reassembly. Getting a host to tie up resources needlessly can be the basis of a denial-of-service attack.

For this reason, among others, IP fragmentation is generally considered a good thing to avoid. Hosts are now strongly encouraged to perform "path MTU discovery," a process by which fragmentation is avoided by sending packets that are small enough to traverse the link with the smallest MTU in the path from sender to receiver.

3.3.3 Global Addresses

In the above discussion of the IP service model, we mentioned that one of the things that it provides is an addressing scheme. After all, if you want to be able to send data to any host on any network, there needs to be a way of identifying all the hosts. Thus, we need a global addressing scheme—one in which no two hosts have the same address. Global uniqueness is the first property that should be provided in an addressing scheme.

Ethernet addresses are globally unique, but that alone does not suffice for an addressing scheme in a large internetwork. Ethernet addresses are also *flat*, which means that they have no structure and provide very few clues to routing protocols. (In fact, Ethernet addresses do have a structure for the purposes of *assignment*—the first 24 bits identify the manufacturer—but this provides no useful information to routing protocols since this structure has nothing to do with network topology.) In contrast, IP addresses are *hierarchical*, by which we mean that they are made up of several parts that correspond to some sort of hierarchy in the internetwork. Specifically, IP addresses consist of two parts, usually referred to as a *network* part and a *host* part. This is a fairly logical structure for an internetwork, which is made up of many interconnected networks. The network part of an IP address identifies the network to which the host is attached; all hosts attached to the same network have the same network part in their IP address. The host part then identifies each host uniquely on that particular network. Thus, in the simple internetwork of Figure 3.15, the addresses of the hosts on network 1, for example, would all have the same network part and different host parts.

Note that the routers in Figure 3.15 are attached to two networks. They need to have an address on each

network, one for each interface. For example, router R1, which sits between the wireless network and an Ethernet, has an IP address on the interface to the wireless network whose network part is the same as all the hosts on that network. It also has an IP address on the interface to the Ethernet that has the same network part as the hosts on that Ethernet. Thus, bearing in mind that a router might be implemented as a host with two network interfaces, it is more precise to think of IP addresses as belonging to interfaces than to hosts.

Now, what do these hierarchical addresses look like? Unlike some other forms of hierarchical address, the sizes of the two parts are not the same for all addresses. Originally, IP addresses were divided into three different classes, as shown in Figure 3.20, each of which defines different-sized network and host parts. (There are also class D addresses that specify a multicast group and class E addresses that are currently unused.) In all cases, the address is 32 bits long.

The class of an IP address is identified in the most significant few bits. If the first bit is 0, it is a class A address. If the first bit is 1 and the second is 0, it is a class B address. If the first two bits are 1 and the third is 0, it is a class C address. Thus, of the approximately 4 billion possible IP addresses, half are class A, one-quarter are class B, and one-eighth are class C. Each class allocates a certain number of bits for the network part of the address and the rest for the host part. Class A networks have 7 bits for the network part and 24 bits for the host part, meaning that there can be only 126 class A networks (the values 0 and 127 are reserved), but each of them can accommodate up to $2^{24} - 2$ (about 16 million) hosts (again, there are two reserved values). Class B addresses allocate 14 bits for the network and 16 bits for the host, meaning that each class B network has room for 65,534 hosts. Finally, class C addresses have only 8 bits for the host and 21 for the network part. Therefore, a class C network can have only 256 unique host identifiers, which means only 254 attached hosts (one host identifier, 255, is reserved for broadcast, and 0 is not a valid host number). However, the addressing scheme supports 2^{21} class C networks.

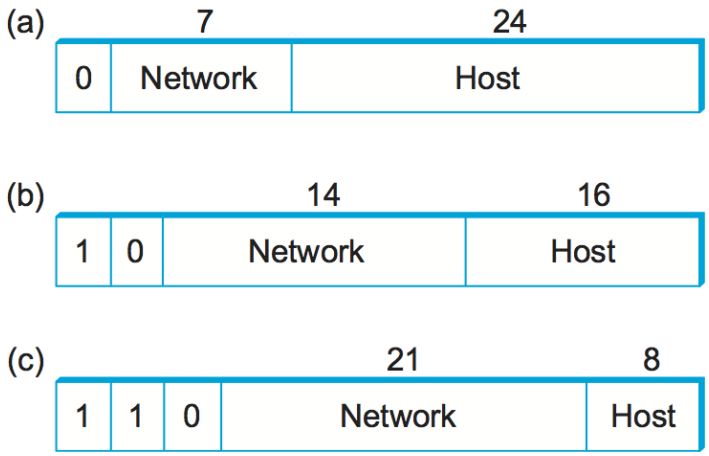


Figure 3.20.: IP addresses: (a) class A; (b) class B; (c) class C.

On the face of it, this addressing scheme has a lot of flexibility, allowing networks of vastly different sizes to be accommodated fairly efficiently. The original idea was that the Internet would consist of a small number of wide area networks (these would be class A networks), a modest number of site- (campus-) sized networks (these would be class B networks), and a large number of LANs (these would be class C networks). However, it turned out not to be flexible enough, as we will see in a moment. Today, IP addresses are normally "classless"; the details of this are explained below.

Before we look at how IP addresses get used, it is helpful to look at some practical matters, such as how you write them down. By convention, IP addresses are written as four *decimal* integers separated by dots. Each

integer represents the decimal value contained in 1 byte of the address, starting at the most significant. For example, the address of the computer on which this sentence was typed is `171.69.210.245`.

It is important not to confuse IP addresses with Internet domain names, which are also hierarchical. Domain names tend to be ASCII strings separated by dots, such as `cs.princeton.edu`. The important thing about IP addresses is that they are what is carried in the headers of IP packets, and it is those addresses that are used in IP routers to make forwarding decisions.

3.3.4 Datagram Forwarding in IP

We are now ready to look at the basic mechanism by which IP routers forward datagrams in an internetwork. Recall from an earlier section that *forwarding* is the process of taking a packet from an input and sending it out on the appropriate output, while *routing* is the process of building up the tables that allow the correct output for a packet to be determined. The discussion here focuses on forwarding; we take up routing in a later section.

The main points to bear in mind as we discuss the forwarding of IP datagrams are the following:

- Every IP datagram contains the IP address of the destination host.
- The network part of an IP address uniquely identifies a single physical network that is part of the larger Internet.
- All hosts and routers that share the same network part of their address are connected to the same physical network and can thus communicate with each other by sending frames over that network.
- Every physical network that is part of the Internet has at least one router that, by definition, is also connected to at least one other physical network; this router can exchange packets with hosts or routers on either network.

Forwarding IP datagrams can therefore be handled in the following way. A datagram is sent from a source host to a destination host, possibly passing through several routers along the way. Any node, whether it is a host or a router, first tries to establish whether it is connected to the same physical network as the destination. To do this, it compares the network part of the destination address with the network part of the address of each of its network interfaces. (Hosts normally have only one interface, while routers normally have two or more, since they are typically connected to two or more networks.) If a match occurs, then that means that the destination lies on the same physical network as the interface, and the packet can be directly delivered over that network. A later section explains some of the details of this process.

If the node is not connected to the same physical network as the destination node, then it needs to send the datagram to a router. In general, each node will have a choice of several routers, and so it needs to pick the best one, or at least one that has a reasonable chance of getting the datagram closer to its destination. The router that it chooses is known as the *next hop* router. The router finds the correct next hop by consulting its forwarding table. The forwarding table is conceptually just a list of `(NetworkNum, NextHop)` pairs. (As we will see below, forwarding tables in practice often contain some additional information related to the next hop.) Normally, there is also a default router that is used if none of the entries in the table matches the destination's network number. For a host, it may be quite acceptable to have a default router and nothing else—this means that all datagrams destined for hosts not on the physical network to which the sending host is attached will be sent out through the default router.

We can describe the datagram forwarding algorithm in the following way:

```
if (NetworkNum of destination = NetworkNum of one of my interfaces) then
    deliver packet to destination over that interface
else
    if (NetworkNum of destination is in my forwarding table) then
        deliver packet to NextHop router
    else
        deliver packet to default router
```

For a host with only one interface and only a default router in its forwarding table, this simplifies to

```
if (NetworkNum of destination = my NetworkNum) then
    deliver packet to destination directly
else
    deliver packet to default router
```

Let's see how this works in the example internetwork of Figure 3.15. First, suppose that H1 wants to send a datagram to H2. Since they are on the same physical network, H1 and H2 have the same network number in their IP address. Thus, H1 deduces that it can deliver the datagram directly to H2 over the Ethernet. The one issue that needs to be resolved is how H1 finds out the correct Ethernet address for H2—the resolution mechanism described in a later section addresses this issue.

Now suppose H5 wants to send a datagram to H8. Since these hosts are on different physical networks, they have different network numbers, so H5 deduces that it needs to send the datagram to a router. R1 is the only choice—the default router—so H1 sends the datagram over the wireless network to R1. Similarly, R1 knows that it cannot deliver a datagram directly to H8 because neither of R1's interfaces are on the same network as H8. Suppose R1's default router is R2; R1 then sends the datagram to R2 over the Ethernet. Assuming R2 has the forwarding table shown in Table 3.6, it looks up H8's network number (network 4) and forwards the datagram over the point-to-point network to R3. Finally, R3, since it is on the same network as H8, forwards the datagram directly to H8.

Table 3.6.: Forwarding table for Router R2.

NetworkNum	NextHop
1	R1
4	R3

Note that it is possible to include the information about directly connected networks in the forwarding table. For example, we could label the network interfaces of router R2 as interface 0 for the point-to-point link (network 3) and interface 1 for the Ethernet (network 2). Then R2 would have the forwarding table shown in Table 3.7.

Table 3.7.: Complete Forwarding table for Router R2.

NetworkNum	NextHop
1	R1
2	Interface 1
3	Interface 0
4	R3

Thus, for any network number that R2 encounters in a packet, it knows pwhat to do. Either that network is

directly connected to R2, in which case the packet can be delivered to its destination over that network, or the network is reachable via some next hop router that R2 can reach over a network to which it is connected. In either case, R2 will use ARP, described below, to find the MAC address of the node to which the packet is to be sent next.

The forwarding table used by R2 is simple enough that it could be manually configured. Usually, however, these tables are more complex and would be built up by running a routing protocol such as one of those described in a later section. Also note that, in practice, the network numbers are usually longer (e.g., 128.96).

We can now see how hierarchical addressing—splitting the address into network and host parts—has improved the scalability of a large network. Routers now contain forwarding tables that list only a set of network numbers rather than all the nodes in the network. In our simple example, that meant that R2 could store the information needed to reach all the hosts in the network (of which there were eight) in a four-entry table. Even if there were 100 hosts on each physical network, R2 would still only need those same four entries. This is a good first step (although by no means the last) in achieving scalability.

Key Takeaway

This illustrates one of the most important principles of building scalable networks: To achieve scalability, you need to reduce the amount of information that is stored in each node and that is exchanged between nodes. The most common way to do that is *hierarchical aggregation*. IP introduces a two-level hierarchy, with networks at the top level and nodes at the bottom level. We have aggregated information by letting routers deal only with reaching the right network; the information that a router needs to deliver a datagram to any node on a given network is represented by a single aggregated piece of information. *[Next]*

3.3.5 Subnetting and Classless Addressing

The original intent of IP addresses was that the network part would uniquely identify exactly one physical network. It turns out that this approach has a couple of drawbacks. Imagine a large campus that has lots of internal networks and decides to connect to the Internet. For every network, no matter how small, the site needs at least a class C network address. Even worse, for any network with more than 255 hosts, they need a class B address. This may not seem like a big deal, and indeed it wasn't when the Internet was first envisioned, but there are only a finite number of network numbers, and there are far fewer class B addresses than class Cs. Class B addresses tend to be in particularly high demand because you never know if your network might expand beyond 255 nodes, so it is easier to use a class B address from the start than to have to renumber every host when you run out of room on a class C network. The problem we observe here is address assignment inefficiency: A network with two nodes uses an entire class C network address, thereby wasting 253 perfectly useful addresses; a class B network with slightly more than 255 hosts wastes over 64,000 addresses.

Assigning one network number per physical network, therefore, uses up the IP address space potentially much faster than we would like. While we would need to connect over 4 billion hosts to use up all the valid addresses, we only need to connect 2^{14} (about 16,000) class B networks before that part of the address space runs out. Therefore, we would like to find some way to use the network numbers more efficiently.

Assigning many network numbers has another drawback that becomes apparent when you think about routing. Recall that the amount of state that is stored in a node participating in a routing protocol is proportional to the number of other nodes, and that routing in an internet consists of building up forwarding tables that

tell a router how to reach different networks. Thus, the more network numbers there are in use, the bigger the forwarding tables get. Big forwarding tables add costs to routers, and they are potentially slower to search than smaller tables for a given technology, so they degrade router performance. This provides another motivation for assigning network numbers carefully.

Subnetting provides a first step to reducing total number of network numbers that are assigned. The idea is to take a single IP network number and allocate the IP addresses with that network number to several physical networks, which are now referred to as *subnets*. Several things need to be done to make this work. First, the subnets should be close to each other. This is because from a distant point in the Internet, they will all look like a single network, having only one network number between them. This means that a router will only be able to select one route to reach any of the subnets, so they had better all be in the same general direction. A perfect situation in which to use subnetting is a large campus or corporation that has many physical networks. From outside the campus, all you need to know to reach any subnet inside the campus is where the campus connects to the rest of the Internet. This is often at a single point, so one entry in your forwarding table will suffice. Even if there are multiple points at which the campus is connected to the rest of the Internet, knowing how to get to one point in the campus network is still a good start.

The mechanism by which a single network number can be shared among multiple networks involves configuring all the nodes on each subnet with a *subnet mask*. With simple IP addresses, all hosts on the same network must have the same network number. The subnet mask enables us to introduce a *subnet number*; all hosts on the same physical network will have the same subnet number, which means that hosts may be on different physical networks but share a single network number. This concept is illustrated in Figure 3.21.

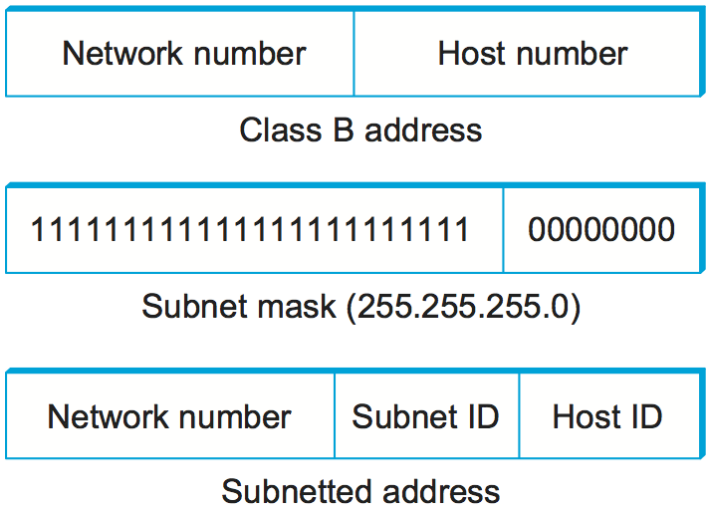


Figure 3.21.: Subnet addressing.

What subnetting means to a host is that it is now configured with both an IP address and a subnet mask for the subnet to which it is attached. For example, host H1 in Figure 3.22 is configured with an address of 128.96.34.15 and a subnet mask of 255.255.255.128. (All hosts on a given subnet are configured with the same mask; that is, there is exactly one subnet mask per subnet.) The bitwise AND of these two numbers defines the subnet number of the host and of all other hosts on the same subnet. In this case, 128.96.34.15 AND 255.255.255.128 equals 128.96.34.0, so this is the subnet number for the topmost subnet in the figure.

When the host wants to send a packet to a certain IP address, the first thing it does is to perform a bitwise AND between its own subnet mask and the destination IP address. If the result equals the subnet number of the sending host, then it knows that the destination host is on the same subnet and the packet can be delivered

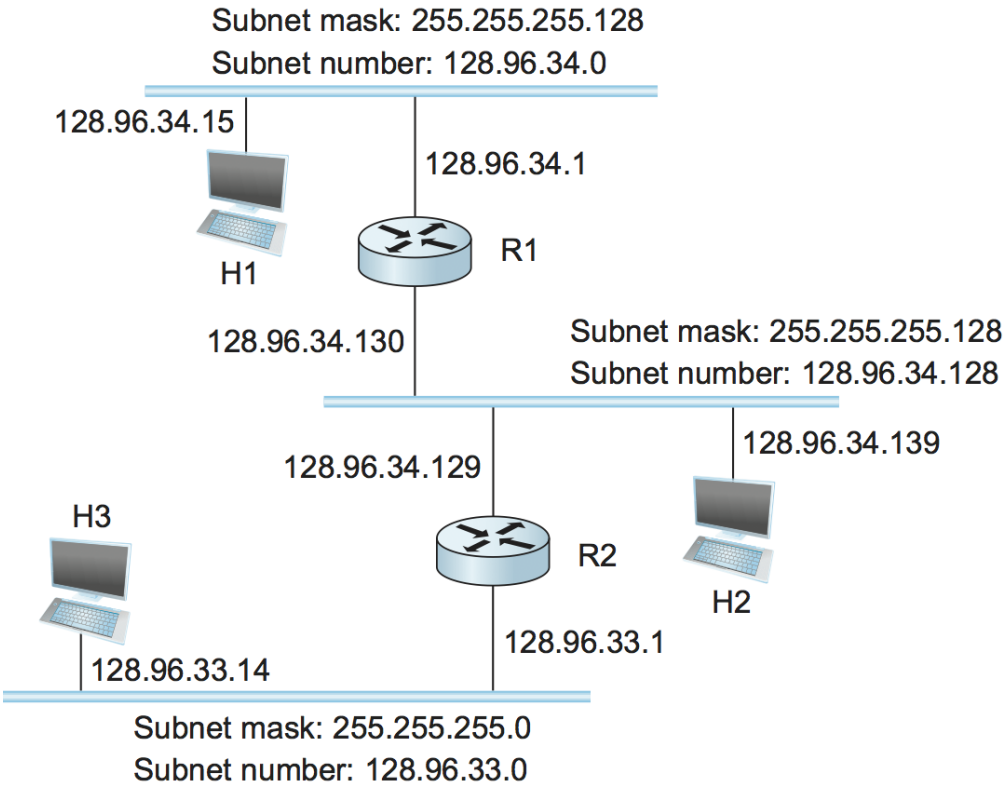


Figure 3.22.: An example of subnetting.

directly over the subnet. If the results are not equal, the packet needs to be sent to a router to be forwarded to another subnet. For example, if H1 is sending to H2, then H1 ANDs its subnet mask (255.255.255.128) with the address for H2 (128.96.34.139) to obtain 128.96.34.128. This does not match the subnet number for H1 (128.96.34.0) so H1 knows that H2 is on a different subnet. Since H1 cannot deliver the packet to H2 directly over the subnet, it sends the packet to its default router R1.

The forwarding table of a router also changes slightly when we introduce subnetting. Recall that we previously had a forwarding table that consisted of entries of the form `(NetworkNum, NextHop)`. To support subnetting, the table must now hold entries of the form `(SubnetNumber, SubnetMask, NextHop)`. To find the right entry in the table, the router ANDs the packet's destination address with the `SubnetMask`for each entry in turn; if the result matches the `SubnetNumber` of the entry, then this is the right entry to use, and it forwards the packet to the next hop router indicated. In the example network of Figure 3.22, router R1 would have the entries shown in Table 3.8.

Table 3.8.: Example Forwarding Table with Subnetting.

SubnetNumber	SubnetMask	NextHop
128.96.34.0	255.255.255.128	Interface 0
128.96.34.128	255.255.255.128	Interface 1
128.96.33.0	255.255.255.0	R2

Continuing with the example of a datagram from H1 being sent to H2, R1 would AND H2's address (128.96.34.139) with the subnet mask of the first entry (255.255.255.128) and compare the result

(128.96.34.128) with the network number for that entry (128.96.34.0). Since this is not a match, it proceeds to the next entry. This time a match does occur, so R1 delivers the datagram to H2 using interface 1, which is the interface connected to the same network as H2.

We can now describe the datagram forwarding algorithm in the following way:

```
D = destination IP address
for each forwarding table entry (SubnetNumber, SubnetMask, NextHop)
    D1 = SubnetMask & D
    if D1 = SubnetNumber
        if NextHop is an interface
            deliver datagram directly to destination
        else
            deliver datagram to NextHop (a router)
```

Although not shown in this example, a default route would usually be included in the table and would be used if no explicit matches were found. Note that a naive implementation of this algorithm—one involving repeated ANDing of the destination address with a subnet mask that may not be different every time, and a linear table search—would be very inefficient.

An important consequence of subnetting is that different parts of the internet see the world differently. From outside our hypothetical campus, routers see a single network. In the example above, routers outside the campus see the collection of networks in Figure 3.22 as just the network 128.96, and they keep one entry in their forwarding tables to tell them how to reach it. Routers within the campus, however, need to be able to route packets to the right subnet. Thus, not all parts of the internet see exactly the same routing information. This is an example of an *aggregation* of routing information, which is fundamental to scaling of the routing system. The next section shows how aggregation can be taken to another level.

Classless Addressing

Subnetting has a counterpart, sometimes called *supernetting*, but more often called *Classless Interdomain Routing* or CIDR, pronounced "cider." CIDR takes the subnetting idea to its logical conclusion by essentially doing away with address classes altogether. Why isn't subnetting alone sufficient? In essence, subnetting only allows us to split a classful address among multiple subnets, while CIDR allows us to coalesce several classful addresses into a single "supernet." This further tackles the address space inefficiency noted above, and does so in a way that keeps the routing system from being overloaded.

To see how the issues of address space efficiency and scalability of the routing system are coupled, consider the hypothetical case of a company whose network has 256 hosts on it. That is slightly too many for a Class C address, so you would be tempted to assign a class B. However, using up a chunk of address space that could address 65535 to address 256 hosts has an efficiency of only 256/65,535 = 0.39%. Even though subnetting can help us to assign addresses carefully, it does not get around the fact that any organization with more than 255 hosts, or an expectation of eventually having that many, wants a class B address.

The first way you might deal with this issue would be to refuse to give a class B address to any organization that requests one unless they can show a need for something close to 64K addresses, and instead giving them an appropriate number of class C addresses to cover the expected number of hosts. Since we would now be handing out address space in chunks of 256 addresses at a time, we could more accurately match the amount of address space consumed to the size of the organization. For any organization with at least 256 hosts, we

can guarantee an address utilization of at least 50%, and typically much more. (Sadly, even if you can justify a request of a class B network number, don't bother, because they were all spoken for long ago.)

This solution, however, raises a problem that is at least as serious: excessive storage requirements at the routers. If a single site has, say, 16 class C network numbers assigned to it, that means every Internet backbone router needs 16 entries in its routing tables to direct packets to that site. This is true even if the path to every one of those networks is the same. If we had assigned a class B address to the site, the same routing information could be stored in one table entry. However, our address assignment efficiency would then be only 6 x 255 / 65,536 = 6.2%.

CIDR, therefore, tries to balance the desire to minimize the number of routes that a router needs to know against the need to hand out addresses efficiently. To do this, CIDR helps us to *aggregate* routes. That is, it lets us use a single entry in a forwarding table to tell us how to reach a lot of different networks. As noted above it does this by breaking the rigid boundaries between address classes. To understand how this works, consider our hypothetical organization with 16 class C network numbers. Instead of handing out 16 addresses at random, we can hand out a block of *contiguous* class C addresses. Suppose we assign the class C network numbers from 192.4.16 through 192.4.31. Observe that the top 20 bits of all the addresses in this range are the same (`11000000 00000100 0001`). Thus, what we have effectively created is a 20-bit network number—something that is between a class B network number and a class C number in terms of the number of hosts that it can support. In other words, we get both the high address efficiency of handing out addresses in chunks smaller than a class B network, and a single network prefix that can be used in forwarding tables. Observe that, for this scheme to work, we need to hand out blocks of class C addresses that share a common prefix, which means that each block must contain a number of class C networks that is a power of two.

CIDR requires a new type of notation to represent network numbers, or *prefixes* as they are known, because the prefixes can be of any length. The convention is to place a `/X` after the prefix, where `X` is the prefix length in bits. So, for the example above, the 20-bit prefix for all the networks 192.4.16 through 192.4.31 is represented as 192.4.16/20. By contrast, if we wanted to represent a single class C network number, which is 24 bits long, we would write it 192.4.16/24. Today, with CIDR being the norm, it is more common to hear people talk about "slash 24" prefixes than class C networks. Note that representing a network address in this way is similar to the `(mask, value)` approach used in subnetting, as long as `masks` consist of contiguous bits starting from the most significant bit (which in practice is almost always the case).

The ability to aggregate routes at the edge of the network as we have just seen is only the first step. Imagine an Internet service provider network, whose primary job is to provide Internet connectivity to a large number of corporations and campuses (customers). If we assign prefixes to the customers in such a way that many different customer networks connected to the provider network share a common, shorter address prefix, then we can get even greater aggregation of routes. Consider the example in Figure 3.23. Assume that eight customers served by the provider network have each been assigned adjacent 24-bit network prefixes. Those prefixes all start with the same 21 bits. Since all of the customers are reachable through the same provider network, it can advertise a single route to all of them by just advertising the common 21-bit prefix they share. And it can do this even if not all the 24-bit prefixes have been handed out, as long as the provider ultimately *will* have the right to hand out those prefixes to a customer. One way to accomplish that is to assign a portion of address space to the provider in advance and then to let the network provider assign addresses from that space to its customers as needed. Note that, in contrast to this simple example, there is no need for all customer prefixes to be the same length.

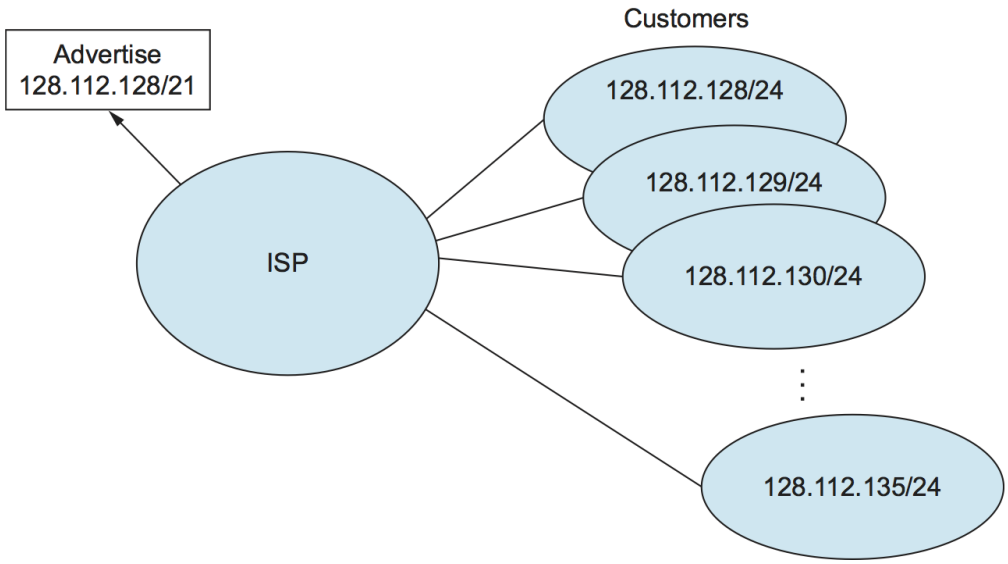


Figure 3.23.: Route aggregation with CIDR.

IP Forwarding Revisited

In all our discussion of IP forwarding so far, we have assumed that we could find the network number in a packet and then look up that number in a forwarding table. However, now that we have introduced CIDR, we need to reexamine this assumption. CIDR means that prefixes may be of any length, from 2 to 32 bits. Furthermore, it is sometimes possible to have prefixes in the forwarding table that "overlap," in the sense that some addresses may match more than one prefix. For example, we might find both 171.69 (a 16-bit prefix) and 171.69.10 (a 24-bit prefix) in the forwarding table of a single router. In this case, a packet destined to, say, 171.69.10.5 clearly matches both prefixes. The rule in this case is based on the principle of "longest match"; that is, the packet matches the longest prefix, which would be 171.69.10 in this example. On the other hand, a packet destined to 171.69.20.5 would match 171.69 and *not* 171.69.10, and in the absence of any other matching entry in the routing table 171.69 would be the longest match.

The task of efficiently finding the longest match between an IP address and the variable-length prefixes in a forwarding table has been a fruitful field of research for many years. The most well-known algorithm uses an approach known as a *PATRICIA tree*, which was actually developed well in advance of CIDR.

3.3.6 Address Translation (ARP)

In the previous section we talked about how to get IP datagrams to the right physical network but glossed over the issue of how to get a datagram to a particular host or router on that network. The main issue is that IP datagrams contain IP addresses, but the physical interface hardware on the host or router to which you want to send the datagram only understands the addressing scheme of that particular network. Thus, we need to translate the IP address to a link-level address that makes sense on this network (e.g., a 48-bit Ethernet address). We can then encapsulate the IP datagram inside a frame that contains that link-level address and send it either to the ultimate destination or to a router that promises to forward the datagram toward the ultimate destination.

One simple way to map an IP address into a physical network address is to encode a host's physical address

in the host part of its IP address. For example, a host with physical address `00100001 01001001` (which has the decimal value 33 in the upper byte and 81 in the lower byte) might be given the IP address `128.96.33.81`. While this solution has been used on some networks, it is limited in that the network's physical addresses can be no more than 16 bits long in this example; they can be only 8 bits long on a class C network. This clearly will not work for 48-bit Ethernet addresses.

A more general solution would be for each host to maintain a table of address pairs; that is, the table would map IP addresses into physical addresses. While this table could be centrally managed by a system administrator and then copied to each host on the network, a better approach would be for each host to dynamically learn the contents of the table using the network. This can be accomplished using the Address Resolution Protocol (ARP). The goal of ARP is to enable each host on a network to build up a table of mappings between IP addresses and link-level addresses. Since these mappings may change over time (e.g., because an Ethernet card in a host breaks and is replaced by a new one with a new address), the entries are timed out periodically and removed. This happens on the order of every 15 minutes. The set of mappings currently stored in a host is known as the ARP cache or ARP table.

ARP takes advantage of the fact that many link-level network technologies, such as Ethernet, support broadcast. If a host wants to send an IP datagram to a host (or router) that it knows to be on the same network (i.e., the sending and receiving nodes have the same IP network number), it first checks for a mapping in the cache. If no mapping is found, it needs to invoke the Address Resolution Protocol over the network. It does this by broadcasting an ARP query onto the network. This query contains the IP address in question (the target IP address). Each host receives the query and checks to see if it matches its IP address. If it does match, the host sends a response message that contains its link-layer address back to the originator of the query. The originator adds the information contained in this response to its ARP table.

The query message also includes the IP address and link-layer address of the sending host. Thus, when a host broadcasts a query message, each host on the network can learn the sender's link-level and IP addresses and place that information in its ARP table. However, not every host adds this information to its ARP table. If the host already has an entry for that host in its table, it "refreshes" this entry; that is, it resets the length of time until it discards the entry. If that host is the target of the query, then it adds the information about the sender to its table, even if it did not already have an entry for that host. This is because there is a good chance that the source host is about to send it an application-level message, and it may eventually have to send a response or ACK back to the source; it will need the source's physical address to do this. If a host is not the target and does not already have an entry for the source in its ARP table, then it does not add an entry for the source. This is because there is no reason to believe that this host will ever need the source's link-level address; there is no need to clutter its ARP table with this information.

Figure 3.24 shows the ARP packet format for IP-to-Ethernet address mappings. In fact, ARP can be used for lots of other kinds of mappings—the major differences are in the address sizes. In addition to the IP and link-layer addresses of both sender and target, the packet contains

- A `HardwareType` field, which specifies the type of physical network (e.g., Ethernet)
- A `ProtocolType` field, which specifies the higher-layer protocol (e.g., IP)
- `HLen` ("hardware" address length) and `PLen` ("protocol" address length) fields, which specify the length of the link-layer address and higher-layer protocol address, respectively
- An `Operation` field, which specifies whether this is a request or a response
- The source and target hardware (Ethernet) and protocol (IP) addresses

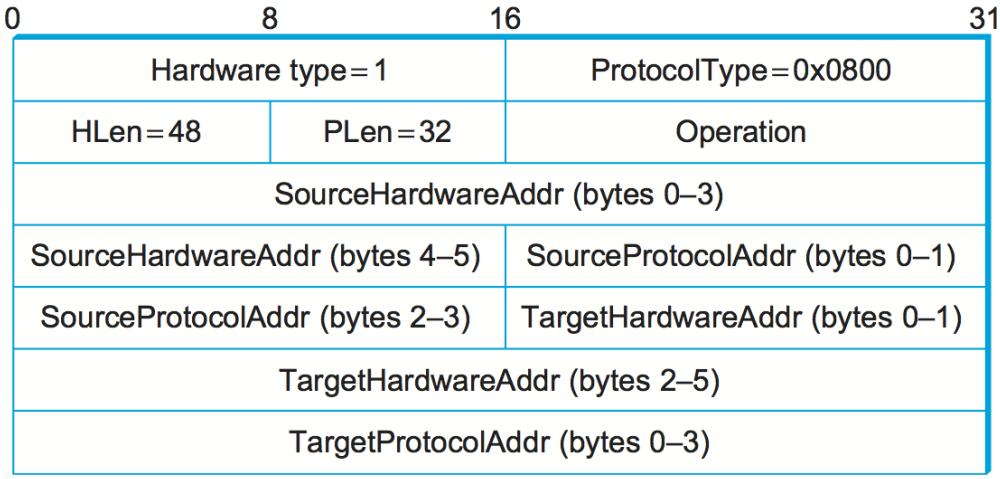


Figure 3.24.: ARP packet format for mapping IP addresses into Ethernet addresses.

Note that the results of the ARP process can be added as an extra column in a forwarding table like the one in Table 3.6. Thus, for example, when R2 needs to forward a packet to network 2, it not only finds that the next hop is R1, but also finds the MAC address to place on the packet to send it to R1.

Key Takeaway

We have now seen the basic mechanisms that IP provides for dealing with both heterogeneity and scale. On the issue of heterogeneity, IP begins by defining a best-effort service model that makes minimal assumptions about the underlying networks; most notably, this service model is based on unreliable datagrams. IP then makes two important additions to this starting point: (1) a common packet format (fragmentation/reassembly is the mechanism that makes this format work over networks with different MTUs) and (2) a global address space for identifying all hosts (ARP is the mechanism that makes this global address space work over networks with different physical addressing schemes). On the issue of scale, IP uses hierarchical aggregation to reduce the amount of information needed to forward packets. Specifically, IP addresses are partitioned into network and host components, with packets first routed toward the destination network and then delivered to the correct host on that network. *[Next]*

3.3.7 Host Configuration (DHCP)

Ethernet addresses are configured into the network adaptor by the manufacturer, and this process is managed in such a way to ensure that these addresses are globally unique. This is clearly a sufficient condition to ensure that any collection of hosts connected to a single Ethernet (including an extended LAN) will have unique addresses. Furthermore, uniqueness is all we ask of Ethernet addresses.

IP addresses, by contrast, not only must be unique on a given internetwork but also must reflect the structure of the internetwork. As noted above, they contain a network part and a host part, and the network part must be the same for all hosts on the same network. Thus, it is not possible for the IP address to be configured once into a host when it is manufactured, since that would imply that the manufacturer knew which hosts were going to end up on which networks, and it would mean that a host, once connected to one network, could never move to another. For this reason, IP addresses need to be reconfigurable.

In addition to an IP address, there are some other pieces of information a host needs to have before it can start sending packets. The most notable of these is the address of a default router—the place to which it can send packets whose destination address is not on the same network as the sending host.

Most host operating systems provide a way for a system administrator, or even a user, to manually configure the IP information needed by a host; however, there are some obvious drawbacks to such manual configuration. One is that it is simply a lot of work to configure all the hosts in a large network directly, especially when you consider that such hosts are not reachable over a network until they are configured. Even more importantly, the configuration process is very error prone, since it is necessary to ensure that every host gets the correct network number and that no two hosts receive the same IP address. For these reasons, automated configuration methods are required. The primary method uses a protocol known as the *Dynamic Host Configuration Protocol* (DHCP).

DHCP relies on the existence of a DHCP server that is responsible for providing configuration information to hosts. There is at least one DHCP server for an administrative domain. At the simplest level, the DHCP server can function just as a centralized repository for host configuration information. Consider, for example, the problem of administering addresses in the internetwork of a large company. DHCP saves the network administrators from having to walk around to every host in the company with a list of addresses and network map in hand and configuring each host manually. Instead, the configuration information for each host could be stored in the DHCP server and automatically retrieved by each host when it is booted or connected to the network. However, the administrator would still pick the address that each host is to receive; he would just store that in the server. In this model, the configuration information for each host is stored in a table that is indexed by some form of unique client identifier, typically the hardware address (e.g., the Ethernet address of its network adaptor).

A more sophisticated use of DHCP saves the network administrator from even having to assign addresses to individual hosts. In this model, the DHCP server maintains a pool of available addresses that it hands out to hosts on demand. This considerably reduces the amount of configuration an administrator must do, since now it is only necessary to allocate a range of IP addresses (all with the same network number) to each network.

Since the goal of DHCP is to minimize the amount of manual configuration required for a host to function, it would rather defeat the purpose if each host had to be configured with the address of a DHCP server. Thus, the first problem faced by DHCP is that of server discovery.

To contact a DHCP server, a newly booted or attached host sends a `DHCPDISCOVER` message to a special IP address (255.255.255.255) that is an IP broadcast address. This means it will be received by all hosts and routers on that network. (Routers do not forward such packets onto other networks, preventing broadcast to the entire Internet.) In the simplest case, one of these nodes is the DHCP server for the network. The server would then reply to the host that generated the discovery message (all the other nodes would ignore it). However, it is not really desirable to require one DHCP server on every network, because this still creates a potentially large number of servers that need to be correctly and consistently configured. Thus, DHCP uses the concept of a *relay agent*. There is at least one relay agent on each network, and it is configured with just one piece of information: the IP address of the DHCP server. When a relay agent receives a `DHCPDISCOVER` message, it unicasts it to the DHCP server and awaits the response, which it will then send back to the requesting client. The process of relaying a message from a host to a remote DHCP server is shown in Figure 3.25.

Figure 3.26 below shows the format of a DHCP message. The message is actually sent using a protocol called the *User Datagram Protocol* (UDP) that runs over IP. UDP is discussed in detail in the next chapter, but the only interesting thing it does in this context is to provide a demultiplexing key that says, "This is a

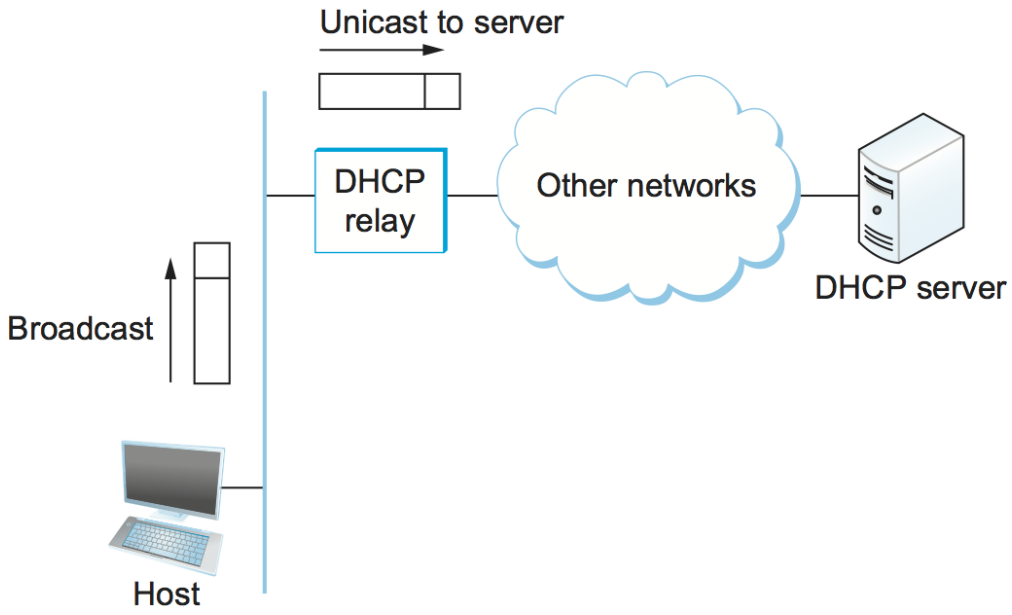


Figure 3.25.: A DHCP relay agent receives a broadcast DHCPDISCOVER message from a host and sends a unicast DHCPDISCOVER to the DHCP server.

DHCP packet."

DHCP is derived from an earlier protocol called BOOTP, and some of the packet fields are thus not strictly relevant to host configuration. When trying to obtain configuration information, the client puts its hardware address (e.g., its Ethernet address) in the `chaddr` field. The DHCP server replies by filling in the `yiaddr` ("your" IP address) field and sending it to the client. Other information such as the default router to be used by this client can be included in the `options` field.

In the case where DHCP dynamically assigns IP addresses to hosts, it is clear that hosts cannot keep addresses indefinitely, as this would eventually cause the server to exhaust its address pool. At the same time, a host cannot be depended upon to give back its address, since it might have crashed, been unplugged from the network, or been turned off. Thus, DHCP allows addresses to be leased for some period of time. Once the lease expires, the server is free to return that address to its pool. A host with a leased address clearly needs to renew the lease periodically if in fact it is still connected to the network and functioning correctly.

Key Takeaway

DHCP illustrates an important aspect of scaling: the scaling of network management. While discussions of scaling often focus on keeping the state in network devices from growing too fast, it is important to pay attention to the growth of network management complexity. By allowing network managers to configure a range of IP addresses per network rather than one IP address per host, DHCP improves the manageability of a network. *[Next]*

Note that DHCP may also introduce some more complexity into network management, since it makes the binding between physical hosts and IP addresses much more dynamic. This may make the network manager's job more difficult if, for example, it becomes necessary to locate a malfunctioning host.

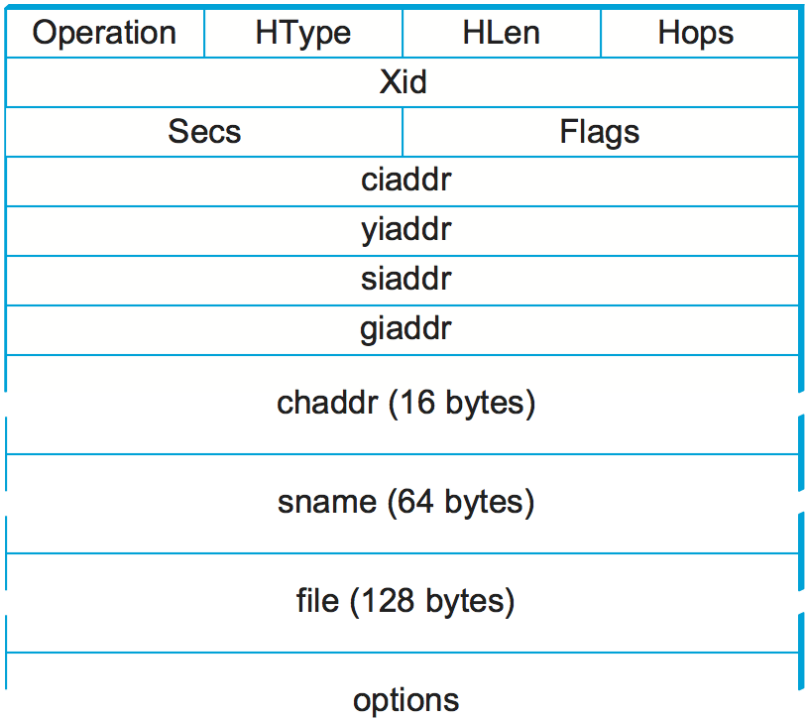


Figure 3.26.: DHCP packet format.

3.3.8 Error Reporting (ICMP)

The next issue is how the Internet treats errors. While IP is perfectly willing to drop datagrams when the going gets tough—for example, when a router does not know how to forward the datagram or when one fragment of a datagram fails to arrive at the destination—it does not necessarily fail silently. IP is always configured with a companion protocol, known as the *Internet Control Message Protocol* (ICMP), that defines a collection of error messages that are sent back to the source host whenever a router or host is unable to process an IP datagram successfully. For example, ICMP defines error messages indicating that the destination host is unreachable (perhaps due to a link failure), that the reassembly process failed, that the TTL had reached 0, that the IP header checksum failed, and so on.

ICMP also defines a handful of control messages that a router can send back to a source host. One of the most useful control messages, called an *ICMP-Redirect*, tells the source host that there is a better route to the destination. ICMP-Redirects are used in the following situation. Suppose a host is connected to a network that has two routers attached to it, called *R1* and *R2*, where the host uses R1 as its default router. Should R1 ever receive a datagram from the host, where based on its forwarding table it knows that R2 would have been a better choice for a particular destination address, it sends an ICMP-Redirect back to the host, instructing it to use R2 for all future datagrams addressed to that destination. The host then adds this new route to its forwarding table.

ICMP also provides the basis for two widely used debugging tools, `ping` and `traceroute`. `ping` uses ICMP echo messages to determine if a node is reachable and alive. `traceroute` uses a slightly non-intuitive technique to determine the set of routers along the path to a destination, which is the topic for one of the exercises at the end of this chapter.

3.3.9 Virtual Networks and Tunnels

We conclude our introduction to IP by considering an issue you might not have anticipated, but one that is increasingly important. Our discussion up to this point has focused on making it possible for nodes on different networks to communicate with each other in an unrestricted way. This is usually the goal in the Internet—everybody wants to be able to send email to everybody, and the creator of a new website wants to reach the widest possible audience. However, there are many situations where more controlled connectivity is required. An important example of such a situation is the *virtual private network* (VPN).

The term *VPN* is heavily overused and definitions vary, but intuitively we can define a VPN by considering first the idea of a private network. Corporations with many sites often build private networks by leasing circuits from the phone companies and using those lines to interconnect sites. In such a network, communication is restricted to take place only among the sites of that corporation, which is often desirable for security reasons. To make a private network *virtual*, the leased transmission lines—which are not shared with any other corporations—would be replaced by some sort of shared network. A virtual circuit (VC) is a very reasonable replacement for a leased line because it still provides a logical point-to-point connection between the corporation's sites. For example, if corporation X has a VC from site A to site B, then clearly it can send packets between sites A and B. But there is no way that corporation Y can get its packets delivered to site B without first establishing its own virtual circuit to site B, and the establishment of such a VC can be administratively prevented, thus preventing unwanted connectivity between corporation X and corporation Y.

Figure 3.27(a) shows two private networks for two separate corporations. In Figure 3.27(b) they are both migrated to a virtual circuit network. The limited connectivity of a real private network is maintained, but since the private networks now share the same transmission facilities and switches we say that two virtual private networks have been created.

In Figure 3.27, a virtual circuit network (using ATM, for example) is used to provide the controlled connectivity among sites. It is also possible to provide a similar function using an IP network to provide the connectivity. However, we cannot just connect the various corporations' sites to a single internetwork because that would provide connectivity between corporation X and corporation Y, which we wish to avoid. To solve this problem, we need to introduce a new concept, the *IP tunnel*.

We can think of an IP tunnel as a virtual point-to-point link between a pair of nodes that are actually separated by an arbitrary number of networks. The virtual link is created within the router at the entrance to the tunnel by providing it with the IP address of the router at the far end of the tunnel. Whenever the router at the entrance of the tunnel wants to send a packet over this virtual link, it encapsulates the packet inside an IP datagram. The destination address in the IP header is the address of the router at the far end of the tunnel, while the source address is that of the encapsulating router.

In the forwarding table of the router at the entrance to the tunnel, this virtual link looks much like a normal link. Consider, for example, the network in Figure 3.28. A tunnel has been configured from R1 to R2 and assigned a virtual interface number of 0. The forwarding table in R1 might therefore look like Table 3.9.

Table 3.9.: Forwarding Table for Router R1.

NetworkNum	NextHop
1	Interface 0
2	Virtual interface 0
Default	Interface 1

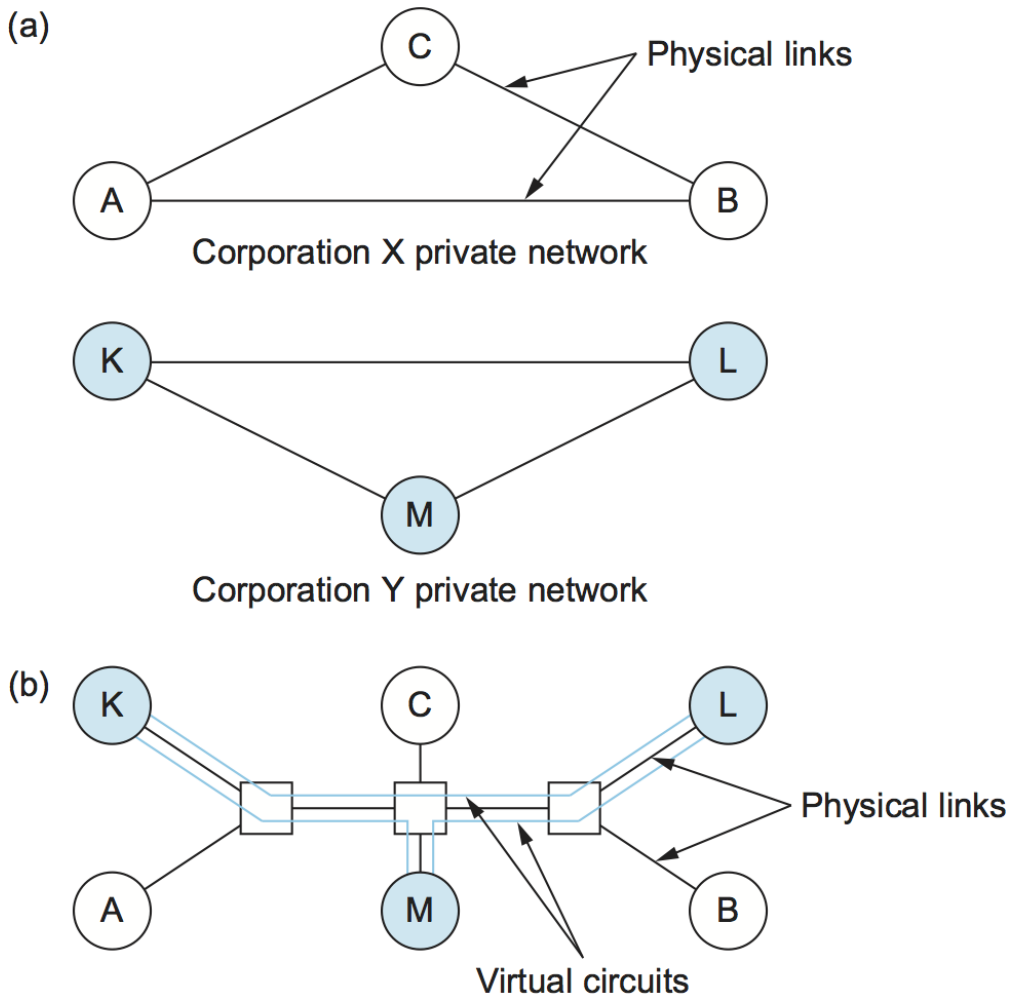


Figure 3.27.: An example of virtual private networks: (a) two separate private networks; (b) two virtual private networks sharing common switches.

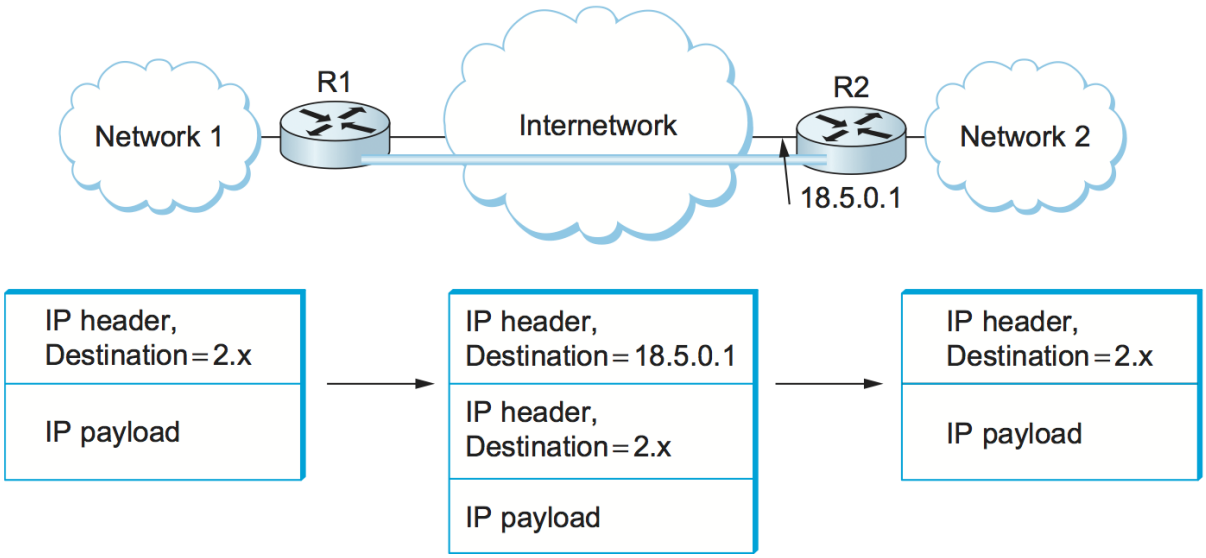


Figure 3.28.: A tunnel through an internetwork. 18.5.0.1 is the address of R2 that can be reached from R1 across the internetwork.

R1 has two physical interfaces. Interface 0 connects to network 1; interface 1 connects to a large internetwork and is thus the default for all traffic that does not match something more specific in the forwarding table. In addition, R1 has a virtual interface, which is the interface to the tunnel. Suppose R1 receives a packet from network 1 that contains an address in network 2. The forwarding table says this packet should be sent out virtual interface 0. In order to send a packet out this interface, the router takes the packet, adds an IP header addressed to R2, and then proceeds to forward the packet as if it had just been received. R2's address is 18.5.0.1; since the network number of this address is 18, not 1 or 2, a packet destined for R2 will be forwarded out the default interface into the internetwork.

Once the packet leaves R1, it looks to the rest of the world like a normal IP packet destined to R2, and it is forwarded accordingly. All the routers in the internetwork forward it using normal means, until it arrives at R2. When R2 receives the packet, it finds that it carries its own address, so it removes the IP header and looks at the payload of the packet. What it finds is an inner IP packet whose destination address is in network 2. R2 now processes this packet like any other IP packet it receives. Since R2 is directly connected to network 2, it forwards the packet on to that network. Figur 3.28 shows the change in encapsulation of the packet as it moves across the network.

While R2 is acting as the endpoint of the tunnel, there is nothing to prevent it from performing the normal functions of a router. For example, it might receive some packets that are not tunneled, but that are addressed to networks that it knows how to reach, and it would forward them in the normal way.

You might wonder why anyone would want to go to all the trouble of creating a tunnel and changing the encapsulation of a packet as it goes across an internetwork. One reason is security. Supplemented with encryption, a tunnel can become a very private sort of link across a public network. Another reason may be that R1 and R2 have some capabilities that are not widely available in the intervening networks, such as multicast routing. By connecting these routers with a tunnel, we can build a virtual network in which all the routers with this capability appear to be directly connected. A third reason to build tunnels is to carry packets from protocols other than IP across an IP network. As long as the routers at either end of the tunnel know how to handle these other protocols, the IP tunnel looks to them like a point-to-point link over which they can send non-IP packets. Tunnels also provide a mechanism by which we can force a packet to be

delivered to a particular place even if its original header—the one that gets encapsulated inside the tunnel header—might suggest that it should go somewhere else. Thus, we see that tunneling is a powerful and quite general technique for building virtual links across internetworks. So general, in fact, that the technique recurses, with the most common use case being to tunnel IP over IP.

Tunneling does have its downsides. One is that it increases the length of packets; this might represent a significant waste of bandwidth for short packets. Longer packets might be subject to fragmentation, which has its own set of drawbacks. There may also be performance implications for the routers at either end of the tunnel, since they need to do more work than normal forwarding as they add and remove the tunnel header. Finally, there is a management cost for the administrative entity that is responsible for setting up the tunnels and making sure they are correctly handled by the routing protocols.

3.4 Routing

So far in this chapter we have assumed that the switches and routers have enough knowledge of the network topology so they can choose the right port onto which each packet should be output. In the case of virtual circuits, routing is an issue only for the connection request packet; all subsequent packets follow the same path as the request. In datagram networks, including IP networks, routing is an issue for every packet. In either case, a switch or router needs to be able to look at a destination address and then to determine which of the output ports is the best choice to get a packet to that address. As we saw in an earlier section, the switch makes this decision by consulting a forwarding table. The fundamental problem of routing is how switches and routers acquire the information in their forwarding tables.

Key Takeaway

We restate an important distinction, which is often neglected, between *forwarding* and *routing*. Forwarding consists of receiving a packet, looking up its destination address in a table, and sending the packet in a direction determined by that table. We saw several examples of forwarding in the preceding section. It is a simple and well-defined process performed locally at each node, and is often referred to as the network's *data plane.* Routing is the process by which forwarding tables are built. It depends on complex distributed algorithms, and is often referred to as the network's *control plane. [Next]*

While the terms *forwarding table* and *routing table* are sometimes used interchangeably, we will make a distinction between them here. The forwarding table is used when a packet is being forwarded and so must contain enough information to accomplish the forwarding function. This means that a row in the forwarding table contains the mapping from a network prefix to an outgoing interface and some MAC information, such as the Ethernet address of the next hop. The routing table, on the other hand, is the table that is built up by the routing algorithms as a precursor to building the forwarding table. It generally contains mappings from network prefixes to next hops. It may also contain information about how this information was learned, so that the router will be able to decide when it should discard some information.

Whether the routing table and forwarding table are actually separate data structures is something of an implementation choice, but there are numerous reasons to keep them separate. For example, the forwarding table needs to be structured to optimize the process of looking up an address when forwarding a packet, while the routing table needs to be optimized for the purpose of calculating changes in topology. In many cases, the forwarding table may even be implemented in specialized hardware, whereas this is rarely if ever done for the routing table.

Table 3.10 gives an example of a row from a routing table, which tells us that network prefix 18/8 is to be reached by a next hop router with the IP address 171.69.245.10

Table 3.10.: Example row from a routing table.

Prefix/Length	Next Hop
18/8	171.69.245.10

In contrast, Table 3.11 gives an example of a row from a forwarding table, which contains the information about exactly how to forward a packet to that next hop: Send it out interface number 0 with a MAC address of 8:0:2b:e4:b:1:2. Note that the last piece of information is provided by the Address Resolution Protocol.

Table 3.11.: Example row from a forwarding table.

Prefix/Length	Interface	MAC Address
18/8	if0	8:0:2b:e4:b:1:2

Before getting into the details of routing, we need to remind ourselves of the key question we should be asking anytime we try to build a mechanism for the Internet: "Does this solution scale?" The answer for the algorithms and protocols described in this section is "not so much." They are designed for networks of fairly modest size—up to a few hundred nodes, in practice. However, the solutions we describe do serve as a building block for a hierarchical routing infrastructure that is used in the Internet today. Specifically, the protocols described in this section are collectively known as *intradomain* routing protocols, or *interior gateway protocols* (IGPs). To understand these terms, we need to define a routing *domain*. A good working definition is an internetwork in which all the routers are under the same administrative control (e.g., a single university campus, or the network of a single Internet Service Provider). The relevance of this definition will become apparent in the next chapter when we look at *interdomain* routing protocols. For now, the important thing to keep in mind is that we are considering the problem of routing in the context of small to midsized networks, not for a network the size of the Internet.

3.4.1 Network as a Graph

Routing is, in essence, a problem of graph theory. Figure 3.29 shows a graph representing a network. The nodes of the graph, labeled A through F, may be hosts, switches, routers, or networks. For our initial discussion, we will focus on the case where the nodes are routers. The edges of the graph correspond to the network links. Each edge has an associated *cost*, which gives some indication of the desirability of sending traffic over that link. A discussion of how edge costs are assigned is given in a later section.

Note that the example networks (graphs) used throughout this chapter have undirected edges that are assigned a single cost. This is actually a slight simplification. It is more accurate to make the edges directed, which typically means that there would be a pair of edges between each node—one flowing in each direction, and each with its own edge cost.

The basic problem of routing is to find the lowest-cost path between any two nodes, where the cost of a path equals the sum of the costs of all the edges that make up the path. For a simple network like the one in Figure 3.29, you could imagine just calculating all the shortest paths and loading them into some nonvolatile storage on each node. Such a static approach has several shortcomings:

- It does not deal with node or link failures.

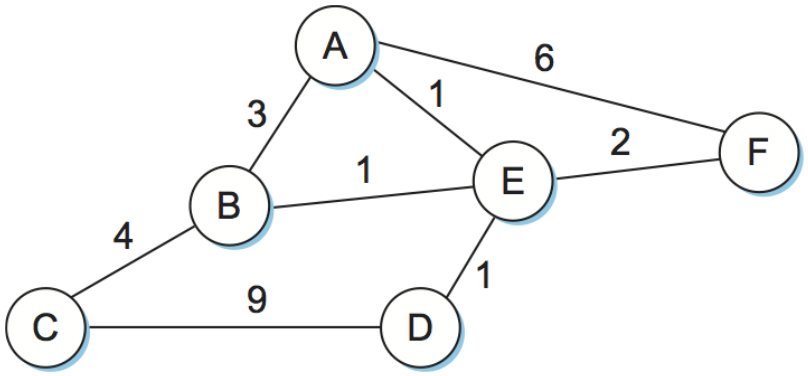


Figure 3.29.: Network represented as a graph.

- It does not consider the addition of new nodes or links.
- It implies that edge costs cannot change, even though we might reasonably wish to have link costs change over time (e.g., assigning high cost to a link that is heavily loaded).

For these reasons, routing is achieved in most practical networks by running routing protocols among the nodes. These protocols provide a distributed, dynamic way to solve the problem of finding the lowest-cost path in the presence of link and node failures and changing edge costs. Note the word *distributed* in the previous sentence; it is difficult to make centralized solutions scalable, so all the widely used routing protocols use distributed algorithms.

The distributed nature of routing algorithms is one of the main reasons why this has been such a rich field of research and development—there are a lot of challenges in making distributed algorithms work well. For example, distributed algorithms raise the possibility that two routers will at one instant have different ideas about the shortest path to some destination. In fact, each one may think that the other one is closer to the destination and decide to send packets to the other one. Clearly, such packets will be stuck in a loop until the discrepancy between the two routers is resolved, and it would be good to resolve it as soon as possible. This is just one example of the type of problem routing protocols must address.

To begin our analysis, we assume that the edge costs in the network are known. We will examine the two main classes of routing protocols: *distance vector* and *link state*. In a later section, we return to the problem of calculating edge costs in a meaningful way.

3.4.2 Distance-Vector (RIP)

The idea behind the distance-vector algorithm is suggested by its name. (The other common name for this class of algorithm is Bellman-Ford, after its inventors.) Each node constructs a one-dimensional array (a vector) containing the "distances" (costs) to all other nodes and distributes that vector to its immediate neighbors. The starting assumption for distance-vector routing is that each node knows the cost of the link to each of its directly connected neighbors. These costs may be provided when the router is configured by a network manager. A link that is down is assigned an infinite cost.

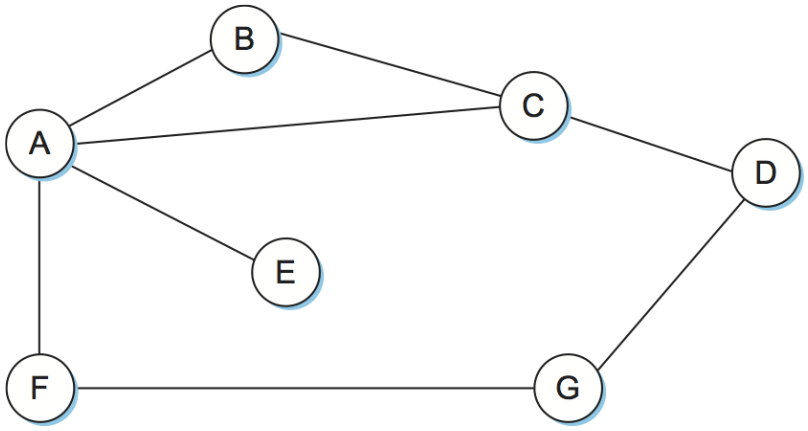


Figure 3.30.: Distance-vector routing: an example network.

Table 3.12.: Initial Distances Stored at Each Node (Global View).

	A	B	C	D	E	F	G
A	0	1	1	∞	1	1	∞
B	1	0	1	∞	∞	∞	∞
C	1	1	0	1	∞	∞	∞
D	∞	∞	1	0	∞	∞	1
E	1	∞	∞	∞	0	∞	∞
F	1	∞	∞	∞	∞	0	1
G	∞	∞	∞	1	∞	1	0

To see how a distance-vector routing algorithm works, it is easiest to consider an example like the one depicted in Figure 3.30. In this example, the cost of each link is set to 1, so that a least-cost path is simply the one with the fewest hops. (Since all edges have the same cost, we do not show the costs in the graph.) We can represent each node's knowledge about the distances to all other nodes as a table like Table 3.12. Note that each node knows only the information in one row of the table (the one that bears its name in the left column). The global view that is presented here is not available at any single point in the network.

We may consider each row in Table 3.12 as a list of distances from one node to all other nodes, representing the current beliefs of that node. Initially, each node sets a cost of 1 to its directly connected neighbors and ∞ to all other nodes. Thus, A initially believes that it can reach B in one hop and that D is unreachable. The routing table stored at A reflects this set of beliefs and includes the name of the next hop that A would use to reach any reachable node. Initially, then, A's routing table would look like Table 3.13.

Table 3.13.: Initial Routing Table at Node A.

Destination	Cost	NextHop
B	1	B
C	1	C
D	∞	—
E	1	E
F	1	F
G	∞	—

The next step in distance-vector routing is that every node sends a message to its directly connected neighbors containing its personal list of distances. For example, node F tells node A that it can reach node G at a cost of 1; A also knows it can reach F at a cost of 1, so it adds these costs to get the cost of reaching G by means of F. This total cost of 2 is less than the current cost of infinity, so A records that it can reach G at a cost of 2 by going through F. Similarly, A learns from C that D can be reached from C at a cost of 1; it adds this to the cost of reaching C (1) and decides that D can be reached via C at a cost of 2, which is better than the old cost of infinity. At the same time, A learns from C that B can be reached from C at a cost of 1, so it concludes that the cost of reaching B via C is 2. Since this is worse than the current cost of reaching B (1), this new information is ignored. At this point, A can update its routing table with costs and next hops for all nodes in the network. The result is shown in Table 3.14.

Table 3.14.: Final Routing Table at Node A.

Destination	Cost	NextHop
B	1	B
C	1	C
D	2	C
E	1	E
F	1	F
G	2	F

In the absence of any topology changes, it takes only a few exchanges of information between neighbors before each node has a complete routing table. The process of getting consistent routing information to all the nodes is called *convergence*. Table 3.15 shows the final set of costs from each node to all other nodes when routing has converged. We must stress that there is no one node in the network that has all the information in this table—each node only knows about the contents of its own routing table. The beauty of a distributed algorithm like this is that it enables all nodes to achieve a consistent view of the network in the absence of any centralized authority.

Table 3.15.: Final Distances Stored at Each Node (Global View).

	A	B	C	D	E	F	G
A	0	1	1	2	1	1	2
B	1	0	1	2	2	2	3
C	1	1	0	1	2	2	2
D	2	2	1	0	3	2	1
E	1	2	2	3	0	2	3
F	1	2	2	2	2	0	1
G	2	3	2	1	3	1	0

There are a few details to fill in before our discussion of distance-vector routing is complete. First we note that there are two different circumstances under which a given node decides to send a routing update to its neighbors. One of these circumstances is the *periodic* update. In this case, each node automatically sends an update message every so often, even if nothing has changed. This serves to let the other nodes know that this node is still running. It also makes sure that they keep getting information that they may need if their current routes become unviable. The frequency of these periodic updates varies from protocol to protocol, but it is typically on the order of several seconds to several minutes. The second mechanism, sometimes called a *triggered* update, happens whenever a node notices a link failure or receives an update from one of

its neighbors that causes it to change one of the routes in its routing table. Whenever a node's routing table changes, it sends an update to its neighbors, which may lead to a change in their tables, causing them to send an update to their neighbors.

Now consider what happens when a link or node fails. The nodes that notice first send new lists of distances to their neighbors, and normally the system settles down fairly quickly to a new state. As to the question of how a node detects a failure, there are a couple of different answers. In one approach, a node continually tests the link to another node by sending a control packet and seeing if it receives an acknowledgment. In another approach, a node determines that the link (or the node at the other end of the link) is down if it does not receive the expected periodic routing update for the last few update cycles.

To understand what happens when a node detects a link failure, consider what happens when F detects that its link to G has failed. First, F sets its new distance to G to infinity and passes that information along to A. Since A knows that its 2-hop path to G is through F, A would also set its distance to G to infinity. However, with the next update from C, A would learn that C has a 2-hop path to G. Thus, A would know that it could reach G in 3 hops through C, which is less than infinity, and so A would update its table accordingly. When it advertises this to F, node F would learn that it can reach G at a cost of 4 through A, which is less than infinity, and the system would again become stable.

Unfortunately, slightly different circumstances can prevent the network from stabilizing. Suppose, for example, that the link from A to E goes down. In the next round of updates, A advertises a distance of infinity to E, but B and C advertise a distance of 2 to E. Depending on the exact timing of events, the following might happen: Node B, upon hearing that E can be reached in 2 hops from C, concludes that it can reach E in 3 hops and advertises this to A; node A concludes that it can reach E in 4 hops and advertises this to C; node C concludes that it can reach E in 5 hops; and so on. This cycle stops only when the distances reach some number that is large enough to be considered infinite. In the meantime, none of the nodes actually knows that E is unreachable, and the routing tables for the network do not stabilize. This situation is known as the *count to infinity* problem.

There are several partial solutions to this problem. The first one is to use some relatively small number as an approximation of infinity. For example, we might decide that the maximum number of hops to get across a certain network is never going to be more than 16, and so we could pick 16 as the value that represents infinity. This at least bounds the amount of time that it takes to count to infinity. Of course, it could also present a problem if our network grew to a point where some nodes were separated by more than 16 hops.

One technique to improve the time to stabilize routing is called *split horizon*. The idea is that when a node sends a routing update to its neighbors, it does not send those routes it learned from each neighbor back to that neighbor. For example, if B has the route (E, 2, A) in its table, then it knows it must have learned this route from A, and so whenever B sends a routing update to A, it does not include the route (E, 2) in that update. In a stronger variation of split horizon, called *split horizon with poison reverse*, B actually sends that route back to A, but it puts negative information in the route to ensure that A will not eventually use B to get to E. For example, B sends the route (E, ∞) to A. The problem with both of these techniques is that they only work for routing loops that involve two nodes. For larger routing loops, more drastic measures are called for. Continuing the above example, if B and C had waited for a while after hearing of the link failure from A before advertising routes to E, they would have found that neither of them really had a route to E. Unfortunately, this approach delays the convergence of the protocol; speed of convergence is one of the key advantages of its competitor, link-state routing, the subject of a later section.

Implementation

The code that implements this algorithm is very straightforward; we give only some of the basics here. Structure `Route` defines each entry in the routing table, and constant `MAX_TTL` specifies how long an entry is kept in the table before it is discarded.

```
#define MAX_ROUTES      128     /* maximum size of routing table */
#define MAX_TTL         120     /* time (in seconds) until route expires */

typedef struct {
    NodeAddr  Destination;    /* address of destination */
    NodeAddr  NextHop;        /* address of next hop */
    int        Cost;          /* distance metric */
    u_short   TTL;            /* time to live */
} Route;

int       numRoutes = 0;
Route     routingTable[MAX_ROUTES];
```

The routine that updates the local node's routing table based on a new route is given by `mergeRoute`. Although not shown, a timer function periodically scans the list of routes in the node's routing table, decrements the `TTL` (time to live) field of each route, and discards any routes that have a time to live of 0. Notice, however, that the `TTL` field is reset to `MAX_TTL` any time the route is reconfirmed by an update message from a neighboring node.

```
void
mergeRoute (Route *new)
{
    int i;

    for (i = 0; i < numRoutes; ++i)
    {
        if (new->Destination == routingTable[i].Destination)
        {
            if (new->Cost + 1 < routingTable[i].Cost)
            {
                /* found a better route: */
                break;
            } else if (new->NextHop == routingTable[i].NextHop) {
                /* metric for current next-hop may have changed: */
                break;
            } else {
                /* route is uninteresting---just ignore it */
                return;
            }
        }
    }
    if (i == numRoutes)
    {
        /* this is a completely new route; is there room for it? */
        if (numRoutes < MAXROUTES)
        {
```

(continues on next page)

(continued from previous page)

```
            ++numRoutes;
        } else {
            /* can`t fit this route in table so give up */
            return;
        }
    }
    routingTable[i] = *new;
    /* reset TTL */
    routingTable[i].TTL = MAX_TTL;
    /* account for hop to get to next node */
    ++routingTable[i].Cost;
}
```

Finally, the procedure `updateRoutingTable` is the main routine that calls `mergeRoute` to incorporate all the routes contained in a routing update that is received from a neighboring node.

```
void
updateRoutingTable (Route *newRoute, int numNewRoutes)
{
    int i;

    for (i=0; i < numNewRoutes; ++i)
    {
        mergeRoute(&newRoute[i]);
    }
}
```

Routing Information Protocol (RIP)

One of the more widely used routing protocols in IP networks is the Routing Information Protocol (RIP). Its widespread use in the early days of IP was due in no small part to the fact that it was distributed along with the popular Berkeley Software Distribution (BSD) version of Unix, from which many commercial versions of Unix were derived. It is also extremely simple. RIP is the canonical example of a routing protocol built on the distance-vector algorithm just described.

Routing protocols in internetworks differ very slightly from the idealized graph model described above. In an internetwork, the goal of the routers is to learn how to forward packets to various *networks*. Thus, rather than advertising the cost of reaching other routers, the routers advertise the cost of reaching networks. For example, in Figure 3.31, router C would advertise to router A the fact that it can reach networks 2 and 3 (to which it is directly connected) at a cost of 0, networks 5 and 6 at cost 1, and network 4 at cost 2.

We can see evidence of this in the RIP (version 2) packet format in Figure 3.32. The majority of the packet is taken up with `(address, mask, distance)` triples. However, the principles of the routing algorithm are just the same. For example, if router A learns from router B that network X can be reached at a lower cost via B than via the existing next hop in the routing table, A updates the cost and next hop information for the network number accordingly.

RIP is in fact a fairly straightforward implementation of distance-vector routing. Routers running RIP send their advertisements every 30 seconds; a router also sends an update message whenever an update from another router causes it to change its routing table. One point of interest is that it supports multiple address

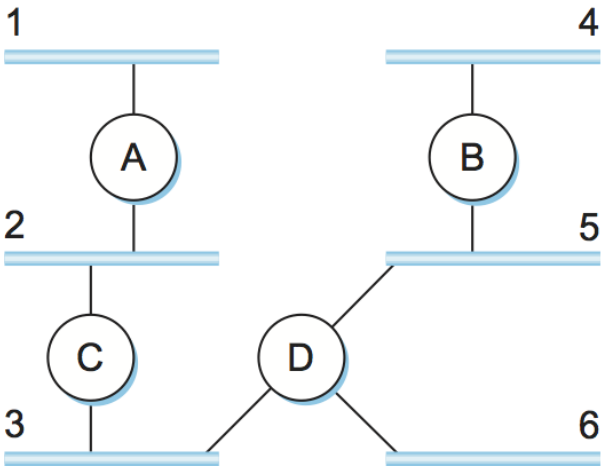


Figure 3.31.: Example network running RIP.

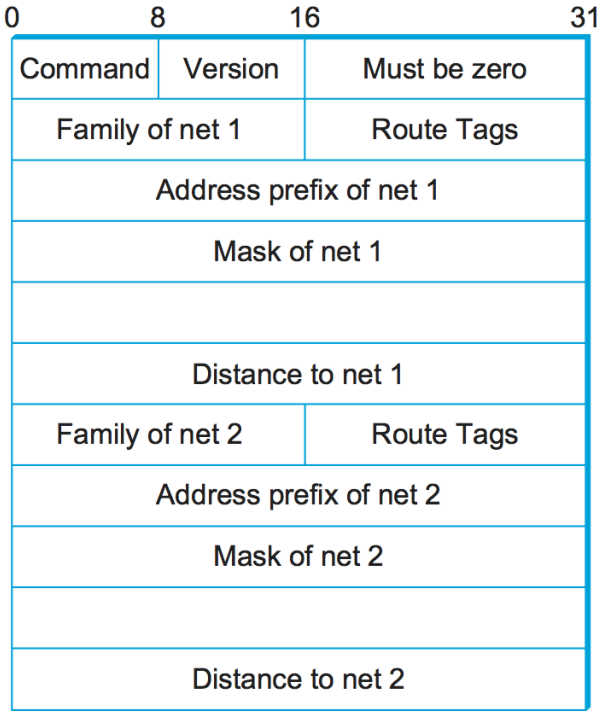


Figure 3.32.: RIPv2 packet format.

families, not just IP—that is the reason for the `Family` part of the advertisements. RIP version 2 (RIPv2) also introduced the subnet masks described in an earlier section, whereas RIP version 1 worked with the old classful addresses of IP.

As we will see below, it is possible to use a range of different metrics or costs for the links in a routing protocol. RIP takes the simplest approach, with all link costs being equal to 1, just as in our example above. Thus, it always tries to find the minimum hop route. Valid distances are 1 through 15, with 16 representing infinity. This also limits RIP to running on fairly small networks—those with no paths longer than 15 hops.

3.4.3 Link State (OSPF)

Link-state routing is the second major class of intradomain routing protocol. The starting assumptions for link-state routing are rather similar to those for distance-vector routing. Each node is assumed to be capable of finding out the state of the link to its neighbors (up or down) and the cost of each link. Again, we want to provide each node with enough information to enable it to find the least-cost path to any destination. The basic idea behind link-state protocols is very simple: Every node knows how to reach its directly connected neighbors, and if we make sure that the totality of this knowledge is disseminated to every node, then every node will have enough knowledge of the network to build a complete map of the network. This is clearly a sufficient condition (although not a necessary one) for finding the shortest path to any point in the network. Thus, link-state routing protocols rely on two mechanisms: reliable dissemination of link-state information, and the calculation of routes from the sum of all the accumulated link-state knowledge.

Reliable Flooding

Reliable flooding is the process of making sure that all the nodes participating in the routing protocol get a copy of the link-state information from all the other nodes. As the term *flooding* suggests, the basic idea is for a node to send its link-state information out on all of its directly connected links; each node that receives this information then forwards it out on all of *its* links. This process continues until the information has reached all the nodes in the network.

More precisely, each node creates an update packet, also called a *link-state packet* (LSP), which contains the following information:

- The ID of the node that created the LSP
- A list of directly connected neighbors of that node, with the cost of the link to each one
- A sequence number
- A time to live for this packet

The first two items are needed to enable route calculation; the last two are used to make the process of flooding the packet to all nodes reliable. Reliability includes making sure that you have the most recent copy of the information, since there may be multiple, contradictory LSPs from one node traversing the network. Making the flooding reliable has proven to be quite difficult. (For example, an early version of link-state routing used in the ARPANET caused that network to fail in 1981.)

Flooding works in the following way. First, the transmission of LSPs between adjacent routers is made reliable using acknowledgments and retransmissions just as in the reliable link-layer protocol. However, several more steps are necessary to reliably flood an LSP to all nodes in a network.

Consider a node X that receives a copy of an LSP that originated at some other node Y. Note that Y may be any other router in the same routing domain as X. X checks to see if it has already stored a copy of an LSP from Y. If not, it stores the LSP. If it already has a copy, it compares the sequence numbers; if the new LSP has a larger sequence number, it is assumed to be the more recent, and that LSP is stored, replacing the old one. A smaller (or equal) sequence number would imply an LSP older (or not newer) than the one stored, so it would be discarded and no further action would be needed. If the received LSP was the newer one, X then sends a copy of that LSP to all of its neighbors except the neighbor from which the LSP was just received. The fact that the LSP is not sent back to the node from which it was received helps to bring an end to the flooding of an LSP. Since X passes the LSP on to all its neighbors, who then turn around and do the same thing, the most recent copy of the LSP eventually reaches all nodes.

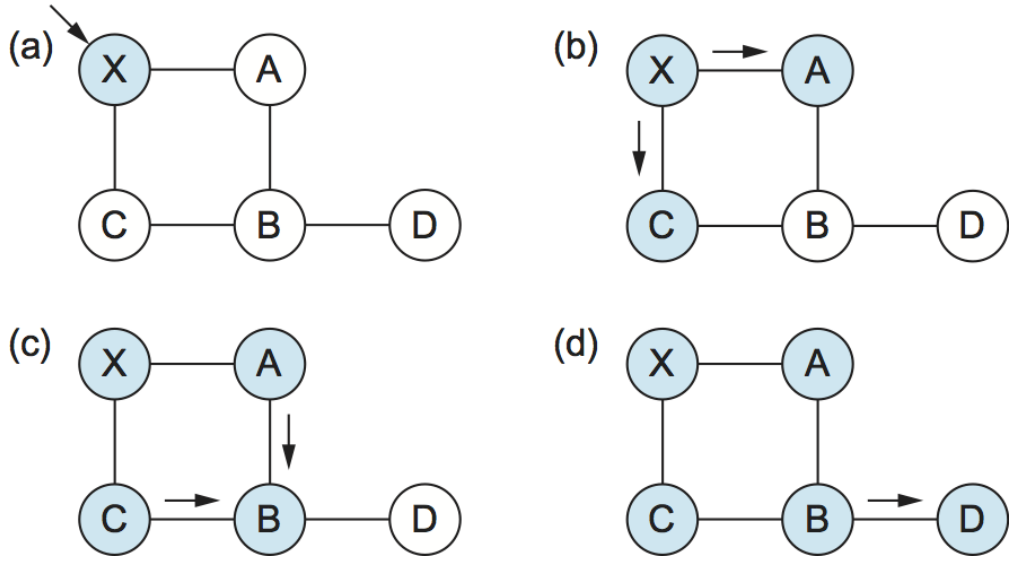


Figure 3.33.: Flooding of link-state packets: (a) LSP arrives at node X; (b) X floods LSP to A and C; (c) A and C flood LSP to B (but not X); (d) flooding is complete.

Figure 3.33 shows an LSP being flooded in a small network. Each node becomes shaded as it stores the new LSP. In Figure 3.33(a) the LSP arrives at node X, which sends it to neighbors A and C in Figure 3.33(b). A and C do not send it back to X, but send it on to B. Since B receives two identical copies of the LSP, it will accept whichever arrived first and ignore the second as a duplicate. It then passes the LSP onto D, which has no neighbors to flood it to, and the process is complete.

Just as in RIP, each node generates LSPs under two circumstances. Either the expiry of a periodic timer or a change in topology can cause a node to generate a new LSP. However, the only topology-based reason for a node to generate an LSP is if one of its directly connected links or immediate neighbors has gone down. The failure of a link can be detected in some cases by the link-layer protocol. The demise of a neighbor or loss of connectivity to that neighbor can be detected using periodic "hello" packets. Each node sends these to its immediate neighbors at defined intervals. If a sufficiently long time passes without receipt of a "hello" from a neighbor, the link to that neighbor will be declared down, and a new LSP will be generated to reflect this fact.

One of the important design goals of a link-state protocol's flooding mechanism is that the newest information must be flooded to all nodes as quickly as possible, while old information must be removed from the network and not allowed to circulate. In addition, it is clearly desirable to minimize the total amount of routing traffic that is sent around the network; after all, this is just overhead from the perspective of those

who actually use the network for their applications. The next few paragraphs describe some of the ways that these goals are accomplished.

One easy way to reduce overhead is to avoid generating LSPs unless absolutely necessary. This can be done by using very long timers—often on the order of hours—for the periodic generation of LSPs. Given that the flooding protocol is truly reliable when topology changes, it is safe to assume that messages saying "nothing has changed" do not need to be sent very often.

To make sure that old information is replaced by newer information, LSPs carry sequence numbers. Each time a node generates a new LSP, it increments the sequence number by 1. Unlike most sequence numbers used in protocols, these sequence numbers are not expected to wrap, so the field needs to be quite large (say, 64 bits). If a node goes down and then comes back up, it starts with a sequence number of 0. If the node was down for a long time, all the old LSPs for that node will have timed out (as described below); otherwise, this node will eventually receive a copy of its own LSP with a higher sequence number, which it can then increment and use as its own sequence number. This will ensure that its new LSP replaces any of its old LSPs left over from before the node went down.

LSPs also carry a time to live. This is used to ensure that old link-state information is eventually removed from the network. A node always decrements the TTL of a newly received LSP before flooding it to its neighbors. It also "ages" the LSP while it is stored in the node. When the TTL reaches 0, the node refloods the LSP with a TTL of 0, which is interpreted by all the nodes in the network as a signal to delete that LSP.

Route Calculation

Once a given node has a copy of the LSP from every other node, it is able to compute a complete map for the topology of the network, and from this map it is able to decide the best route to each destination. The question, then, is exactly how it calculates routes from this information. The solution is based on a well-known algorithm from graph theory—Dijkstra's shortest-path algorithm.

We first define Dijkstra's algorithm in graph-theoretic terms. Imagine that a node takes all the LSPs it has received and constructs a graphical representation of the network, in which N denotes the set of nodes in the graph, l(i,j) denotes the nonnegative cost (weight) associated with the edge between nodes i, j in N and l(i, j) = ∞ if no edge connects i and j. In the following description, we let s in N denote this node, that is, the node executing the algorithm to find the shortest path to all the other nodes in N. Also, the algorithm maintains the following two variables: M denotes the set of nodes incorporated so far by the algorithm, and C(n) denotes the cost of the path from s to each node n. Given these definitions, the algorithm is defined as follows:

```
M = {s}
for each n in N - {s}
    C(n) = l(s,n)
while (N != M)
    M = M + {w} such that C(w) is the minimum for all w in (N-M)
    for each n in (N-M)
    C(n) = MIN(C(n), C(w)+l(w,n))
```

Basically, the algorithm works as follows. We start with M containing this node s and then initialize the table of costs (the array `C(n)`) to other nodes using the known costs to directly connected nodes. We then look for the node that is reachable at the lowest cost (w) and add it to M. Finally, we update the table of costs by considering the cost of reaching nodes through w. In the last line of the algorithm, we choose a new

route to node n that goes through node w if the total cost of going from the source to w and then following the link from w to n is less than the old route we had to n. This procedure is repeated until all nodes are incorporated in M.

In practice, each switch computes its routing table directly from the LSPs it has collected using a realization of Dijkstra's algorithm called the *forward search* algorithm. Specifically, each switch maintains two lists, known as `Tentative` and `Confirmed`. Each of these lists contains a set of entries of the form `(Destination, Cost, NextHop)`. The algorithm works as follows:

1. Initialize the `Confirmed` list with an entry for myself; this entry has a cost of 0.
2. For the node just added to the `Confirmed` list in the previous step, call it node `Next` and select its LSP.
3. For each neighbor (`Neighbor`) of `Next`, calculate the cost (`Cost`) to reach this `Neighbor` as the sum of the cost from myself to `Next` and from `Next` to `Neighbor`.
    1. If `Neighbor` is currently on neither the `Confirmed` nor the `Tentative` list, then add `(Neighbor, Cost, NextHop)` to the `Tentative` list, where `NextHop` is the direction I go to reach `Next`.
    2. If `Neighbor` is currently on the `Tentative` list, and the `Cost` is less than the currently listed cost for `Neighbor`, then replace the current entry with `(Neighbor, Cost, NextHop)`, where `NextHop` is the direction I go to reach `Next`.
4. If the `Tentative` list is empty, stop. Otherwise, pick the entry from the `Tentative` list with the lowest cost, move it to the `Confirmed` list, and return to step 2.

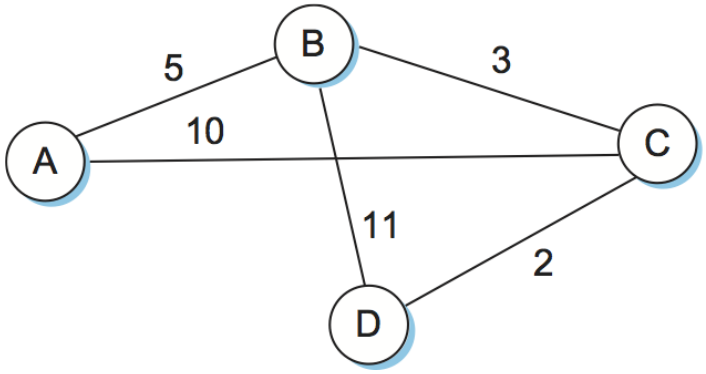


Figure 3.34.: Link-state routing: an example network.

This will become a lot easier to understand when we look at an example. Consider the network depicted in Figure 3.34. Note that, unlike our previous example, this network has a range of different edge costs. Table 3.16 traces the steps for building the routing table for node D. We denote the two outputs of D by using the names of the nodes to which they connect, B and C. Note the way the algorithm seems to head off on false leads (like the 11-unit cost path to B that was the first addition to the `Tentative` list) but ends up with the least-cost paths to all nodes.

Table 3.16.: Steps for Building Routing Table for Node D. :align: center :widths: auto

Step	Confirmed	Tentative	Comments
1	(D,0,–)		Since D is the only new member of the confirmed list, look at its LSP.
2	(D,0,–)	(B,11,B) (C,2,C)	D's LSP says we can reach B through B at cost 11, which is better than anything else on either list, so put it on `Tentative` list; same for C.
3	(D,0,–) (C,2,C)	(B,11,B)	Put lowest-cost member of `Tentative` (C) onto `Confirmed` list. Next, examine LSP of newly confirmed member (C).
4	(D,0,–) (C,2,C)	(B,5,C) (A,12,C)	Cost to reach B through C is 5, so replace (B,11,B). C's LSP tells us that we can reach A at cost 12.
5	(D,0,–) (C,2,C) (B,5,C)	(A,12,C)	Move lowest-cost member of `Tentative` (B) to `Confirmed`, then look at its LSP.
6	(D,0,–) (C,2,C) (B,5,C)	(A,10,C)	Since we can reach A at cost 5 through B, replace the `Tentative` entry.
7	(D,0,–) (C,2,C) (B,5,C) (A,10,C)		Move lowest-cost member of `Tentative` (A) to `Confirmed`, and we are all done.

The link-state routing algorithm has many nice properties: It has been proven to stabilize quickly, it does not generate much traffic, and it responds rapidly to topology changes or node failures. On the downside, the amount of information stored at each node (one LSP for every other node in the network) can be quite large. This is one of the fundamental problems of routing and is an instance of the more general problem of scalability. Some solutions to both the specific problem (the amount of storage potentially required at each node) and the general problem (scalability) will be discussed in the next section.

Key Takeaway

Distance-vector and link-state are both distributed routing algorithms, but they adopt different strategies. In distance-vector, each node talks only to its directly connected neighbors, but it tells them everything it has learned (i.e., distance to all nodes). In link-state, each node talks to all other nodes, but it tells them only what it knows for sure (i.e., only the state of its directly connected links). In contrast to both of these algorithms, we will consider a more centralized approach to routing in *Section 3.5* when we introduce Software Defined Networking (SDN). *[Next]*

The Open Shortest Path First Protocol (OSPF)

One of the most widely used link-state routing protocols is OSPF. The first word, "Open," refers to the fact that it is an open, nonproprietary standard, created under the auspices of the Internet Engineering Task Force (IETF). The "SPF" part comes from an alternative name for link-state routing. OSPF adds quite a number of features to the basic link-state algorithm described above, including the following:

- *Authentication of routing messages*—One feature of distributed routing algorithms is that they dis-

perse information from one node to many other nodes, and the entire network can thus be impacted by bad information from one node. For this reason, it's a good idea to be sure that all the nodes taking part in the protocol can be trusted. Authenticating routing messages helps achieve this. Early versions of OSPF used a simple 8-byte password for authentication. This is not a strong enough form of authentication to prevent dedicated malicious users, but it alleviates some problems caused by misconfiguration or casual attacks. (A similar form of authentication was added to RIP in version 2.) Strong cryptographic authentication was later added.

- *Additional hierarchy*—Hierarchy is one of the fundamental tools used to make systems more scalable. OSPF introduces another layer of hierarchy into routing by allowing a domain to be partitioned into *areas*. This means that a router within a domain does not necessarily need to know how to reach every network within that domain—it may be able to get by knowing only how to get to the right area. Thus, there is a reduction in the amount of information that must be transmitted to and stored in each node.
- *Load balancing*—OSPF allows multiple routes to the same place to be assigned the same cost and will cause traffic to be distributed evenly over those routes, thus making better use of the available network capacity.

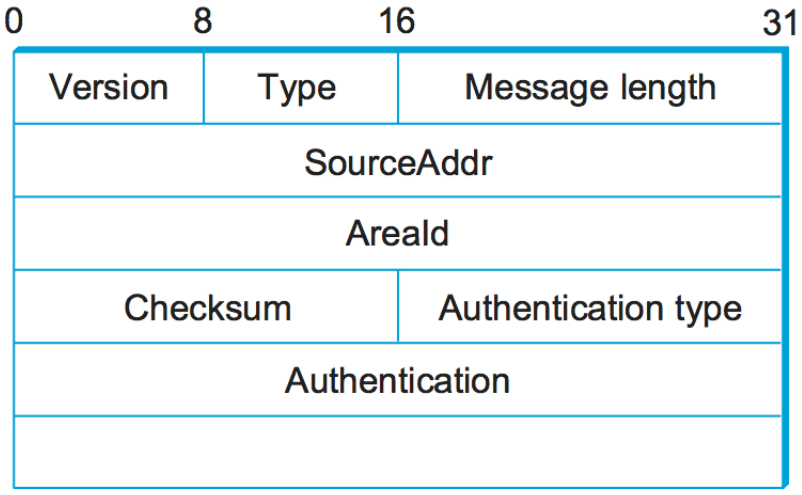


Figure 3.35.: OSPF header format.

There are several different types of OSPF messages, but all begin with the same header, as shown in Figure 3.35. The `Version` field is currently set to 2, and the `Type` field may take the values 1 through 5. The `SourceAddr` identifies the sender of the message, and the `AreaId` is a 32-bit identifier of the area in which the node is located. The entire packet, except the authentication data, is protected by a 16-bit checksum using the same algorithm as the IP header. The `Authentication type` is 0 if no authentication is used; otherwise, it may be 1, implying that a simple password is used, or 2, which indicates that a cryptographic authentication checksum is used. In the latter cases, the `Authentication` field carries the password or cryptographic checksum.

Of the five OSPF message types, type 1 is the "hello" message, which a router sends to its peers to notify them that it is still alive and connected as described above. The remaining types are used to request, send, and acknowledge the receipt of link-state messages. The basic building block of link-state messages in OSPF is the link-state advertisement (LSA). One message may contain many LSAs. We provide a few details of the LSA here.

Like any internetwork routing protocol, OSPF must provide information about how to reach networks. Thus,

OSPF must provide a little more information than the simple graph-based protocol described above. Specifically, a router running OSPF may generate link-state packets that advertise one or more of the networks that are directly connected to that router. In addition, a router that is connected to another router by some link must advertise the cost of reaching that router over the link. These two types of advertisements are necessary to enable all the routers in a domain to determine the cost of reaching all networks in that domain and the appropriate next hop for each network.

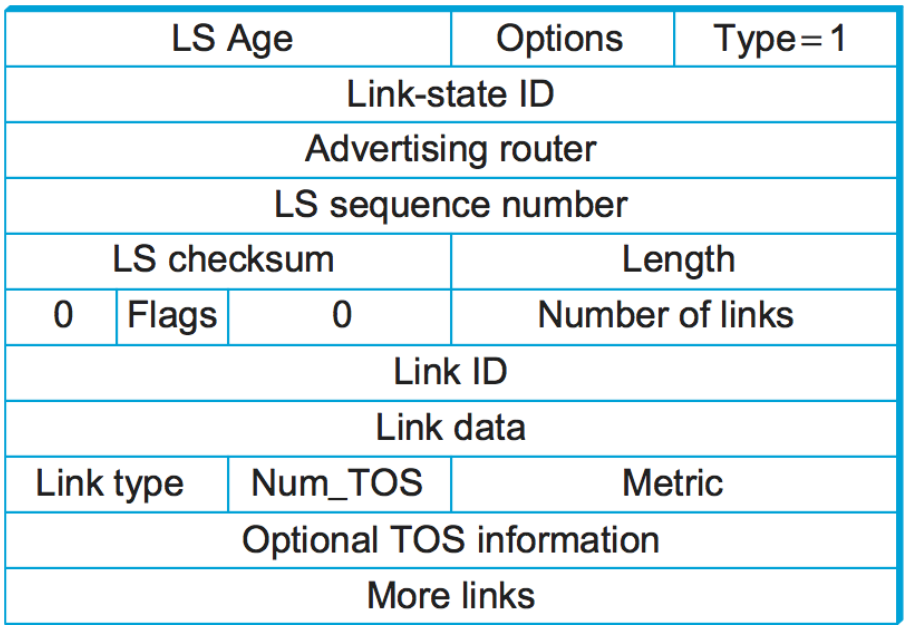


Figure 3.36.: OSPF link-state advertisement.

Figure 3.36 shows the packet format for a type 1 link-state advertisement. Type 1 LSAs advertise the cost of links between routers. Type 2 LSAs are used to advertise networks to which the advertising router is connected, while other types are used to support additional hierarchy as described in the next section. Many fields in the LSA should be familiar from the preceding discussion. The `LS Age` is the equivalent of a time to live, except that it counts up and the LSA expires when the age reaches a defined maximum value. The `Type` field tells us that this is a type 1 LSA.

In a type 1 LSA, the `Link state ID` and the `Advertising router` field are identical. Each carries a 32-bit identifier for the router that created this LSA. While a number of assignment strategies may be used to assign this ID, it is essential that it be unique in the routing domain and that a given router consistently uses the same router ID. One way to pick a router ID that meets these requirements would be to pick the lowest IP address among all the IP addresses assigned to that router. (Recall that a router may have a different IP address on each of its interfaces.)

The `LS sequence number` is used exactly as described above to detect old or duplicate LSAs. The `LS checksum` is similar to others we have seen in other protocols; it is, of course, used to verify that data has not been corrupted. It covers all fields in the packet except `LS Age`, so it is not necessary to recompute a checksum every time `LS Age` is incremented. `Length` is the length in bytes of the complete LSA.

Now we get to the actual link-state information. This is made a little complicated by the presence of TOS (type of service) information. Ignoring that for a moment, each link in the LSA is represented by a `Link ID`, some `Link Data`, and a `metric`. The first two of these fields identify the link; a common

way to do this would be to use the router ID of the router at the far end of the link as the `Link ID` and then use the `Link Data` to disambiguate among multiple parallel links if necessary. The `metric` is of course the cost of the link. `Type` tells us something about the link—for example, if it is a point-to-point link.

The TOS information is present to allow OSPF to choose different routes for IP packets based on the value in their TOS field. Instead of assigning a single metric to a link, it is possible to assign different metrics depending on the TOS value of the data. For example, if we had a link in our network that was very good for delay-sensitive traffic, we could give it a low metric for the TOS value representing low delay and a high metric for everything else. OSPF would then pick a different shortest path for those packets that had their TOS field set to that value. It is worth noting that, at the time of writing, this capability has not been widely deployed.

3.4.4 Metrics

The preceding discussion assumes that link costs, or metrics, are known when we execute the routing algorithm. In this section, we look at some ways to calculate link costs that have proven effective in practice. One example that we have seen already, which is quite reasonable and very simple, is to assign a cost of 1 to all links—the least-cost route will then be the one with the fewest hops. Such an approach has several drawbacks, however. First, it does not distinguish between links on a latency basis. Thus, a satellite link with 250-ms latency looks just as attractive to the routing protocol as a terrestrial link with 1-ms latency. Second, it does not distinguish between routes on a capacity basis, making a 1-Mbps link look just as good as a 10-Gbps link. Finally, it does not distinguish between links based on their current load, making it impossible to route around overloaded links. It turns out that this last problem is the hardest because you are trying to capture the complex and dynamic characteristics of a link in a single scalar cost.

The ARPANET was the testing ground for a number of different approaches to link-cost calculation. (It was also the place where the superior stability of link-state over distance-vector routing was demonstrated; the original mechanism used distance vector while the later version used link state.) The following discussion traces the evolution of the ARPANET routing metric and, in so doing, explores the subtle aspects of the problem.

The original ARPANET routing metric measured the number of packets that were queued waiting to be transmitted on each link, meaning that a link with 10 packets queued waiting to be transmitted was assigned a larger cost weight than a link with 5 packets queued for transmission. Using queue length as a routing metric did not work well, however, since queue length is an artificial measure of load—it moves packets toward the shortest queue rather than toward the destination, a situation all too familiar to those of us who hop from line to line at the grocery store. Stated more precisely, the original ARPANET routing mechanism suffered from the fact that it did not take either the bandwidth or the latency of the link into consideration.

A second version of the ARPANET routing algorithm took both link bandwidth and latency into consideration and used delay, rather than just queue length, as a measure of load. This was done as follows. First, each incoming packet was timestamped with its time of arrival at the router (`ArrivalTime`); its departure time from the router (`DepartTime`) was also recorded. Second, when the link-level ACK was received from the other side, the node computed the delay for that packet as

```
Delay = (DepartTime - ArrivalTime) + TransmissionTime + Latency
```

where `TransmissionTime` and `Latency` were statically defined for the link and captured the link's bandwidth and latency, respectively. Notice that in this case, `DepartTime - ArrivalTime` represents

the amount of time the packet was delayed (queued) in the node due to load. If the ACK did not arrive, but instead the packet timed out, then `DepartTime` was reset to the time the packet was *retransmitted*. In this case, `DepartTime - ArrivalTime` captures the reliability of the link—the more frequent the retransmission of packets, the less reliable the link, and the more we want to avoid it. Finally, the weight assigned to each link was derived from the average delay experienced by the packets recently sent over that link.

Although an improvement over the original mechanism, this approach also had a lot of problems. Under light load, it worked reasonably well, since the two static factors of delay dominated the cost. Under heavy load, however, a congested link would start to advertise a very high cost. This caused all the traffic to move off that link, leaving it idle, so then it would advertise a low cost, thereby attracting back all the traffic, and so on. The effect of this instability was that, under heavy load, many links would in fact spend a great deal of time being idle, which is the last thing you want under heavy load.

Another problem was that the range of link values was much too large. For example, a heavily loaded 9.6-kbps link could look 127 times more costly than a lightly loaded 56-kbps link. (Keep in mind, we're talking about the ARPANET circa 1975.) This means that the routing algorithm would choose a path with 126 hops of lightly loaded 56-kbps links in preference to a 1-hop 9.6-kbps path. While shedding some traffic from an overloaded line is a good idea, making it look so unattractive that it loses all its traffic is excessive. Using 126 hops when 1 hop will do is in general a bad use of network resources. Also, satellite links were unduly penalized, so that an idle 56-kbps satellite link looked considerably more costly than an idle 9.6-kbps terrestrial link, even though the former would give better performance for high-bandwidth applications.

A third approach addressed these problems. The major changes were to compress the dynamic range of the metric considerably, to account for the link type, and to smooth the variation of the metric with time.

The smoothing was achieved by several mechanisms. First, the delay measurement was transformed to a link utilization, and this number was averaged with the last reported utilization to suppress sudden changes. Second, there was a hard limit on how much the metric could change from one measurement cycle to the next. By smoothing the changes in the cost, the likelihood that all nodes would abandon a route at once is greatly reduced.

The compression of the dynamic range was achieved by feeding the measured utilization, the link type, and the link speed into a function that is shown graphically in Figure 3.37. below. Observe the following:

- A highly loaded link never shows a cost of more than three times its cost when idle.
- The most expensive link is only seven times the cost of the least expensive.
- A high-speed satellite link is more attractive than a low-speed terrestrial link.
- Cost is a function of link utilization only at moderate to high loads.

All of these factors mean that a link is much less likely to be universally abandoned, since a threefold increase in cost is likely to make the link unattractive for some paths while letting it remain the best choice for others. The slopes, offsets, and breakpoints for the curves in Figure 3.37 were arrived at by a great deal of trial and error, and they were carefully tuned to provide good performance.

Despite all these improvements, it turns out that in the majority of real-world network deployments, metrics change rarely if at all and only under the control of a network administrator, not automatically as described above. The reason for this is partly that conventional wisdom now holds that dynamically changing metrics are too unstable, even though this probably need not be true. Perhaps more significantly, many networks today lack the great disparity of link speeds and latencies that prevailed in the ARPANET. Thus, static metrics

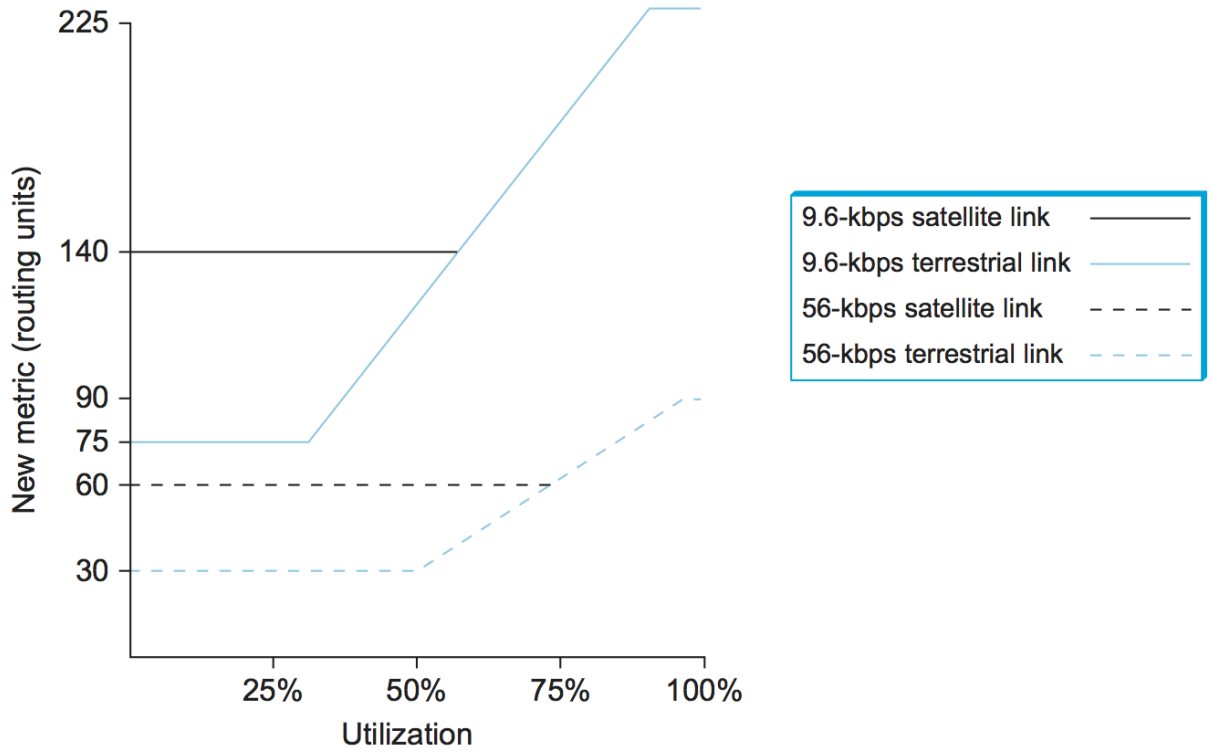


Figure 3.37.: Revised ARPANET routing metric versus link utilization.

are the norm. One common approach to setting metrics is to use a constant multiplied by (1/link_bandwidth).

Key Takeaway

Why do we still tell the story about a decades old algorithm that's no longer in use? Because it perfectly illustrates two valuable lessons. The first is that computer systems are often *designed iteratively based on experience.* We seldom get it right the first time, so it's important to deploy a simple solution sooner rather than later, and expect to improve it over time. Staying stuck in the design phase indefinitely is usually not a good plan. The second is the well-know KISS principle: *Keep it Simple, Stupid.* When building a complex system, less is often more. Opportunities to invent sophisticated optimizations are plentiful, and it's a tempting opportunity to pursue. While such optimizations sometimes have short-term value, it is shocking how often a simple approach proves best over time. This is because when a system has many moving parts, as the Internet most certainly does, keeping each part as simple as possible is usually the best approach. *[Next]*

3.5 Implementation

So far, we have talked about what switches and routers must do without describing how they do it. There is a straightforward way to build a switch or router: Buy a general-purpose processor and equip it with multiple network interfaces. Such a device, running suitable software, can receive packets on one of its interfaces, perform any of the switching or forwarding functions described in this chapter, and send packets out another

of its interfaces. This so called *software switch* is not too far removed from the architecture of many commercial mid- to low-end network devices.[1] Implementations that deliver high-end performance typically take advantage of additional hardware acceleration. We refer to these as *hardware switches*, although both approaches obviously include a combination of hardware and software.

This section gives an overview of both software-centric and hardware-centric designs, but it is worth noting that on the question of switches versus routers, the distinction isn't such a big deal. It turns out that the implementation of switches and routers have so much in common that a network administrator typically buys a single forwarding box and then configures it to be an L2 switch, an L3 router, or some combination of the two. Since their internal designs are so similar, we'll use the word *switch* to cover both variants throughout this section, avoiding the tedium of saying "switch or router" all the time. We'll call out the differences between the two when appropriate.

3.5.1 Software Switch

Figure 3.38 shows a software switch built using a general-purpose processor with four network interface cards (NICs). The path for a typical packet that arrives on, say, NIC 1 and is forwarded out on NIC 2 is straightforward: as NIC 1 receives the packet it copies its bytes directly into the main memory over the I/O bus (PCIe in this example) using a technique called *direct memory access* (DMA). Once the packet is in memory, the CPU examines its header to determine which interface the packet should be sent out on, and instructs NIC 2 to transmit the packet, again directly out of main memory using DMA. The important take-away is that the packet is buffered in main memory (this is the "store" half of store-and-forward), with the CPU reading only the necessary header fields into its internal registers for processing.

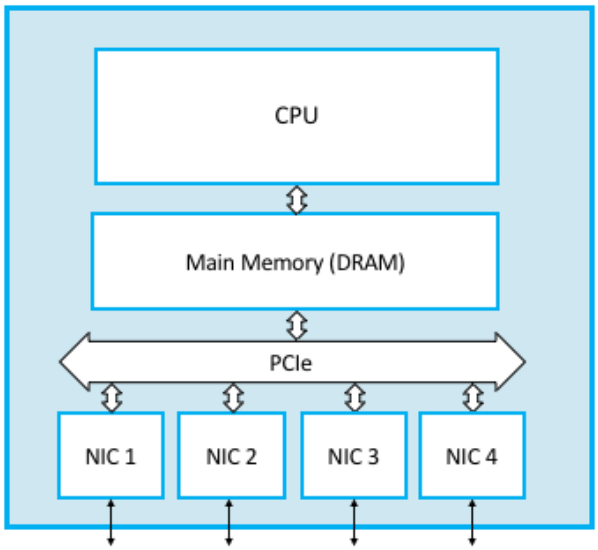


Figure 3.38.: A general-purpose processor used as a software switch.

There are two potential bottlenecks with this approach, one or both of which limits the aggregate packet forwarding capacity of the software switch.

The first problem is that performance is limited by the fact that all packets must pass into and out of main

[1] This is also how the very first Internet routers, often called *gateways* at the time, were implemented in the early days of the Internet.

memory. Your mileage will vary based on how much you are willing to pay for hardware, but as an example, a machine limited by a 1333-MHz, 64-bit-wide memory bus can transmit data at a peak rate of a little over 100 Gbps—enough to build a switch with a handful of 10-Gbps Ethernet ports, but hardly enough for a high-end router in the core of the Internet.

Moreover, this upper bound assumes that moving data is the only problem. This is a fair approximation for long packets but a bad one when packets are short, which is the worst-case situation switch designers have to plan for. With minimum-sized packets, the cost of processing each packet—parsing its header and deciding which output link to transmit it on—is likely to dominate, and potentially become a bottleneck. Suppose, for example, that a processor can perform all the necessary processing to switch 40 million packets each second. This is sometimes called the packet per second (pps) rate. If the average packet is 64 bytes, this would imply

$$\text{Throughput} = \text{pps x BitsPerPacket}$$

$$= 40 \times 10^6 \times 64 \times 8$$

$$= 2048 \times 10^7$$

that is, a throughput of about 20 Gbps—fast, but substantially below the range users are demanding from their switches today. Bear in mind that this 20 Gbps would be shared by all users connected to the switch, just as the bandwidth of a single (unswitched) Ethernet segment is shared among all users connected to the shared medium. Thus, for example, a 16-port switch with this aggregate throughput would only be able to cope with an average data rate of about 1 Gbps on each port.[2]

One final consideration is important to understand when evaluating switch implementations. The non-trivial algorithms discussed in this chapter—the spanning tree algorithm used by learning bridges, the distance-vector algorithm used by RIP, and the link-state algorithm used by OSPF—are *not* directly part of the per-packet forwarding decision. They run periodically in the background, but switches do not have to execute, say, OSPF code for every packet it forwards. The most costly routine the CPU is likely to execute on a per-packet basis is a table lookup, for example, looking up a VCI number in a VC table, an IP address in an L3 forwarding table, or an Ethernet address in an L2 forwarding table.

Key Takeaway

The distinction between these two kinds of processing is important enough to give it a name: the *control plane* corresponds to the background processing required to "control" the network (e.g., running OSPF, RIP, or the BGP protocol described in the next chapter) and the *data plane* corresponds to the per-packet processing required to move packets from input port to output port. For historical reasons, this distinction is called *control plane* and *user plane* in cellular access networks, but the idea is the same, and in fact, the 3GPP standard defines CUPS (Control/User Plane Separation) as an architectural principle.

These two kinds of processing are easy to conflate when both run on the same CPU, as is the case in software switch depicted in Figure 3.38, but performance can be dramatically improved by optimizing how the data plane is implemented, and correspondingly, specifying a well-defined interface between the control and data planes. *[Next]*

---

[2] These example performance numbers do not represent the absolute maximum throughput rate that highly tuned software running on a high-end server could achieve, but they are indicative of limits one ultimately faces in pursuing this approach.

3.5.2 Hardware Switch

Throughout much of the Internet's history, high-performance switches and routers have been specialized devices, built with Application-Specific Integrated Circuits (ASICs). While it was possible to build low-end routers and switches using commodity servers running C programs, ASICs were required to achieve the required throughput rates.

The problem with ASICs is that hardware takes a long time to design and fabricate, meaning the delay for adding new features to a switch is usually measured in years, not the days or weeks today's software industry is accustomed to. Ideally, we'd like to benefit from the performance of ASICs and the agility of software.

Fortunately, recent advances in domain specific processors (and other commodity components) have made this possible. Just as importantly, the full architectural specification for switches that take advantage of these new processors is now available on-line—the hardware equivalent of *open source software*. This means anyone can build a high-performance switch by pulling the blueprint off the web (see the Open Compute Project, OCP, for examples) in the same way it is possible to build your own PC. In both cases you still need software to run on the hardware, but just as Linux is available to run on your home-built PC, there are now open source L2 and L3 stacks available on GitHub to run on your home-built switch. Alternatively, you can simply buy a pre-built switch from a commodity switch manufacturer and then load your own software onto it. The following describes these open *white-box switches*, so called to contrast them with closed "black-box" devices that have historically dominated the industry.

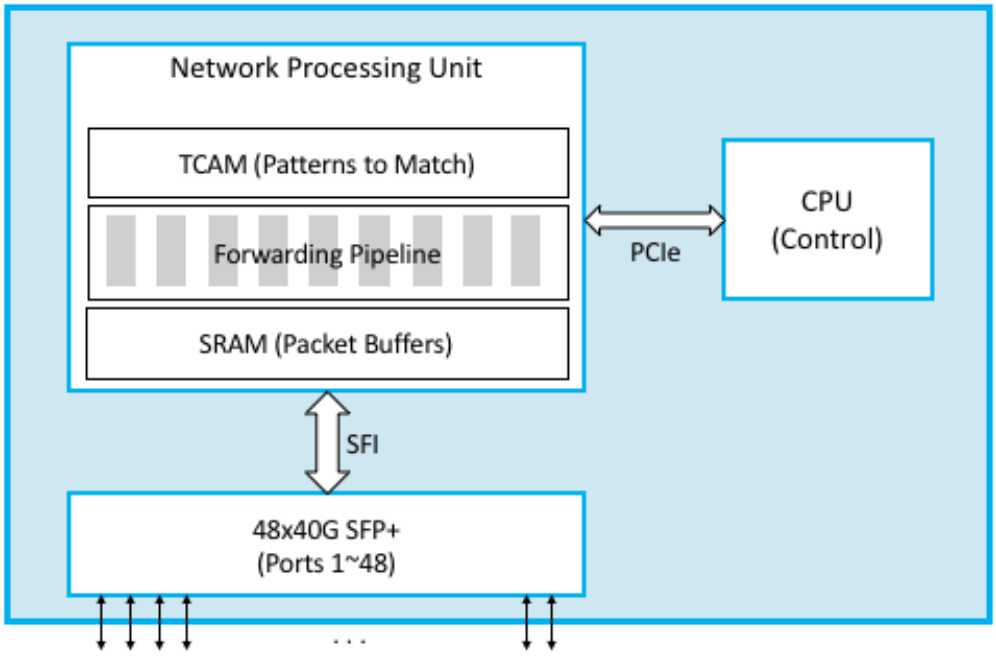


Figure 3.39.: White-box switch using a Network Processing Unit.

Figure 3.39 is a simplified depiction of a white-box switch. The key difference from the earlier implementation on a general-purpose processor is the addition of a Network Processor Unit (NPU), a domain-specific processor with an architecture and instruction set that has been optimized for processing packet headers (i.e., for implementing the data plane). NPUs are similar in spirit to GPUs that have an architecture optimized for rendering computer graphics, but in this case, the NPU is optimized for parsing packet headers and making a forwarding decision. NPUs are able to process packets (input, make a forwarding decision, and output) at

rates measured in Terabits-per-second (Tbps), easily fast enough to keep up with 32x100-Gbps ports, or the 48x40-Gbps ports shown in the diagram.

Network Processing Units

Our use of the term NPU is a bit non-standard. Historically, NPU was the name given more narrowly-defined network processing chips used, for example, to implement intelligent firewalls or deep packet inspection. They were not as general-purpose as the NPUs we're discussing here; nor were they as high-performance. It seems likely that the current approach will make purpose-built network processors obsolete, but in any case, we prefer the NPU nomenclator because it is consistent with the trend to build programmable domain-specific processors, including GPUs for graphics and TPUs (Tensor Processing Units) for AI.

The beauty of this new switch design is that a given white-box can now be programmed to be an L2 switch, and L3 router, or a combination of both, just by a matter of programming. The exact same control plane software stack used in a software switch still runs on the control CPU, but in addition, data plane "programs" are loaded onto the NPU to reflect the forwarding decisions made by the control plane software. Exactly how one "programs" the NPU depends on the chip vendor, of which there are currently several. In some cases, the forwarding pipeline is fixed and the control processor merely loads the forwarding table into the NPU (by fixed we mean the NPU only knows how to process certain headers, like Ethernet and IP), but in other cases, the forwarding pipeline is itself programmable. P4 is a new programming language that can be used to program such NPU-based forwarding pipelines. Among other things, P4 tries to hide many of the differences in the underlying NPU instruction sets.

Internally, an NPU takes advantage of three technologies. First, a fast SRAM-based memory buffers packets while they are being processed. SRAM (Static Random Access Memory), is roughly an order of magnitude faster than the DRAM (Dynamic Random Access Memory) that is used by main memory. Second, a TCAM-based memory stores bit patterns to be matched in the packets being processed. The "CAM" in TCAM stands for "Content Addressable Memory," which means that the key you want to look up in a table can effectively be used as the address into the memory that implements the table. The "T" stands for "Ternary" which is a fancy way to say the key you want to look up can have wildcards in it (e.g, key `10*1` matches both `1001` and `1011`). Finally, the processing involved to forward each packet is implemented by a forwarding pipeline. This pipeline is implemented by an ASIC, but when well-designed, the pipeline's forwarding behavior can be modified by changing the program it runs. At a high level, this program is expressed as a collection of *(Match, Action)* pairs: if you match such-and-such field in the header, then execute this-or-that action.

The relevance of packet processing being implemented by a multi-stage pipeline rather than a single-stage processor is that forwarding a single packet likely involves looking at multiple header fields. Each stage can be programmed to look at a different combination of fields. A multi-stage pipeline adds a little end-to-end latency to each packet (measured in nanoseconds), but also means that multiple packets can be processed at the same time. For example, Stage 2 can be making a second lookup on packet A while Stage 1 is doing an initial lookup on packet B, and so on. This means the NPU as a whole is able to keep up with line speeds. As of this writing, the state-of-the-art is 12.8 Tbps.

Finally, Figure 3.39 includes other commodity components that make this all practical. In particular, it is now possible to buy pluggable *transceiver* modules that take care of all the media access details—be it Gigabit Ethernet, 10-Gigabit Ethernet, or SONET—as well as the optics. These transceivers all conform to standardized form factors, such as SFP+, that can in turn be connected to other components over a

standardized bus (e.g., SFI). Again, the key takeaway is that the networking industry is just now entering into the same commoditized world that the computing industry has enjoyed for the last two decades.

3.5.3 Software Defined Networks

With switches becoming increasingly commoditized, attention is rightfully shifting to the software that controls them. This puts us squarely in the middle of a trend to build *Software Defined Networks* (SDN), an idea that started to germinate about ten years ago. In fact, it was the early stages of SDN that triggered the networking industry to move towards white-box switches.

The fundamental idea of SDN is one we've already discussed: to decouple the network control plane (i.e., where routing algorithms like RIP, OSPF, and BGP run) from the network data plane (i.e., where packet forwarding decisions get made), with the former moved into software running on commodity servers and the latter implemented by white-box switches. The key enabling idea behind SDN was to take this decoupling a step further, and to define a standard interface between the control plane and the data plane. Doing so allows any implementation of the control plane to talk to any implementation of the data plane; this breaks the dependency on any one vendor's bundled solution. The original interface is called *OpenFlow*, and this idea of decoupling the control and data planes came to be known as disaggregation. (The P4 language mentioned in the previous subsection is a second-generation attempt to define this interface by generalizing OpenFlow.)

Another important aspect of disaggregation is that a logically centralized control plane can be used to control a distributed network data plane. We say logically centralized because while the state collected by the control plane is maintained in a global data structure, such as a Network Map, the implementation of this data structure could still be distributed over multiple servers. For example, it could run in a cloud. This is important for both scalability and availability, where the key is that the two planes are configured and scaled independent of each other. This idea took off quickly in the cloud, where today's cloud providers run SDN-based solutions both within their datacenters and across the backbone networks that interconnect their datacenters.

One consequence of this design that isn't immediately obvious is that a logically centralized control plane doesn't just manage a network of physical (hardware) switches that interconnects physical servers, but it also manages a network of virtual (software) switches that interconnect virtual servers (e.g., Virtual Machines and containers). If you're counting "switch ports" (a good measure of all the devices connected to your network) then the number of virtual ports in the Internet rocketed past the number of physical ports in 2012.

One of other key enablers for SDN's success, as depicted in Figure 3.40, is the Network Operating System (NOS). Like a server operating system (e.g., Linux, iOS, Android, Windows) that provides a set of high-level abstractions that make it easier to implement applications (e.g., you can read and write files instead of directly accessing disk drives), a NOS makes it easier to implement network control functionality, otherwise known as *Control Apps*. A good NOS abstracts the details of the network switches and provides a *Network Map* abstraction to the application developer. The NOS detects changes in the underlying network (e.g., switches, ports, and links going up-and-down) and the control application simply implements the behavior it wants on this abstract graph. This means the NOS takes on the burden of collecting network state (the hard part of distributed algorithms like Link-State and Distance-Vector algorithms) and the app is free to simply implement the shortest path algorithm and load the forwarding rules into the underlying switches. By centralizing this logic, the goal is to come up with a globally optimized solution. The published evidence from cloud providers that have embraced this approach confirms this advantage.

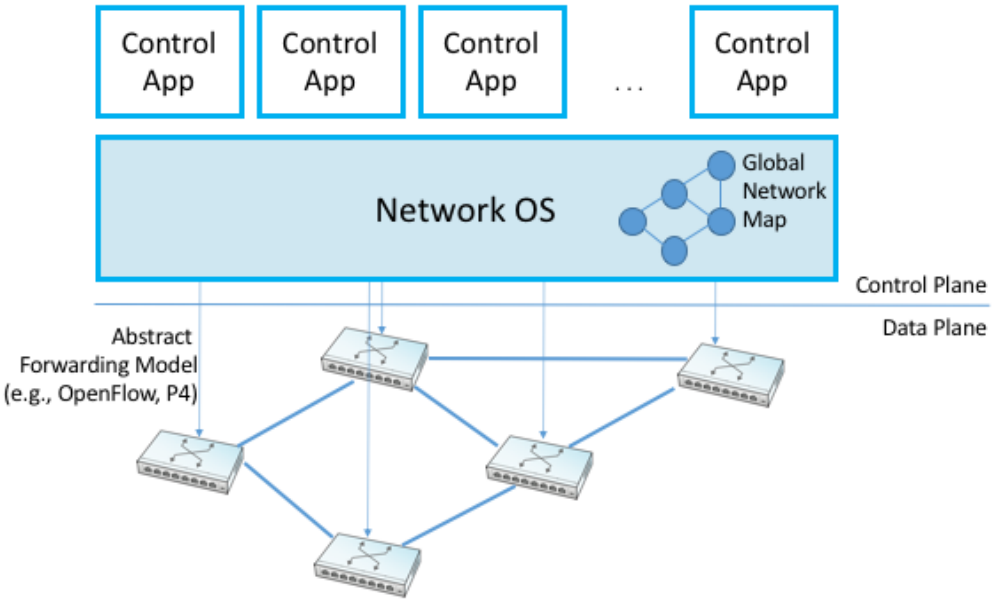


Figure 3.40.: Network Operating System (NOS) hosting a set of control applications and providing a logically centralized point of control for an underlying network data plane.

Key Takeaway

It is important to understand that SDN is an implementation strategy. It does not magically make fundamental problems like needing to compute a forwarding table go away. But instead of burdening the switches with having to exchange messages with each other as part of a distributed routing algorithm, the logically centralized SDN controller is charged with collecting link and port status information from the individual switches, constructing a global view of the network graph, and making that graph available to the control apps. From the control application's perspective, all the information it needs to compute the forwarding table is locally available. Keeping in mind that the SDN Controller is logically centralized but physically replicated on multple servers—for both scalable performance and high availability—it is still a hotly contested question whether the centralized or distributed approach is best. *[Next]*

As much of an advantage as the cloud providers have been able to get out of SDN, its adoption in enterprises and Telcos has been much slower. This is partly about the ability of different markets to manage their networks. The Googles, Microsofts, and Amazons of the world have the engineers and DevOps skills needed to take advantage of this technology, whereas others still prefer pre-packaged and integrated solutions that support the management and command line interfaces they are familiar with.

Perspective: Virtual Networks All the Way Down

For almost as long as there have been packet-switched networks, there have been ideas about how to virtualize them, starting with virtual circuits. But what exactly does it mean to virtualize a network?

Virtual memory is a helpful example. Virtual memory creates an abstraction of a large and private pool of memory, even though the underlying physical memory may be shared by many applications and consid-

erably smaller that the apparent pool of virtual memory. This abstraction enables programmers to operate under the illusion that there is plenty of memory and that no-one else is using it, while under the covers the memory management system takes care of things like mapping the virtual memory to physical resources and avoiding conflict between users.

Similarly, server virtualization presents the abstraction of a virtual machine (VM), which has all the features of a physical machine. Again, there may be many VMs supported on a single physical server, and the operating system and users on the virtual machine are happily unaware that the VM is being mapped onto physical resources.

A key point is the virtualization of computing resources preserves the abstractions and interfaces that existed before they were virtualized. This is important because it means that users of those abstractions don't need to change—they see a faithful reproduction of the resource being virtualized. Virtualization also means that the different users (sometimes called *tenants*) cannot interfere with each other. So what happens when we try to virtualize a network?

VPNs, as described in *Section 3.3*, were one early success for virtual networking. They allowed carriers to present corporate customers with the illusion that they had their own private network, even though in reality they were sharing underlying links and switches with many other users. VPNs, however, only virtualize a few resources, notably addressing and routing tables. Network virtualization as commonly understood today goes further, virtualizing every aspect of networking. That means that a virtual network should support all the basic abstractions of a physical network. In this sense, they are analogous to the virtual machine, with its support of all the resources of a server: CPU, storage, I/O, and so on.

To this end, VLANS, as described in *Section 3.2*, are how we typically virtualize an L2 network. VLANs proved to be quite useful to enterprises that wanted to isolate different internal groups (e.g., departments, labs), giving each of them the appearance of having their own private LAN. VLANs were also seen as a promising way to virtualize L2 networks in cloud datacenters, making it possible to give each tenant their own L2 network so as to isolate their traffic from the traffic of all other tenants. But there was a problem: the 4096 possible VLANs was not sufficient to account for all the tenants that a cloud might host, and to complicate matters, in a cloud the network needs to connect *virtual machines* rather than the physical machines that those VMs run on.

To address this problem, another standard called *Virtual Extensible LAN* (VXLAN) was introduced. Unlike the original approach, which effectively encapsulated a virtualized ethernet frame inside another ethernet frame, VXLAN encapsulates a virtual ethernet frame inside a UDP packet. This means a VXLAN-based virtual network (which is often referred to as an *overlay network*) runs on top of an IP-based network, which in turn runs on an underlying ethernet (or perhaps in just one VLAN of the underlying ethernet). VXLAN also makes it possible for one cloud tenant to have multiple VLANs of their own, which allows them to segregate their own internal traffic. This means it is ultimately possible to have a VLAN encapsulated in a VXLAN overlay encapsulated in a VLAN.

The powerful thing about virtualization is that when done right, it should be possible to nest one virtualized resource inside another virtualized resource, since after all, a virtual resource should behave just like a physical resources and we know how to virtualize physical resources! Said another way, being able to virtualize a virtual resource is the best proof that you have done a good job of virtualizing the original physical resource. To re-purpose the mythology of the World Turtle: It's virtual networks all the way down.

The actual VXLAN header is simple, as shown in Figure 3.41. It includes a 24-bit *Virtual Network Id* (VNI), plus some flag and reserved bits. It also implies a particular setting of the UDP source and destination port fields (see *Section 5.1*), with the destination port 4789 officially reserved for VXLANs. Figuring out how to

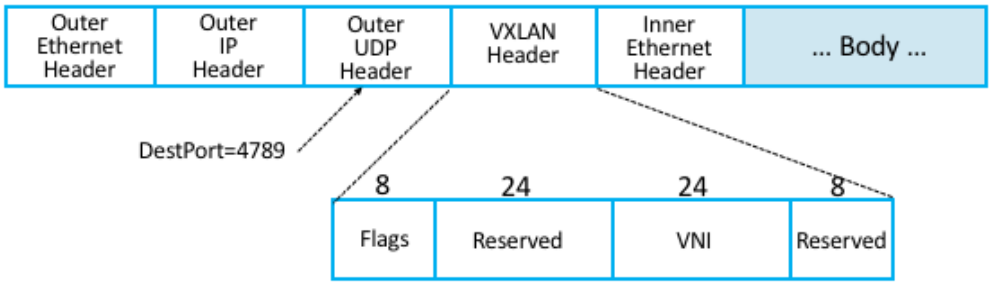


Figure 3.41.: VXLAN Header encapsulated in a UDP/IP packet. header.

uniquely identify virtual LANs (VLAN tags) and virtual networks (VXLAN VIDs) is the easy part. This is because encapsulation is the fundamental cornerstone of virtualization; all you need to add is an identifier that tells you which of many possible users this encapsulated packet belongs to.

The hard part is grappling with the idea of virtual networks being nested (encapsulated) inside virtual networks, which is networking's version of recursion. The other challenge is understanding how to automate the creation, management, migration, and deletion of virtual networks, and on this front there is still a lot of room for improvement. Mastering this challenge will be at the heart of networking in the next decade, and while some of this work will undoubtedly happen in proprietary settings, there are open source network virtualization platforms (e.g., the Linux Foundation's *Tungsten Fabric* project) leading the way.

Broader Perspective

To continue reading about the cloudification of the Internet, see *Perspective: The Cloud is Eating the Internet*.

To learn more about the maturation of virtual networks, we recommend:

- Network Heresy, 2012.
- Tungsten Fabric, 2018.

CHAPTER

FOUR

ADVANCED INTERNETWORKING

Every seeming equality conceals a hierarchy.

—*Mason Cooley*

Problem: Scaling to Billions

We have now seen how to build an internetwork that consists of a number of networks of different types. That is, we have dealt with the problem of *heterogeneity*. The second critical problem in internetworking—arguably the fundamental problem for all networking—is *scale*. To understand the problem of scaling a network, it is worth considering the growth of the Internet, which has roughly doubled in size each year for 30 years. This sort of growth forces us to face a number of challenges.

Chief among these is how do you build a routing system that can handle hundreds of thousands of networks and billions of end nodes? As we will see in this chapter, most approaches to tackling the scalability of routing depend on the introduction of hierarchy. We can introduce hierarchy in the form of areas within a domain; we also use hierarchy to scale the routing system among domains. The interdomain routing protocol that has enabled the Internet to scale to its current size is BGP. We will take a look at how BGP operates, and consider the challenges faced by BGP as the Internet continues to grow.

Closely related to the scalability of routing is the problem of addressing. Even two decades ago it had become apparent that the 32-bit addressing scheme of IP version 4 would not last forever. That led to the definition of a new version of IP—version 6, since version 5 had been used in an earlier experiment. IPv6 primarily expands the address space but also adds a number of new features, some of which have been retrofitted to IPv4.

While the Internet continues to grow in size, it also needs to evolve its functionality. The final sections of this chapter cover some significant enhancements to the Internet's capabilities. The first, multicast, is an enhancement of the basic service model. We show how multicast—the ability to deliver the same packets to a group of receivers efficiently—can be incorporated into an internet, and we describe several of the routing protocols that have been developed to support multicast. The second enhancement, Multiprotocol Label Switching (MPLS), modifies the forwarding mechanism of IP networks. This modification has enabled some changes in the way IP routing is performed and in the services offered by IP networks. Finally, we look at the effects of mobility on routing and describe some enhancements to IP to support mobile hosts and routers. For each of these enhancements, issues of scalability continue to be important.

4.1 Global Internet

At this point, we have seen how to connect a heterogeneous collection of networks to create an internetwork and how to use the simple hierarchy of the IP address to make routing in an internet somewhat scalable. We say "somewhat" scalable because, even though each router does not need to know about all the hosts connected to the internet, it does, in the model described so far, need to know about all the networks connected to the internet. Today's Internet has hundreds of thousands of networks connected to it (or more, depending on how you count). Routing protocols such as those we have just discussed do not scale to those kinds of numbers. This section looks at a variety of techniques that greatly improve scalability and that have enabled the Internet to grow as far as it has.

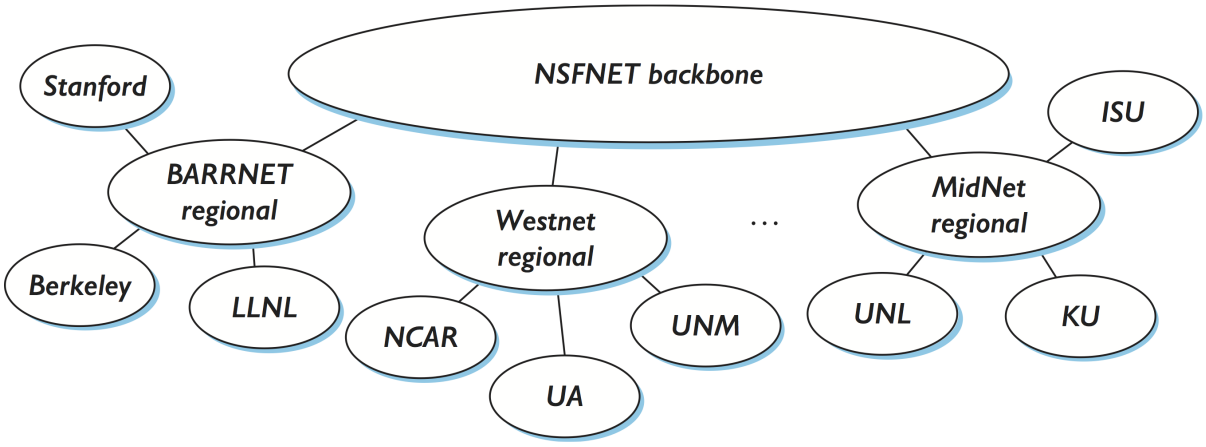


Figure 4.1.: The tree structure of the Internet in 1990.

Before getting to these techniques, we need to have a general picture in our heads of what the global Internet looks like. It is not just a random interconnection of Ethernets, but instead it takes on a shape that reflects the fact that it interconnects many different organizations. Figure 4.1 gives a simple depiction of the state of the Internet in 1990. Since that time, the Internet's topology has grown much more complex than this figure suggests—we present a slightly more accurate picture of the current Internet in a later section—but this picture will do for now.

One of the salient features of this topology is that it consists of end-user sites (e.g., Stanford University) that connect to service provider networks (e.g., BARRNET was a provider network that served sites in the San Francisco Bay Area). In 1990, many providers served a limited geographic region and were thus known as *regional networks*. The regional networks were, in turn, connected by a nationwide backbone. In 1990, this backbone was funded by the National Science Foundation (NSF) and was therefore called the *NSFNET backbone*.

NSFNET gave way to Internet2, which still runs a backbone on behalf of Research and Education institutions in the US (there are similar R&E networks in other countries), but of course most people get their Internet connectivity from commercial providers. Although the detail is not shown in the figure, today the largest provider networks (they are called tier-1) are typically built from dozens of high-end routers located in major meteropolitan areas (colloquially referred to as "NFL cities") connected by point-to-point links (often with 100 Gbps capacity). Similarly, each end-user site is typically not a single network but instead consists of multiple physical networks connected by switches and routers.

Notice in that each provider and end-user is likely to be an administratively independent entity. This has

some significant consequences on routing. For example, it is quite likely that different providers will have different ideas about the best routing protocol to use within their networks and on how metrics should be assigned to links in their network. Because of this independence, each provider's network is usually a single *autonomous system* (AS). We will define this term more precisely in a later section, but for now it is adequate to think of an AS as a network that is administered independently of other ASs.

The fact that the Internet has a discernible structure can be used to our advantage as we tackle the problem of scalability. In fact, we need to deal with two related scaling issues. The first is the scalability of routing. We need to find ways to minimize the number of network numbers that get carried around in routing protocols and stored in the routing tables of routers. The second is address utilization—that is, making sure that the IP address space does not get consumed too quickly.

Throughout this book, we see the principle of hierarchy used again and again to improve scalability. We saw in the previous chapter how the hierarchical structure of IP addresses, especially with the flexibility provided by Classless Interdomain Routing (CIDR) and subnetting, can improve the scalability of routing. In the next two sections, we'll see further uses of hierarchy (and its partner, aggregation) to provide greater scalability, first in a single domain and then between domains. Our final subsection looks at IP version 6, the invention of which was largely the result of scalability concerns.

4.1.1 Routing Areas

As a first example of using hierarchy to scale up the routing system, we'll examine how link-state routing protocols (such as OSPF and IS-IS) can be used to partition a routing domain into subdomains called *areas*. (The terminology varies somewhat among protocols—we use the OSPF terminology here.) By adding this extra level of hierarchy, we enable single domains to grow larger without overburdening the routing protocols or resorting to the more complex interdomain routing protocols described later.

An area is a set of routers that are administratively configured to exchange link-state information with each other. There is one special area—the backbone area, also known as area 0. An example of a routing domain divided into areas is shown in Figure 4.2 . Routers R1, R2, and R3 are members of the backbone area. They are also members of at least one nonbackbone area; R1 is actually a member of both area 1 and area 2. A router that is a member of both the backbone area and a nonbackbone area is an area border router (ABR). Note that these are distinct from the routers that are at the edge of an AS, which are referred to as AS border routers for clarity.

Routing within a single area is exactly as described in the previous chapter. All the routers in the area send link-state advertisements to each other and thus develop a complete, consistent map of the area. However, the link-state advertisements of routers that are not area border routers do not leave the area in which they originated. This has the effect of making the flooding and route calculation processes considerably more scalable. For example, router R4 in area 3 will never see a link-state advertisement from router R8 in area 1. As a consequence, it will know nothing about the detailed topology of areas other than its own.

How, then, does a router in one area determine the right next hop for a packet destined to a network in another area? The answer to this becomes clear if we imagine the path of a packet that has to travel from one nonbackbone area to another as being split into three parts. First, it travels from its source network to the backbone area, then it crosses the backbone, then it travels from the backbone to the destination network. To make this work, the area border routers summarize routing information that they have learned from one area and make it available in their advertisements to other areas. For example, R1 receives link-state advertisements from all the routers in area 1 and can thus determine the cost of reaching any network in area

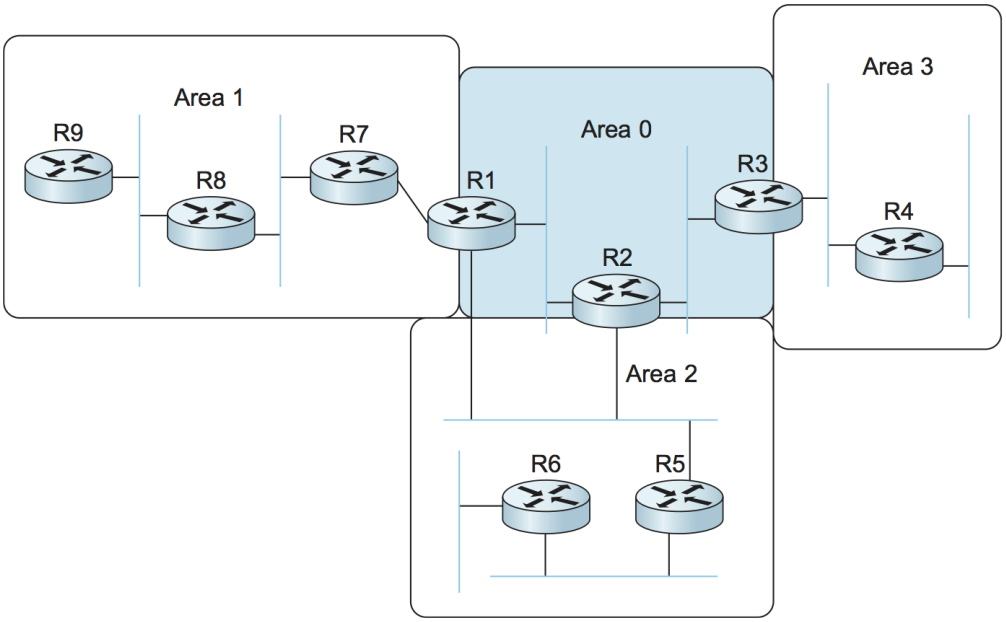


Figure 4.2.: A domain divided into areas.

1. When R1 sends link-state advertisements into area 0, it advertises the costs of reaching the networks in area 1 much as if all those networks were directly connected to R1. This enables all the area 0 routers to learn the cost to reach all networks in area 1. The area border routers then summarize this information and advertise it into the nonbackbone areas. Thus, all routers learn how to reach all networks in the domain.

Note that, in the case of area 2, there are two ABRs and that routers in area 2 will thus have to make a choice as to which one they use to reach the backbone. This is easy enough, since both R1 and R2 will be advertising costs to various networks, so it will become clear which is the better choice as the routers in area 2 run their shortest-path algorithm. For example, it is pretty clear that R1 is going to be a better choice than R2 for destinations in area 1.

When dividing a domain into areas, the network administrator makes a tradeoff between scalability and optimality of routing. The use of areas forces all packets traveling from one area to another to go via the backbone area, even if a shorter path might have been available. For example, even if R4 and R5 were directly connected, packets would not flow between them because they are in different nonbackbone areas. It turns out that the need for scalability is often more important than the need to use the absolute shortest path.

Key Takeaway

This illustrates an important principle in network design. There is frequently a trade-off between scalability and some sort of optimality. When hierarchy is introduced, information is hidden from some nodes in the network, hindering their ability to make perfect decisions. However, information hiding is essential to scaling a solution, since it saves all nodes from having global knowledge. It is invariably true in large networks that scalability is a more pressing design goal than selecting the optimal route. *[Next]*

Finally, we note that there is a trick by which network administrators can more flexibly decide which routers go in area 0. This trick uses the idea of a *virtual link* between routers. Such a virtual link is obtained by

configuring a router that is not directly connected to area 0 to exchange backbone routing information with a router that is. For example, a virtual link could be configured from R8 to R1, thus making R8 part of the backbone. R8 would now participate in link-state advertisement flooding with the other routers in area 0. The cost of the virtual link from R8 to R1 is determined by the exchange of routing information that takes place in area 1. This technique can help to improve the optimality of routing.

4.1.2 Interdomain Routing (BGP)

At the beginning of this chapter, we introduced the notion that the Internet is organized as autonomous systems, each of which is under the control of a single administrative entity. A corporation's complex internal network might be a single AS, as may the national network of any single Internet Service Provider (ISP). Figure 4.3 shows a simple network with two autonomous systems.

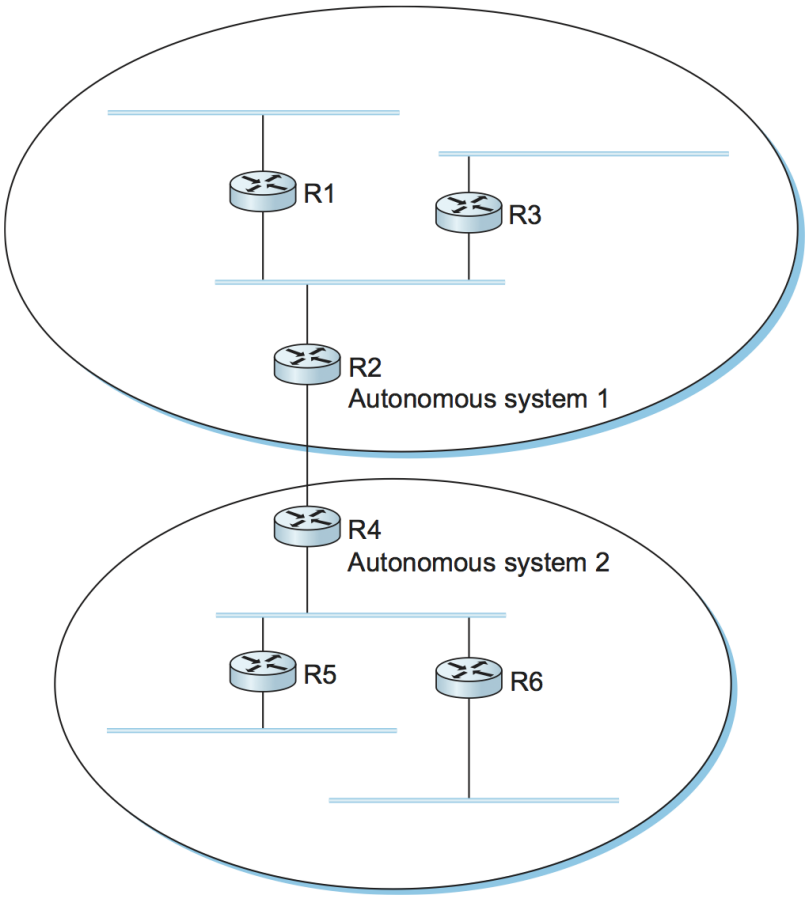


Figure 4.3.: A network with two autonomous systems.

The basic idea behind autonomous systems is to provide an additional way to hierarchically aggregate routing information in a large internet, thus improving scalability. We now divide the routing problem into two parts: routing within a single autonomous system and routing between autonomous systems. Since another name for autonomous systems in the Internet is routing *domains*, we refer to the two parts of the routing problem as interdomain routing and intradomain routing. In addition to improving scalability, the AS model decouples the intradomain routing that takes place in one AS from that taking place in another. Thus, each AS can run whatever intradomain routing protocols it chooses. It can even use static routes or multiple pro-

tocols, if desired. The interdomain routing problem is then one of having different ASs share reachability information—descriptions of the set of IP addresses that can be reached via a given AS—with each other.

Challenges in Interdomain Routing

Perhaps the most important challenge of interdomain routing today is the need for each AS to determine its own routing *policies*. A simple example routing policy implemented at a particular AS might look like this: "Whenever possible, I prefer to send traffic via AS X than via AS Y, but I'll use AS Y if it is the only path, and I never want to carry traffic from AS X to AS Y or *vice versa*." Such a policy would be typical when I have paid money to both AS X and AS Y to connect my AS to the rest of the Internet, and AS X is my preferred provider of connectivity, with AS Y being the fallback. Because I view both AS X and AS Y as providers (and presumably I paid them to play this role), I don't expect to help them out by carrying traffic between them across my network (this is called *transit* traffic). The more autonomous systems I connect to, the more complex policies I might have, especially when you consider backbone providers, who may interconnect with dozens of other providers and hundreds of customers and have different economic arrangements (which affect routing policies) with each one.

A key design goal of interdomain routing is that policies like the example above, and much more complex ones, should be supported by the interdomain routing system. To make the problem harder, I need to be able to implement such a policy without any help from other autonomous systems, and in the face of possible misconfiguration or malicious behavior by other autonomous systems. Furthermore, there is often a desire to keep the policies *private*, because the entities that run the autonomous systems—mostly ISPs—are often in competition with each other and don't want their economic arrangements made public.

There have been two major interdomain routing protocols in the history of the Internet. The first was the Exterior Gateway Protocol (EGP), which had a number of limitations, perhaps the most severe of which was that it constrained the topology of the Internet rather significantly. EGP was designed when the Internet had a treelike topology, such as that illustrated in Figure 4.1, and did not allow for the topology to become more general. Note that in this simple treelike structure there is a single backbone, and autonomous systems are connected only as parents and children and not as peers.

The replacement for EGP was the Border Gateway Protocol (BGP), which has iterated through four versions (BGP-4). BGP is often regarded as one of the more complex parts of the Internet. We'll cover some of its high points here.

Unlike its predecessor EGP, BGP makes virtually no assumptions about how autonomous systems are interconnected—they form an arbitrary graph. This model is clearly general enough to accommodate non-tree-structured internetworks, like the simplified picture of a multi-provider Internet shown in Figure 4.4. (It turns out there is still some sort of structure to the Internet, as we'll see below, but it's nothing like as simple as a tree, and BGP makes no assumptions about such structure.)

Unlike the simple tree-structured Internet shown in Figure 4.1, or even the fairly simple picture in Figure 4.4, today's Internet consists of a richly interconnected set of networks, mostly operated by private companies (ISPs) rather than governments. Many Internet Service Providers (ISPs) exist mainly to provide service to "consumers" (i.e., individuals with computers in their homes), while others offer something more like the old backbone service, interconnecting other providers and sometimes larger corporations. Often, many providers arrange to interconnect with each other at a single *peering point*.

To get a better sense of how we might manage routing among this complex interconnection of autonomous systems, we can start by defining a few terms. We define *local traffic* as traffic that originates at or terminates

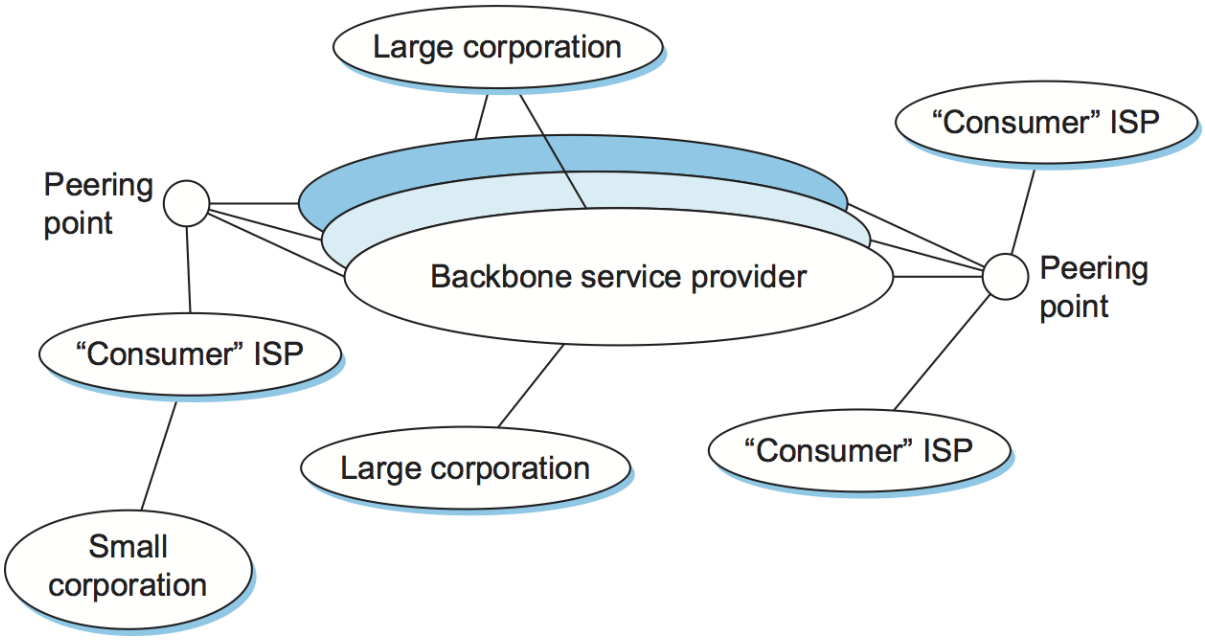


Figure 4.4.: A simple multi-provider Internet.

on nodes within an AS, and *transit traffic* as traffic that passes through an AS. We can classify autonomous systems into three broad types:

- Stub AS—an AS that has only a single connection to one other AS; such an AS will only carry local traffic. The small corporation in Figure 4.4 is an example of a stub AS.
- Multihomed AS—an AS that has connections to more than one other AS but that refuses to carry transit traffic, such as the large corporation at the top of Figure 4.4.
- Transit AS—an AS that has connections to more than one other AS and that is designed to carry both transit and local traffic, such as the backbone providers in Figure 4.4.

Whereas the discussion of routing in the previous chapter focused on finding optimal paths based on minimizing some sort of link metric, the goals of interdomain routing are rather more complex. First, it is necessary to find *some* path to the intended destination that is loop free. Second, paths must be compliant with the policies of the various autonomous systems along the path—and, as we have already seen, those policies might be almost arbitrarily complex. Thus, while intradomain focuses on a well-defined problem of optimizing the scalar cost of the path, interdomain focuses on finding a non-looping, *policy-compliant* path—a much more complex optimization problem.

There are additional factors that make interdomain routing hard. The first is simply a matter of scale. An Internet backbone router must be able to forward any packet destined anywhere in the Internet. That means having a routing table that will provide a match for any valid IP address. While CIDR has helped to control the number of distinct prefixes that are carried in the Internet's backbone routing, there is inevitably a lot of routing information to pass around—roughly 700,000 prefixes in mid-2018.

A further challenge in interdomain routing arises from the autonomous nature of the domains. Note that each domain may run its own interior routing protocols and use any scheme it chooses to assign metrics to paths. This means that it is impossible to calculate meaningful path costs for a path that crosses multiple autonomous systems. A cost of 1000 across one provider might imply a great path, but it might mean an

unacceptably bad one from another provider. As a result, interdomain routing advertises only *reachability*. The concept of reachability is basically a statement that "you can reach this network through this AS." This means that for interdomain routing to pick an optimal path is essentially impossible.

The autonomous nature of interdomain raises issue of trust. Provider A might be unwilling to believe certain advertisements from provider B for fear that provider B will advertise erroneous routing information. For example, trusting provider B when he advertises a great route to anywhere in the Internet can be a disastrous choice if provider B turns out to have made a mistake configuring his routers or to have insufficient capacity to carry the traffic.

The issue of trust is also related to the need to support complex policies as noted above. For example, I might be willing to trust a particular provider only when he advertises reachability to certain prefixes, and thus I would have a policy that says, "Use AS X to reach only prefixes p and q, if and only if AS X advertises reachability to those prefixes."

Basics of BGP

Each AS has one or more *border routers* through which packets enter and leave the AS. In our simple example in Figure 4.3, routers R2 and R4 would be border routers. (Over the years, routers have sometimes also been known as *gateways*, hence the names of the protocols BGP and EGP). A border router is simply an IP router that is charged with the task of forwarding packets between autonomous systems.

Each AS that participates in BGP must also have at least one *BGP* speaker, a router that "speaks" BGP to other BGP speakers in other autonomous systems. It is common to find that border routers are also BGP speakers, but that does not have to be the case.

BGP does not belong to either of the two main classes of routing protocols, distance-vector or link-state. Unlike these protocols, BGP advertises *complete paths* as an enumerated list of autonomous systems to reach a particular network. It is sometimes called a *path-vector* protocol for this reason. The advertisement of complete paths is necessary to enable the sorts of policy decisions described above to be made in accordance with the wishes of a particular AS. It also enables routing loops to be readily detected.

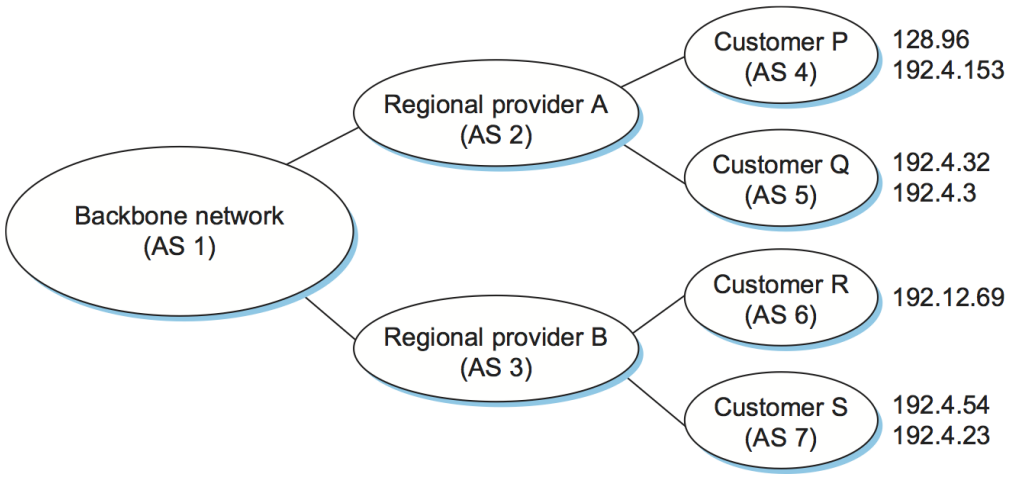


Figure 4.5.: Example of a network running BGP.

To see how this works, consider the very simple example network in Figure 4.5. Assume that the providers are transit networks, while the customer networks are stubs. A BGP speaker for the AS of provider A (AS 2)

would be able to advertise reachability information for each of the network numbers assigned to customers P and Q. Thus, it would say, in effect, "The networks 128.96, 192.4.153, 192.4.32, and 192.4.3 can be reached directly from AS 2." The backbone network, on receiving this advertisement, can advertise, "The networks 128.96, 192.4.153, 192.4.32, and 192.4.3 can be reached along the path (AS 1, AS 2)." Similarly, it could advertise, "The networks 192.12.69, 192.4.54, and 192.4.23 can be reached along the path (AS 1, AS 3)."

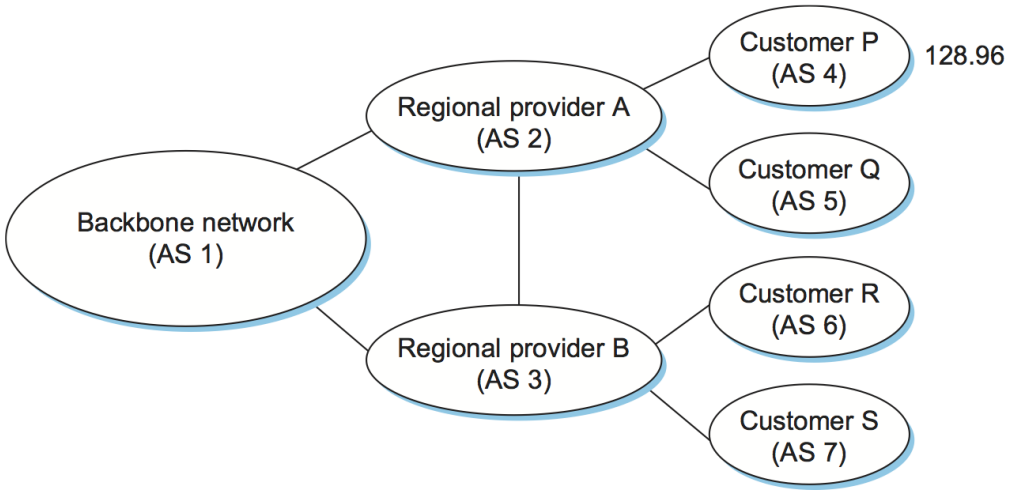


Figure 4.6.: Example of loop among autonomous systems.

An important job of BGP is to prevent the establishment of looping paths. For example, consider the network illustrated in Figure 4.6. It differs from Figure 4.5 only in the addition of an extra link between AS 2 and AS 3, but the effect now is that the graph of autonomous systems has a loop in it. Suppose AS 1 learns that it can reach network 128.96 through AS 2, so it advertises this fact to AS 3, who in turn advertises it back to AS 2. In the absence of any loop prevention mechanism, AS 2 could now decide that AS 3 was the preferred route for packets destined for 128.96. If AS 2 starts sending packets addressed to 128.96 to AS 3, AS 3 would send them to AS 1; AS 1 would send them back to AS 2; and they would loop forever. This is prevented by carrying the complete AS path in the routing messages. In this case, the advertisement for a path to 128.96 received by AS 2 from AS 3 would contain an AS path of (AS 3, AS 1, AS 2, AS 4). AS 2 sees itself in this path, and thus concludes that this is not a useful path for it to use.

In order for this loop prevention technique to work, the AS numbers carried in BGP clearly need to be unique. For example, AS 2 can only recognize itself in the AS path in the above example if no other AS identifies itself in the same way. AS numbers are now 32-bits long, and they are assigned by a central authority to assure uniqueness.

A given AS will only advertise routes that it considers good enough for itself. That is, if a BGP speaker has a choice of several different routes to a destination, it will choose the best one according to its own local policies, and then that will be the route it advertises. Furthermore, a BGP speaker is under no obligation to advertise any route to a destination, even if it has one. This is how an AS can implement a policy of not providing transit—by refusing to advertise routes to prefixes that are not contained within that AS, even if it knows how to reach them.

Given that links fail and policies change, BGP speakers need to be able to cancel previously advertised paths. This is done with a form of negative advertisement known as a *withdrawn route*. Both positive and negative reachability information are carried in a BGP update message, the format of which is shown in Figure 4.7. (Note that the fields in this figure are multiples of 16 bits, unlike other packet formats in this chapter.)

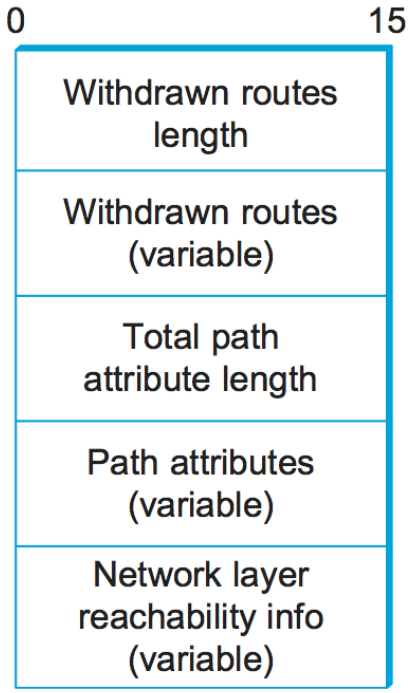


Figure 4.7.: BGP-4 update packet format.

Unlike the routing protocols described in the previous chapter, BGP is defined to run on top of TCP, the reliable transport protocol. Because BGP speakers can count on TCP to be reliable, this means that any information that has been sent from one speaker to another does not need to be sent again. Thus, as long as nothing has changed, a BGP speaker can simply send an occasional *keepalive* message that says, in effect, "I'm still here and nothing has changed." If that router were to crash or become disconnected from its peer, it would stop sending the keepalives, and the other routers that had learned routes from it would assume that those routes were no longer valid.

Common AS Relationships and Policies

Having said that policies may be arbitrarily complex, there turn out to be a few common ones, reflecting common relationships between autonomous systems. The most common relationships are illustrated in Figure 4.8. The three common relationships and the policies that go with them are as follows:

- *Provider-Customer*—Providers are in the business of connecting their customers to the rest of the Internet. A customer might be a corporation, or it might be a smaller ISP (which may have customers of its own). So the common policy is to advertise all the routes I know about to my customer, and advertise routes I learn from my customer to everyone.
- *Customer-Provider*—In the other direction, the customer wants to get traffic directed to him (and his customers, if he has them) by his provider, and he wants to be able to send traffic to the rest of the Internet through his provider. So the common policy in this case is to advertise my own prefixes and routes learned from my customers to my provider, advertise routes learned from my provider to my customers, but don't advertise routes learned from one provider to another provider. That last part is to make sure the customer doesn't find himself in the business of carrying traffic from one provider to

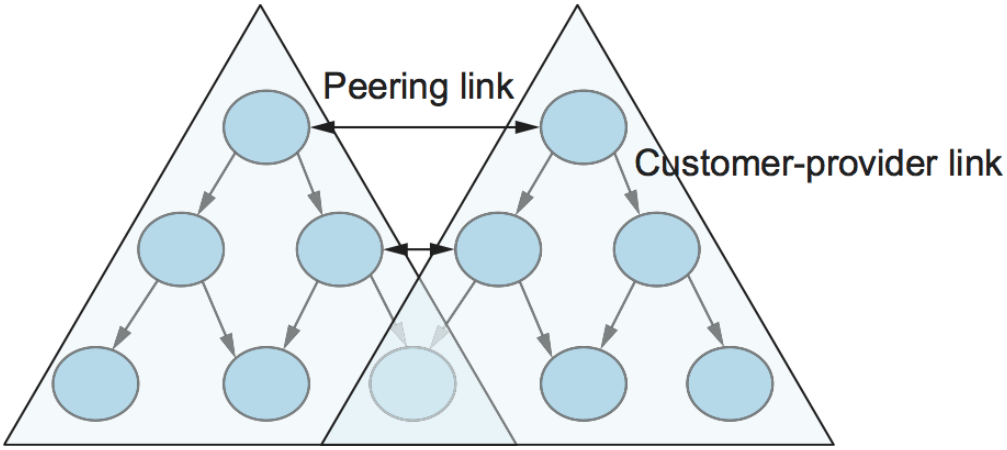


Figure 4.8.: Common AS relationships.

another, which isn't in his interests if he is paying the providers to carry traffic for him.

- *Peer*—The third option is a symmetrical peering between autonomous systems. Two providers who view themselves as equals usually peer so that they can get access to each other's customers without having to pay another provider. The typical policy here is to advertise routes learned from my customers to my peer, advertise routes learned from my peer to my customers, but don't advertise routes from my peer to any provider or *vice versa*.

One thing to note about this figure is the way it has brought back some structure to the apparently unstructured Internet. At the bottom of the hierarchy we have the stub networks that are customers of one or more providers, and as we move up the hierarchy we see providers who have other providers as their customers. At the top, we have providers who have customers and peers but are not customers of anyone. These providers are known as the *Tier-1* providers.

Key Takeaway

Let's return to the real question: How does all this help us to build scalable networks? First, the number of nodes participating in BGP is on the order of the number of autonomous systems, which is much smaller than the number of networks. Second, finding a good interdomain route is only a matter of finding a path to the right border router, of which there are only a few per AS. Thus, we have neatly subdivided the routing problem into manageable parts, once again using a new level of hierarchy to increase scalability. The complexity of interdomain routing is now on the order of the number of autonomous systems, and the complexity of intradomain routing is on the order of the number of networks in a single AS. *[Next]*

Integrating Interdomain and Intradomain Routing

While the preceding discussion illustrates how a BGP speaker learns interdomain routing information, the question still remains as to how all the other routers in a domain get this information. There are several ways this problem can be addressed.

Let's start with a very simple situation, which is also very common. In the case of a stub AS that only connects to other autonomous systems at a single point, the border router is clearly the only choice for all

routes that are outside the AS. Such a router can inject a *default route* into the intradomain routing protocol. In effect, this is a statement that any network that has not been explicitly advertised in the intradomain protocol is reachable through the border router. Recall from the discussion of IP forwarding in the previous chapter that the default entry in the forwarding table comes after all the more specific entries, and it matches anything that failed to match a specific entry.

The next step up in complexity is to have the border routers inject specific routes they have learned from outside the AS. Consider, for example, the border router of a provider AS that connects to a customer AS. That router could learn that the network prefix 192.4.54/24 is located inside the customer AS, either through BGP or because the information is configured into the border router. It could inject a route to that prefix into the routing protocol running inside the provider AS. This would be an advertisement of the sort, "I have a link to 192.4.54/24 of cost X." This would cause other routers in the provider AS to learn that this border router is the place to send packets destined for that prefix.

The final level of complexity comes in backbone networks, which learn so much routing information from BGP that it becomes too costly to inject it into the intradomain protocol. For example, if a border router wants to inject 10,000 prefixes that it learned about from another AS, it will have to send very big link-state packets to the other routers in that AS, and their shortest-path calculations are going to become very complex. For this reason, the routers in a backbone network use a variant of BGP called *interior BGP* (iBGP) to effectively redistribute the information that is learned by the BGP speakers at the edges of the AS to all the other routers in the AS. (The other variant of BGP, discussed above, runs between autonomous systems and is called *exterior BGP*, or eBGP). iBGP enables any router in the AS to learn the best border router to use when sending a packet to any address. At the same time, each router in the AS keeps track of how to get to each border router using a conventional intradomain protocol with no injected information. By combining these two sets of information, each router in the AS is able to determine the appropriate next hop for all prefixes.

To see how this all works, consider the simple example network, representing a single AS, in Figure 4.9. The three border routers, A, D, and E, speak eBGP to other autonomous systems and learn how to reach various prefixes. These three border routers communicate with other and with the interior routers B and C by building a mesh of iBGP sessions among all the routers in the AS. Let's now focus in on how router B builds up its complete view of how to forward packets to any prefix. Look at the top left of Figure 4.10, which shows the information that router B learns from its iBGP sessions. It learns that some prefixes are best reached via router A, some via D, and some via E. At the same time, all the routers in the AS are also running some intradomain routing protocol such as Routing Information Protocol (RIP) or Open Shortest Path First (OSPF). (A generic term for intradomain protocols is an interior gateway protocol, or IGP.) From this completely separate protocol, B learns how to reach other nodes *inside* the domain, as shown in the top right table. For example, to reach router E, B needs to send packets toward router C. Finally, in the bottom table, B puts the whole picture together, combining the information about external prefixes learned from iBGP with the information about interior routes to the border routers learned from the IGP. Thus, if a prefix like 18.0/16 is reachable via border router E, and the best interior path to E is via C, then it follows that any packet destined for 18.0/16 should be forwarded toward C. In this way, any router in the AS can build up a complete routing table for any prefix that is reachable via some border router of the AS.

4.2 IP Version 6

The motivation for defining a new version of IP is simple: to deal with exhaustion of the IP address space. CIDR helped considerably to contain the rate at which the Internet address space was being consumed and

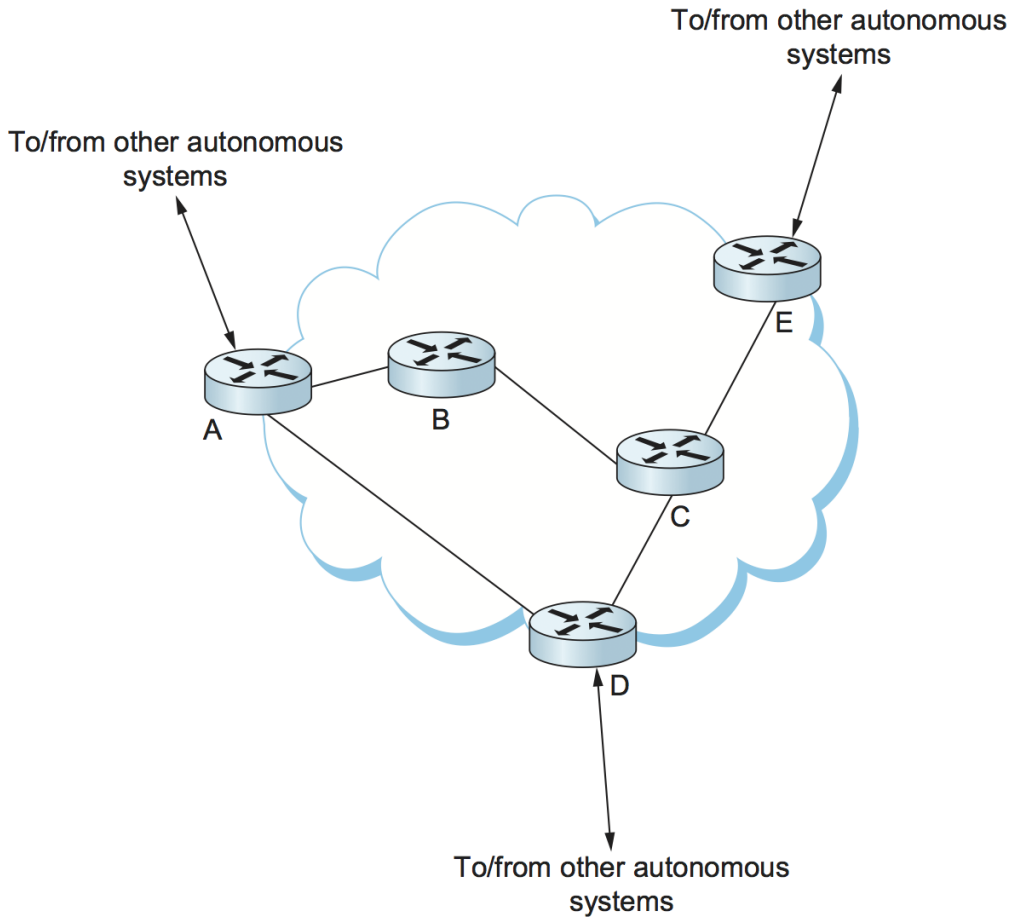


Figure 4.9.: Example of interdomain and intradomain routing. All routers run iBGP and an intradomain routing protocol. Border routers A, D, and E also run eBGP to other autonomous systems.

Prefix	BGP Next Hop
18.0/16	E
12.5.5/24	A
128.34/16	D
128.69./16	A

BGP table for the AS

Router	IGP Path
A	A
C	C
D	C
E	C

IGP table for router B

Prefix	IGP Path
18.0/16	C
12.5.5/24	A
128.34/16	C
128.69./16	A

Combined table for router B

Figure 4.10.: BGP routing table, IGP routing table, and combined table at router B.

also helped to control the growth of routing table information needed in the Internet's routers. However, these techniques are no longer adequate. In particular, it is virtually impossible to achieve 100% address utilization efficiency, so the address space was consumed well before the 4 billionth host was connected to the Internet. Even if we were able to use all 4 billion addresses, it is now clear that IP addresses need to be assigned to more than traditional computers, including smart phones, televisions, household appliances, and drones. With the clarity of 20/20 hindsight, a 32-bit address space is quite small.

4.2.1 Historical Perspective

The IETF began looking at the problem of expanding the IP address space in 1991, and several alternatives were proposed. Since the IP address is carried in the header of every IP packet, increasing the size of the address dictates a change in the packet header. This means a new version of the Internet Protocol and, as a consequence, a need for new software for every host and router in the Internet. This is clearly not a trivial matter—it is a major change that needs to be thought about very carefully.

The effort to define a new version of IP was originally known as IP Next Generation, or IPng. As the work progressed, an official IP version number was assigned, so IPng became IPv6. Note that the version of IP discussed so far in this chapter is version 4 (IPv4). The apparent discontinuity in numbering is the result of version number 5 being used for an experimental protocol many years ago.

The significance of changing to a new version of IP caused a snowball effect. The general feeling among network designers was that if you are going to make a change of this magnitude you might as well fix as many other things in IP as possible at the same time. Consequently, the IETF solicited white papers from anyone who cared to write one, asking for input on the features that might be desired in a new version of IP. In addition to the need to accommodate scalable routing and addressing, some of the other wish list items for IPng included:

- Support for real-time services
- Security support
- Autoconfiguration (i.e., the ability of hosts to automatically configure themselves with such information as their own IP address and domain name)
- Enhanced routing functionality, including support for mobile hosts

It is interesting to note that, while many of these features were absent from IPv4 at the time IPv6 was being designed, support for all of them has made its way into IPv4 in recent years, often using similar techniques in both protocols. It can be argued that the freedom to think of IPv6 as a clean slate facilitated the design of new capabilities for IP that were then retrofitted into IPv4.

In addition to the wish list, one absolutely non-negotiable feature for IPv6 was that there must be a transition plan to move from the current version of IP (version 4) to the new version. With the Internet being so large and having no centralized control, it would be completely impossible to have a "flag day" on which everyone shut down their hosts and routers and installed a new version of IP. The architects expected a long transition period in which some hosts and routers would run IPv4 only, some will run IPv4 and IPv6, and some will run IPv6 only. It is doubtful they anticipated that transition period would be approaching its 30th anniversary.

4.2.2 Addresses and Routing

First and foremost, IPv6 provides a 128-bit address space, as opposed to the 32 bits of version 4. Thus, while version 4 can potentially address 4 billion nodes if address assignment efficiency reaches 100%, IPv6 can address 3.4×10^{38} nodes, again assuming 100% efficiency. As we have seen, though, 100% efficiency in address assignment is not likely. Some analysis of other addressing schemes, such as those of the French and U.S. telephone networks, as well as that of IPv4, have turned up some empirical numbers for address assignment efficiency. Based on the most pessimistic estimates of efficiency drawn from this study, the IPv6 address space is predicted to provide over 1500 addresses per square foot of the Earth's surface, which certainly seems like it should serve us well even when toasters on Venus have IP addresses.

Address Space Allocation

Drawing on the effectiveness of CIDR in IPv4, IPv6 addresses are also classless, but the address space is still subdivided in various ways based on the leading bits. Rather than specifying different address classes, the leading bits specify different uses of the IPv6 address. The current assignment of prefixes is listed in Table 4.1.

Table 4.1.: Address Prefix Assignments for IPv6.

Prefix	Use
00...0 (128 bits)	Unspecified
00...1 (128 bits)	Loopback
1111 1111	Multicast addresses
1111 1110 10	Link-local unicast
Everything else	Global Unicast

This allocation of the address space warrants a little discussion. First, the entire functionality of IPv4's three main address classes (A, B, and C) is contained inside the "everything else" range. Global Unicast Addresses, as we will see shortly, are a lot like classless IPv4 addresses, only much longer. These are the main ones of interest at this point, with over 99% of the total IPv6 address space available to this important form of address. (At the time of writing, IPv6 unicast addresses are being allocated from the block that begins `001`, with the remaining address space—about 87%—being reserved for future use.)

The multicast address space is (obviously) for multicast, thereby serving the same role as class D addresses in IPv4. Note that multicast addresses are easy to distinguish—they start with a byte of all 1s. We will see how these addresses are used in a later section.

The idea behind link-local use addresses is to enable a host to construct an address that will work on the network to which it is connected without being concerned about the global uniqueness of the address. This may be useful for autoconfiguration, as we will see below. Similarly, the site-local use addresses are intended to allow valid addresses to be constructed on a site (e.g., a private corporate network) that is not connected to the larger Internet; again, global uniqueness need not be an issue.

Within the global unicast address space are some important special types of addresses. A node may be assigned an IPv4-compatible IPv6 address by zero-extending a 32-bit IPv4 address to 128 bits. A node that is only capable of understanding IPv4 can be assigned an IPv4-mapped IPv6 address by prefixing the 32-bit IPv4 address with 2 bytes of all 1s and then zero-extending the result to 128 bits. These two special address types have uses in the IPv4-to-IPv6 transition (see the sidebar on this topic).

Address Notation

Just as with IPv4, there is some special notation for writing down IPv6 addresses. The standard representation is `x:x:x:x:x:x:x:x`, where each `x` is a hexadecimal representation of a 16-bit piece of the address. An example would be

```
47CD:1234:4422:ACO2:0022:1234:A456:0124
```

Any IPv6 address can be written using this notation. Since there are a few special types of IPv6 addresses, there are some special notations that may be helpful in certain circumstances. For example, an address with a large number of contiguous 0s can be written more compactly by omitting all the 0 fields. Thus,

```
47CD:0000:0000:0000:0000:0000:A456:0124
```

could be written

```
47CD::A456:0124
```

Clearly, this form of shorthand can only be used for one set of contiguous 0s in an address to avoid ambiguity.

The two types of IPv6 addresses that contain an embedded IPv4 address have their own special notation that makes extraction of the IPv4 address easier. For example, the IPv4-mapped IPv6 address of a host whose IPv4 address was 128.96.33.81 could be written as

```
::FFFF:128.96.33.81
```

That is, the last 32 bits are written in IPv4 notation, rather than as a pair of hexadecimal numbers separated by a colon. Note that the double colon at the front indicates the leading 0s.

Global Unicast Addresses

By far the most important sort of addressing that IPv6 must provide is plain old unicast addressing. It must do this in a way that supports the rapid rate of addition of new hosts to the Internet and that allows routing to be done in a scalable way as the number of physical networks in the Internet grows. Thus, at the heart of IPv6 is the unicast address allocation plan that determines how unicast addresses will be assigned to service providers, autonomous systems, networks, hosts, and routers.

In fact, the address allocation plan that is proposed for IPv6 unicast addresses is extremely similar to that being deployed with CIDR in IPv4. To understand how it works and how it provides scalability, it is helpful to define some new terms. We may think of a nontransit AS (i.e., a stub or multihomed AS) as a *subscriber*, and we may think of a transit AS as a *provider*. Furthermore, we may subdivide providers into *direct* and *indirect*. The former are directly connected to subscribers. The latter primarily connect other providers, are not connected directly to subscribers, and are often known as *backbone networks*.

With this set of definitions, we can see that the Internet is not just an arbitrarily interconnected set of autonomous systems; it has some intrinsic hierarchy. The difficulty lies in making use of this hierarchy without inventing mechanisms that fail when the hierarchy is not strictly observed, as happened with EGP. For example, the distinction between direct and indirect providers becomes blurred when a subscriber connects to a backbone or when a direct provider starts connecting to many other providers.

As with CIDR, the goal of the IPv6 address allocation plan is to provide aggregation of routing information to reduce the burden on intradomain routers. Again, the key idea is to use an address prefix—a set of contiguous bits at the most significant end of the address—to aggregate reachability information to a large number of networks and even to a large number of autonomous systems. The main way to achieve this is to assign an address prefix to a direct provider and then for that direct provider to assign longer prefixes that begin with that prefix to its subscribers. Thus, a provider can advertise a single prefix for all of its subscribers.

Of course, the drawback is that if a site decides to change providers, it will need to obtain a new address prefix and renumber all the nodes in the site. This could be a colossal undertaking, enough to dissuade most people from ever changing providers. For this reason, there is ongoing research on other addressing schemes, such as geographic addressing, in which a site's address is a function of its location rather than the provider to which it attaches. At present, however, provider-based addressing is necessary to make routing work efficiently.

Note that while IPv6 address assignment is essentially equivalent to the way address assignment has happened in IPv4 since the introduction of CIDR, IPv6 has the significant advantage of not having a large installed base of assigned addresses to fit into its plans.

One question is whether it makes sense for hierarchical aggregation to take place at other levels in the hierarchy. For example, should all providers obtain their address prefixes from within a prefix allocated to the backbone to which they connect? Given that most providers connect to multiple backbones, this probably doesn't make sense. Also, since the number of providers is much smaller than the number of sites, the benefits of aggregating at this level are much fewer.

One place where aggregation may make sense is at the national or continental level. Continental boundaries form natural divisions in the Internet topology. If all addresses in Europe, for example, had a common prefix, then a great deal of aggregation could be done, and most routers in other continents would only need one routing table entry for all networks with the Europe prefix. Providers in Europe would all select their prefixes such that they began with the European prefix. Using this scheme, an IPv6 address might look like Figure 4.11. The `RegistryID` might be an identifier assigned to a European address registry, with different IDs assigned to other continents or countries. Note that prefixes would be of different lengths under this scenario. For example, a provider with few customers could have a longer prefix (and thus less total address space available) than one with many customers.

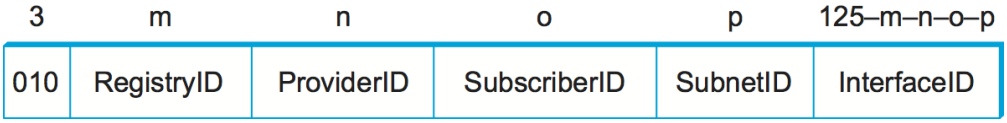


Figure 4.11.: An IPv6 provider-based unicast address.

One tricky situation could occur when a subscriber is connected to more than one provider. Which prefix should the subscriber use for his or her site? There is no perfect solution to the problem. For example, suppose a subscriber is connected to two providers, X and Y. If the subscriber takes his prefix from X, then Y has to advertise a prefix that has no relationship to its other subscribers and that as a consequence cannot be aggregated. If the subscriber numbers part of his AS with the prefix of X and part with the prefix of Y, he runs the risk of having half his site become unreachable if the connection to one provider goes down. One solution that works fairly well if X and Y have a lot of subscribers in common is for them to have three prefixes between them: one for subscribers of X only, one for subscribers of Y only, and one for the sites that are subscribers of both X and Y.

4.2.3 Packet Format

Despite the fact that IPv6 extends IPv4 in several ways, its header format is actually simpler. This simplicity is due to a concerted effort to remove unnecessary functionality from the protocol. Figure 4.12 shows the result.

As with many headers, this one starts with a `Version` field, which is set to 6 for IPv6. The `Version` field is in the same place relative to the start of the header as IPv4's `Version` field so that header-processing software can immediately decide which header format to look for. The `TrafficClass` and `FlowLabel` fields both relate to quality of service issues.

The `PayloadLen` field gives the length of the packet, excluding the IPv6 header, measured in bytes. The `NextHeader` field cleverly replaces both the IP options and the `Protocol` field of IPv4. If options are required, then they are carried in one or more special headers following the IP header, and this is indicated by the value of the `NextHeader` field. If there are no special headers, the `NextHeader` field is the demux key identifying the higher-level protocol running over IP (e.g., TCP or UDP); that is, it serves the same purpose as the IPv4 `Protocol` field. Also, fragmentation is now handled as an optional header, which means that the fragmentation-related fields of IPv4 are not included in the IPv6 header. The `HopLimit` field is simply the `TTL` of IPv4, renamed to reflect the way it is actually used.

Finally, the bulk of the header is taken up with the source and destination addresses, each of which is 16 bytes (128 bits) long. Thus, the IPv6 header is always 40 bytes long. Considering that IPv6 addresses are four times longer than those of IPv4, this compares quite well with the IPv4 header, which is 20 bytes long in the absence of options.

The way that IPv6 handles options is quite an improvement over IPv4. In IPv4, if any options were present, every router had to parse the entire options field to see if any of the options were relevant. This is because the options were all buried at the end of the IP header, as an unordered collection of '(type, length, value)' tuples. In contrast, IPv6 treats options as *extension headers* that must, if present, appear in a specific order. This means that each router can quickly determine if any of the options are relevant to it; in most cases, they will not be. Usually this can be determined by just looking at the `NextHeader` field. The end result is that option processing is much more efficient in IPv6, which is an important factor in router performance. In addition, the new formatting of options as extension headers means that they can be of arbitrary length, whereas in IPv4 they were limited to 44 bytes at most. We will see how some of the options are used below.

Each option has its own type of extension header. The type of each extension header is identified by the value of the `NextHeader` field in the header that precedes it, and each extension header contains a `NextHeader` field to identify the header following it. The last extension header will be followed by a transport-layer header (e.g., TCP) and in this case the value of the `NextHeader` field is the same as the value of the `Protocol` field would be in an IPv4 header. Thus, the `NextHeader` field does double duty; it may either identify the type of extension header to follow, or, in the last extension header, it serves as a demux key to identify the higher-layer protocol running over IPv6.

Consider the example of the fragmentation header, shown in Figure 4.13. This header provides functionality similar to the fragmentation fields in the IPv4 header, but it is only present if fragmentation is necessary. Assuming it is the only extension header present, then the `NextHeader` field of the IPv6 header would contain the value `44`, which is the value assigned to indicate the fragmentation header. The `NextHeader` field of the fragmentation header itself contains a value describing the header that follows it. Again, assuming no other extension headers are present, then the next header might be the TCP header, which results in `NextHeader` containing the value `6`, just as the `Protocol` field would in IPv4. If the fragmentation

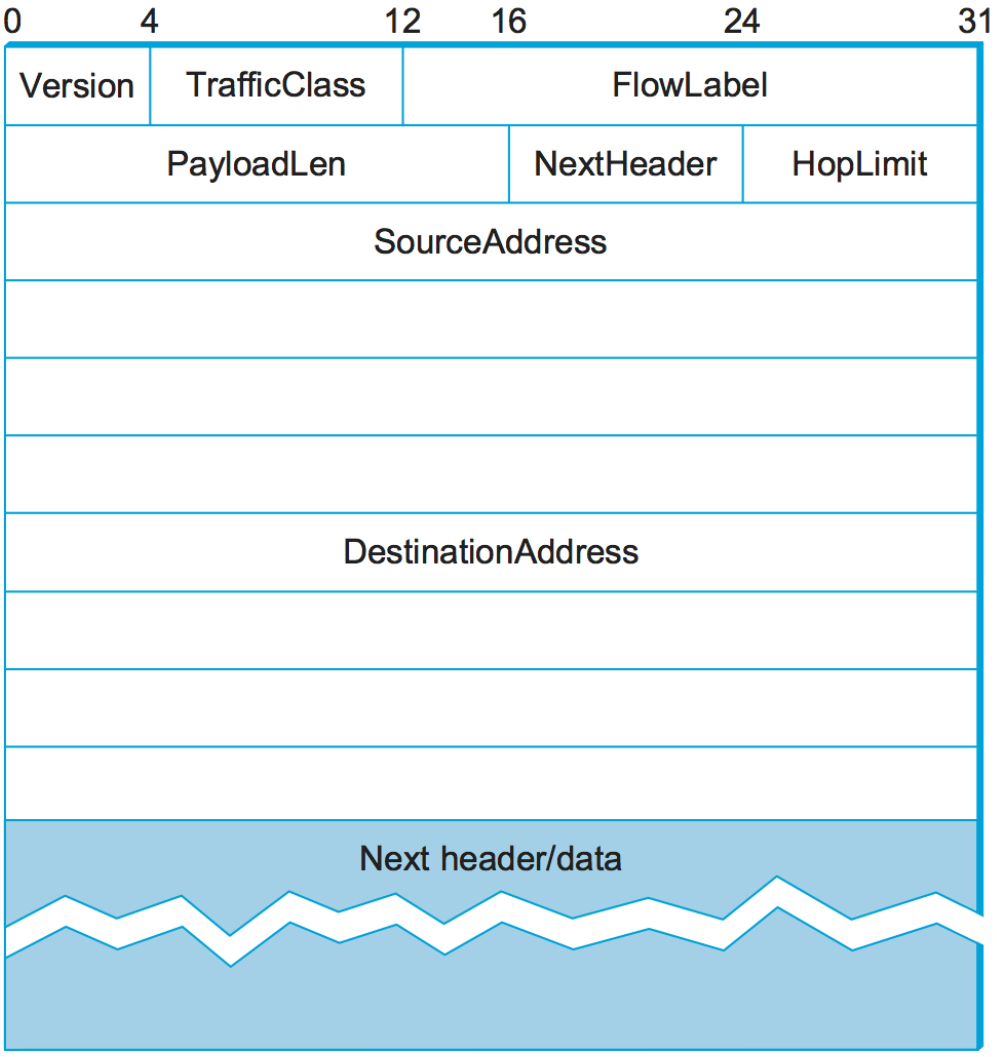


Figure 4.12.: IPv6 packet header.

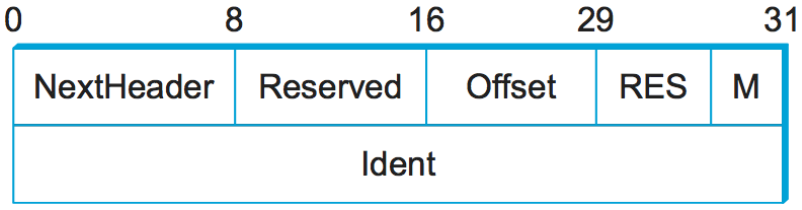


Figure 4.13.: IPv6 fragmentation extension header.

header were followed by, say, an authentication header, then the fragmentation header's `NextHeader` field would contain the value `51`.

4.2.4 Advanced Capabilities

As mentioned at the beginning of this section, the primary motivation behind the development of IPv6 was to support the continued growth of the Internet. Once the IP header had to be changed for the sake of the addresses, however, the door was open for a wide variety of other changes, two of which we describe below. But IPv6 includes several additional features, most of which are covered elsewhere in this book; e.g., mobility, security, quality-of-service. It is interesting to note that, in most of these areas, the IPv4 and IPv6 capabilities have become virtually indistinguishable, so that the main driver for IPv6 remains the need for larger addresses.

Autoconfiguration

While the Internet's growth has been impressive, one factor that has inhibited faster acceptance of the technology is the fact that getting connected to the Internet has typically required a fair amount of system administration expertise. In particular, every host that is connected to the Internet needs to be configured with a certain minimum amount of information, such as a valid IP address, a subnet mask for the link to which it attaches, and the address of a name server. Thus, it has not been possible to unpack a new computer and connect it to the Internet without some preconfiguration. One goal of IPv6, therefore, is to provide support for autoconfiguration, sometimes referred to as *plug-and-play* operation.

As we saw in the previous chapter, autoconfiguration is possible for IPv4, but it depends on the existence of a server that is configured to hand out addresses and other configuration information to Dynamic Host Configuration Protocol (DHCP) clients. The longer address format in IPv6 helps provide a useful, new form of autoconfiguration called *stateless* autoconfiguration, which does not require a server.

Recall that IPv6 unicast addresses are hierarchical, and that the least significant portion is the interface ID. Thus, we can subdivide the autoconfiguration problem into two parts:

1. Obtain an interface ID that is unique on the link to which the host is attached.
2. Obtain the correct address prefix for this subnet.

The first part turns out to be rather easy, since every host on a link must have a unique link-level address. For example, all hosts on an Ethernet have a unique 48-bit Ethernet address. This can be turned into a valid link-local use address by adding the appropriate prefix from :numref"*Table %s <fig-v6tab>* (`1111 1110 10`) followed by enough 0s to make up 128 bits. For some devices—for example, printers or hosts on a small routerless network that do not connect to any other networks—this address may be perfectly adequate. Those devices that need a globally valid address depend on a router on the same link to periodically advertise the appropriate prefix for the link. Clearly, this requires that the router be configured with the correct address prefix, and that this prefix be chosen in such a way that there is enough space at the end (e.g., 48 bits) to attach an appropriate link-level address.

The ability to embed link-level addresses as long as 48 bits into IPv6 addresses was one of the reasons for choosing such a large address size. Not only does 128 bits allow the embedding, but it leaves plenty of space for the multilevel hierarchy of addressing that we discussed above.

Source-Directed Routing

Another of IPv6's extension headers is the routing header. In the absence of this header, routing for IPv6 differs very little from that of IPv4 under CIDR. The routing header contains a list of IPv6 addresses that represent nodes or topological areas that the packet should visit en route to its destination. A topological area may be, for example, a backbone provider's network. Specifying that packets must visit this network would be a way of implementing provider selection on a packet-by-packet basis. Thus, a host could say that it wants some packets to go through a provider that is cheap, others through a provider that provides high reliability, and still others through a provider that the host trusts to provide security.

To provide the ability to specify topological entities rather than individual nodes, IPv6 defines an *anycast* address. An anycast address is assigned to a set of interfaces, and packets sent to that address will go to the "nearest" of those interfaces, with nearest being determined by the routing protocols. For example, all the routers of a backbone provider could be assigned a single anycast address, which would be used in the routing header.

4.3 Multicast

Multi-access networks like Ethernet implement multicast in hardware. There are, however, applications that need a broader multicasting capability that is effective at the scale of the Internet. For example, when a radio station is broadcast over the Internet, the same data must be sent to all the hosts where a user has tuned in to that station. In that example, the communication is one-to-many. Other examples of one-to-many applications include transmitting the same news, current stock prices, software updates, or TV channels to multiple hosts. The latter example is commonly called IPTV.

There are also applications whose communication is many-to-many, such as multimedia teleconferencing, online multiplayer gaming, or distributed simulations. In such cases, members of a group receive data from multiple senders, typically each other. From any particular sender, they all receive the same data.

Normal IP communication, in which each packet must be addressed and sent to a single host, is not well suited to such applications. If an application has data to send to a group, it would have to send a separate packet with the identical data to each member of the group. This redundancy consumes more bandwidth than necessary. Furthermore, the redundant traffic is not distributed evenly but rather is focused around the sending host, and may easily exceed the capacity of the sending host and the nearby networks and routers.

To better support many-to-many and one-to-many communication, IP provides an IP-level multicast analogous to the link-level multicast provided by multi-access networks like Ethernet. Now that we are introducing the concept of multicast for IP, we also need a term for the traditional one-to-one service of IP that has been described so far: That service is referred to as *unicast*.

The basic IP multicast model is a many-to-many model based on multicast *groups*, where each group has its own IP *multicast address*. The hosts that are members of a group receive copies of any packets sent to that group's multicast address. A host can be in multiple groups, and it can join and leave groups freely by telling its local router using a protocol that we will discuss shortly. Thus, while we think of unicast addresses as being associated with a node or an interface, multicast addresses are associated with an abstract group, the membership of which changes dynamically over time. Further, the original IP multicast service model allows *any* host to send multicast traffic to a group; it doesn't have to be a member of the group, and there may be any number of such senders to a given group.

Using IP multicast to send the identical packet to each member of the group, a host sends a single copy of the packet addressed to the group's multicast address. The sending host doesn't need to know the individual unicast IP address of each member of the group because, as we will see, that knowledge is distributed among the routers in the internetwork. Similarly, the sending host doesn't need to send multiple copies of the packet because the routers will make copies whenever they have to forward the packet over more than one link. Compared to using unicast IP to deliver the same packets to many receivers, IP multicast is more scalable because it eliminates the redundant traffic (packets) that would have been sent many times over the same links, especially those near to the sending host.

IP's original many-to-many multicast has been supplemented with support for a form of one-to-many multicast. In this model of one-to-many multicast, called *Source-Specific Multicast* (SSM), a receiving host specifies both a multicast group and a specific sending host. The receiving host would then receive multicasts addressed to the specified group, but only if they are from the specified sender. Many Internet multicast applications (e.g., radio broadcasts) fit the SSM model. To contrast it with SSM, IP's original many-to-many model is sometimes referred to as *Any Source Multicast* (ASM).

A host signals its desire to join or leave a multicast group by communicating with its local router using a special protocol for just that purpose. In IPv4, that protocol is the *Internet Group Management Protocol* (IGMP); in IPv6, it is *Multicast Listener Discovery* (MLD). The router then has the responsibility for making multicast behave correctly with regard to that host. Because a host may fail to leave a multicast group when it should (after a crash or other failure, for example), the router periodically polls the network to determine which groups are still of interest to the attached hosts.

4.3.1 Multicast Addresses

IP has a subrange of its address space reserved for multicast addresses. In IPv4, these addresses are assigned in the class D address space, and IPv6 also has a portion of its address space reserved for multicast group addresses. Some subranges of the multicast ranges are reserved for intradomain multicast, so they can be reused independently by different domains.

There are thus 28 bits of possible multicast address in IPv4 when we ignore the prefix shared by all multicast addresses. This presents a problem when attempting to take advantage of hardware multicasting on a local area network (LAN). Let's take the case of Ethernet. Ethernet multicast addresses have only 23 bits when we ignore their shared prefix. In other words, to take advantage of Ethernet multicasting, IP has to map 28-bit IP multicast addresses into 23-bit Ethernet multicast addresses. This is implemented by taking the low-order 23 bits of any IP multicast address to use as its Ethernet multicast address and ignoring the high-order 5 bits. Thus, 32 (2^5) IP addresses map into each one of the Ethernet addresses.

> In this section we use Ethernet as a canonical example of a networking technology that supports multicast in hardware, but the same is also true of PON (Passive Optical Networks), which is the access network technology often used to deliver fiber-to-the-home. In fact, IP Multicast over PON is now a common way to deliver IPTV to homes.

When a host on an Ethernet joins an IP multicast group, it configures its Ethernet interface to receive any packets with the corresponding Ethernet multicast address. Unfortunately, this causes the receiving host to receive not only the multicast traffic it desired but also traffic sent to any of the other 31 IP multicast groups that map to the same Ethernet address, if they are routed to that Ethernet. Therefore, IP at the receiving host must examine the IP header of any multicast packet to determine whether the packet really belongs to the desired group. In summary, the mismatch of multicast address sizes means that multicast traffic may

place a burden on hosts that are not even interested in the group to which the traffic was sent. Fortunately, in some switched networks (such as switched Ethernet) this problem can be mitigated by schemes wherein the switches recognize unwanted packets and discard them.

One perplexing question is how senders and receivers learn which multicast addresses to use in the first place. This is normally handled by out-of-band means, and there are some quite sophisticated tools to enable group addresses to be advertised on the Internet.

4.3.2 Multicast Routing (DVMRP, PIM, MSDP)

A router's unicast forwarding tables indicate, for any IP address, which link to use to forward the unicast packet. To support multicast, a router must additionally have multicast forwarding tables that indicate, based on multicast address, which links—possibly more than one—to use to forward the multicast packet (the router duplicates the packet if it is to be forwarded over multiple links). Thus, where unicast forwarding tables collectively specify a set of paths, multicast forwarding tables collectively specify a set of trees: *multicast distribution trees*. Furthermore, to support Source-Specific Multicast (and, it turns out, for some types of Any Source Multicast), the multicast forwarding tables must indicate which links to use based on the combination of multicast address and the (unicast) IP address of the source, again specifying a set of trees.

Multicast routing is the process by which the multicast distribution trees are determined or, more concretely, the process by which the multicast forwarding tables are built. As with unicast routing, it is not enough that a multicast routing protocol "work"; it must also scale reasonably well as the network grows, and it must accommodate the autonomy of different routing domains.

DVMRP

Distance-vector routing used in unicast can be extended to support multicast. The resulting protocol is called *Distance Vector Multicast Routing Protocol*, or DVMRP. DVMRP was the first multicast routing protocol to see widespread use.

Recall that, in the distance-vector algorithm, each router maintains a table of `Destination, Cost, NextHop` tuples, and exchanges a list of `(Destination, Cost)` pairs with its directly connected neighbors. Extending this algorithm to support multicast is a two-stage process. First, we create a broadcast mechanism that allows a packet to be forwarded to all the networks on the internet. Second, we need to refine this mechanism so that it prunes back networks that do not have hosts that belong to the multicast group. Consequently, DVMRP is one of several multicast routing protocols described as *flood-and-prune* protocols.

Given a unicast routing table, each router knows that the current shortest path to a given `destination` goes through `NextHop`. Thus, whenever it receives a multicast packet from source S, the router forwards the packet on all outgoing links (except the one on which the packet arrived) if and only if the packet arrived over the link that is on the shortest path to S (i.e., the packet came *from* the `NextHop` associated with S in the routing table). This strategy effectively floods packets outward from S but does not loop packets back toward S.

There are two major shortcomings to this approach. The first is that it truly floods the network; it has no provision for avoiding LANs that have no members in the multicast group. We address this problem below. The second limitation is that a given packet will be forwarded over a LAN by each of the routers connected

to that LAN. This is due to the forwarding strategy of flooding packets on all links other than the one on which the packet arrived, without regard to whether or not those links are part of the shortest-path tree rooted at the source.

The solution to this second limitation is to eliminate the duplicate broadcast packets that are generated when more than one router is connected to a given LAN. One way to do this is to designate one router as the *parent* router for each link, relative to the source, where only the parent router is allowed to forward multicast packets from that source over the LAN. The router that has the shortest path to source S is selected as the parent; a tie between two routers would be broken according to which router has the smallest address. A given router can learn if it is the parent for the LAN (again relative to each possible source) based upon the distance-vector messages it exchanges with its neighbors.

Notice that this refinement requires that each router keep, for each source, a bit for each of its incident links indicating whether or not it is the parent for that source/link pair. Keep in mind that in an internet setting, a source is a network, not a host, since an internet router is only interested in forwarding packets between networks. The resulting mechanism is sometimes called *Reverse Path Broadcast* (RPB) or *Reverse Path Forwarding* (RPF). The path is reverse because we are considering the shortest path toward the *source* when making our forwarding decisions, as compared to unicast routing, which looks for the shortest path to a given *destination*.

The RPB mechanism just described implements shortest-path broadcast. We now want to prune the set of networks that receives each packet addressed to group G to exclude those that have no hosts that are members of G. This can be accomplished in two stages. First, we need to recognize when a *leaf* network has no group members. Determining that a network is a leaf is easy—if the parent router as described above is the only router on the network, then the network is a leaf. Determining if any group members reside on the network is accomplished by having each host that is a member of group G periodically announce this fact over the network, as described in our earlier description of link-state multicast. The router then uses this information to decide whether or not to forward a multicast packet addressed to G over this LAN.

The second stage is to propagate this "no members of G here" information up the shortest-path tree. This is done by having the router augment the `(Destination, Cost)` pairs it sends to its neighbors with the set of groups for which the leaf network is interested in receiving multicast packets. This information can then be propagated from router to router, so that for each of its links a given router knows for what groups it should forward multicast packets.

Note that including all of this information in the routing update is a fairly expensive thing to do. In practice, therefore, this information is exchanged only when some source starts sending packets to that group. In other words, the strategy is to use RPB, which adds a small amount of overhead to the basic distance-vector algorithm, until a particular multicast address becomes active. At that time, routers that are not interested in receiving packets addressed to that group speak up, and that information is propagated to the other routers.

PIM-SM

Protocol Independent Multicast, or PIM, was developed in response to the scaling problems of earlier multicast routing protocols. In particular, it was recognized that the existing protocols did not scale well in environments where a relatively small proportion of routers want to receive traffic for a certain group. For example, broadcasting traffic to all routers until they explicitly ask to be removed from the distribution is not a good design choice if most routers don't want to receive the traffic in the first place. This situation is sufficiently common that PIM divides the problem space into *sparse mode* and *dense mode,* where sparse

and dense refer to the proportion of routers that will want the multicast. PIM dense mode (PIM-DM) uses a flood-and-prune algorithm like DVMRP and suffers from the same scalability problem. PIM sparse mode (PIM-SM) has become the dominant multicast routing protocol and is the focus of our discussion here. The "protocol independent" aspect of PIM, by the way, refers to the fact that, unlike earlier protocols such as DVMRP, PIM does not depend on any particular sort of unicast routing—it can be used with any unicast routing protocol, as we will see below.

In PIM-SM, routers explicitly join the multicast distribution tree using PIM protocol messages known as `Join` messages. Note the contrast to DVMRP's approach of creating a broadcast tree first and then pruning the uninterested routers. The question that arises is where to send those `Join` messages because, after all, any host (and any number of hosts) could send to the multicast group. To address this, PIM-SM assigns to each group a special router known as the *rendezvous point* (RP). In general, a number of routers in a domain are configured to be candidate RPs, and PIM-SM defines a set of procedures by which all the routers in a domain can agree on the router to use as the RP for a given group. These procedures are rather complex, as they must deal with a wide variety of scenarios, such as the failure of a candidate RP and the partitioning of a domain into two separate networks due to a number of link or node failures. For the rest of this discussion, we assume that all routers in a domain know the unicast IP address of the RP for a given group.

A multicast forwarding tree is built as a result of routers sending `Join` messages to the RP. PIM-SM allows two types of trees to be constructed: a *shared* tree, which may be used by all senders, and a *source-specific* tree, which may be used only by a specific sending host. The normal mode of operation creates the shared tree first, followed by one or more source-specific trees if there is enough traffic to warrant it. Because building trees installs state in the routers along the tree, it is important that the default is to have only one tree for a group, not one for every sender to a group.

When a router sends a `Join` message toward the RP for a group G, it is sent using normal IP unicast transmission. This is illustrated in Figure 4.14(a), in which router R4 is sending a `Join` to the rendezvous point for some group. The initial `Join` message is "wildcarded"; that is, it applies to all senders. A `Join` message clearly must pass through some sequence of routers before reaching the RP (e.g., R2). Each router along the path looks at the `Join` and creates a forwarding table entry for the shared tree, called a (*, G) entry (where * means "all senders"). To create the forwarding table entry, it looks at the interface on which the `Join` arrived and marks that interface as one on which it should forward data packets for this group. It then determines which interface it will use to forward the `Join` toward the RP. This will be the only acceptable interface for incoming packets sent to this group. It then forwards the `Join` toward the RP. Eventually, the message arrives at the RP, completing the construction of the tree branch. The shared tree thus constructed is shown as a solid line from the RP to R4 in Figure 4.14(a).

As more routers send `Join`s toward the RP, they cause new branches to be added to the tree, as illustrated in Figure 4.14(b). Note that, in this case, the `Join` only needs to travel to R2, which can add the new branch to the tree simply by adding a new outgoing interface to the forwarding table entry created for this group. R2 need not forward the `Join` on to the RP. Note also that the end result of this process is to build a tree whose root is the RP.

At this point, suppose a host wishes to send a message to the group. To do so, it constructs a packet with the appropriate multicast group address as its destination and sends it to a router on its local network known as the *designated router* (DR). Suppose the DR is R1 in Figure 4.14. There is no state for this multicast group between R1 and the RP at this point, so instead of simply forwarding the multicast packet, R1 *tunnels* it to the RP. That is, R1 encapsulates the multicast packet inside a PIM `Register` message that it sends to the unicast IP address of the RP. Just like an IP tunnel endpoint, the RP receives the packet addressed to it, looks at the payload of the `Register` message, and finds inside an IP packet addressed to the multicast address

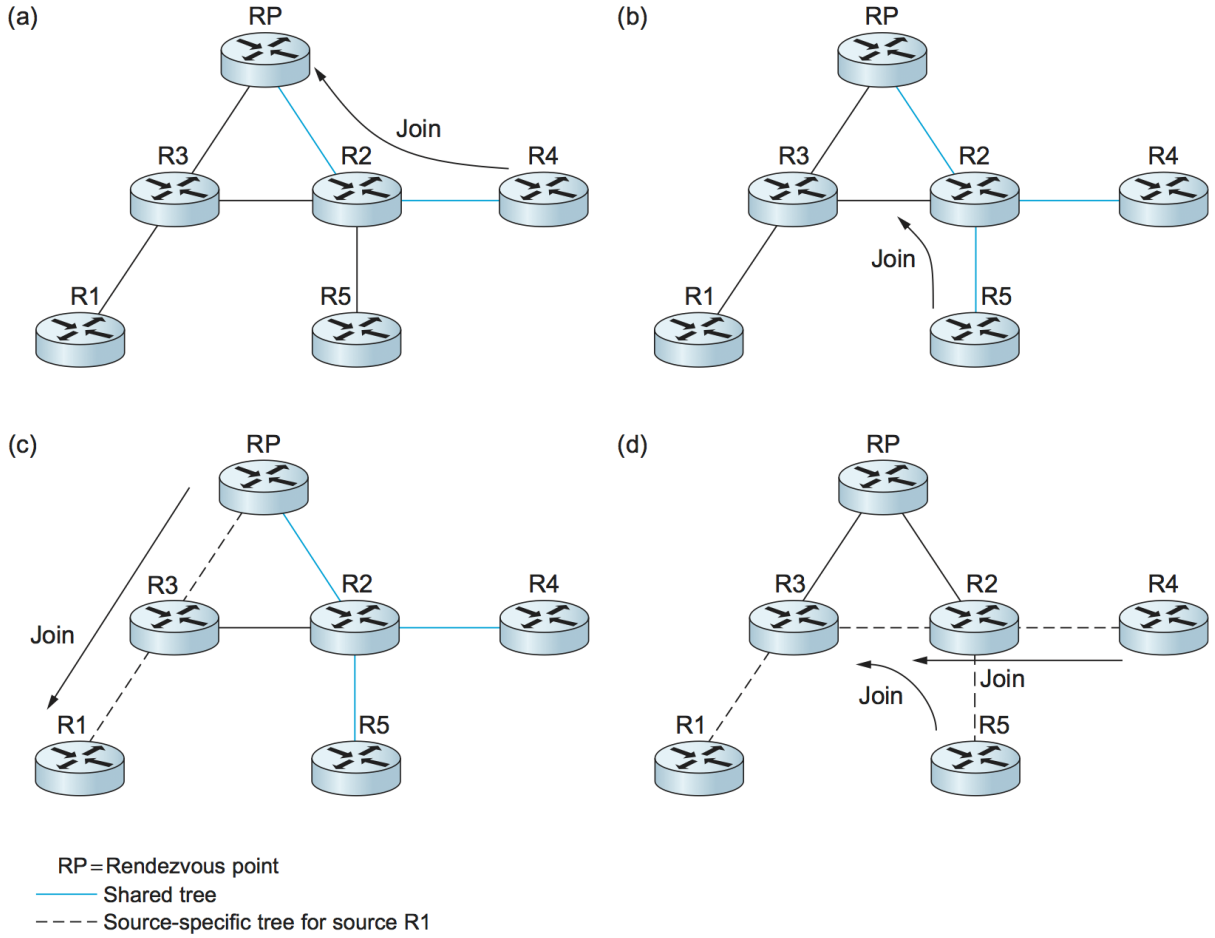


Figure 4.14.: PIM operation: (a) R4 sends a Join message to RP and joins shared tree; (b) R5 joins shared tree; (c) RP builds source-specific tree to R1 by sending a Join message to R1; (d) R4 and R5 build source-specific tree to R1 by sending Join messages to R1.

of this group. The RP, of course, does know what to do with such a packet—it sends it out onto the shared tree of which the RP is the root. In the example of Figure 4.14, this means that the RP sends the packet on to R2, which is able to forward it on to R4 and R5. The complete delivery of a packet from R1 to R4 and R5 is shown in Figure 4.15. We see the tunneled packet travel from R1 to the RP with an extra IP header containing the unicast address of RP, and then the multicast packet addressed to G making its way along the shared tree to R4 and R5.

At this point, we might be tempted to declare success, since all hosts can send to all receivers this way. However, there is some bandwidth inefficiency and processing cost in the encapsulation and decapsulation of packets on the way to the RP, so the RP forces knowledge about this group into the intervening routers so tunneling can be avoided. It sends a `Join` message toward the sending host (Figure 4.14(c)). As this `Join` travels toward the host, it causes the routers along the path (R3) to learn about the group, so that it will be possible for the DR to send the packet to the group as *native* (i.e., not tunneled) multicast packets.

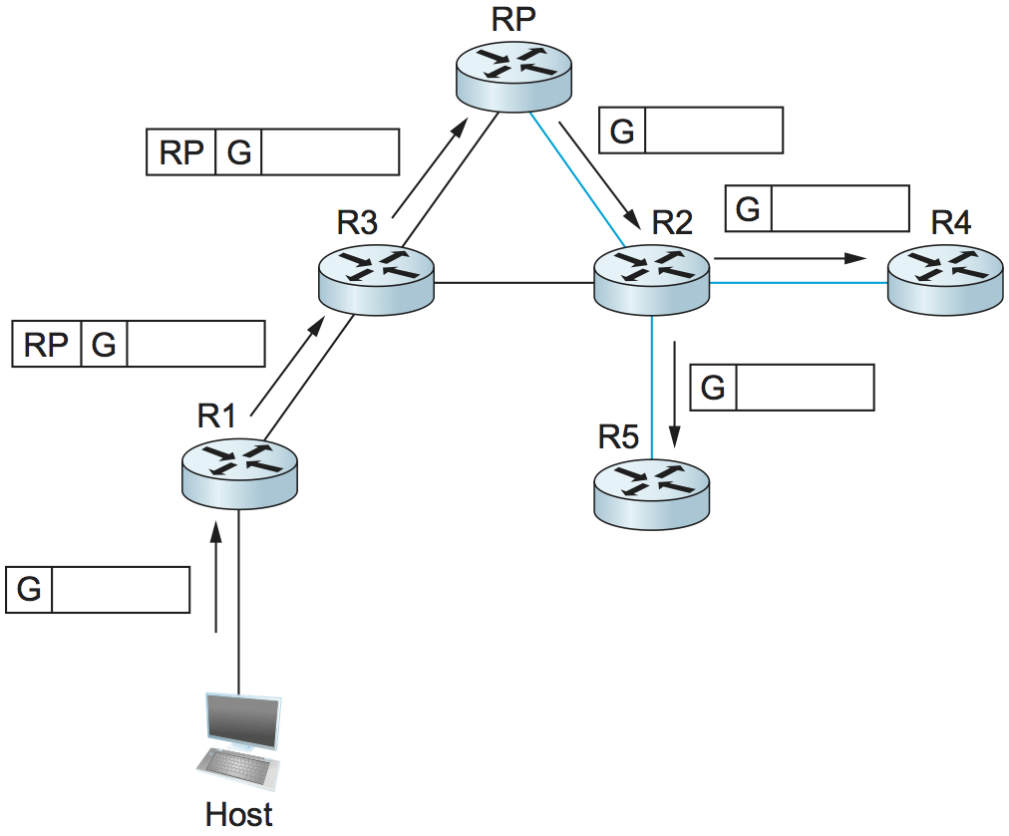


Figure 4.15.: Delivery of a packet along a shared tree. R1 tunnels the packet to the RP, which forwards it along the shared tree to R4 and R5.

An important detail to note at this stage is that the `Join` message sent by the RP to the sending host is specific to that sender, whereas the previous ones sent by R4 and R5 applied to all senders. Thus, the effect of the new `Join` is to create *sender-specific* state in the routers between the identified source and the RP. This is referred to as (S, G) state, since it applies to one sender to one group, and contrasts with the (*, G) state that was installed between the receivers and the RP that applies to all senders. Thus, in Figure 4.14(c), we see a source-specific route from R1 to the RP (indicated by the dashed line) and a tree that is valid for all senders from the RP to the receivers (indicated by the solid line).

The next possible optimization is to replace the entire shared tree with a source-specific tree. This is desirable because the path from sender to receiver via the RP might be significantly longer than the shortest possible path. This again is likely to be triggered by a high data rate being observed from some sender. In this case, the router at the downstream end of the tree—say, R4 in our example—sends a source-specific `Join` toward the source. As it follows the shortest path toward the source, the routers along the way create (S, G) state for this tree, and the result is a tree that has its root at the source, rather than the RP. Assuming both R4 and R5 made the switch to the source-specific tree, we would end up with the tree shown in Figure 4.14(d). Note that this tree no longer involves the RP at all. We have removed the shared tree from this picture to simplify the diagram, but in reality all routers with receivers for a group must stay on the shared tree in case new senders show up.

We can now see why PIM is protocol independent. All of its mechanisms for building and maintaining trees take advantage of unicast routing without depending on any particular unicast routing protocol. The formation of trees is entirely determined by the paths that `Join` messages follow, which is determined by the choice of shortest paths made by unicast routing. Thus, to be precise, PIM is "unicast routing protocol independent," as compared to DVMRP. Note that PIM is very much bound up with the Internet Protocol—it is not protocol independent in terms of network-layer protocols.

The design of PIM-SM again illustrates the challenges in building scalable networks and how scalability is sometimes pitted against some sort of optimality. The shared tree is certainly more scalable than a source-specific tree, in the sense that it reduces the total state in routers to be on the order of the number of groups rather than the number of senders times the number of groups. However, the source-specific tree is likely to be necessary to achieve efficient routing and effective use of link bandwidth.

Interdomain Multicast (MSDP)

PIM-SM has some significant shortcomings when it comes to interdomain multicast. In particular, the existence of a single RP for a group goes against the principle that domains are autonomous. For a given multicast group, all the participating domains would be dependent on the domain where the RP is located. Furthermore, if there is a particular multicast group for which a sender and some receivers shared a single domain, the multicast traffic would still have to be routed initially from the sender to those receivers via whatever domain has the RP for that multicast group. Consequently, the PIM-SM protocol is typically not used across domains, only within a domain.

To extend multicast across domains using PIM-SM, the Multicast Source Discovery Protocol (MSDP) was devised. MSDP is used to connect different domains—each running PIM-SM internally, with its own RPs—by connecting the RPs of the different domains. Each RP has one or more MSDP peer RPs in other domains. Each pair of MSDP peers is connected by a TCP connection over which the MSDP protocol runs. Together, all the MSDP peers for a given multicast group form a loose mesh that is used as a broadcast network. MSDP messages are broadcast through the mesh of peer RPs using the Reverse Path Broadcast algorithm that we discussed in the context of DVMRP.

What information does MSDP broadcast through the mesh of RPs? Not group membership information; when a host joins a group, the furthest that information will flow is its own domain's RP. Instead, it is source—multicast sender—information. Each RP knows the sources in its own domain because it receives a `Register` message whenever a new source arises. Each RP periodically uses MSDP to broadcast `Source Active` messages to its peers, giving the IP address of the source, the multicast group address, and the IP address of the originating RP.

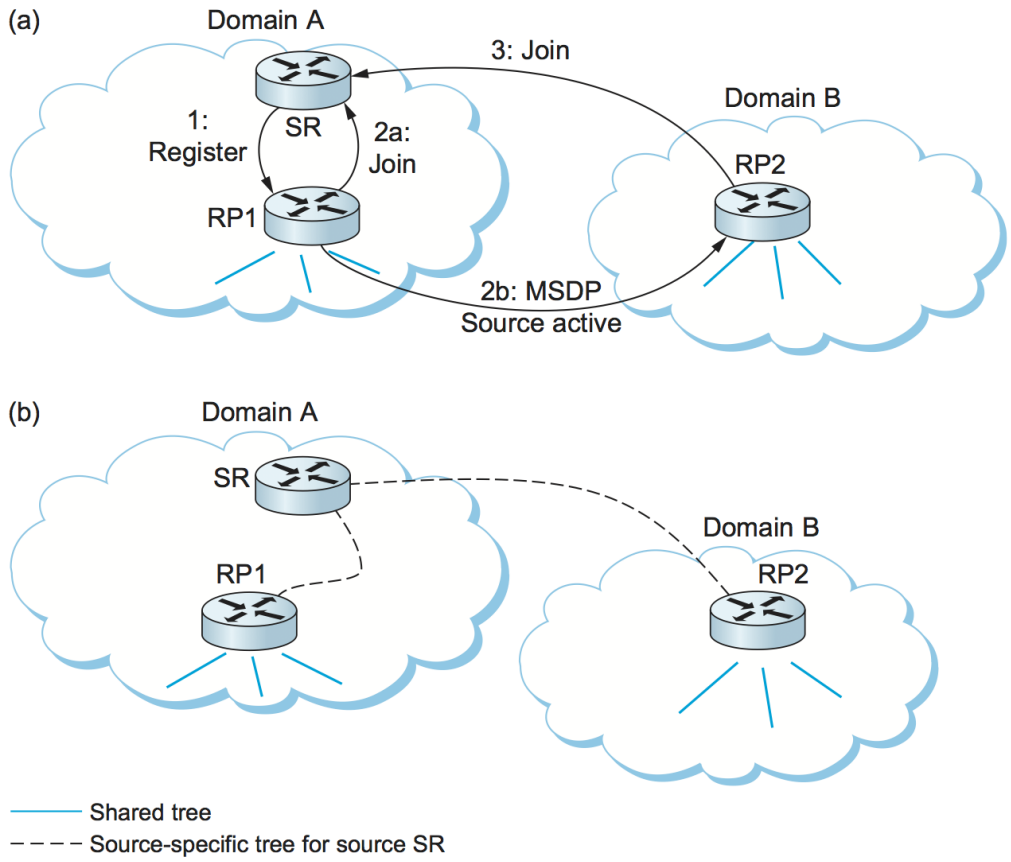


Figure 4.16.: MSDP operation: (a) The source SR sends a Register message to its domain's RP, RP1; then RP1 sends a source-specific Join message to SR and an MSDP Source Active message to its MSDP peer in Domain B, RP2; then RP2 sends a source-specific Join message to SR. (b) As a result, RP1 and RP2 are in the source-specific tree for source SR.

If an MSDP peer RP that receives one of these broadcasts has active receivers for that multicast group, it sends a source-specific `Join`, on that RP's own behalf, to the source host, as shown in Figure 4.16(a). The `Join` message builds a branch of the source-specific tree to this RP, as shown in Figure 4.16(b). The result is that every RP that is part of the MSDP network and has active receivers for a particular multicast group is added to the source-specific tree of the new source. When an RP receives a multicast from the source, the RP uses its shared tree to forward the multicast to the receivers in its domain.

Source-Specific Multicast (PIM-SSM)

The original service model of PIM was, like earlier multicast protocols, a many-to-many model. Receivers joined a group, and any host could send to the group. However, it was recognized in the late 1990s that it might be useful to add a one-to-many model. Lots of multicast applications, after all, have only one legitimate sender, such as the speaker at a conference being sent over the Internet. We already saw that PIM-SM can create source-specific shortest path trees as an optimization after using the shared tree initially. In the original PIM design, this optimization was invisible to hosts—only routers joined source-specific trees. However, once the need for a one-to-many service model was recognized, it was decided to make the source-specific routing capability of PIM-SM explicitly available to hosts. It turns out that this mainly required changes to IGMP and its IPv6 analog, MLD, rather than PIM itself. The newly exposed capability is now known as PIM-SSM (PIM Source-Specific Multicast).

PIM-SSM introduces a new concept, the *channel*, which is the combination of a source address S and a group address G. The group address G looks just like a normal IP multicast address, and both IPv4 and IPv6 have allocated subranges of the multicast address space for SSM. To use PIM-SSM, a host specifies both the group and the source in an IGMP Membership Report message to its local router. That router then sends a PIM-SM source-specific `Join` message toward the source, thereby adding a branch to itself in the source-specific tree, just as was described above for "normal" PIM-SM, but bypassing the whole shared-tree stage. Since the tree that results is source specific, only the designated source can send packets on that tree.

The introduction of PIM-SSM has provided some significant benefits, particularly since there is relatively high demand for one-to-many multicasting:

- Multicasts travel more directly to receivers.
- The address of a channel is effectively a multicast group address plus a source address. Therefore, given that a certain range of multicast group addresses will be used for SSM exclusively, multiple domains can use the same multicast group address independently and without conflict, as long as they use it only with sources in their own domains.
- Because only the specified source can send to an SSM group, there is less risk of attacks based on malicious hosts overwhelming the routers or receivers with bogus multicast traffic.
- PIM-SSM can be used across domains exactly as it is used within a domain, without reliance on anything like MSDP.

SSM, therefore, is quite a useful addition to the multicast service model.

Bidirectional Trees (BIDIR-PIM)

We round off our discussion of multicast with another enhancement to PIM known as *Bidirectional PIM*. BIDIR-PIM is a recent variant of PIM-SM that is well suited to many-to-many multicasting within a domain,

especially when senders and receivers to a group may be the same, as in a multiparty videoconference, for example. As in PIM-SM, would-be receivers join groups by sending IGMP Membership Report messages (which must not be source specific), and a shared tree rooted at an RP is used to forward multicast packets to receivers. Unlike PIM-SM, however, the shared tree also has branches to the *sources*. That wouldn't make any sense with PIM-SM's unidirectional tree, but BIDIR-PIM's trees are bidirectional—a router that receives a multicast packet from a downstream branch can forward it both up the tree and down other branches. The route followed to deliver a packet to any particular receiver goes only as far up the tree as necessary before going down the branch to that receiver. See the multicast route from R1 to R2 in Figure 4.17(b) for an example. R4 forwards a multicast packet downstream to R2 at the same time that it forwards a copy of the same packet upstream to R5.

A surprising aspect of BIDIR-PIM is that there need not actually be an RP. All that is needed is a routable address, which is known as an RP address even though it need not be the address of an RP or anything at all. How can this be? A `Join` from a receiver is forwarded toward the RP address until it reaches a router with an interface on the link where the RP address would reside, where the Join terminates. Figure 4.17(a) shows a `Join` from R2 terminating at R5, and a `Join` from R3 terminating at R6. The upstream forwarding of a multicast packet similarly flows toward the RP address until it reaches a router with an interface on the link where the RP address would reside, but then the router forwards the multicast packet onto that link as the final step of upstream forwarding, ensuring that all other routers on that link receive the packet. Figure 4.17(b) illustrates the flow of multicast traffic originating at R1.

BIDIR-PIM cannot thus far be used across domains. On the other hand, it has several advantages over PIM-SM for many-to-many multicast within a domain:

- There is no source registration process because the routers already know how to route a multicast packet toward the RP address.
- The routes are more direct than those that use PIM-SM's shared tree because they go only as far up the tree as necessary, not all the way to the RP.
- Bidirectional trees use much less state than the source-specific trees of PIM-SM because there is never any source-specific state. (On the other hand, the routes will be longer than those of source-specific trees.)
- The RP cannot be a bottleneck, and indeed no actual RP is needed.

One conclusion to draw from the fact that there are so many different approaches to multicast just within PIM is that multicast is a difficult problem space in which to find optimal solutions. You need to decide which criteria you want to optimize (bandwidth usage, router state, path length, etc.) and what sort of application you are trying to support (one-to-many, many-to-many, etc.) before you can make a choice of the "best" multicast mode for the task.

4.4 Multiprotocol Label Switching

We continue our discussion of enhancements to IP by describing an addition to the Internet architecture that is very widely used but largely hidden from end users. The enhancement, called *Multiprotocol Label Switching* (MPLS), combines some of the properties of virtual circuits with the flexibility and robustness of datagrams. On the one hand, MPLS is very much associated with the Internet Protocol's datagram-based architecture—it relies on IP addresses and IP routing protocols to do its job. On the other hand, MPLS-enabled routers also forward packets by examining relatively short, fixed-length labels, and these

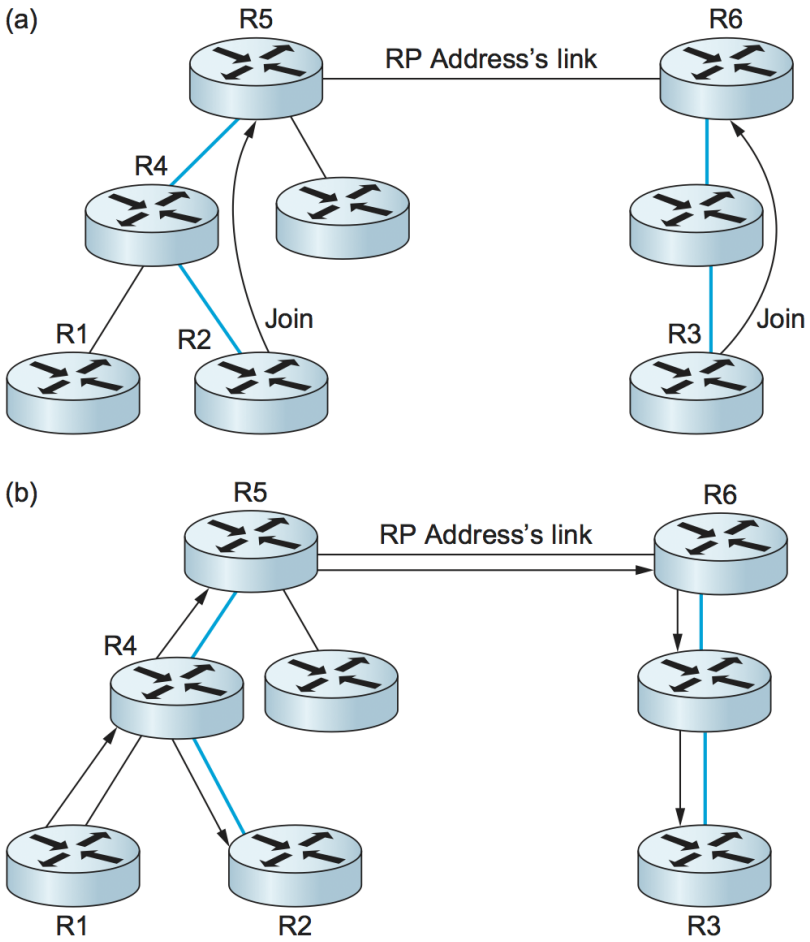


Figure 4.17.: BIDIR-PIM operation: (a) R2 and R3 send Join messages toward the RP address that terminate when they reach a router on the RP address's link. (b) A multicast packet from R1 is forwarded upstream to the RP address's link and downstream wherever it intersects a group member branch.

labels have local scope, just like in a virtual circuit network. It is perhaps this marriage of two seemingly opposed technologies that has caused MPLS to have a somewhat mixed reception in the Internet engineering community.

Before looking at how MPLS works, it is reasonable to ask "what is it good for?" Many claims have been made for MPLS, but there are three main things that it is used for today:

- To enable IP capabilities on devices that do not have the capability to forward IP datagrams in the normal manner
- To forward IP packets along explicit routes—precalculated routes that don't necessarily match those that normal IP routing protocols would select
- To support certain types of virtual private network services

It is worth noting that one of the original goals—improving performance—is not on the list. This has a lot to do with the advances that have been made in forwarding algorithms for IP routers in recent years and with the complex set of factors beyond header processing that determine performance.

The best way to understand how MPLS works is to look at some examples of its use. In the next three sections, we will look at examples to illustrate the three applications of MPLS mentioned above.

4.4.1 Destination-Based Forwarding

One of the earliest publications to introduce the idea of attaching labels to IP packets was a paper by Chandranmenon and Vargese that described an idea called *threaded indices*. A very similar idea is now implemented in MPLS-enabled routers. The following example shows how this idea works.

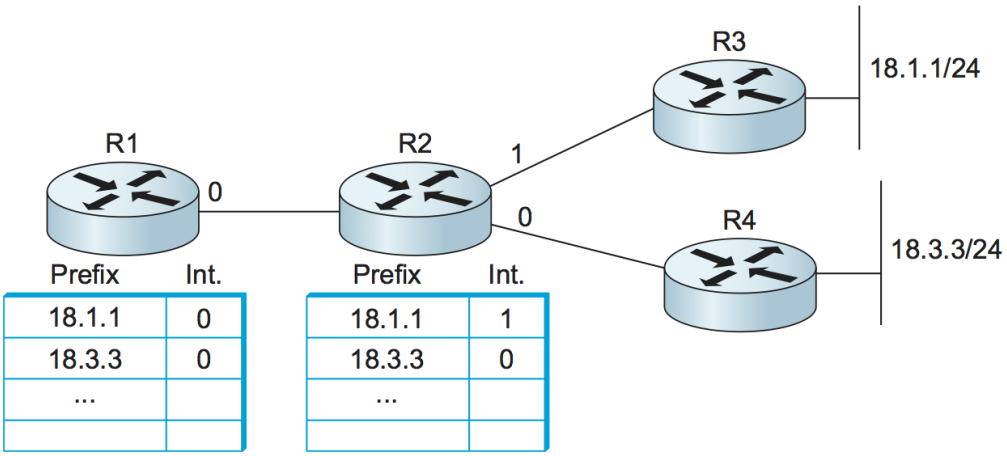


Figure 4.18.: Routing tables in example network.

Consider the network in Figure 4.18. Each of the two routers on the far right (R3 and R4) has one connected network, with prefixes `18.1.1/24` and `18.3.3/24`. The remaining routers (R1 and R2) have routing tables that indicate which outgoing interface each router would use when forwarding packets to one of those two networks.

When MPLS is enabled on a router, the router allocates a label for each prefix in its routing table and advertises both the label and the prefix that it represents to its neighboring routers. This advertisement is

carried in the Label Distribution Protocol. This is illustrated in Figure 4.19. Router R2 has allocated the label value 15 for the prefix 18.1.1 and the label value 16 for the prefix 18.3.3. These labels can be chosen at the convenience of the allocating router and can be thought of as indices into the routing table. After allocating the labels, R2 advertises the label bindings to its neighbors; in this case, we see R2 advertising a binding between the label 15 and the prefix 18.1.1 to R1. The meaning of such an advertisement is that R2 has said, in effect, "Please attach the label 15 to all packets sent to me that are destined to prefix 18.1.1." R1 stores the label in a table alongside the prefix that it represents as the remote or outgoing label for any packets that it sends to that prefix.

In Figure 4.19(c), we see another label advertisement from router R3 to R2 for the prefix 18.1.1, and R2 places the remote label that it learned from R3 in the appropriate place in its table.

At this point, we can look at what happens when a packet is forwarded in this network. Suppose a packet destined to the IP address 18.1.1.5 arrives from the left to router R1. R1 in this case is referred to as a *Label Edge Router* (LER); an LER performs a complete IP lookup on arriving IP packets and then applies labels to them as a result of the lookup. In this case, R1 would see that 18.1.1.5 matches the prefix 18.1.1 in its forwarding table and that this entry contains both an outgoing interface and a remote label value. R1 therefore attaches the remote label 15 to the packet before sending it.

When the packet arrives at R2, R2 looks only at the label in the packet, not the IP address. The forwarding table at R2 indicates that packets arriving with a label value of 15 should be sent out interface 1 and that they should carry the label value 24, as advertised by router R3. R2 therefore rewrites, or swaps, the label and forwards it on to R3.

What has been accomplished by all this application and swapping of labels? Observe that when R2 forwarded the packet in this example it never actually needed to examine the IP address. Instead, R2 looked only at the incoming label. Thus, we have replaced the normal IP destination address lookup with a label lookup. To understand why this is significant, it helps to recall that, although IP addresses are always the same length, IP prefixes are of variable length, and the IP destination address lookup algorithm needs to find the *longest match*—the longest prefix that matches the high order bits in the IP address of the packet being forwarded. By contrast, the label forwarding mechanism just described is an *exact match* algorithm. It is possible to implement a very simple exact match algorithm, for example, by using the label as an index into an array, where each element in the array is one line in the forwarding table.

Note that, while the forwarding algorithm has been changed from longest match to exact match, the routing algorithm can be any standard IP routing algorithm (e.g., OSPF). The path that a packet will follow in this environment is the exact same path that it would have followed if MPLS were not involved: the path chosen by the IP routing algorithms. All that has changed is the forwarding algorithm.

An important fundamental concept of MPLS is illustrated by this example. Every MPLS label is associated with a *forwarding equivalence class* (FEC)—a set of packets that are to receive the same forwarding treatment in a particular router. In this example, each prefix in the routing table is an FEC; that is, all packets that match the prefix 18.1.1—no matter what the low order bits of the IP address are—get forwarded along the same path. Thus, each router can allocate one label that maps to 18.1.1, and any packet that contains an IP address whose high order bits match that prefix can be forwarded using that label.

As we will see in the subsequent examples, FECs are a very powerful and flexible concept. FECs can be formed using almost any criteria; for example, all the packets corresponding to a particular customer could be considered to be in the same FEC.

Returning to the example at hand, we observe that changing the forwarding algorithm from normal IP forwarding to label swapping has an important consequence: Devices that previously didn't know how to

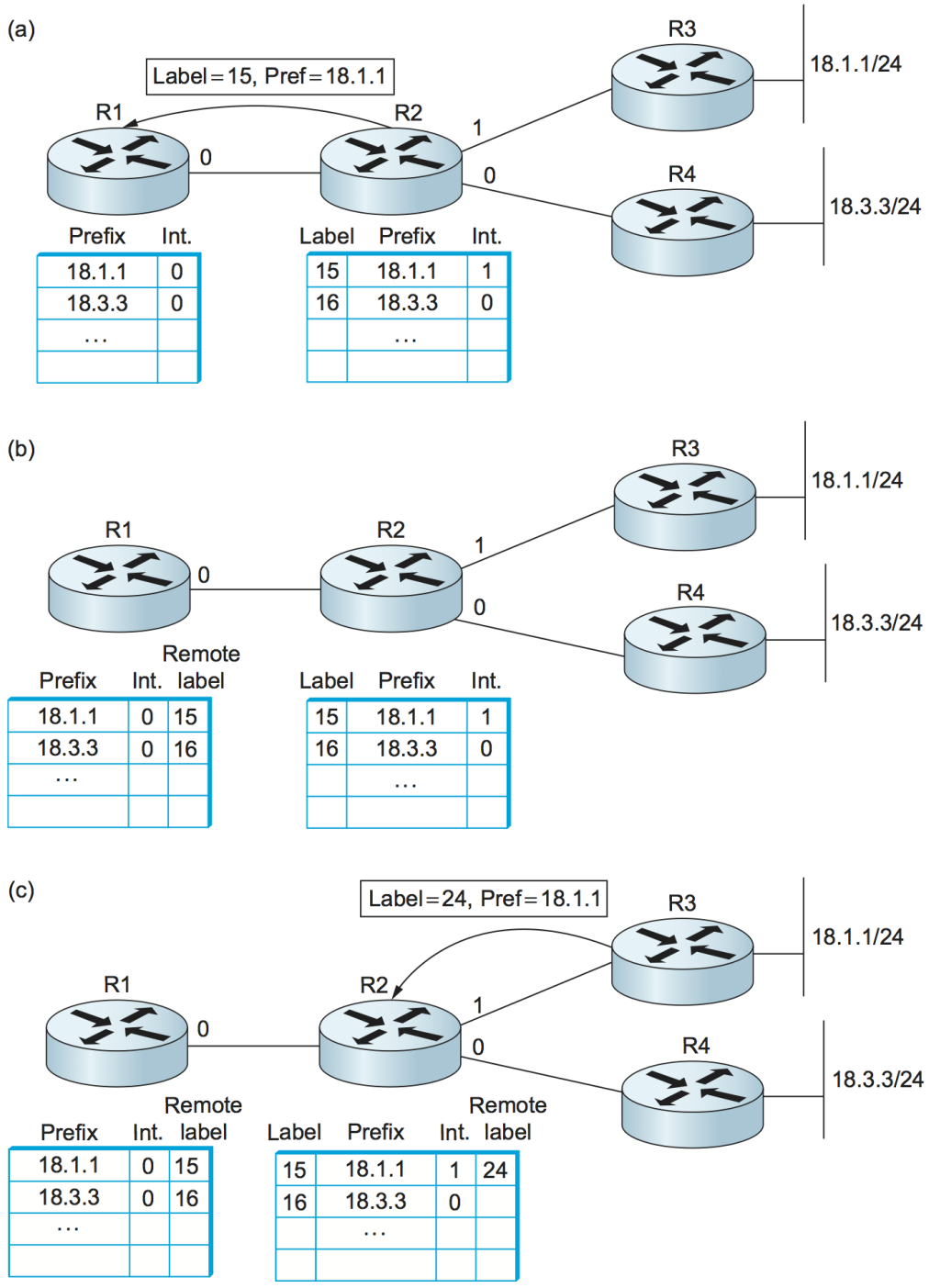


Figure 4.19.: (a) R2 allocates labels and advertises bindings to R1. (b) R1 stores the received labels in a table. (c) R3 advertises another binding, and R2 stores the received label in a table.

forward IP packets can be used to forward IP traffic in an MPLS network. The most notable early application of this result was to ATM switches, which can support MPLS without any changes to their forwarding hardware. ATM switches support the label-swapping forwarding algorithm just described, and by providing these switches with IP routing protocols and a method to distribute label bindings they could be turned into *Label Switching Routers* (LSRs)—devices that run IP control protocols but use the label switching forwarding algorithm. More recently, the same idea has been applied to optical switches.

Before we consider the purported benefits of turning an ATM switch into an LSR, we should tie up some loose ends. We have said that labels are "attached" to packets, but where exactly are they attached? The answer depends on the type of link on which packets are carried. Two common methods for carrying labels on packets are shown in Figure 4.20. When IP packets are carried as complete frames, as they are on most link types including Ethernet and PPP, the label is inserted as a "shim" between the layer 2 header and the IP (or other layer 3) header, as shown in the lower part of the figure. However, if an ATM switch is to function as an MPLS LSR, then the label needs to be in a place where the switch can use it, and that means it needs to be in the ATM cell header, exactly where one would normally find the virtual circuit identifier (VCI) and virtual path identifier (VPI) fields.

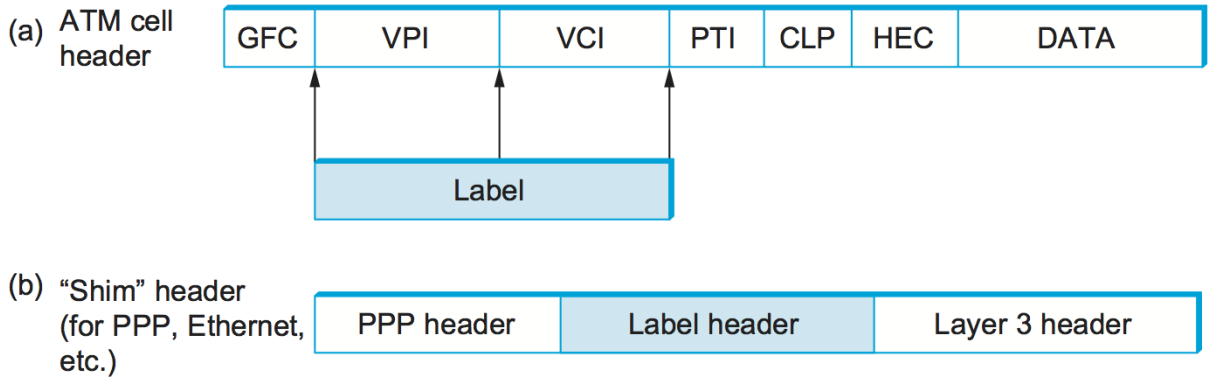


Figure 4.20.: (a) Label on an ATM-encapsulated packet; (b) label on a frame-encapsulated packet.

Having now devised a scheme by which an ATM switch can function as an LSR, what have we gained? One thing to note is that we could now build a network that uses a mixture of conventional IP routers, label edge routers, and ATM switches functioning as LSRs, and they would all use the same routing protocols. To understand the benefits of using the same protocols, consider the alternative. In Figure 4.21(a), we see a set of routers interconnected by virtual circuits over an ATM network, a configuration called an *overlay* network. At one point in time, networks of this type were often built because commercially available ATM switches supported higher total throughput than routers. Today, networks like this are less common because routers have caught up with and even surpassed ATM switches. However, these networks still exist because of the significant installed base of ATM switches in network backbones, which in turn is partly a result of ATM's ability to support a range of capabilities such as circuit emulation and virtual circuit services.

In an overlay network, each router would potentially be connected to each of the other routers by a virtual circuit, but in this case for clarity we have just shown the circuits from R1 to all of its peer routers. R1 has five routing neighbors and needs to exchange routing protocol messages with all of them—we say that R1 has five routing adjacencies. By contrast, in Figure 4.21(b), the ATM switches have been replaced with LSRs. There are no longer virtual circuits interconnecting the routers. Thus, R1 has only one adjacency, with LSR1. In large networks, running MPLS on the switches leads to a significant reduction in the number of adjacencies that each router must maintain and can greatly reduce the amount of work that the routers have to do to keep each other informed of topology changes.

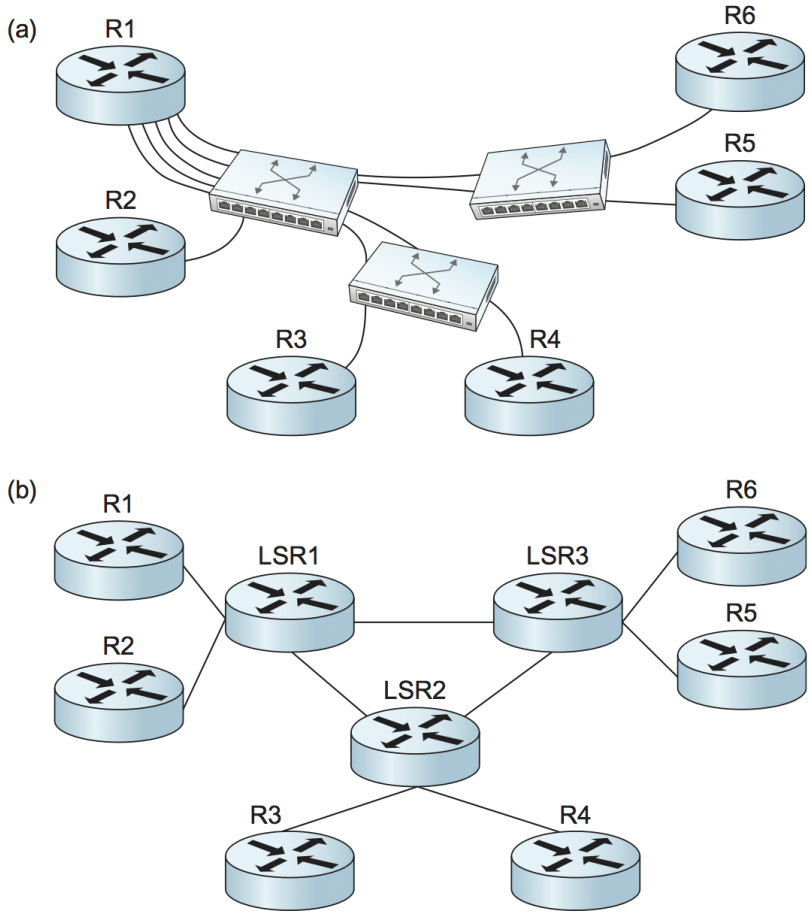


Figure 4.21.: (a) Routers connect to each other using an overlay of virtual circuits. (b) Routers peer directly with LSRs.

A second benefit of running the same routing protocols on edge routers and on the LSRs is that the edge routers now have a full view of the topology of the network. This means that if some link or node fails inside the network, the edge routers will have a better chance of picking a good new path than if the ATM switches rerouted the affected VCs without the knowledge of the edge routers.

Note that the step of "replacing" ATM switches with LSRs is actually achieved by changing the protocols running on the switches, but typically no change to the forwarding hardware is needed; that is, an ATM switch can often be converted to an MPLS LSR by upgrading only its software. Furthermore, an MPLS LSR might continue to support standard ATM capabilities at the same time as it runs the MPLS control protocols, in what is referred to as "ships in the night" mode.

The idea of running IP control protocols on devices that are unable to forward IP packets natively has been extended to Wavelength Division Multiplexing (WDM) and Time Division Multiplexing (TDM) networks (e.g., SONET). This is known as *Generalized MPLS* (GMPLS). Part of the motivation for GMPLS was to provide routers with topological knowledge of an optical network, just as in the ATM case. Even more important was the fact that there were no standard protocols for controlling optical devices, so MPLS proved to be a natural fit for that job.

4.4.2 Explicit Routing

IP has a source routing option, but it is not widely used for several reasons, including the fact that only a limited number of hops can be specified and because it is usual processed outside the "fast path" on most routers.

MPLS provides a convenient way to add capabilities similar to source-routing to IP networks, although the capability is more often referred to as *explicit routing* rather than *source routing*. One reason for the distinction is that it usually isn't the real source of the packet that picks the route. More often it is one of the routers inside a service provider's network. Figure 4.22 shows an example of how the explicit routing capability of MPLS might be applied. This sort of network is often called a *fish* network because of its shape (the routers R1 and R2 form the tail; R7 is at the head).

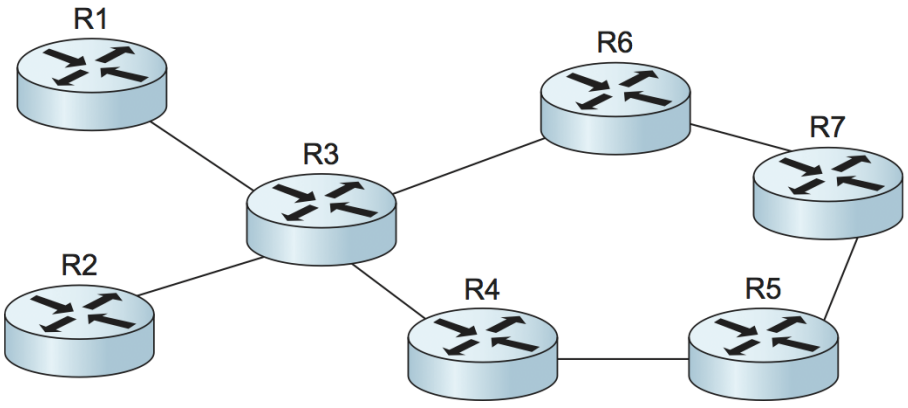


Figure 4.22.: A network requiring explicit routing.

Suppose that the operator of the network in Figure 4.22 has determined that any traffic flowing from R1 to R7 should follow the path R1-R3-R6-R7 and that any traffic going from R2 to R7 should follow the path R2-R3-R4-R5-R7. One reason for such a choice would be to make good use of the capacity available along the two distinct paths from R3 to R7. We can think of the R1-to-R7 traffic as constituting one forwarding

equivalence class, and the R2-to-R7 traffic constitutes a second FEC. Forwarding traffic in these two classes along different paths is difficult with normal IP routing, because R3 doesn't normally look at where traffic came from in making its forwarding decisions.

Because MPLS uses label swapping to forward packets, it is easy enough to achieve the desired routing if the routers are MPLS enabled. If R1 and R2 attach distinct labels to packets before sending them to R3—thus identifying them as being in different FECs—then R3 can forward packets from R1 and R2 along different paths. The question that then arises is how do all the routers in the network agree on what labels to use and how to forward packets with particular labels? Clearly, we can't use the same procedures as described in the preceding section to distribute labels, because those procedures establish labels that cause packets to follow the normal paths picked by IP routing, which is exactly what we are trying to avoid. Instead, a new mechanism is needed. It turns out that the protocol used for this task is the Resource Reservation Protocol (RSVP). For now it suffices to say that it is possible to send an RSVP message along an explicitly specified path (e.g., R1-R3-R6-R7) and use it to set up label forwarding table entries all along that path. This is very similar to the process of establishing a virtual circuit.

One of the applications of explicit routing is *traffic engineering*, which refers to the task of ensuring that sufficient resources are available in a network to meet the demands placed on it. Controlling exactly which paths the traffic flows on is an important part of traffic engineering. Explicit routing can also help to make networks more resilient in the face of failure, using a capability called *fast reroute*. For example, it is possible to precalculate a path from router A to router B that explicitly avoids a certain link L. In the event that link L fails, router A could send all traffic destined to B down the precalculated path. The combination of precalculation of the backup path and the explicit routing of packets along the path means that A doesn't need to wait for routing protocol packets to make their way across the network or for routing algorithms to be executed by various other nodes in the network. In certain circumstances, this can significantly reduce the time taken to reroute packets around a point of failure.

One final point to note about explicit routing is that explicit routes need not be calculated by a network operator as in the above example. Routers can use various algorithms to calculate explicit routes automatically. The most common of these is *constrained shortest path first* (CSPF), which is a link-state algorithm, but which also takes various *constraints* into account. For example, if it was required to find a path from R1 to R7 that could carry an offered load of 100 Mbps, we could say that the constraint is that each link must have at least 100 Mbps of available capacity. CSPF addresses this sort of problem.

4.4.3 Virtual Private Networks and Tunnels

One way to build virtual private networks (VPNs) is to use tunnels. It turns out that MPLS can be thought of as a way to build tunnels, and this makes it suitable for building VPNs of various types.

The simplest form of MPLS VPN to understand is a layer 2 VPN. In this type of VPN, MPLS is used to tunnel layer 2 data (such as Ethernet frames or ATM cells) across a network of MPLS-enabled routers. One reason for tunnels is to provide some sort of network service (such as multicast) that is not supported by some routers in the network. The same logic applies here: IP routers are not ATM switches, so you cannot provide an ATM virtual circuit service across a network of conventional routers. However, if you had a pair of routers interconnected by a tunnel, they could send ATM cells across the tunnel and emulate an ATM circuit. The term for this technique within the IETF is *pseudowire emulation*. Figure 4.23 illustrates the idea.

We have already seen how IP tunnels are built: The router at the entrance of the tunnel wraps the data to

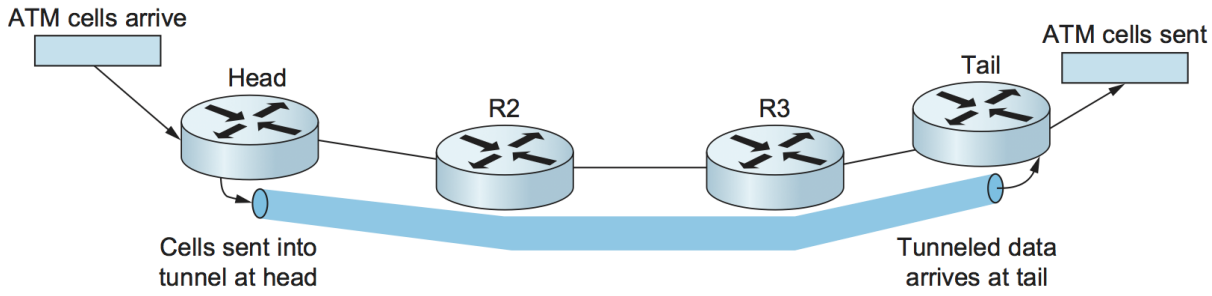


Figure 4.23.: An ATM circuit is emulated by a tunnel.

be tunneled in an IP header (the *tunnel header*), which represents the address of the router at the far end of the tunnel and sends the data like any other IP packet. The receiving router receives the packet with its own address in the header, strips the tunnel header, and finds the data that was tunneled, which it then processes. Exactly what it does with that data depends on what it is. For example, if it were another IP packet, it would then be forwarded on like a normal IP packet. However, it need not be an IP packet, as long as the receiving router knows what to do with non-IP packets. We'll return to the issue of how to handle non-IP data in a moment.

An MPLS tunnel is not too different from an IP tunnel, except that the tunnel header consists of an MPLS header rather than an IP header. Looking back to our first example, in Figure 4.19, we saw that router R1 attached a label (`15`) to every packet that it sent towards prefix 18.1.1. Such a packet would then follow the path R1-R2-R3, with each router in the path examining only the MPLS label. Thus, we observe that there was no requirement that R1 only send IP packets along this path—any data could be wrapped up in the MPLS header and it would follow the same path, because the intervening routers never look beyond the MPLS header. In this regard, an MPLS header is just like an IP tunnel header (except only 4 bytes long instead of 20 bytes). The only issue with sending non-IP traffic along a tunnel, MPLS or otherwise, is what to do with non-IP traffic when it reaches the end of the tunnel. The general solution is to carry some sort of demultiplexing identifier in the tunnel payload that tells the router at the end of the tunnel what to do. It turns out that an MPLS label is a perfect fit for such an identifier. An example will make this clear.

Let's assume we want to tunnel ATM cells from one router to another across a network of MPLS-enabled routers, as in Figure 4.23. Further, we assume that the goal is to emulate an ATM virtual circuit; that is, cells arrive at the entrance, or head, of the tunnel on a certain input port with a certain VCI and should leave the tail end of the tunnel on a certain output port and potentially different VCI. This can be accomplished by configuring the head and tail routers as follows:

- The head router needs to be configured with the incoming port, the incoming VCI, the demultiplexing label for this emulated circuit, and the address of the tunnel end router.
- The tail router needs to be configured with the outgoing port, the outgoing VCI, and the demultiplexing label.

Once the routers are provided with this information, we can see how an ATM cell would be forwarded. Figure 4.24 illustrates the steps.

1. An ATM cell arrives on the designated input port with the appropriate VCI value (101 in this example).
2. The head router attaches the demultiplexing label that identifies the emulated circuit.
3. The head router then attaches a second label, which is the tunnel label that will get the packet to the

tail router. This label is learned by mechanisms just like those described elsewhere in this section.

4. Routers between the head and tail forward the packet using only the tunnel label.
5. The tail router removes the tunnel label, finds the demultiplexing label, and recognizes the emulated circuit.
6. The tail router modifies the ATM VCI to the correct value (202 in this case) and sends it out the correct port.

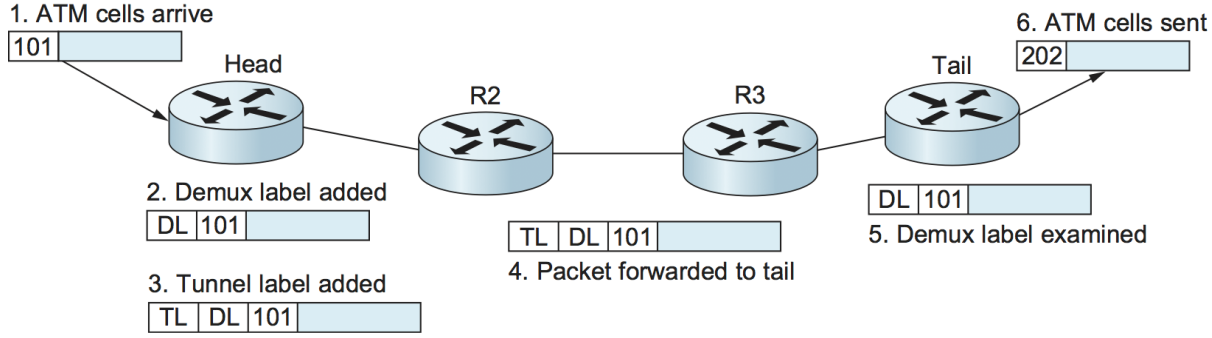


Figure 4.24.: Forward ATM cells along a tunnel.

One item in this example that might be surprising is that the packet has two labels attached to it. This is one of the interesting features of MPLS—labels may be stacked on a packet to any depth. This provides some useful scaling capabilities. In this example, it allows a single tunnel to carry a potentially large number of emulated circuits.

The same techniques described here can be applied to emulate many other layer 2 services, including Frame Relay and Ethernet. It is worth noting that virtually identical capabilities can be provided using IP tunnels; the main advantage of MPLS here is the shorter tunnel header.

Before MPLS was used to tunnel layer 2 services, it was also being used to support layer 3 VPNs. We won't go into the details of layer 3 VPNs, which are quite complex, but we will note that they represent one of the most popular uses of MPLS today. Layer 3 VPNs also use stacks of MPLS labels to tunnel packets across an IP network. However, the packets that are tunneled are themselves IP packets—hence, the name *layer 3* VPNs. In a layer 3 VPN, a single service provider operates a network of MPLS-enabled routers and provides a "virtually private" IP network service to any number of distinct customers. That is, each customer of the provider has some number of sites, and the service provider creates the illusion for each customer that there are no other customers on the network. The customer sees an IP network interconnecting his own sites and no other sites. This means that each customer is isolated from all other customers in terms of both routing and addressing. Customer A can't sent packets directly to customer B, and *vice versa*. Customer A can even use IP addresses that have also been used by customer B. The basic idea is illustrated in Figure 4.25. As in layer 2 VPNs, MPLS is used to tunnel packets from one site to another; however, the configuration of the tunnels is performed automatically by some fairly elaborate use of BGP, which is beyond the scope of this book.

Customer A in fact usually *can* send data to customer B in some restricted way. Most likely, both customer A and customer B have some connection to the global Internet, and thus it is probably possible for customer A to send email messages, for example, to the mail server inside customer B's network. The "privacy" offered by a VPN prevents customer A from having unrestricted access to all the machines and subnets inside customer B's network.

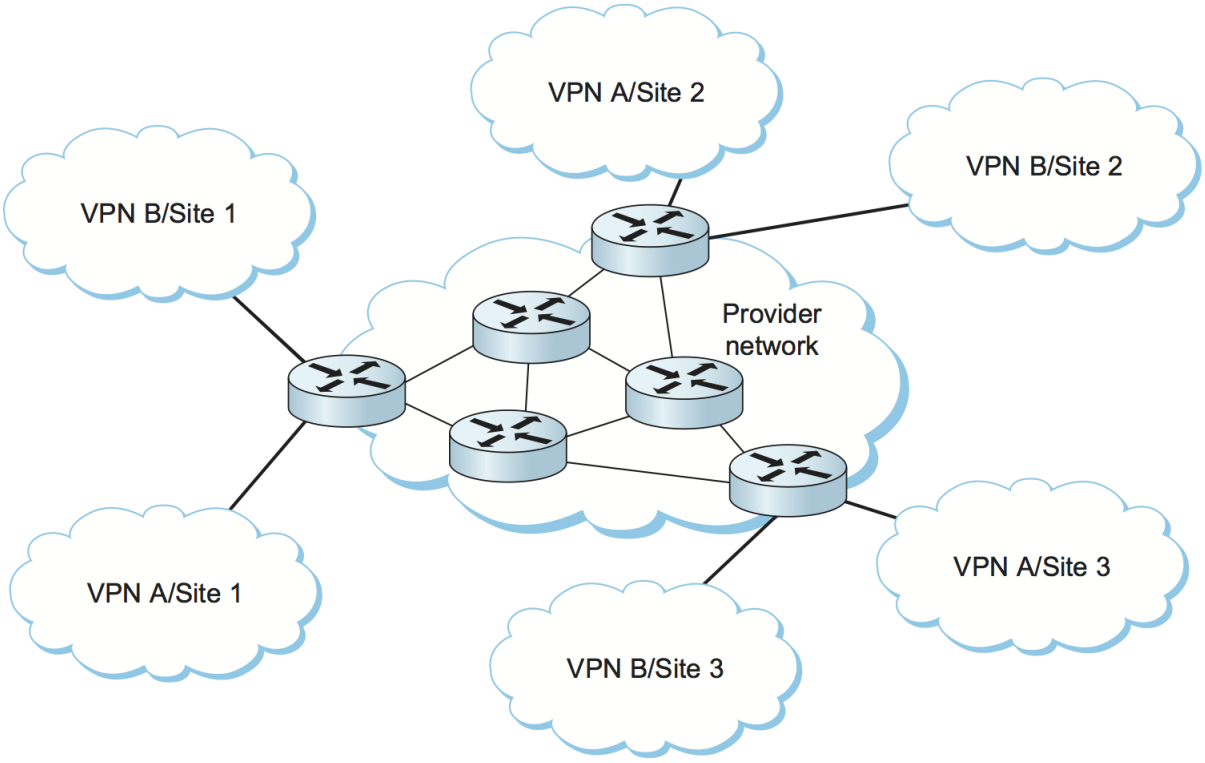


Figure 4.25.: Example of a layer 3 VPN. Customers A and B each obtain a virtually private IP service from a single provider.

In summary, MPLS is a rather versatile tool that has been applied to a wide range of different networking problems. It combines the label-swapping forwarding mechanism that is normally associated with virtual circuit networks with the routing and control protocols of IP datagram networks to produce a class of network that is somewhere between the two conventional extremes. This extends the capabilities of IP networks to enable, among other things, more precise control of routing and the support of a range of VPN services.

4.5 Routing Among Mobile Devices

It probably should not be a great surprise to learn that mobile devices present some challenges for the Internet architecture. The Internet was designed in an era when computers were large, immobile devices, and, while the Internet's designers probably had some notion that mobile devices might appear in the future, it's fair to assume it was not a top priority to accommodate them. Today, of course, mobile computers are everywhere, notably in the form of laptops and smartphones, and increasingly in other forms, such as drones. In this section, we will look at some of the challenges posed by the appearance of mobile devices and some of the current approaches to accommodating them.

4.5.1 Challenges for Mobile Networking

It is easy enough today to turn up in a wireless hotspot, connect to the Internet using 802.11 or some other wireless networking protocol, and obtain pretty good Internet service. One key enabling technology that

made the hotspot feasible is DHCP. You can settle in at a coffee shop, open your laptop, obtain an IP address for your laptop, and get your laptop talking to a default router and a Domain Name System (DNS) server, and for a broad class of applications you have everything you need.

If we look a little more closely, however, it's clear that for some application scenarios, just getting a new IP address every time you move—which is what DHCP does for you—isn't always enough. Suppose you are using your laptop or smartphone for a Voice over IP telephone call, and while talking on the phone you move from one hotspot to another, or even switch from Wi-Fi to the cellular network for your Internet connection.

Clearly, when you move from one access network to another, you need to get a new IP address—one that corresponds to the new network. But, the computer or telephone at the other end of your conversation doesn't immediately know where you have moved or what your new IP address is. Consequently, in the absence of some other mechanism, packets would continue to be sent to the address where you *used* to be, not where you are now. This problem is illustrated in Figure 4.26; as the mobile node moves from the 802.11 network in Figure 4.26(a) to the cellular network in Figure 4.26(b), somehow packets from the *correspondent node* need to find their way to the new network and then on to the mobile node.

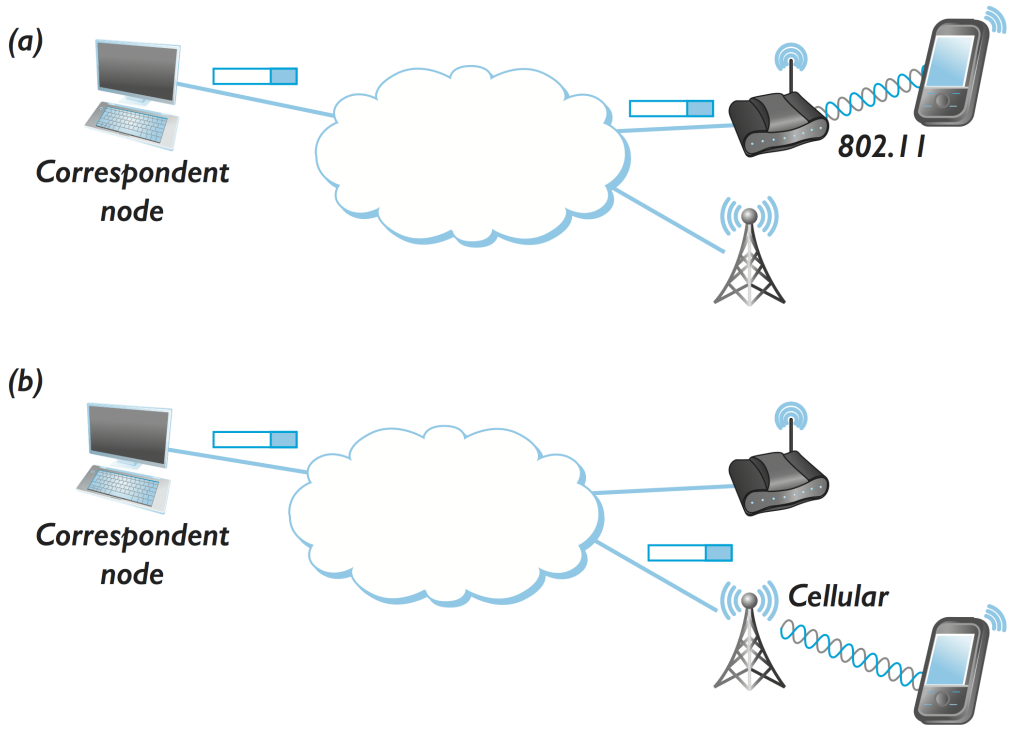


Figure 4.26.: Forwarding packets from a correspondent node to a mobile node.

There are many different ways to tackle the problem just described, and we will look at some of them below. Assuming that there is some way to redirect packets so that they come to your new address rather than your old address, the next immediately apparent problems relate to security. For example, if there is a mechanism by which I can say, "My new IP address is X," how do I prevent some attacker from making such a statement without my permission, thus enabling him to either receive my packets, or to redirect my packets to some unwitting third party? Thus, we see that security and mobility are quite closely related.

One issue that the above discussion highlights is the fact that IP addresses actually serve two tasks. They are used as an *identifier* of an endpoint, and they are also used to *locate* the endpoint. Think of the identifier as a long-lived name for the endpoint, and the locator as some possibly more temporary information about

how to route packets to the endpoint. As long as devices do not move, or do not move often, using a single address for both jobs seem pretty reasonable. But once devices start to move, you would rather like to have an identifier that does not change as you move—this is sometimes called an *Endpoint Identifier* or *Host Identifier*—and a separate *locator*. This idea of separating locators from identifiers has been around for a long time, and most of the approaches to handling mobility described below provide such a separation in some form.

The assumption that IP addresses don't change shows up in many different places. For example, transport protocols like TCP have historically made assumptions about the IP address staying constant for the life of a connection, so one approach could be to redesign transport protocols so they can operate with changing end-point addresses.

But rather than try to change TCP, a common alternative is for the application to periodically re-establish the TCP connection in case the client's IP address has changed. As strange as this sounds, if the application is HTTP-based (e.g., a web browser like Chrome or a streaming application like Netflix) then that is exactly what happens. In other words, the strategy is for the application to work around situations where the user's IP address may have changed, instead of trying to maintain the appearance that it does not change.

While we are all familiar with endpoints that move, it is worth noting that routers can also move. This is certainly less common today than endpoint mobility, but there are plenty of environments where a mobile router might make sense. One example might be an emergency response team trying to deploy a network after some natural disaster has knocked out all the fixed infrastructure. There are additional considerations when *all* the nodes in a network, not just the endpoints, are mobile, a topic we will discuss later in this section.

Before we start to look at some of the approaches to supporting mobile devices, a couple of points of clarification. It is common to find that people confuse wireless networks with mobility. After all, mobility and wireless often are found together for obvious reasons. But wireless communication is really about getting data from A to B without a wire, while mobility is about dealing with what happens when a node moves around as it communicates. Certainly many nodes that use wireless communication channels are not mobile, and sometimes mobile nodes will use wired communication (although this is less common).

Finally, in this chapter we are mostly interested in what we might call *network-layer mobility*. That is, we are interested in how to deal with nodes that move from one network to another. Moving from one access point to another in the same 802.11 network can be handled by mechanisms specific to 802.11, and cellular networks also have ways to handle mobility, of course, but in large heterogeneous systems like the Internet we need to support mobility more broadly across networks.

4.5.2 Routing to Mobile Hosts (Mobile IP)

Mobile IP is the primary mechanism in today's Internet architecture to tackle the problem of routing packets to mobile hosts. It introduces a few new capabilities but does not require any change from non-mobile hosts or most routers—thus making it incrementally deployable.

The mobile host is assumed to have a permanent IP address, called its *home address*, which has a network prefix equal to that of its *home network*. This is the address that will be used by other hosts when they initially send packets to the mobile host; because it does not change, it can be used by long-lived applications as the host roams. We can think of this as the long-lived identifier of the host.

When the host moves to a new foreign network away from its home network, it typically acquires a new

address on that network using some means such as DHCP. This address is going to change every time the host roams to a new network, so we can think of this as being more like the locator for the host, but it is important to note that the host does not lose its permanent home address when it acquires a new address on the foreign network. This home address is critical to its ability to sustain communications as it moves, as we'll see below.

> Because DHCP was developed around the same time as Mobile IP, the original Mobile IP standards did not require DHCP, but DHCP is ubiquitous today.

While the majority of routers remain unchanged, mobility support does require some new functionality in at least one router, known as the *home agent* of the mobile node. This router is located on the home network of the mobile host. In some cases, a second router with enhanced functionality, the *foreign agent,* is also required. This router is located on a network to which the mobile node attaches itself when it is away from its home network. We will consider first the operation of Mobile IP when a foreign agent is used. An example network with both home and foreign agents is shown in Figure 4.27.

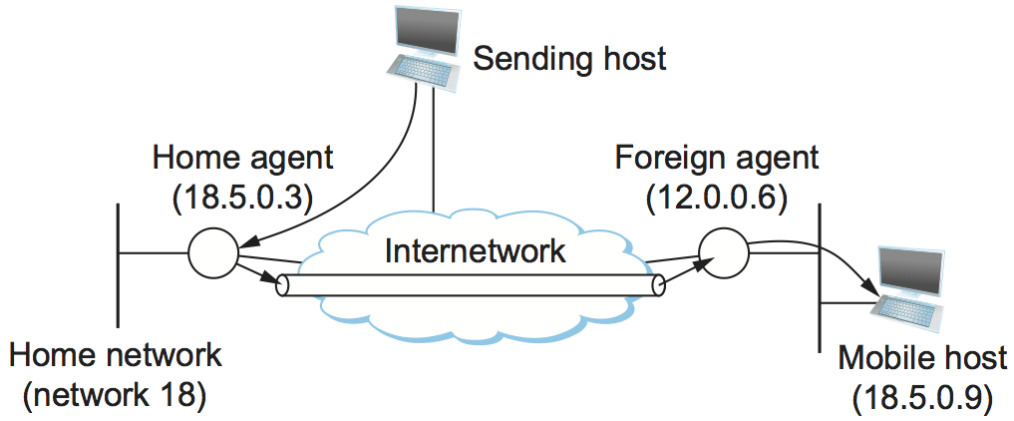


Figure 4.27.: Mobile host and mobility agents.

Both home and foreign agents periodically announce their presence on the networks to which they are attached using agent advertisement messages. A mobile host may also solicit an advertisement when it attaches to a new network. The advertisement by the home agent enables a mobile host to learn the address of its home agent before it leaves its home network. When the mobile host attaches to a foreign network, it hears an advertisement from a foreign agent and registers with the agent, providing the address of its home agent. The foreign agent then contacts the home agent, providing a *care-of address*. This is usually the IP address of the foreign agent.

At this point, we can see that any host that tries to send a packet to the mobile host will send it with a destination address equal to the home address of that node. Normal IP forwarding will cause that packet to arrive on the home network of the mobile node on which the home agent is sitting. Thus, we can divide the problem of delivering the packet to the mobile node into three parts:

1. How does the home agent intercept a packet that is destined for the mobile node?
2. How does the home agent then deliver the packet to the foreign agent?
3. How does the foreign agent deliver the packet to the mobile node?

The first problem might look easy if you just look at Figure 4.27, in which the home agent is clearly the only path between the sending host and the home network and thus must receive packets that are destined

to the mobile node. But what if the sending (correspondent) node were on network 18, or what if there were another router connected to network 18 that tried to deliver the packet without its passing through the home agent? To address this problem, the home agent actually impersonates the mobile node, using a technique called *proxy ARP*. This works just like Address Resolution Protocol (ARP), except that the home agent inserts the IP address of the mobile node, rather than its own, in the ARP messages. It uses its own hardware address, so that all the nodes on the same network learn to associate the hardware address of the home agent with the IP address of the mobile node. One subtle aspect of this process is the fact that ARP information may be cached in other nodes on the network. To make sure that these caches are invalidated in a timely way, the home agent issues an ARP message as soon as the mobile node registers with a foreign agent. Because the ARP message is not a response to a normal ARP request, it is termed a *gratuitous ARP*.

The second problem is the delivery of the intercepted packet to the foreign agent. Here we use the tunneling technique described elsewhere. The home agent simply wraps the packet inside an IP header that is destined for the foreign agent and transmits it into the internetwork. All the intervening routers just see an IP packet destined for the IP address of the foreign agent. Another way of looking at this is that an IP tunnel is established between the home agent and the foreign agent, and the home agent just drops packets destined for the mobile node into that tunnel.

When a packet finally arrives at the foreign agent, it strips the extra IP header and finds inside an IP packet destined for the home address of the mobile node. Clearly the foreign agent cannot treat this like any old IP packet because this would cause it to send it back to the home network. Instead, it has to recognize the address as that of a registered mobile node. It then delivers the packet to the *hardware* address of the mobile node (e.g., its Ethernet address), which was learned as part of the registration process.

One observation that can be made about these procedures is that it is possible for the foreign agent and the mobile node to be in the same box; that is, a mobile node can perform the foreign agent function itself. To make this work, however, the mobile node must be able to dynamically acquire an IP address that is located in the address space of the foreign network (e.g., using DHCP). This address will then be used as the care-of address. In our example, this address would have a network number of 12. This approach has the desirable feature of allowing mobile nodes to attach to networks that don't have foreign agents; thus, mobility can be achieved with only the addition of a home agent and some new software on the mobile node (assuming DHCP is used on the foreign network).

What about traffic in the other direction (i.e., from mobile node to fixed node)? This turns out to be much easier. The mobile node just puts the IP address of the fixed node in the destination field of its IP packets while putting its permanent address in the source field, and the packets are forwarded to the fixed node using normal means. Of course, if both nodes in a conversation are mobile, then the procedures described above are used in each direction.

Route Optimization in Mobile IP

There is one significant drawback to the above approach: The route from the correspondent node to the mobile node can be significantly suboptimal. One of the most extreme examples is when a mobile node and the correspondent node are on the same network, but the home network for the mobile node is on the far side of the Internet. The sending correspondent node addresses all packets to the home network; they traverse the Internet to reach the home agent, which then tunnels them back across the Internet to reach the foreign agent. Clearly, it would be nice if the correspondent node could find out that the mobile node is actually on the same network and deliver the packet directly. In the more general case, the goal is to deliver packets as directly as possible from correspondent node to mobile node without passing through a home agent. This is

sometimes referred to as the *triangle routing problem* since the path from correspondent to mobile node via home agent takes two sides of a triangle, rather than the third side that is the direct path.

The basic idea behind the solution to triangle routing is to let the correspondent node know the care-of address of the mobile node. The correspondent node can then create its own tunnel to the foreign agent. This is treated as an optimization of the process just described. If the sender has been equipped with the necessary software to learn the care-of address and create its own tunnel, then the route can be optimized; if not, packets just follow the suboptimal route.

When a home agent sees a packet destined for one of the mobile nodes that it supports, it can deduce that the sender is not using the optimal route. Therefore, it sends a "binding update" message back to the source, in addition to forwarding the data packet to the foreign agent. The source, if capable, uses this binding update to create an entry in a *binding cache,* which consists of a list of mappings from mobile node addresses to care-of addresses. The next time this source has a data packet to send to that mobile node, it will find the binding in the cache and can tunnel the packet directly to the foreign agent.

There is an obvious problem with this scheme, which is that the binding cache may become out-of-date if the mobile host moves to a new network. If an out-of-date cache entry is used, the foreign agent will receive tunneled packets for a mobile node that is no longer registered on its network. In this case, it sends a *binding warning* message back to the sender to tell it to stop using this cache entry. This scheme works only in the case where the foreign agent is not the mobile node itself, however. For this reason, cache entries need to be deleted after some period of time; the exact amount is specified in the binding update message.

As noted above, mobile routing provides some interesting security challenges, which are clearer now that we have seen how Mobile IP works. For example, an attacker wishing to intercept the packets destined to some other node in an internetwork could contact the home agent for that node and announce itself as the new foreign agent for the node. Thus, it is clear that some authentication mechanisms are required.

Mobility in IPv6

There are a handful of significant differences between mobility support in IPv4 and IPv6. Most importantly, it was possible to build mobility support into the standards for IPv6 pretty much from the beginning, thus alleviating a number of incremental deployment problems. (It may be more correct to say that IPv6 is one big incremental deployment problem, which, once solved, will deliver mobility support as part of the package.)

Since all IPv6-capable hosts can acquire an address whenever they are attached to a foreign network (using several mechanisms defined as part of the core v6 specifications), Mobile IPv6 does away with the foreign agent and includes the necessary capabilities to act as a foreign agent in every host.

One other interesting aspect of IPv6 that comes into play with Mobile IP is its inclusion of a flexible set of extension headers, as described elsewhere in this chapter. This is used in the optimized routing scenario described above. Rather than *tunneling* a packet to the mobile node at its care-of address, an IPv6 node can send an IP packet to the care-of address with the home address contained in a *routing header*. This header is ignored by all the intermediate nodes, but it enables the mobile node to treat the packet as if it were sent to the home address, thus enabling it to continue presenting higher layer protocols with the illusion that its IP address is fixed. Using an extension header rather than a tunnel is more efficient from the perspective of both bandwidth consumption and processing.

Finally, we note that many open issues remain in mobile networking. Managing the power consumption of

mobile devices is increasingly important, so that smaller devices with limited battery power can be built. There is also the problem of *ad hoc* mobile networks—enabling a group of mobile nodes to form a network in the absence of any fixed nodes—which has some special challenges. A particularly challenging class of mobile networks is *sensor networks*. Sensors typically are small, inexpensive, and often battery powered, meaning that issues of very low power consumption and limited processing capability must also be considered. Furthermore, since wireless communications and mobility typically go hand in hand, the continual advances in wireless technologies keep on producing new challenges and opportunities for mobile networking.

Perspective: The Cloud is Eating the Internet

The Cloud and the Internet are symbiotic systems. They were historically distinct, but today the line between them is increasingly fuzzy. If you start with the textbook definition, the Internet provides end-to-end connectivity between any two hosts (e.g., a client laptop and a remote server machine), and the cloud supports several warehouse-sized datacenters, each of which provides a cost-effective way to power, cool, and operate a large number of server machines. End-users connect to the nearest datacenter over the Internet in exactly the same way they connect to a server in a remote machine room.

That's an accurate description of the relationship between the Internet and the Cloud in the early days of commercial cloud providers like Amazon, Microsoft, and Google. For example, Amazon's cloud circa 2009 had two datacenters, one on the east coast of the US and one on the west coast. Today, however, each of the major cloud providers operates several dozen datacenters spread across the globe, and it should be no surprise that they are strategically located in close proximity to Internet Exchange Points (IXP), each of which provides rich connectivity to the rest of the Internet. There are over 150 IXPs worldwide, and while not every cloud provider replicates a full datacenter near each one (many of these sites are co-location facilities), it is fair to say the cloud's most frequently accessed content (e.g., the most popular Netflix movies, YouTube videos, and Facebook photos) is potentially distributed to that many locations.

There are two consequences to this wide dispersion of the cloud. One is that the end-to-end path from client to server doesn't necessarily traverse the entire Internet. A user is likely to find the content he or she wants to access has been replicated at a nearby IXP—which is usually just one AS hop away—as opposed to being on the far side of the globe. The second consequence is that the major cloud providers do not use the public Internet to interconnect their distributed datacenters. It is common for cloud providers to keep their content synchronized across distributed datacenters, but they typically do this over a private backbone. This allows them to take advantage of whatever optimizations they want without needing to fully inter-operate with anyone else.

In other words, while the figures in *Section 4.1* fairly represents the Internet's overall shape, and BGP makes it *possible* to connect any pair of hosts, in practice most users interact with applications running in the Cloud, which looks more like Figure 4.28. (One important detail that the figure does not convey is that Cloud providers do not typically build a WAN by laying their own fiber, but they instead lease fiber from servicer providers, meaning that the private cloud backbone and the service provider backbones often share the same physical infrastructure.)

Note that while it is possible to replicate *content* across the cloud's many locations, we do not yet have the technology to replicate *people*. This means that when widely dispersed users want to talk with each other—for example, as part of a video conference call—it's the multicast tree that gets distributed across the cloud. In other words, multicast isn't typically running in the routers of the service provider backbones

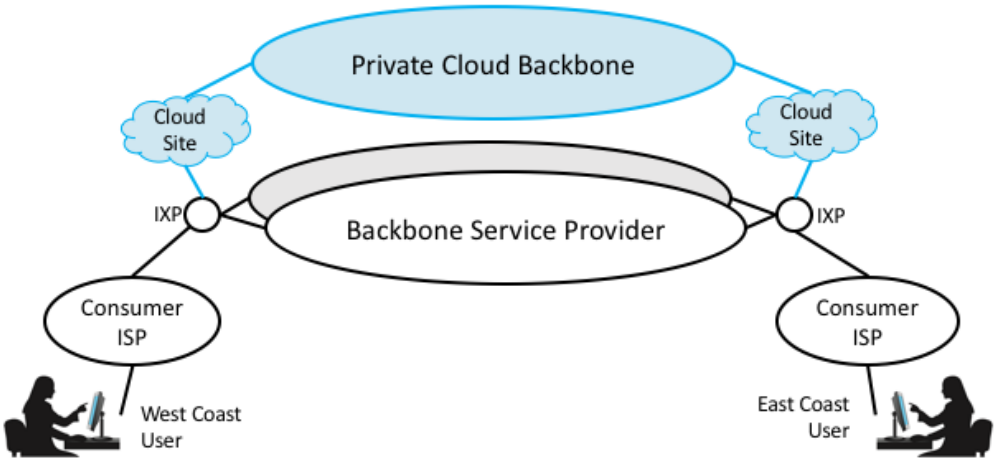


Figure 4.28.: Cloud is widely distributed throughout the Internet with private backbones.

(as *Section 4.3* suggests), but it is instead running in server processes distributed across some subset of the 150+ locations that serve as the Internet's major interconnection points. A multicast tree constructed in this way is called an overlay, which is a topic that we return to in *Section 9.4*.

Broader Perspective

To continue reading about the cloudification of the Internet, see *Perspective: HTTP is the New Narrow Waist*.

To learn more about the Cloud's distributed footprint, we recommend How the Internet Travels Across the Ocean, New York Times, March 2019.

CHAPTER

FIVE

END-TO-END PROTOCOLS

> Victory is the beautiful, bright coloured flower. Transport is the stem without which it could never have blossomed.
>
> —*Winston Churchill*

Problem: Getting Processes to Communicate

Many technologies can be used to connect together a collection of computers, ranging from simple Ethernets and wireless networks to global-scale internetworks. Once interconnected, the next problem is to turn this host-to-host packet delivery service into a process-to-process communication channel. This is the role played by the *transport* level of the network architecture, which, because it supports communication between application programs running in end nodes, is sometimes called the *end-to-end* protocol.

Two forces shape the end-to-end protocol. From above, the application-level processes that use its services have certain requirements. The following list itemizes some of the common properties that a transport protocol can be expected to provide:

- Guarantees message delivery
- Delivers messages in the same order they are sent
- Delivers at most one copy of each message
- Supports arbitrarily large messages
- Supports synchronization between the sender and the receiver
- Allows the receiver to apply flow control to the sender
- Supports multiple application processes on each host

Note that this list does not include all the functionality that application processes might want from the network. For example, it does not include security features like authentication or encryption, which are typically provided by protocols that sit above the transport level. (We discuss security-related topics in a later chapter.)

From below, the underlying network upon which the transport protocol operates has certain limitations in the level of service it can provide. Some of the more typical limitations of the network are that it may

- Drop messages

- Reorder messages
- Deliver duplicate copies of a given message
- Limit messages to some finite size
- Deliver messages after an arbitrarily long delay

Such a network is said to provide a *best-effort* level of service, as exemplified by the Internet.

The challenge, therefore, is to develop algorithms that turn the less-than-perfect properties of the underlying network into the high level of service required by application programs. Different transport protocols employ different combinations of these algorithms. This chapter looks at these algorithms in the context of four representative services—a simple asynchronous demultiplexing service, a reliable byte-stream service, a request/reply service, and a service for real-time applications.

In the case of the demultiplexing and byte-stream services, we use the Internet's User Datagram Protocol (UDP) and Transmission Control Protocol (TCP), respectively, to illustrate how these services are provided in practice. In the case of a request/reply service, we discuss the role it plays in a Remote Procedure Call (RPC) service and what features that entails. The Internet does not have a single RPC protocol, so we cap this discussion off with a description of three widely used RPC protocols: SunRPC, DCE-RPC, and gRPC.

Finally, real-time applications make particular demands on the transport protocol, such as the need to carry timing information that allows audio or video samples to be played back at the appropriate point in time. We look at the requirements placed by applications on such a protocol and the most widely used example, the Real-Time Transport Protocol (RTP).

5.1 Simple Demultiplexor (UDP)

The simplest possible transport protocol is one that extends the host-to-host delivery service of the underlying network into a process-to-process communication service. There are likely to be many processes running on any given host, so the protocol needs to add a level of demultiplexing, thereby allowing multiple application processes on each host to share the network. Aside from this requirement, the transport protocol adds no other functionality to the best-effort service provided by the underlying network. The Internet's User Datagram Protocol is an example of such a transport protocol.

The only interesting issue in such a protocol is the form of the address used to identify the target process. Although it is possible for processes to *directly* identify each other with an OS-assigned process id (pid), such an approach is only practical in a closed distributed system in which a single OS runs on all hosts and assigns each process a unique id. A more common approach, and the one used by UDP, is for processes to *indirectly* identify each other using an abstract locater, usually called a *port*. The basic idea is for a source process to send a message to a port and for the destination process to receive the message from a port.

The header for an end-to-end protocol that implements this demultiplexing function typically contains an identifier (port) for both the sender (source) and the receiver (destination) of the message. For example, the UDP header is given in Figure 5.1. Notice that the UDP port field is only 16 bits long. This means that there are up to 64K possible ports, clearly not enough to identify all the processes on all the hosts in the Internet. Fortunately, ports are not interpreted across the entire Internet, but only on a single host. That is, a process is really identified by a port on some particular host: a (port, host) pair. This pair constitutes the demultiplexing key for the UDP protocol.

The next issue is how a process learns the port for the process to which it wants to send a message. Typically, a client process initiates a message exchange with a server process. Once a client has contacted a server, the server knows the client's port (from the `SrcPrt` field contained in the message header) and can reply to it. The real problem, therefore, is how the client learns the server's port in the first place. A common approach is for the server to accept messages at a *well-known port*. That is, each server receives its messages at some fixed port that is widely published, much like the emergency telephone service available in the United States at the well-known phone number 911. In the Internet, for example, the Domain Name Server (DNS) receives messages at well-known port 53 on each host, the mail service listens for messages at port 25, and the Unix `talk` program accepts messages at well-known port 517, and so on. This mapping is published periodically in an RFC and is available on most Unix systems in file `/etc/services`. Sometimes a well-known port is just the starting point for communication: The client and server use the well-known port to agree on some other port that they will use for subsequent communication, leaving the well-known port free for other clients.

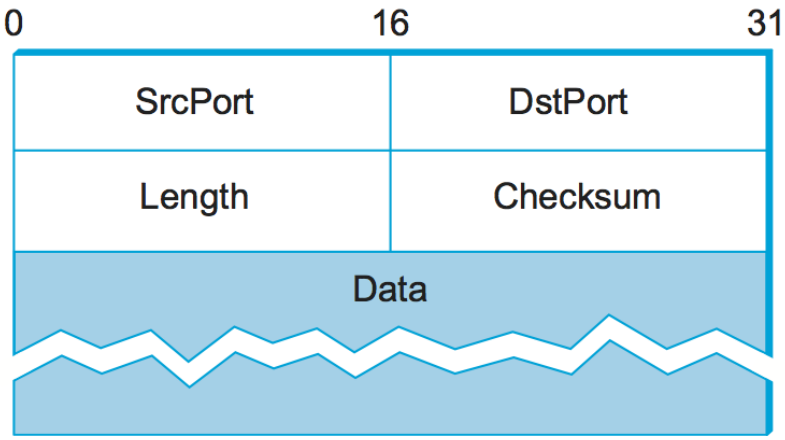


Figure 5.1.: Format for UDP header.

An alternative strategy is to generalize this idea, so that there is only a single well-known port—the one at which the *port mapper* service accepts messages. A client would send a message to the port mapper's well-known port asking for the port it should use to talk to the "whatever" service, and the port mapper returns the appropriate port. This strategy makes it easy to change the port associated with different services over time and for each host to use a different port for the same service.

As just mentioned, a port is purely an abstraction. Exactly how it is implemented differs from system to system, or more precisely, from OS to OS. For example, the socket API described in Chapter 1 is an example implementation of ports. Typically, a port is implemented by a message queue, as illustrated in Figure 5.2. When a message arrives, the protocol (e.g., UDP) appends the message to the end of the queue. Should the queue be full, the message is discarded. There is no flow-control mechanism in UDP to tell the sender to slow down. When an application process wants to receive a message, one is removed from the front of the queue. If the queue is empty, the process blocks until a message becomes available.

Finally, although UDP does not implement flow control or reliable/ordered delivery, it does provide one more function aside from demultiplexing messages to some application process—it also ensures the correctness of the message by the use of a checksum. (The UDP checksum is optional in IPv4 but is mandatory in IPv6.) The basic UDP checksum algorithm is the same one used for IP—that is, it adds up a set of 16-bit words using ones' complement arithmetic and takes the ones' complement of the result. But the input data that is used for the checksum is a little counterintuitive.

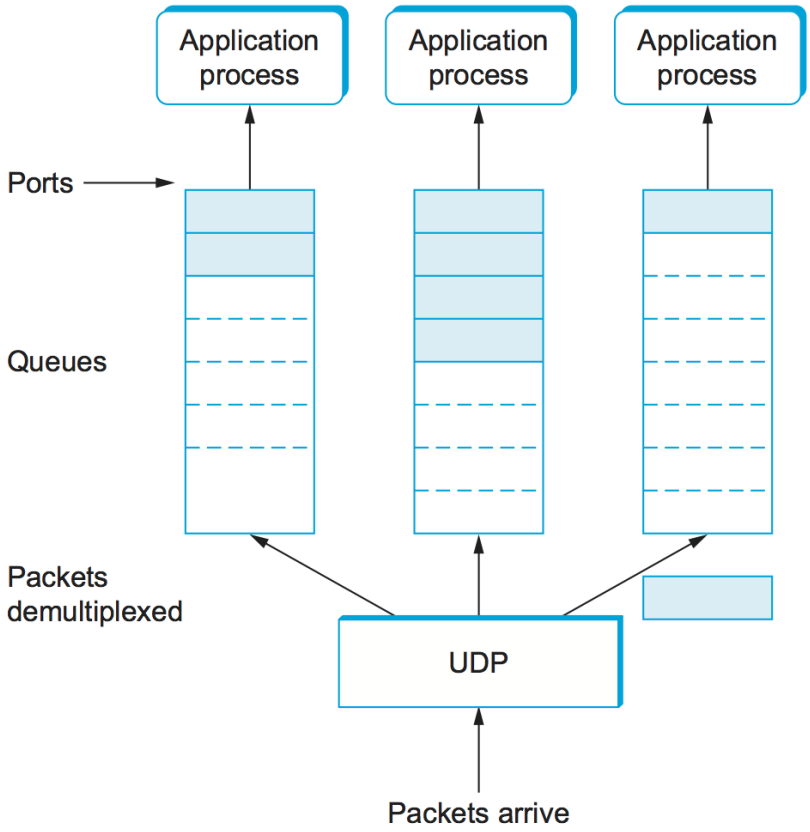


Figure 5.2.: UDP message queue.

The UDP checksum takes as input the UDP header, the contents of the message body, and something called the *pseudoheader*. The pseudoheader consists of three fields from the IP header—protocol number, source IP address, and destination IP address—plus the UDP length field. (Yes, the UDP length field is included twice in the checksum calculation.) The motivation behind having the pseudoheader is to verify that this message has been delivered between the correct two endpoints. For example, if the destination IP address was modified while the packet was in transit, causing the packet to be misdelivered, this fact would be detected by the UDP checksum.

5.2 Reliable Byte Stream (TCP)

In contrast to a simple demultiplexing protocol like UDP, a more sophisticated transport protocol is one that offers a reliable, connection-oriented, byte-stream service. Such a service has proven useful to a wide assortment of applications because it frees the application from having to worry about missing or reordered data. The Internet's Transmission Control Protocol is probably the most widely used protocol of this type; it is also the most carefully tuned. It is for these two reasons that this section studies TCP in detail, although we identify and discuss alternative design choices at the end of the section.

In terms of the properties of transport protocols given in the problem statement at the start of this chapter, TCP guarantees the reliable, in-order delivery of a stream of bytes. It is a full-duplex protocol, meaning that each TCP connection supports a pair of byte streams, one flowing in each direction. It also includes a flow-control mechanism for each of these byte streams that allows the receiver to limit how much data the sender can transmit at a given time. Finally, like UDP, TCP supports a demultiplexing mechanism that allows multiple application programs on any given host to simultaneously carry on a conversation with their peers.

In addition to the above features, TCP also implements a highly tuned congestion-control mechanism. The idea of this mechanism is to throttle how fast TCP sends data, not for the sake of keeping the sender from over-running the receiver, but so as to keep the sender from overloading the network. A description of TCP's congestion-control mechanism is postponed until the next chapter, where we discuss it in the larger context of how network resources are fairly allocated.

Since many people confuse congestion control and flow control, we restate the difference. *Flow control* involves preventing senders from over-running the capacity of receivers. *Congestion control* involves preventing too much data from being injected into the network, thereby causing switches or links to become overloaded. Thus, flow control is an end-to-end issue, while congestion control is concerned with how hosts and networks interact.

5.2.1 End-to-End Issues

At the heart of TCP is the sliding window algorithm. Even though this is the same basic algorithm as is often used at the link level, because TCP runs over the Internet rather than a physical point-to-point link, there are many important differences. This subsection identifies these differences and explains how they complicate TCP. The following subsections then describe how TCP addresses these and other complications.

First, whereas the link-level sliding window algorithm presented runs over a single physical link that always connects the same two computers, TCP supports logical connections between processes that are running on any two computers in the Internet. This means that TCP needs an explicit connection establishment phase during which the two sides of the connection agree to exchange data with each other. This difference is

analogous to having to dial up the other party, rather than having a dedicated phone line. TCP also has an explicit connection teardown phase. One of the things that happens during connection establishment is that the two parties establish some shared state to enable the sliding window algorithm to begin. Connection teardown is needed so each host knows it is OK to free this state.

Second, whereas a single physical link that always connects the same two computers has a fixed round-trip time (RTT), TCP connections are likely to have widely different round-trip times. For example, a TCP connection between a host in San Francisco and a host in Boston, which are separated by several thousand kilometers, might have an RTT of 100 ms, while a TCP connection between two hosts in the same room, only a few meters apart, might have an RTT of only 1 ms. The same TCP protocol must be able to support both of these connections. To make matters worse, the TCP connection between hosts in San Francisco and Boston might have an RTT of 100 ms at 3 a.m., but an RTT of 500 ms at 3 p.m. Variations in the RTT are even possible during a single TCP connection that lasts only a few minutes. What this means to the sliding window algorithm is that the timeout mechanism that triggers retransmissions must be adaptive. (Certainly, the timeout for a point-to-point link must be a settable parameter, but it is not necessary to adapt this timer for a particular pair of nodes.)

A third difference is that packets may be reordered as they cross the Internet, but this is not possible on a point-to-point link where the first packet put into one end of the link must be the first to appear at the other end. Packets that are slightly out of order do not cause a problem since the sliding window algorithm can reorder packets correctly using the sequence number. The real issue is how far out of order packets can get or, said another way, how late a packet can arrive at the destination. In the worst case, a packet can be delayed in the Internet until the IP time to live (`TTL`) field expires, at which time the packet is discarded (and hence there is no danger of it arriving late). Knowing that IP throws packets away after their `TTL` expires, TCP assumes that each packet has a maximum lifetime. The exact lifetime, known as the *maximum segment lifetime* (MSL), is an engineering choice. The current recommended setting is 120 seconds. Keep in mind that IP does not directly enforce this 120-second value; it is simply a conservative estimate that TCP makes of how long a packet might live in the Internet. The implication is significant—TCP has to be prepared for very old packets to suddenly show up at the receiver, potentially confusing the sliding window algorithm.

Fourth, the computers connected to a point-to-point link are generally engineered to support the link. For example, if a link's delay × bandwidth product is computed to be 8 KB—meaning that a window size is selected to allow up to 8 KB of data to be unacknowledged at a given time—then it is likely that the computers at either end of the link have the ability to buffer up to 8 KB of data. Designing the system otherwise would be silly. On the other hand, almost any kind of computer can be connected to the Internet, making the amount of resources dedicated to any one TCP connection highly variable, especially considering that any one host can potentially support hundreds of TCP connections at the same time. This means that TCP must include a mechanism that each side uses to "learn" what resources (e.g., how much buffer space) the other side is able to apply to the connection. This is the flow control issue.

Fifth, because the transmitting side of a directly connected link cannot send any faster than the bandwidth of the link allows, and only one host is pumping data into the link, it is not possible to unknowingly congest the link. Said another way, the load on the link is visible in the form of a queue of packets at the sender. In contrast, the sending side of a TCP connection has no idea what links will be traversed to reach the destination. For example, the sending machine might be directly connected to a relatively fast Ethernet—and capable of sending data at a rate of 10 Gbps—but somewhere out in the middle of the network, a 1.5-Mbps link must be traversed. And, to make matters worse, data being generated by many different sources might be trying to traverse this same slow link. This leads to the problem of network congestion. Discussion of this topic is delayed until the next chapter.

We conclude this discussion of end-to-end issues by comparing TCP's approach to providing a reliable/ordered delivery service with the approach used by virtual-circuit-based networks like the historically important X.25 network. In TCP, the underlying IP network is assumed to be unreliable and to deliver messages out of order; TCP uses the sliding window algorithm on an end-to-end basis to provide reliable/ordered delivery. In contrast, X.25 networks use the sliding window protocol within the network, on a hop-by-hop basis. The assumption behind this approach is that if messages are delivered reliably and in order between each pair of nodes along the path between the source host and the destination host, then the end-to-end service also guarantees reliable/ordered delivery.

The problem with this latter approach is that a sequence of hop-by-hop guarantees does not necessarily add up to an end-to-end guarantee. First, if a heterogeneous link (say, an Ethernet) is added to one end of the path, then there is no guarantee that this hop will preserve the same service as the other hops. Second, just because the sliding window protocol guarantees that messages are delivered correctly from node A to node B, and then from node B to node C, it does not guarantee that node B behaves perfectly. For example, network nodes have been known to introduce errors into messages while transferring them from an input buffer to an output buffer. They have also been known to accidentally reorder messages. As a consequence of these small windows of vulnerability, it is still necessary to provide true end-to-end checks to guarantee reliable/ordered service, even though the lower levels of the system also implement that functionality.

Key Takeaway

This discussion serves to illustrate one of the most important principles in system design—the *end-to-end argument*. In a nutshell, the end-to-end argument says that a function (in our example, providing reliable/ordered delivery) should not be provided in the lower levels of the system unless it can be completely and correctly implemented at that level. Therefore, this rule argues in favor of the TCP/IP approach. This rule is not absolute, however. It does allow for functions to be incompletely provided at a low level as a performance optimization. This is why it is perfectly consistent with the end-to-end argument to perform error detection (e.g., CRC) on a hop-by-hop basis; detecting and retransmitting a single corrupt packet across one hop is preferable to having to retransmit an entire file end-to-end. *[Next]*

5.2.2 Segment Format

TCP is a byte-oriented protocol, which means that the sender writes bytes into a TCP connection and the receiver reads bytes out of the TCP connection. Although "byte stream" describes the service TCP offers to application processes, TCP does not, itself, transmit individual bytes over the Internet. Instead, TCP on the source host buffers enough bytes from the sending process to fill a reasonably sized packet and then sends this packet to its peer on the destination host. TCP on the destination host then empties the contents of the packet into a receive buffer, and the receiving process reads from this buffer at its leisure. This situation is illustrated in Figure 5.3, which, for simplicity, shows data flowing in only one direction. Remember that, in general, a single TCP connection supports byte streams flowing in both directions.

The packets exchanged between TCP peers in Figure 5.3 are called *segments*, since each one carries a segment of the byte stream. Each TCP segment contains the header schematically depicted in Figure 5.4. The relevance of most of these fields will become apparent throughout this section. For now, we simply introduce them.

The `SrcPort` and `DstPort` fields identify the source and destination ports, respectively, just as in UDP.

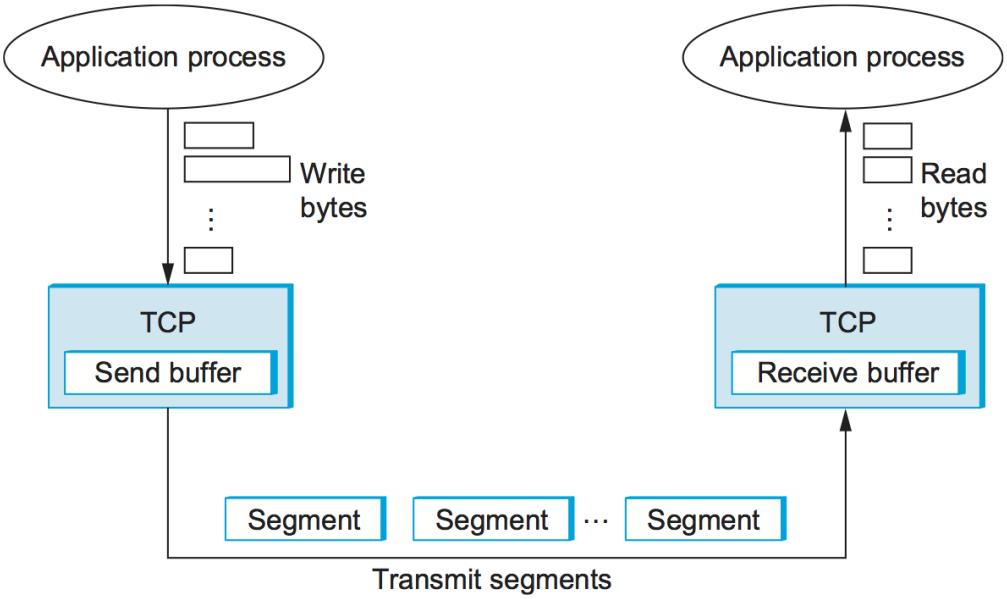


Figure 5.3.: How TCP manages a byte stream.

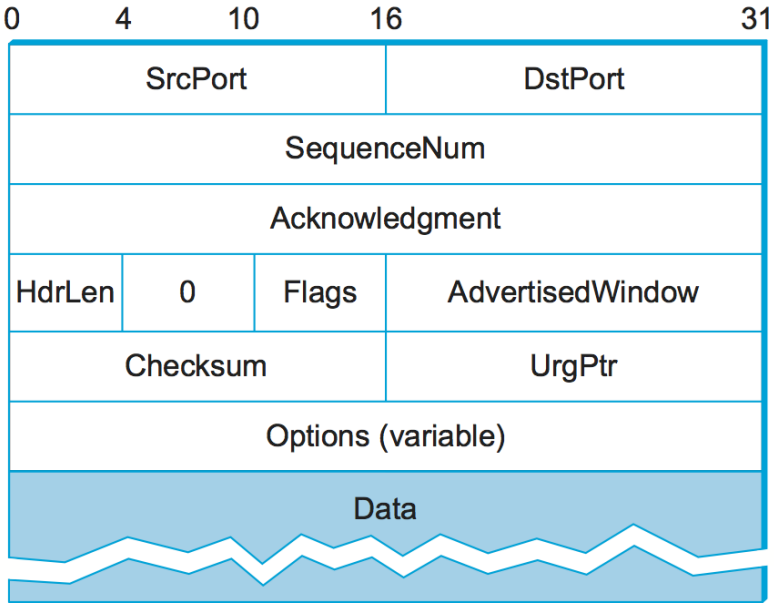


Figure 5.4.: TCP header format.

These two fields, plus the source and destination IP addresses, combine to uniquely identify each TCP connection. That is, TCP's demux key is given by the 4-tuple

```
(SrcPort, SrcIPAddr, DstPort, DstIPAddr)
```

Note that because TCP connections come and go, it is possible for a connection between a particular pair of ports to be established, used to send and receive data, and closed, and then at a later time for the same pair of ports to be involved in a second connection. We sometimes refer to this situation as two different *incarnations* of the same connection.

The `Acknowledgement`, `SequenceNum`, and `AdvertisedWindow` fields are all involved in TCP's sliding window algorithm. Because TCP is a byte-oriented protocol, each byte of data has a sequence number. The `SequenceNum` field contains the sequence number for the first byte of data carried in that segment, and the `Acknowledgement` and `AdvertisedWindow` fields carry information about the flow of data going in the other direction. To simplify our discussion, we ignore the fact that data can flow in both directions, and we concentrate on data that has a particular `SequenceNum` flowing in one direction and `Acknowledgement` and `AdvertisedWindow` values flowing in the opposite direction, as illustrated in Figure 5.5. The use of these three fields is described more fully later in this chapter.

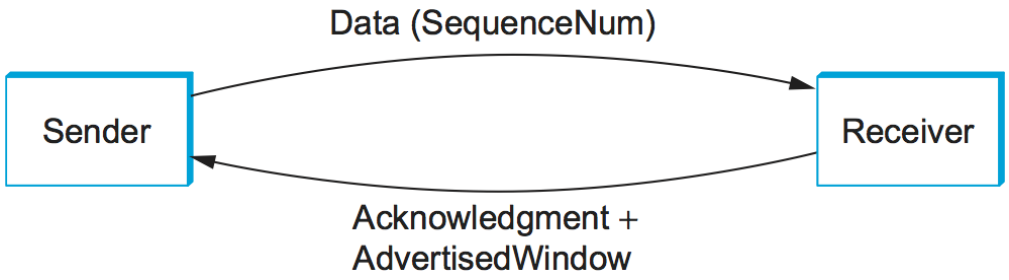


Figure 5.5.: Simplified illustration (showing only one direction) of the TCP process, with data flow in one direction and ACKs in the other.

The 6-bit `Flags` field is used to relay control information between TCP peers. The possible flags include `SYN`, `FIN`, `RESET`, `PUSH`, `URG`, and `ACK`. The `SYN` and `FIN` flags are used when establishing and terminating a TCP connection, respectively. Their use is described in a later section. The `ACK` flag is set any time the `Acknowledgement` field is valid, implying that the receiver should pay attention to it. The `URG` flag signifies that this segment contains urgent data. When this flag is set, the `UrgPtr` field indicates where the nonurgent data contained in this segment begins. The urgent data is contained at the front of the segment body, up to and including a value of `UrgPtr` bytes into the segment. The `PUSH` flag signifies that the sender invoked the push operation, which indicates to the receiving side of TCP that it should notify the receiving process of this fact. We discuss these last two features more in a later section. Finally, the `RESET` flag signifies that the receiver has become confused—for example, because it received a segment it did not expect to receive—and so wants to abort the connection.

Finally, the `Checksum` field is used in exactly the same way as for UDP—it is computed over the TCP header, the TCP data, and the pseudoheader, which is made up of the source address, destination address, and length fields from the IP header. The checksum is required for TCP in both IPv4 and IPv6. Also, since the TCP header is of variable length (options can be attached after the mandatory fields), a `HdrLen` field is included that gives the length of the header in 32-bit words. This field is also known as the `Offset` field, since it measures the offset from the start of the packet to the start of the data.

5.2.3 Connection Establishment and Termination

A TCP connection begins with a client (caller) doing an active open to a server (callee). Assuming that the server had earlier done a passive open, the two sides engage in an exchange of messages to establish the connection. (Recall from Chapter 1 that a party wanting to initiate a connection performs an active open, while a party willing to accept a connection does a passive open.[1]) Only after this connection establishment phase is over do the two sides begin sending data. Likewise, as soon as a participant is done sending data, it closes one direction of the connection, which causes TCP to initiate a round of connection termination messages. Notice that, while connection setup is an asymmetric activity (one side does a passive open and the other side does an active open), connection teardown is symmetric (each side has to close the connection independently). Therefore, it is possible for one side to have done a close, meaning that it can no longer send data, but for the other side to keep the other half of the bidirectional connection open and to continue sending data.

Three-Way Handshake

The algorithm used by TCP to establish and terminate a connection is called a *three-way handshake*. We first describe the basic algorithm and then show how it is used by TCP. The three-way handshake involves the exchange of three messages between the client and the server, as illustrated by the timeline given in Figure 5.6.

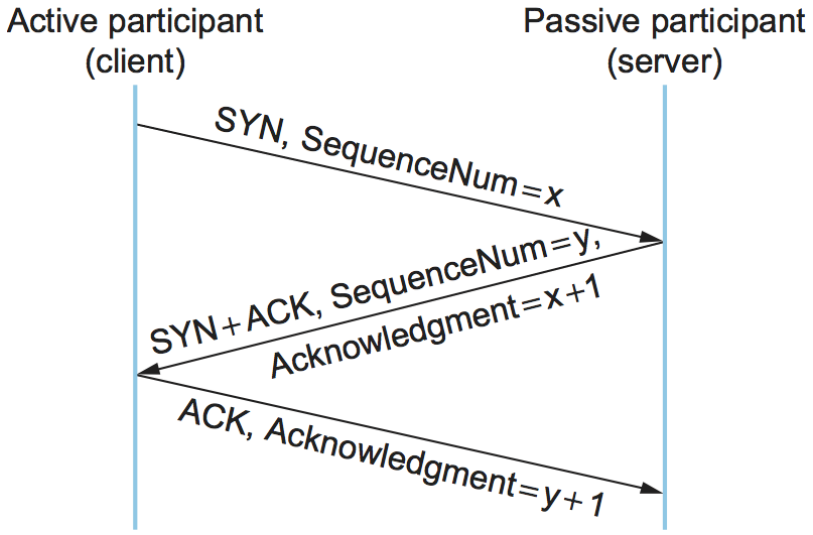


Figure 5.6.: Timeline for three-way handshake algorithm.

The idea is that two parties want to agree on a set of parameters, which, in the case of opening a TCP connection, are the starting sequence numbers the two sides plan to use for their respective byte streams. In general, the parameters might be any facts that each side wants the other to know about. First, the client (the active participant) sends a segment to the server (the passive participant) stating the initial sequence number it plans to use (`Flags` = `SYN`, `SequenceNum` = x). The server then responds with a single segment that both acknowledges the client's sequence number (`Flags = ACK, Ack = x + 1`) and states its own

[1] To be more precise, TCP allows connection setup to be symmetric, with both sides trying to open the connection at the same time, but the common case is for one side to do an active open and the other side to do a passive open.

beginning sequence number (`Flags = SYN, SequenceNum = y`). That is, both the `SYN` and `ACK` bits are set in the `Flags` field of this second message. Finally, the client responds with a third segment that acknowledges the server's sequence number (`Flags = ACK, Ack = y + 1`). The reason why each side acknowledges a sequence number that is one larger than the one sent is that the `Acknowledgement` field actually identifies the "next sequence number expected," thereby implicitly acknowledging all earlier sequence numbers. Although not shown in this timeline, a timer is scheduled for each of the first two segments, and if the expected response is not received the segment is retransmitted.

You may be asking yourself why the client and server have to exchange starting sequence numbers with each other at connection setup time. It would be simpler if each side simply started at some "well-known" sequence number, such as 0. In fact, the TCP specification requires that each side of a connection select an initial starting sequence number at random. The reason for this is to protect against two incarnations of the same connection reusing the same sequence numbers too soon—that is, while there is still a chance that a segment from an earlier incarnation of a connection might interfere with a later incarnation of the connection.

State-Transition Diagram

TCP is complex enough that its specification includes a state-transition diagram. A copy of this diagram is given in Figure 5.7. This diagram shows only the states involved in opening a connection (everything above ESTABLISHED) and in closing a connection (everything below ESTABLISHED). Everything that goes on while a connection is open—that is, the operation of the sliding window algorithm—is hidden in the ESTABLISHED state.

TCP's state-transition diagram is fairly easy to understand. Each box denotes a state that one end of a TCP connection can find itself in. All connections start in the CLOSED state. As the connection progresses, the connection moves from state to state according to the arcs. Each arc is labeled with a tag of the form *event/action*. Thus, if a connection is in the LISTEN state and a SYN segment arrives (i.e., a segment with the `SYN` flag set), the connection makes a transition to the SYN_RCVD state and takes the action of replying with an `ACK+SYN` segment.

Notice that two kinds of events trigger a state transition: (1) a segment arrives from the peer (e.g., the event on the arc from LISTEN to SYN_RCVD), or (2) the local application process invokes an operation on TCP (e.g., the *active open* event on the arc from CLOSED to SYN_SENT). In other words, TCP's state-transition diagram effectively defines the *semantics* of both its peer-to-peer interface and its service interface. The *syntax* of these two interfaces is given by the segment format (as illustrated in Figure 5.4) and by some application programming interface, such as the socket API, respectively.

Now let's trace the typical transitions taken through the diagram in Figure 5.7. Keep in mind that at each end of the connection, TCP makes different transitions from state to state. When opening a connection, the server first invokes a passive open operation on TCP, which causes TCP to move to the LISTEN state. At some later time, the client does an active open, which causes its end of the connection to send a SYN segment to the server and to move to the SYN_SENT state. When the SYN segment arrives at the server, it moves to the SYN_RCVD state and responds with a SYN+ACK segment. The arrival of this segment causes the client to move to the ESTABLISHED state and to send an ACK back to the server. When this ACK arrives, the server finally moves to the ESTABLISHED state. In other words, we have just traced the three-way handshake.

There are three things to notice about the connection establishment half of the state-transition diagram. First,

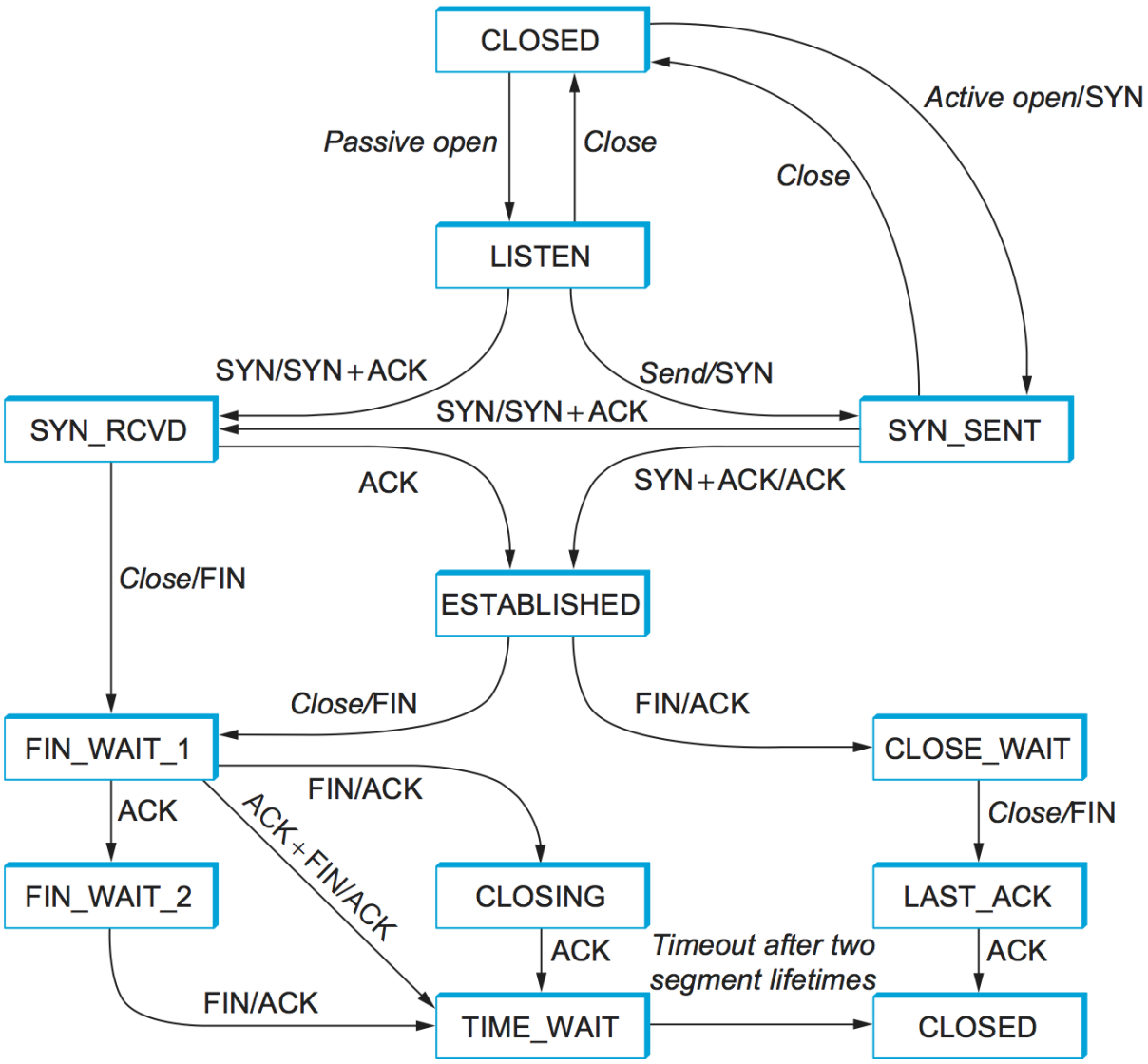


Figure 5.7.: TCP state-transition diagram.

if the client's ACK to the server is lost, corresponding to the third leg of the three-way handshake, then the connection still functions correctly. This is because the client side is already in the ESTABLISHED state, so the local application process can start sending data to the other end. Each of these data segments will have the `ACK` flag set, and the correct value in the `Acknowledgement` field, so the server will move to the ESTABLISHED state when the first data segment arrives. This is actually an important point about TCP—every segment reports what sequence number the sender is expecting to see next, even if this repeats the same sequence number contained in one or more previous segments.

The second thing to notice about the state-transition diagram is that there is a funny transition out of the LISTEN state whenever the local process invokes a *send* operation on TCP. That is, it is possible for a passive participant to identify both ends of the connection (i.e., itself and the remote participant that it is willing to have connect to it), and then for it to change its mind about waiting for the other side and instead actively establish the connection. To the best of our knowledge, this is a feature of TCP that no application process actually takes advantage of.

The final thing to notice about the diagram is the arcs that are not shown. Specifically, most of the states that involve sending a segment to the other side also schedule a timeout that eventually causes the segment to be present if the expected response does not happen. These retransmissions are not depicted in the state-transition diagram. If after several tries the expected response does not arrive, TCP gives up and returns to the CLOSED state.

Turning our attention now to the process of terminating a connection, the important thing to keep in mind is that the application process on both sides of the connection must independently close its half of the connection. If only one side closes the connection, then this means it has no more data to send, but it is still available to receive data from the other side. This complicates the state-transition diagram because it must account for the possibility that the two sides invoke the *close* operator at the same time, as well as the possibility that first one side invokes close and then, at some later time, the other side invokes close. Thus, on any one side there are three combinations of transitions that get a connection from the ESTABLISHED state to the CLOSED state:

- This side closes first: ESTABLISHED → FIN_WAIT_1 → FIN_WAIT_2 → TIME_WAIT → CLOSED.
- The other side closes first: ESTABLISHED → CLOSE_WAIT → LAST_ACK → CLOSED.
- Both sides close at the same time: ESTABLISHED → FIN_WAIT_1 → CLOSING → TIME_WAIT → CLOSED.

There is actually a fourth, although rare, sequence of transitions that leads to the CLOSED state; it follows the arc from FIN_WAIT_1 to TIME_WAIT. We leave it as an exercise for you to figure out what combination of circumstances leads to this fourth possibility.

The main thing to recognize about connection teardown is that a connection in the TIME_WAIT state cannot move to the CLOSED state until it has waited for two times the maximum amount of time an IP datagram might live in the Internet (i.e., 120 seconds). The reason for this is that, while the local side of the connection has sent an ACK in response to the other side's FIN segment, it does not know that the ACK was successfully delivered. As a consequence, the other side might retransmit its FIN segment, and this second FIN segment might be delayed in the network. If the connection were allowed to move directly to the CLOSED state, then another pair of application processes might come along and open the same connection (i.e., use the same pair of port numbers), and the delayed FIN segment from the earlier incarnation of the connection would immediately initiate the termination of the later incarnation of that connection.

5.2.4 Sliding Window Revisited

We are now ready to discuss TCP's variant of the sliding window algorithm, which serves several purposes: (1) it guarantees the reliable delivery of data, (2) it ensures that data is delivered in order, and (3) it enforces flow control between the sender and the receiver. TCP's use of the sliding window algorithm is the same as at the link level in the case of the first two of these three functions. Where TCP differs from the link-level algorithm is that it folds the flow-control function in as well. In particular, rather than having a fixed-size sliding window, the receiver *advertises* a window size to the sender. This is done using the `AdvertisedWindow` field in the TCP header. The sender is then limited to having no more than a value of `AdvertisedWindow` bytes of unacknowledged data at any given time. The receiver selects a suitable value for `AdvertisedWindow` based on the amount of memory allocated to the connection for the purpose of buffering data. The idea is to keep the sender from over-running the receiver's buffer. We discuss this at greater length below.

Reliable and Ordered Delivery

To see how the sending and receiving sides of TCP interact with each other to implement reliable and ordered delivery, consider the situation illustrated in Figure 5.8. TCP on the sending side maintains a send buffer. This buffer is used to store data that has been sent but not yet acknowledged, as well as data that has been written by the sending application but not transmitted. On the receiving side, TCP maintains a receive buffer. This buffer holds data that arrives out of order, as well as data that is in the correct order (i.e., there are no missing bytes earlier in the stream) but that the application process has not yet had the chance to read.

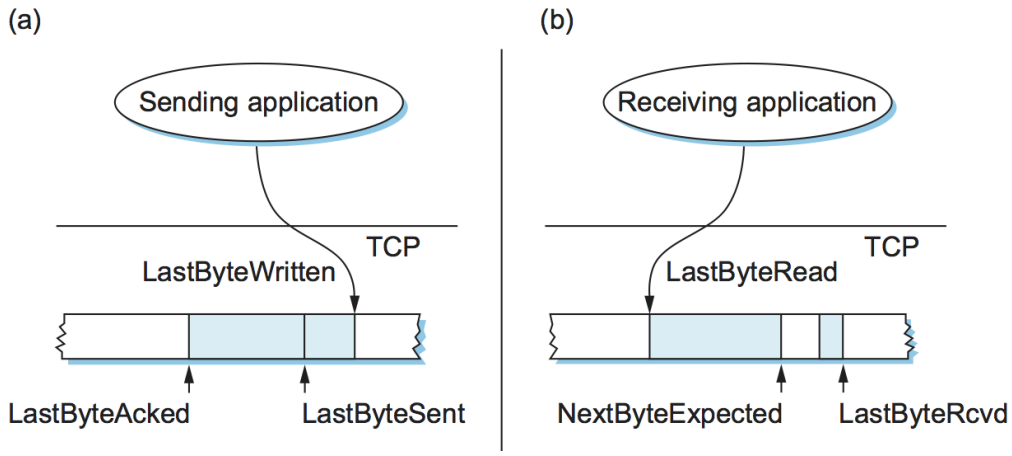


Figure 5.8.: Relationship between TCP send buffer (a) and receive buffer (b).

To make the following discussion simpler to follow, we initially ignore the fact that both the buffers and the sequence numbers are of some finite size and hence will eventually wrap around. Also, we do not distinguish between a pointer into a buffer where a particular byte of data is stored and the sequence number for that byte.

Looking first at the sending side, three pointers are maintained into the send buffer, each with an obvious meaning: `LastByteAcked`, `LastByteSent`, and `LastByteWritten`. Clearly,

```
LastByteAcked <= LastByteSent
```

since the receiver cannot have acknowledged a byte that has not yet been sent, and

```
LastByteSent <= LastByteWritten
```

since TCP cannot send a byte that the application process has not yet written. Also note that none of the bytes to the left of `LastByteAcked` need to be saved in the buffer because they have already been acknowledged, and none of the bytes to the right of `LastByteWritten` need to be buffered because they have not yet been generated.

A similar set of pointers (sequence numbers) are maintained on the receiving side: `LastByteRead`, `NextByteExpected`, and `LastByteRcvd`. The inequalities are a little less intuitive, however, because of the problem of out-of-order delivery. The first relationship

```
LastByteRead < NextByteExpected
```

is true because a byte cannot be read by the application until it is received *and* all preceding bytes have also been received. `NextByteExpected` points to the byte immediately after the latest byte to meet this criterion. Second,

```
NextByteExpected <= LastByteRcvd + 1
```

since, if data has arrived in order, `NextByteExpected` points to the byte after `LastByteRcvd`, whereas if data has arrived out of order, then `NextByteExpected` points to the start of the first gap in the data, as in Figure 5.8. Note that bytes to the left of `LastByteRead` need not be buffered because they have already been read by the local application process, and bytes to the right of `LastByteRcvd` need not be buffered because they have not yet arrived.

Flow Control

Most of the above discussion is similar to that found in the standard sliding window algorithm; the only real difference is that this time we elaborated on the fact that the sending and receiving application processes are filling and emptying their local buffer, respectively. (The earlier discussion glossed over the fact that data arriving from an upstream node was filling the send buffer and data being transmitted to a downstream node was emptying the receive buffer.)

You should make sure you understand this much before proceeding because now comes the point where the two algorithms differ more significantly. In what follows, we reintroduce the fact that both buffers are of some finite size, denoted `MaxSendBuffer` and `MaxRcvBuffer`, although we don't worry about the details of how they are implemented. In other words, we are only interested in the number of bytes being buffered, not in where those bytes are actually stored.

Recall that in a sliding window protocol, the size of the window sets the amount of data that can be sent without waiting for acknowledgment from the receiver. Thus, the receiver throttles the sender by advertising a window that is no larger than the amount of data that it can buffer. Observe that TCP on the receive side must keep

```
LastByteRcvd - LastByteRead <= MaxRcvBuffer
```

to avoid overflowing its buffer. It therefore advertises a window size of

```
AdvertisedWindow = MaxRcvBuffer - ((NextByteExpected - 1) - LastByteRead)
```

which represents the amount of free space remaining in its buffer. As data arrives, the receiver acknowledges it as long as all the preceding bytes have also arrived. In addition, `LastByteRcvd` moves to the right (is incremented), meaning that the advertised window potentially shrinks. Whether or not it shrinks depends on how fast the local application process is consuming data. If the local process is reading data just as fast as it arrives (causing `LastByteRead` to be incremented at the same rate as `LastByteRcvd`), then the advertised window stays open (i.e., `AdvertisedWindow = MaxRcvBuffer`). If, however, the receiving process falls behind, perhaps because it performs a very expensive operation on each byte of data that it reads, then the advertised window grows smaller with every segment that arrives, until it eventually goes to 0.

TCP on the send side must then adhere to the advertised window it gets from the receiver. This means that at any given time, it must ensure that

```
LastByteSent - LastByteAcked <= AdvertisedWindow
```

Said another way, the sender computes an *effective* window that limits how much data it can send:

```
EffectiveWindow = AdvertisedWindow - (LastByteSent - LastByteAcked)
```

Clearly, `EffectiveWindow` must be greater than 0 before the source can send more data. It is possible, therefore, that a segment arrives acknowledging x bytes, thereby allowing the sender to increment `LastByteAcked` by x, but because the receiving process was not reading any data, the advertised window is now x bytes smaller than the time before. In such a situation, the sender would be able to free buffer space, but not to send any more data.

All the while this is going on, the send side must also make sure that the local application process does not overflow the send buffer—that is,

```
LastByteWritten - LastByteAcked <= MaxSendBuffer
```

If the sending process tries to write y bytes to TCP, but

```
(LastByteWritten - LastByteAcked) + y > MaxSendBuffer
```

then TCP blocks the sending process and does not allow it to generate more data.

It is now possible to understand how a slow receiving process ultimately stops a fast sending process. First, the receive buffer fills up, which means the advertised window shrinks to 0. An advertised window of 0 means that the sending side cannot transmit any data, even though data it has previously sent has been successfully acknowledged. Finally, not being able to transmit any data means that the send buffer fills up, which ultimately causes TCP to block the sending process. As soon as the receiving process starts to read data again, the receive-side TCP is able to open its window back up, which allows the send-side TCP to transmit data out of its buffer. When this data is eventually acknowledged, `LastByteAcked` is incremented, the buffer space holding this acknowledged data becomes free, and the sending process is unblocked and allowed to proceed.

There is only one remaining detail that must be resolved—how does the sending side know that the advertised window is no longer 0? As mentioned above, TCP *always* sends a segment in response to a received data segment, and this response contains the latest values for the `Acknowledge` and

`AdvertisedWindow` fields, even if these values have not changed since the last time they were sent. The problem is this. Once the receive side has advertised a window size of 0, the sender is not permitted to send any more data, which means it has no way to discover that the advertised window is no longer 0 at some time in the future. TCP on the receive side does not spontaneously send nondata segments; it only sends them in response to an arriving data segment.

TCP deals with this situation as follows. Whenever the other side advertises a window size of 0, the sending side persists in sending a segment with 1 byte of data every so often. It knows that this data will probably not be accepted, but it tries anyway, because each of these 1-byte segments triggers a response that contains the current advertised window. Eventually, one of these 1-byte probes triggers a response that reports a nonzero advertised window.

Note that these 1-byte messages are called *Zero Window Probes* and in practice they are sent every 5 to 60 seconds. As for what single byte of data to send in the probe: it's the next byte of actual data just outside the window. (It has to be real data in case it's accepted by the receiver.)

Key Takeaway

Note that the reason the sending side periodically sends this probe segment is that TCP is designed to make the receive side as simple as possible—it simply responds to segments from the sender, and it never initiates any activity on its own. This is an example of a well-recognized (although not universally applied) protocol design rule, which, for lack of a better name, we call the *smart sender/ dumb receiver* rule. Recall that we saw another example of this rule when we discussed the use of NAKs in sliding window algorithm. *[Next]*

Protecting Against Wraparound

This subsection and the next consider the size of the `SequenceNum` and `AdvertisedWindow` fields and the implications of their sizes on TCP's correctness and performance. TCP's `SequenceNum` field is 32 bits long, and its `AdvertisedWindow` field is 16 bits long, meaning that TCP has easily satisfied the requirement of the sliding window algorithm that the sequence number space be twice as big as the window size: $2^{32} >> 2 \times 2^{16}$. However, this requirement is not the interesting thing about these two fields. Consider each field in turn.

The relevance of the 32-bit sequence number space is that the sequence number used on a given connection might wrap around—a byte with sequence number S could be sent at one time, and then at a later time a second byte with the same sequence number S might be sent. Once again, we assume that packets cannot survive in the Internet for longer than the recommended MSL. Thus, we currently need to make sure that the sequence number does not wrap around within a 120-second period of time. Whether or not this happens depends on how fast data can be transmitted over the Internet—that is, how fast the 32-bit sequence number space can be consumed. (This discussion assumes that we are trying to consume the sequence number space as fast as possible, but of course we will be if we are doing our job of keeping the pipe full.) Table 5.1 shows how long it takes for the sequence number to wrap around on networks with various bandwidths.

Table 5.1.: Time Until 32-Bit Sequence Number Space Wraps Around.

Bandwidth	Time until Wraparound
T1 (1.5 Mbps)	6.4 hours
T3 (45 Mbps)	13 minutes
Fast Ethernet (100 Mbps)	6 minutes
OC-3 (155 Mbps)	4 minutes
OC-48 (2.5 Gbps)	14 seconds
OC-192 (10 Gbps)	3 seconds
10GigE (10 Gbps)	3 seconds

As you can see, the 32-bit sequence number space is adequate at modest bandwidths, but given that OC-192 links are now common in the Internet backbone, and that most servers now come with 10Gig Ethernet (or 10 Gbps) interfaces, we're now well-past the point where 32 bits is too small. Fortunately, the IETF has worked out an extension to TCP that effectively extends the sequence number space to protect against the sequence number wrapping around. This and related extensions are described in a later section.

Keeping the Pipe Full

The relevance of the 16-bit `AdvertisedWindow` field is that it must be big enough to allow the sender to keep the pipe full. Clearly, the receiver is free to not open the window as large as the `AdvertisedWindow` field allows; we are interested in the situation in which the receiver has enough buffer space to handle as much data as the largest possible `AdvertisedWindow` allows.

In this case, it is not just the network bandwidth but the delay x bandwidth product that dictates how big the `AdvertisedWindow` field needs to be—the window needs to be opened far enough to allow a full delay × bandwidth product's worth of data to be transmitted. Assuming an RTT of 100 ms (a typical number for a cross-country connection in the United States), Table 5.2 gives the delay × bandwidth product for several network technologies.

Table 5.2.: Required Window Size for 100-ms RTT

Bandwidth	Delay × Bandwidth Product
T1 (1.5 Mbps)	18 KB
T3 (45 Mbps)	549 KB
Fast Ethernet (100 Mbps)	1.2 MB
OC-3 (155 Mbps)	1.8 MB
OC-48 (2.5 Gbps)	29.6 MB
OC-192 (10 Gbps)	118.4 MB
10GigE (10 Gbps)	118.4 MB

As you can see, TCP's `AdvertisedWindow` field is in even worse shape than its `SequenceNum` field—it is not big enough to handle even a T3 connection across the continental United States, since a 16-bit field allows us to advertise a window of only 64 KB. The very same TCP extension mentioned above provides a mechanism for effectively increasing the size of the advertised window.

5.2.5 Triggering Transmission

We next consider a surprisingly subtle issue: how TCP decides to transmit a segment. As described earlier, TCP supports a byte-stream abstraction; that is, application programs write bytes into the stream, and it is up to TCP to decide that it has enough bytes to send a segment. What factors govern this decision?

If we ignore the possibility of flow control—that is, we assume the window is wide open, as would be the case when a connection first starts—then TCP has three mechanisms to trigger the transmission of a segment. First, TCP maintains a variable, typically called the *maximum segment size* (`MSS`), and it sends a segment as soon as it has collected `MSS` bytes from the sending process. `MSS` is usually set to the size of the largest segment TCP can send without causing the local IP to fragment. That is, `MSS` is set to the maximum transmission unit (MTU) of the directly connected network, minus the size of the TCP and IP headers. The second thing that triggers TCP to transmit a segment is that the sending process has explicitly asked it to do so. Specifically, TCP supports a *push* operation, and the sending process invokes this operation to effectively flush the buffer of unsent bytes. The final trigger for transmitting a segment is that a timer fires; the resulting segment contains as many bytes as are currently buffered for transmission. However, as we will soon see, this "timer" isn't exactly what you expect.

Silly Window Syndrome

Of course, we can't just ignore flow control, which plays an obvious role in throttling the sender. If the sender has `MSS` bytes of data to send and the window is open at least that much, then the sender transmits a full segment. Suppose, however, that the sender is accumulating bytes to send, but the window is currently closed. Now suppose an ACK arrives that effectively opens the window enough for the sender to transmit, say, `MSS/2` bytes. Should the sender transmit a half-full segment or wait for the window to open to a full `MSS`? The original specification was silent on this point, and early implementations of TCP decided to go ahead and transmit a half-full segment. After all, there is no telling how long it will be before the window opens further.

It turns out that the strategy of aggressively taking advantage of any available window leads to a situation now known as the *silly window syndrome*. Figure 5.9 helps visualize what happens. If you think of a TCP stream as a conveyor belt with "full" containers (data segments) going in one direction and empty containers (ACKs) going in the reverse direction, then `MSS`-sized segments correspond to large containers and 1-byte segments correspond to very small containers. As long as the sender is sending `MSS`-sized segments and the receiver ACKs at least one `MSS` of data at a time, everything is good (Figure 5.9(a)). But, what if the receiver has to reduce the window, so that at some time the sender can't send a full `MSS` of data? If the sender aggressively fills a smaller-than-`MSS` empty container as soon as it arrives, then the receiver will ACK that smaller number of bytes, and hence the small container introduced into the system remains in the system indefinitely. That is, it is immediately filled and emptied at each end and is never coalesced with adjacent containers to create larger containers, as in Figure 5.9(b). This scenario was discovered when early implementations of TCP regularly found themselves filling the network with tiny segments.

Note that the silly window syndrome is only a problem when either the sender transmits a small segment or the receiver opens the window a small amount. If neither of these happens, then the small container is never introduced into the stream. It's not possible to outlaw sending small segments; for example, the application might do a *push* after sending a single byte. It is possible, however, to keep the receiver from introducing a small container (i.e., a small open window). The rule is that after advertising a zero window the receiver must wait for space equal to an `MSS` before it advertises an open window.

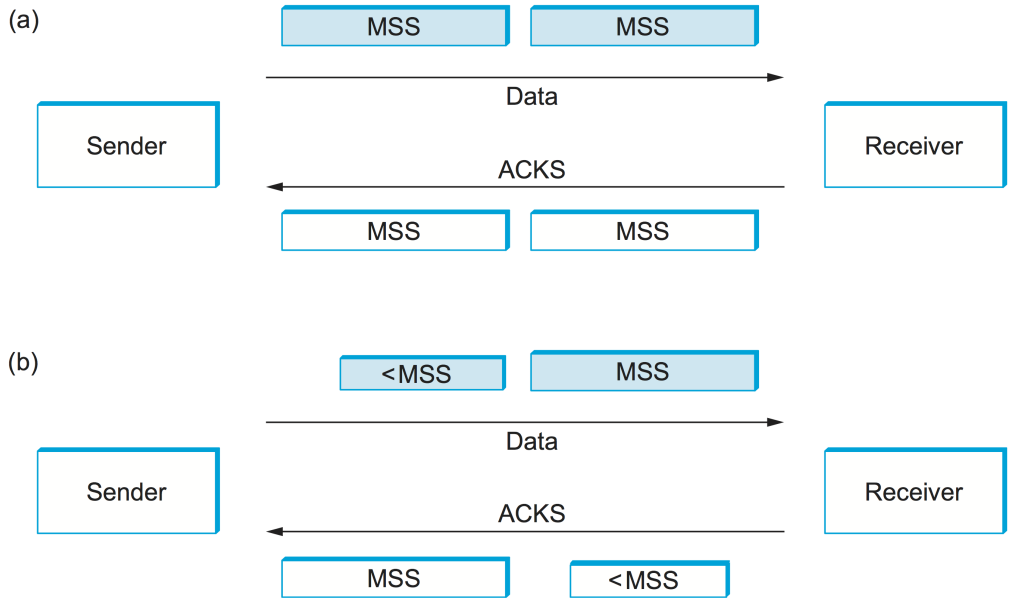


Figure 5.9.: Silly window syndrome. (a) As long as the sender sends MSS-sized segments and the receiver ACKs one MSS at a time, the system works smoothly. (b) As soon as the sender sends less than one MSS, or the receiver ACKs less than one MSS, a small "container" enters the system and continues to circulate.

Since we can't eliminate the possibility of a small container being introduced into the stream, we also need mechanisms to coalesce them. The receiver can do this by delaying ACKs—sending one combined ACK rather than multiple smaller ones—but this is only a partial solution because the receiver has no way of knowing how long it is safe to delay waiting either for another segment to arrive or for the application to read more data (thus opening the window). The ultimate solution falls to the sender, which brings us back to our original issue: When does the TCP sender decide to transmit a segment?

Nagle's Algorithm

Returning to the TCP sender, if there is data to send but the window is open less than `MSS`, then we may want to wait some amount of time before sending the available data, but the question is how long? If we wait too long, then we hurt interactive applications like Telnet. If we don't wait long enough, then we risk sending a bunch of tiny packets and falling into the silly window syndrome. The answer is to introduce a timer and to transmit when the timer expires.

While we could use a clock-based timer—for example, one that fires every 100 ms—Nagle introduced an elegant *self-clocking* solution. The idea is that as long as TCP has any data in flight, the sender will eventually receive an ACK. This ACK can be treated like a timer firing, triggering the transmission of more data. Nagle's algorithm provides a simple, unified rule for deciding when to transmit:

```
When the application produces data to send
    if both the available data and the window >= MSS
        send a full segment
    else
        if there is unACKed data in flight
            buffer the new data until an ACK arrives
```

(continues on next page)

(continued from previous page)

```
        else
            send all the new data now
```

In other words, it's always OK to send a full segment if the window allows. It's also all right to immediately send a small amount of data if there are currently no segments in transit, but if there is anything in flight the sender must wait for an ACK before transmitting the next segment. Thus, an interactive application like Telnet that continually writes one byte at a time will send data at a rate of one segment per RTT. Some segments will contain a single byte, while others will contain as many bytes as the user was able to type in one round-trip time. Because some applications cannot afford such a delay for each write it does to a TCP connection, the socket interface allows the application to turn off Nagel's algorithm by setting the `TCP_NODELAY` option. Setting this option means that data is transmitted as soon as possible.

5.2.6 Adaptive Retransmission

Because TCP guarantees the reliable delivery of data, it retransmits each segment if an ACK is not received in a certain period of time. TCP sets this timeout as a function of the RTT it expects between the two ends of the connection. Unfortunately, given the range of possible RTTs between any pair of hosts in the Internet, as well as the variation in RTT between the same two hosts over time, choosing an appropriate timeout value is not that easy. To address this problem, TCP uses an adaptive retransmission mechanism. We now describe this mechanism and how it has evolved over time as the Internet community has gained more experience using TCP.

Original Algorithm

We begin with a simple algorithm for computing a timeout value between a pair of hosts. This is the algorithm that was originally described in the TCP specification—and the following description presents it in those terms—but it could be used by any end-to-end protocol.

The idea is to keep a running average of the RTT and then to compute the timeout as a function of this RTT. Specifically, every time TCP sends a data segment, it records the time. When an ACK for that segment arrives, TCP reads the time again, and then takes the difference between these two times as a `SampleRTT`. TCP then computes an `EstimatedRTT` as a weighted average between the previous estimate and this new sample. That is,

```
EstimatedRTT = alpha x EstimatedRTT + (1 - alpha) x SampleRTT
```

The parameter `alpha` is selected to *smooth* the `EstimatedRTT`. A small `alpha` tracks changes in the RTT but is perhaps too heavily influenced by temporary fluctuations. On the other hand, a large `alpha` is more stable but perhaps not quick enough to adapt to real changes. The original TCP specification recommended a setting of `alpha` between 0.8 and 0.9. TCP then uses `EstimatedRTT` to compute the timeout in a rather conservative way:

```
TimeOut = 2 x EstimatedRTT
```

Karn/Partridge Algorithm

After several years of use on the Internet, a rather obvious flaw was discovered in this simple algorithm. The problem was that an ACK does not really acknowledge a transmission; it actually acknowledges the receipt of data. In other words, whenever a segment is retransmitted and then an ACK arrives at the sender, it is impossible to determine if this ACK should be associated with the first or the second transmission of the segment for the purpose of measuring the sample RTT. It is necessary to know which transmission to associate it with so as to compute an accurate `SampleRTT`. As illustrated in Figure 5.10, if you assume that the ACK is for the original transmission but it was really for the second, then the `SampleRTT` is too large (a); if you assume that the ACK is for the second transmission but it was actually for the first, then the `SampleRTT` is too small (b).

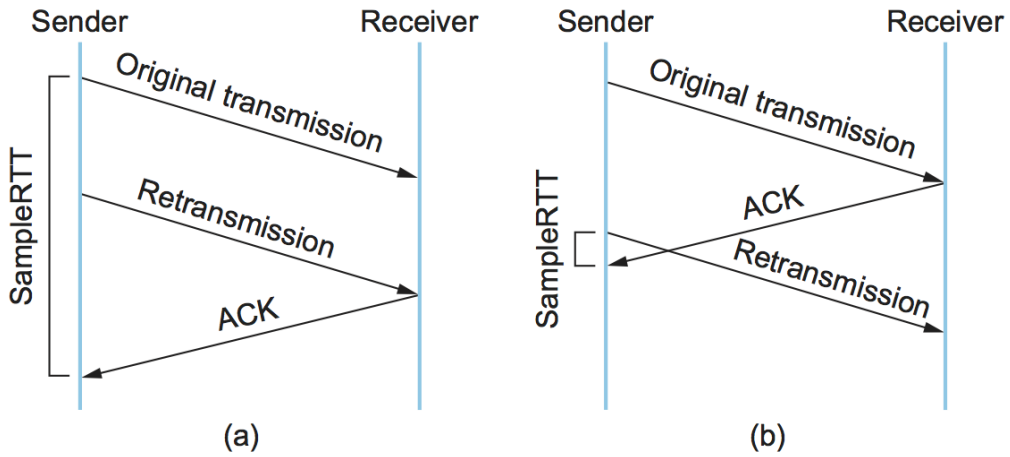


Figure 5.10.: Associating the ACK with (a) original transmission versus (b) retransmission.

The solution, which was proposed in 1987, is surprisingly simple. Whenever TCP retransmits a segment, it stops taking samples of the RTT; it only measures `SampleRTT` for segments that have been sent only once. This solution is known as the Karn/Partridge algorithm, after its inventors. Their proposed fix also includes a second small change to TCP's timeout mechanism. Each time TCP retransmits, it sets the next timeout to be twice the last timeout, rather than basing it on the last `EstimatedRTT`. That is, Karn and Partridge proposed that TCP use exponential backoff, similar to what the Ethernet does. The motivation for using exponential backoff is simple: Congestion is the most likely cause of lost segments, meaning that the TCP source should not react too aggressively to a timeout. In fact, the more times the connection times out, the more cautious the source should become. We will see this idea again, embodied in a much more sophisticated mechanism, in the next chapter.

Jacobson/Karels Algorithm

The Karn/Partridge algorithm was introduced at a time when the Internet was suffering from high levels of network congestion. Their approach was designed to fix some of the causes of that congestion, but, although it was an improvement, the congestion was not eliminated. The following year (1988), two other researchers—Jacobson and Karels—proposed a more drastic change to TCP to battle congestion. The bulk of that proposed change is described in the next chapter. Here, we focus on the aspect of that proposal that is related to deciding when to time out and retransmit a segment.

As an aside, it should be clear how the timeout mechanism is related to congestion—if you time out too soon, you may unnecessarily retransmit a segment, which only adds to the load on the network. The other reason for needing an accurate timeout value is that a timeout is taken to imply congestion, which triggers a congestion-control mechanism. Finally, note that there is nothing about the Jacobson/Karels timeout computation that is specific to TCP. It could be used by any end-to-end protocol.

The main problem with the original computation is that it does not take the variance of the sample RTTs into account. Intuitively, if the variation among samples is small, then the `EstimatedRTT` can be better trusted and there is no reason for multiplying this estimate by 2 to compute the timeout. On the other hand, a large variance in the samples suggests that the timeout value should not be too tightly coupled to the `EstimatedRTT`.

In the new approach, the sender measures a new `SampleRTT` as before. It then folds this new sample into the timeout calculation as follows:

```
Difference = SampleRTT - EstimatedRTT
EstimatedRTT = EstimatedRTT + ( delta x Difference)
Deviation = Deviation + delta (|Difference| - Deviation)
```

where `delta` and delta are fractions between 0 and 1. That is, we calculate both the mean RTT and the variation in that mean.

TCP then computes the timeout value as a function of both `EstimatedRTT` and `Deviation` as follows:

```
TimeOut = mu x EstimatedRTT + phi x Deviation
```

where based on experience, `mu` is typically set to 1 and `phi` is set to 4. Thus, when the variance is small, `TimeOut` is close to `EstimatedRTT`; a large variance causes the `Deviation` term to dominate the calculation.

Implementation

There are two items of note regarding the implementation of timeouts in TCP. The first is that it is possible to implement the calculation for `EstimatedRTT` and `Deviation` without using floating-point arithmetic. Instead, the whole calculation is scaled by 2^n, with delta selected to be $1/2^n$. This allows us to do integer arithmetic, implementing multiplicationand division using shifts, thereby achieving higher performance. The resulting calculation is given by the following code fragment, where n=3 (i.e., `delta = 1/8`). Note that `EstimatedRTT` and `Deviation` are stored in their scaled-up forms, while the value of `SampleRTT` at the start of the code and of `TimeOut` at the end are real, unscaled values. If you find the code hard to follow, you might want to try plugging some real numbers into it and verifying that it gives the same results as the equations above.

```
{
    SampleRTT -= (EstimatedRTT >> 3);
    EstimatedRTT += SampleRTT;
    if (SampleRTT < 0)
        SampleRTT = -SampleRTT;
    SampleRTT -= (Deviation >> 3);
    Deviation += SampleRTT;
```

(continues on next page)

(continued from previous page)

```
    TimeOut = (EstimatedRTT >> 3) + (Deviation >> 1);
}
```

The second point of note is that the Jacobson/Karels algorithm is only as good as the clock used to read the current time. On typical Unix implementations at the time, the clock granularity was as large as 500 ms, which is significantly larger than the average cross-country RTT of somewhere between 100 and 200 ms. To make matters worse, the Unix implementation of TCP only checked to see if a timeout should happen every time this 500-ms clock ticked and would only take a sample of the round-trip time once per RTT. The combination of these two factors could mean that a timeout would happen 1 second after the segment was transmitted. Once again, the extensions to TCP include a mechanism that makes this RTT calculation a bit more precise.

All of the retransmission algorithms we have discussed are based on acknowledgment timeouts, which indicate that a segment has probably been lost. Note that a timeout does not, however, tell the sender whether any segments it sent after the lost segment were successfully received. This is because TCP acknowledgments are cumulative; they identify only the last segment that was received without any preceding gaps. The reception of segments that occur after a gap grows more frequent as faster networks lead to larger windows. If ACKs also told the sender which subsequent segments, if any, had been received, then the sender could be more intelligent about which segments it retransmits, draw better conclusions about the state of congestion, and make better RTT estimates. A TCP extension supporting this is described in a later section.

Key Takeaway

There is one other point to make about computing timeouts. It is a surprisingly tricky business, so much so, that there is an entire RFC dedicated to the topic: RFC 6298. The takeaway is that sometimes fully specifying a protocol involves so much minutiae that the line between specification and implementation becomes blurred. That has happened more than once with TCP, causing some to argue that "the implementation **is** the specification." But that's not necessarily a bad thing as long as the reference implementation is available as open source software. More generally, the industry is seeing open source software grow in importance as open standards receed in importance. *[Next]*

5.2.7 Record Boundaries

Since TCP is a byte-stream protocol, the number of bytes written by the sender are not necessarily the same as the number of bytes read by the receiver. For example, the application might write 8 bytes, then 2 bytes, then 20 bytes to a TCP connection, while on the receiving side the application reads 5 bytes at a time inside a loop that iterates 6 times. TCP does not interject record boundaries between the 8th and 9th bytes, nor between the 10th and 11th bytes. This is in contrast to a message-oriented protocol, such as UDP, in which the message that is sent is exactly the same length as the message that is received.

Even though TCP is a byte-stream protocol, it has two different features that can be used by the sender to insert record boundaries into this byte stream, thereby informing the receiver how to break the stream of bytes into records. (Being able to mark record boundaries is useful, for example, in many database applications.) Both of these features were originally included in TCP for completely different reasons; they have only come to be used for this purpose over time.

The first mechanism is the urgent data feature, as implemented by the `URG` flag and the `UrgPtr` field in the TCP header. Originally, the urgent data mechanism was designed to allow the sending application to send *out-of-band* data to its peer. By "out of band" we mean data that is separate from the normal flow of data (e.g., a command to interrupt an operation already under way). This out-of-band data was identified in the segment using the `UrgPtr` field and was to be delivered to the receiving process as soon as it arrived, even if that meant delivering it before data with an earlier sequence number. Over time, however, this feature has not been used, so instead of signifying "urgent" data, it has come to be used to signify "special" data, such as a record marker. This use has developed because, as with the push operation, TCP on the receiving side must inform the application that urgent data has arrived. That is, the urgent data in itself is not important. It is the fact that the sending process can effectively send a signal to the receiver that is important.

The second mechanism for inserting end-of-record markers into a byte is the *push* operation. Originally, this mechanism was designed to allow the sending process to tell TCP that it should send (flush) whatever bytes it had collected to its peer. The *push* operation can be used to implement record boundaries because the specification says that TCP must send whatever data it has buffered at the source when the application says push, and, optionally, TCP at the destination notifies the application whenever an incoming segment has the PUSH flag set. If the receiving side supports this option (the socket interface does not), then the push operation can be used to break the TCP stream into records.

Of course, the application program is always free to insert record boundaries without any assistance from TCP. For example, it can send a field that indicates the length of a record that is to follow, or it can insert its own record boundary markers into the data stream.

5.2.8 TCP Extensions

We have mentioned at four different points in this section that there are now extensions to TCP that help to mitigate some problem that TCP faced as the underlying network got faster. These extensions are designed to have as small an impact on TCP as possible. In particular, they are realized as options that can be added to the TCP header. (We glossed over this point earlier, but the reason why the TCP header has a `HdrLen` field is that the header can be of variable length; the variable part of the TCP header contains the options that have been added.) The significance of adding these extensions as options rather than changing the core of the TCP header is that hosts can still communicate using TCP even if they do not implement the options. Hosts that do implement the optional extensions, however, can take advantage of them. The two sides agree that they will use the options during TCP's connection establishment phase.

The first extension helps to improve TCP's timeout mechanism. Instead of measuring the RTT using a coarse-grained event, TCP can read the actual system clock when it is about to send a segment, and put this time—think of it as a 32-bit *timestamp*—in the segment's header. The receiver then echoes this timestamp back to the sender in its acknowledgment, and the sender subtracts this timestamp from the current time to measure the RTT. In essence, the timestamp option provides a convenient place for TCP to store the record of when a segment was transmitted; it stores the time in the segment itself. Note that the endpoints in the connection do not need synchronized clocks, since the timestamp is written and read at the same end of the connection.

The second extension addresses the problem of TCP's 32-bit `SequenceNum` field wrapping around too soon on a high-speed network. Rather than define a new 64-bit sequence number field, TCP uses the 32-bit timestamp just described to effectively extend the sequence number space. In other words, TCP decides whether to accept or reject a segment based on a 64-bit identifier that has the `SequenceNum` field in the low-order 32 bits and the timestamp in the high-order 32 bits. Since the timestamp is always increasing,

it serves to distinguish between two different incarnations of the same sequence number. Note that the timestamp is being used in this setting only to protect against wraparound; it is not treated as part of the sequence number for the purpose of ordering or acknowledging data.

The third extension allows TCP to advertise a larger window, thereby allowing it to fill larger delay × bandwidth pipes that are made possible by high-speed networks. This extension involves an option that defines a *scaling factor* for the advertised window. That is, rather than interpreting the number that appears in the `AdvertisedWindow` field as indicating how many bytes the sender is allowed to have unacknowledged, this option allows the two sides of TCP to agree that the `AdvertisedWindow` field counts larger chunks (e.g., how many 16-byte units of data the sender can have unacknowledged). In other words, the window scaling option specifies how many bits each side should left-shift the `AdvertisedWindow` field before using its contents to compute an effective window.

The fourth extension allows TCP to augment its cumulative acknowledgment with selective acknowledgments of any additional segments that have been received but aren't contiguous with all previously received segments. This is the *selective acknowledgment*, or *SACK*, option. When the SACK option is used, the receiver continues to acknowledge segments normally—the meaning of the `Acknowledge` field does not change—but it also uses optional fields in the header to acknowledge any additional blocks of received data. This allows the sender to retransmit just the segments that are missing according to the selective acknowledgment.

Without SACK, there are only two reasonable strategies for a sender. The pessimistic strategy responds to a timeout by retransmitting not just the segment that timed out, but any segments transmitted subsequently. In effect, the pessimistic strategy assumes the worst: that all those segments were lost. The disadvantage of the pessimistic strategy is that it may unnecessarily retransmit segments that were successfully received the first time. The other strategy is the optimistic strategy, which responds to a timeout by retransmitting only the segment that timed out. In effect, the optimistic approach assumes the rosiest scenario: that only the one segment has been lost. The disadvantage of the optimistic strategy is that it is very slow, unnecessarily, when a series of consecutive segments has been lost, as might happen when there is congestion. It is slow because each segment's loss is not discovered until the sender receives an ACK for its retransmission of the previous segment. So it consumes one RTT per segment until it has retransmitted all the segments in the lost series. With the SACK option, a better strategy is available to the sender: retransmit just the segments that fill the gaps between the segments that have been selectively acknowledged.

These extensions, by the way, are not the full story. We'll see some more extensions in the next chapter when we look at how TCP handles congestion. The Internet Assigned Numbers Authority (IANA) keeps track of all the options that are defined for TCP (and for many other Internet protocols). See the references at the end of the chapter for a link to IANA's protocol number registry.

5.2.9 Performance

Recall that Chapter 1 introduced the two quantitative metrics by which network performance is evaluated: latency and throughput. As mentioned in that discussion, these metrics are influenced not only by the underlying hardware (e.g., propagation delay and link bandwidth) but also by software overheads. Now that we have a complete software-based protocol graph available to us that includes alternative transport protocols, we can discuss how to meaningfully measure its performance. The importance of such measurements is that they represent the performance seen by application programs.

We begin, as any report of experimental results should, by describing our experimental method. This in-

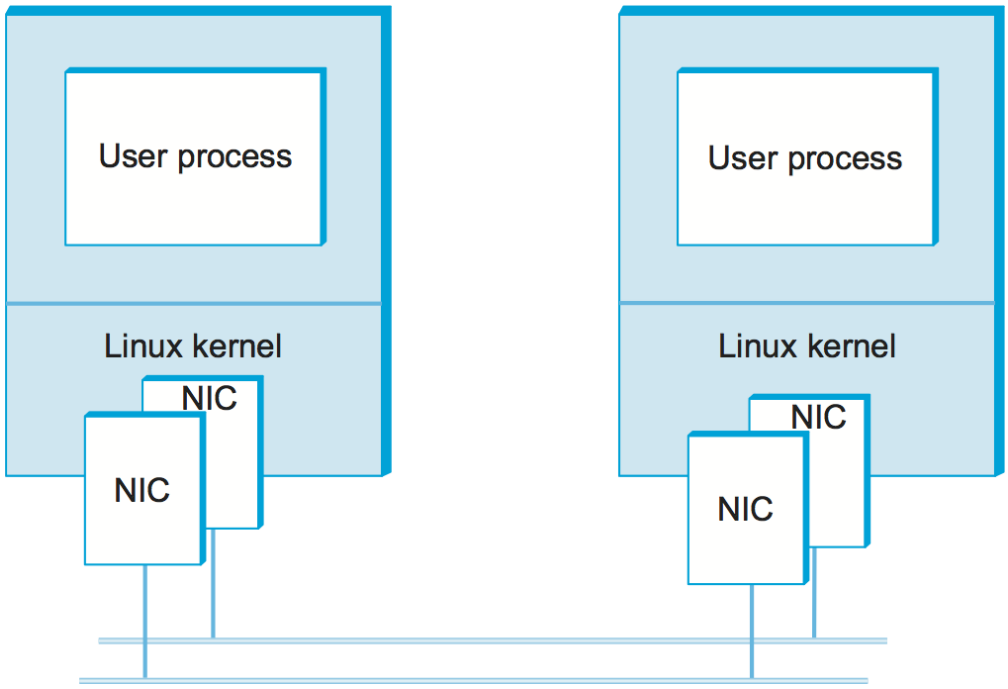


Figure 5.11.: Measured system: Two Linux workstations and a pair of Gbps Ethernet links.

cludes the apparatus used in the experiments; in this case, each workstation has a pair of dual CPU 2.4-GHz Xeon processors running Linux. In order to enable speeds above 1 Gbps, a pair of Ethernet adaptors (labeled NIC, for network interface card) are used on each machine. The Ethernet spans a single machine room so propagation is not an issue, making this a measure of processor/software overheads. A test program running on top of the socket interface simply tries to transfer data as quickly as possible from one machine to the other. Figure 5.11 illustrates the setup.

You may notice that this experimental setup is not especially bleeding edge in terms of the hardware or link speed. The point of this section is not to show how fast a particular protocol can run, but to illustrate the general methodology for measuring and reporting protocol performance.

The throughput test is performed for a variety of message sizes using a standard benchmarking tool called TTCP. The results of the throughput test are given in Figure 5.12. The key thing to notice in this graph is that throughput improves as the messages get larger. This makes sense—each message involves a certain amount of overhead, so a larger message means that this overhead is amortized over more bytes. The throughput curve flattens off above 1 KB, at which point the per-message overhead becomes insignificant when compared to the large number of bytes that the protocol stack has to process.

It's worth noting that the maximum throughput is less than 2 Gbps, the available link speed in this setup. Further testing and analysis of results would be needed to figure out where the bottleneck is (or if there is more than one). For example, looking at CPU load might give an indication of whether the CPU is the bottleneck or whether memory bandwidth, adaptor performance, or some other issue is to blame.

We also note that the network in this test is basically "perfect." It has almost no delay or loss, so the only factors affecting performance are the TCP implementation and the workstation hardware and software. By contrast, most of the time we deal with networks that are far from perfect, notably our bandwidth-constrained, last-mile links and loss-prone wireless links. Before we can fully appreciate how these links affect TCP

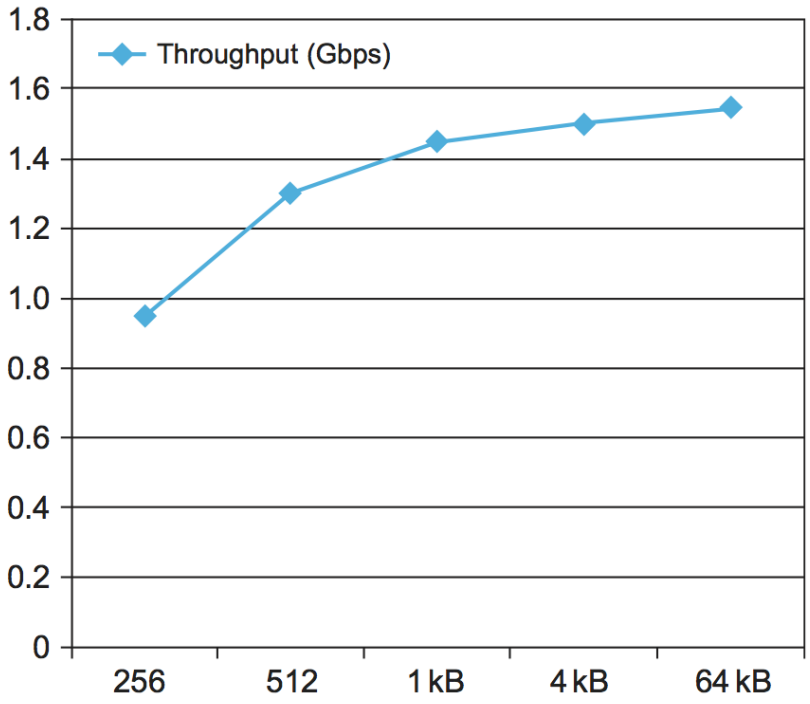


Figure 5.12.: Measured throughput using TCP, for various message sizes.

performance, we need to understand how TCP deals with *congestion*, which is the topic of the next chapter.

At various times in the history of networking, the steadily increasing speed of network links has threatened to run ahead of what could be delivered to applications. For example, a large research effort was begun in the United States in 1989 to build "gigabit networks," where the goal was not only to build links and switches that could run at 1Gbps or higher but also to deliver that throughput all the way to a single application process. There were some real problems (e.g., network adaptors, workstation architectures, and operating systems all had to be designed with network-to-application throughput in mind) and also some perceived problems that turned out to be not so serious. High on the list of such problems was the concern that existing transport protocols, TCP in particular, might not be up to the challenge of gigabit operation.

As it turns out, TCP has done well keeping up with the increasing demands of high-speed networks and applications. One of the most important factors was the introduction of window scaling to deal with larger bandwidth-delay products. However, there is often a big difference between the theoretical performance of TCP and what is achieved in practice. Relatively simple problems like copying the data more times than necessary as it passes from network adaptor to application can drive down performance, as can insufficient buffer memory when the bandwidth-delay product is large. And the dynamics of TCP are complex enough (as will become even more apparent in the next chapter) that subtle interactions among network behavior, application behavior, and the TCP protocol itself can dramatically alter performance.

For our purposes, it's worth noting that TCP continues to perform very well as network speeds increase, and when it runs up against some limit (normally related to congestion, increasing bandwidth-delay products, or both), researchers rush in to find solutions. We've seen some of those in this chapter, and we'll see some more in the next.

5.2.10 Alternative Design Choices (SCTP, QUIC)

Although TCP has proven to be a robust protocol that satisfies the needs of a wide range of applications, the design space for transport protocols is quite large. TCP is by no means the only valid point in that design space. We conclude our discussion of TCP by considering alternative design choices. While we offer an explanation for why TCP's designers made the choices they did, we observe that other protocols exist that have made other choices, and more such protocols may appear in the future.

First, we have suggested from the very first chapter of this book that there are at least two interesting classes of transport protocols: stream-oriented protocols like TCP and request/reply protocols like RPC. In other words, we have implicitly divided the design space in half and placed TCP squarely in the stream-oriented half of the world. We could further divide the stream-oriented protocols into two groups—reliable and unreliable—with the former containing TCP and the latter being more suitable for interactive video applications that would rather drop a frame than incur the delay associated with a retransmission.

This exercise in building a transport protocol taxonomy is interesting and could be continued in greater and greater detail, but the world isn't as black and white as we might like. Consider the suitability of TCP as a transport protocol for request/reply applications, for example. TCP is a full-duplex protocol, so it would be easy to open a TCP connection between the client and server, send the request message in one direction, and send the reply message in the other direction. There are two complications, however. The first is that TCP is a *byte*-oriented protocol rather than a *message*-oriented protocol, and request/reply applications always deal with messages. (We explore the issue of bytes versus messages in greater detail in a moment.) The second complication is that in those situations where both the request message and the reply message fit in a single network packet, a well-designed request/reply protocol needs only two packets to implement the exchange, whereas TCP would need at least nine: three to establish the connection, two for the message exchange, and four to tear down the connection. Of course, if the request or reply messages are large enough to require multiple network packets (e.g., it might take 100 packets to send a 100,000-byte reply message), then the overhead of setting up and tearing down the connection is inconsequential. In other words, it isn't always the case that a particular protocol cannot support a certain functionality; it's sometimes the case that one design is more efficient than another under particular circumstances.

Second, as just suggested, you might question why TCP chose to provide a reliable *byte*-stream service rather than a reliable *message*-stream service; messages would be the natural choice for a database application that wants to exchange records. There are two answers to this question. The first is that a message-oriented protocol must, by definition, establish an upper bound on message sizes. After all, an infinitely long message is a byte stream. For any message size that a protocol selects, there will be applications that want to send larger messages, rendering the transport protocol useless and forcing the application to implement its own transport-like services. The second reason is that, while message-oriented protocols are definitely more appropriate for applications that want to send records to each other, you can easily insert record boundaries into a byte stream to implement this functionality.

A third decision made in the design of TCP is that it delivers bytes *in order* to the application. This means that it may hold onto bytes that were received out of order from the network, awaiting some missing bytes to fill a hole. This is enormously helpful for many applications but turns out to be quite unhelpful if the application is capable of processing data out of order. As a simple example, a Web page containing multiple embedded objects doesn't need all the objects to be delivered in order before starting to display the page. In fact, there is a class of applications that would prefer to handle out-of-order data at the application layer, in return for getting data sooner when packets are dropped or misordered within the network. The desire to support such applications led to the creation of not one but two IETF standard transport protocols. The first

of these was SCTP, the *Stream Control Transmission Protocol*. SCTP provides a partially ordered delivery service, rather than the strictly ordered service of TCP. (SCTP also makes some other design decisions that differ from TCP, including message orientation and support of multiple IP addresses for a single session.) More recently, the IETF has been standardizing a protocol optimized for Web traffic, known as QUIC. More on QUIC in a moment.

Fourth, TCP chose to implement explicit setup/teardown phases, but this is not required. In the case of connection setup, it would be possible to send all necessary connection parameters along with the first data message. TCP elected to take a more conservative approach that gives the receiver the opportunity to reject the connection before any data arrives. In the case of teardown, we could quietly close a connection that has been inactive for a long period of time, but this would complicate applications like remote login that want to keep a connection alive for weeks at a time; such applications would be forced to send out-of-band "keep alive" messages to keep the connection state at the other end from disappearing.

Finally, TCP is a window-based protocol, but this is not the only possibility. The alternative is a *rate-based* design, in which the receiver tells the sender the rate—expressed in either bytes or packets per second—at which it is willing to accept incoming data. For example, the receiver might inform the sender that it can accommodate 100 packets a second. There is an interesting duality between windows and rate, since the number of packets (bytes) in the window, divided by the RTT, is exactly the rate. For example, a window size of 10 packets and a 100-ms RTT implies that the sender is allowed to transmit at a rate of 100 packets a second. It is by increasing or decreasing the advertised window size that the receiver is effectively raising or lowering the rate at which the sender can transmit. In TCP, this information is fed back to the sender in the `AdvertisedWindow` field of the ACK for every segment. One of the key issues in a rate-based protocol is how often the desired rate—which may change over time—is relayed back to the source: Is it for every packet, once per RTT, or only when the rate changes? While we have just now considered window versus rate in the context of flow control, it is an even more hotly contested issue in the context of congestion control, which we will discuss in the next chapter.

QUIC

QUIC, *Quick UDP Internet Connections*, originated at Google in 2012 and, at the time of writing, is still undergoing standardization at the IETF. It has already seen a moderate amount of deployment (in some Web browsers and quite a number of popular Web sites). The fact that it has been successful to this degree is in itself an interesting part of the QUIC story, and indeed deployability was a key consideration for the designers of the protocol.

The motivation for QUIC comes directly from the points we noted above about TCP: certain design decisions have turned out to be non-optimal for a range of applications that run over TCP, with HTTP (Web) traffic being a particularly notable example. These issues have become more noticeable over time, due to factors such as the rise of high-latency wireless networks, the availability of multiple networks for a single device (e.g., Wi-Fi and cellular), and the increasing use of encrypted, authenticated connections on the Web. While a full description of QUIC is beyond our scope, some of the key design decisions are worth discussing.

Multipath TCP

It isn't always necessary to define a new protocol if you find an existing protocol does not adequately serve a particular use case. Sometimes it's possible to make substantial changes in how an existing protocol is

implemented, yet remain true to the original spec. Multipath TCP is an example of such a situation.

The idea of Multipath TCP is to steer packets over multiple paths through the Internet, for example, by using two different IP addresses for one of the end-points. This can be especially helpful when delivering data to a mobile device that is connected to both Wi-Fi and the cellular network (and hence, has two unique IP addresses). Being wireless, both networks can experience significant packet-loss, so being able to use both to carry packets can dramatically improve the user experience. The key is for the receiving side of TCP to reconstruct the original, in-order byte stream before passing data up to the application, which remains unaware it is sitting on top of Multipath TCP. (This is in contrast to applications that purposely open two or more TCP connections to get better performance.)

As simple as Multipath TCP sounds, it is incredibly difficult to get right because it breaks many assumptions about how TCP flow control, in-order segment reassembly, and congestion control are implemented. We leave it as an exercise for the reader to explore these subtleties. Doing so is a great way to make sure your basic understanding of TCP is sound.

If network latency is high—in the hundreds of milliseconds—then a few RTTs can quickly add up to a visible annoyance for an end user. Establishing an HTTP session over TCP with Transport Layer Security (*Section 8.5*) would typically take three round trips (one for TCP session establishment and two for setting up the encryption parameters) before the first HTTP message could be sent. The designers of QUIC recognized that this delay—the direct result of a layered approach to protocol design—could be dramatically reduced if connection setup and the required security handshakes were combined and optimized for minimal round trips.

Note also how the presence of multiple network interfaces might affect the design. If your mobile phone loses its Wi-Fi connection and needs to switch to a cellular connection, that would typically require both a TCP timeout on one connection and a new series of handshakes on the other. Making the connection something that can persist over different network layer connections was another design goal for QUIC.

Finally, as noted above, the reliable byte stream model for TCP is a poor match to a Web page request, when many objects need to be fetched and page rendering could begin before they have all arrived. While one workaround for this would be to open multiple TCP connections in parallel, this approach (which was used in the early days of the Web) has its own set of drawbacks, notably on congestion control (see *Chapter 6*).

Interestingly, by the time QUIC emerged, many design decisions had been made that presented challenges for the deployment of a new transport protocol. Notably, many "middleboxes" such as NATs and firewalls (see *Section 8.5*) have enough understanding of the existing widespread transport protocols (TCP and UDP) that they can't be relied upon to pass a new transport protocol. As a result, QUIC actually rides on top of UDP. In other words, it is a transport protocol running on top of a transport protocol. This is not as uncommon as our focus on layering might suggest, as the next two subsections also illustrate.

QUIC implements fast connection establishment with encryption and authentication in the first RTT. It provides a connection identifier than persists across changes in the underlying network. It supports the multiplexing of several streams onto a single transport connection, to avoid the head-of-line blocking that may arise when a single packet is dropped while other useful data continues to arrive. And it preserves the congestion avoidance properties of TCP, an important aspect of transport protocols that we return to in *Chapter 6*.

QUIC is a most interesting development in the world of transport protocols. Many of the limitations of TCP have been known for decades, but QUIC represents one of the most successful efforts to date to stake out a different point in the design space. Because QUIC was inspired by experience with HTTP and the

Web—which arose long after TCP was well established in the Internet—it presents a fascinating case study in the unforeseen consequences of layered designs and in the evolution of the Internet.

5.3 Remote Procedure Call

A common pattern of communication used by application programs structured as a *client/server* pair is the request/reply message transaction: A client sends a request message to a server, and the server responds with a reply message, with the client blocking (suspending execution) to wait for the reply. Figure 5.13 illustrates the basic interaction between the client and server in such an exchange.

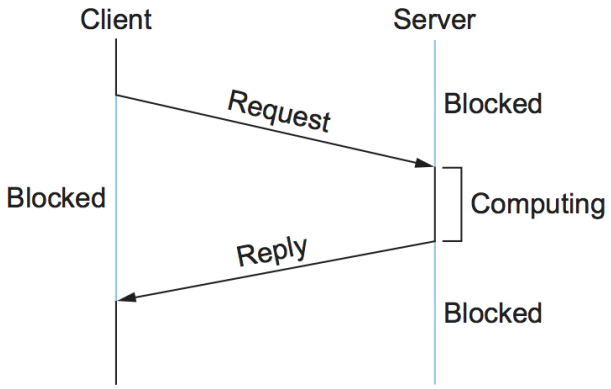


Figure 5.13.: Timeline for RPC.

A transport protocol that supports the request/reply paradigm is much more than a UDP message going in one direction followed by a UDP message going in the other direction. It needs to deal with correctly identifying processes on remote hosts and correlating requests with responses. It may also need to overcome some or all of the limitations of the underlying network outlined in the problem statement at the beginning of this chapter. While TCP overcomes these limitations by providing a reliable byte-stream service, it doesn't perfectly match the request/reply paradigm either. This section describes a third category of transport protocol, called *Remote Procedure Call* (RPC), that more closely matches the needs of an application involved in a request/reply message exchange.

5.3.1 RPC Fundamentals

RPC is not technically a protocol—it is better thought of as a general mechanism for structuring distributed systems. RPC is popular because it is based on the semantics of a local procedure call—the application program makes a call into a procedure without regard for whether it is local or remote and blocks until the call returns. An application developer can be largely unaware of whether the procedure is local or remote, simplifying his task considerably. When the procedures being called are actually methods of remote objects in an object-oriented language, RPC is known as *remote method invocation* (RMI). While the RPC concept is simple, there are two main problems that make it more complicated than local procedure calls:

- The network between the calling process and the called process has much more complex properties than the backplane of a computer. For example, it is likely to limit message sizes and has a tendency to lose and reorder messages.

- The computers on which the calling and called processes run may have significantly different architectures and data representation formats.

Thus, a complete RPC mechanism actually involves two major components:

1. A protocol that manages the messages sent between the client and the server processes and that deals with the potentially undesirable properties of the underlying network.
2. Programming language and compiler support to package the arguments into a request message on the client machine and then to translate this message back into the arguments on the server machine, and likewise with the return value (this piece of the RPC mechanism is usually called a *stub compiler*).

Figure 5.14 schematically depicts what happens when a client invokes a remote procedure. First, the client calls a local stub for the procedure, passing it the arguments required by the procedure. This stub hides the fact that the procedure is remote by translating the arguments into a request message and then invoking an RPC protocol to send the request message to the server machine. At the server, the RPC protocol delivers the request message to the server stub, which translates it into the arguments to the procedure and then calls the local procedure. After the server procedure completes, it returns in a reply message that it hands off to the RPC protocol for transmission back to the client. The RPC protocol on the client passes this message up to the client stub, which translates it into a return value that it returns to the client program.

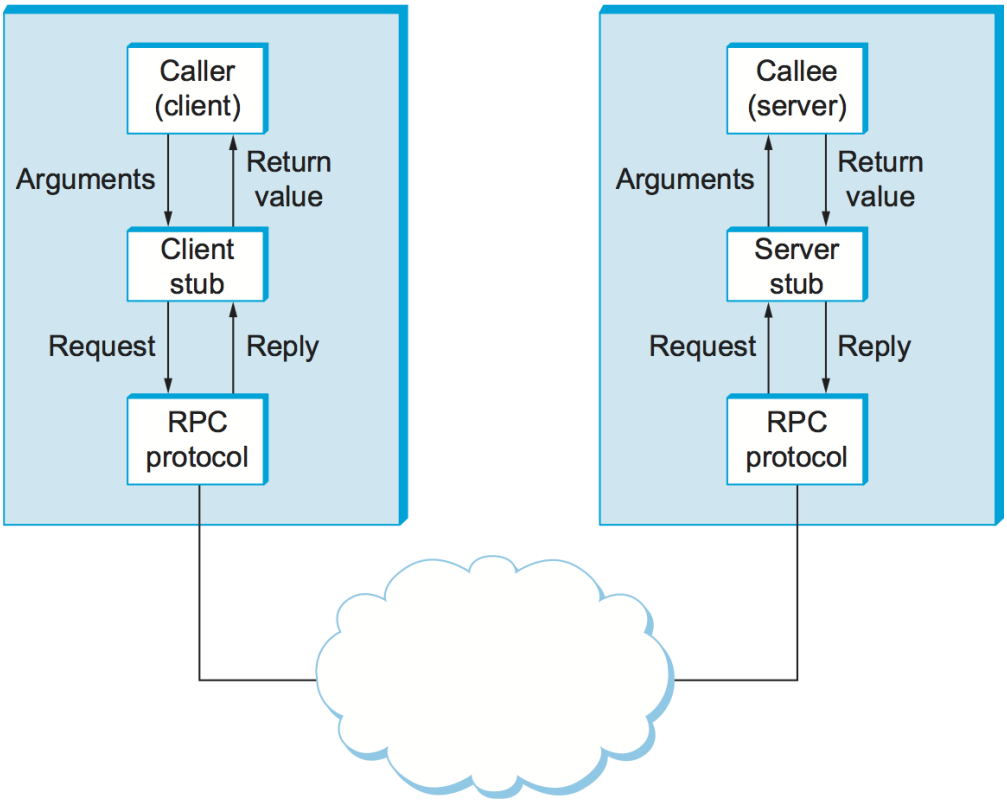


Figure 5.14.: Complete RPC mechanism.

This section considers just the protocol-related aspects of an RPC mechanism. That is, it ignores the stubs and focuses instead on the RPC protocol, sometimes referred to as a request/reply protocol, that transmits messages between client and server. The transformation of arguments into messages and *vice versa* is covered elsewhere. It is also important to keep in mind that the client and server programs are written in

some programming language, meaning that a given RPC mechanism might support Python stubs, Java stubs, GoLang stubs, and so on, each of which includes language-specific idioms for how procedures are invoked.

The term *RPC* refers to a type of protocol rather than a specific standard like TCP, so specific RPC protocols vary in the functions they perform. And, unlike TCP, which is the dominant reliable byte-stream protocol, there is no one dominant RPC protocol. Thus, in this section we will talk more about alternative design choices than previously.

Identifiers in RPC

Two functions that must be performed by any RPC protocol are:

- Provide a name space for uniquely identifying the procedure to be called.
- Match each reply message to the corresponding request message.

The first problem has some similarities to the problem of identifying nodes in a network, something IP addresses do, for example). One of the design choices when identifying things is whether to make this name space flat or hierarchical. A flat name space would simply assign a unique, unstructured identifier (e.g., an integer) to each procedure, and this number would be carried in a single field in an RPC request message. This would require some kind of central coordination to avoid assigning the same procedure number to two different procedures. Alternatively, the protocol could implement a hierarchical name space, analogous to that used for file pathnames, which requires only that a file's "basename" be unique within its directory. This approach potentially simplifies the job of ensuring uniqueness of procedure names. A hierarchical name space for RPC could be implemented by defining a set of fields in the request message format, one for each level of naming in, say, a two- or three-level hierarchical name space.

The key to matching a reply message to the corresponding request is to uniquely identify request-replies pairs using a message ID field. A reply message had its message ID field set to the same value as the request message. When the client RPC module receives the reply, it uses the message ID to search for the corresponding outstanding request. To make the RPC transaction appear like a local procedure call to the caller, the caller is blocked until the reply message is received. When the reply is received, the blocked caller is identified based on the request number in the reply, the remote procedure's return value is obtained from the reply, and the caller is unblocked so that it can return with that return value.

One of the recurrent challenges in RPC is dealing with unexpected responses, and we see this with message IDs. For example, consider the following pathological (but realistic) situation. A client machine sends a request message with a message ID of 0, then crashes and reboots, and then sends an unrelated request message, also with a message ID of 0. The server may not have been aware that the client crashed and rebooted and, upon seeing a request message with a message ID of 0, acknowledges it and discards it as a duplicate. The client never gets a response to the request.

One way to eliminate this problem is to use a *boot ID*. A machine's boot ID is a number that is incremented each time the machine reboots; this number is read from nonvolatile storage (e.g., a disk or flash drive), incremented, and written back to the storage device during the machine's start-up procedure. This number is then put in every message sent by that host. If a message is received with an old message ID but a new boot ID, it is recognized as a new message. In effect, the message ID and boot ID combine to form a unique ID for each transaction.

Overcoming Network Limitations

RPC protocols often perform additional functions to deal with the fact that networks are not perfect channels. Two such functions are:

- Provide reliable message delivery
- Support large message sizes through fragmentation and reassembly

An RPC protocol might "define this problem away" by choosing to run on top of a reliable protocol like TCP, but in many cases, the RCP protocol implements its own reliable message delivery layer on top of an unreliable substrate (e.g., UDP/IP). Such an RPC protocol would likely implement reliability using acknowledgments and timeouts, similarly to TCP.

The basic algorithm is straightforward, as illustrated by the timeline given in Figure 5.15. The client sends a request message and the server acknowledges it. Then, after executing the procedure, the server sends a reply message and the client acknowledges the reply.

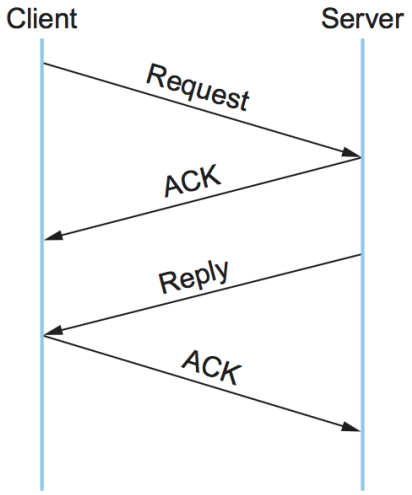


Figure 5.15.: Simple timeline for a reliable RPC protocol.

Either a message carrying data (a request message or a reply message) or the ACK sent to acknowledge that message may be lost in the network. To account for this possibility, both client and server save a copy of each message they send until an ACK for it has arrived. Each side also sets a RETRANSMIT timer and resends the message should this timer expire. Both sides reset this timer and try again some agreed-upon number of times before giving up and freeing the message.

If an RPC client receives a reply message, clearly the corresponding request message must have been received by the server. Hence, the reply message itself is an *implicit acknowledgment*, and any additional acknowledgment from the server is not logically necessary. Similarly, a request message could implicitly acknowledge the preceding reply message—assuming the protocol makes request-reply transactions sequential, so that one transaction must complete before the next begins. Unfortunately, this sequentiality would severely limit RPC performance.

A way out of this predicament is for the RPC protocol to implement a *channel* abstraction. Within a given channel, request/reply transactions are sequential—there can be only one transaction active on a given channel at any given time—but there can be multiple channels. Or said another way, the channel abstraction makes it possible to *multiplex* multiple RPC request/reply transactions between a client/server pair.

Each message includes a channel ID field to indicate which channel the message belongs to. A request message in a given channel would implicitly acknowledge the previous reply in that channel, if it hadn't already been acknowledged. An application program can open multiple channels to a server if it wants to have more than one request/reply transaction between them at the same time (the application would need multiple threads). As illustrated in Figure 5.16, the reply message serves to acknowledge the request message, and a subsequent request acknowledges the preceding reply. Note that we saw a very similar approach—called *concurrent logical channels*—in an earlier section as a way to improve on the performance of a stop-and-wait reliability mechanism.

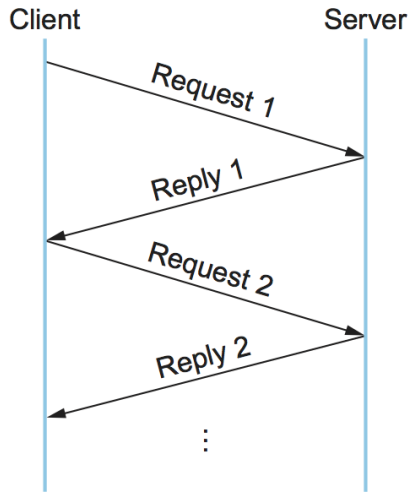


Figure 5.16.: Timeline for a reliable RPC protocol using implicit acknowledgment.

Another complication that RPC must address is that the server may take an arbitrarily long time to produce the result, and, worse yet, it may crash before generating the reply. Keep in mind that we are talking about the period of time after the server has acknowledged the request but before it has sent the reply. To help the client distinguish between a slow server and a dead server, the RPC's client side can periodically send an "Are you alive?" message to the server, and the server side responds with an ACK. Alternatively, the server could send "I am still alive" messages to the client without the client having first solicited them. The approach is more scalable because it puts more of the per-client burden (managing the timeout timer) on the clients.

RPC reliability may include the property known as *at-most-once semantics*. This means that for every request message that the client sends, at most one copy of that message is delivered to the server. Each time the client calls a remote procedure, that procedure is invoked at most one time on the server machine. We say "at most once" rather than "exactly once" because it is always possible that either the network or the server machine has failed, making it impossible to deliver even one copy of the request message.

To implement at-most-once semantics, RPC on the server side must recognize duplicate requests (and ignore them), even if it has already successfully replied to the original request. Hence, it must maintain some state information that identifies past requests. One approach is to identify requests using sequence numbers, so a server need only remember the most recent sequence number. Unfortunately, this would limit an RPC to one outstanding request (to a given server) at a time, since one request must be completed before the request with the next sequence number can be transmitted. Once again, channels provide a solution. The server could recognize duplicate requests by remembering the current sequence number for each channel, without limiting the client to one request at a time.

As obvious as at-most-once sounds, not all RPC protocols support this behavior. Some support a semantics that is facetiously called *zero-or-more* semantics; that is, each invocation on a client results in the remote procedure being invoked zero or more times. It is not difficult to understand how this would cause problems for a remote procedure that changed some local state variable (e.g., incremented a counter) or that had some externally visible side effect (e.g., launched a missile) each time it was invoked. On the other hand, if the remote procedure being invoked is *idempotent*—multiple invocations have the same effect as just one—then the RPC mechanism need not support at-most-once semantics; a simpler (possibly faster) implementation will suffice.

As was the case with reliability, the two reasons why an RPC protocol might implement message fragmentation and reassembly are that it is not provided by the underlying protocol stack or that it can be implemented more efficiently by the RPC protocol. Consider the case where RPC is implemented on top of UDP/IP and relies on IP for fragmentation and reassembly. If even one fragment of a message fails to arrive within a certain amount of time, IP discards the fragments that did arrive and the message is effectively lost. Eventually, the RPC protocol (assuming it implements reliability) would time out and retransmit the message. In contrast, consider an RPC protocol that implements its own fragmentation and reassembly and aggressively ACKs or NACKs (negatively acknowledges) individual fragments. Lost fragments would be more quickly detected and retransmitted, and only the lost fragments would be retransmitted, not the whole message.

Synchronous versus Asynchronous Protocols

One way to characterize a protocol is by whether it is *synchronous* or *asynchronous*. The precise meaning of these terms depends on where in the protocol hierarchy you use them. At the transport layer, it is most accurate to think of them as defining the extremes of a spectrum rather than as two mutually exclusive alternatives. The key attribute of any point along the spectrum is how much the sending process knows after the operation to send a message returns. In other words, if we assume that an application program invokes a `send` operation on a transport protocol, then exactly what does the application know about the success of the operation when the `send` operation returns?

At the *asynchronous* end of the spectrum, the application knows absolutely nothing when `send` returns. Not only does it not know if the message was received by its peer, but it doesn't even know for sure that the message has successfully left the local machine. At the *synchronous* end of the spectrum, the `send` operation typically returns a reply message. That is, the application not only knows that the message it sent was received by its peer, but it also knows that the peer has returned an answer. Thus, synchronous protocols implement the request/reply abstraction, while asynchronous protocols are used if the sender wants to be able to transmit many messages without having to wait for a response. Using this definition, RPC protocols are usually synchronous protocols.

Although we have not discussed them in this chapter, there are interesting points between these two extremes. For example, the transport protocol might implement `send` so that it blocks (does not return) until the message has been successfully received at the remote machine, but returns before the sender's peer on that machine has actually processed and responded to it. This is sometimes called a *reliable datagram protocol*.

5.3.2 RPC Implementations (SunRPC, DCE, gRPC)

We now turn our discussion to some example implementations of RPC protocols. These will serve to highlight some of the different design decisions that protocol designers have made. Our first example is

SunRPC, a widely used RPC protocol also known as Open Network Computing RPC (ONC RPC). Our second example, which we will refer to as DCE-RPC, is part of the Distributed Computing Environment (DCE). DCE is a set of standards and software for building distributed systems that was defined by the Open Software Foundation (OSF), a consortium of computer companies that originally included IBM, Digital Equipment Corporation, and Hewlett-Packard; today, OSF goes by the name The Open Group. Our third example is gRPC, a popular RPC mechanism that Google has open sourced, based on an RPC mechanism that they have been using internally to implement cloud services in their datacenters.

These three examples represent interesting alternative design choices in the RPC solution space, but least you think they are the only options, we describe three other RPC-like mechanisms (WSDL, SOAP, and REST) in the context of web services in Chapter 9.

SunRPC

SunRPC became a *de facto* standard thanks to its wide distribution with Sun workstations and the central role it plays in Sun's popular Network File System (NFS). The IETF subsequently adopted it as a standard Internet protocol under the name ONC RPC.

SunRPC can be implemented over several different transport protocols. Figure 5.17 illustrates the protocol graph when SunRPC is implemented on UDP. As we noted earlier in this section, a strict layerist might frown on the idea of running a transport protocol over a transport protocol, or argue that RPC must be something other than a transport protocol since it appears "above" the transport layer. Pragmatically, the design decision to run RPC over an existing transport layer makes quite a lot of sense, as will be apparent in the following discussion.

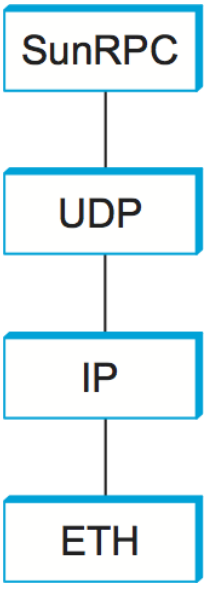


Figure 5.17.: Protocol graph for SunRPC on top of UDP.

SunRPC uses two-tier identifiers to identify remote procedures: a 32-bit program number and a 32-bit procedure number. (There is also a 32-bit version number, but we ignore that in the following discussion.) For example, the NFS server has been assigned program number `x00100003`, and within this program `getattr` is procedure `1`, `setattr` is procedure `2`, `read` is procedure `6`, `write` is procedure `8`, and so

on. The program number and procedure number are transmitted in the SunRPC request message's header, whose fields are shown in Figure 5.18. The server—which may support several program numbers—is responsible for calling the specified procedure of the specified program. A SunRPC request really represents a request to call the specified program and procedure on the particular machine to which the request was sent, even though the same program number may be implemented on other machines in the same network. Thus, the address of the server's machine (e.g., an IP address) is an implicit third tier of the RPC address.

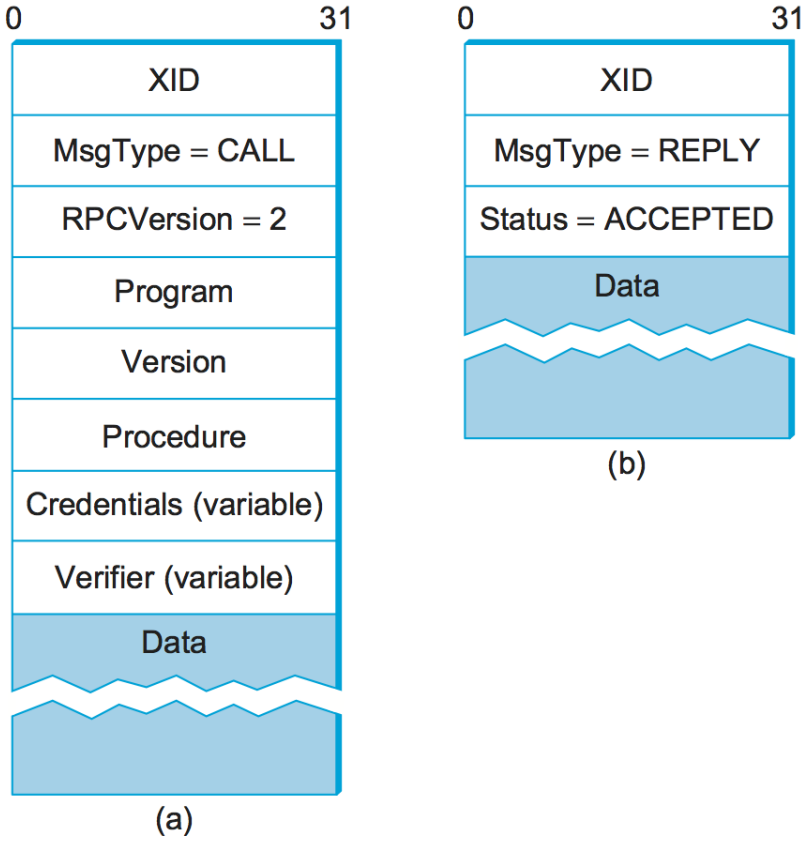


Figure 5.18.: SunRPC header formats: (a) request; (b) reply.

Different program numbers may belong to different servers on the same machine. These different servers have different transport layer demux keys (e.g., UDP ports), most of which are not well-known numbers but instead are assigned dynamically. These demux keys are called *transport selectors*. How can a SunRPC client that wants to talk to a particular program determine which transport selector to use to reach the corresponding server? The solution is to assign a well-known address to *just one* program on the remote machine and let that program handle the task of telling clients which transport selector to use to reach any other program on the machine. The original version of this SunRPC program is called the *Port Mapper*, and it supports only UDP and TCP as underlying protocols. Its program number is `x00100000`, and its well-known port is `111`. RPCBIND, which evolved from the Port Mapper, supports arbitrary underlying transport protocols. As each SunRPC server starts, it calls an RPCBIND registration procedure, on the server's own home machine, to register its transport selector and the program numbers that it supports. A remote client can then call an RPCBIND lookup procedure to look up the transport selector for a particular program number.

To make this more concrete, consider an example using the Port Mapper with UDP. To send a request message to NFS's `read` procedure, a client first sends a request message to the Port Mapper at well-known

UDP port `111`, asking that procedure `3` be invoked to map program number `x00100003` to the UDP port where the NFS program currently resides. The client then sends a SunRPC request message with program number `x00100003` and procedure number `6` to this UDP port, and the SunRPC module listening at that port calls the NFS `read` procedure. The client also caches the program-to-port number mapping so that it need not go back to the Port Mapper each time it wants to talk to the NFS program.[1]

To match up a reply message with the corresponding request, so that the result of the RPC can be returned to the correct caller, both request and reply message headers include a `XID` (transaction ID) field, as in Figure 5.18. A `XID` is a unique transaction ID used only by one request and the corresponding reply. After the server has successfully replied to a given request, it does not remember the `XID`. Because of this, SunRPC cannot guarantee at-most-once semantics.

The details of SunRPC's semantics depend on the underlying transport protocol. It does not implement its own reliability, so it is only reliable if the underlying transport is reliable. (Of course, any application that runs over SunRPC may also choose to implement its own reliability mechanisms above the level of SunRPC.) The ability to send request and reply messages that are larger than the network MTU is also dependent on the underlying transport. In other words, SunRPC does not make any attempt to improve on the underlying transport when it comes to reliability and message size. Since SunRPC can run over many different transport protocols, this gives it considerable flexibility without complicating the design of the RPC protocol itself.

Returning to the SunRPC header format of Figure 5.18, the request message contains variable-length `Credentials` and `Verifier` fields, both of which are used by the client to authenticate itself to the server—that is, to give evidence that the client has the right to invoke the server. How a client authenticates itself to a server is a general issue that must be addressed by any protocol that wants to provide a reasonable level of security. This topic is discussed in more detail in another chapter.

DCE-RPC

DCE-RPC is the RPC protocol at the core of the DCE system and was the basis of the RPC mechanism underlying Microsoft's DCOM and ActiveX. It can be used with the Network Data Representation (NDR) stub compiler described in another chapter, but it also serves as the underlying RPC protocol for the Common Object Request Broker Architecture (CORBA), which is an industry-wide standard for building distributed, object-oriented systems.

DCE-RPC, like SunRPC, can be implemented on top of several transport protocols including UDP and TCP. It is also similar to SunRPC in that it defines a two-level addressing scheme: the transport protocol demultiplexes to the correct server, DCE-RPC dispatches to a particular procedure exported by that server, and clients consult an "endpoint mapping service" (similar to SunRPC's Port Mapper) to learn how to reach a particular server. Unlike SunRPC, however, DCE-RPC implements at-most-once call semantics. (In truth, DCE-RPC supports multiple call semantics, including an idempotent semantics similar to SunRPC's, but at-most-once is the default behavior.) There are some other differences between the two approaches, which we will highlight in the following paragraphs.

Figure 5.19 gives a timeline for the typical exchange of messages, where each message is labeled by its DCE-RPC type. The client sends a `Request` message, the server eventually replies with a `Response` message, and the client acknowledges (`Ack`) the response. Instead of the server acknowledging the request

[1] In practice, NFS is such an important program that it has been given its own well-known UDP port, but for the purposes of illustration we're pretending that's not the case.

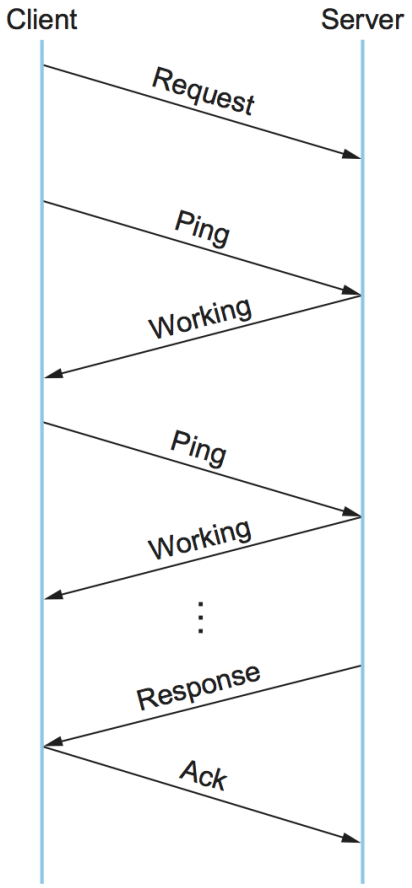


Figure 5.19.: Typical DCE-RPC message exchange.

messages, however, the client periodically sends a `Ping` message to the server, which responds with a `Working` message to indicate that the remote procedure is still in progress. If the server's reply is received reasonably quickly, no `Ping`s are sent. Although not shown in the figure, other message types are also supported. For example, the client can send a `Quit` message to the server, asking it to abort an earlier call that is still in progress; the server responds with a `Quack` (quit acknowledgment) message. Also, the server can respond to a `Request` message with a `Reject` message (indicating that a call has been rejected), and it can respond to a `Ping` message with a `Nocall` message (indicating that the server has never heard of the caller).

Each request/reply transaction in DCE-RPC takes place in the context of an *activity*. An activity is a logical request/reply channel between a pair of participants. At any given time, there can be only one message transaction active on a given channel. Like the concurrent logical channel approach described above, the application programs have to open multiple channels if they want to have more than one request/reply transaction between them at the same time. The activity to which a message belongs is identified by the message's `ActivityId` field. A `SequenceNum` field then distinguishes between calls made as part of the same activity; it serves the same purpose as SunRPC's `XID` (transaction id) field. Unlike SunRPC, DCE-RPC keeps track of the last sequence number used as part of a particular activity, so as to ensure at-most-once semantics. To distinguish between replies sent before and after a server machine reboots, DCE-RPC uses a `ServerBoot` field to hold the machine's boot ID.

Another design choice made in DCE-RPC that differs from SunRPC is the support of fragmentation and reassembly in the RPC protocol. As noted above, even if an underlying protocol such as IP provides fragmentation/reassembly, a more sophisticated algorithm implemented as part of RPC can result in quicker recovery and reduced bandwidth consumption when fragments are lost. The `FragmentNum` field uniquely identifies each fragment that makes up a given request or reply message. Each DCE-RPC fragment is assigned a unique fragment number (0, 1, 2, 3, and so on). Both the client and server implement a selective acknowledgment mechanism, which works as follows. (We describe the mechanism in terms of a client sending a fragmented request message to the server; the same mechanism applies when a server sends a fragment response to the client.)

First, each fragment that makes up the request message contains both a unique `FragmentNum` and a flag indicating whether this packet is a fragment of a call (`frag`) or the last fragment of a call (); request messages that fit in a single packet carry a flag. The server knows it has received the complete request message when it has the packet and there are no gaps in the fragment numbers. Second, in response to each arriving fragment, the server sends a `Fack` (fragment acknowledgment) message to the client. This acknowledgment identifies the highest fragment number that the server has successfully received. In other words, the acknowledgment is cumulative, much like in TCP. In addition, however, the server selectively acknowledges any higher fragment numbers it has received out of order. It does so with a bit vector that identifies these out-of-order fragments relative to the highest in-order fragment it has received. Finally, the client responds by retransmitting the missing fragments.

Figure 5.20 illustrates how this all works. Suppose the server has successfully received fragments up through number 20, plus fragments 23, 25, and 26. The server responds with a `Fack` that identifies fragment 20 as the highest in-order fragment, plus a bit-vector (`SelAck`) with the third (23=20+3), fifth (25=20+5), and sixth (26=20+6) bits turned on. So as to support an (almost) arbitrarily long bit vector, the size of the vector (measured in 32-bit words) is given in the `SelAckLen` field.

Given DCE-RPC's support for very large messages—the `FragmentNum` field is 16 bits long, meaning it can support 64K fragments—it is not appropriate for the protocol to blast all the fragments that make up a message as fast as it can since doing so might overrun the receiver. Instead, DCE-RPC implements a

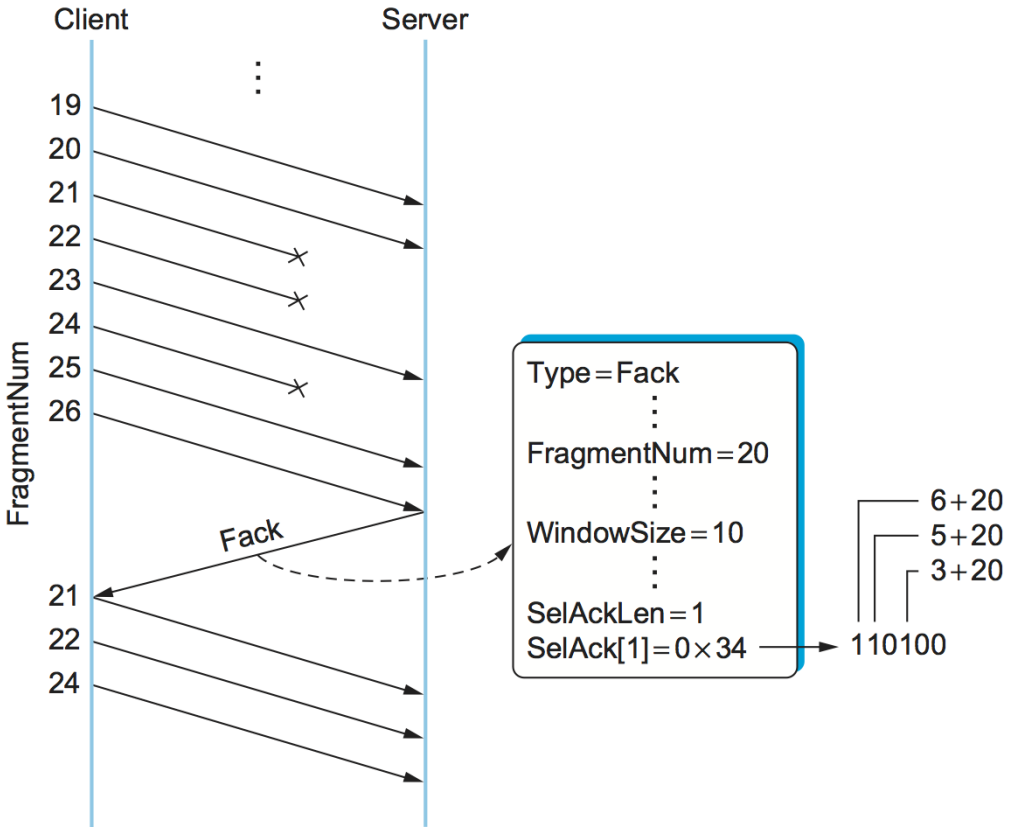


Figure 5.20.: Fragmentation with selective acknowledgments.

flow-control algorithm that is very similar to TCP's. Specifically, each `Fack` message not only acknowledges received fragments but also informs the sender of how many fragments it may now send. This is the purpose of the `WindowSize` field in Figure 5.20, which serves exactly the same purpose as TCP's `AdvertisedWindow` field except it counts fragments rather than bytes. DCE-RPC also implements a congestion-control mechanism that is similar to TCP's. Given the complexity of congestion control, it is perhaps not surprising that some RPC protocols avoid it by avoiding fragmentation.

In summary, designers have quite a range of options open to them when designing an RPC protocol. SunRPC takes the more minimalist approach and adds relatively little to the underlying transport beyond the essentials of locating the right procedure and identifying messages. DCE-RPC adds more functionality, with the possibility of improved performance in some environments at the cost of greater complexity.

gRPC

Despite its origins in Google, gRPC does not stand for Google RPC. The "g" stands for something different in each release. For version 1.10 it stood for "glamorous" and for 1.18 it stood for "goose". Googlers are wild and crazy people. Nonetheless, gRPC is popular because it makes available to everyone—as open source—a decade's worth of experience within Google using RPC to build scalable cloud services.

Before getting into the details, there are some major differences between gRPC and the other two examples we've just covered. The biggest is that gRPC is designed for cloud services rather than the simpler client/server paradigm that preceded it. The difference is essentially an extra level of indirection. In the client/server world, the client invokes a method on a specific server process running on a specific server machine. One server process is presumed to be enough to serve calls from all the client processes that might call it.

With cloud services, the client invokes a method on a *service*, which in order to support calls from arbitrarily many clients at the same time, is implemented by a scalable number of server processes, each potentially running on a different server machine. This is where the cloud comes into play: datacenters make a seemingly infinite number of server machines available to scale up cloud services. When we use the term "scalable" we mean that the number of identical server processes you elect to create depends on the workload (i.e., the number of clients that want service at any given time) and that number can be adjusted dynamically over time. One other detail is that cloud services don't typically create a new process, per se, but rather, they launch a new *container*, which is essentially a process encapsulated inside an isolated environment that includes all the software packages the process needs to run. Docker is today's canonical example of a container platform.

Back to the claim that a service is essentially an extra level of indirection layered on top of a server, all this means is that the caller identifies the service it wants to invoke, and a *load balancer* directs that invocation to one of the many available server processes (containers) that implement that service, as shown in Figure 5.21. The load balancer can be implemented in different ways, including a hardware device, but it is typically implemented by a proxy process that runs in a virtual machine (also hosted in the cloud) rather than as a physical appliance.

There is a set of best practices for implementing the actual server code that eventually responds to that request, and some additional cloud machinery to create/destroy containers and load balance requests across those containers. Kubernetes is today's canonical example of such a container management system, and the *micro-services architecture* is what we call the best practices in building services in this cloud native manner. Both are interesting topics, but beyond the scope of this book.

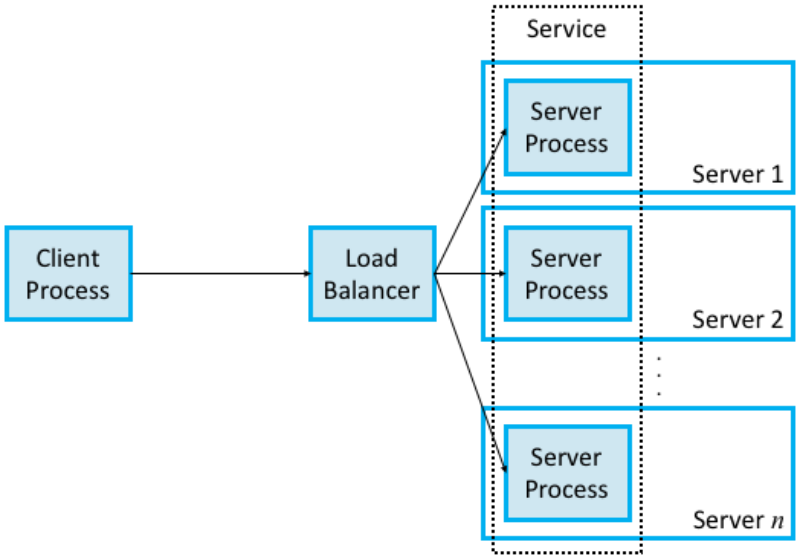


Figure 5.21.: Using RPC to invoke a scalable cloud service.

What we are interested in here is transport protocol at the core of gRPC. Here again, there is a major departure from the two previous example protocols, not in terms of fundamental problems that need to be addressed, but in terms of gRPC's approach to addressing them. In short, gRPC "outsources" many of the problems to other protocols, leaving gRPC to essentially package those capabilities in an easy-to-use form. Let's look at the details.

First, gRPC runs on top of TCP instead of UDP, which means it outsources the problems of connection management and reliably transmitting request and reply messages of arbitrary size. Second, gRPC actually runs on top of a secured version of TCP called *Transport Layer Security* (TLS)—a thin layer that sits above TCP in the protocol stack—which means it outsources responsibility for securing the communication channel so adversaries can't eavesdrop or hijack the message exchange. Third, gRPC actually, actually runs on top of HTTP/2 (which is itself layered on top of TCP and TLS), meaning gRPC outsources yet two other problems: (1) efficiently encoding/compressing binary data into a message, (2) multiplexing multiple remote procedure calls onto a single TCP connection. In other words, gRPC encodes the identifier for the remote method as a URI, the request parameters to the remote method as content in the HTTP message, and the return value from the remote method in the HTTP response. The full gRPC stack is depicted in Figure 5.22, which also includes the language-specific elements. (One strength of gRPC is the wide set of programming languages it supports, with only a small subset shown in Figure 5.22.)

We discuss TLS in Chapter 8 (in the context of a broad range of security topics) and HTTP in Chapter 9 (in the context of what are traditionally viewed as application level protocols). But we find ourselves in an interesting dependency loop: RPC is a flavor of transport protocol used to implement distributed applications, HTTP is an example of an application-level protocol, and yet gRPC runs on top of HTTP rather than the other way around.

The short explanation is that layering provides a convenient way for humans to wrap their heads around complex systems, but what we're really trying to do is solve a set of problem (e.g., reliably transfer messages of arbitrary size, identify senders and recipients, match requests messages with reply messages, and so on) and the way these solutions get bundled into protocols, and those protocols then layered on top of each other, is the consequence of incremental changes over time. You could argue it's an historical accident. Had the

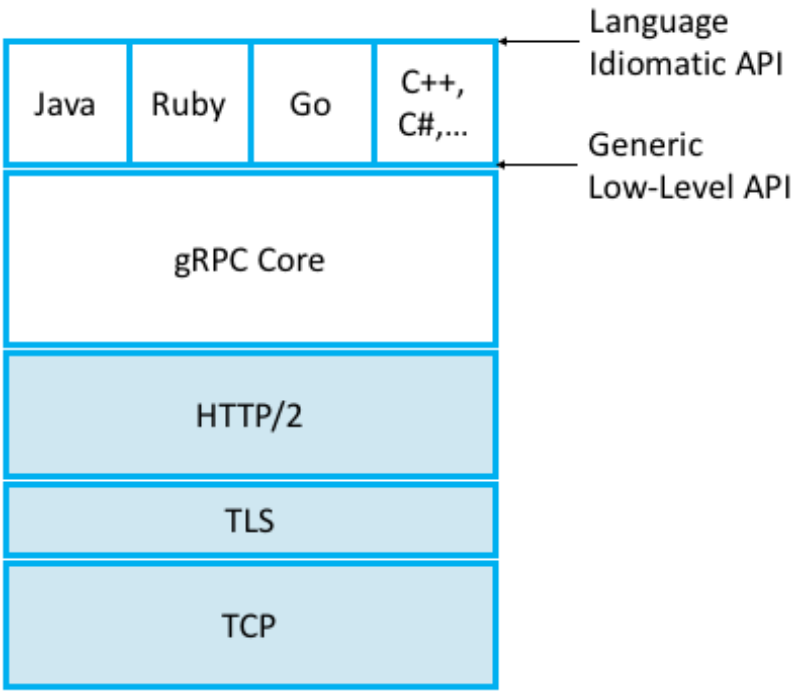


Figure 5.22.: gRPC core stacked on top of HTTP, TLS, and TCP and supporting a collection of languages.

Internet started with an RPC mechanism as ubiquitous as TCP, HTTP might have been implemented on top of it (as might have almost all of the other application-level protocols described in Chapter 9) and Google would have spent their time improving *that* protocol rather than inventing one of their own (as they and others have been doing with TCP). What happened instead is that the web became the Internet's killer app, which meant that its application protocol (HTTP) became universally supported by the rest of the Internet's infrastructure: Firewalls, Load Balancers, Encryption, Authentication, Compression, and so on. Because all of these network elements have been designed to work well with HTTP, HTTP has effectively become the Internet's universal request/reply transport protocol.

Returning to the unique characteristics of gRPC, the biggest value it brings to the table is to incorporate *streaming* into the RPC mechanism, which is to say, gRPC supports four different request/reply patterns:

1. Simple RPC: The client sends a single request message and the server responds with a single reply message.
2. Server Streaming RPC: The client sends a single request message and the server responds with a stream of reply messages. The client completes once it has all the server's responses.
3. Client Streaming RPC: The client sends a stream of requests to the server, and the server sends back a single response, typically (but not necessarily) after it has received all the client's requests.
4. Bidirectional Streaming RPC: The call is initiated by the client, but after that, the client and server can read and write requests and responses in any order; the streams are completely independent.

This extra freedom in how the client and server interact means the gRPC transport protocol needs to send additional metadata and control messages—in addition to the actual request and reply messages—between the two peers. Examples include `Error` and `Status` codes (to indicate success or why something failed), `Timeouts` (to indicate how long a client is willing to wait for a response), `PING` (a keep-alive notice to

indicate that one side or the other is still running), EOS (end-of-stream notice to indicate that there are no more requests or responses), and GOAWAY (a notice from servers to clients to indicate that they will no longer accept any new streams). Unlike many other protocols in this book, where we show the protocol's header format, the way this control information gets passed between the two sides is largely dictated by the underlying transport protocol, in this case HTTP/2. For example, as we'll see in Chapter 9, HTTP already includes a set of header fields and reply codes that gRPC takes advantage of.

You may want to peruse the HTTP discussion in Chapter 9 before continuing, but the following is fairly straightforward. A simple RPC request (with no streaming) might include the following HTTP message from the client to the server:

```
HEADERS (flags = END_HEADERS)
:method = POST
:scheme = http
:path = /google.pubsub.v2.PublisherService/CreateTopic
:authority = pubsub.googleapis.com
grpc-timeout = 1S
content-type = application/grpc+proto
grpc-encoding = gzip
authorization = Bearer y235.wef315yfh138vh31hv93hv8h3v
DATA (flags = END_STREAM)
<Length-Prefixed Message>
```

leading to the following response message from the server back to the client:

```
HEADERS (flags = END_HEADERS)
:status = 200
grpc-encoding = gzip
content-type = application/grpc+proto
DATA
<Length-Prefixed Message>
HEADERS (flags = END_STREAM, END_HEADERS)
grpc-status = 0 # OK
trace-proto-bin = jher831yy13JHy3hc
```

In this example, HEADERS and DATA are two standard HTTP control messages, which effectively delineate between "the message's header" and "the message's payload." Specifically, each line following HEADERS (but before DATA) is an attribute = value pair that makes up the header (think of each line as analogous to a header field); those pairs that start with colon (e.g., :status = 200) are part of the HTTP standard (e.g., status 200 indicates success); and those pairs that do not start with a colon are gRPC-specific customizations (e.g., grpc-encoding = gzip indicates that the data in the message that follows has been compressed using gzip, and grpc-timeout = 1S indicates that the client has set a one second timeout).

There is one final piece to explain. The header line

```
content-type = application/grpc+proto
```

indicates that the message body (as demarcated by the DATA line) is meaningful only to the application program (i.e., the server method) that this client is requesting service from. More specifically, the +proto string specifies that the recipient will be able to interpret the bits in the message according to a *Protocol Buffer* (abbreviated proto) interface specification. Protocol Buffers are gRPC's way of specifying how the

parameters being passed to the server are encoded into a message, which is in turn used to generate the stubs that sit between the underlying RPC mechanism and the actual functions being called (see Figure 5.14). This is a topic we'll take up in Chapter 7.

Key Takeaway

The bottom line is that complex mechanisms like RPC, once packaged as a monolithic bundle of software (as with SunRPC and DCE-RPC), is nowadays built by assembling an assortment of smaller pieces, each of which solves a narrow problem. gRPC is both an example of that approach, and a tool that enables further adoption of the approach. The micro-services architecture mentioned earlier in this subsection applies the "built from small parts" strategy to entire cloud applications (e.g., Uber, Lyft, Netflix, Yelp, Spotify), where gRPC is often the communication mechanism used by those small pieces to exchange messages with each other. *[Next]*

5.4 Transport for Real-Time (RTP)

In the early days of packet switching, most applications were concerned with transfering files, although as early as 1981, experiments were under way to carry real-time traffic, such as digitized voice samples. We call an application "real-time" when it has strong requirements for the timely delivery of information. Voice over IP (VoIP) is a classic example of a real-time application because you can't easily carry on a conversation with someone if it takes more than a fraction of a second to get a response. As we will see shortly, real-time applications place some specific demands on the transport protocol that are not well met by the protocols discussed so far in this chapter.

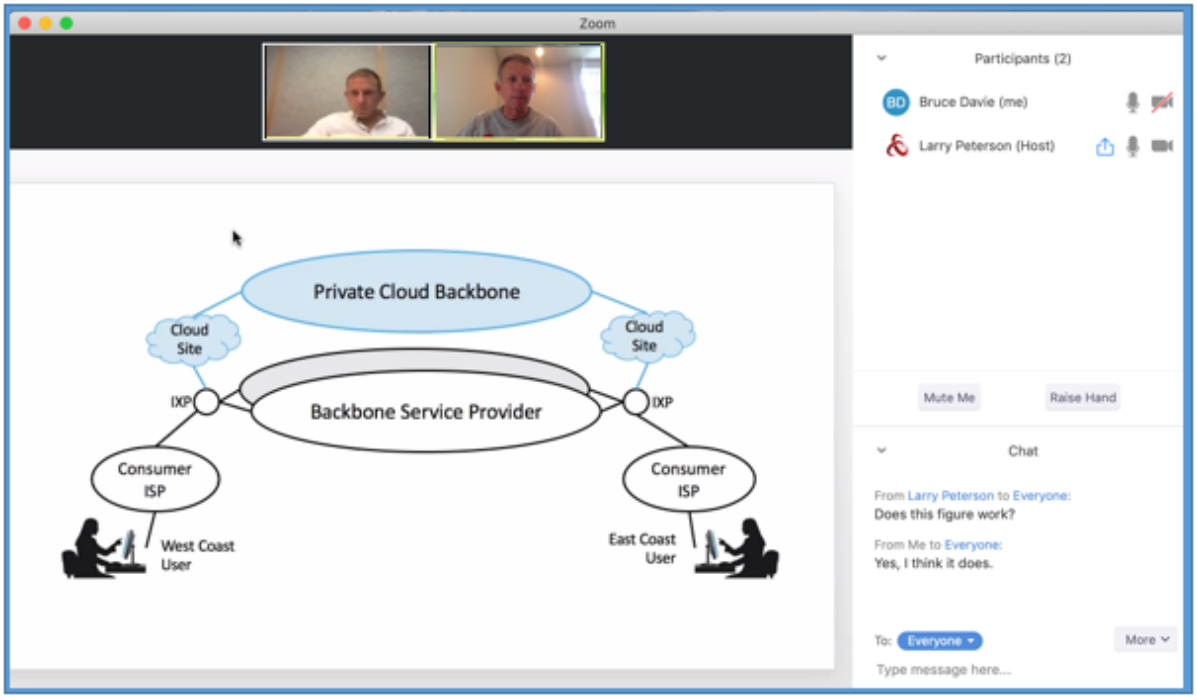


Figure 5.23.: User interface of a videoconferencing tool.

Multimedia applications—those that involve video, audio, and data—are sometimes divided into two classes: *interactive* applications and *streaming* applications. Figure 5.23 shows the authors using an example conferencing tool that's typical of the interactive class. Along with VoIP, these are the sort of applications with the most stringent real-time requirements.

Streaming applications typically deliver audio or video streams from a server to a client and are typified by such commercial products as Spotify. Streaming video, typified by YouTube and Netflix, has become one of the dominant forms of traffic on the Internet. Because streaming applications lack human-to-human interaction, they place somewhat less stringent real-time requirements on the underlying protocols. Timeliness is still important, however—for example, you want a video to start playing soon after pushing "play," and once it starts to play, late packets will either cause it to stall or create some sort of visual degradation. So, while streaming applications are not strictly real time, they still have enough in common with interactive multimedia applications to warrant consideration of a common protocol for both types of application.

It should by now be apparent that designers of a transport protocol for real-time and multimedia applications face a real challenge in defining the requirements broadly enough to meet the needs of very different applications. They must also pay attention to the interactions among different applications, such as the synchronization of audio and video streams. We will see below how these concerns affected the design of the primary real-time transport protocol in use today: *Real-time Transport Protocol* (RTP).

Much of RTP actually derives from protocol functionality that was originally embedded in the application itself. Two of the first such applications were `vic` and `vat`, the former supporting real-time video and the latter supporting real-time audio. Both applications originally ran directly over UDP, while the designers figured out which features were needed to handle the real-time nature of the communication. Later, they realized that these features could be useful to many other applications and defined a protocol with those features. That protocol was eventually standardized as RTP.

RTP can run over many lower-layer protocols, but still commonly runs over UDP. That leads to the protocol stack shown in Figure 5.24. Note that we are therefore running a transport protocol over a transport protocol. There is no rule against that, and in fact it makes a lot of sense, since UDP provides such a minimal level of functionality, and the basic demultiplexing based on port numbers happens to be just what RTP needs as a starting point. So, rather than recreate port numbers in RTP, RTP outsources the demultiplexing function to UDP.

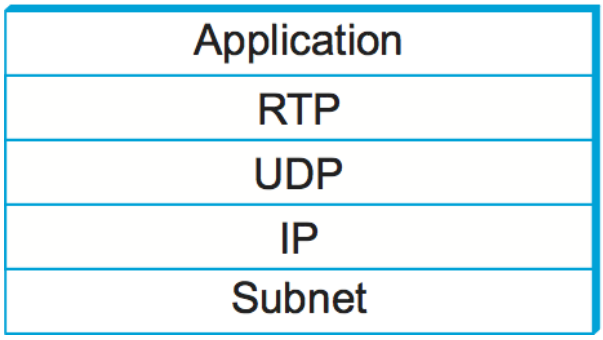


Figure 5.24.: Protocol stack for multimedia applications using RTP.

5.4.1 Requirements

The most basic requirement for a general-purpose multimedia protocol is that it allows similar applications to interoperate with each other. For example, it should be possible for two independently implemented audioconferencing applications to talk to each other. This immediately suggests that the applications had better use the same method of encoding and compressing voice; otherwise, the data sent by one party will be incomprehensible to the receiving party. Since there are quite a few different coding schemes for voice, each with its own trade-offs among quality, bandwidth requirements, and computational cost, it would probably be a bad idea to decree that only one such scheme can be used. Instead, our protocol should provide a way that a sender can tell a receiver which coding scheme it wants to use, and possibly negotiate until a scheme that is available to both parties is identified.

Just as with audio, there are many different video coding schemes. Thus, we see that the first common function that RTP can provide is the ability to communicate that choice of coding scheme. Note that this also serves to identify the type of application (e.g., audio or video); once we know what coding algorithm is being used, we know what type of data is being encoded as well.

Another important requirement is to enable the recipient of a data stream to determine the timing relationship among the received data. Real-time applications need to place received data into a *playback buffer* to smooth out the jitter that may have been introduced into the data stream during transmission across the network. Thus, some sort of timestamping of the data will be necessary to enable the receiver to play it back at the appropriate time.

Related to the timing of a single media stream is the issue of synchronization of multiple media in a conference. The obvious example of this would be to synchronize an audio and video stream that are originating from the same sender. As we will see below, this is a slightly more complex problem than playback time determination for a single stream.

Another important function to be provided is an indication of packet loss. Note that an application with tight latency bounds generally cannot use a reliable transport like TCP because retransmission of data to correct for loss would probably cause the packet to arrive too late to be useful. Thus, the application must be able to deal with missing packets, and the first step in dealing with them is noticing that they are in fact missing. As an example, a video application using MPEG encoding may take different actions when a packet is lost, depending on whether the packet came from an I frame, a B frame, or a P frame.

Packet loss is also a potential indicator of congestion. Since multimedia applications generally do not run over TCP, they also miss out on the congestion avoidance features of TCP. Yet, many multimedia applications are capable of responding to congestion—for example, by changing the parameters of the coding algorithm to reduce the bandwidth consumed. Clearly, to make this work, the receiver needs to notify the sender that losses are occurring so that the sender can adjust its coding parameters.

Another common function across multimedia applications is the concept of frame boundary indication. A frame in this context is application specific. For example, it may be helpful to notify a video application that a certain set of packets correspond to a single frame. In an audio application it is helpful to mark the beginning of a "talkspurt," which is a collection of sounds or words followed by silence. The receiver can then identify the silences between talkspurts and use them as opportunities to move the playback point. This follows the observation that slight shortening or lengthening of the spaces between words are not perceptible to users, whereas shortening or lengthening the words themselves is both perceptible and annoying.

A final function that we might want to put into the protocol is some way of identifying senders that is more user-friendly than an IP address. As illustrated in Figure 5.23, audio and video conferencing applications

can display strings such as on their control panels, and thus the application protocol should support the association of such a string with a data stream.

In addition to the functionality that is required from our protocol, we note an additional requirement: It should make reasonably efficient use of bandwidth. Put another way, we don't want to introduce a lot of extra bits that need to be sent with every packet in the form of a long header. The reason for this is that audio packets, which are one of the most common types of multimedia data, tend to be small, so as to reduce the time it takes to fill them with samples. Long audio packets would mean high latency due to packetization, which has a negative effect on the perceived quality of conversations. (This was one of the factors in choosing the length of ATM cells.) Since the data packets themselves are short, a large header would mean that a relatively large amount of link bandwidth would be used by headers, thus reducing the available capacity for "useful" data. We will see several aspects of the design of RTP that have been influenced by the necessity of keeping the header short.

You could argue whether every single feature just described *really* needs to be in a real-time transport protocol, and you could probably find some more that could be added. The key idea here is to make life easier for application developers by giving them a useful set of abstractions and building blocks for their applications. For example, by putting a timestamping mechanism into RTP, we save every developer of a real-time application from inventing his own. We also increase the chances that two different real-time applications might interoperate.

5.4.2 RTP Design

Now that we have seen the rather long list of requirements for our transport protocol for multimedia, we turn to the details of the protocol that has been specified to meet those requirements. This protocol, RTP, was developed in the IETF and is in widespread use. The RTP standard actually defines a pair of protocols, RTP and the Real-time Transport Control Protocol (RTCP). The former is used for the exchange of multimedia data, while the latter is used to periodically send control information associated with a certain data flow. When running over UDP, the RTP data stream and the associated RTCP control stream use consecutive transport-layer ports. The RTP data uses an even port number and the RTCP control information uses the next higher (odd) port number.

Because RTP is designed to support a wide variety of applications, it provides a flexible mechanism by which new applications can be developed without repeatedly revising the RTP protocol itself. For each class of application (e.g., audio), RTP defines a *profile* and one or more *formats*. The profile provides a range of information that ensures a common understanding of the fields in the RTP header for that application class, as will be apparent when we examine the header in detail. The format specification explains how the data that follows the RTP header is to be interpreted. For example, the RTP header might just be followed by a sequence of bytes, each of which represents a single audio sample taken a defined interval after the previous one. Alternatively, the format of the data might be much more complex; an MPEG-encoded video stream, for example, would need to have a good deal of structure to represent all the different types of information.

Key Takeaway

The design of RTP embodies an architectural principle known as *Application Level Framing* (ALF). This principle was put forward by Clark and Tennenhouse in 1990 as a new way to design protocols for emerging multimedia applications. They recognized that these new applications were unlikely to be well served by existing protocols such as TCP, and that furthermore they might not be well served by any sort of "one-size-

fits-all" protocol. At the heart of this principle is the belief that an application understands its own needs best. For example, an MPEG video application knows how best to recover from lost frames and how to react differently if an I frame or a B frame is lost. The same application also understands best how to segment the data for transmission—for example, it's better to send the data from different frames in different datagrams, so that a lost packet only corrupts a single frame, not two. It is for this reason that RTP leaves so many of the protocol details to the profile and format documents that are specific to an application. *[Next]*

Header Format

Figure 5.25 shows the header format used by RTP. The first 12 bytes are always present, whereas the contributing source identifiers are only used in certain circumstances. After this header there may be optional header extensions, as described below. Finally, the header is followed by the RTP payload, the format of which is determined by the application. The intention of this header is that it contain only the fields that are likely to be used by many different applications, since anything that is very specific to a single application would be more efficiently carried in the RTP payload for that application only.

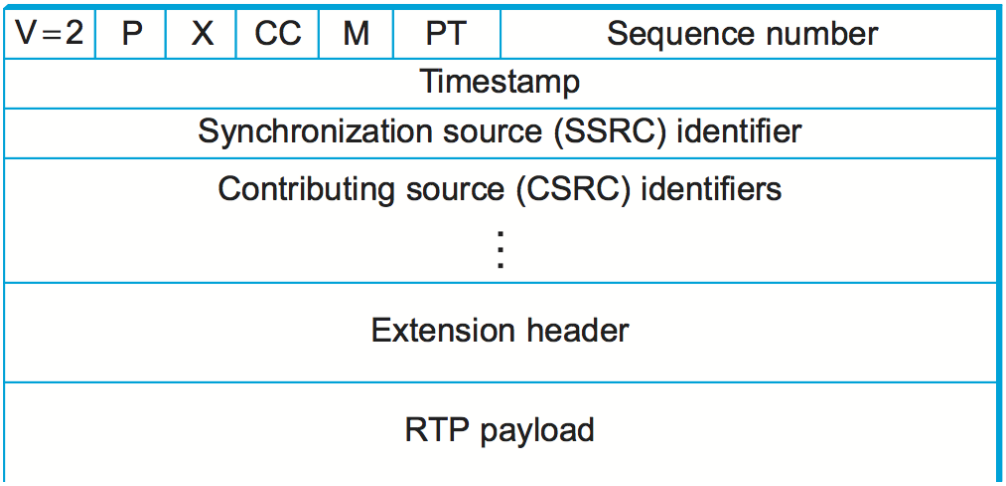


Figure 5.25.: RTP header format.

The first two bits are a version identifier, which contains the value 2 in the RTP version deployed at the time of writing. You might think that the designers of the protocol were rather bold to think that 2 bits would be enough to contain all future versions of RTP, but recall that bits are at a premium in the RTP header. Furthermore, the use of profiles for different applications makes it less likely that many revisions to the base RTP protocol would be needed. In any case, if it turns out that another version of RTP is needed beyond version 2, it would be possible to consider a change to the header format so that more than one future version would be possible. For example, a new RTP header with the value 3 in the version field could have a "subversion" field somewhere else in the header.

The next bit is the *padding* (`P`) bit, which is set in circumstances in which the RTP payload has been padded for some reason. RTP data might be padded to fill up a block of a certain size as required by an encryption algorithm, for example. In such a case, the complete length of the RTP header, data, and padding would be conveyed by the lower-layer protocol header (e.g., the UDP header), and the last byte of the padding would contain a count of how many bytes should be ignored. This is illustrated in Figure 5.26. Note that this

approach to padding removes any need for a length field in the RTP header (thus serving the goal of keeping the header short); in the common case of no padding, the length is deduced from the lower-layer protocol.

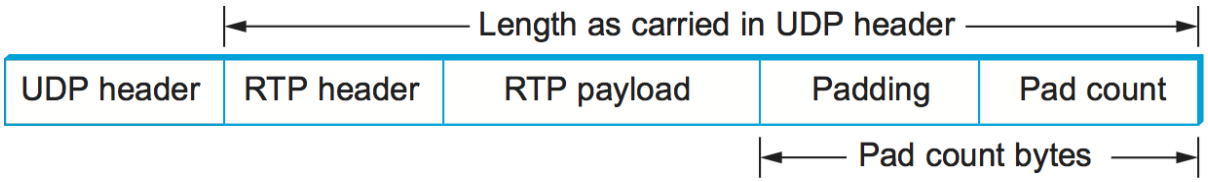


Figure 5.26.: Padding of an RTP packet.

The *extension* (X) bit is used to indicate the presence of an extension header, which would be defined for a specific application and follow the main header. Such headers are rarely used, since it is generally possible to define a payload-specific header as part of the payload format definition for a particular application.

The X bit is followed by a 4-bit field that counts the number of *contributing sources*, if any are included in the header. Contributing sources are discussed below.

We noted above the frequent need for some sort of frame indication; this is provided by the marker bit, which has a profile-specific use. For a voice application, it could be set at the beginning of a talkspurt, for example. The 7-bit payload type field follows; it indicates what type of multimedia data is carried in this packet. One possible use of this field would be to enable an application to switch from one coding scheme to another based on information about resource availability in the network or feedback on application quality. The exact usage of the payload type is also determined by the application profile.

Note that the payload type is generally not used as a demultiplexing key to direct data to different applications (or to different streams within a single application, such as the audio and video stream for a videoconference). This is because such demultiplexing is typically provided at a lower layer (e.g., by UDP, as described in a previous section). Thus, two media streams using RTP would typically use different UDP port numbers.

The sequence number is used to enable the receiver of an RTP stream to detect missing and misordered packets. The sender simply increments the value by one for each transmitted packet. Note that RTP does not do anything when it detects a lost packet, in contrast to TCP, which both corrects for the loss (by retransmission) and interprets the loss as a congestion indication (which may cause it to reduce its window size). Rather, it is left to the application to decide what to do when a packet is lost because this decision is likely to be highly application dependent. For example, a video application might decide that the best thing to do when a packet is lost is to replay the last frame that was correctly received. Some applications might also decide to modify their coding algorithms to reduce bandwidth needs in response to loss, but this is not a function of RTP. It would not be sensible for RTP to decide that the sending rate should be reduced, as this might make the application useless.

The function of the timestamp field is to enable the receiver to play back samples at the appropriate intervals and to enable different media streams to be synchronized. Because different applications may require different granularities of timing, RTP itself does not specify the units in which time is measured. Instead, the timestamp is just a counter of "ticks," where the time between ticks is dependent on the encoding in use. For example, an audio application that samples data once every 125 μs could use that value as its clock resolution. The clock granularity is one of the details that is specified in the RTP profile or payload format for an application.

The timestamp value in the packet is a number representing the time at which the *first* sample in the packet

was generated. The timestamp is not a reflection of the time of day; only the differences between timestamps are relevant. For example, if the sampling interval is 125 μs and the first sample in packet n+1 was generated 10 ms after the first sample in packet n, then the number of sampling instants between these two samples is

$$\text{TimeBetweenPackets / TimePerSample}$$

$$= (10 \times 10^{-3})/(125 \times 10^{-6}) = 80$$

Assuming the clock granularity is the same as the sampling interval, then the timestamp in packet n+1 would be greater than that in packet n by 80. Note that fewer than 80 samples might have been sent due to compression techniques such as silence detection, and yet the timestamp allows the receiver to play back the samples with the correct temporal relationship.

The synchronization source (SSRC) is a 32-bit number that uniquely identifies a single source of an RTP stream. In a given multimedia conference, each sender picks a random SSRC and is expected to resolve conflicts in the unlikely event that two sources pick the same value. By making the source identifier something other than the network or transport address of the source, RTP ensures independence from the lower-layer protocol. It also enables a single node with multiple sources (e.g., several cameras) to distinguish those sources. When a single node generates different media streams (e.g., audio and video), it is not required to use the same SSRC in each stream, as there are mechanisms in RTCP (described below) to allow intermedia synchronization.

The contributing source (CSRC) is used only when a number of RTP streams pass through a mixer. A mixer can be used to reduce the bandwidth requirements for a conference by receiving data from many sources and sending it as a single stream. For example, the audio streams from several concurrent speakers could be decoded and recoded as a single audio stream. In this case, the mixer lists itself as the synchronization source but also lists the contributing sources—the SSRC values of the speakers who contributed to the packet in question.

5.4.3 Control Protocol

RTCP provides a control stream that is associated with a data stream for a multimedia application. This control stream provides three main functions:

1. Feedback on the performance of the application and the network
2. A way to correlate and synchronize different media streams that have come from the same sender
3. A way to convey the identity of a sender for display on a user interface.

The first function may be useful for detecting and responding to congestion. Some applications are able to operate at different rates and may use performance data to decide to use a more aggressive compression scheme to reduce congestion, for example, or to send a higher-quality stream when there is little congestion. Performance feedback can also be useful in diagnosing network problems.

You might think that the second function is already provided by the synchronization source ID (SSRC) of RTP, but in fact it is not. As already noted, multiple cameras from a single node might have different SSRC values. Furthermore, there is no requirement that an audio and video stream from the same node use the same SSRC. Because collisions of SSRC values may occur, it may be necessary to change the SSRC value of a stream. To deal with this problem, RTCP uses the concept of a *canonical name* (CNAME) that is assigned to a sender, which is then associated with the various SSRC values that might be used by that sender using RTCP mechanisms.

Simply correlating two streams is only part of the problem of intermedia synchronization. Because different streams may have completely different clocks (with different granularities and even different amounts of inaccuracy, or drift), there needs to be a way to accurately synchronize streams with each other. RTCP addresses this problem by conveying timing information that correlates actual time of day with the clock-rate-dependent timestamps that are carried in RTP data packets.

RTCP defines a number of different packet types, including

- Sender reports, which enable active senders to a session to report transmission and reception statistics
- Receiver reports, which receivers who are not senders use to report reception statistics
- Source descriptions, which carry CNAMEs and other sender description information
- Application-specific control packets

These different RTCP packet types are sent over the lower-layer protocol, which, as we have noted, is typically UDP. Several RTCP packets can be packed into a single PDU of the lower-level protocol. It is required that at least two RTCP packets are sent in every lower-level PDU: One of these is a report packet; the other is a source description packet. Other packets may be included up to the size limits imposed by the lower-layer protocols.

Before looking further at the contents of an RTCP packet, we note that there is a potential problem with every member of a multicast group sending periodic control traffic. Unless we take some steps to limit it, this control traffic has the potential to be a significant consumer of bandwidth. In an audioconference, for example, no more than two or three senders are likely to send audio data at any instant, since there is no point in everyone talking at once. But there is no such social limit on everyone sending control traffic, and this could be a severe problem in a conference with thousands of participants. To deal with this problem, RTCP has a set of mechanisms by which the participants scale back their reporting frequency as the number of participants increases. These rules are somewhat complex, but the basic goal is this: Limit the total amount of RTCP traffic to a small percentage (typically 5%) of the RTP data traffic. To accomplish this goal, the participants should know how much data bandwidth is likely to be in use (e.g., the amount to send three audio streams) and the number of participants. They learn the former from means outside RTP (known as *session management*, discussed at the end of this section), and they learn the latter from the RTCP reports of other participants. Because RTCP reports might be sent at a very low rate, it might only be possible to get an approximate count of the current number of recipients, but that is typically sufficient. Also, it is recommended to allocate more RTCP bandwidth to active senders, on the assumption that most participants would like to see reports from them—for example, to find out who is speaking.

Once a participant has determined how much bandwidth it can consume with RTCP traffic, it sets about sending periodic reports at the appropriate rate. Sender reports and receiver reports differ only in that the former include some extra information about the sender. Both types of reports contain information about the data that was received from all sources in the most recent reporting period.

The extra information in a sender report consists of

- A timestamp containing the actual time of day when this report was generated
- The RTP timestamp corresponding to the time when the report was generated
- Cumulative counts of the packets and bytes sent by this sender since it began transmission

Note that the first two quantities can be used to enable synchronization of different media streams from the same source, even if those streams use different clock granularities in their RTP data streams, since it gives

the key to convert time of day to the RTP timestamps.

Both sender and receiver reports contain one block of data per source that has been heard from since the last report. Each block contains the following statistics for the source in question:

- Its SSRC
- The fraction of data packets from this source that were lost since the last report was sent (calculated by comparing the number of packets received with the number of packets expected; this last value can be determined from the RTP sequence numbers)
- Total number of packets lost from this source since the first time it was heard from
- Highest sequence number received from this source (extended to 32 bits to account for wrapping of the sequence number)
- Estimated interarrival jitter for the source (calculated by comparing the interarrival spacing of received packets with the expected spacing at transmission time)
- Last actual timestamp received via RTCP for this source
- Delay since last sender report received via RTCP for this source

As you might imagine, the recipients of this information can learn all sorts of things about the state of the session. In particular, they can see if other recipients are getting much better quality from some sender than they are, which might be an indication that a resource reservation needs to be made, or that there is a problem in the network that needs to be attended to. In addition, if a sender notices that many receivers are experiencing high loss of its packets, it might decide that it should reduce its sending rate or use a coding scheme that is more resilient to loss.

The final aspect of RTCP that we will consider is the source description packet. Such a packet contains, at a minimum, the SSRC of the sender and the sender's CNAME. The canonical name is derived in such a way that all applications that generate media streams that might need to be synchronized (e.g., separately generated audio and video streams from the same user) will choose the same CNAME even though they might choose different SSRC values. This enables a receiver to identify the media stream that came from the same sender. The most common format of the CNAME is , where `host` is the fully qualified domain name of the sending machine. Thus, an application launched by the user whose user name is `jdoe` running on the machine would use the string as its CNAME. The large and variable number of bytes used in this representation would make it a bad choice for the format of an SSRC, since the SSRC is sent with every data packet and must be processed in real time. Allowing CNAMEs to be bound to SSRC values in periodic RTCP messages enables a compact and efficient format for the SSRC.

Other items may be included in the source description packet, such as the real name and email address of the user. These are used in user interface displays and to contact participants, but are less essential to the operation of RTP than the CNAME.

Like TCP, RTP and RTCP are a fairly complex pair of protocols. This complexity comes in large part from the desire to make life easier for application designers. Because there is an infinite number of possible applications, the challenge in designing a transport protocol is to make it general enough to meet the widely varying needs of many different applications without making the protocol itself impossible to implement. RTP has proven very successful in this regard, forming the basis for many real-time multimedia applications run over the Internet today.

Perspective: HTTP is the New Narrow Waist

The Internet has been described as having a *narrow waist* architecture, with one universal protocol in the middle (IP), widening to support many transport and application protocols above it (e.g., TCP, UDP, RTP, SunRPC, DCE-RPC, gRPC, SMTP, HTTP, SNMP) and able to run on top of many network technologies below (e.g., Ethernet, PPP, WiFi, SONET, ATM). This general structure has been a key to the Internet becoming ubiquitous: by keeping the IP layer that everyone has to agree to minimal, a thousand flowers were allowed to bloom both above and below. This is now a widely understood strategy for any platform trying to achieve universal adoption.

But something else has happened over the last 30 years. By not addressing all the issues the Internet would eventually face as it grew (e.g., security, congestion, mobility, real-time responsiveness, and so on) it became necessary to introduce a series of additional features into the Internet architecture. Having IP's universal addresses and best-effort service model was a necessary condition for adoption, but not a sufficient foundation for all the applications people wanted to build.

We're yet to see some of these solutions—future chapters will describe how the Internet manages congestion (*Chapter 6*), provides security (*Chapter 8*), and supports real-time multimedia applications (*Chapters 7* and *9*)—but it is informative to take this opportunity to reconcile the value of a universal narrow waist with the evolution that inevitably happens in any long-lived system: the "fixed point" around which the rest of the architecture evolves has moved to a new spot in the software stack. In short, HTTP has become the new narrow waist; the one shared/assumed piece of the global infrastructure that makes everything else possible. This didn't happen overnight or by proclamation, although some did anticipate it would happen. The narrow waist drifted slowly up the protocol stack as a consequence of an evolution (to mix geoscience and biological metaphors).

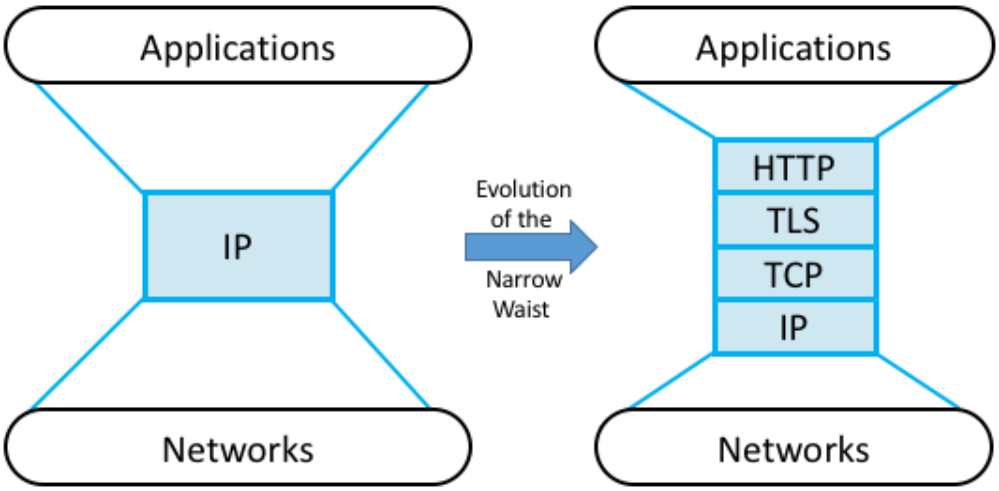


Figure 5.27.: HTTP (plus TLS, TCP, and IP) forming the narrow waist of today's Internet architecture.

Putting the narrow waist label purely on HTTP is an over simplification. It's actually a team effort, with the HTTP/TLS/TCP/IP combination now serving as the Internet's common platform.

- HTTP provides global object identifiers (URIs) and a simple GET/PUT interface.
- TLS provides end-to-end communication security.
- TCP provides connection management, reliable transmission, and congestion control.

- IP provides global host addresses and a network abstraction layer.

In other words, even though you are free to invent your own congestion control algorithm, TCP solves this problem quite well, so it makes sense to reuse that solution. Similarly, even though you are free to invent your own RPC protocol, HTTP provides a perfectly serviceable one (which because it comes bundled with proven security, has the added feature of not being blocked by enterprise firewalls), so again, it makes sense to reuse it rather than reinvent the wheel.

Somewhat less obviously, HTTP also provides a good foundation for dealing with mobility. If the resource you want to access has moved, you can have HTTP return a *redirect response* that points the client to a new location. Similarly, HTTP enables injecting *caching proxies* between the client and server, making it possible to replicate popular content in multiple locations and save clients the delay of going all the way across the Internet to retrieve some piece of information. (Both of these capabilities are discussed in *Section 9.1*.) Finally, HTTP has been used to deliver real-time multimedia, in an approach known as *adaptive streaming*. (See how in *Section 7.2*.)

Broader Perspective

To continue reading about the cloudification of the Internet, see *Perspective: Software-Defined Traffic Engineering*.

To learn more about the centrality of HTTP, we recommend: HTTP: An Evolvable Narrow Waist for the Future Internet, January 2012.

CHAPTER

SIX

CONGESTION CONTROL

The hand that hath made you fair hath made you good.

—*William Shakespeare*

Problem: Allocating Resources

By now we have seen enough layers of the network protocol hierarchy to understand how data can be transferred among processes across heterogeneous networks. We now turn to a problem that spans the entire protocol stack—how to effectively and fairly allocate resources among a collection of competing users. The resources being shared include the bandwidth of the links and the buffers on the routers or switches where packets are queued awaiting transmission. Packets *contend* at a router for the use of a link, with each contending packet placed in a queue waiting its turn to be transmitted over the link. When too many packets are contending for the same link, the queue fills and two undesirable things happen: packets experience increased end-to-end delay, and in the worst case, the queue overflows and packets have to be dropped. When long queues persist and drops become common, the network is said to be *congested*. Most networks provide a *congestion-control* mechanism to deal with just such a situation.

Congestion control and resource allocation are two sides of the same coin. On the one hand, if the network takes an active role in allocating resources—for example, scheduling which virtual circuit gets to use a given physical link during a certain period of time—then congestion may be avoided, thereby making congestion control unnecessary. Allocating network resources with any precision is difficult, however, because the resources in question are distributed throughout the network; multiple links connecting a series of routers need to be scheduled. On the other hand, you can always let packet sources send as much data as they want and then recover from congestion should it occur. This is the easier approach, but it can be disruptive because many packets may be discarded by the network before congestion can be controlled. Furthermore, it is precisely at those times when the network is congested—that is, resources have become scarce relative to demand—that the need for resource allocation among competing users is most keenly felt. There are also solutions in the middle, whereby inexact allocation decisions are made, but congestion can still occur and hence some mechanism is still needed to recover from it. Whether you call such a mixed solution congestion control or resource allocation does not really matter. In some sense, it is both.

Congestion control and resource allocation involve both hosts and network elements such as routers. In network elements, various queuing disciplines can be used to control the order in which packets get transmitted and which packets get dropped. The queuing discipline can also segregate traffic to keep one user's packets from unduly affecting another user's packets. At the end hosts, the congestion-control mechanism paces

how fast sources are allowed to send packets. This is done in an effort to keep congestion from occurring in the first place and, should it occur, to help eliminate the congestion.

This chapter starts with an overview of congestion control and resource allocation. We then discuss different queuing disciplines that can be implemented on the routers inside the network, followed by a description of the congestion-control algorithm provided by TCP on the hosts. The fourth section explores various techniques involving both routers and hosts that aim to avoid congestion before it becomes a problem. Finally, we examine the broad area of *quality of service*. We consider the needs of applications to receive different levels of resource allocation in the network and describe a number of ways in which they can request these resources and the network can meet the requests.

6.1 Issues in Resource Allocation

Resource allocation and congestion control are complex issues that have been the subject of much study ever since the first network was designed. They are still active areas of research. One factor that makes these issues complex is that they are not isolated to one single level of a protocol hierarchy. Resource allocation is partially implemented in the routers, switches, and links inside the network and partially in the transport protocol running on the end hosts. End systems may use signalling protocols to convey their resource requirements to network nodes, which respond with information about resource availability. One of the main goals of this chapter is to define a framework in which these mechanisms can be understood, as well as to give the relevant details about a representative sample of mechanisms.

We should clarify our terminology before going any further. By *resource allocation*, we mean the process by which network elements try to meet the competing demands that applications have for network resources—primarily link bandwidth and buffer space in routers or switches. Of course, it will often not be possible to meet all the demands, meaning that some users or applications may receive fewer network resources than they want. Part of the resource allocation problem is deciding when to say no and to whom.

We use the term *congestion control* to describe the efforts made by network nodes to prevent or respond to overload conditions. Since congestion is generally bad for everyone, the first order of business is making congestion subside, or preventing it in the first place. This might be achieved simply by persuading a few hosts to stop sending, thus improving the situation for everyone else. However, it is more common for congestion-control mechanisms to have some aspect of fairness—that is, they try to share the pain among all users, rather than causing great pain to a few. Thus, we see that many congestion-control mechanisms have some sort of resource allocation built into them.

It is also important to understand the difference between flow control and congestion control. Flow control involves keeping a fast sender from overrunning a slow receiver. Congestion control, by contrast, is intended to keep a set of senders from sending too much data *into the network* because of lack of resources at some point. These two concepts are often confused; as we will see, they also share some mechanisms.

6.1.1 Network Model

We begin by defining three salient features of the network architecture. For the most part, this is a summary of material presented in the previous chapters that is relevant to the problem of resource allocation.

Packet-Switched Network

We consider resource allocation in a packet-switched network (or internet) consisting of multiple links and switches (or routers). Since most of the mechanisms described in this chapter were designed for use on the Internet, and therefore were originally defined in terms of routers rather than switches, we use the term *router* throughout our discussion. The problem is essentially the same, whether on a network or an internetwork.

In such an environment, a given source may have more than enough capacity on the immediate outgoing link to send a packet, but somewhere in the middle of a network its packets encounter a link that is being used by many different traffic sources. Figure 6.1 illustrates this situation—two high-speed links are feeding a low-speed link. This is in contrast to shared-access networks like Ethernet and wireless networks, where the source can directly observe the traffic on the network and decide accordingly whether or not to send a packet. We have already seen the algorithms used to allocate bandwidth on shared-access networks (e.g., Ethernet and Wi-Fi). These access-control algorithms are, in some sense, analogous to congestion-control algorithms in a switched network.

Key Takeaway

Note that congestion control is a different problem than routing. While it is true that a congested link could be assigned a large edge weight by the routing protocol, and, as a consequence, routers would route around it, "routing around" a congested link does not generally solve the congestion problem. To see this, we need look no further than the simple network depicted in Figure 6.1, where all traffic has to flow through the same router to reach the destination. Although this is an extreme example, it is common to have a certain router that it is not possible to route around. This router can become congested, and there is nothing the routing mechanism can do about it. This congested router is sometimes called the *bottleneck* router. *[Next]*

Connectionless Flows

For much of our discussion, we assume that the network is essentially connectionless, with any connection-oriented service implemented in the transport protocol that is running on the end hosts. (We explain the qualification "essentially" in a moment.) This is precisely the model of the Internet, where IP provides a connectionless datagram delivery service and TCP implements an end-to-end connection abstraction. Note that this assumption does not hold in virtual circuit networks such as ATM and X.25. In such networks, a connection setup message traverses the network when a circuit is established. This setup message reserves a set of buffers for the connection at each router, thereby providing a form of congestion control—a connection is established only if enough buffers can be allocated to it at each router. The major shortcoming of this approach is that it leads to an underutilization of resources—buffers reserved for a particular circuit are not available for use by other traffic even if they were not currently being used by that circuit. The focus of this chapter is on resource allocation approaches that apply in an internetwork, and thus we focus mainly on connectionless networks.

We need to qualify the term *connectionless* because our classification of networks as being either connectionless or connection oriented is a bit too restrictive; there is a gray area in between. In particular, the assumption that all datagrams are completely independent in a connectionless network is too strong. The datagrams are certainly switched independently, but it is usually the case that a stream of datagrams between a particular pair of hosts flows through a particular set of routers. This idea of a *flow*—a sequence of packets

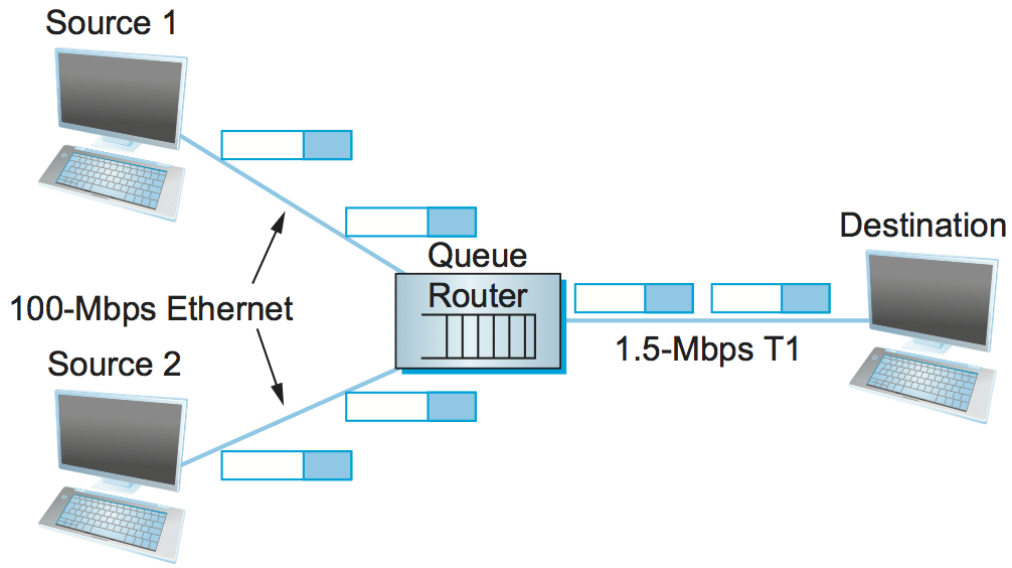


Figure 6.1.: A potential bottleneck router.

sent between a source/destination pair and following the same route through the network—is an important abstraction in the context of resource allocation; it is one that we will use in this chapter.

One of the powers of the flow abstraction is that flows can be defined at different granularities. For example, a flow can be host-to-host (i.e., have the same source/destination host addresses) or process-to-process (i.e., have the same source/destination host/port pairs). In the latter case, a flow is essentially the same as a channel, as we have been using that term throughout this book. The reason we introduce a new term is that a flow is visible to the routers inside the network, whereas a channel is an end-to-end abstraction. Figure 6.2 illustrates several flows passing through a series of routers.

Because multiple related packets flow through each router, it sometimes makes sense to maintain some state information for each flow, information that can be used to make resource allocation decisions about the packets that belong to the flow. This state is sometimes called *soft state*. The main difference between soft state and hard state is that soft state need not always be explicitly created and removed by signalling. Soft state represents a middle ground between a purely connectionless network that maintains *no* state at the routers and a purely connection-oriented network that maintains hard state at the routers. In general, the correct operation of the network does not depend on soft state being present (each packet is still routed correctly without regard to this state), but when a packet happens to belong to a flow for which the router is currently maintaining soft state, then the router is better able to handle the packet.

Note that a flow can be either implicitly defined or explicitly established. In the former case, each router watches for packets that happen to be traveling between the same source/destination pair—the router does this by inspecting the addresses in the header—and treats these packets as belonging to the same flow for the purpose of congestion control. In the latter case, the source sends a flow setup message across the network, declaring that a flow of packets is about to start. While explicit flows are arguably no different than a connection across a connection-oriented network, we call attention to this case because, even when explicitly established, a flow does not imply any end-to-end semantics and, in particular, does not imply the reliable and ordered delivery of a virtual circuit. It simply exists for the purpose of resource allocation. We will see examples of both implicit and explicit flows in this chapter.

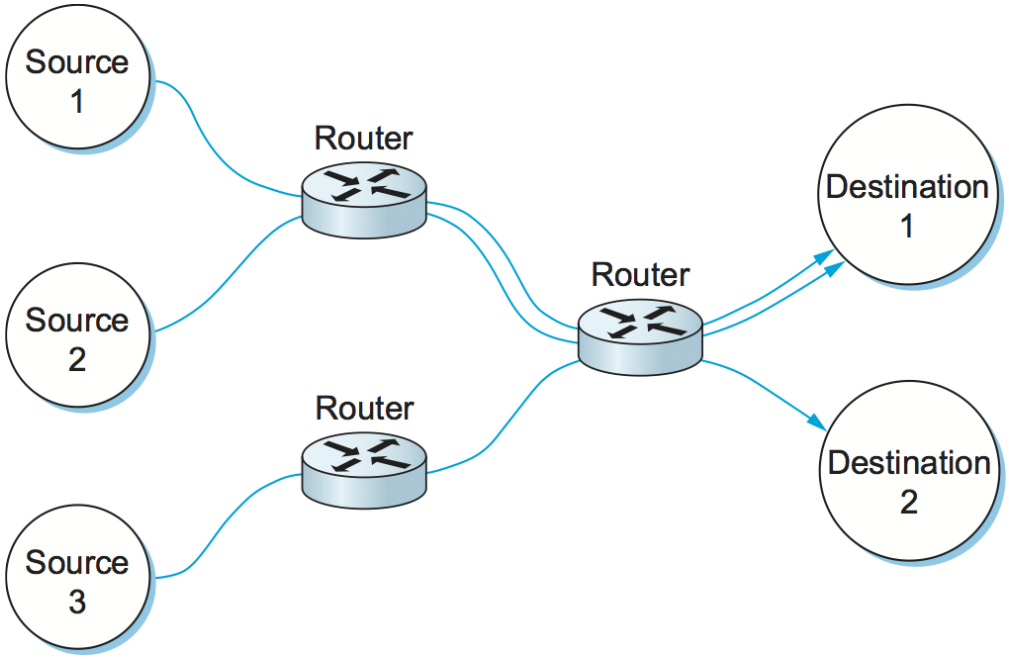


Figure 6.2.: Multiple flows passing through a set of routers.

Service Model

In the early part of this chapter, we will focus on mechanisms that assume the best-effort service model of the Internet. With best-effort service, all packets are given essentially equal treatment, with end hosts given no opportunity to ask the network that some packets or flows be given certain guarantees or preferential service. Defining a service model that supports some kind of preferred service or guarantee—for example, guaranteeing the bandwidth needed for a video stream—is the subject of a later section. Such a service model is said to provide multiple *qualities of service* (QoS). As we will see, there is actually a spectrum of possibilities, ranging from a purely best-effort service model to one in which individual flows receive quantitative guarantees of QoS. One of the greatest challenges is to define a service model that meets the needs of a wide range of applications and even allows for the applications that will be invented in the future.

6.1.2 Taxonomy

There are countless ways in which resource allocation mechanisms differ, so creating a thorough taxonomy is a difficult proposition. For now, we describe three dimensions along which resource allocation mechanisms can be characterized; more subtle distinctions will be called out during the course of this chapter.

Router-Centric versus Host-Centric

Resource allocation mechanisms can be classified into two broad groups: those that address the problem from inside the network (i.e., at the routers or switches) and those that address it from the edges of the network (i.e., in the hosts, perhaps inside the transport protocol). Since it is the case that both the routers inside the network and the hosts at the edges of the network participate in resource allocation, the real issue is where the majority of the burden falls.

In a router-centric design, each router takes responsibility for deciding when packets are forwarded and selecting which packets are to be dropped, as well as for informing the hosts that are generating the network traffic how many packets they are allowed to send. In a host-centric design, the end hosts observe the network conditions (e.g., how many packets they are successfully getting through the network) and adjust their behavior accordingly. Note that these two groups are not mutually exclusive. For example, a network that places the primary burden for managing congestion on routers still expects the end hosts to adhere to any advisory messages the routers send, while the routers in networks that use end-to-end congestion control still have some policy, no matter how simple, for deciding which packets to drop when their queues do overflow.

Reservation-Based versus Feedback-Based

A second way that resource allocation mechanisms are sometimes classified is according to whether they use *reservations* or *feedback*. In a reservation-based system, some entity (e.g., the end host) asks the network for a certain amount of capacity to be allocated for a flow. Each router then allocates enough resources (buffers and/or percentage of the link's bandwidth) to satisfy this request. If the request cannot be satisfied at some router, because doing so would overcommit its resources, then the router rejects the reservation. This is analogous to getting a busy signal when trying to make a phone call. In a feedback-based approach, the end hosts begin sending data without first reserving any capacity and then adjust their sending rate according to the feedback they receive. This feedback can be either *explicit* (i.e., a congested router sends a "please slow down" message to the host) or *implicit* (i.e., the end host adjusts its sending rate according to the externally observable behavior of the network, such as packet losses).

Note that a reservation-based system always implies a router-centric resource allocation mechanism. This is because each router is responsible for keeping track of how much of its capacity is currently available and deciding whether new reservations can be admitted. Routers may also have to make sure each host lives within the reservation it made. If a host sends data faster than it claimed it would when it made the reservation, then that host's packets are good candidates for discarding, should the router become congested. On the other hand, a feedback-based system can imply either a router- or host-centric mechanism. Typically, if the feedback is explicit, then the router is involved, to at least some degree, in the resource allocation scheme. If the feedback is implicit, then almost all of the burden falls to the end host; the routers silently drop packets when they become congested.

Reservations do not have to be made by end hosts. It is possible for a network administrator to allocate resources to flows or to larger aggregates of traffic, as we will see in a later section.

Window Based versus Rate Based

A third way to characterize resource allocation mechanisms is according to whether they are *window based* or *rate based*. This is one of the areas, noted above, where similar mechanisms and terminology are used for both flow control and congestion control. Both flow-control and resource allocation mechanisms need a way to express, to the sender, how much data it is allowed to transmit. There are two general ways of doing this: with a *window* or with a *rate*. We have already seen window-based transport protocols, such as TCP, in which the receiver advertises a window to the sender. This window corresponds to how much buffer space the receiver has, and it limits how much data the sender can transmit; that is, it supports flow control. A similar mechanism—window advertisement—can be used within the network to reserve buffer space (i.e., to support resource allocation). TCP's congestion-control mechanisms are window based.

It is also possible to control a sender's behavior using a rate—that is, how many bits per second the receiver or network is able to absorb. Rate-based control makes sense for many multimedia applications, which tend to generate data at some average rate and which need at least some minimum throughput to be useful. For example, a video codec might generate video at an average rate of 1 Mbps with a peak rate of 2 Mbps. As we will see later in this chapter, rate-based characterization of flows is a logical choice in a reservation-based system that supports different qualities of service—the sender makes a reservation for so many bits per second, and each router along the path determines if it can support that rate, given the other flows it has made commitments to.

Summary of Resource Allocation Taxonomy

Classifying resource allocation approaches at two different points along each of three dimensions, as we have just done, would seem to suggest up to eight unique strategies. While eight different approaches are certainly possible, we note that in practice two general strategies seem to be most prevalent; these two strategies are tied to the underlying service model of the network.

On the one hand, a best-effort service model usually implies that feedback is being used, since such a model does not allow users to reserve network capacity. This, in turn, means that most of the responsibility for congestion control falls to the end hosts, perhaps with some assistance from the routers. In practice, such networks use window-based information. This is the general strategy adopted in the Internet.

On the other hand, a QoS-based service model probably implies some form of reservation. Support for these reservations is likely to require significant router involvement, such as queuing packets differently depending on the level of reserved resources they require. Moreover, it is natural to express such reservations in terms of rate, since windows are only indirectly related to how much bandwidth a user needs from the network. We discuss this topic in a later section.

6.1.3 Evaluation Criteria

The final issue is one of knowing whether a resource allocation mechanism is good or not. Recall that in the problem statement at the start of this chapter we posed the question of how a network *effectively* and *fairly* allocates its resources. This suggests at least two broad measures by which a resource allocation scheme can be evaluated. We consider each in turn.

Effective Resource Allocation

A good starting point for evaluating the effectiveness of a resource allocation scheme is to consider the two principal metrics of networking: throughput and delay. Clearly, we want as much throughput and as little delay as possible. Unfortunately, these goals are often somewhat at odds with each other. One sure way for a resource allocation algorithm to increase throughput is to allow as many packets into the network as possible, so as to drive the utilization of all the links up to 100%. We would do this to avoid the possibility of a link becoming idle because an idle link necessarily hurts throughput. The problem with this strategy is that increasing the number of packets in the network also increases the length of the queues at each router. Longer queues, in turn, mean packets are delayed longer in the network.

To describe this relationship, some network designers have proposed using the ratio of throughput to delay as a metric for evaluating the effectiveness of a resource allocation scheme. This ratio is sometimes referred

to as the *power* of the network:

```
Power = Throughput / Delay
```

Note that it is not obvious that power is the right metric for judging resource allocation effectiveness. For one thing, the theory behind power is based on an M/M/1 queuing network that assumes infinite queues;[1] real networks have finite buffers and sometimes have to drop packets. For another, power is typically defined relative to a single connection (flow); it is not clear how it extends to multiple, competing connections. Despite these rather severe limitations, however, no alternatives have gained wide acceptance, and so power continues to be used.

The objective is to maximize this ratio, which is a function of how much load you place on the network. The load, in turn, is set by the resource allocation mechanism. Figure 6.3 gives a representative power curve, where, ideally, the resource allocation mechanism would operate at the peak of this curve. To the left of the peak, the mechanism is being too conservative; that is, it is not allowing enough packets to be sent to keep the links busy. To the right of the peak, so many packets are being allowed into the network that increases in delay due to queuing are starting to dominate any small gains in throughput.

Interestingly, this power curve looks very much like the system throughput curve in a timesharing computer system. System throughput improves as more jobs are admitted into the system, until it reaches a point when there are so many jobs running that the system begins to thrash (spends all of its time swapping memory pages) and the throughput begins to drop.

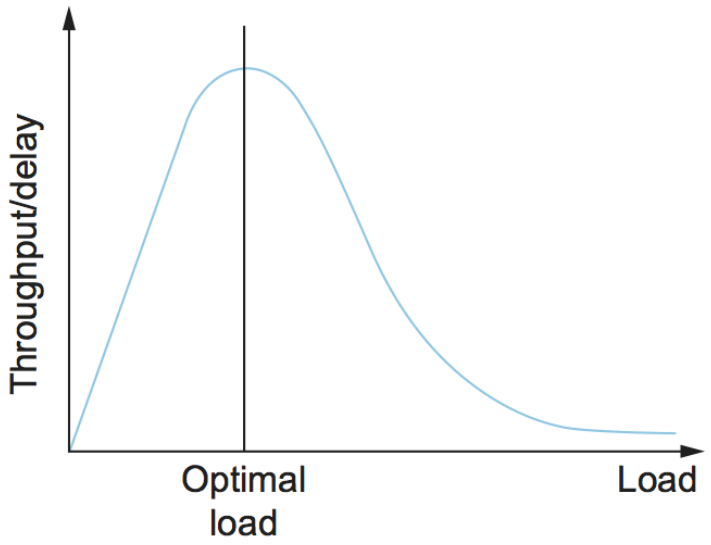


Figure 6.3.: Ratio of throughput to delay as a function of load.

As we will see in later sections of this chapter, many congestion-control schemes are able to control load in only very crude ways; that is, it is simply not possible to turn the "knob" a little and allow only a small number of additional packets into the network. As a consequence, network designers need to be concerned about what happens even when the system is operating under extremely heavy load—that is, at the rightmost end of the curve in Figure 6.3. Ideally, we would like to avoid the situation in which the system throughput goes to zero because the system is thrashing. In networking terminology, we want a system that

[1] Since this is not a queuing theory book, we provide only this brief description of an M/M/1 queue. The 1 means it has a single server, and the Ms mean that the distribution of both packet arrival and service times is *Markovian,* that is, exponential.

is *stable*—where packets continue to get through the network even when the network is operating under heavy load. If a mechanism is not stable, the network may experience *congestion collapse*.

Fair Resource Allocation

The effective utilization of network resources is not the only criterion for judging a resource allocation scheme. We must also consider the issue of fairness. However, we quickly get into murky waters when we try to define what exactly constitutes fair resource allocation. For example, a reservation-based resource allocation scheme provides an explicit way to create controlled unfairness. With such a scheme, we might use reservations to enable a video stream to receive 1 Mbps across some link while a file transfer receives only 10 kbps over the same link.

In the absence of explicit information to the contrary, when several flows share a particular link, we would like for each flow to receive an equal share of the bandwidth. This definition presumes that a *fair* share of bandwidth means an *equal* share of bandwidth. But, even in the absence of reservations, equal shares may not equate to fair shares. Should we also consider the length of the paths being compared? For example, as illustrated in Figure 6.4, what is fair when one four-hop flow is competing with three one-hop flows?

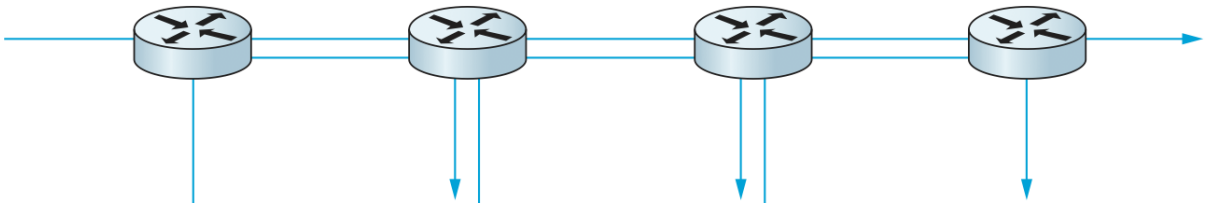


Figure 6.4.: One four-hop flow competing with three one-hop flows.

Assuming that fair implies equal and that all paths are of equal length, networking researcher Raj Jain proposed a metric that can be used to quantify the fairness of a congestion-control mechanism. Jain's fairness index is defined as follows. Given a set of flow throughputs

$$(x_1, x_2, \ldots, x_n)$$

(measured in consistent units such as bits/second), the following function assigns a fairness index to the flows:

$$f(x_1, x_2, \ldots, x_n) = \frac{(\sum_{i=1}^{n} x_i)^2}{n \sum_{i=1}^{n} x_i^2}$$

The fairness index always results in a number between 0 and 1, with 1 representing greatest fairness. To understand the intuition behind this metric, consider the case where all *n* flows receive a throughput of 1 unit of data per second. We can see that the fairness index in this case is

$$\frac{n^2}{n \times n} = 1$$

Now, suppose one flow receives a throughput of $1 + \Delta$. Now the fairness index is

$$\frac{((n-1) + 1 + \Delta)^2}{n(n - 1 + (1 + \Delta)^2)} = \frac{n^2 + 2n\Delta + \Delta^2}{n^2 + 2n\Delta + n\Delta^2}$$

Note that the denominator exceeds the numerator by $(n-1)\Delta^2$. Thus, whether the odd flow out was getting more or less than all the other flows (positive or negative Δ), the fairness index has now dropped below one. Another simple case to consider is where only *k* of the *n* flows receive equal throughput, and the remaining *n-k* users receive zero throughput, in which case the fairness index drops to *k/n*.

6.2 Queuing Disciplines

Regardless of how simple or how sophisticated the rest of the resource allocation mechanism is, each router must implement some queuing discipline that governs how packets are buffered while waiting to be transmitted. The queuing algorithm can be thought of as allocating both bandwidth (which packets get transmitted) and buffer space (which packets get discarded). It also directly affects the latency experienced by a packet by determining how long a packet waits to be transmitted. This section introduces two common queuing algorithms—first-in, first-out (FIFO) and fair queuing (FQ)—and identifies several variations that have been proposed.

6.2.1 FIFO

The idea of FIFO queuing, also called first-come, first-served (FCFS) queuing, is simple: The first packet that arrives at a router is the first packet to be transmitted. This is illustrated in Figure 6.5(a), which shows a FIFO with "slots" to hold up to eight packets. Given that the amount of buffer space at each router is finite, if a packet arrives and the queue (buffer space) is full, then the router discards that packet, as shown in Figure 6.5(b). This is done without regard to which flow the packet belongs to or how important the packet is. This is sometimes called *tail drop*, since packets that arrive at the tail end of the FIFO are dropped.

Note that tail drop and FIFO are two separable ideas. FIFO is a *scheduling discipline*—it determines the order in which packets are transmitted. Tail drop is a *drop policy*—it determines which packets get dropped. Because FIFO and tail drop are the simplest instances of scheduling discipline and drop policy, respectively, they are sometimes viewed as a bundle—the vanilla queuing implementation. Unfortunately, the bundle is often referred to simply as *FIFO queuing*, when it should more precisely be called *FIFO with tail drop*. A later section provides an example of another drop policy, which uses a more complex algorithm than "Is there a free buffer?" to decide when to drop packets. Such a drop policy may be used with FIFO, or with more complex scheduling disciplines.

FIFO with tail drop, as the simplest of all queuing algorithms, is the most widely used in Internet routers at the time of writing. This simple approach to queuing pushes all responsibility for congestion control and resource allocation out to the edges of the network. Thus, the prevalent form of congestion control in the Internet currently assumes no help from the routers: TCP takes responsibility for detecting and responding to congestion. We will see how this works in the next section.

A simple variation on basic FIFO queuing is priority queuing. The idea is to mark each packet with a priority; the mark could be carried, for example, in the IP header, as we'll discuss in a later section. The routers then implement multiple FIFO queues, one for each priority class. The router always transmits packets out of the highest-priority queue if that queue is nonempty before moving on to the next priority queue. Within each priority, packets are still managed in a FIFO manner. This idea is a small departure from the best-effort delivery model, but it does not go so far as to make guarantees to any particular priority class. It just allows high-priority packets to cut to the front of the line.

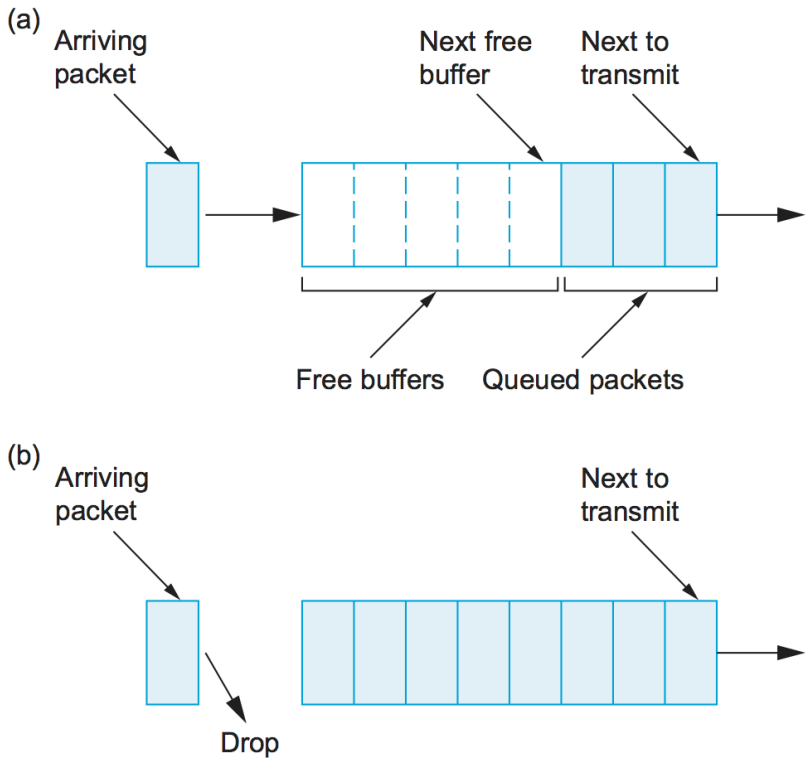


Figure 6.5.: FIFO queuing (a), and tail drop at a FIFO queue (b).

The problem with priority queuing, of course, is that the high-priority queue can starve out all the other queues; that is, as long as there is at least one high-priority packet in the high-priority queue, lower-priority queues do not get served. For this to be viable, there needs to be hard limits on how much high-priority traffic is inserted in the queue. It should be immediately clear that we can't allow users to set their own packets to high priority in an uncontrolled way; we must either prevent them from doing this altogether or provide some form of "pushback" on users. One obvious way to do this is to use economics—the network could charge more to deliver high-priority packets than low-priority packets. However, there are significant challenges to implementing such a scheme in a decentralized environment such as the Internet.

One situation in which priority queuing is used in the Internet is to protect the most important packets—typically, the routing updates that are necessary to stabilize the routing tables after a topology change. Often there is a special queue for such packets, which can be identified by the Differentiated Services Code Point (formerly the TOS field) in the IP header. This is in fact a simple case of the idea of "Differentiated Services."

6.2.2 Fair Queuing

The main problem with FIFO queuing is that it does not discriminate between different traffic sources, or, in the language introduced in the previous section, it does not separate packets according to the flow to which they belong. This is a problem at two different levels. At one level, it is not clear that any congestion-control algorithm implemented entirely at the source will be able to adequately control congestion with so little help from the routers. We will suspend judgment on this point until the next section when we discuss TCP congestion control. At another level, because the entire congestion-control mechanism is implemented

at the sources and FIFO queuing does not provide a means to police how well the sources adhere to this mechanism, it is possible for an ill-behaved source (flow) to capture an arbitrarily large fraction of the network capacity. Considering the Internet again, it is certainly possible for a given application not to use TCP and, as a consequence, to bypass its end-to-end congestion-control mechanism. (Applications such as Internet telephony do this today.) Such an application is able to flood the Internet's routers with its own packets, thereby causing other applications' packets to be discarded.

Fair queuing (FQ) is an algorithm that has been designed to address this problem. The idea of FQ is to maintain a separate queue for each flow currently being handled by the router. The router then services these queues in a sort of round-robin, as illustrated in Figure 6.6. When a flow sends packets too quickly, then its queue fills up. When a queue reaches a particular length, additional packets belonging to that flow's queue are discarded. In this way, a given source cannot arbitrarily increase its share of the network's capacity at the expense of other flows.

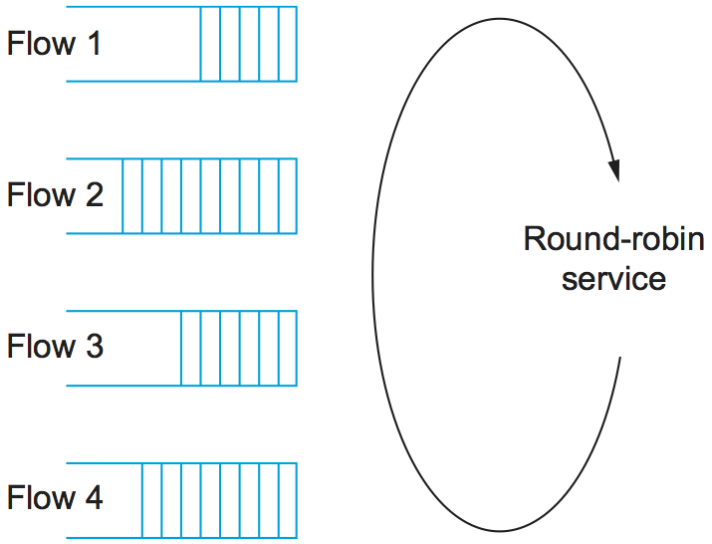


Figure 6.6.: Round-robin service of four flows at a router.

Note that FQ does not involve the router telling the traffic sources anything about the state of the router or in any way limiting how quickly a given source sends packets. In other words, FQ is still designed to be used in conjunction with an end-to-end congestion-control mechanism. It simply segregates traffic so that ill-behaved traffic sources do not interfere with those that are faithfully implementing the end-to-end algorithm. FQ also enforces fairness among a collection of flows managed by a well-behaved congestion-control algorithm.

As simple as the basic idea is, there are still a modest number of details that you have to get right. The main complication is that the packets being processed at a router are not necessarily the same length. To truly allocate the bandwidth of the outgoing link in a fair manner, it is necessary to take packet length into consideration. For example, if a router is managing two flows, one with 1000-byte packets and the other with 500-byte packets (perhaps because of fragmentation upstream from this router), then a simple round-robin servicing of packets from each flow's queue will give the first flow two-thirds of the link's bandwidth and the second flow only one-third of its bandwidth.

What we really want is bit-by-bit round-robin, where the router transmits a bit from flow 1, then a bit from flow 2, and so on. Clearly, it is not feasible to interleave the bits from different packets. The FQ mechanism therefore simulates this behavior by first determining when a given packet would finish being transmitted if

it were being sent using bit-by-bit round-robin and then using this finishing time to sequence the packets for transmission.

To understand the algorithm for approximating bit-by-bit round-robin, consider the behavior of a single flow and imagine a clock that ticks once each time one bit is transmitted from all of the active flows. (A flow is active when it has data in the queue.) For this flow, let P_i denote the length of packet *i*, let S_i denote the time when the router starts to transmit packet *i*, and let F_i denote the time when the router finishes transmitting packet *i*. If P_i is expressed in terms of how many clock ticks it takes to transmit packet *i* (keeping in mind that time advances 1 tick each time this flow gets 1 bit's worth of service), then it is easy to see that $F_i = S_i + P_i$.

When do we start transmitting packet *i*? The answer to this question depends on whether packet *i* arrived before or after the router finished transmitting packet *i-1* from this flow. If it was before, then logically the first bit of packet *i* is transmitted immediately after the last bit of packet *i-1*. On the other hand, it is possible that the router finished transmitting packet *i-1* long before *i* arrived, meaning that there was a period of time during which the queue for this flow was empty, so the round-robin mechanism could not transmit any packets from this flow. If we let A_i denote the time that packet *i* arrives at the router, then $S_i = \max(F_{i-1}, A_i)$. Thus, we can compute

$$F_i = \max(F_{i-1}, A_i) + P_i$$

Now we move on to the situation in which there is more than one flow, and we find that there is a catch to determining A_i. We can't just read the wall clock when the packet arrives. As noted above, we want time to advance by one tick each time all the active flows get one bit of service under bit-by-bit round-robin, so we need a clock that advances more slowly when there are more flows. Specifically, the clock must advance by one tick when *n* bits are transmitted if there are *n* active flows. This clock will be used to calculate A_i.

Now, for every flow, we calculate F_i for each packet that arrives using the above formula. We then treat all the F_i as timestamps, and the next packet to transmit is always the packet that has the lowest timestamp—the packet that, based on the above reasoning, should finish transmission before all others.

Note that this means that a packet can arrive on a flow, and, because it is shorter than a packet from some other flow that is already in the queue waiting to be transmitted, it can be inserted into the queue in front of that longer packet. However, this does not mean that a newly arriving packet can preempt a packet that is currently being transmitted. It is this lack of preemption that keeps the implementation of FQ just described from exactly simulating the bit-by-bit round-robin scheme that we are attempting to approximate.

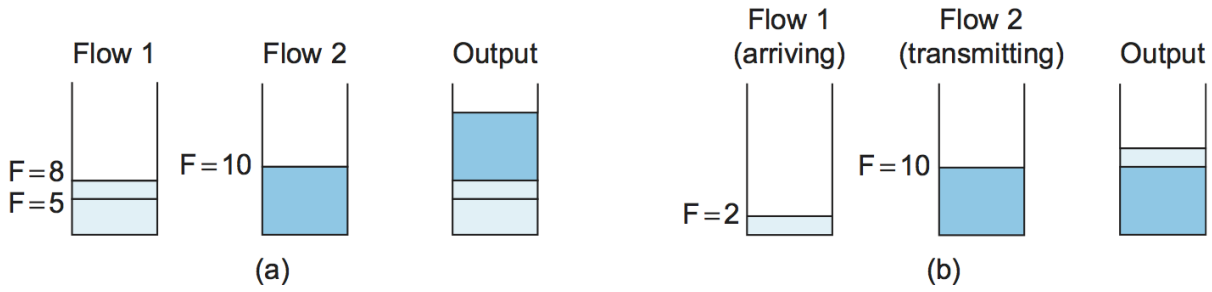


Figure 6.7.: Example of fair queuing in action: (a) Packets with earlier finishing times are sent first; (b) sending of a packet already in progress is completed.

To better see how this implementation of fair queuing works, consider the example given in Figure 6.7. Part (a) shows the queues for two flows; the algorithm selects both packets from flow 1 to be transmitted

before the packet in the flow 2 queue, because of their earlier finishing times. In (b), the router has already begun to send a packet from flow 2 when the packet from flow 1 arrives. Though the packet arriving on flow 1 would have finished before flow 2 if we had been using perfect bit-by-bit fair queuing, the implementation does not preempt the flow 2 packet.

There are two things to notice about fair queuing. First, the link is never left idle as long as there is at least one packet in the queue. Any queuing scheme with this characteristic is said to be *work conserving*. One effect of being work conserving is that if I am sharing a link with a lot of flows that are not sending any data then; I can use the full link capacity for my flow. As soon as the other flows start sending, however, they will start to use their share and the capacity available to my flow will drop.

The second thing to notice is that if the link is fully loaded and there are *n* flows sending data, I cannot use more than 1/nth of the link bandwidth. If I try to send more than that, my packets will be assigned increasingly large timestamps, causing them to sit in the queue longer awaiting transmission. Eventually, the queue will overflow—although whether it is my packets or someone else's that are dropped is a decision that is not determined by the fact that we are using fair queuing. This is determined by the drop policy; FQ is a scheduling algorithm, which, like FIFO, may be combined with various drop policies.

Because FQ is work conserving, any bandwidth that is not used by one flow is automatically available to other flows. For example, if we have four flows passing through a router, and all of them are sending packets, then each one will receive one-quarter of the bandwidth. But, if one of them is idle long enough that all its packets drain out of the router's queue, then the available bandwidth will be shared among the remaining three flows, which will each now receive one-third of the bandwidth. Thus, we can think of FQ as providing a guaranteed minimum share of bandwidth to each flow, with the possibility that it can get more than its guarantee if other flows are not using their shares.

It is possible to implement a variation of FQ, called *weighted fair queuing* (WFQ), that allows a weight to be assigned to each flow (queue). This weight logically specifies how many bits to transmit each time the router services that queue, which effectively controls the percentage of the link's bandwidth that that flow will get. Simple FQ gives each queue a weight of 1, which means that logically only 1 bit is transmitted from each queue each time around. This results in each flow getting $1/n^{th}$ of the bandwidth when there are *n* flows. With WFQ, however, one queue might have a weight of 2, a second queue might have a weight of 1, and a third queue might have a weight of 3. Assuming that each queue always contains a packet waiting to be transmitted, the first flow will get one-third of the available bandwidth, the second will get one-sixth of the available bandwidth, and the third will get one-half of the available bandwidth.

While we have described WFQ in terms of flows, note that it could be implemented on *classes* of traffic, where classes are defined in some other way than the simple flows introduced at the start of this chapter. For example, we could use some bits in the IP header to identify classes and allocate a queue and a weight to each class. This is exactly what is proposed as part of the Differentiated Services architecture described in a later section.

Note that a router performing WFQ must learn what weights to assign to each queue from somewhere, either by manual configuration or by some sort of signalling from the sources. In the latter case, we are moving toward a reservation-based model. Just assigning a weight to a queue provides a rather weak form of reservation because these weights are only indirectly related to the bandwidth the flow receives. (The bandwidth available to a flow also depends, for example, on how many other flows are sharing the link.) We will see in a later section how WFQ can be used as a component of a reservation-based resource allocation mechanism.

Key Takeaway

Finally, we observe that this whole discussion of queue management illustrates an important system design principle known as *separating policy and mechanism*. The idea is to view each mechanism as a black box that provides a multifaceted service that can be controlled by a set of knobs. A policy specifies a particular setting of those knobs but does not know (or care) about how the black box is implemented. In this case, the mechanism in question is the queuing discipline, and the policy is a particular setting of which flow gets what level of service (e.g., priority or weight). We discuss some policies that can be used with the WFQ mechanism in a later section. *[Next]*

6.3 TCP Congestion Control

This section describes the predominant example of end-to-end congestion control in use today, that implemented by TCP. The essential strategy of TCP is to send packets into the network without a reservation and then to react to observable events that occur. TCP assumes only FIFO queuing in the network's routers, but also works with fair queuing.

TCP congestion control was introduced into the Internet in the late 1980s by Van Jacobson, roughly eight years after the TCP/IP protocol stack had become operational. Immediately preceding this time, the Internet was suffering from congestion collapse—hosts would send their packets into the Internet as fast as the advertised window would allow, congestion would occur at some router (causing packets to be dropped), and the hosts would time out and retransmit their packets, resulting in even more congestion.

Broadly speaking, the idea of TCP congestion control is for each source to determine how much capacity is available in the network, so that it knows how many packets it can safely have in transit. Once a given source has this many packets in transit, it uses the arrival of an ACK as a signal that one of its packets has left the network and that it is therefore safe to insert a new packet into the network without adding to the level of congestion. By using ACKs to pace the transmission of packets, TCP is said to be *self-clocking*. Of course, determining the available capacity in the first place is no easy task. To make matters worse, because other connections come and go, the available bandwidth changes over time, meaning that any given source must be able to adjust the number of packets it has in transit. This section describes the algorithms used by TCP to address these and other problems.

Note that, although we describe the TCP congestion-control mechanisms one at a time, thereby giving the impression that we are talking about three independent mechanisms, it is only when they are taken as a whole that we have TCP congestion control. Also, while we are going to begin here with the variant of TCP congestion control most often referred to as *standard TCP*, we will see that there are actually quite a few variants of TCP congestion control in use today, and researchers continue to explore new approaches to addressing this problem. Some of these new approaches are discussed below.

6.3.1 Additive Increase/Multiplicative Decrease

TCP maintains a new state variable for each connection, called `CongestionWindow`, which is used by the source to limit how much data it is allowed to have in transit at a given time. The congestion window is congestion control's counterpart to flow control's advertised window. TCP is modified such that the maximum number of bytes of unacknow- ledged data allowed is now the minimum of the congestion window and

the advertised window. Thus, using the variables defined in the previous chapter, TCP's effective window is revised as follows:

```
MaxWindow = MIN(CongestionWindow, AdvertisedWindow)
EffectiveWindow = MaxWindow -  (LastByteSent - LastByteAcked)
```

That is, `MaxWindow` replaces `AdvertisedWindow` in the calculation of `EffectiveWindow`. Thus, a TCP source is allowed to send no faster than the slowest component—the network or the destination host—can accommodate.

The problem, of course, is how TCP comes to learn an appropriate value for `CongestionWindow`. Unlike the `AdvertisedWindow`, which is sent by the receiving side of the connection, there is no one to send a suitable `CongestionWindow` to the sending side of TCP. The answer is that the TCP source sets the `CongestionWindow` based on the level of congestion it perceives to exist in the network. This involves decreasing the congestion window when the level of congestion goes up and increasing the congestion window when the level of congestion goes down. Taken together, the mechanism is commonly called *additive increase/multiplicative decrease* (AIMD); the reason for this mouthful of a name will become apparent below.

The key question, then, is how does the source determine that the network is congested and that it should decrease the congestion window? The answer is based on the observation that the main reason packets are not delivered, and a timeout results, is that a packet was dropped due to congestion. It is rare that a packet is dropped because of an error during transmission. Therefore, TCP interprets timeouts as a sign of congestion and reduces the rate at which it is transmitting. Specifically, each time a timeout occurs, the source sets `CongestionWindow` to half of its previous value. This halving of the `CongestionWindow` for each timeout corresponds to the "multiplicative decrease" part of AIMD.

Although `CongestionWindow` is defined in terms of bytes, it is easiest to understand multiplicative decrease if we think in terms of whole packets. For example, suppose the `CongestionWindow` is currently set to 16 packets. If a loss is detected, `CongestionWindow` is set to 8. (Normally, a loss is detected when a timeout occurs, but as we see below, TCP has another mechanism to detect dropped packets.) Additional losses cause `CongestionWindow` to be reduced to 4, then 2, and finally to 1 packet. `CongestionWindow` is not allowed to fall below the size of a single packet, or in TCP terminology, the *maximum segment size* .

A congestion-control strategy that only decreases the window size is obviously too conservative. We also need to be able to increase the congestion window to take advantage of newly available capacity in the network. This is the "additive increase" part of AIMD, and it works as follows. Every time the source successfully sends a `CongestionWindow`'s worth of packets—that is, each packet sent out during the last round-trip time (RTT) has been ACKed—it adds the equivalent of 1 packet to `CongestionWindow`. This linear increase is illustrated in Figure 6.8. Note that, in practice, TCP does not wait for an entire window's worth of ACKs to add 1 packet's worth to the congestion window, but instead increments `CongestionWindow` by a little for each ACK that arrives. Specifically, the congestion window is incremented as follows each time an ACK arrives:

```
Increment = MSS x (MSS/CongestionWindow)
CongestionWindow += Increment
```

That is, rather than incrementing `CongestionWindow` by an entire `MSS` bytes each RTT, we increment it by a fraction of `MSS` every time an ACK is received. Assuming that each ACK acknowledges the receipt of `MSS` bytes, then that fraction is .

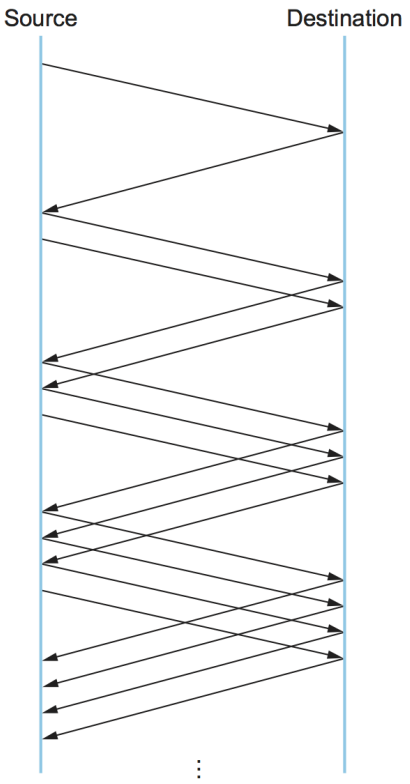


Figure 6.8.: Packets in transit during additive increase, with one packet being added each RTT.

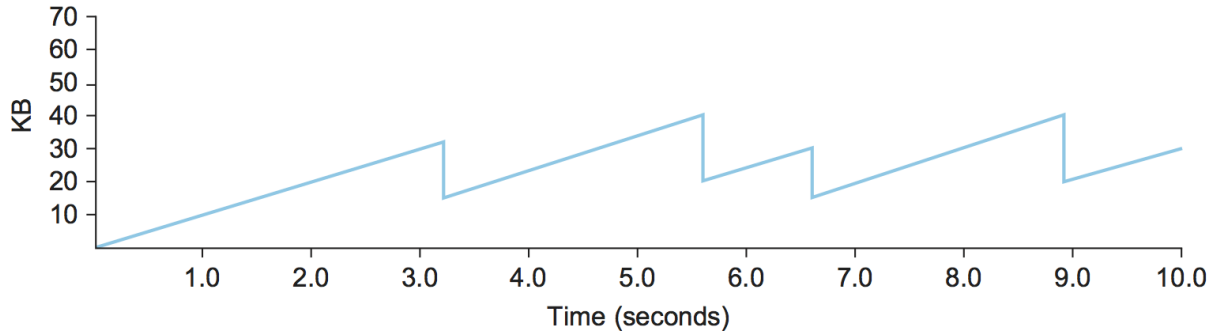


Figure 6.9.: Typical TCP sawtooth pattern.

This pattern of continually increasing and decreasing the congestion window continues throughout the lifetime of the connection. In fact, if you plot the current value of `CongestionWindow` as a function of time, you get a sawtooth pattern, as illustrated in Figure 6.9. The important concept to understand about AIMD is that the source is willing to reduce its congestion window at a much faster rate than it is willing to increase its congestion window. This is in contrast to an additive increase/additive decrease strategy in which the window would be increased by 1 packet when an ACK arrives and decreased by 1 when a timeout occurs. It has been shown that AIMD is a necessary condition for a congestion-control mechanism to be stable (see the Further Reading section). One intuitive reason to decrease the window aggressively and increase it conservatively is that the consequences of having too large a window are much worse than those of it being too small. For example, when the window is too large, packets that are dropped will be retransmitted, making congestion even worse; thus, it is important to get out of this state quickly.

Finally, since a timeout is an indication of congestion that triggers multiplicative decrease, TCP needs the most accurate timeout mechanism it can afford. We already covered TCP's timeout mechanism in an earlier chapter, so we do not repeat it here. The two main things to remember about that mechanism are that (1) timeouts are set as a function of both the average RTT and the standard deviation in that average, and (2) due to the cost of measuring each transmission with an accurate clock, TCP only samples the round-trip time once per RTT (rather than once per packet) using a coarse-grained (500-ms) clock.

6.3.2 Slow Start

The additive increase mechanism just described is the right approach to use when the source is operating close to the available capacity of the network, but it takes too long to ramp up a connection when it is starting from scratch. TCP therefore provides a second mechanism, ironically called *slow start*, which is used to increase the congestion window rapidly from a cold start. Slow start effectively increases the congestion window exponentially, rather than linearly.

Specifically, the source starts out by setting `CongestionWindow` to one packet. When the ACK for this packet arrives, TCP adds 1 to `CongestionWindow` and then sends two packets. Upon receiving the corresponding two ACKs, TCP increments `CongestionWindow` by 2—one for each ACK—and next sends four packets. The end result is that TCP effectively doubles the number of packets it has in transit every RTT. Figure 6.10 shows the growth in the number of packets in transit during slow start. Compare this to the linear growth of additive increase illustrated in Figure 6.8.

Why any exponential mechanism would be called "slow" is puzzling at first, but it can be explained if put in the proper historical context. We need to compare slow start not against the linear mechanism of the previous subsection, but against the original behavior of TCP. Consider what happens when a connection is established and the source first starts to send packets—that is, when it currently has no packets in transit. If the source sends as many packets as the advertised window allows—which is exactly what TCP did before slow start was developed—then even if there is a fairly large amount of bandwidth available in the network, the routers may not be able to consume this burst of packets. It all depends on how much buffer space is available at the routers. Slow start was therefore designed to space packets out so that this burst does not occur. In other words, even though its exponential growth is faster than linear growth, slow start is much "slower" than sending an entire advertised window's worth of data all at once.

There are actually two different situations in which slow start runs. The first is at the very beginning of a connection, at which time the source has no idea how many packets it is going to be able to have in transit at a given time. (Keep in mind that today TCP runs over everything from 1-Mbps links to 40-Gbps links, so there is no way for the source to know the network's capacity.) In this situation, slow start continues to

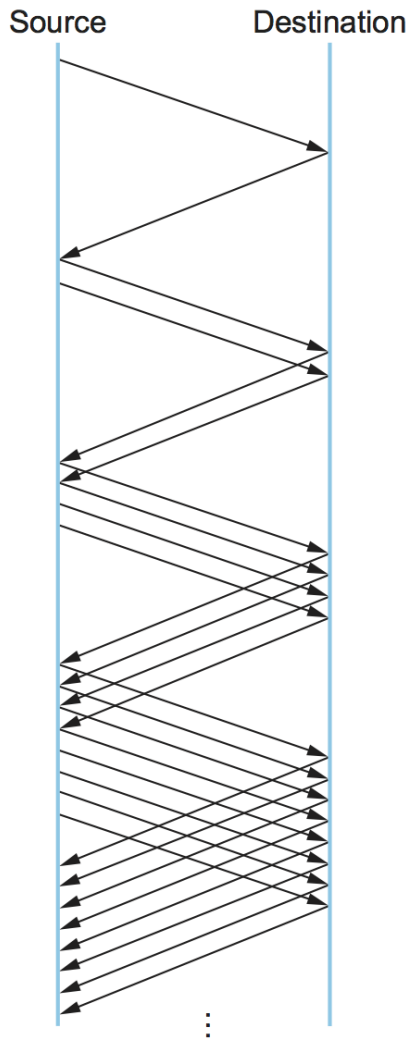


Figure 6.10.: Packets in transit during slow start.

double CongestionWindow each RTT until there is a loss, at which time a timeout causes multiplicative decrease to divide CongestionWindow by 2.

The second situation in which slow start is used is a bit more subtle; it occurs when the connection goes dead while waiting for a timeout to occur. Recall how TCP's sliding window algorithm works—when a packet is lost, the source eventually reaches a point where it has sent as much data as the advertised window allows, and so it blocks while waiting for an ACK that will not arrive. Eventually, a timeout happens, but by this time there are no packets in transit, meaning that the source will receive no ACKs to "clock" the transmission of new packets. The source will instead receive a single cumulative ACK that reopens the entire advertised window, but, as explained above, the source then uses slow start to restart the flow of data rather than dumping a whole window's worth of data on the network all at once.

Although the source is using slow start again, it now knows more information than it did at the beginning of a connection. Specifically, the source has a current (and useful) value of CongestionWindow; this is the value of CongestionWindow that existed prior to the last packet loss, divided by 2 as a result of the loss. We can think of this as the *target* congestion window. Slow start is used to rapidly increase the sending rate up to this value, and then additive increase is used beyond this point. Notice that we have a small bookkeeping problem to take care of, in that we want to remember the target congestion window resulting from multiplicative decrease as well as the *actual* congestion window being used by slow start. To address this problem, TCP introduces a temporary variable to store the target window, typically called CongestionThreshold, that is set equal to the CongestionWindow value that results from multiplicative decrease. The variable CongestionWindow is then reset to one packet, and it is incremented by one packet for every ACK that is received until it reaches CongestionThreshold, at which point it is incremented by one packet per RTT.

In other words, TCP increases the congestion window as defined by the following code fragment:

```
{
    u_int    cw = state->CongestionWindow;
    u_int    incr = state->maxseg;

    if (cw > state->CongestionThreshold)
        incr = incr * incr / cw;
    state->CongestionWindow = MIN(cw + incr, TCP_MAXWIN);
}
```

where state represents the state of a particular TCP connection and defines an upper bound on how large the congestion window is allowed to grow.

Figure 6.11 traces how TCP's CongestionWindow increases and decreases over time and serves to illustrate the interplay of slow start and additive increase/multiplicative decrease. This trace was taken from an actual TCP connection and shows the current value of CongestionWindow—the colored line—over time.

There are several things to notice about this trace. The first is the rapid increase in the congestion window at the beginning of the connection. This corresponds to the initial slow start phase. The slow start phase continues until several packets are lost at about 0.4 seconds into the connection, at which time CongestionWindow flattens out at about 34 KB. (Why so many packets are lost during slow start is discussed below.) The reason why the congestion window flattens is that there are no ACKs arriving, due to the fact that several packets were lost. In fact, no new packets are sent during this time, as denoted by the lack of hash marks at the top of the graph. A timeout eventually happens at approximately 2 seconds, at

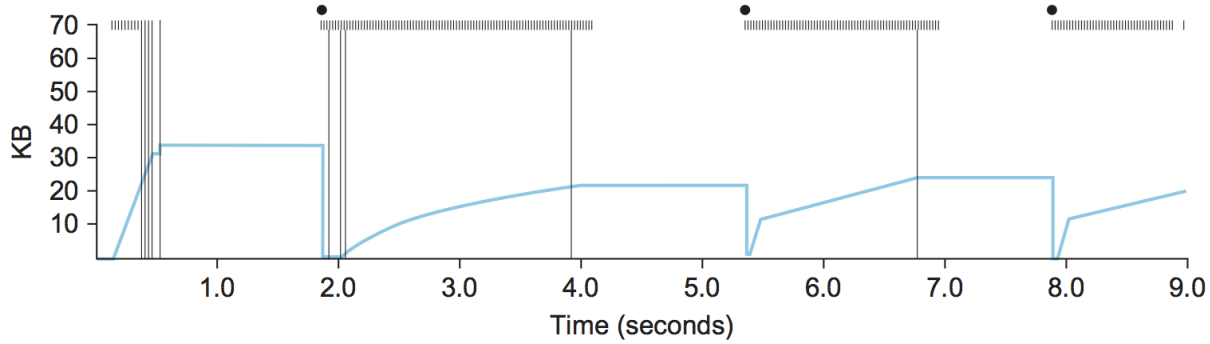


Figure 6.11.: Behavior of TCP congestion control. Colored line = value of CongestionWindow over time; solid bullets at top of graph = timeouts; hash marks at top of graph = time when each packet is transmitted; vertical bars = time when a packet that was eventually retransmitted was first transmitted.

which time the congestion window is divided by 2 (i.e., cut from approximately 34 KB to around 17 KB) and `CongestionThreshold` is set to this value. Slow start then causes `CongestionWindow` to be reset to one packet and to start ramping up from there.

There is not enough detail in the trace to see exactly what happens when a couple of packets are lost just after 2 seconds, so we jump ahead to the linear increase in the congestion window that occurs between 2 and 4 seconds. This corresponds to additive increase. At about 4 seconds, `CongestionWindow` flattens out, again due to a lost packet. Now, at about 5.5 seconds:

1. A timeout happens, causing the congestion window to be divided by 2, dropping it from approximately 22 KB to 11 KB, and `CongestionThreshold` is set to this amount.
2. `CongestionWindow` is reset to one packet, as the sender enters slow start.
3. Slow start causes `CongestionWindow` to grow exponentially until it reaches `CongestionThreshold`.
4. `CongestionWindow` then grows linearly.

The same pattern is repeated at around 8 seconds when another timeout occurs.

We now return to the question of why so many packets are lost during the initial slow start period. At this point, TCP is attempting to learn how much bandwidth is available on the network. This is a very difficult task. If the source is not aggressive at this stage—for example, if it only increases the congestion window linearly—then it takes a long time for it to discover how much bandwidth is available. This can have a dramatic impact on the throughput achieved for this connection. On the other hand, if the source is aggressive at this stage, as TCP is during exponential growth, then the source runs the risk of having half a window's worth of packets dropped by the network.

To see what can happen during exponential growth, consider the situation in which the source was just able to successfully send 16 packets through the network, causing it to double its congestion window to 32. Suppose, however, that the network happens to have just enough capacity to support 16 packets from this source. The likely result is that 16 of the 32 packets sent under the new congestion window will be dropped by the network; actually, this is the worst-case outcome, since some of the packets will be buffered in some router. This problem will become increasingly severe as the delay × bandwidth product of networks increases. For example, a delay × bandwidth product of 500 KB means that each connection has the potential to lose up to 500 KB of data at the beginning of each connection. Of course, this assumes that both the source

and the destination implement the "big windows" extension.

Alternatives to slow start, whereby the source tries to estimate the available bandwidth by more sophisticated means, have also been explored. One example is called *quick-start*. The basic idea is that a TCP sender can ask for an initial sending rate greater than slow start would allow by putting a requested rate in its SYN packet as an IP option. Routers along the path can examine the option, evaluate the current level of congestion on the outgoing link for this flow, and decide if that rate is acceptable, if a lower rate would be acceptable, or if standard slow start should be used. By the time the SYN reaches the receiver, it will contain either a rate that was acceptable to all routers on the path or an indication that one or more routers on the path could not support the quick-start request. In the former case, the TCP sender uses that rate to begin transmission; in the latter case, it falls back to standard slow start. If TCP is allowed to start off sending at a higher rate, a session could more quickly reach the point of filling the pipe, rather than taking many round-trip times to do so.

Clearly one of the challenges to this sort of enhancement to TCP is that it requires substantially more cooperation from the routers than standard TCP does. If a single router in the path does not support quick-start, then the system reverts to standard slow start. Thus, it could be a long time before these types of enhancements could make it into the Internet; for now, they are more likely to be used in controlled network environments (e.g., research networks).

6.3.3 Fast Retransmit and Fast Recovery

The mechanisms described so far were part of the original proposal to add congestion control to TCP. It was soon discovered, however, that the coarse-grained implementation of TCP timeouts led to long periods of time during which the connection went dead while waiting for a timer to expire. Because of this, a new mechanism called *fast retransmit* was added to TCP. Fast retransmit is a heuristic that sometimes triggers the retransmission of a dropped packet sooner than the regular timeout mechanism. The fast retransmit mechanism does not replace regular timeouts; it just enhances that facility.

The idea of fast retransmit is straightforward. Every time a data packet arrives at the receiving side, the receiver responds with an acknowledgment, even if this sequence number has already been acknowledged. Thus, when a packet arrives out of order—when TCP cannot yet acknowledge the data the packet contains because earlier data has not yet arrived—TCP resends the same acknowledgment it sent the last time. This second transmission of the same acknowledgment is called a *duplicate ACK*. When the sending side sees a duplicate ACK, it knows that the other side must have received a packet out of order, which suggests that an earlier packet might have been lost. Since it is also possible that the earlier packet has only been delayed rather than lost, the sender waits until it sees some number of duplicate ACKs and then retransmits the missing packet. In practice, TCP waits until it has seen three duplicate ACKs before retransmitting the packet.

Figure 6.12 illustrates how duplicate ACKs lead to a fast retransmit. In this example, the destination receives packets 1 and 2, but packet 3 is lost in the network. Thus, the destination will send a duplicate ACK for packet 2 when packet 4 arrives, again when packet 5 arrives, and so on. (To simplify this example, we think in terms of packets 1, 2, 3, and so on, rather than worrying about the sequence numbers for each byte.) When the sender sees the third duplicate ACK for packet 2—the one sent because the receiver had gotten packet 6—it retransmits packet 3. Note that when the retransmitted copy of packet 3 arrives at the destination, the receiver then sends a cumulative ACK for everything up to and including packet 6 back to the source.

Figure 6.13 illustrates the behavior of a version of TCP with the fast retransmit mechanism. It is interesting

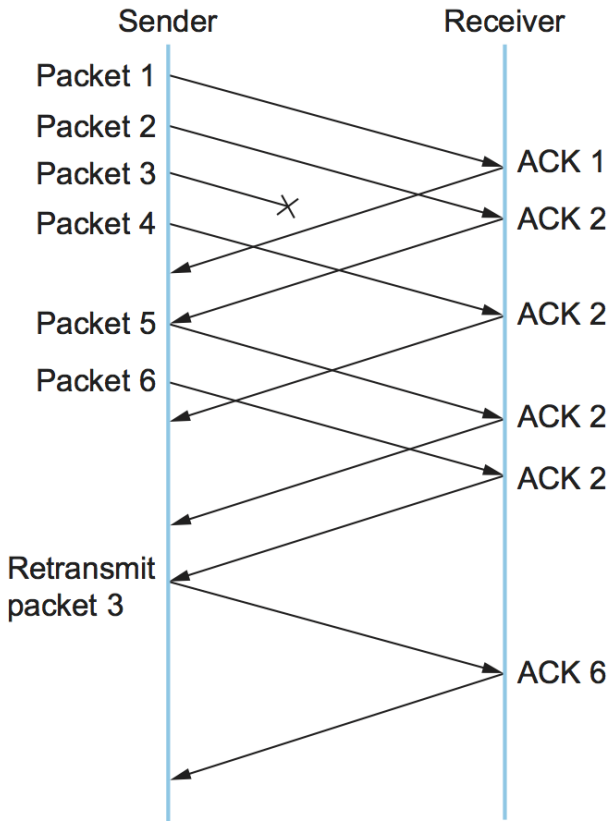


Figure 6.12.: Fast retransmit based on duplicate ACKs.

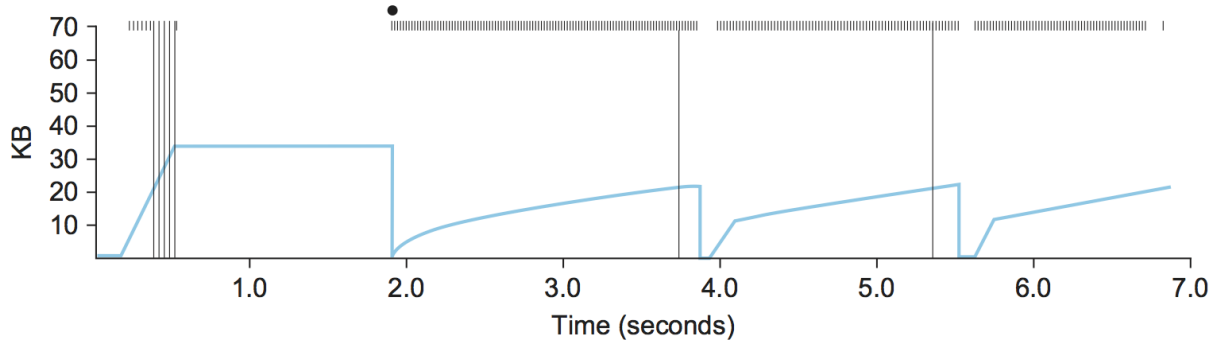


Figure 6.13.: Trace of TCP with fast retransmit. Colored line = CongestionWindow; solid bullet = timeout; hash marks = time when each packet is transmitted; vertical bars = time when a packet that was eventually retransmitted was first transmitted.

to compare this trace with that given in Figure 6.11, where fast retransmit was not implemented—the long periods during which the congestion window stays flat and no packets are sent has been eliminated. In general, this technique is able to eliminate about half of the coarse-grained timeouts on a typical TCP connection, resulting in roughly a 20% improvement in the throughput over what could otherwise have been achieved. Notice, however, that the fast retransmit strategy does not eliminate all coarse-grained timeouts. This is because for a small window size there will not be enough packets in transit to cause enough duplicate ACKs to be delivered. Given enough lost packets—for example, as happens during the initial slow start phase—the sliding window algorithm eventually blocks the sender until a timeout occurs. In practice, TCP's fast retransmit mechanism can detect up to three dropped packets per window.

Finally, there is one last improvement we can make. When the fast retransmit mechanism signals congestion, rather than drop the congestion window all the way back to one packet and run slow start, it is possible to use the ACKs that are still in the pipe to clock the sending of packets. This mechanism, which is called *fast recovery*, effectively removes the slow start phase that happens between when fast retransmit detects a lost packet and additive increase begins. For example, fast recovery avoids the slow start period between 3.8 and 4 seconds in Figure 6.13 and instead simply cuts the congestion window in half (from 22 KB to 11 KB) and resumes additive increase. In other words, slow start is only used at the beginning of a connection and whenever a coarse-grained timeout occurs. At all other times, the congestion window is following a pure additive increase/multiplicative decrease pattern.

6.3.4 TCP CUBIC

A variant of the standard TCP algorithm just described, called CUBIC, is the default congestion control algorithm distributed with Linux. CUBIC's primary goal is to support networks with large delay × bandwidth products, which are sometimes called *long-fat networks*. Such networks suffer from the original TCP algorithm requiring too many round-trips to reach the available capacity of the end-to-end path. CUBIC does this by being more aggressive in how it increases the window size, but of course the trick is to be more aggressive without being so aggressive as to adversely affect other flows.

One important aspect of CUBIC's approach is to adjust its congestion window at regular intervals, based on the amount of time that has elapsed since the last congestion event (e.g., the arrival of a duplicate ACK), rather than only when ACKs arrive (the latter being a function of RTT). This allows CUBIC to behave fairly when competing with short-RTT flows, which will have ACKs arriving more frequently.

The second important aspect of CUBIC is its use of a cubic function to adjust the congestion window. The basic idea is easiest to understand by looking at the general shape of a cubic function, which has three phases: slowing growth, flatten plateau, increasing growth. A generic example is shown in Figure 6.14, which we have annotated with one extra piece of information: the maximum congestion window size achieved just before the last congestion event as a target (denoted W_{max}). The idea is to start fast but slow the growth rate as you get close to W_{max}, be cautious and have near-zero growth when close to W_{max}, and then increase the growth rate as you move away from W_{max}. The latter phase is essentially probing for a new achievable W_{max}.

Specifically, CUBIC computes the congestion window as a function of time (t) since the last congestion event

$$\mathsf{CWND(t)} = \mathsf{C} \times (\mathsf{t} - \mathsf{K})^3 + \mathsf{W}_{max}$$

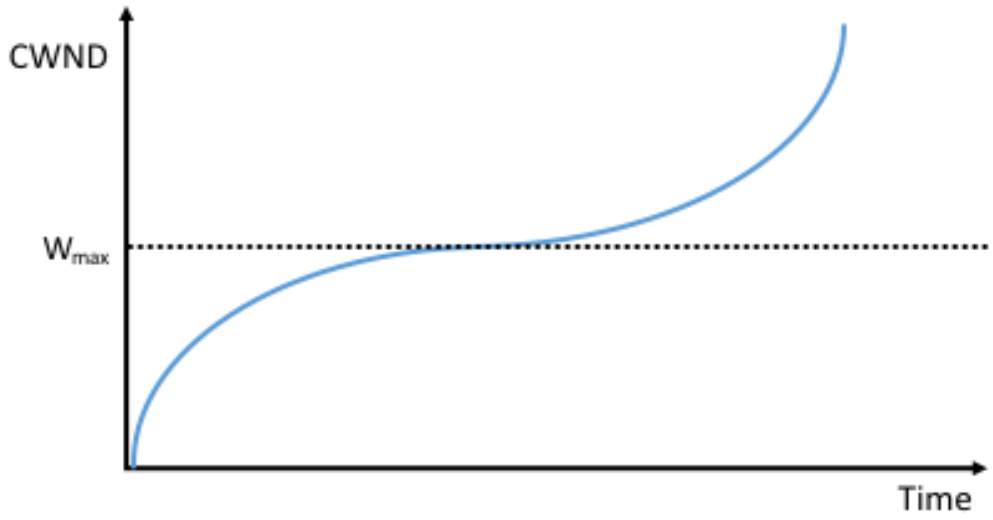


Figure 6.14.: Generic cubic function illustrsting the change in the congestion window as a function of time.

where

$$K = \sqrt[3]{W_{max} \times (1 - \beta)/C}$$

C is a scaling constant and β is the multiplicative decrease factor. CUBIC sets the latter to 0.7 rather than the 0.5 that standard TCP uses. Looking back at Figure 6.14, CUBIC is often described as shifting between a concave function to being convex (whereas standard TCP's additive function is only convex).

6.4 Advanced Congestion Control

This section explores congestion control more deeply. In doing so, it is important to understand that the standard TCP's strategy is to control congestion once it happens, as opposed to trying to avoid congestion in the first place. In fact, TCP repeatedly increases the load it imposes on the network in an effort to find the point at which congestion occurs, and then it backs off from this point. Said another way, TCP *needs* to create losses to find the available bandwidth of the connection. An appealing alternative is to predict when congestion is about to happen and then to reduce the rate at which hosts send data just before packets start being discarded. We call such a strategy *congestion avoidance* to distinguish it from *congestion control*, but it's probably most accurate to think of "avoidance" as a subset of "control."

We describe two different approaches to congestion-avoidance. The first puts a small amount of additional functionality into the router to assist the end node in the anticipation of congestion. This approach is often referred to as *Active Queue Management* (AQM). The second approach attempts to avoid congestion purely from the end hosts. This approach is implemented in TCP, making it variant of the congestion control mechanisms described in the previous section.

6.4.1 Active Queue Management (DECbit, RED, ECN)

The first approach requires changes to routers, which has never been the Internet's preferred way of introducing new features, but nonetheless, has been a constant source of consternation over the last 20 years. The

problem is that while it's generally agreed that routers are in an ideal position to detect the onset of congestion—i.e., their queues start to fill up—there has not been a consensus on exactly what the best algorithm is. The following describes two of the classic mechanisms, and concludes with a brief discussion of where things stand today.

DECbit

The first mechanism was developed for use on the Digital Network Architecture (DNA), a connectionless network with a connection-oriented transport protocol. This mechanism could, therefore, also be applied to TCP and IP. As noted above, the idea here is to more evenly split the responsibility for congestion control between the routers and the end nodes. Each router monitors the load it is experiencing and explicitly notifies the end nodes when congestion is about to occur. This notification is implemented by setting a binary congestion bit in the packets that flow through the router, hence the name *DECbit*. The destination host then copies this congestion bit into the ACK it sends back to the source. Finally, the source adjusts its sending rate so as to avoid congestion. The following discussion describes the algorithm in more detail, starting with what happens in the router.

A single congestion bit is added to the packet header. A router sets this bit in a packet if its average queue length is greater than or equal to 1 at the time the packet arrives. This average queue length is measured over a time interval that spans the last busy+idle cycle, plus the current busy cycle. (The router is *busy* when it is transmitting and *idle* when it is not.) Figure 6.15 shows the queue length at a router as a function of time. Essentially, the router calculates the area under the curve and divides this value by the time interval to compute the average queue length. Using a queue length of 1 as the trigger for setting the congestion bit is a trade-off between significant queuing (and hence higher throughput) and increased idle time (and hence lower delay). In other words, a queue length of 1 seems to optimize the power function.

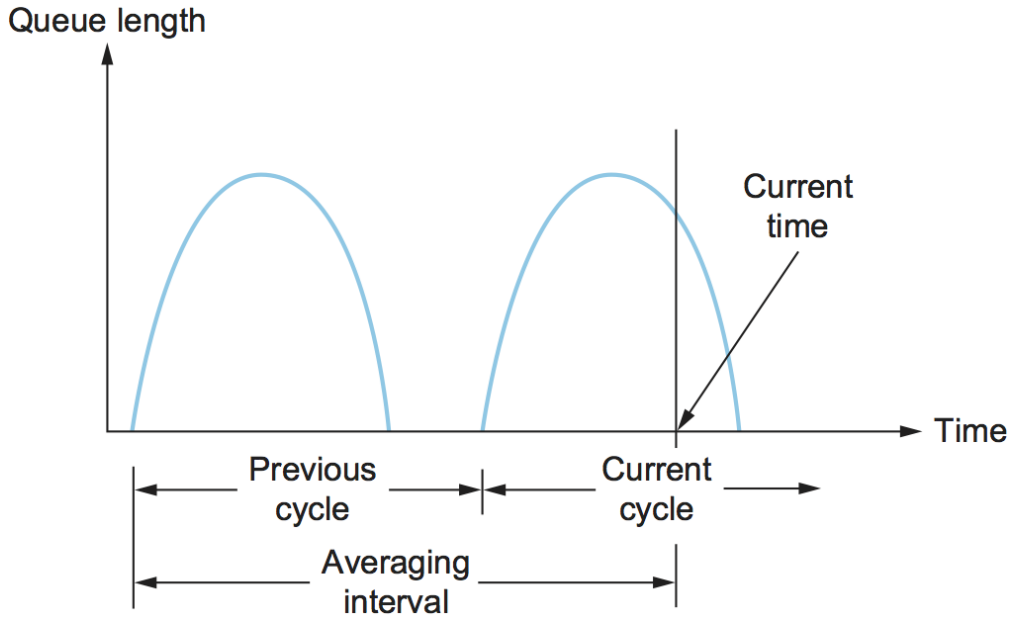


Figure 6.15.: Computing average queue length at a router.

Now turning our attention to the host half of the mechanism, the source records how many of its packets resulted in some router setting the congestion bit. In particular, the source maintains a congestion window,

just as in TCP, and watches to see what fraction of the last window's worth of packets resulted in the bit being set. If less than 50% of the packets had the bit set, then the source increases its congestion window by one packet. If 50% or more of the last window's worth of packets had the congestion bit set, then the source decreases its congestion window to 0.875 times the previous value. The value 50% was chosen as the threshold based on analysis that showed it to correspond to the peak of the power curve. The "increase by 1, decrease by 0.875" rule was selected because additive increase/multiplicative decrease makes the mechanism stable.

Random Early Detection

A second mechanism, called *random early detection* (RED), is similar to the DECbit scheme in that each router is programmed to monitor its own queue length and, when it detects that congestion is imminent, to notify the source to adjust its congestion window. RED, invented by Sally Floyd and Van Jacobson in the early 1990s, differs from the DECbit scheme in two major ways.

The first is that rather than explicitly sending a congestion notification message to the source, RED is most commonly implemented such that it *implicitly* notifies the source of congestion by dropping one of its packets. The source is, therefore, effectively notified by the subsequent timeout or duplicate ACK. In case you haven't already guessed, RED is designed to be used in conjunction with TCP, which currently detects congestion by means of timeouts (or some other means of detecting packet loss such as duplicate ACKs). As the "early" part of the RED acronym suggests, the gateway drops the packet earlier than it would have to, so as to notify the source that it should decrease its congestion window sooner than it would normally have. In other words, the router drops a few packets before it has exhausted its buffer space completely, so as to cause the source to slow down, with the hope that this will mean it does not have to drop lots of packets later on.

The second difference between RED and DECbit is in the details of how RED decides when to drop a packet and what packet it decides to drop. To understand the basic idea, consider a simple FIFO queue. Rather than wait for the queue to become completely full and then be forced to drop each arriving packet (the tail drop policy of the previous section), we could decide to drop each arriving packet with some *drop probability* whenever the queue length exceeds some *drop level*. This idea is called *early random drop*. The RED algorithm defines the details of how to monitor the queue length and when to drop a packet.

In the following paragraphs, we describe the RED algorithm as originally proposed by Floyd and Jacobson. We note that several modifications have since been proposed both by the inventors and by other researchers. However, the key ideas are the same as those presented below, and most current implementations are close to the algorithm that follows.

First, RED computes an average queue length using a weighted running average similar to the one used in the original TCP timeout computation. That is, `AvgLen` is computed as

```
AvgLen = (1 - Weight) x AvgLen + Weight x SampleLen
```

where 0 < `Weight` < 1 and `SampleLen` is the length of the queue when a sample measurement is made. In most software implementations, the queue length is measured every time a new packet arrives at the gateway. In hardware, it might be calculated at some fixed sampling interval.

The reason for using an average queue length rather than an instantaneous one is that it more accurately captures the notion of congestion. Because of the bursty nature of Internet traffic, queues can become full very quickly and then become empty again. If a queue is spending most of its time empty, then it's probably

not appropriate to conclude that the router is congested and to tell the hosts to slow down. Thus, the weighted running average calculation tries to detect long-lived congestion, as indicated in the right-hand portion of Figure 6.16, by filtering out short-term changes in the queue length. You can think of the running average as a low-pass filter, where `Weight` determines the time constant of the filter. The question of how we pick this time constant is discussed below.

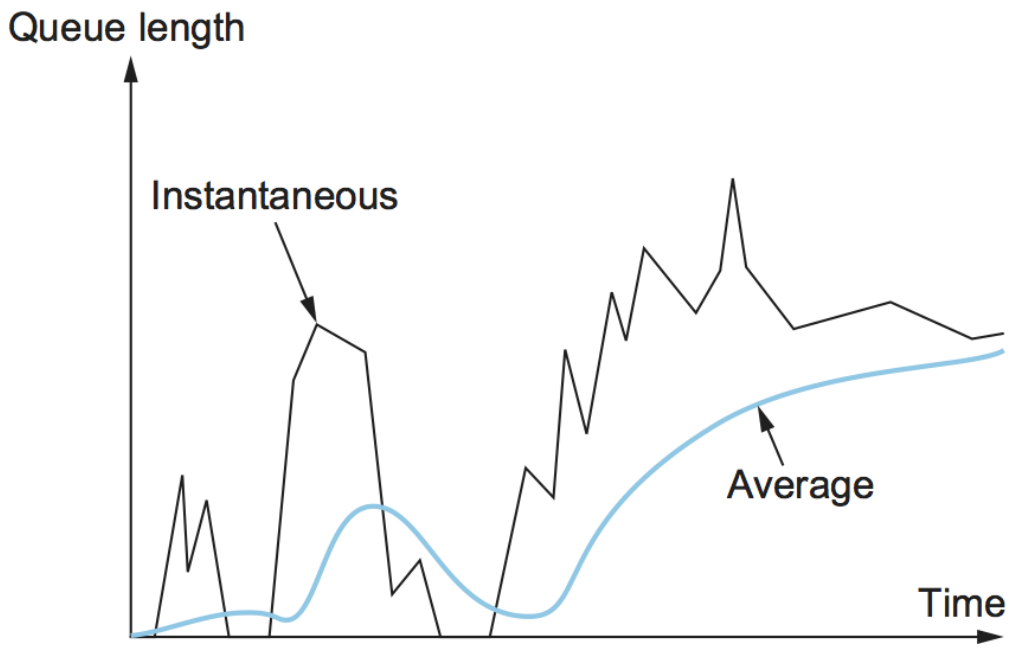


Figure 6.16.: Weighted running average queue length.

Second, RED has two queue length thresholds that trigger certain activity: `MinThreshold` and `MaxThreshold`. When a packet arrives at the gateway, RED compares the current `AvgLen` with these two thresholds, according to the following rules:

```
if AvgLen <= MinThreshold
    queue the packet
if MinThreshold < AvgLen < MaxThreshold
    calculate probability P
    drop the arriving packet with probability P
if MaxThreshold <= AvgLen
    drop the arriving packet
```

If the average queue length is smaller than the lower threshold, no action is taken, and if the average queue length is larger than the upper threshold, then the packet is always dropped. If the average queue length is between the two thresholds, then the newly arriving packet is dropped with some probability `P`. This situation is depicted in Figure 6.17. The approximate relationship between `P` and `AvgLen` is shown in Figure 6.18. Note that the probability of drop increases slowly when `AvgLen` is between the two thresholds, reaching `MaxP` at the upper threshold, at which point it jumps to unity. The rationale behind this is that, if `AvgLen` reaches the upper threshold, then the gentle approach (dropping a few packets) is not working and drastic measures are called for: dropping all arriving packets. Some research has suggested that a smoother transition from random dropping to complete dropping, rather than the discontinuous approach shown here, may be appropriate.

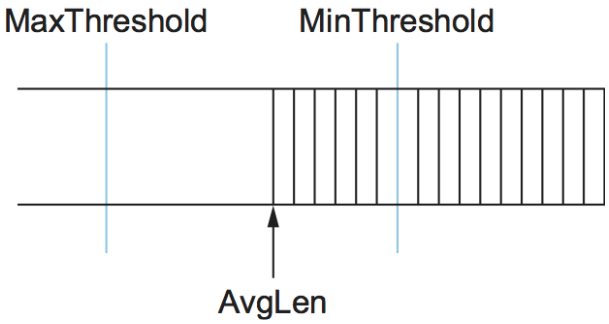


Figure 6.17.: RED thresholds on a FIFO queue.

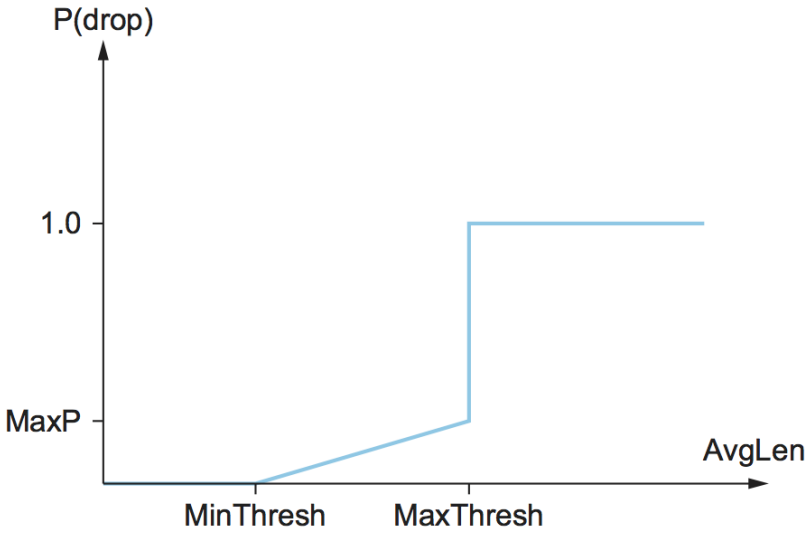


Figure 6.18.: Drop probability function for RED.

Although Figure 6.18 shows the probability of drop as a function only of AvgLen, the situation is actually a little more complicated. In fact, P is a function of both AvgLen and how long it has been since the last packet was dropped. Specifically, it is computed as follows:

```
TempP = MaxP x (AvgLen - MinThreshold) / (MaxThreshold - MinThreshold)
P = TempP/(1 - count x TempP)
```

TempP is the variable that is plotted on the y-axis in Figure 6.18, count keeps track of how many newly arriving packets have been queued (not dropped), and AvgLen has been between the two thresholds. P increases slowly as count increases, thereby making a drop increasingly likely as the time since the last drop increases. This makes closely spaced drops relatively less likely than widely spaced drops. This extra step in calculating P was introduced by the inventors of RED when they observed that, without it, the packet drops were not well distributed in time but instead tended to occur in clusters. Because packet arrivals from a certain connection are likely to arrive in bursts, this clustering of drops is likely to cause multiple drops in a single connection. This is not desirable, since only one drop per round-trip time is enough to cause a connection to reduce its window size, whereas multiple drops might send it back into slow start.

As an example, suppose that we set MaxP to 0.02 and count is initialized to zero. If the average queue length were halfway between the two thresholds, then TempP, and the initial value of P, would be half of MaxP, or 0.01. An arriving packet, of course, has a 99 in 100 chance of getting into the queue at this point. With each successive packet that is not dropped, P slowly increases, and by the time 50 packets have arrived without a drop, P would have doubled to 0.02. In the unlikely event that 99 packets arrived without loss, P reaches 1, guaranteeing that the next packet is dropped. The important thing about this part of the algorithm is that it ensures a roughly even distribution of drops over time.

The intent is that, if RED drops a small percentage of packets when AvgLen exceeds MinThreshold, this will cause a few TCP connections to reduce their window sizes, which in turn will reduce the rate at which packets arrive at the router. All going well, AvgLen will then decrease and congestion is avoided. The queue length can be kept short, while throughput remains high since few packets are dropped.

Note that, because RED is operating on a queue length averaged over time, it is possible for the instantaneous queue length to be much longer than AvgLen. In this case, if a packet arrives and there is nowhere to put it, then it will have to be dropped. When this happens, RED is operating in tail drop mode. One of the goals of RED is to prevent tail drop behavior if possible.

The random nature of RED confers an interesting property on the algorithm. Because RED drops packets randomly, the probability that RED decides to drop a particular flow's packet(s) is roughly proportional to the share of the bandwidth that that flow is currently getting at that router. This is because a flow that is sending a relatively large number of packets is providing more candidates for random dropping. Thus, there is some sense of fair resource allocation built into RED, although it is by no means precise. While arguably fair, because RED punishes high-bandwidth flows more than low-bandwidth flows, it increases the probability of a TCP restart, which is doubly painful for those high-bandwidth flows.

Key Takeaway

Note that a fair amount of analysis has gone into setting the various RED parameters—for example, and Weight—all in the name of optimizing the power function (throughput-to-delay ratio). The performance of these parameters has also been confirmed through simulation, and the algorithm has been shown not to be overly sensitive to them. It is important to keep in mind, however, that all of this analysis and simulation hinges on a particular characterization of the network workload. The real contribution of RED is a mecha-

nism by which the router can more accurately manage its queue length. Defining precisely what constitutes an optimal queue length depends on the traffic mix and is still a subject of research, with real information now being gathered from operational deployment of RED in the Internet. *[Next]*

Consider the setting of the two thresholds, `MinThreshold` and `MaxThreshold`. If the traffic is fairly bursty, then `MinThreshold` should be sufficiently large to allow the link utilization to be maintained at an acceptably high level. Also, the difference between the two thresholds should be larger than the typical increase in the calculated average queue length in one RTT. Setting `MaxThreshold` to twice `MinThreshold` seems to be a reasonable rule of thumb given the traffic mix on today's Internet. In addition, since we expect the average queue length to hover between the two thresholds during periods of high load, there should be enough free buffer space *above* `MaxThreshold` to absorb the natural bursts that occur in Internet traffic without forcing the router to enter tail drop mode.

We noted above that `Weight` determines the time constant for the running average low-pass filter, and this gives us a clue as to how we might pick a suitable value for it. Recall that RED is trying to send signals to TCP flows by dropping packets during times of congestion. Suppose that a router drops a packet from some TCP connection and then immediately forwards some more packets from the same connection. When those packets arrive at the receiver, it starts sending duplicate ACKs to the sender. When the sender sees enough duplicate ACKs, it will reduce its window size. So, from the time the router drops a packet until the time when the same router starts to see some relief from the affected connection in terms of a reduced window size, at least one round-trip time must elapse for that connection. There is probably not much point in having the router respond to congestion on time scales much less than the round-trip time of the connections passing through it. As noted previously, 100 ms is not a bad estimate of average round-trip times in the Internet. Thus, `Weight` should be chosen such that changes in queue length over time scales much less than 100 ms are filtered out.

Since RED works by sending signals to TCP flows to tell them to slow down, you might wonder what would happen if those signals are ignored. This is often called the *unresponsive flow* problem. Unresponsive flows use more than their fair share of network resources and could cause congestive collapse if there were enough of them, just as in the days before TCP congestion control. Some of the techniques described in the next section can help with this problem by isolating certain classes of traffic from others. There is also the possibility that a variant of RED could drop more heavily from flows that are unresponsive to the initial hints that it sends.

Explicit Congestion Notification

RED is the most extensively studied AQM mechanism, but it has not been widely deployed, due in part to the fact that it does not result in ideal behavior in all circumstances. It is, however, the benchmark for understanding AQM behavior. The other thing that came out of RED is the recognition that TCP could do a better job if routers were to send a more explicit congestion signal.

That is, instead of *dropping* a packet and assuming TCP will eventually notice (e.g., due to the arrival of a duplicate ACK), RED (or any AQM algorithm for that matter) can do a better job if it instead *marks* the packet and continues to send it along its way to the destination. This idea was codified in changes to the IP and TCP headers known as *Explicit Congestion Notification* (ECN).

Specifically, this feedback is implemented by treating two bits in the IP `TOS` field as ECN bits. One bit is set by the source to indicate that it is ECN-capable, that is, able to react to a congestion notification. This is

called the ECT bit (ECN-Capable Transport). The other bit is set by routers along the end-to-end path when congestion is encountered, as computed by whatever AQM algorithm it is running. This is called the CE bit (Congestion Encountered).

In addition to these two bits in the IP header (which are transport-agnostic), ECN also includes the addition of two optional flags to the TCP header. The first, ECE (ECN-Echo), communicates from the receiver to the sender that it has received a packet with the CE bit set. The second, CWR (Congestion Window Reduced) communicates from the sender to the receiver that it has reduced the congestion window.

While ECN is now the standard interpretation of two of the eight bits in the TOS field of the IP header and support for ECN is highly recommended, it is not required. Moreover, there is no single recommended AQM algorithm, but instead, there is a list of requirements a good AQM algorithm should meet. Like TCP congestion control algorithms, every AQM algorithm has its advantages and disadvantages, and so we need a lot of them. There is one particular scenario, however, where the TCP congestion control algorithm and AQM algorithm are designed to work in concert: the datacenter. We return to this use case at the end of this section.

6.4.2 Source-Based Approaches (Vegas, BBR, DCTCP)

Unlike the previous congestion-avoidance schemes, which depended on cooperation from routers, we now describe a strategy for detecting the incipient stages of congestion—before losses occur—from the end hosts. We first give a brief overview of a collection of related mechanisms that use different information to detect the early stages of congestion, and then we describe two specific mechanisms in more detail.

The general idea of these techniques is to watch for a sign from the network that some router's queue is building up and that congestion will happen soon if nothing is done about it. For example, the source might notice that as packet queues build up in the network's routers, there is a measurable increase in the RTT for each successive packet it sends. One particular algorithm exploits this observation as follows: The congestion window normally increases as in TCP, but every two round-trip delays the algorithm checks to see if the current RTT is greater than the average of the minimum and maximum RTTs seen so far. If it is, then the algorithm decreases the congestion window by one-eighth.

A second algorithm does something similar. The decision as to whether or not to change the current window size is based on changes to both the RTT and the window size. The window is adjusted once every two round-trip delays based on the product

```
(CurrentWindow - OldWindow) x (CurrentRTT - OldRTT)
```

If the result is positive, the source decreases the window size by one-eighth; if the result is negative or 0, the source increases the window by one maximum packet size. Note that the window changes during every adjustment; that is, it oscillates around its optimal point.

Another change seen as the network approaches congestion is the flattening of the sending rate. A third scheme takes advantage of this fact. Every RTT, it increases the window size by one packet and compares the throughput achieved to the throughput when the window was one packet smaller. If the difference is less than one-half the throughput achieved when only one packet was in transit—as was the case at the beginning of the connection—the algorithm decreases the window by one packet. This scheme calculates the throughput by dividing the number of bytes outstanding in the network by the RTT.

TCP Vegas

The mechanism we are going to describe in more detail is similar to the last algorithm in that it looks at changes in the throughput rate or, more specifically, changes in the sending rate. However, it differs from the previous algorithm in the way it calculates throughput, and instead of looking for a change in the slope of the throughput it compares the measured throughput rate with an expected throughput rate. The algorithm, TCP Vegas, is not widely deployed in the Internet today, but the strategy it uses has been adopted by other implementations that are now being deployed.

The intuition behind the Vegas algorithm can be seen in the trace of standard TCP given in Figure 6.19. The top graph shown in Figure 6.19 traces the connection's congestion window; it shows the same information as the traces given earlier in this section. The middle and bottom graphs depict new information: The middle graph shows the average sending rate as measured at the source, and the bottom graph shows the average queue length as measured at the bottleneck router. All three graphs are synchronized in time. In the period between 4.5 and 6.0 seconds (shaded region), the congestion window increases (top graph). We expect the observed throughput to also increase, but instead it stays flat (middle graph). This is because the throughput cannot increase beyond the available bandwidth. Beyond this point, any increase in the window size only results in packets taking up buffer space at the bottleneck router (bottom graph).

A useful metaphor that describes the phenomenon illustrated in Figure 6.19 is driving on ice. The speedometer (congestion window) may say that you are going 30 miles an hour, but by looking out the car window and seeing people pass you on foot (measured sending rate) you know that you are going no more than 5 miles an hour. The extra energy is being absorbed by the car's tires (router buffers).

TCP Vegas uses this idea to measure and control the amount of extra data this connection has in transit, where by "extra data" we mean data that the source would not have transmitted had it been trying to match exactly the available bandwidth of the network. The goal of TCP Vegas is to maintain the "right" amount of extra data in the network. Obviously, if a source is sending too much extra data, it will cause long delays and possibly lead to congestion. Less obviously, if a connection is sending too little extra data, it cannot respond rapidly enough to transient increases in the available network bandwidth. TCP Vegas's congestion-avoidance actions are based on changes in the estimated amount of extra data in the network, not only on dropped packets. We now describe the algorithm in detail.

First, define a given flow's `BaseRTT` to be the RTT of a packet when the flow is not congested. In practice, TCP Vegas sets `BaseRTT` to the minimum of all measured round-trip times; it is commonly the RTT of the first packet sent by the connection, before the router queues increase due to traffic generated by this flow. If we assume that we are not overflowing the connection, then the expected throughput is given by

```
ExpectedRate = CongestionWindow / BaseRTT
```

where `CongestionWindow` is the TCP congestion window, which we assume (for the purpose of this discussion) to be equal to the number of bytes in transit.

Second, TCP Vegas calculates the current sending rate, `ActualRate`. This is done by recording the sending time for a distinguished packet, recording how many bytes are transmitted between the time that packet is sent and when its acknowledgment is received, computing the sample RTT for the distinguished packet when its acknowledgment arrives, and dividing the number of bytes transmitted by the sample RTT. This calculation is done once per round-trip time.

Third, TCP Vegas compares `ActualRate` to `ExpectedRate` and adjusts the window accordingly. We let `Diff = ExpectedRate - ActualRate`. Note that `Diff` is positive or 0 by definition, since

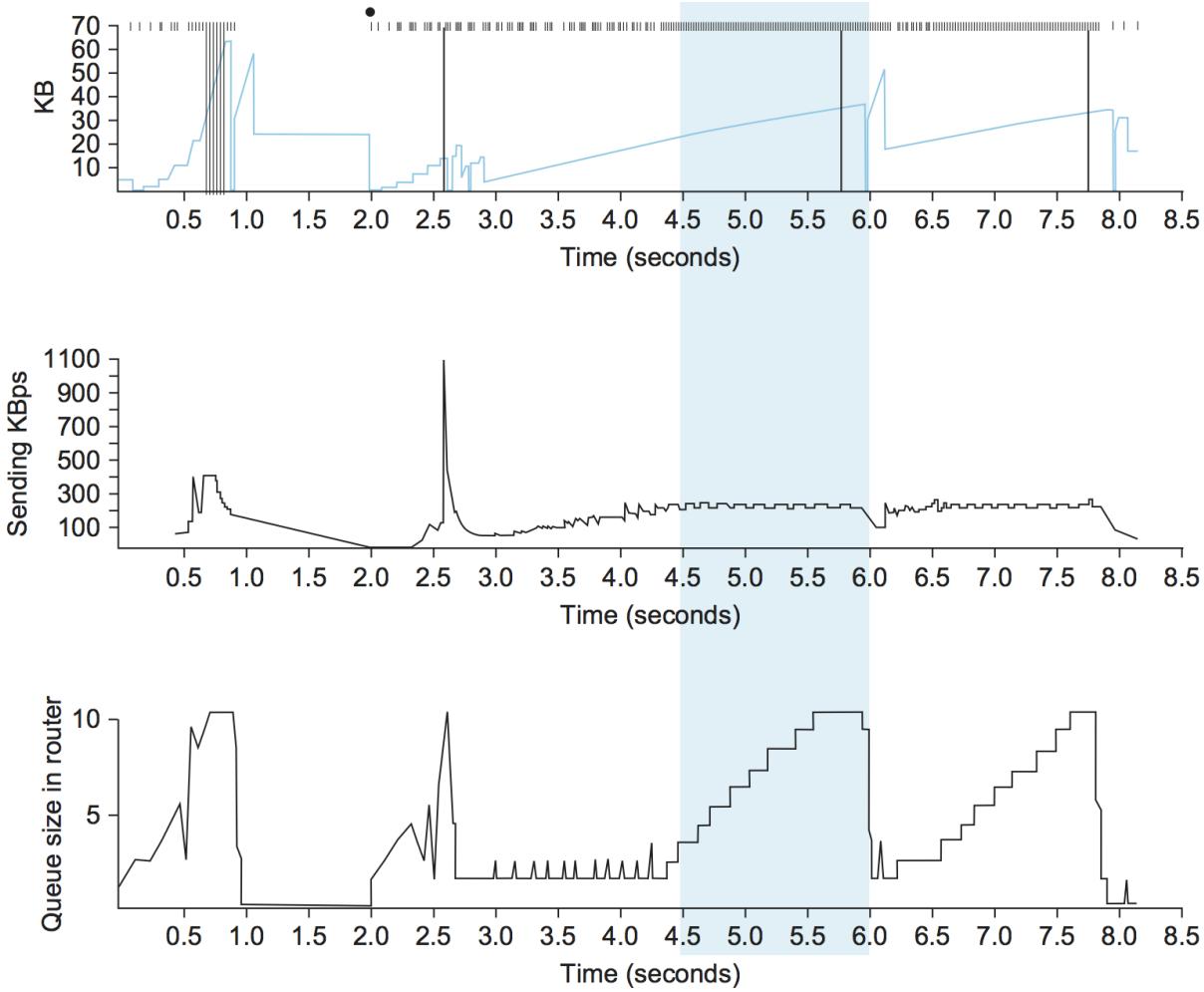


Figure 6.19.: Congestion window versus observed throughput rate (the three graphs are synchronized). Top, congestion window; middle, observed throughput; bottom, buffer space taken up at the router. Colored line = *CongestionWindow*; solid bullet = timeout; hash marks = time when each packet is transmitted; vertical bars = time when a packet that was eventually retransmitted was first transmitted.

`ActualRate >ExpectedRate` implies that we need to change `BaseRTT` to the latest sampled RTT. We also define two thresholds, $\alpha < \beta$, roughly corresponding to having too little and too much extra data in the network, respectively. When `Diff` $< \alpha$, TCP Vegas increases the congestion window linearly during the next RTT, and when `Diff` $> \beta$, TCP Vegas decreases the congestion window linearly during the next RTT. TCP Vegas leaves the congestion window unchanged when $\alpha <$ `Diff` $< \beta$.

Intuitively, we can see that the farther away the actual throughput gets from the expected throughput, the more congestion there is in the network, which implies that the sending rate should be reduced. The β threshold triggers this decrease. On the other hand, when the actual throughput rate gets too close to the expected throughput, the connection is in danger of not utilizing the available bandwidth. The α threshold triggers this increase. The overall goal is to keep betweenα and β extra bytes in the network.

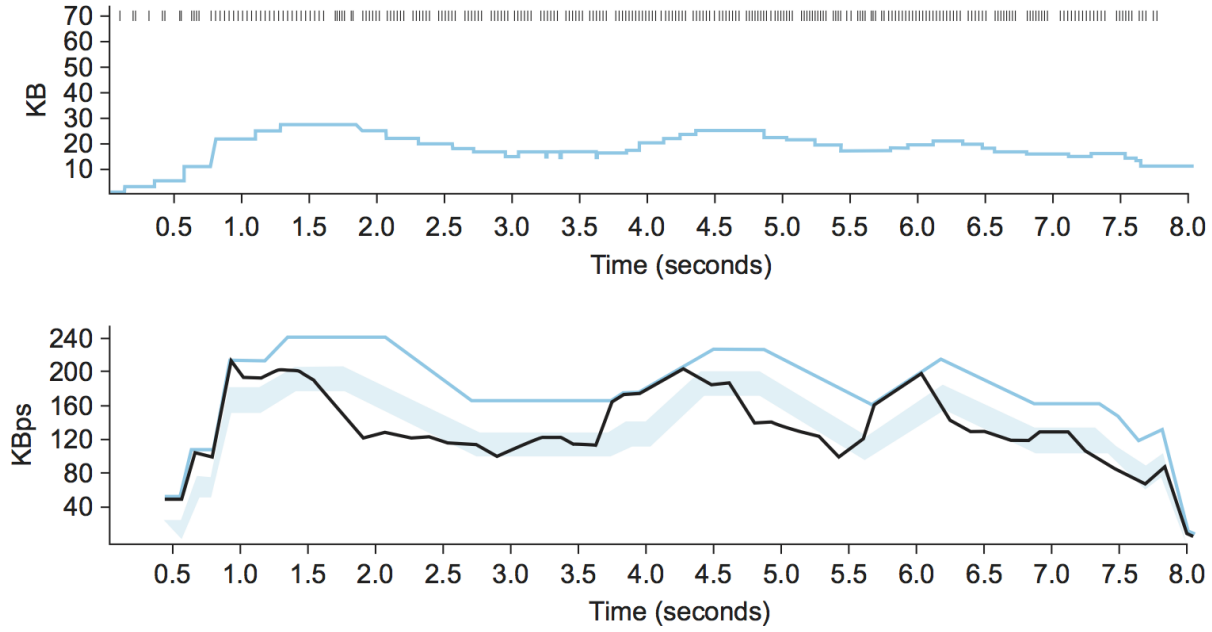


Figure 6.20.: Trace of TCP Vegas congestion-avoidance mechanism. Top, congestion window; bottom, expected (colored line) and actual (black line) throughput. The shaded area is the region between the α and β thresholds.

Figure 6.20 traces the TCP Vegas congestion-avoidance algorithm. The top graph traces the congestion window, showing the same information as the other traces given throughout this chapter. The bottom graph traces the expected and actual throughput rates that govern how the congestion window is set. It is this bottom graph that best illustrates how the algorithm works. The colored line tracks the `ExpectedRate`, while the black line tracks the `ActualRate`. The wide shaded strip gives the region between the α and β thresholds; the top of the shaded strip is α KBps away from `ExpectedRate`, and the bottom of the shaded strip is β KBps away from `ExpectedRate`. The goal is to keep the `ActualRate` between these two thresholds, within the shaded region. Whenever `ActualRate` falls below the shaded region (i.e., gets too far from `ExpectedRate`), TCP Vegas decreases the congestion window because it fears that too many packets are being buffered in the network. Likewise, whenever `ActualRate` goes above the shaded region (i.e., gets too close to the `ExpectedRate`), TCP Vegas increases the congestion window because it fears that it is underutilizing the network.

Because the algorithm, as just presented, compares the difference between the actual and expected throughput rates to the α and β thresholds, these two thresholds are defined in terms of KBps. However, it is perhaps

more accurate to think in terms of how many extra *buffers* the connection is occupying in the network. For example, on a connection with a `BaseRTT` of 100 ms and a packet size of 1 KB, if α = 30 KBps and β = 60 KBps, then we can think of α as specifying that the connection needs to be occupying at least 3 extra buffers in the network and β as specifying that the connection should occupy no more than 6 extra buffers in the network. In practice, a setting of α to 1 buffer and β to 3 buffers works well.

Finally, you will notice that TCP Vegas decreases the congestion window linearly, seemingly in conflict with the rule that multiplicative decrease is needed to ensure stability. The explanation is that TCP Vegas does use multiplicative decrease when a timeout occurs; the linear decrease just described is an *early* decrease in the congestion window that should happen before congestion occurs and packets start being dropped.

TCP BBR

BBR (Bottleneck Bandwidth and RTT) is a new TCP congestion control algorithm developed by researchers at Google. Like Vegas, BBR is delay based, which means it tries to detect buffer growth so as to avoid congestion and packet loss. Both BBR and Vegas use the minimum RTT and maximum RTT, as calculated over some time interval, as their main control signals.

BBR also introduces new mechanisms to improve performance, including packet pacing, bandwidth probing, and RTT probing. Packet pacing spaces the packets based on the estimate of the available bandwidth. This eliminates bursts and unnecessary queueing, which results in a better feedback signal. BBR also periodically increases its rate, thereby probing the available bandwidth. Similarly, BBR periodically decreases its rate, thereby probing for a new minimum RTT. The RTT probing mechanism attempts to be self-synchronizing, which is to say, when there are multiple BBR flows, their respective RTT probes happen at the same time. This gives a more accurate view of the actual uncongested path RTT, which solves one of the major issues with delay-based congestion control mechanisms: having accurate knowledge of the uncongested path RTT.

BBR is actively being worked on and rapidly evolving. One major focus is fairness. For example, some experiments show CUBIC flows get 100× less bandwidth when competing with BBR flows, and other experiments show that unfairness among BBR flows is even possible. Another major focus is avoiding high retransmission rates, where in some cases as many as 10% of packets are retransmitted.

DCTCP

We conclude with an example of a situation where a variant of the TCP congestion control algorithm has been designed to work in concert with ECN: in cloud datacenters. The combination is called DCTCP, which stands for *Data Center TCP*. The situation is unique in that a datacenter is self-contained, and so it is possible to deploy a tailor-made version of TCP that does not need to worry about treating other TCP flows fairly. Datacenters are also unique in that they are built using low-cost white-box switches, and because there is no need to worry about long-fat pipes spanning a continent, the switches are typically provisioned without an excess of buffers.

The idea is straightforward. DCTCP adapts ECN by estimating the fraction of bytes that encounter congestion rather than simply detecting that some congestion is about to occur. At the end hosts, DCTCP then scales the congestion window based on this estimate. The standard TCP algorithm still kicks in should a packet actually be lost. The approach is designed to achieve high-burst tolerance, low latency, and high throughput with shallow-buffered switches.

The key challenge DCTCP faces is to estimate the fraction of bytes encountering congestion. Each switch is simple. If a packet arrives and the switch sees the queue length (K) is above some threshold; e.g.,

$$K > (RTT \times C)/7$$

where C is the link rate in packets per second, then the switch sets the CE bit in the IP header. The complexity of RED is not required.

The receiver then maintains a boolean variable for every flow, which we'll denote `SeenCE`, and implements the following state machine in response to every received packet:

- If the CE bit is set and `SeenCE=False`, set `SeenCE` to True and send an immediate ACK.
- If the CE bit is not set and `SeenCE=True`, set `SeenCE` to False and send an immediate ACK.
- Otherwise, ignore the CE bit.

The non-obvious consequence of the "otherwise" case is that the receiver continues to send delayed ACKs once every *n* packets, whether or not the CE bit is set. This has proven important to maintaining high performance.

Finally, the sender computes the fraction of bytes that encountered congestion during the previous observation window (usually chosen to be approximately the RTT), as the ratio of the total bytes transmitted and the bytes acknowledged with the ECE flag set. DCTCP grows the congestion window in exactly the same way as the standard algorithm, but it reduces the window in proportion to how many bytes encountered congestion during the last observation window.

6.5 Quality of Service

The promise of general-purpose packet-switched networks is that they support all kinds of applications and data, including multimedia applications that transmit digitized audio and video streams. In the early days, one obstacle to the fulfillment of this promise was the need for higher-bandwidth links. That is no longer an issue, but there is more to transmitting audio and video over a network than just providing sufficient bandwidth.

Participants in a telephone conversation, for example, expect to be able to converse in such a way that one person can respond to something said by the other and be heard almost immediately. Thus, the timeliness of delivery can be very important. We refer to applications that are sensitive to the timeliness of data as *real-time applications*. Voice and video applications tend to be the canonical examples, but there are others such as industrial control—you would like a command sent to a robot arm to reach it before the arm crashes into something. Even file transfer applications can have timeliness constraints, such as a requirement that a database update complete overnight before the business that needs the data resumes on the next day.

The distinguishing characteristic of real-time applications is that they need some sort of assurance *from the network* that data is likely to arrive on time (for some definition of "on time"). Whereas a non-real-time application can use an end-to-end retransmission strategy to make sure that data arrives *correctly*, such a strategy cannot provide timeliness: Retransmission only adds to total latency if data arrives late. Timely arrival must be provided by the network itself (the routers), not just at the network edges (the hosts). We therefore conclude that the best-effort model, in which the network tries to deliver your data but makes no promises and leaves the cleanup operation to the edges, is not sufficient for real-time applications. What we need is a new service model, in which applications that need higher assurances can ask the network for them.

The network may then respond by providing an assurance that it will do better or perhaps by saying that it cannot promise anything better at the moment. Note that such a service model is a superset of the original model: Applications that are happy with best-effort service should be able to use the new service model; their requirements are just less stringent. This implies that the network will treat some packets differently from others—something that is not done in the best-effort model. A network that can provide these different levels of service is often said to support quality of service (QoS).

6.5.1 Application Requirements

Before looking at the various protocols and mechanisms that may be used to provide quality of service to applications, we should try to understand what the needs of those applications are. To begin, we can divide applications into two types: real-time and non-real-time. The latter are sometimes called *traditional data* applications, since they have traditionally been the major applications found on data networks. They include most popular applications like SSH, file transfer, email, web browsing, and so on. All of these applications can work without guarantees of timely delivery of data. Another term for this non-real-time class of applications is *elastic*, since they are able to stretch gracefully in the face of increased delay. Note that these applications can benefit from shorter-length delays, but they do not become unusable as delays increase. Also note that their delay requirements vary from the interactive applications like SSH to more asynchronous ones like email, with interactive bulk transfers like file transfer in the middle.



Figure 6.21.: An audio application.

Real-Time Audio Example

As a concrete example of a real-time application, consider an audio application similar to the one illustrated in Figure 6.21. Data is generated by collecting samples from a microphone and digitizing them using an analog-to-digital (A-to-D) converter. The digital samples are placed in packets, which are transmitted across the network and received at the other end. At the receiving host, the data must be *played back* at some appropriate rate. For example, if the voice samples were collected at a rate of one per 125 μs, they should be played back at the same rate. Thus, we can think of each sample as having a particular *playback time*: the point in time at which it is needed in the receiving host. In the voice example, each sample has a playback time that is 125 μs later than the preceding sample. If data arrives after its appropriate playback time, either because it was delayed in the network or because it was dropped and subsequently retransmitted, it is essentially useless. It is the complete worthlessness of late data that characterizes real-time applications. In elastic applications, it might be nice if data turns up on time, but we can still use it when it does not.

One way to make our voice application work would be to make sure all samples take exactly the same amount of time to traverse the network. Then, since samples are injected at a rate of one per 125 μs, they will appear at the receiver at the same rate, ready to be played back. However, it is generally difficult to

guarantee that all data traversing a packet-switched network will experience exactly the same delay. Packets encounter queues in switches or routers, and the lengths of these queues vary with time, meaning that the delays tend to vary with time and, as a consequence, are potentially different for each packet in the audio stream. The way to deal with this at the receiver end is to buffer up some amount of data in reserve, thereby always providing a store of packets waiting to be played back at the right time. If a packet is delayed a short time, it goes in the buffer until its playback time arrives. If it gets delayed a long time, then it will not need to be stored for very long in the receiver's buffer before being played back. Thus, we have effectively added a constant offset to the playback time of all packets as a form of insurance. We call this offset the *playback point*. The only time we run into trouble is if packets get delayed in the network for such a long time that they arrive after their playback time, causing the playback buffer to be drained.

The operation of a playback buffer is illustrated in Figure 6.22. The left-hand diagonal line shows packets being generated at a steady rate. The wavy line shows when the packets arrive, some variable amount of time after they were sent, depending on what they encountered in the network. The right-hand diagonal line shows the packets being played back at a steady rate, after sitting in the playback buffer for some period of time. As long as the playback line is far enough to the right in time, the variation in network delay is never noticed by the application. However, if we move the playback line a little to the left, then some packets will begin to arrive too late to be useful.

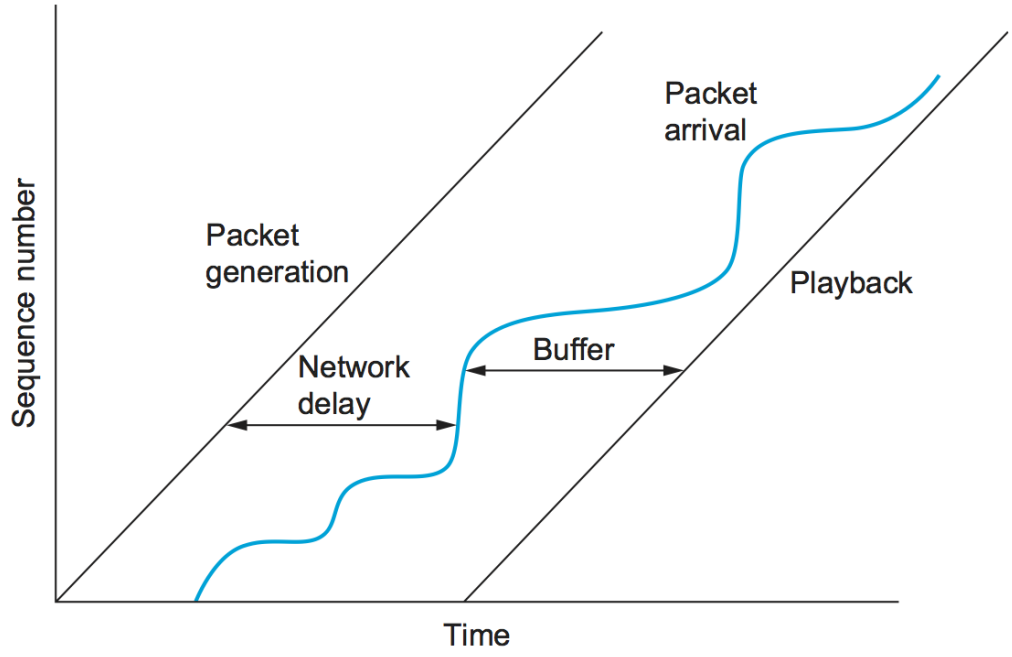


Figure 6.22.: A playback buffer.

For our audio application, there are limits to how far we can delay playing back data. It is hard to carry on a conversation if the time between when you speak and when your listener hears you is more than 300 ms. Thus, what we want from the network in this case is a guarantee that all our data will arrive within 300 ms. If data arrives early, we buffer it until its correct playback time. If it arrives late, we have no use for it and must discard it.

To get a better appreciation of how variable network delay can be, Figure 6.23 shows the one-way delay measured over a certain path across the Internet over the course of one particular day. While the exact numbers would vary depending on the path and the date, the key factor here is the *variability* of the delay,

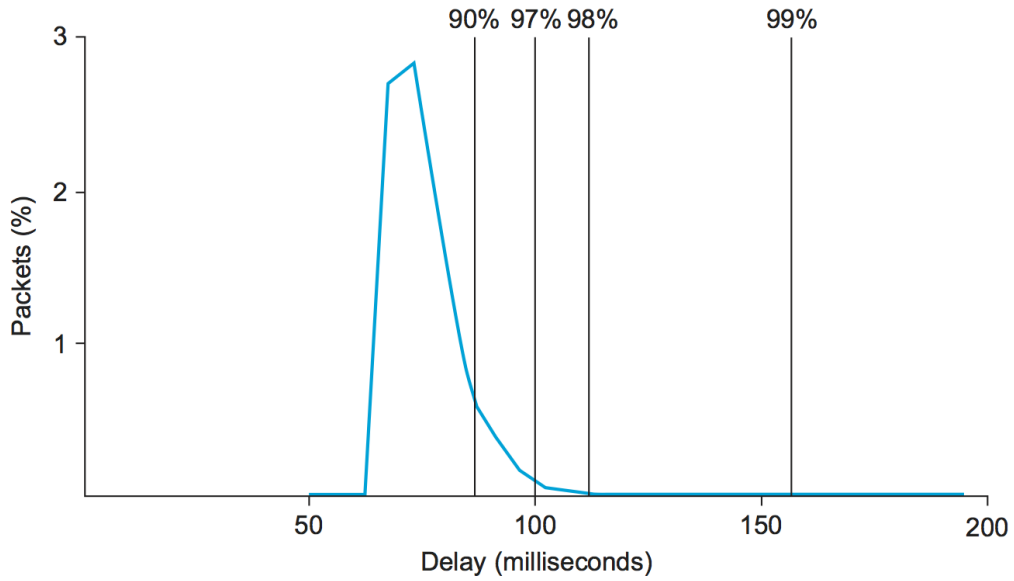


Figure 6.23.: Example distribution of delays for an Internet connection.

which is consistently found on almost any path at any time. As denoted by the cumulative percentages given across the top of the graph, 97% of the packets in this case had a latency of 100 ms or less. This means that if our example audio application were to set the playback point at 100 ms, then, on average, 3 out of every 100 packets would arrive too late to be of any use. One important thing to notice about this graph is that the tail of the curve—how far it extends to the right—is very long. We would have to set the playback point at over 200 ms to ensure that all packets arrived in time.

Taxonomy of Real-Time Applications

Now that we have a concrete idea of how real-time applications work, we can look at some different classes of applications that serve to motivate our service model. The following taxonomy owes much to the work of Clark, Braden, Shenker, and Zhang, whose papers on this subject can be found in the Further Reading section for this chapter. The taxonomy of applications is summarized in Figure 6.24.

The first characteristic by which we can categorize applications is their tolerance of loss of data, where "loss" might occur because a packet arrived too late to be played back as well as arising from the usual causes in the network. On the one hand, one lost audio sample can be interpolated from the surrounding samples with relatively little effect on the perceived audio quality. It is only as more and more samples are lost that quality declines to the point that the speech becomes incomprehensible. On the other hand, a robot control program is likely to be an example of a real-time application that cannot tolerate loss—losing the packet that contains the command instructing the robot arm to stop is unacceptable. Thus, we can categorize real-time applications as *tolerant* or *intolerant* depending on whether they can tolerate occasional loss. (As an aside, note that many real-time applications are more tolerant of occasional loss than non-real-time applications; for example, compare our audio application to file transfer, where the uncorrected loss of one bit might render a file completely useless.)

A second way to characterize real-time applications is by their adaptability. For example, an audio application might be able to adapt to the amount of delay that packets experience as they traverse the network. If we notice that packets are almost always arriving within 300 ms of being sent, then we can set our play-

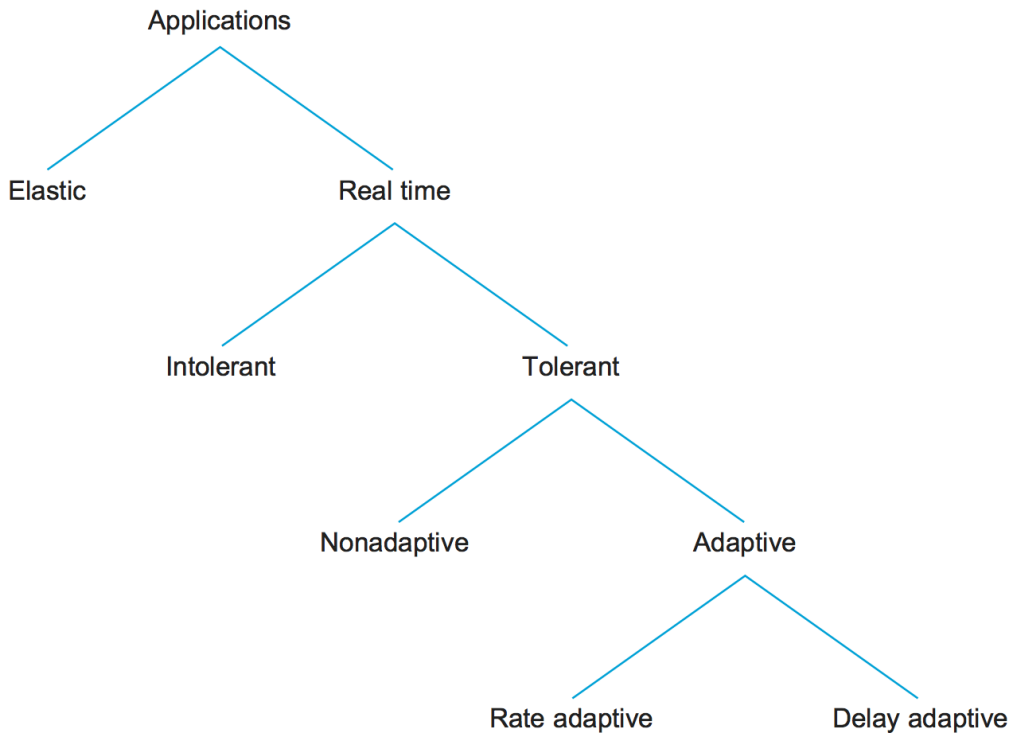


Figure 6.24.: Taxonomy of applications.

back point accordingly, buffering any packets that arrive in less than 300 ms. Suppose that we subsequently observe that all packets are arriving within 100 ms of being sent. If we moved up our playback point to 100 ms, then the users of the application would probably perceive an improvement. The process of shifting the playback point would actually require us to play out samples at an increased rate for some period of time. With a voice application, this can be done in a way that is barely perceptible, simply by shortening the silences between words. Thus, playback point adjustment is fairly easy in this case, and it has been effectively implemented for several voice applications such as the audio teleconferencing program known as `vat`. Note that playback point adjustment can happen in either direction, but that doing so actually involves distorting the played-back signal during the period of adjustment, and that the effects of this distortion will very much depend on how the end user uses the data.

Observe that if we set our playback point on the assumption that all packets will arrive within 100 ms and then find that some packets are arriving slightly late, we will have to drop them, whereas we would not have had to drop them if we had left the playback point at 300 ms. Thus, we should advance the playback point only when it provides a perceptible advantage and only when we have some evidence that the number of late packets will be acceptably small. We may do this because of observed recent history or because of some assurance from the network.

We call applications that can adjust their playback point *delay-adaptive* applications. Another class of adaptive applications is *rate adaptive*. For example, many video coding algorithms can trade off bit rate versus quality. Thus, if we find that the network can support a certain bandwidth, we can set our coding parameters accordingly. If more bandwidth becomes available later, we can change parameters to increase the quality.

Approaches to QoS Support

Considering this rich space of application requirements, what we need is a richer service model that meets the needs of any application. This leads us to a service model with not just one class (best effort), but with several classes, each available to meet the needs of some set of applications. Towards this end, we are now ready to look at some of the approaches that have been developed to provide a range of qualities of service. These can be divided into two broad categories:

- *Fine-grained* approaches, which provide QoS to individual applications or flows
- *Coarse-grained* approaches, which provide QoS to large classes of data or aggregated traffic

In the first category, we find *Integrated Services*, a QoS architecture developed in the IETF and often associated with the Resource Reservation Protocol (RSVP). In the second category lies *Differentiated Services*, which is probably the most widely deployed QoS mechanism today. We discuss these in turn in the next two subsections.

Finally, as we suggested at the start of this section, adding QoS support to the network isn't necessarily the entire story about supporting real-time applications. We conclude our discussion by revisiting what the end-host might do to better support real-time streams, independent of how widely deployed QoS mechanisms like Integrated or Differentiated Services become.

6.5.2 Integrated Services (RSVP)

The term *Integrated Services* (often called IntServ for short) refers to a body of work that was produced by the IETF around 1995-97. The IntServ working group developed specifications of a number of *service classes* designed to meet the needs of some of the application types described above. It also defined how RSVP could be used to make reservations using these service classes. The following paragraphs provide an overview of these specifications and the mechanisms that are used to implement them.

Service Classes

One of the service classes is designed for intolerant applications. These applications require that a packet never arrive late. The network should guarantee that the maximum delay that any packet will experience has some specified value; the application can then set its playback point so that no packet will ever arrive after its playback time. We assume that early arrival of packets can always be handled by buffering. This service is referred to as the *guaranteed* service.

In addition to the guaranteed service, the IETF considered several other services, but eventually settled on one to meet the needs of tolerant, adaptive applications. The service is known as *controlled load* and was motivated by the observation that existing applications of this type run quite well on networks that are not heavily loaded. Audio applications, for example, adjust their playback point as network delay varies and produces reasonable audio quality as long as loss rates remain on the order of 10% or less.

The aim of the controlled load service is to emulate a lightly loaded network for those applications that request the service, even though the network as a whole may in fact be heavily loaded. The trick to this is to use a queuing mechanism such as WFQ to isolate the controlled load traffic from the other traffic and some form of admission control to limit the total amount of controlled load traffic on a link such that the load is kept reasonably low. We discuss admission control in more detail below.

Clearly, these two service classes are a subset of all the classes that might be provided. In fact, other services were specified but never standardized as part of the IETF's work. So far, the two services described above (along with traditional best effort) have proven flexible enough to meet the needs of a wide range of applications.

Overview of Mechanisms

Now that we have augmented our best-effort service model with some new service classes, the next question is how we implement a network that provides these services to applications. This section outlines the key mechanisms. Keep in mind while reading this section that the mechanisms being described are still being hammered out by the Internet design community. The main thing to take away from the discussion is a general understanding of the pieces involved in supporting the service model outlined above.

First, whereas with a best-effort service we can just tell the network where we want our packets to go and leave it at that, a real-time service involves telling the network something more about the type of service we require. We may give it qualitative information such as "use a controlled load service" or quantitative information such as "I need a maximum delay of 100 ms." In addition to describing what we want, we need to tell the network something about what we are going to inject into it, since a low-bandwidth application is going to require fewer network resources than a high-bandwidth application. The set of information that we provide to the network is referred to as a *flowspec*. This name comes from the idea that a set of packets associated with a single application and that share common requirements is called a *flow*, consistent with our use of the term in the earlier section outlining the relevant issues.

Second, when we ask the network to provide us with a particular service, the network needs to decide if it can in fact provide that service. For example, if 10 users ask for a service in which each will consistently use 2 Mbps of link capacity, and they all share a link with 10-Mbps capacity, the network will have to say no to some of them. The process of deciding when to say no is called *admission control*.

Third, we need a mechanism by which the users of the network and the components of the network itself exchange information such as requests for service, flowspecs, and admission control decisions. This is sometimes called *signalling*, but since that word has several meanings, we refer to this process as *resource reservation*, and it is achieved using a resource reservation protocol.

Finally, when flows and their requirements have been described, and admission control decisions have been made, the network switches and routers need to meet the requirements of the flows. A key part of meeting these requirements is managing the way packets are queued and scheduled for transmission in the switches and routers. This last mechanism is *packet scheduling*.

Flowspecs

There are two separable parts to the flowspec: the part that describes the flow's traffic characteristics (called the *TSpec*) and the part that describes the service requested from the network (the *RSpec*). The RSpec is very service specific and relatively easy to describe. For example, with a controlled load service, the RSpec is trivial: The application just requests controlled load service with no additional parameters. With a guaranteed service, you could specify a delay target or bound. (In the IETF's guaranteed service specification, you specify not a delay but another quantity from which delay can be calculated.)

The TSpec is a little more complicated. As our example above showed, we need to give the network enough information about the bandwidth used by the flow to allow intelligent admission control decisions to be

made. For most applications, however, the bandwidth is not a single number; it is something that varies constantly. A video application, for example, will generally generate more bits per second when the scene is changing rapidly than when it is still. Just knowing the long-term average bandwidth is not enough, as the following example illustrates. Suppose that we have 10 flows that arrive at a switch on separate input ports and that all leave on the same 10-Mbps link. Assume that over some suitably long interval each flow can be expected to send no more than 1 Mbps. You might think that this presents no problem. However, if these are variable bit rate applications, such as compressed video, then they will occasionally send more than their average rates. If enough sources send at above their average rates, then the total rate at which data arrives at the switch will be greater than 10 Mbps. This excess data will be queued before it can be sent on the link. The longer this condition persists, the longer the queue will get. Packets might have to be dropped and, even if it doesn't come to that, data sitting in the queue is being delayed. If packets are delayed long enough, the service that was requested will not be provided.

Exactly how we manage our queues to control delay and avoid dropping packets is something we discuss below. However, note here that we need to know something about how the bandwidth of our sources varies with time. One way to describe the bandwidth characteristics of sources is called a *token bucket* filter. Such a filter is described by two parameters: a token rate r, and a bucket depth B. It works as follows. To be able to send a byte, I must have a token. To send a packet of length n, I need n tokens. I start with no tokens and I accumulate them at a rate of r per second. I can accumulate no more than B tokens. What this means is that I can send a burst of as many as B bytes into the network as fast as I want, but over a sufficiently long interval I can't send more than r bytes per second. It turns out that this information is very helpful to the admission control algorithm when it tries to figure out whether it can accommodate a new request for service.

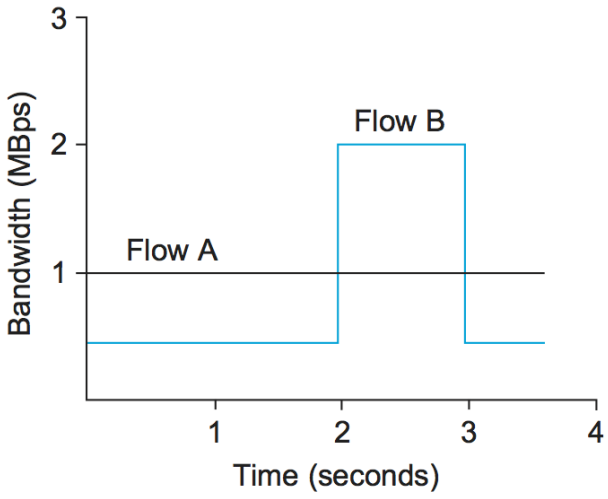


Figure 6.25.: Two flows with equal average rates but different token bucket descriptions.

Figure 6.25 illustrates how a token bucket can be used to characterize a flow's bandwidth requirements. For simplicity, assume that each flow can send data as individual bytes rather than as packets. Flow A generates data at a steady rate of 1 MBps, so it can be described by a token bucket filter with a rate $r = 1$ MBps and a bucket depth of 1 byte. This means that it receives tokens at a rate of 1 MBps but that it cannot store more than 1 token—it spends them immediately. Flow B also sends at a rate that averages out to 1 MBps over the long term, but does so by sending at 0.5 MBps for 2 seconds and then at 2 MBps for 1 second. Since the token bucket rate r is, in a sense, a long-term average rate, flow B can be described by a token bucket with a rate of 1 MBps. Unlike flow A, however, flow B needs a bucket depth B of at least 1 MB, so that it can store up tokens while it sends at less than 1 MBps to be used when it sends at 2 MBps. For the first 2 seconds in

this example, it receives tokens at a rate of 1 MBps but spends them at only 0.5 MBps, so it can save up 2 × 0.5 = 1 MB of tokens, which it then spends in the third second (along with the new tokens that continue to accrue in that second) to send data at 2 MBps. At the end of the third second, having spent the excess tokens, it starts to save them up again by sending at 0.5 MBps again.

It is interesting to note that a single flow can be described by many different token buckets. As a trivial example, flow A could be described by the same token bucket as flow B, with a rate of 1 MBps and a bucket depth of 1 MB. The fact that it never actually needs to accumulate tokens does not make that an inaccurate description, but it does mean that we have failed to convey some useful information to the network—the fact that flow A is actually very consistent in its bandwidth needs. In general, it is good to be as explicit about the bandwidth needs of an application as possible to avoid over-allocation of resources in the network.

Admission Control

The idea behind admission control is simple: When some new flow wants to receive a particular level of service, admission control looks at the TSpec and RSpec of the flow and tries to decide if the desired service can be provided to that amount of traffic, given the currently available resources, without causing any previously admitted flow to receive worse service than it had requested. If it can provide the service, the flow is admitted; if not, then it is denied. The hard part is figuring out when to say yes and when to say no.

Admission control is very dependent on the type of requested service and on the queuing discipline employed in the routers; we discuss the latter topic later in this section. For a guaranteed service, you need to have a good algorithm to make a definitive yes/no decision. The decision is fairly straightforward if weighted fair queuing is used at each router. For a controlled load service, the decision may be based on heuristics, such as "The last time I allowed a flow with this TSpec into this class, the delays for the class exceeded the acceptable bound, so I'd better say no" or "My current delays are so far inside the bounds that I should be able to admit another flow without difficulty."

Admission control should not be confused with *policing*. The former is a per-flow decision to admit a new flow or not. The latter is a function applied on a per-packet basis to make sure that a flow conforms to the TSpec that was used to make the reservation. If a flow does not conform to its TSpec—for example, because it is sending twice as many bytes per second as it said it would—then it is likely to interfere with the service provided to other flows, and some corrective action must be taken. There are several options, the obvious one being to drop offending packets. However, another option would be to check if the packets really are interfering with the service of other flows. If they are not interfering, the packets could be sent on after being marked with a tag that says, in effect, "This is a nonconforming packet. Drop it first if you need to drop any packets."

Admission control is closely related to the important issue of *policy*. For example, a network administrator might wish to allow reservations made by his company's CEO to be admitted while rejecting reservations made by more lowly employees. Of course, the CEO's reservation request might still fail if the requested resources aren't available, so we see that issues of policy and resource availability may both be addressed when admission control decisions are made. The application of policy to networking is an area receiving much attention at the time of writing.

Reservation Protocol

While connection-oriented networks have always needed some sort of setup protocol to establish the necessary virtual circuit state in the switches, connectionless networks like the Internet have had no such protocols. As this section has indicated, however, we need to provide a lot more information to our network when we want a real-time service from it. While there have been a number of setup protocols proposed for the Internet, the one on which most current attention is focused is the RSVP. It is particularly interesting because it differs so substantially from conventional signalling protocols for connection-oriented networks.

One of the key assumptions underlying RSVP is that it should not detract from the robustness that we find in today's connectionless networks. Because connectionless networks rely on little or no state being stored in the network itself, it is possible for routers to crash and reboot and for links to go up and down while end-to-end connectivity is still maintained. RSVP tries to maintain this robustness by using the idea of *soft state* in the routers. Soft state—in contrast to the hard state found in connection-oriented networks—does not need to be explicitly deleted when it is no longer needed. Instead, it times out after some fairly short period (say, a minute) if it is not periodically refreshed. We will see later how this helps robustness.

Another important characteristic of RSVP is that it aims to support multicast flows just as effectively as unicast flows. This is not surprising, since many of the first applications that could benefit from improved quality of service were also multicast applications—video conferenceing tools, for example. One of the insights of RSVP's designers is that most multicast applications have many more receivers than senders, as typified by the large audience and one speaker for a lecture. Also, receivers may have different requirements. For example, one receiver might want to receive data from only one sender, while others might wish to receive data from all senders. Rather than having the senders keep track of a potentially large number of receivers, it makes more sense to let the receivers keep track of their own needs. This suggests the *receiver-oriented* approach adopted by RSVP. In contrast, connection-oriented networks usually leave resource reservation to the sender, just as it is normally the originator of a phone call who causes resources to be allocated in the phone network.

The soft state and receiver-oriented nature of RSVP give it a number of good properties. One such property is that it is very straightforward to increase or decrease the level of resource allocation provided to a receiver. Since each receiver periodically sends refresh messages to keep the soft state in place, it is easy to send a new reservation that asks for a new level of resources. Further, soft state deals gracefully with the possibility of network or node failures. In the event of a host crash, resources allocated by that host to a flow will naturally time out and be released. To see what happens in the event of a router or link failure, we need to look a little more closely at the mechanics of making a reservation.

Initially, consider the case of one sender and one receiver trying to get a reservation for traffic flowing between them. There are two things that need to happen before a receiver can make the reservation. First, the receiver needs to know what traffic the sender is likely to send so that it can make an appropriate reservation. That is, it needs to know the sender's TSpec. Second, it needs to know what path the packets will follow from sender to receiver, so that it can establish a resource reservation at each router on the path. Both of these requirements can be met by sending a message from the sender to the receiver that contains the TSpec. Obviously, this gets the TSpec to the receiver. The other thing that happens is that each router looks at this message (called a PATH message) as it goes past, and it figures out the *reverse path* that will be used to send reservations from the receiver back to the sender in an effort to get the reservation to each router on the path. Building the multicast tree in the first place is done by mechanisms such as those described in another chapter.

Having received a PATH message, the receiver sends a reservation back up the multicast tree in a RESV

message. This message contains the sender's TSpec and an RSpec describing the requirements of this receiver. Each router on the path looks at the reservation request and tries to allocate the necessary resources to satisfy it. If the reservation can be made, the RESV request is passed on to the next router. If not, an error message is returned to the receiver who made the request. If all goes well, the correct reservation is installed at every router between the sender and the receiver. As long as the receiver wants to retain the reservation, it sends the same RESV message about once every 30 seconds.

Now we can see what happens when a router or link fails. Routing protocols will adapt to the failure and create a new path from sender to receiver. PATH messages are sent about every 30 seconds, and may be sent sooner if a router detects a change in its forwarding table, so the first one after the new route stabilizes will reach the receiver over the new path. The receiver's next RESV message will follow the new path and, if all goes well, establish a new reservation on the new path. Meanwhile, the routers that are no longer on the path will stop getting RESV messages, and these reservations will time out and be released. Thus, RSVP deals quite well with changes in topology, as long as routing changes are not excessively frequent.

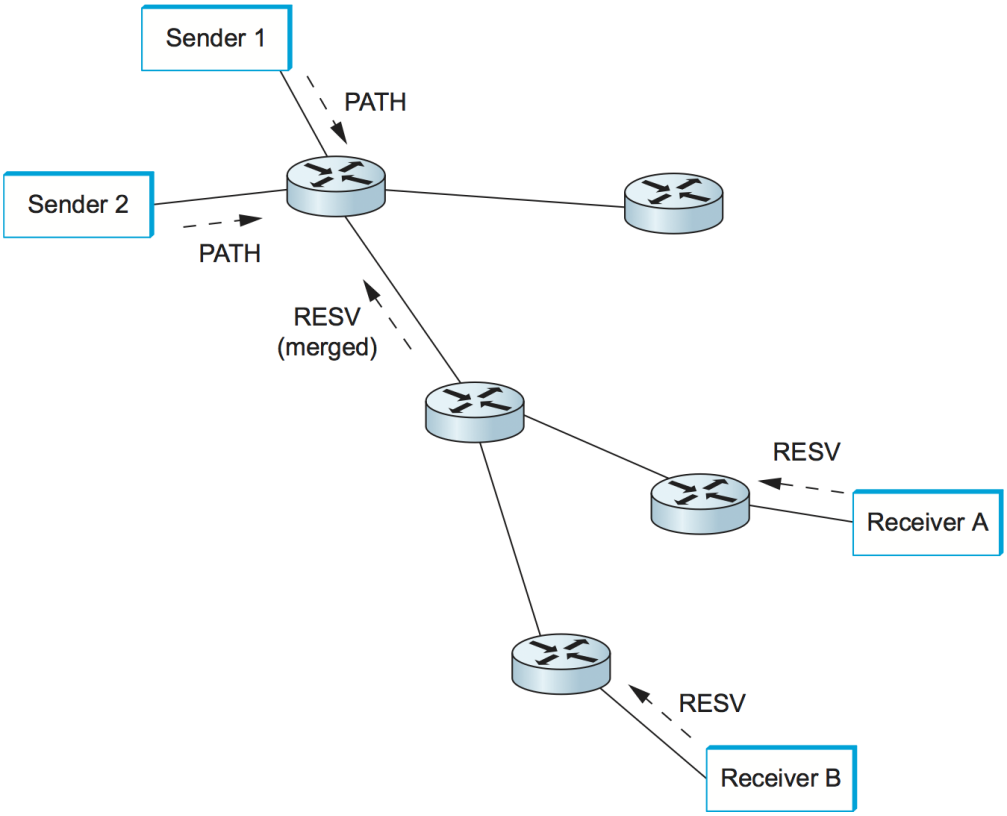


Figure 6.26.: Making reservations on a multicast tree.

The next thing we need to consider is how to cope with multicast, where there may be multiple senders to a group and multiple receivers. This situation is illustrated in Figure 6.26. First, let's deal with multiple receivers for a single sender. As a RESV message travels up the multicast tree, it is likely to hit a piece of the tree where some other receiver's reservation has already been established. It may be the case that the resources reserved upstream of this point are adequate to serve both receivers. For example, if receiver A has already made a reservation that provides for a guaranteed delay of less than 100 ms, and the new request from receiver B is for a delay of less than 200 ms, then no new reservation is required. On the other hand, if the new request were for a delay of less than 50 ms, then the router would first need to see if it could accept

the request; if so, it would send the request on upstream. The next time receiver A asked for a minimum of a 100-ms delay, the router would not need to pass this request on. In general, reservations can be merged in this way to meet the needs of all receivers downstream of the merge point.

If there are also multiple senders in the tree, receivers need to collect the TSpecs from all senders and make a reservation that is large enough to accommodate the traffic from all senders. However, this may not mean that the TSpecs need to be added up. For example, in an audioconference with 10 speakers, there is not much point in allocating enough resources to carry 10 audio streams, since the result of 10 people speaking at once would be incomprehensible. Thus, we could imagine a reservation that is large enough to accommodate two speakers and no more. Calculating the correct overall TSpec from all of the sender TSpecs is clearly application specific. Also, we may only be interested in hearing from a subset of all possible speakers; RSVP has different reservation styles to deal with such options as "Reserve resources for all speakers," "Reserve resources for any n speakers," and "Reserve resources for speakers A and B only."

Packet Classifying and Scheduling

Once we have described our traffic and our desired network service and have installed a suitable reservation at all the routers on the path, the only thing that remains is for the routers to actually deliver the requested service to the data packets. There are two things that need to be done:

- Associate each packet with the appropriate reservation so that it can be handled correctly, a process known as *classifying* packets.
- Manage the packets in the queues so that they receive the service that has been requested, a process known as packet *scheduling*.

The first part is done by examining up to five fields in the packet: the source address, destination address, protocol number, source port, and destination port. (In IPv6, it is possible that the `FlowLabel` field in the header could be used to enable the lookup to be done based on a single, shorter key.) Based on this information, the packet can be placed in the appropriate class. For example, it may be classified into the controlled load classes, or it may be part of a guaranteed flow that needs to be handled separately from all other guaranteed flows. In short, there is a mapping from the flow-specific information in the packet header to a single class identifier that determines how the packet is handled in the queue. For guaranteed flows this might be a one-to-one mapping, while for other services it might be many to one. The details of classification are closely related to the details of queue management.

It should be clear that something as simple as a FIFO queue in a router will be inadequate to provide many different services and to provide different levels of delay within each service. Several more sophisticated queue management disciplines were discussed in an earlier section, and some combination of these is likely to be used in a router.

The details of packet scheduling ideally should not be specified in the service model. Instead, this is an area where implementors can try to do creative things to realize the service model efficiently. In the case of guaranteed service, it has been established that a weighted fair queuing discipline, in which each flow gets its own individual queue with a certain share of the link, will provide a guaranteed end-to-end delay bound that can readily be calculated. For controlled load, simpler schemes may be used. One possibility includes treating all the controlled load traffic as a single, aggregated flow (as far as the scheduling mechanism is concerned), with the weight for that flow being set based on the total amount of traffic admitted in the controlled load class. The problem is made harder when you consider that, in a single router, many different services are likely to be provided concurrently and that each of these services may require a different scheduling

algorithm. Thus, some overall queue management algorithm is needed to manage the resources between the different services.

Scalability Issues

Although the Integrated Services architecture and RSVP represented a significant enhancement of the best-effort service model of IP, many Internet service providers felt that it was not the right model for them to deploy. The reason for this reticence relates to one of the fundamental design goals of IP: scalability. In the best-effort service model, routers in the Internet store little or no state about the individual flows passing through them. Thus, as the Internet grows, the only thing routers have to do to keep up with that growth is to move more bits per second and to deal with larger routing tables, but RSVP raises the possibility that every flow passing through a router might have a corresponding reservation. To understand the severity of this problem, suppose that every flow on an OC-48 (2.5-Gbps) link represents a 64-kbps audio stream. The number of such flows is

$$2.5 \times 10^9 / 64 \times 10^3 = 39{,}000$$

Each of those reservations needs some amount of state that needs to be stored in memory and refreshed periodically. The router needs to classify, police, and queue each of those flows. Admission control decisions need to be made every time such a flow requests a reservation, and some mechanism is needed to "push back" on users (e.g., charge their credit cards) so that they don't make arbitrarily large reservations for long periods of time.

These scalability concerns have limited widespread deployment of IntServ. Because of these concerns, other approaches that do not require so much "per-flow" state have been developed. The next section discusses a number of such approaches.

6.5.3 Differentiated Services (EF, AF)

Whereas the Integrated Services architecture allocates resources to individual flows, the Differentiated Services model (often called DiffServ for short) allocates resources to a small number of classes of traffic. In fact, some proposed approaches to DiffServ simply divide traffic into two classes. This is an eminently sensible approach to take: If you consider the difficulty that network operators experience just trying to keep a best-effort internet running smoothly, it makes sense to add to the service model in small increments.

Suppose that we have decided to enhance the best-effort service model by adding just one new class, which we'll call "premium." Clearly, we will need some way to figure out which packets are premium and which are regular old best effort. Rather than using a protocol like RSVP to tell all the routers that some flow is sending premium packets, it would be much easier if the packets could just identify themselves to the router when they arrive. This could obviously be done by using a bit in the packet header—if that bit is a 1, the packet is a premium packet; if it's a 0, the packet is best effort. With this in mind, there are two questions we need to address:

- Who sets the premium bit and under what circumstances?
- What does a router do differently when it sees a packet with the bit set?

There are many possible answers to the first question, but a common approach is to set the bit at an administrative boundary. For example, the router at the edge of an Internet service provider's network might set the

bit for packets arriving on an interface that connects to a particular company's network. The Internet service provider might do this because that company has paid for a higher level of service than best effort. It is also possible that not all packets would be marked as premium; for example, the router might be configured to mark packets as premium up to some maximum rate and to leave all excess packets as best effort.

Assuming that packets have been marked in some way, what do the routers that encounter marked packets do with them? Here again there are many answers. In fact, the IETF standardized a set of router behaviors to be applied to marked packets. These are called *per-hop behaviors* (PHBs), a term that indicates that they define the behavior of individual routers rather than end-to-end services. Because there is more than one new behavior, there is also a need for more than 1 bit in the packet header to tell the routers which behavior to apply. The IETF decided to take the old `TOS` byte from the IP header, which had not been widely used, and redefine it. Six bits of this byte have been allocated for DiffServ code points (DSCPs), where each DSCP is a 6-bit value that identifies a particular PHB to be applied to a packet. (The remaining two bits are used by ECN.)

The Expedited Forwarding (EF) PHB

One of the simplest PHBs to explain is known as *expedited forwarding* (EF). Packets marked for EF treatment should be forwarded by the router with minimal delay and loss. The only way that a router can guarantee this to all EF packets is if the arrival rate of EF packets at the router is strictly limited to be less than the rate at which the router can forward EF packets. For example, a router with a 100-Mbps interface needs to be sure that the arrival rate of EF packets destined for that interface never exceeds 100 Mbps. It might also want to be sure that the rate will be somewhat below 100 Mbps, so that it occasionally has time to send other packets such as routing updates.

The rate limiting of EF packets is achieved by configuring the routers at the edge of an administrative domain to allow a certain maximum rate of EF packet arrivals into the domain. A simple, albeit conservative, approach would be to ensure that the sum of the rates of all EF packets entering the domain is less than the bandwidth of the slowest link in the domain. This would ensure that, even in the worst case where all EF packets converge on the slowest link, it is not overloaded and can provide the correct behavior.

There are several possible implementation strategies for the EF behavior. One is to give EF packets strict priority over all other packets. Another is to perform weighted fair queuing between EF packets and other packets, with the weight of EF set sufficiently high that all EF packets can be delivered quickly. This has an advantage over strict priority: The non-EF packets can be assured of getting some access to the link, even if the amount of EF traffic is excessive. This might mean that the EF packets fail to get exactly the specified behavior, but it could also prevent essential routing traffic from being locked out of the network in the event of an excessive load of EF traffic.

The Assured Forwarding (AF) PHB

The *assured forwarding* (AF) PHB has its roots in an approach known as *RED* with In and Out (RIO) or Weighted RED, both of which are enhancements to the basic RED algorithm described in an earlier section. Figure 6.27 shows how RIO works; as with RED, we see drop probability on the y-axis increasing as average queue length increases along the x-axis. But now, for our two classes of traffic, we have two separate drop probability curves. RIO calls the two classes "in" and "out" for reasons that will become clear shortly. Because the "out" curve has a lower `MinThreshold` than the "in" curve, it is clear that, under low levels of congestion, only packets marked "out" will be discarded by the RED algorithm. If the

congestion becomes more serious, a higher percentage of "out" packets are dropped, and then if the average queue length exceeds , RED starts to drop "in" packets as well.

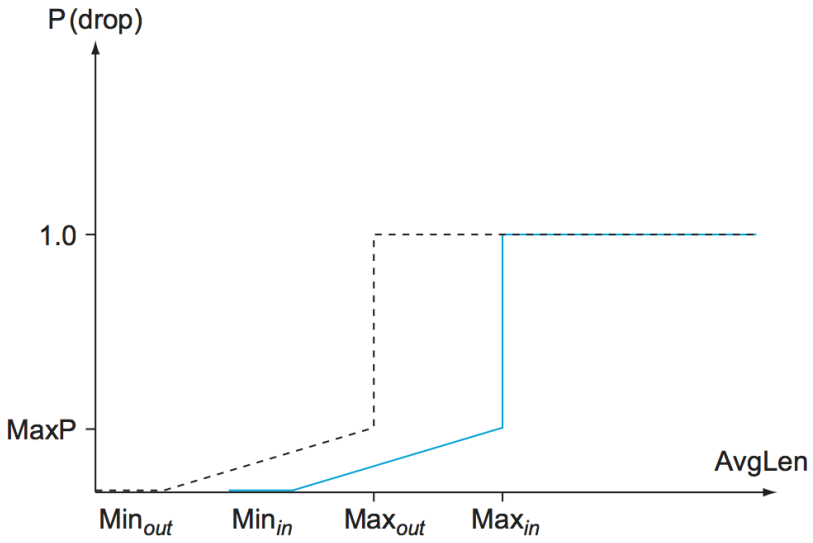


Figure 6.27.: RED with In and Out drop probabilities.

The reason for calling the two classes of packets "in" and "out" stems from the way the packets are marked. We already noted that packet marking can be performed by a router at the edge of an administrative domain. We can think of this router as being at the boundary between a network service provider and some customer of that network. The customer might be any other network—for example, the network of a corporation or of another network service provider. The customer and the network service provider agree on some sort of profile for the assured service (and perhaps the customer pays the network service provider for this profile). The profile might be something like "Customer X is allowed to send up to y Mbps of assured traffic," or it could be significantly more complex. Whatever the profile is, the edge router can clearly mark the packets that arrive from this customer as being either in or out of profile. In the example just mentioned, as long as the customer sends less than y Mbps, all his packets will be marked "in," but once he exceeds that rate the excess packets will be marked "out."

The combination of a profile meter at the edge and RIO in all the routers of the service provider's network should provide the customer with a high assurance (but not a guarantee) that packets within his profile can be delivered. In particular, if the majority of packets, including those sent by customers who have not paid extra to establish a profile, are "out" packets, then it should usually be the case that the RIO mechanism will act to keep congestion low enough that "in" packets are rarely dropped. Clearly, there must be enough bandwidth in the network so that the "in" packets alone are rarely able to congest a link to the point where RIO starts dropping "in" packets.

Just like RED, the effectiveness of a mechanism like RIO depends to some extent on correct parameter choices, and there are considerably more parameters to set for RIO. Exactly how well the scheme will work in production networks is not known at the time of writing.

One interesting property of RIO is that it does not change the order of "in" and "out" packets. For example, if a TCP connection is sending packets through a profile meter, and some packets are being marked "in" while others are marked "out," those packets will receive different drop probabilities in the router queues, but they will be delivered to the receiver in the same order in which they were sent. This is important for most TCP implementations, which perform much better when packets arrive in order, even if they are designed

to cope with misordering. Note also that mechanisms such as fast retransmit can be falsely triggered when misordering happens.

The idea of RIO can be generalized to provide more than two drop probability curves, and this is the idea behind the approach known as *weighted RED* (WRED). In this case, the value of the DSCP field is used to pick one of several drop probability curves, so that several different classes of service can be provided.

A third way to provide Differentiated Services is to use the DSCP value to determine which queue to put a packet into in a weighted fair queuing scheduler. As a very simple case, we might use one code point to indicate the *best-effort* queue and a second code point to select the *premium* queue. We then need to choose a weight for the premium queue that makes the premium packets get better service than the best-effort packets. This depends on the offered load of premium packets. For example, if we give the premium queue a weight of 1 and the best-effort queue a weight of 4, that ensures that the bandwidth available to premium packets is

$$B_{premium} = W_{premium}/(W_{premium} + W_{best-effort}) = 1/(1+4) = 0.2$$

That is, we have effectively reserved 20% of the link for premium packets, so if the offered load of premium traffic is only 10% of the link on average, then the premium traffic will behave as if it is running on a very underloaded network and the service will be very good. In particular, the delay experienced by the premium class can be kept low, since WFQ will try to transmit premium packets as soon as they arrive in this scenario. On the other hand, if the premium traffic load were 30%, it would behave like a highly loaded network, and delay could be very high for the premium packets—even worse than for the so-called best-effort packets. Thus, knowledge of the offered load and careful setting of weights is important for this type of service. However, note that the safe approach is to be very conservative in setting the weight for the premium queue. If this weight is made very high relative to the expected load, it provides a margin of error and yet does not prevent the best-effort traffic from using any bandwidth that has been reserved for premium but is not used by premium packets.

Just as in WRED, we can generalize this WFQ-based approach to allow more than two classes represented by different code points. Furthermore, we can combine the idea of a queue selector with a drop preference. For example, with 12 code points we can have four queues with different weights, each of which has three drop preferences. This is exactly what the IETF has done in the definition of "assured service."

6.5.4 Equation-Based Congestion Control

We conclude our discussion of QoS by returning full circle to TCP congestion control, but this time in the context of real-time applications. Recall that TCP adjusts the sender's congestion window (and, hence, the rate at which it can transmit) in response to ACK and timeout events. One of the strengths of this approach is that it does not require cooperation from the network's routers; it is a purely host-based strategy. Such a strategy complements the QoS mechanisms we've been considering, because (1) applications can use host-based solutions without depending on router support, and (2) even with DiffServ fully deployed, it is still possible for a router queue to be oversubscribed, and we would like real-time applications to react in a reasonable way should this happen.

While we would like to take advantage of TCP's congestion control algorithm, TCP itself is not appropriate for real-time applications. One reason is that TCP is a reliable protocol, and real-time applications often cannot afford the delays introduced by retransmission. However, what if we were to decouple TCP from its congestion control mechanism, to add TCP-like congestion control to an unreliable protocol like UDP? Could real-time applications make use of such a protocol?

On the one hand, this is an appealing idea because it would cause real-time streams to compete fairly with TCP streams. The alternative (which happens today) is that video applications use UDP without any form of congestion control and, as a consequence, steal bandwidth away from TCP flows that back off in the presence of congestion. On the other hand, the sawtooth behavior of TCP's congestion-control algorithm is not appropriate for real-time applications; it means that the rate at which the application transmits is constantly going up and down. In contrast, real-time applications work best when they are able to sustain a smooth transmission rate over a relatively long period of time.

Is it possible to achieve the best of both worlds: compatibility with TCP congestion control for the sake of fairness, while sustaining a smooth transmission rate for the sake of the application? Recent work suggests that the answer is yes. Specifically, several so called TCP-friendly congestion-control algorithms have been proposed. These algorithms have two main goals. One is to slowly adapt the congestion window. This is done by adapting over relatively longer time periods (e.g., an RTT) rather than on a per-packet basis. This smooths out the transmission rate. The second is to be TCP friendly in the sense of being fair to competing TCP flows. This property is often enforced by ensuring that the flow's behavior adheres to an equation that models TCP's behavior. Hence, this approach is sometimes called *equation-based congestion control*.

We saw a simplified form of the TCP rate equation in an earlier section. For our purposes, it is sufficient to note that the equation takes this general form:

$$Rate \propto \left(\frac{1}{RTT \times \sqrt{\rho}} \right)$$

which says that to be TCP-friendly, the transmission rate must be inversely proportional to the round-trip time (RTT) and the square root of the loss rate (ρ). In other words, to build a congestion control mechanism out of this relationship, the receiver must periodically report the loss rate it is experiencing back to the sender (e.g., it might report that it failed to receive 10% of the last 100 packets), and the sender then adjusts its sending rate up or down, such that this relationship continues to hold. Of course, it is still up to the application to adapt to these changes in the available rate, but as we will see in the next chapter, many real-time applications are quite adaptable.

Perspective: Software-Defined Traffic Engineering

The overarching problem this chapter addresses is how to allocate the available network bandwidth to a set of end-to-end flows. Whether it's TCP congestion control, integrated services, or differentiated services, there is an assumption that the underlying network bandwidth being allocated is fixed: a 1-Gbps link between site A and site B is always a 1-Gbps link, and the algorithms focus on how to best share that 1-Gbps among competing users. But what if that's not the case? What if you could "instantly" acquire additional capacity, so the 1-Gbps link is upgraded to a 10-Gbps link, or perhaps you could add a new link between two sites that had not previously been connected?

This possibility is real, and it's a topic that's usually referred to as *traffic engineering*, a term that dates back to the early days of networking when operators would analyze the traffic workloads on their network, and periodically re-engineer their networks to add capacity when the existing links became chronically overloaded. In those early days, the decision to add capacity was not taken lightly; you needed to be sure the usage trend you observed was not just a passing blip since it would take a significant amount of time and money to change the network. In the worse case, it might involve laying cable across an ocean or launching a satellite into space.

But with the advent of technologies like DWDM (*Section 3.1*) and MPLS (*Section 4.4*), we don't always have to lay more fiber, but can instead turn on additional wavelengths or establish new circuits between any pair of sites. (These sites need not be directly connected by fiber. For example, a wavelength between Boston and San Francisco might run through ROADMs in Chicago and Denver, but from the perspective of the L2/L3 network topology, Boston and San Francisco are connected by a direct link.) This dramatically lowers the time-to-availability, but reconfiguring hardware still requires manual intervention, and so our definition of "instantly" is still measured in days, if not weeks. Afterall, there are requisition forms to be filled out, in triplicate!

But as we have seen again and again, once you provide the right programmatic interfaces, software can be brought to bear on the problem, and "instantly" can, for all practical purposes, be truly instantaneous. This is effectively what cloud providers do with the private backbones they build to interconnect their datacenters. For example, Google has publicly described their private WAN, called B4, which is built entirely using white-box switches and SDN. B4 does not add/drop wavelengths to adjust inter-node bandwidth—it dynamically builds end-to-end tunnels using a technique called *Equal-Cost Multipath* (ECMP), an alternative to CSPF introduced in *Section 4.4*—but the flexibility it affords is similar.

A Traffic Engineering (TE) control program then provisions the network according to the needs of various classes of applications. B4 identifies three such classes: (1) copying user data (e.g., email, documents, audio/video) to remote datacenters for availability; (2) accessing remote storage by computations that run over distributed data sources; and (3) pushing large-scale data to synchronize state across multiple datacenters. These classes are ordered in increasing volume, decreasing latency sensitivity, and decreasing overall priority. For example, user-data represents the lowest volume on B4, is the most latency sensitive, and is of the highest priority.

By centralizing the decision-making process, which is one of the claimed advantages of SDN, Google has been able to drive their link utilizations to near 100%. This is two to three times better than the 30-40% average utilization that WAN links are typically provisioned for, which is necessary to allow those networks to deal with both traffic bursts and link/switch failures. If you can centrally decide how to allocate resources across the entire network, it is possible to run the network much closer to maximum utilization. Keep in mind that provisioning links in the network is done for coarse-grain application classes. TCP congestion control still operates on a connection-by-connection basis, and routing decisions are still made on top of the B4 topology. (As an aside, it is worth noting that because B4 is a private WAN, Google is free to run their own congestion control algorithm, such as BBR, without fear that it will unfairly disadvantage other algorithms.)

One lesson to take away from systems like B4 is that the line between traffic engineering and congestion control (as well as between traffic engineering and routing) is fuzzy. There are different mechanisms working to address the same general problem, and so there is no fixed-and-hard line that says where one mechanism stops and another begins. In short, layer boundaries become soft (and easy to move) when the layers are implemented in software rather than hardware. This is increasingly becoming the norm.

Broader Perspective

To continue reading about the cloudification of the Internet, see *Perspective: Big Data and Analytics*.

To learn more about the B4, we recommend: B4: Experience with a Globally Deployed Software Defined WAN, August 2013.

CHAPTER

SEVEN

END-TO-END DATA

It is a capital mistake to theorize before one has data.

—*Sir Arthur Conan Doyle*

Problem: What Do We Do with the Data?

From the network's perspective, application programs send messages to each other. Each of these messages is just an uninterpreted string of bytes. From the application's perspective, however, these messages contain various kinds of *data*—arrays of integers, video frames, lines of text, digital images, and so on. In other words, these bytes have meaning. We now consider the problem of how best to encode the different kinds of data that application programs want to exchange into byte strings. In many respects, this is similar to the problem of encoding byte strings into electromagnetic signals that we saw in an earlier chapter.

Thinking back to our discussion of encoding, there are essentially two concerns. The first is that the receiver be able to extract the same message from the signal as the transmitter sent; this is the framing problem. The second is making the encoding as efficient as possible. Both of these concerns are also present when encoding application data into network messages.

In order for the receiver to extract the message sent by the transmitter, the two sides need to agree to a message format, often called the *presentation format*. If the sender wants to send the receiver an array of integers, for example, then the two sides have to agree what each integer looks like (how many bits long it is, what order the bytes are arranged in, and whether the most significant bit comes first or last, for example) and how many elements are in the array. The first section describes various encodings of traditional computer data, such as integers, floating-point numbers, character strings, arrays, and structures. Well-established formats also exist for multimedia data: Video, for example, is typically transmitted in one of the formats created by the Moving Picture Experts Group (MPEG), and still images are usually transmitted in Joint Photographic Experts Group (JPEG) format. The particular issues that arise in the encoding of multimedia data are discussed in the next section.

Multimedia data types require us to think about both presentation and *compression*. The well-known formats for the transmission and storage of audio and video deal with both these issues: making sure that what was recorded, photographed, or heard at the sender can be interpreted correctly by the receiver, and doing so in a way that does not overwhelm the network with massive amounts of multimedia data.

Compression and, more generally, the efficiency of encoding have a rich history, dating back to Shannon's pioneering work on information theory in the 1940s. In effect, there are two opposing forces at work here. In one direction, you would like as much redundancy in the data as possible so that the receiver is able

to extract the right data even if errors are introduced into the message. The error detection and correcting codes we saw in an earlier chapter add redundant information to messages for exactly this purpose. In the other direction, we would like to remove as much redundancy from the data as possible so that we may encode it in as few bits as possible. It turns out the multimedia data offers a wealth of opportunities for compression because of the way our senses and brains process visual and auditory signals. We don't hear high frequencies as well as lower ones, and we don't notice fine detail as much as the bigger picture in an image, especially if the image is moving.

Compression is important to the designers of networks for many reasons, not just because we rarely find ourselves with an abundance of bandwidth everywhere in the network. For example, the way we design a compression algorithm affects our sensitivity to lost or delayed data and thus may influence the design of resource allocation mechanisms and end-to-end protocols. Conversely, if the underlying network is unable to guarantee a fixed amount of bandwidth for the duration of a videoconference, we may choose to design compression algorithms that can adapt to changing network conditions.

Finally, an important aspect of both presentation formatting and data compression is that they require the sending and receiving hosts to process every byte of data in the message. It is for this reason that presentation formatting and compression are sometimes called *data manipulation* functions. This is in contrast to most of the protocols we have seen up to this point, which process a message without ever looking at its contents. Because of this need to read, compute on, and write every byte of data in a message, data manipulations affect end-to-end throughput over the network. In some cases, these manipulations can be the limiting factor.

7.1 Presentation Formatting

One of the most common transformations of network data is from the representation used by the application program into a form that is suitable for transmission over a network and *vice versa*. This transformation is typically called *presentation formatting*. As illustrated in Figure 7.1, the sending program translates the data it wants to transmit from the representation it uses internally into a message that can be transmitted over the network; that is, the data is *encoded* in a message. On the receiving side, the application translates this arriving message into a representation that it can then process; that is, the message is *decoded*. This process is sometimes called *argument marshalling* or *serialization*. This terminology comes from the Remote Procedure Call (RPC) world, where the client thinks it is invoking a procedure with a set of arguments, but these arguments are then "brought together and ordered in an appropriate and effective way" to form a network message.

You might ask what makes this problem challenging. One reason is that computers represent data in different ways. For example, some computers represent floating-point numbers in IEEE standard 754 format, while some older machines still use their own nonstandard format. Even for something as simple as integers, different architectures use different sizes (e.g., 16-bit, 32-bit, 64-bit). To make matters worse, on some machines integers are represented in *big-endian* form (the most significant bit of a word is in the byte with the highest address), while on other machines integers are represented in *little-endian* form (the most significant bit is in the byte with the lowest address). For example, PowerPC processors are big-endian machines, and the Intel x86 family is a little-endian architecture. Today, many architectures support both representations (and so are called *bi-endian*), but the point is that you can never be sure how the host you are communicating with stores integers. The big-endian and little-endian representations of the integer 34,677,374 are given in Figure 7.2.

Another reason that marshalling is difficult is that application programs are written in different languages,

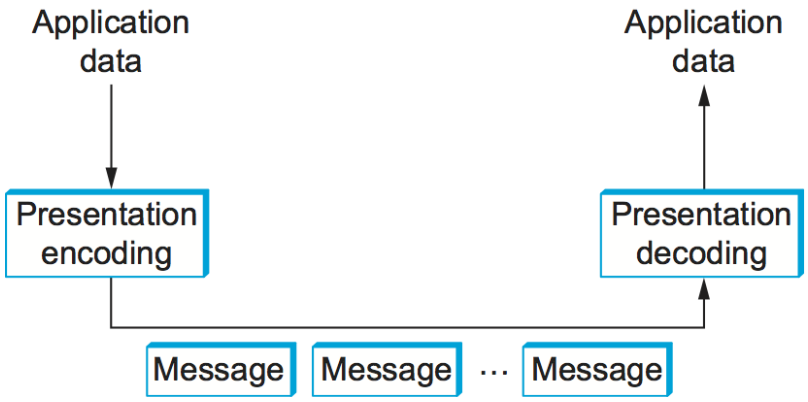


Figure 7.1.: Presentation formatting involves encoding and decoding application data.

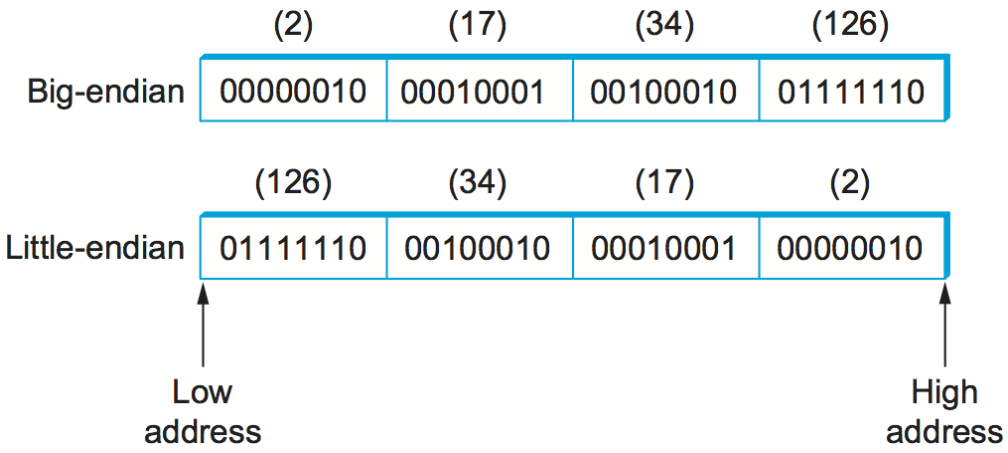


Figure 7.2.: Big-endian and little-endian byte order for the integer 34,677,374

and even when you are using a single language there may be more than one compiler. For example, compilers have a fair amount of latitude in how they lay out structures (records) in memory, such as how much padding they put between the fields that make up the structure. Thus, you could not simply transmit a structure from one machine to another, even if both machines were of the same architecture and the program was written in the same language, because the compiler on the destination machine might align the fields in the structure differently.

7.1.1 Taxonomy

Although argument marshalling is not rocket science—it is a small matter of bit twiddling—there are a surprising number of design choices that you must address. We begin by giving a simple taxonomy for argument marshalling systems. The following is by no means the only viable taxonomy, but it is sufficient to cover most of the interesting alternatives.

Data Types

The first question is what data types the system is going to support. In general, we can classify the types supported by an argument marshalling mechanism at three levels. Each level complicates the task faced by the marshalling system.

At the lowest level, a marshalling system operates on some set of *base types*. Typically, the base types include integers, floating-point numbers, and characters. The system might also support ordinal types and Booleans. As described above, the implication of the set of base types is that the encoding process must be able to convert each base type from one representation to another—for example, convert an integer from big-endian to little-endian.

At the next level are *flat types*—structures and arrays. While flat types might at first not appear to complicate argument marshalling, the reality is that they do. The problem is that the compilers used to compile application programs sometimes insert padding between the fields that make up the structure so as to align these fields on word boundaries. The marshalling system typically *packs* structures so that they contain no padding.

At the highest level, the marshalling system might have to deal with *complex types*—those types that are built using pointers. That is, the data structure that one program wants to send to another might not be contained in a single structure, but might instead involve pointers from one structure to another. A tree is a good example of a complex type that involves pointers. Clearly, the data encoder must prepare the data structure for transmission over the network because pointers are implemented by memory addresses, and just because a structure lives at a certain memory address on one machine does not mean it will live at the same address on another machine. In other words, the marshalling system must *serialize* (flatten) complex data structures.

In summary, depending on how complicated the type system is, the task of argument marshalling usually involves converting the base types, packing the structures, and linearizing the complex data structures, all to form a contiguous message that can be transmitted over the network. Figure 7.3 illustrates this task.

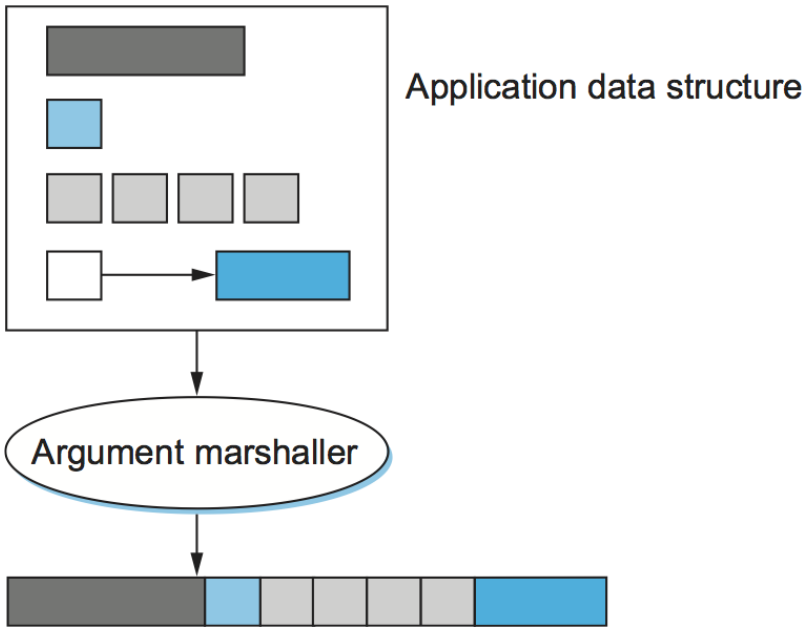


Figure 7.3.: Argument marshalling: converting, packing, and linearizing

Conversion Strategy

Once the type system is established, the next issue is what conversion strategy the argument marshaller will use. There are two general options: *canonical intermediate form* and *receiver-makes-right*. We consider each, in turn.

The idea of canonical intermediate form is to settle on an external representation for each type; the sending host translates from its internal representation to this external representation before sending data, and the receiver translates from this external representation into its local representation when receiving data. To illustrate the idea, consider integer data; other types are treated in a similar manner. You might declare that the big-endian format will be used as the external representation for integers. The sending host must translate each integer it sends into big-endian form, and the receiving host must translate big-endian integers into whatever representation it uses. (This is what is done in the Internet for protocol headers.) Of course, a given host might already use big-endian form, in which case no conversion is necessary.

The alternative, receiver-makes-right, has the sender transmit data in its own internal format; the sender does not convert the base types, but usually has to pack and flatten more complex data structures. The receiver is then responsible for translating the data from the sender's format into its own local format. The problem with this strategy is that every host must be prepared to convert data from all other machine architectures. In networking, this is known as an *N-by-N solution*: Each of N machine architectures must be able to handle all N architectures. In contrast, in a system that uses a canonical intermediate form, each host needs to know only how to convert between its own representation and a single other representation—the external one.

Using a common external format is clearly the correct thing to do, right? This has certainly been the conventional wisdom in the networking community for over 30 years. The answer is not cut and dried, however. It turns out that there are not that many different representations for the various base classes, or, said another way, N is not that large. In addition, the most common case is for two machines of the same type to be communicating with each other. In this situation, it seems silly to translate data from that architecture's

representation into some foreign external representation, only to have to translate the data back into the same architecture's representation on the receiver.

A third option, although we know of no existing system that exploits it, is to use receiver-makes-right if the sender knows that the destination has the same architecture; the sender would use some canonical intermediate form if the two machines use different architectures. How would a sender learn the receiver's architecture? It could learn this information either from a name server or by first using a simple test case to see if the appropriate result occurs.

Tags

The third issue in argument marshalling is how the receiver knows what kind of data is contained in the message it receives. There are two common approaches: *tagged* and *untagged* data. The tagged approach is more intuitive, so we describe it first.

A tag is any additional information included in a message—beyond the concrete representation of the base types—that helps the receiver decode the message. There are several possible tags that might be included in a message. For example, each data item might be augmented with a *type* tag. A type tag indicates that the value that follows is an integer, a floating-point number, or whatever. Another example is a *length* tag. Such a tag is used to indicate the number of elements in an array or the size of an integer. A third example is an *architecture* tag, which might be used in conjunction with the receiver-makes-right strategy to specify the architecture on which the data contained in the message was generated. Figure 7.4a depicts how a simple 32-bit integer might be encoded in a tagged message.

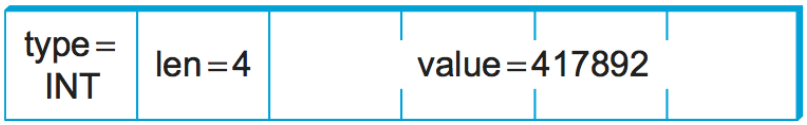


Figure 7.4.: A 32-bit integer encoded in a tagged message.

The alternative, of course, is not to use tags. How does the receiver know how to decode the data in this case? It knows because it was programmed to know. In other words, if you call a remote procedure that takes two integers and a floating-point number as arguments, then there is no reason for the remote procedure to inspect tags to know what it has just received. It simply assumes that the message contains two integers and a float and decodes it accordingly. Note that, while this works for most cases, the one place it breaks down is when sending variable-length arrays. In such a case, a length tag is commonly used to indicate how long the array is.

It is also worth noting that the untagged approach means that the presentation formatting is truly end to end. It is not possible for some intermediate agent to interpret the message unless the data is tagged. Why would an intermediate agent need to interpret a message, you might ask? Stranger things have happened, mostly resulting from *ad hoc* solutions to unexpected problems that the system was not engineered to handle. Poor network design is beyond the scope of this book.

Stubs

A stub is the piece of code that implements argument marshalling. Stubs are typically used to support RPC. On the client side, the stub marshals the procedure arguments into a message that can be transmitted by

means of the RPC protocol. On the server side, the stub converts the message back into a set of variables that can be used as arguments to call the remote procedure. Stubs can either be interpreted or compiled.

In a compilation-based approach, each procedure has a customized client and server stub. While it is possible to write stubs by hand, they are typically generated by a stub compiler, based on a description of the procedure's interface. This situation is illustrated in Figure 7.5. Since the stub is compiled, it is usually very efficient. In an interpretation-based approach, the system provides generic client and server stubs that have their parameters set by a description of the procedure's interface. Because it is easy to change this description, interpreted stubs have the advantage of being flexible. Compiled stubs are more common in practice.

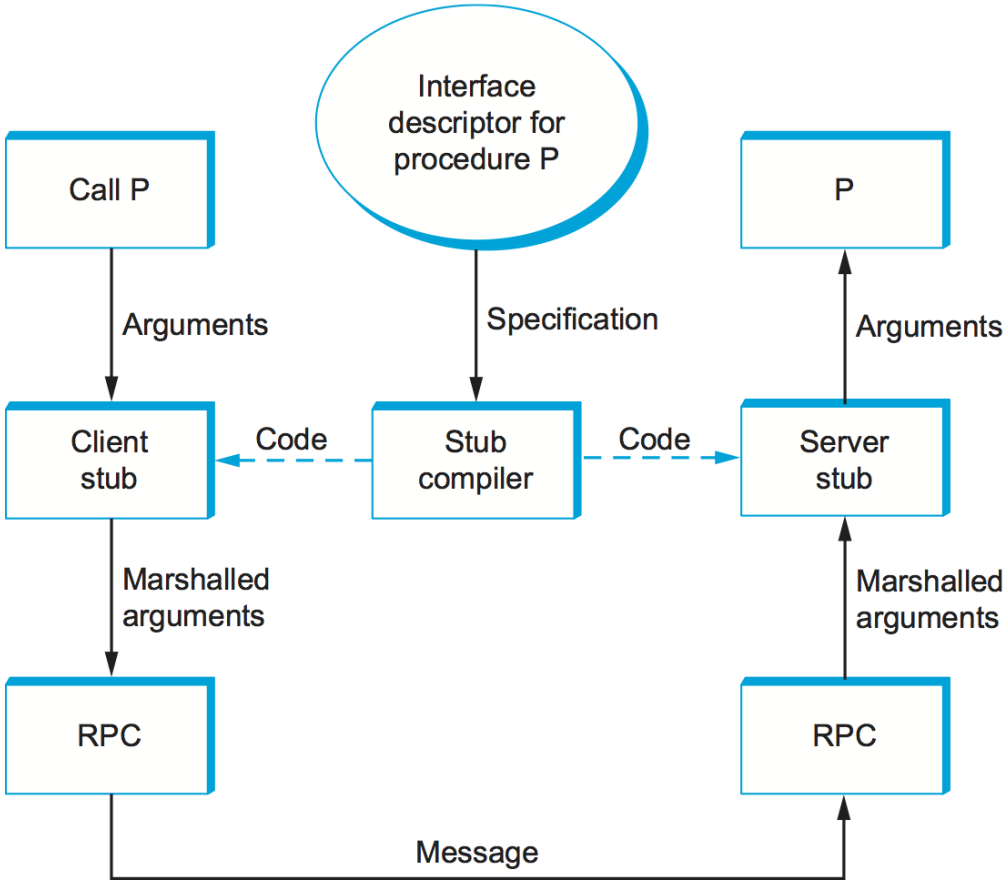


Figure 7.5.: Stub compiler takes interface description as input and outputs client and server stubs.

7.1.2 Examples (XDR, ASN.1, NDR, ProtoBufs)

We now briefly describe four popular network data representations in terms of this taxonomy. We use the integer base type to illustrate how each system works.

XDR

External Data Representation (XDR) is the network format used with SunRPC. In the taxonomy just introduced, XDR

- Supports the entire C-type system with the exception of function pointers
- Defines a canonical intermediate form
- Does not use tags (except to indicate array lengths)
- Uses compiled stubs

An XDR integer is a 32-bit data item that encodes a C integer. It is represented in twos' complement notation, with the most significant byte of the C integer in the first byte of the XDR integer and the least significant byte of the C integer in the fourth byte of the XDR integer. That is, XDR uses big-endian format for integers. XDR supports both signed and unsigned integers, just as C does.

XDR represents variable-length arrays by first specifying an unsigned integer (4 bytes) that gives the number of elements in the array, followed by that many elements of the appropriate type. XDR encodes the components of a structure in the order of their declaration in the structure. For both arrays and structures, the size of each element/component is represented in a multiple of 4 bytes. Smaller data types are padded out to 4 bytes with 0s. The exception to this "pad to 4 bytes" rule is made for characters, which are encoded one per byte.

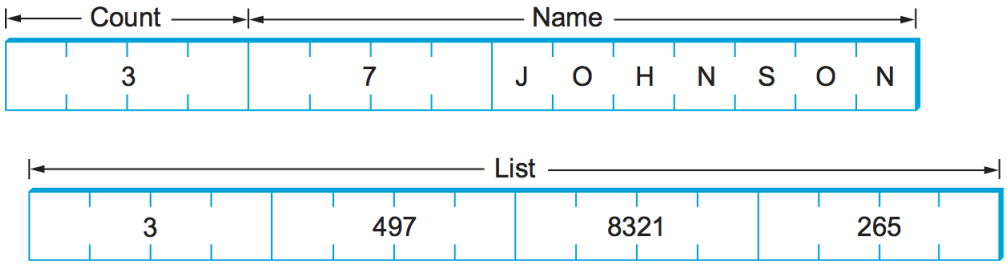


Figure 7.6.: Example encoding of a structure in XDR.

The following code fragment gives an example C structure (`item`) and the XDR routine that encodes/decodes this structure (`xdr_item`). Figure 7.6 schematically depicts XDR's on-the-wire representation of this structure when the field `name` is seven characters long and the array `list` has three values in it.

In this example, `xdr_array`, `xdr_int`, and `xdr_string` are three primitive functions provided by XDR to encode and decode arrays, integers, and character strings, respectively. Argument `xdrs` is a context variable that XDR uses to keep track of where it is in the message being processed; it includes a flag that indicates whether this routine is being used to encode or decode the message. In other words, routines like `xdr_item` are used on both the client and the server. Note that the application programmer can either write the routine `xdr_item` by hand or use a stub compiler called `rpcgen` (not shown) to generate this encoding/decoding routine. In the latter case, `rpcgen` takes the remote procedure that defines the data structure `item` as input and outputs the corresponding stub.

```
#define MAXNAME 256;
#define MAXLIST 100;

struct item {
   int      count;
   char     name[MAXNAME];
   int      list[MAXLIST];
```

(continues on next page)

(continued from previous page)

```
};

bool_t
xdr_item(XDR *xdrs, struct item *ptr)
{
    return(xdr_int(xdrs, &ptr->count) &&
       xdr_string(xdrs, &ptr->name, MAXNAME) &&
       xdr_array(xdrs, &ptr->list, &ptr->count, MAXLIST,
                 sizeof(int), xdr_int));
}
```

Exactly how XDR performs depends, of course, on the complexity of the data. In a simple case of an array of integers, where each integer has to be converted from one byte order to another, an average of three instructions are required for each byte, meaning that converting the whole array is likely to be limited by the memory bandwidth of the machine. More complex conversions that require significantly more instructions per byte will be CPU limited and thus perform at a data rate less than the memory bandwidth.

ASN.1

Abstract Syntax Notation One (ASN.1) is an ISO standard that defines, among other things, a representation for data sent over a network. The representation-specific part of ASN.1 is called the *Basic Encoding Rules* (BER). ASN.1 supports the C-type system without function pointers, defines a canonical intermediate form, and uses type tags. Its stubs can be either interpreted or compiled. One of the claims to fame of ASN.1 BER is that it is used by the Internet standard Simple Network Management Protocol (SNMP).

ASN.1 represents each data item with a triple of the form

```
(tag, length, value)
```

The `tag` is typically an 8-bit field, although ASN.1 allows for the definition of multibyte tags. The `length` field specifies how many bytes make up the `value`; we discuss `length` more below. Compound data types, such as structures, can be constructed by nesting primitive types, as illustrated in Figure 7.7.

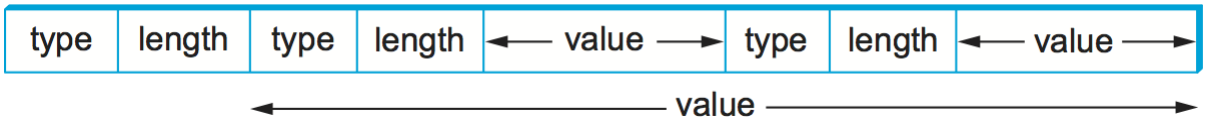


Figure 7.7.: Compound types created by means of nesting in ASN.1 BER.

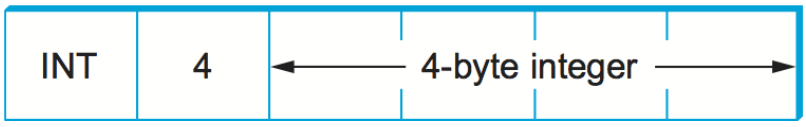


Figure 7.8.: ASN.1 BER representation for a 4-byte integer.

If the `value` is 127 or fewer bytes long, then the `length` is specified in a single byte. Thus, for example, a 32-bit integer is encoded as a 1-byte `type`, a 1-byte `length`, and the 4 bytes that encode the integer, as illustrated in Figure 7.8. The `value` itself, in the case of an integer, is represented in twos' complement

notation and big-endian form, just as in XDR. Keep in mind that, even though the `value` of the integer is represented in exactly the same way in both XDR and ASN.1, the XDR representation has neither the `type` nor the `length` tags associated with that integer. These two tags both take up space in the message and, more importantly, require processing during marshalling and unmarshalling. This is one reason why ASN.1 is not as efficient as XDR. Another is that the very fact that each data value is preceded by a `length` field means that the data value is unlikely to fall on a natural byte boundary (e.g., an integer beginning on a word boundary). This complicates the encoding/decoding process.

If the `value` is 128 or more bytes long, then multiple bytes are used to specify its `length`. At this point you may be asking why a byte can specify a length of up to 127 bytes rather than 256. The reason is that 1 bit of the `length` field is used to denote how long the `length` field is. A 0 in the eighth bit indicates a 1-byte `length` field. To specify a longer `length`, the eighth bit is set to 1, and the other 7 bits indicate how many additional bytes make up the `length`. Figure 7.9 illustrates a simple 1-byte `length` and a multibyte `length`.

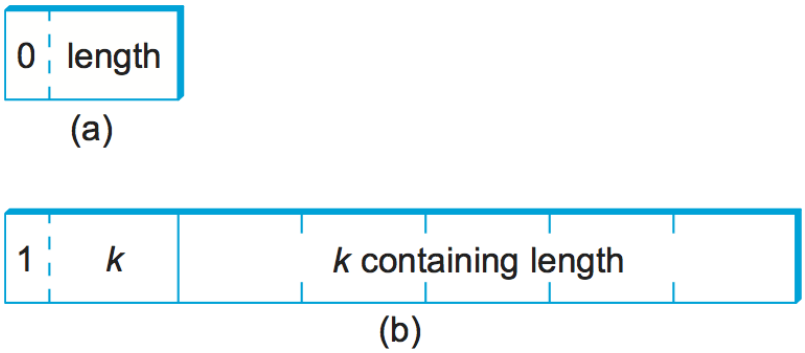


Figure 7.9.: ASN.1 BER representation for length: (a) 1 byte; (b) multibyte.

NDR

Network Data Representation (NDR) is the data-encoding standard used in the Distributed Computing Environment (DCE). Unlike XDR and ASN.1, NDR uses receiver-makes-right. It does this by inserting an architecture tag at the front of each message; individual data items are untagged. NDR uses a compiler to generate stubs. This compiler takes a description of a program written in the Interface Definition Language (IDL) and generates the necessary stubs. IDL looks pretty much like C, and so essentially supports the C-type system.

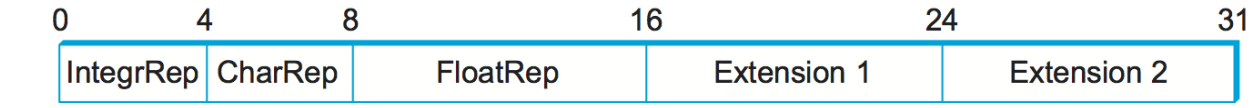


Figure 7.10.: NDR's architecture tag.

Figure 7.10 illustrates the 4-byte architecture definition tag that is included at the front of each NDR-encoded message. The first byte contains two 4-bit fields. The first field, `IntegrRep`, defines the format for all integers contained in the message. A 0 in this field indicates big-endian integers, and a 1 indicates little-endian integers. The `CharRep` field indicates what character format is used: 0 means ASCII (American Standard Code for Information Interchange) and 1 means EBCDIC (an older, IBM-defined alternative to ASCII). Next, the `FloatRep` byte defines which floating-point representation is being used: 0 means IEEE

754, 1 means VAX, 2 means Cray, and 3 means IBM. The final 2 bytes are reserved for future use. Note that, in simple cases such as arrays of integers, NDR does the same amount of work as XDR, and so it is able to achieve the same performance.

ProtoBufs

Protocol Buffers (Protobufs, for short) is a language-neutral and platform-neutral way of serializing structured data, commonly used with gRPC. It uses a tagged strategy with a canonical intermediate form, where the stub on both sides is generated from a shared `.proto` file. This specification uses a simple C-like syntax, as the following example illustrates:

```
message Person {
    required string name = 1;
    required int32 id = 2;
    optional string email = 3;

    enum PhoneType {
        MOBILE = 0;
        HOME = 1;
        WORK = 2;
    }

    message PhoneNumber {
        required string number = 1;
        optional PhoneType type = 2 [default = HOME];
    }

    required PhoneNumber phone = 4;
}
```

where `message` could roughly be interpreted as equivalent to `typedef struct` in C. The rest of the example is fairly intuitive, except that every field is given a numeric identifier to ensure uniqueness should the specification change over time, and each field can be annotated as being either `required` or `optional`.

The way Protobufs encodes integers is novel. It uses a technique called *varints* (variable length integers) in which each 8-bit byte uses the most significant bit to indicate whether there are more bytes in the integer, and the lower seven bits to encode the two's complement representation of the next group of seven bits in the value. The least significant group is first in the serialization.

This means a small integer (less than 128) can be encoded in a single byte (e.g., the integer 2 is encoded as `0000 0010`), while for an integer bigger than 128, more bytes are needed. For example, 365 would be encoded as

```
1110 1101 0000 0010
```

To see this, first drop the most significant bit from each byte, as it is there to tell us whether we've reached the end of the integer. In this example, the `1` in the most significant bit of the first byte indicates there is more than one byte in the varint:

```
1110 1101 0000 0010
→ 110 1101  000 0010
```

Since varints store numbers with the least significant group first, you next reverse the two groups of seven bits. Then you concatenate them to get your final value:

```
000 0010  110 1101
→  000 0010 || 110 1101
→  101101101
→  256 + 64 + 32 + 8 + 4 + 1 = 365
```

For the larger message specification, you can think of the serialized byte stream as a collection of key/value pairs, where the key (i.e., tag) has two sub-parts: the unique identifier for the field (i.e., those extra numbers in the example `.proto` file) and the *wire type* of the value (e.g., `Varint` is the one example wire type we have seen so far). Other supported wire types include `32-bit` and `64-bit` (for fixed-length integers), and `length-delimited` (for strings and embedded messages). The latter tells you how many bytes long the embedded message (structure) is, but it's another `message` specification in the `.proto` file that tells you how to interpret those bytes.

7.1.3 Markup Languages (XML)

Although we have been discussing the presentation formatting problem from the perspective of RPC—that is, how does one encode primitive data types and compound data structures so they can be sent from a client program to a server program—the same basic problem occurs in other settings. For example, how does a web server describe a Web page so that any number of different browsers know what to display on the screen? In this specific case, the answer is the HyperText Markup Language (HTML), which indicates that certain character strings should be displayed in bold or italics, what font type and size should be used, and where images should be positioned.

The availability of all sorts of Web applications and data have also created a situation in which different Web applications need to communicate with each other and understand each other's data. For example, an e-commerce website might need to talk to a shipping company's website to allow a customer to track a package without ever leaving the e-commerce website. This quickly starts to look a lot like RPC, and the approach taken in the Web today to enable such communication among web servers is based on the *Extensible Markup Language* (XML)—a way to describe the data being exchanged between Web apps.

Markup languages, of which HTML and XML are both examples, take the tagged data approach to the extreme. Data is represented as text, and text tags known as *markup* are intermingled with the data text to express information about the data. In the case of HTML, markup indicates how the text should be displayed; other markup languages like XML can express the type and structure of the data.

XML is actually a framework for defining different markup languages for different kinds of data. For example, XML has been used to define a markup language that is roughly equivalent to HTML called *Extensible HyperText Markup Language* (XHTML). XML defines a basic syntax for mixing markup with data text, but the designer of a specific markup language has to name and define its markup. It is common practice to refer to individual XML-based languages simply as XML, but we will emphasize the distinction in this introductory material.

XML syntax looks much like HTML. For example, an employee record in a hypothetical XML-based language might look like the following XML *document*, which might be stored in a file named `employee.xml`. The first line indicates the version of XML being used, and the remaining lines represent four fields that make up the employee record, the last of which (`hiredate`) contains three subfields. In other words, XML syntax provides for a nested structure of tag/value pairs, which is equivalent to a tree structure for

the represented data (with `employee` as the root). This is similar to XDR, ASN.1, and NDR's ability to represent compound types, but in a format that can be both processed by programs and read by humans. More importantly, programs such as parsers can be used across different XML-based languages, because the definitions of those languages are themselves expressed as machine-readable data that can be input to the programs.

```
<?xml version="1.0"?>
<employee>
   <name>John Doe</name>
   <title>Head Bottle Washer</title>
   <id>123456789</id>
   <hiredate>
      <day>5</day>
      <month>June</month>
      <year>1986</year>
   </hiredate>
</employee>
```

Although the markup and the data in this document are highly suggestive to the human reader, it is the definition of the employee record language that actually determines what tags are legal, what they mean, and what data types they imply. Without some formal definition of the tags, a human reader (or a computer) can't tell whether `1986` in the `year` field, for example, is a string, an integer, an unsigned integer, or a floating point number.

The definition of a specific XML-based language is given by a *schema*, which is a database term for a specification of how to interpret a collection of data. Several schema languages have been defined for XML; we will focus here on the leading standard, known by the none-too-surprising name *XML Schema*. An individual schema defined using XML Schema is known as an *XML Schema Document* (XSD). The following is an XSD specification for the example; in other words, it defines the language to which the example document conforms. It might be stored in a file named `employee.xsd`.

```
<?xml version="1.0"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema">
  <element name="employee">
    <complexType>
      <sequence>
        <element name="name" type="string"/>
        <element name="title" type="string"/>
        <element name="id" type="string"/>
        <element name="hiredate">
          <complexType>
            <sequence>
              <element name="day" type="integer"/>
              <element name="month" type="string"/>
              <element name="year" type="integer"/>
            </sequence>
          </complexType>
        </element>
      </sequence>
    </complexType>
  </element>
</schema>
```

This XSD looks superficially similar to our example document `employee.xml`, and for good reason: XML Schema is itself an XML-based language. There is an obvious relationship between this XSD and the document defined above. For example,

```
<element name="title" type="string"/>
```

indicates that the value bracketed by the markup `title` is to be interpreted as a string. The sequence and nesting of that line in the XSD indicate that a `title` field must be the second item in an employee record.

Unlike some schema languages, XML Schema provides datatypes such as string, integer, decimal, and Boolean. It allows the datatypes to be combined in sequences or nested, as in `employee.xsd`, to create compound data types. So an XSD defines more than a syntax; it defines its own abstract data model. A document that conforms to the XSD represents a collection of data that conforms to the data model.

The significance of an XSD defining an abstract data model and not just a syntax is that there can be other ways besides XML of representing data that conforms to the model. And XML does, after all, have some shortcomings as an on-the-wire representation: it is not as compact as other data representations, and it is relatively slow to parse. A number of alternative representations described as binary are in use. The International Standards Organization (ISO) has published one called *Fast Infoset*, while the World Wide Web Consortium (W3C) has produced the *Efficient XML Interchange* (EXI) proposal. Binary representations sacrifice human readability for greater compactness and faster parsing.

XML Namespaces

XML has to solve a common problem, that of name clashes. The problem arises because schema languages such as XML Schema support modularity in the sense that a schema can be reused as part of another schema. Suppose two XSDs are defined independently, and both happen to define the markup name *idNumber*. Perhaps one XSD uses that name to identify employees of a company, and the other XSD uses it to identify laptop computers owned by the company. We might like to reuse those two XSDs in a third XSD for describing which assets are associated with which employees, but to do that we need some mechanism for distinguishing employees' idNumbers from laptop idNumbers.

XML's solution to this problem is *XML namespaces*. A namespace is a collection of names. Each XML namespace is identified by a Uniform Resource Identifier (URI). URIs will be described in some detail in a later chapter; for now, all you really need to know is that URIs are a form of globally unique identifier. (An HTTP URL is a particular type of UNI.) A simple markup name like *idNumber* can be added to a namespace as long as it is unique within that namespace. Since the namespace is globally unique and the simple name is unique within the namespace, the combination of the two is a globally unique *qualified name* that cannot clash.

An XSD usually specifies a *target namespace* with a line like the following:

```
targetNamespace="http://www.example.com/employee"
```

is a Uniform Resource Identifier, identifying a made-up namespace. All the new markup defined in that XSD will belong to that namespace.

Now, if an XSD wants to reference names that have been defined in other XSDs, it can do so by qualifying those names with a namespace prefix. This prefix is a short abbreviation for the full URI that actually

identifies the namespace. For example, the following line assigns `emp` as the namespace prefix for the employee namespace:

```
xmlns:emp="http://www.example.com/employee"
```

Any markup from that namespace would be qualified by prefixing it with `emp:` , as is `title` in the following line:

```
<emp:title>Head Bottle Washer</emp:title>
```

In other words, `emp:title` is a qualified name, which will not clash with the name `title` from some other namespace.

It is remarkable how widely XML is now used in applications that range from RPC-style communication among Web-based services to office productivity tools to instant messaging. It is certainly one of the core protocols on which the upper layers of the Internet now depend.

7.2 Multimedia Data

Multimedia data, comprised of audio, video, and still images, now makes up the majority of traffic on the Internet. Part of what has made the widespread transmission of multimedia across networks possible is advances in compression technology. Because multimedia data is consumed mostly by humans using their senses—vision and hearing—and processed by the human brain, there are unique challenges to compressing it. You want to try to keep the information that is most important to a human, while getting rid of anything that doesn't improve the human's perception of the visual or auditory experience. Hence, both computer science and the study of human perception come into play. In this section, we'll look at some of the major efforts in representing and compressing multimedia data.

The uses of compression are not limited to multimedia data of course—for example, you may well have used a utility like `zip` or `compress` to compress files before sending them over a network, or to uncompress a data file after downloading. It turns out that the techniques used for compressing data—which are typically *lossless*, because most people don't like to lose data from a file—also show up as part of the solution for multimedia compression. In contrast, *lossy compression*, commonly used for multimedia data, does not promise that the data received is exactly the same as the data sent. As noted above, this is because multimedia data often contains information that is of little utility to the human who receives it. Our senses and brains can only perceive so much detail. They are also very good at filling in missing pieces and even correcting some errors in what we see or hear. And, lossy algorithms typically achieve much better compression ratios than do their lossless counterparts; they can be an order of magnitude better or more.

To get a sense of how important compression has been to the spread of networked multimedia, consider the following example. A high-definition TV screen has something like 1080 × 1920 pixels, each of which has 24 bits of color information, so each frame is

$$1080 \times 1920 \times 24 = 50\ Mb$$

so if you want to send 24 frames per second, that would be over 1 Gbps. That's more than most Internet users have access to. By contrast, modern compression techniques can get a reasonably high-quality HDTV signal down to the range of 10 Mbps, a two order of magnitude reduction and well within the reach of most broadband users. Similar compression gains apply to lower quality video such as YouTube clips—Web

video could never have reached its current popularity without compression to make all those entertaining videos fit within the bandwidth of today's networks.

Compression techniques as applied to multimedia have been an area of great innovation, particularly lossy compression. Lossless techniques also have an important role to play, however. Indeed, most of the lossy techniques include some steps that are lossless, so we begin our discussion with an overview of lossless compression.

7.2.1 Lossless Compression Techniques

In many ways, compression is inseparable from data encoding. When thinking about how to encode a piece of data in a set of bits, we might just as well think about how to encode the data in the smallest set of bits possible. For example, if you have a block of data that is made up of the 26 symbols A through Z, and if all of these symbols have an equal chance of occurring in the data block you are encoding, then encoding each symbol in 5 bits is the best you can do (since $2^5 = 32$ is the lowest power of 2 above 26). If, however, the symbol R occurs 50% of the time, then it would be a good idea to use fewer bits to encode the R than any of the other symbols. In general, if you know the relative probability that each symbol will occur in the data, then you can assign a different number of bits to each possible symbol in a way that minimizes the number of bits it takes to encode a given block of data. This is the essential idea of *Huffman codes*, one of the important early developments in data compression.

Run Length Encoding

Run length encoding (RLE) is a compression technique with a brute-force simplicity. The idea is to replace consecutive occurrences of a given symbol with only one copy of the symbol, plus a count of how many times that symbol occurs—hence, the name *run length*. For example, the string `AAABBCDDDD` would be encoded as `3A2B1C4D`.

RLE turns out to be useful for compressing some classes of images. It can be used in this context by comparing adjacent pixel values and then encoding only the changes. For images that have large homogeneous regions, this technique is quite effective. For example, it is not uncommon that RLE can achieve compression ratios on the order of 8-to-1 for scanned text images. RLE works well on such files because they often contain a large amount of white space that can be removed. For those old enough to remember the technology, RLE was the key compression algorithm used to transmit faxes. However, for images with even a small degree of local variation, it is not uncommon for compression to actually increase the image byte size, since it takes 2 bytes to represent a single symbol when that symbol is not repeated.

Differential Pulse Code Modulation

Another simple lossless compression algorithm is Differential Pulse Code Modulation (DPCM). The idea here is to first output a reference symbol and then, for each symbol in the data, to output the difference between that symbol and the reference symbol. For example, using symbol A as the reference symbol, the string `AAABBCDDDD` would be encoded as `A0001123333` because A is the same as the reference symbol, B has a difference of 1 from the reference symbol, and so on. Note that this simple example does not illustrate the real benefit of DPCM, which is that when the differences are small they can be encoded with fewer bits than the symbol itself. In this example, the range of differences, 0-3, can be represented with

2 bits each, rather than the 7 or 8 bits required by the full character. As soon as the difference becomes too large, a new reference symbol is selected.

DPCM works better than RLE for most digital imagery, since it takes advantage of the fact that adjacent pixels are usually similar. Due to this correlation, the dynamic range of the differences between the adjacent pixel values can be significantly less than the dynamic range of the original image, and this range can therefore be represented using fewer bits. Using DPCM, we have measured compression ratios of 1.5-to-1 on digital images. DPCM also works on audio, because adjacent samples of an audio waveform are likely to be close in value.

A slightly different approach, called *delta encoding*, simply encodes a symbol as the difference from the previous one. Thus, for example, `AAABBCDDDD` would be represented as `A001011000`. Note that delta encoding is likely to work well for encoding images where adjacent pixels are similar. It is also possible to perform RLE after delta encoding, since we might find long strings of 0s if there are many similar symbols next to each other.

Dictionary-Based Methods

The final lossless compression method we consider is the dictionary-based approach, of which the Lempel-Ziv (LZ) compression algorithm is the best known. The Unix `compress` and `gzip` commands use variants of the LZ algorithm.

The idea of a dictionary-based compression algorithm is to build a dictionary (table) of variable-length strings (think of them as common phrases) that you expect to find in the data and then to replace each of these strings when it appears in the data with the corresponding index to the dictionary. For example, instead of working with individual characters in text data, you could treat each word as a string and output the index in the dictionary for that word. To further elaborate on this example, the word *compression* has the index 4978 in one particular dictionary; it is the 4978th word in . To compress a body of text, each time the string "compression" appears, it would be replaced by 4978. Since this particular dictionary has just over 25,000 words in it, it would take 15 bits to encode the index, meaning that the string "compression" could be represented in 15 bits rather than the 77 bits required by 7-bit ASCII. This is a compression ratio of 5-to-1! At another data point, we were able to get a 2-to-1 compression ratio when we applied the `compress` command to the source code for the protocols described in this book.

Of course, this leaves the question of where the dictionary comes from. One option is to define a static dictionary, preferably one that is tailored for the data being compressed. A more general solution, and the one used by LZ compression, is to adaptively define the dictionary based on the contents of the data being compressed. In this case, however, the dictionary constructed during compression has to be sent along with the data so that the decompression half of the algorithm can do its job. Exactly how you build an adaptive dictionary has been a subject of extensive research.

7.2.2 Image Representation and Compression (GIF, JPEG)

Given the ubiquitous use of digital imagery—this use was spawned by the invention of graphical displays, not high-speed networks—the need for standard representation formats and compression algorithms for digital imagery data has become essential. In response to this need, the ISO defined a digital image format known as *JPEG*, named after the Joint Photographic Experts Group that designed it. (The "Joint" in JPEG stands for a joint ISO/ITU effort.) JPEG is the most widely used format for still images in use today. At the

heart of the definition of the format is a compression algorithm, which we describe below. Many techniques used in JPEG also appear in MPEG, the set of standards for video compression and transmission created by the Moving Picture Experts Group.

Before delving into the details of JPEG, we observe that there are quite a few steps to get from a digital image to a compressed representation of that image that can be transmitted, decompressed, and displayed correctly by a receiver. You probably know that digital images are made up of pixels (hence, the megapixels quoted in smartphone camera advertisements). Each pixel represents one location in the two-dimensional grid that makes up the image, and for color images each pixel has some numerical value representing a color. There are lots of ways to represent colors, referred to as *color spaces*; the one most people are familiar with is RGB (red, green, blue). You can think of color as being a three dimensional quantity—you can make any color out of red, green, and blue light in different amounts. In a three-dimensional space, there are lots of different, valid ways to describe a given point (consider Cartesian and polar coordinates, for example). Similarly, there are various ways to describe a color using three quantities, and the most common alternative to RGB is YUV. The Y is luminance, roughly the overall brightness of the pixel, and U and V contain chrominance, or color information. Confoundingly, there are a few different variants of the YUV color space as well. More on this in a moment.

The significance of this discussion is that the encoding and transmission of color images (either still or moving) requires agreement between the two ends on the color space. Otherwise, of course, you'd end up with different colors being displayed by the receiver than were captured by the sender. Hence, agreeing on a color space definition (and perhaps a way to communicate which particular space is in use) is part of the definition of any image or video format.

Let's look at the example of the Graphical Interchange Format (GIF). GIF uses the RGB color space and starts out with 8 bits to represent each of the three dimensions of color for a total of 24 bits. Rather than sending those 24 bits per pixel, however, GIF first reduces 24-bit color images to 8-bit color images. This is done by identifying the colors used in the picture, of which there will typically be considerably fewer than 2^{24}, and then picking the 256 colors that most closely approximate the colors used in the picture. There might be more than 256 colors, however, so the trick is to try not to distort the color too much by picking 256 colors such that no pixel has its color changed too much.

The 256 colors are stored in a table, which can be indexed with an 8-bit number, and the value for each pixel is replaced by the appropriate index. Note that this is an example of lossy compression for any picture with more than 256 colors. GIF then runs an LZ variant over the result, treating common sequences of pixels as the strings that make up the dictionary—a lossless operation. Using this approach, GIF is sometimes able to achieve compression ratios on the order of 10:1, but only when the image consists of a relatively small number of discrete colors. Graphical logos, for example, are handled well by GIF. Images of natural scenes, which often include a more continuous spectrum of colors, cannot be compressed at this ratio using GIF. It is also not too hard for a human eye to detect the distortion caused by the lossy color reduction of GIF in some cases.

The JPEG format is considerably more well suited to photographic images, as you would hope given the name of the group that created it. JPEG does not reduce the number of colors like GIF. Instead, JPEG starts off by transforming the RGB colors (which are what you usually get out of a digital camera) to the YUV space. The reason for this has to do with the way the eye perceives images. There are receptors in the eye for brightness, and separate receptors for color. Because we're very good at perceiving variations in brightness, it makes sense to spend more bits on transmitting brightness information. Since the Y component of YUV is, roughly, the brightness of the pixel, we can compress that component separately, and less aggressively, from the other two (chrominance) components.

As noted above, YUV and RGB are alternative ways to describe a point in a 3-dimensional space, and it's possible to convert from one color space to another using linear equations. For one YUV space that is commonly used to represent digital images, the equations are:

```
Y = 0.299R + 0.587G + 0.114B
U = (B-Y) x 0.565
V =  (R-Y) x 0.713
```

The exact values of the constants here are not important, as long as the encoder and decoder agree on what they are. (The decoder will have to apply the inverse transformations to recover the RGB components needed to drive a display.) The constants are, however, carefully chosen based on the human perception of color. You can see that Y, the luminance, is a sum of the red, green, and blue components, while U and V are color difference components. U represents the difference between the luminance and blue, and V the difference between luminance and red. You may notice that setting R, G, and B to their maximum values (which would be 255 for 8-bit representations) will also produce a value of Y=255 while U and V in this case would be zero. That is, a fully white pixel is (255,255,255) in RGB space and (255,0,0) in YUV space.

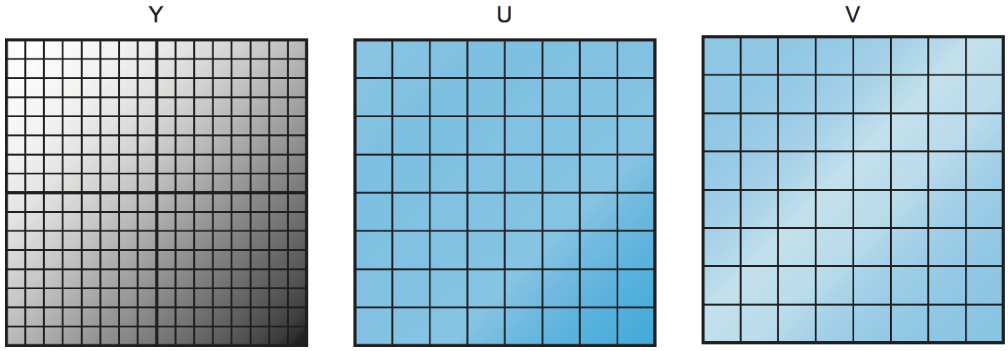


Figure 7.11.: Subsampling the U and V components of an image.

Once the image has been transformed into YUV space, we can now think about compressing each of the three components separately. We want to be more aggressive in compressing the U and V components, to which human eyes are less sensitive. One way to compress the U and V components is to *subsample* them. The basic idea of subsampling is to take a number of adjacent pixels, calculate the average U or V value for that group of pixels, and transmit that, rather than sending the value for every pixel. Figure 7.11 illustrates the point. The luminance (Y) component is not subsampled, so the Y value of all the pixels will be transmitted, as indicated by the 16 × 16 grid of pixels on the left. In the case of U and V, we treat each group of four adjacent pixels as a group, calculate the average of the U or V value for that group, and transmit that. Hence, we end up with an 8 × 8 grid of U and V values to transmit. Thus, in this example, for every four pixels, we transmit six values (four Y and one each of U and V) rather than the original 12 values (four each for all three components), for a 50% reduction in information.

It's worth noting that you could be either more or less aggressive in the subsampling, with corresponding increases in compression and decreases in quality. The subsampling approach shown here, in which chrominance is subsampled by a factor of two in both horizontal and vertical directions (and which goes by the identification 4:2:0), happens to match the most common approach used for both JPEG and MPEG.

Once subsampling is done, we now have three grids of pixels to deal with, and each one is dealt with separately. JPEG compression of each component takes place in three phases, as illustrated in Figure 7.12. On the compression side, the image is fed through these three phases one 8 × 8 block at a time. The first

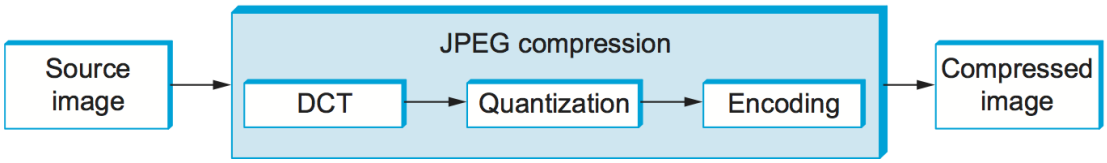


Figure 7.12.: Block diagram of JPEG compression.

phase applies the discrete cosine transform (DCT) to the block. If you think of the image as a signal in the spatial domain, then DCT transforms this signal into an equivalent signal in the *spatial frequency* domain. This is a lossless operation but a necessary precursor to the next, lossy step. After the DCT, the second phase applies a quantization to the resulting signal and, in so doing, loses the least significant information contained in that signal. The third phase encodes the final result, but in so doing also adds an element of lossless compression to the lossy compression achieved by the first two phases. Decompression follows these same three phases, but in reverse order.

DCT Phase

DCT is a transformation closely related to the fast Fourier transform (FFT). It takes an 8 × 8 matrix of pixel values as input and outputs an 8 × 8 matrix of frequency coefficients. You can think of the input matrix as a 64-point signal that is defined in two spatial dimensions (*x* and *y*); DCT breaks this signal into 64 spatial frequencies. To get an intuitive feel for spatial frequency, imagine yourself moving across a picture in, say, the *x* direction. You would see the value of each pixel varying as some function of *x*. If this value changes slowly with increasing *x*, then it has a low spatial frequency; if it changes rapidly, it has a high spatial frequency. So the low frequencies correspond to the gross features of the picture, while the high frequencies correspond to fine detail. The idea behind the DCT is to separate the gross features, which are essential to viewing the image, from the fine detail, which is less essential and, in some cases, might be barely perceived by the eye.

DCT, along with its inverse, which recovers the original pixels and during decompression, are defined by the following formulas:

$$DCT(i,j) = \frac{1}{\sqrt{2N}}C(i)C(j)\sum_{x=0}^{N-1}\sum_{y=0}^{N-1} pixel(x,y)\cos\left[\frac{(2x+1)i\pi}{2N}\right]\cos\left[\frac{(2y+1)j\pi}{2N}\right]$$

$$pixel(x,y) = \frac{1}{\sqrt{2N}}\sum_{i=0}^{N-1}\sum_{j=0}^{N-1} C(i)C(j)DCT(i,j)\cos\left[\frac{(2x+1)i\pi}{2N}\right]\cos\left[\frac{(2y+1)j\pi}{2N}\right]$$

where $C(x) = 1/\sqrt{2}$ when $x = 0$ and 1 when $x > 0$, and $pixel(x,y)$ is the grayscale value of the pixel at position *(x,y)* in the 8 × 8 block being compressed; N = 8 in this case.

The first frequency coefficient, at location (0,0) in the output matrix, is called the *DC coefficient*. Intuitively, we can see that the DC coefficient is a measure of the average value of the 64 input pixels. The other 63 elements of the output matrix are called the *AC coefficients*. They add the higher-spatial-frequency information to this average value. Thus, as you go from the first frequency coefficient toward the 64th frequency coefficient, you are moving from low-frequency information to high-frequency information, from the broad strokes of the image to finer and finer detail. These higher-frequency coefficients are increasingly unimportant to the perceived quality of the image. It is the second phase of JPEG that decides which portion of which coefficients to throw away.

Quantization Phase

The second phase of JPEG is where the compression becomes lossy. DCT does not itself lose information; it just transforms the image into a form that makes it easier to know what information to remove. (Although not lossy, *per se*, there is of course some loss of precision during the DCT phase because of the use of fixed-point arithmetic.) Quantization is easy to understand—it's simply a matter of dropping the insignificant bits of the frequency coefficients.

To see how the quantization phase works, imagine that you want to compress some whole numbers less than 100, such as 45, 98, 23, 66, and 7. If you decided that knowing these numbers truncated to the nearest multiple of 10 is sufficient for your purposes, then you could divide each number by the quantum 10 using integer arithmetic, yielding 4, 9, 2, 6, and 0. These numbers can each be encoded in 4 bits rather than the 7 bits needed to encode the original numbers.

Table 7.1.: Example JPEG Quantization Table.

Quantum							
3	5	7	9	11	13	15	17
5	7	9	11	13	15	17	19
7	9	11	13	15	17	19	21
9	11	13	15	17	19	21	23
11	13	15	17	19	21	23	25
13	15	17	19	21	23	25	27
15	17	19	21	23	25	27	29
17	19	21	23	25	27	29	31

Rather than using the same quantum for all 64 coefficients, JPEG uses a quantization table that gives the quantum to use for each of the coefficients, as specified in the formula given below. You can think of this table (`Quantum`) as a parameter that can be set to control how much information is lost and, correspondingly, how much compression is achieved. In practice, the JPEG standard specifies a set of quantization tables that have proven effective in compressing digital images; an example quantization table is given in Table 7.1. In tables like this one, the low coefficients have a quantum close to 1 (meaning that little low-frequency information is lost) and the high coefficients have larger values (meaning that more high-frequency information is lost). Notice that as a result of such quantization tables many of the high-frequency coefficients end up being set to 0 after quantization, making them ripe for further compression in the third phase.

The basic quantization equation is

```
QuantizedValue(i,j) = IntegerRound(DCT(i,j), Quantum(i,j))
```

where

```
IntegerRound(x) =
    Floor(x + 0.5) if x >= 0
    Floor(x - 0.5) if x < 0
```

Decompression is then simply defined as

```
DCT(i,j) = QuantizedValue(i,j) x Quantum(i,j)
```

For example, if the DC coefficient (i.e., DCT(0,0)) for a particular block was equal to 25, then the quantization of this value using Table 7.1 would result in

```
Floor(25/3+0.5) = 8
```

During decompression, this coefficient would then be restored as 8 × 3 = 24.

Encoding Phase

The final phase of JPEG encodes the quantized frequency coefficients in a compact form. This results in additional compression, but this compression is lossless. Starting with the DC coefficient in position (0,0), the coefficients are processed in the zigzag sequence shown in Figure 7.13. Along this zigzag, a form of run length encoding is used—RLE is applied to only the 0 coefficients, which is significant because many of the later coefficients are 0. The individual coefficient values are then encoded using a Huffman code. (The JPEG standard allows the implementer to use an arithmetic coding instead of the Huffman code.)

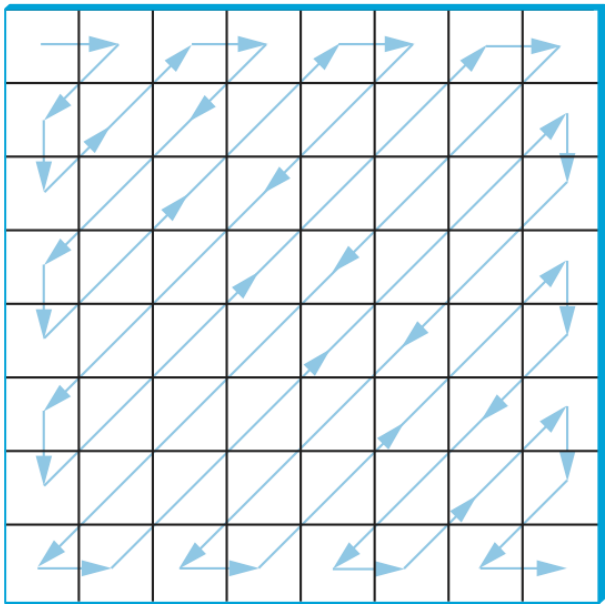


Figure 7.13.: Zigzag traversal of quantized frequency coefficients.

In addition, because the DC coefficient contains a large percentage of the information about the 8 × 8 block from the source image, and images typically change slowly from block to block, each DC coefficient is encoded as the difference from the previous DC coefficient. This is the delta encoding approach described in a later section.

JPEG includes a number of variations that control how much compression you achieve versus the fidelity of the image. This can be done, for example, by using different quantization tables. These variations, plus the fact that different images have different characteristics, make it impossible to say with any precision the compression ratios that can be achieved with JPEG. Ratios of 30:1 are common, and higher ratios are certainly possible, but *artifacts* (noticeable distortion due to compression) become more severe at higher ratios.

7.2.3 Video Compression (MPEG)

We now turn our attention to the MPEG format, named after the Moving Picture Experts Group that defined it. To a first approximation, a moving picture (i.e., video) is simply a succession of still images—also called *frames* or *pictures*—displayed at some video rate. Each of these frames can be compressed using the same DCT-based technique used in JPEG. Stopping at this point would be a mistake, however, because it fails to remove the interframe redundancy present in a video sequence. For example, two successive frames of video will contain almost identical information if there is not much motion in the scene, so it would be unnecessary to send the same information twice. Even when there is motion, there may be plenty of redundancy since a moving object may not change from one frame to the next; in some cases, only its position changes. MPEG takes this interframe redundancy into consideration. MPEG also defines a mechanism for encoding an audio signal with the video, but we consider only the video aspect of MPEG in this section.

Frame Types

MPEG takes a sequence of video frames as input and compresses them into three types of frames, called *I frames* (intrapicture), *P frames* (predicted picture), and *B frames* (bidirectional predicted picture). Each frame of input is compressed into one of these three frame types. I frames can be thought of as reference frames; they are self-contained, depending on neither earlier frames nor later frames. To a first approximation, an I frame is simply the JPEG compressed version of the corresponding frame in the video source. P and B frames are not self-contained; they specify relative differences from some reference frame. More specifically, a P frame specifies the differences from the previous I frame, while a B frame gives an interpolation between the previous and subsequent I or P frames.

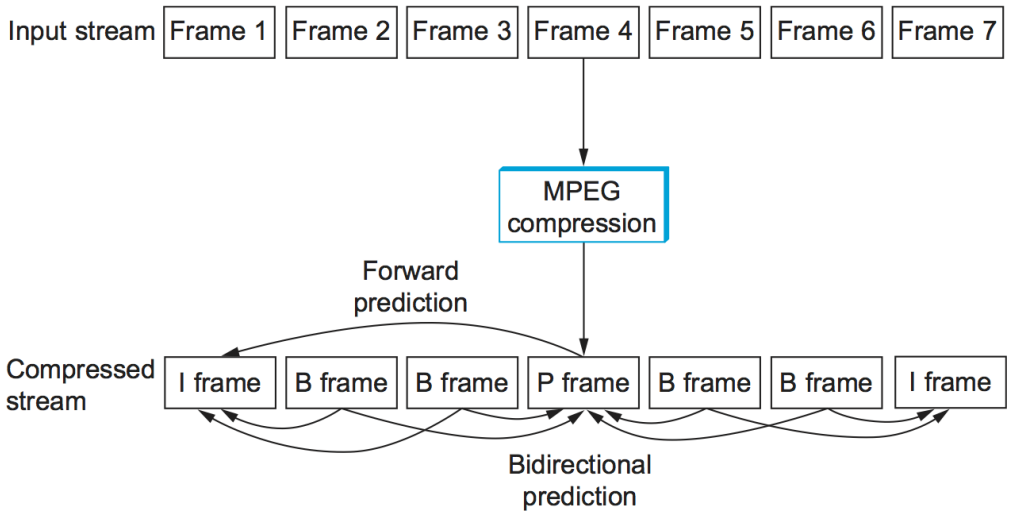


Figure 7.14.: Sequence of I, P, and B frames generated by MPEG.

Figure 7.14 illustrates a sequence of seven video frames that, after being compressed by MPEG, result in a sequence of I, P, and B frames. The two I frames stand alone; each can be decompressed at the receiver independently of any other frames. The P frame depends on the preceding I frame; it can be decompressed at the receiver only if the preceding I frame also arrives. Each of the B frames depends on both the preceding I or P frame and the subsequent I or P frame. Both of these reference frames must arrive at the receiver before MPEG can decompress the B frame to reproduce the original video frame.

Note that, because each B frame depends on a later frame in the sequence, the compressed frames are not transmitted in sequential order. Instead, the sequence I B B P B B I shown in Figure 7.14 is transmitted as I P B B I B B. Also, MPEG does not define the ratio of I frames to P and B frames; this ratio may vary depending on the required compression and picture quality. For example, it is permissible to transmit only I frames. This would be similar to using JPEG to compress the video.

In contrast to the preceding discussion of JPEG, the following focuses on the *decoding* of an MPEG stream. It is a little easier to describe, and it is the operation that is more often implemented in networking systems today, since MPEG coding is so expensive that it is frequently done offline (i.e., not in real time). For example, in a video-on-demand system, the video would be encoded and stored on disk ahead of time. When a viewer wanted to watch the video, the MPEG stream would then be transmitted to the viewer's machine, which would decode and display the stream in real time.

Let's look more closely at the three frame types. As mentioned above, I frames are approximately equal to the JPEG compressed version of the source frame. The main difference is that MPEG works in units of 16 × 16 macroblocks. For a color video represented in YUV, the U and V components in each macroblock are subsampled into an 8 × 8 block, as we discussed above in the context of JPEG. Each 2 × 2 subblock in the macroblock is given by one U value and one V value—the average of the four pixel values. The subblock still has four Y values. The relationship between a frame and the corresponding macroblocks is given in Figure 7.15.

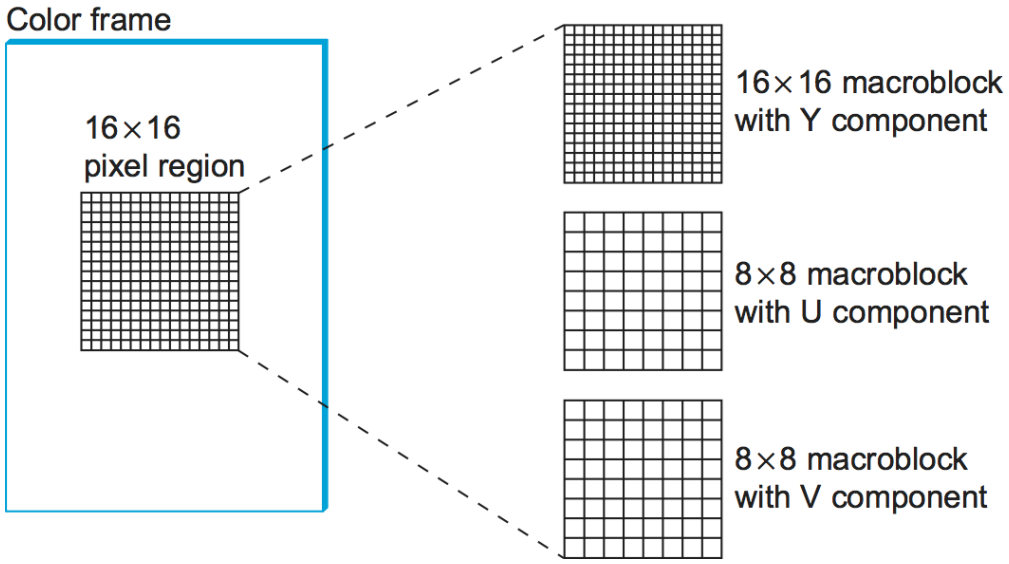


Figure 7.15.: Each frame as a collection of macroblocks.

The P and B frames are also processed in units of macroblocks. Intuitively, we can see that the information they carry for each macroblock captures the motion in the video; that is, it shows in what direction and how far the macroblock moved relative to the reference frame(s). The following describes how a B frame is used to reconstruct a frame during decompression; P frames are handled in a similar manner, except that they depend on only one reference frame instead of two.

Before getting to the details of how a B frame is decompressed, we first note that each macroblock in a B frame is not necessarily defined relative to both an earlier and a later frame, as suggested above, but may instead simply be specified relative to just one or the other. In fact, a given macroblock in a B frame can use the same intracoding as is used in an I frame. This flexibility exists because if the motion picture is

changing too rapidly then it sometimes makes sense to give the intrapicture encoding rather than a forward- or backward-predicted encoding. Thus, each macroblock in a B frame includes a type field that indicates which encoding is used for that macroblock. In the following discussion, however, we consider only the general case in which the macroblock uses bidirectional predictive encoding.

In such a case, each macroblock in a B frame is represented with a 4-tuple: (1) a coordinate for the macroblock in the frame, (2) a motion vector relative to the previous reference frame, (3) a motion vector relative to the subsequent reference frame, and (4) a delta (δ) for each pixel in the macroblock (i.e., how much each pixel has changed relative to the two reference pixels). For each pixel in the macroblock, the first task is to find the corresponding reference pixel in the past and future reference frames. This is done using the two motion vectors associated with the macroblock. Then, the delta for the pixel is added to the average of these two reference pixels. Stated more precisely, if we let F_p and F_f denote the past and future reference frames, respectively, and the past/future motion vectors are given by (x_p, y_p) and (x_f, y_f), then the pixel at coordinate *(x,y)* in the current frame (denoted F_c) is computed as

$$F_c(x, y) = (F_p(x + x_p, y + y_p) + F_f(x + x_f, y + y_f))/2 + \delta(x, y)$$

where δ is the delta for the pixel as specified in the B frame. These deltas are encoded in the same way as pixels in I frames; that is, they are run through DCT and then quantized. Since the deltas are typically small, most of the DCT coefficients are 0 after quantization; hence, they can be effectively compressed.

It should be fairly clear from the preceding discussion how encoding would be performed, with one exception. When generating a B or P frame during compression, MPEG must decide where to place the macroblocks. Recall that each macroblock in a P frame, for example, is defined relative to a macroblock in an I frame, but that the macroblock in the P frame need not be in the same part of the frame as the corresponding macroblock in the I frame—the difference in position is given by the motion vector. You would like to pick a motion vector that makes the macroblock in the P frame as similar as possible to the corresponding macroblock in the I frame, so that the deltas for that macroblock can be as small as possible. This means that you need to figure out where objects in the picture moved from one frame to the next. This is the problem of *motion estimation*, and several techniques (heuristics) for solving this problem are known. (We discuss papers that consider this problem at the end of this chapter.) The difficulty of this problem is one of the reasons why MPEG encoding takes longer than decoding on equivalent hardware. MPEG does not specify any particular technique; it only defines the format for encoding this information in B and P frames and the algorithm for reconstructing the pixel during decompression, as given above.

Effectiveness and Performance

MPEG typically achieves a compression ratio of 90:1, although ratios as high as 150:1 are not unheard of. In terms of the individual frame types, we can expect a compression ratio of approximately 30:1 for the I frames (this is consistent with the ratios achieved using JPEG when 24-bit color is first reduced to 8-bit color), while P and B frame compression ratios are typically three to five times smaller than the rates for the I frame. Without first reducing the 24 bits of color to 8 bits, the achievable compression with MPEG is typically between 30:1 and 50:1.

MPEG involves an expensive computation. On the compression side, it is typically done offline, which is not a problem for preparing movies for a video-on-demand service. Video can be compressed in real time using hardware today, but software implementations are quickly closing the gap. On the decompression side, low-cost MPEG video boards are available, but they do little more than YUV color lookup, which fortunately is the most expensive step. Most of the actual MPEG decoding is done in software. In recent years, processors

have become fast enough to keep pace with 30-frames-per-second video rates when decoding MPEG streams purely in software—modern processors can even decode MPEG streams of high definition video (HDTV).

Video Encoding Standards

We conclude by noting that MPEG is an evolving standard of significant complexity. This complexity comes from a desire to give the encoding algorithm every possible degree of freedom in how it encodes a given video stream, resulting in different video transmission rates. It also comes from the evolution of the standard over time, with the Moving Picture Experts Group working hard to retain backwards compatibility (e.g., MPEG-1, MPEG-2, MPEG-4). What we describe in this book is the essential ideas underlying MPEG-based compression, but certainly not all the intricacies involved in an international standard.

What's more, MPEG is not the only standard available for encoding video. For example, the ITU-T has also defined the *H series* for encoding real-time multimedia data. Generally, the H series includes standards for video, audio, control, and multiplexing (e.g., mixing audio, video, and data onto a single bit stream). Within the series, H.261 and H.263 were the first- and second-generation video encoding standards. In principle, both H.261 and H.263 look a lot like MPEG: They use DCT, quantization, and interframe compression. The differences between H.261/H.263 and MPEG are in the details.

Today, a partnership between the ITU-T and the MPEG group has lead to the joint H.264/MPEG-4 standard, which is used for both Blu-ray Discs and by many popular streaming sources (e.g., YouTube, Vimeo).

7.2.4 Transmitting MPEG over a Network

As we've noted, MPEG and JPEG are not just compression standards but also definitions of the format of video and images, respectively. Focusing on MPEG, the first thing to keep in mind is that it defines the format of a video *stream*; it does not specify how this stream is broken into network packets. Thus, MPEG can be used for videos stored on disk, as well as videos transmitted over a stream-oriented network connection, like that provided by TCP.

What we describe below is called the *main profile* of an MPEG video stream that is being sent over a network. You can think of an MPEG profile as being analogous to a "version," except the profile is not explicitly specified in an MPEG header; the receiver has to deduce the profile from the combination of header fields it sees.

A main profile MPEG stream has a nested structure, as illustrated in Figure 7.16. (Keep in mind that this figure hides a *lot* of messy details.) At the outermost level, the video contains a sequence of groups of pictures (GOP) separated by a `SeqHdr`. The sequence is terminated by a `SeqEndCode` (`0xb7`). The `SeqHdr` that precedes every GOP specifies—among other things—the size of each picture (frame) in the GOP (measured in both pixels and macroblocks), the interpicture period (measured in μs), and two quantization matrices for the macroblocks within this GOP: one for intracoded macroblocks (I blocks) and one for intercoded macroblocks (B and P blocks). Since this information is given for each GOP—rather than once for the entire video stream, as you might expect—it is possible to change the quantization table and frame rate at GOP boundaries throughout the video. This makes it possible to adapt the video stream over time, as we discuss below.

Each GOP is given by a `GOPHdr`, followed by the set of pictures that make up the GOP. The `GOPHdr` specifies the number of pictures in the GOP, as well as synchronization information for the GOP (i.e., when the GOP should play, relative to the beginning of the video). Each picture, in turn, is given by a

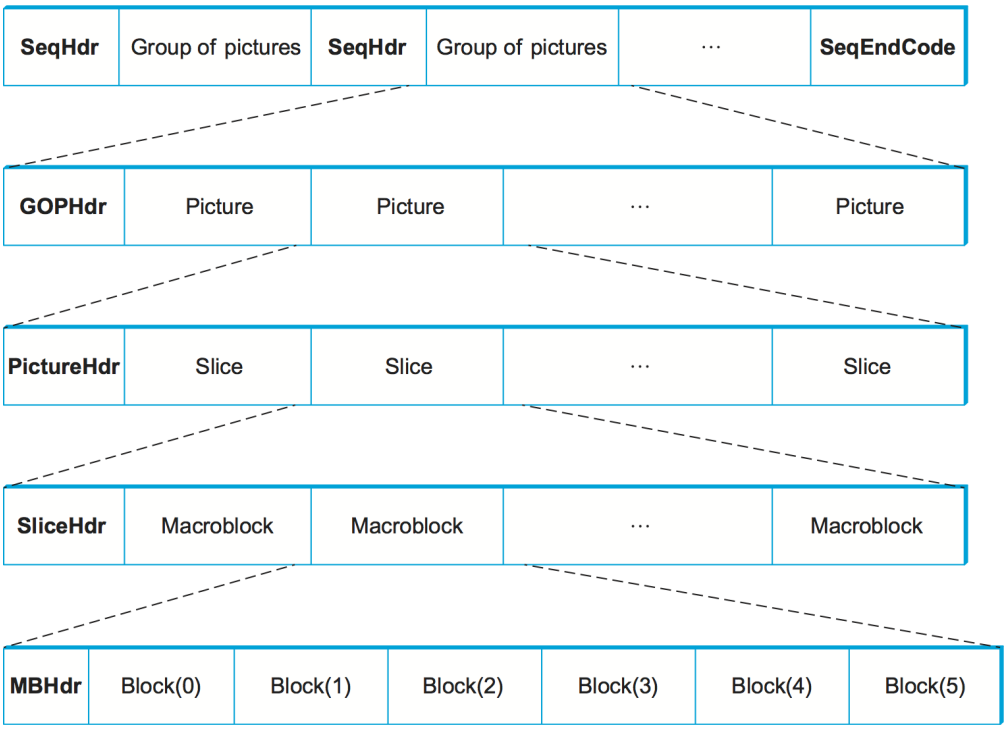


Figure 7.16.: Format of an MPEG-compressed video stream.

`PictureHdr` and a set of *slices* that make up the picture. (A slice is a region of the picture, such as one horizontal line.) The `PictureHdr` identifies the type of the picture (I, B, or P) and defines a picture-specific quantization table. The `SliceHdr` gives the vertical position of the slice, plus another opportunity to change the quantization table—this time by a constant scaling factor rather than by giving a whole new table. Next, the `SliceHdr` is followed by a sequence of macroblocks. Finally, each macroblock includes a header that specifies the block address within the picture, along with data for the six blocks within the macroblock: one for the U component, one for the V component, and four for the Y component. (Recall that the Y component is 16×16, while the U and V components are 8×8.)

It should be clear that one of the powers of the MPEG format is that it gives the encoder an opportunity to change the encoding over time. It can change the frame rate, the resolution, the mix of frame types that define a GOP, the quantization table, and the encoding used for individual macroblocks. As a consequence, it is possible to adapt the rate at which a video is transmitted over a network by trading picture quality for network bandwidth. Exactly how a network protocol might exploit this adaptability is currently a subject of research (see sidebar).

Another interesting aspect of sending an MPEG stream over the network is exactly how the stream is broken into packets. If sent over a TCP connection, packetization is not an issue; TCP decides when it has enough bytes to send the next IP datagram. When using video interactively, however, it is rare to transmit it over TCP, since TCP has several features that are ill suited to highly latency-sensitive applications (such as abrupt rate changes after a packet loss and retransmission of lost packets). If we are transmitting video using UDP, say, then it makes sense to break the stream at carefully selected points, such as at macroblock boundaries. This is because we would like to confine the effects of a lost packet to a single macroblock, rather than damaging several macroblocks with a single loss. This is an example of Application Level Framing, which was discussed in an earlier chapter.

Packetizing the stream is only the first problem in sending MPEG-compressed video over a network. The next complication is dealing with packet loss. On the one hand, if a B frame is dropped by the network, then it is possible to simply replay the previous frame without seriously compromising the video; 1 frame out of 30 is no big deal. On the other hand, a lost I frame has serious consequences—none of the subsequent B and P frames can be processed without it. Thus, losing an I frame would result in losing multiple frames of the video. While you could retransmit the missing I frame, the resulting delay would probably not be acceptable in a real-time videoconference. One solution to this problem would be to use the Differentiated Services techniques described in the previous chapter to mark the packets containing I frames with a lower drop probability than other packets.

One final observation is that how you choose to encode video depends on more than just the available network bandwidth. It also depends on the application's latency constraints. Once again, an interactive application like videoconferencing needs small latencies. The critical factor is the combination of I, P, and B frames in the GOP. Consider the following GOP:

I B B B B P B B B B I

The problem this GOP causes a videoconferencing application is that the sender has to delay the transmission of the four B frames until the P or I that follows them is available. This is because each B frame depends on the subsequent P or I frame. If the video is playing at 15 frames per second (i.e., one frame every 67 ms), this means the first B frame is delayed 4×67 ms, which is more than a quarter of a second. This delay is in addition to any propagation delay imposed by the network. A quarter of a second is far greater than the 100-ms threshold that humans are able to perceive. It is for this reason that many videoconference applications encode video using JPEG, which is often called motion-JPEG. (Motion-JPEG also addresses the problem of dropping a reference frame since all frames are able to stand alone.) Notice, however, that an interframe encoding that depends upon only prior frames rather than later frames is not a problem. Thus, a GOP of

I P P P P I

would work just fine for interactive videoconferencing.

Adaptive Streaming

Because encoding schemes like MPEG allow for a trade-off between the bandwidth consumed and the quality of the image, there is an opportunity to adapt a video stream to match the available network bandwidth. This is effectively what video streaming services like Netflix do today.

For starters, let's assume that we have some way to measure the amount of free capacity and level of congestion along a path, for example, by observing the rate at which packets are successfully arriving at the destination. As the available bandwidth fluctuates, we can feed that information back to the codec so that it adjusts its coding parameters to back off during congestion and to send more aggressively (with a higher picture quality) when the network is idle. This is analogous to the behavior of TCP, except in the video case we are actually modifying the total amount of data sent rather than how long we take to send a fixed amount of data, since we don't want to introduce delay into a video application.

In the case of video-on-demand services like Netflix, we don't adapt the encoding on the fly, but instead we encode a handful of video quality levels ahead of time, and save them to files named accordingly. The receiver simply changes the file name it requests to match the quality its measurements indicate the network will be able to deliver. The receiver watches its playback queue, and asks for a higher quality encoding when the queue becomes too full and a lower quality encoding when the queue becomes too empty.

How does this approach know where in the movie to jump to should the requested quality change? In effect, the receiver never asks the sender to stream the whole movie, but instead it requests a sequence of short movie segments, typically a few seconds long (and always on GOP boundary). Each segment is an opportunity to change the quality level to match what the network is able to deliver. (It turns out that requesting movie chunks also makes it easier to implement *trick play*, jumping around from one place to another in the movie.) In other words, a movie is typically stored as a set of N × M chunks (files): N quality levels for each of M segments.

There's one last detail. Since the receiver is effectively requesting a sequence of discrete video chunks by name, the most common approach for issuing these requests is to use HTTP. Each chuck is a separate HTTP GET request with the URL identifying the specific chunk the receiver wants next. When you start downloading a movie, your video player first downloads a *manifest* file that contains nothing more than the URLs for the N × M chunks in the movie, and then it issues a sequence of HTTP requests using the appropriate URL for the situation. This general approach is called *HTTP adaptive streaming*, although it has been standardized in slightly different ways by various organizations, most notably MPEG's DASH (*Dynamic Adaptive Streaming over HTTP*) and Apple's HLS (*HTTP Live Streaming*).

7.2.5 Audio Compression (MP3)

Not only does MPEG define how video is compressed, but it also defines a standard for compressing audio. This standard can be used to compress the audio portion of a movie (in which case the MPEG standard defines how the compressed audio is interleaved with the compressed video in a single MPEG stream) or it can be used to compress stand-alone audio (for example, an audio CD).

To understand audio compression, we need to begin with the data. CD-quality audio, which is the *de facto* digital representation for high-quality audio, is sampled at a rate of 44.1 KHz (i.e., a sample is collected approximately once every 23 μs). Each sample is 16 bits, which means that a stereo (2-channel) audio stream results in a bit rate of

$$2 \times 44.1 \times 1000 \times 16 = 1.41\ Mbps$$

By comparison, telephone-quality voice is sampled at a rate of 8 KHz, with 8-bit samples, resulting in a bit rate of 64 kbps.

Clearly, some amount of compression is going to be required to transmit CD-quality audio over, say, the 128-kbps capacity of an ISDN data/voice line pair. To make matters worse, synchronization and error correction overhead require that 49 bits be used to encode each 16-bit sample, resulting in an actual bit rate of

$$49/16 \times 1.41\ Mbps = 4.32\ Mbps$$

MPEG addresses this need by defining three levels of compression, as enumerated in Table 7.2. Of these, Layer III, which is more widely known as MP3, is the most commonly used.

Table 7.2.: MP3 Compression Rates.

Coding	Bit Rates	Compression Factor
Layer I	384 kbps	14
Layer II	192 kbps	18
Layer III	128 kbps	12

To achieve these compression ratios, MP3 uses techniques that are similar to those used by MPEG to compress video. First, it splits the audio stream into some number of frequency subbands, loosely analogous to the way MPEG processes the Y, U, and V components of a video stream separately. Second, each subband is broken into a sequence of blocks, which are similar to MPEG's macroblocks except they can vary in length from 64 to 1024 samples. (The encoding algorithm can vary the block size depending on certain distortion effects that are beyond our discussion.) Finally, each block is transformed using a modified DCT algorithm, quantized, and Huffman encoded, just as for MPEG video.

The trick to MP3 is how many subbands it elects to use and how many bits it allocates to each subband, keeping in mind that it is trying to produce the highest-quality audio possible for the target bit rate. Exactly how this allocation is made is governed by psychoacoustic models that are beyond the scope of this book, but to illustrate the idea consider that it makes sense to allocate more bits to low-frequency subbands when compressing a male voice and more bits to high-frequency subbands when compressing a female voice. Operationally, MP3 dynamically changes the quantization tables used for each subband to achieve the desired effect.

Once compressed, the subbands are packaged into fixed-size frames, and a header is attached. This header includes synchronization information, as well as the bit allocation information needed by the decoder to determine how many bits are used to encode each subband. As mentioned above, these audio frames can then be interleaved with video frames to form a complete MPEG stream. One interesting side note is that, while it might work to drop B frames in the network should congestion occur, experience teaches us that it is not a good idea to drop audio frames since users are better able to tolerate bad video than bad audio.

Perspective: Big Data and Analytics

This chapter is about data, and since no topic in Computer Science is receiving more attention than *big data* (or alternatively, *data analytics*), a natural question is what relationship there might be between big data and computer networks. Although the term is often used informally by the popular press, a working definition is quite simple: sensor data is collected by monitoring some physical or man-made system and then analyzed for insights using the statistical methods of Machine Learning. Because the amount of raw data that's collected is often voluminous, the "big" qualifier is applied. So are there any implications to networking?

At first blush, networks are purposely designed to be data-agnostic. If you collect it and want to ship to somewhere for analysis, the network is happy to do that for you. You might compress the data to reduce the bandwidth required to transmit it, but otherwise big data is no different than plain old regular data. But this ignores two important factors.

The first is that while the network doesn't care about the meaning of the data (i.e., what the bits represent), it does concern itself with the volume of data. This impacts the access network in particular, which has been engineered to favor download speeds over upload speeds. That bias makes sense when the dominant use case is video that flows out to end-users, but in a world where your car, every appliance in your house, and the drones flying over your city are all reporting data back into the network (uploaded into the cloud), the situation is reversed. In fact, the amount of data being generated by *Autonomous Vehicles* and the *Internet-of-Things (IoT)* is potentially overwhelming.

While one could imagine dealing with this problem by using one of the compression algorithms described in *Section 7.2*, people are instead thinking outside the box, and pursuing new applications that reside at the edge of the network. These *edge-native applications* both provide better sub-millisecond response time and

they dramatically reduce the volume of data that ultimately needs to be uploaded into the cloud. You can think of this data reduction as application-specific compression, but it's more accurate to say that the edge application needs only write summaries of the data, not the raw data, back to the cloud.

We introduced the access-edge cloud technology needed to support edge-native applications at the end of *Chapter 2*, but what is perhaps more interesting is to look at some examples of edge-native applications. One such example is enterprises in the automotive, factory, and warehouse space increasingly want to deploy private 5G networks for a variety of *physical automation* use cases. These include a garage where a remote valet parks your car or a factory floor making use of automation robots. The common theme is high bandwidth, low latency connectivity from the robot to intelligence sitting nearby in an edge cloud. This drives lower robot costs (you don't need to place heavy compute on each one) and enables robot swarms and coordination more scalably.

Another illustrative example is *Wearable Cognitive Assistance*. The idea is to generalize what navigation software does for us: it uses one sensor (GPS), gives us step-by-step guidance on a complex task (getting around an unknown city), catches our errors promptly, and helps us recover. Can we generalize this metaphor? Could a person wearing a device (e.g., Google Glass, Microsoft Hololens) be guided step-by-step on a complex task, perhaps for the first time? The system would effectively act as "an angel on your shoulder." All the sensors on the device (e.g., video, audio, accelerometer, gyroscope) are streamed over wireless (possibly after some device preprocessing) to a nearby edge-cloud that performs the heavy lifting. This is a human-in-the-loop metaphor, with the "look and feel of augmented reality" but implemented by AI algorithms (e.g., computer vision, natural language recognition.)

The second factor is that because a network is like many other man-made systems, it is possible to collect data about its behavior (e.g., performance, failures, traffic patterns), apply analytics programs to that data, and use the insights gained to improve the network. It should not come as a surprise that this is an active area of research, with the goal of building a closed control loop. Setting aside the analytics itself, which are well outside the scope of this book, the interesting questions are (1) what useful data can we collect, and (2) what aspects of the network are most promising to control? Let's look at two promising answers.

One is 5G cellular networks, which are inherently complex. They include multiple layers of virtual functions, virtual and physical RAN assets, spectrum usage, and as we have just discussed, edge computing nodes. It is widely expected that network analytics will be essential to building a flexible 5G network. This will include network planning, which will need to decide where to scale specific network functions and application services based on machine learning algorithms that analyze network utilization and traffic data patterns.

A second is *In-band Network Telemetry* (INT), a framework to collect and report network state, directly in the data plane. This is in contrast to the conventional reporting done by the network control plane, as typified by the example systems described in *Section 9.3*. In the INT architecture, packets contain header fields that are interpreted as "telemetry instructions" by network devices. These instructions tell an INT-capable device what state to collect and write into the packet as it transits the network. INT *traffic sources* (e.g., applications, end-host networking stacks, VM hypervisors) can embed the instructions either in normal data packets or in special probe packets. Similarly, INT *traffic sinks* retrieve (and optionally report) the collected results of these instructions, allowing the traffic sinks to monitor the exact data plane state that the packets "observed" while being forwarded. INT is still early-stage, and takes advantage of the programmable pipelines described in *Section 3.5*, but it has the potential to provide a qualitatively deeper insights into traffic patterns and the root causes of network failures.

Broader Perspective

To continue reading about the cloudification of the Internet, see *Perspective: Blockchain and a Decentralized Internet*.

To learn more about promising edge-native applications, we recommend: Open Edge Computing Initiative, 2019.

To learn more about In-band Network Telemetry, we recommend: In-band Network Telemetry via Programmable Dataplanes, August 2015.

CHAPTER

EIGHT

NETWORK SECURITY

It is true greatness to have in one the frailty of a man and the security of a god.

—*Seneca*

Problem: Security Attacks

Computer networks are typically a shared resource used by many applications representing different interests. The Internet is particularly widely shared, being used by competing businesses, mutually antagonistic governments, and opportunistic criminals. Unless security measures are taken, a network conversation or a distributed application may be compromised by an adversary.

Consider, for example, some threats to secure use of the web. Suppose you are a customer using a credit card to order an item from a website. An obvious threat is that an adversary would eavesdrop on your network communication, reading your messages to obtain your credit card information. How might that eavesdropping be accomplished? It is trivial on a broadcast network such as an Ethernet or Wi-Fi, where any node can be configured to receive all the message traffic on that network. More elaborate approaches include wiretapping and planting spy software on any of the chain of nodes involved. Only in the most extreme cases (e.g.,national security) are serious measures taken to prevent such monitoring, and the Internet is not one of those cases. It is possible and practical, however, to encrypt messages so as to prevent an adversary from understanding the message contents. A protocol that does so is said to provide *confidentiality*. Taking the concept a step farther, concealing the quantity or destination of communication is called *traffic confidentiality*—because merely knowing how much communication is going where can be useful to an adversary in some situations.

Even with confidentiality there still remains threats for the website customer. An adversary who can't read the contents of your encrypted message might still be able to change a few bits in it, resulting in a valid order for, say, a completely different item or perhaps 1000 units of the item. There are techniques to detect, if not prevent, such tampering. A protocol that detects such message tampering is said to provide *integrity*.

Another threat to the customer is unknowingly being directed to a false website. This can result from a Domain Name System (DNS) attack, in which false information is entered in a DNS server or the name service cache of the customer's computer. This leads to translating a correct URL into an incorrect IP address—the address of a false website. A protocol that ensures that you really are talking to whom you think you're talking is said to provide *authentication*. Authentication entails integrity, since it is meaningless to say that a message came from a certain participant if it is no longer the same message.

The owner of the website can be attacked as well. Some websites have been defaced; the files that make up the website content have been remotely accessed and modified without authorization. That is an issue of *access control*: enforcing the rules regarding who is allowed to do what. Websites have also been subject to denial of service (DoS) attacks, during which would-be customers are unable to access the website because it is being overwhelmed by bogus requests. Ensuring a degree of access is called *availability*.

In addition to these issues, the Internet has notably been used as a means for deploying malicious code, generally called *malware*, that exploits vulnerabilities in end systems. *Worms*, pieces of self-replicating code that spread over networks, have been known for several decades and continue to cause problems, as do their relatives, *viruses*, which are spread by the transmission of infected files. Infected machines can then be arranged into *botnets*, which can be used to inflict further harm, such as launching DoS attacks.

8.1 Trust and Threats

Before we address the how's and why's of building secure networks, it is important to establish one simple truth: We will inevitably fail. This is because security is ultimately an exercise in making assumptions about trust, evaluating threats, and mitigating risk. There is no such thing as perfect security.

Trust and threats are two sides of the same coin. A threat is a potential failure scenario that you design your system to avoid, and trust is an assumption you make about how external actors and internal components you build upon will behave. For example, if you are transmitting a message over WiFi on an open campus, you would likely identify an eavesdropper that can intercept the message as a threat (and adopt some of the methods discussed in this chapter as a countermeasure), but if you are transmitting a message over a fiber link between two machines in a locked datacenter, you might trust that channel is secure, and so take no additional steps.

You could argue that since you already have a way to protect WiFi-based communication you just as well use it to protect the fiber-based channel, but that presumes the outcome of a cost/benefit analysis. Suppose protecting any message, whether sent over WiFi or fiber, slows the communication down by 10% due to the overhead of encryption. If you need to squeeze every last ounce of performance out of a scientific computation (e.g., you are trying to model a hurricane) and the odds of someone breaking into the datacenter are one in a million (and even if they did, the data being transmitted has little value), then you would be well-justified in not securing the fiber communication channel.

These sorts of calculations happen all the time, although they are often implicit and unstated. For example, you may run the world's most secure encryption algorithm on a message before transmitting it, but you've implicitly trusted that the server you're running on is both faithfully executing that algorithm and not leaking a copy of your unencrypted message to an adversary. Do you treat this as a threat or do you trust that the server does not misbehave? At the end of the day, the best you can do is mitigate risk: identify those threats that you can eliminate in a cost effective way, and be explicit about what trust assumptions you are making so you aren't caught off-guard by changing circumstances, such as an ever more determined or sophisticated adversary.

In this particular example, the threat of an adversary compromising a server has become quite real as more of our computations move from local servers into the cloud, and so research is now going into building a *Trusted Computing Base* (TCB), an interesting topic, but one that is in the realm of computer architecture rather than computer networks. For the purpose of this chapter, our recommendation is to pay attention to the words *trust* and *threat* (or adversary), as they are key to understanding the context in which security claims are made.

There is one final historical note that helps set the table for this chapter. The Internet (and the ARPANET before it) where funded by the U.S. Department of Defense, an organization that certainly understands threat analysis. The original assessment was dominated by concerns about the network surviving in the face of routers and networks failing (or being destroyed), which explains why the routing algorithms are decentralized, with no central point of failure. On the other hand, the original design assumed all actors *inside* the network were trusted, and so little or no attention was paid what today we would call cybersecurity (attacks from bad actors that are able to connect to the network). What this means is that many of the tools described in this chapter could be considered patches. They are strongly-grounded in cryptography, but "add-ons" nonetheless. If a comprehensive redesign of the Internet were to take place, integrating security would likely be the foremost driving factor.

8.2 Cryptographic Building Blocks

We introduce the concepts of cryptography-based security step by step. The first step is the cryptographic algorithms—ciphers and cryptographic hashes—that are introduced in this section. They are not a solution in themselves, but rather building blocks from which a solution can be built. Cryptographic algorithms are parameterized by *keys*, and a later section then addresses the problem of distributing the keys. In the next step, we describe how to incorporate the cryptographic building blocks into protocols that provide secure communication between participants who possess the correct keys. A final section then examines several complete security protocols and systems in current use.

8.2.1 Principles of Ciphers

Encryption transforms a message in such a way that it becomes unintelligible to any party that does not have the secret of how to reverse the transformation. The sender applies an *encryption* function to the original *plaintext* message, resulting in a *ciphertext* message that is sent over the network, as shown in Figure 8.1. The receiver applies a secret *decryption* function—the inverse of the encryption function—to recover the original plaintext. The ciphertext transmitted across the network is unintelligible to any eavesdropper, assuming the eavesdropper doesn't know the decryption function. The transformation represented by an encryption function and its corresponding decryption function is called a *cipher*.

Cryptographers have been led to the principle, first stated in 1883, that encryption and decryption functions should be parameterized by a *key*, and furthermore that the functions should be considered public knowledge—only the key need be secret. Thus, the ciphertext produced for a given plaintext message depends on both the encryption function and the key. One reason for this principle is that if you depend on the cipher being kept secret, then you have to retire the cipher (not just the keys) when you believe it is no longer secret. This means potentially frequent changes of cipher, which is problematic since it takes a lot of work to develop a new cipher. Also, one of the best ways to know that a cipher is secure is to use it for a long time—if no one breaks it, it's probably secure. (Fortunately, there are plenty of people who will try to break ciphers and who will let it be widely known when they have succeeded, so no news is generally good news.) Thus, there is considerable cost and risk in deploying a new cipher. Finally, parameterizing a cipher with keys provides us with what is in effect a very large family of ciphers; by switching keys, we essentially switch ciphers, thereby limiting the amount of data that a *cryptanalyst* (code-breaker) can use to try to break our key/cipher and the amount she can read if she succeeds.

The basic requirement for an encryption algorithm is that it turn plaintext into ciphertext in such a way that only the intended recipient—the holder of the decryption key—can recover the plaintext. What this means

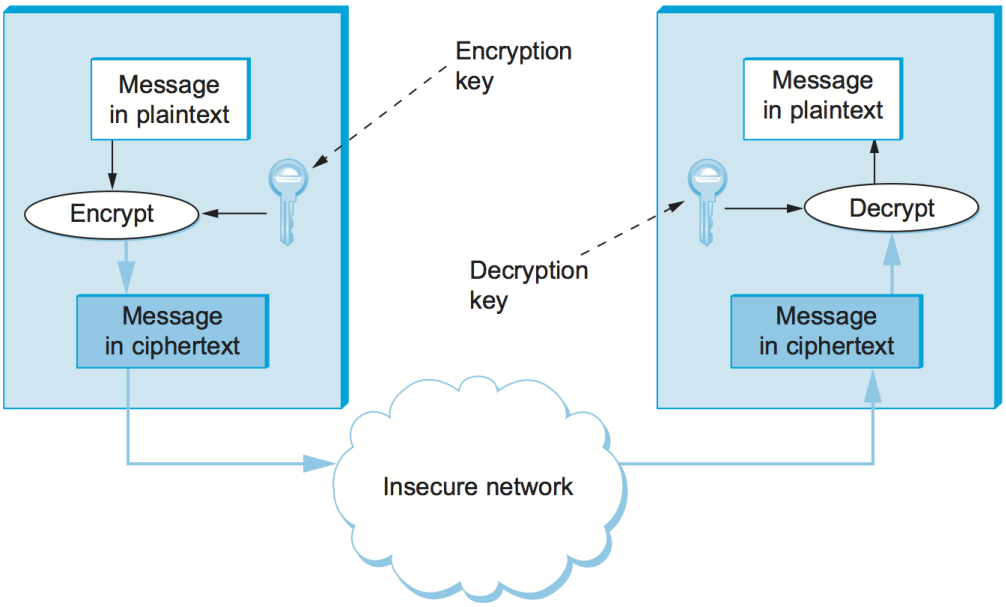


Figure 8.1.: Secret-key encryption and decryption.

is that encrypted messages cannot be read by people who do not hold the key.

It is important to realize that when a potential attacker receives a piece of ciphertext, he may have more information at his disposal than just the ciphertext itself. For example, he may know that the plaintext was written in English, which means that the letter *e* occurs more often in the plaintext that any other letter; the frequency of many other letters and common letter combinations can also be predicted. This information can greatly simplify the task of finding the key. Similarly, he may know something about the likely contents of the message; for example, the word "login" is likely to occur at the start of a remote login session. This may enable a *known plaintext* attack, which has a much higher chance of success than a *ciphertext only* attack. Even better is a *chosen plaintext* attack, which may be enabled by feeding some information to the sender that you know the sender is likely to transmit—such things have happened in wartime, for example.

The best cryptographic algorithms, therefore, can prevent the attacker from deducing the key even when the individual knows both the plaintext and the ciphertext. This leaves the attacker with no choice but to try all the possible keys—exhaustive, "brute force" search. If keys have n bits, then there are 2^n possible values for a key (each of the n bits could be either a zero or a one). An attacker could be so lucky as to try the correct value immediately, or so unlucky as to try every incorrect value before finally trying the correct value of the key, having tried all 2^n possible values; the average number of guesses to discover the correct value is halfway between those extremes, $2^{n/2}$. This can be made computationally impractical by choosing a sufficiently large key space and by making the operation of checking a key reasonably costly. What makes this difficult is that computing speeds keep increasing, making formerly infeasible computations feasible. Furthermore, although we are concentrating on the security of data as it moves through the network—that is, the data is sometimes vulnerable for only a short period of time—in general, security people have to consider the vulnerability of data that needs to be stored in archives for tens of years. This argues for a generously large key size. On the other hand, larger keys make encryption and decryption slower.

Most ciphers are *block ciphers*; they are defined to take as input a plaintext block of a certain fixed size, typically 64 to 128 bits. Using a block cipher to encrypt each block independently—known as *electronic codebook (ECB) mode* encryption—has the weakness that a given plaintext block value will always result

in the same ciphertext block. Hence, recurring block values in the plaintext are recognizable as such in the ciphertext, making it much easier for a cryptanalyst to break the cipher.

To prevent this, block ciphers are always augmented to make the ciphertext for a block vary depending on context. Ways in which a block cipher may be augmented are called *modes of operation*. A common mode of operation is *cipher block chaining* (CBC), in which each plaintext block is XORed with the previous block's ciphertext before being encrypted. The result is that each block's ciphertext depends in part on the preceding blocks (i.e., on its context). Since the first plaintext block has no preceding block, it is XORed with a random number. That random number, called an *initialization vector* (IV), is included with the series of ciphertext blocks so that the first ciphertext block can be decrypted. This mode is illustrated in Figure 8.2. Another mode of operation is *counter mode*, in which successive values of a counter (e.g., 1, 2, 3, . . .) are incorporated into the encryption of successive blocks of plaintext.

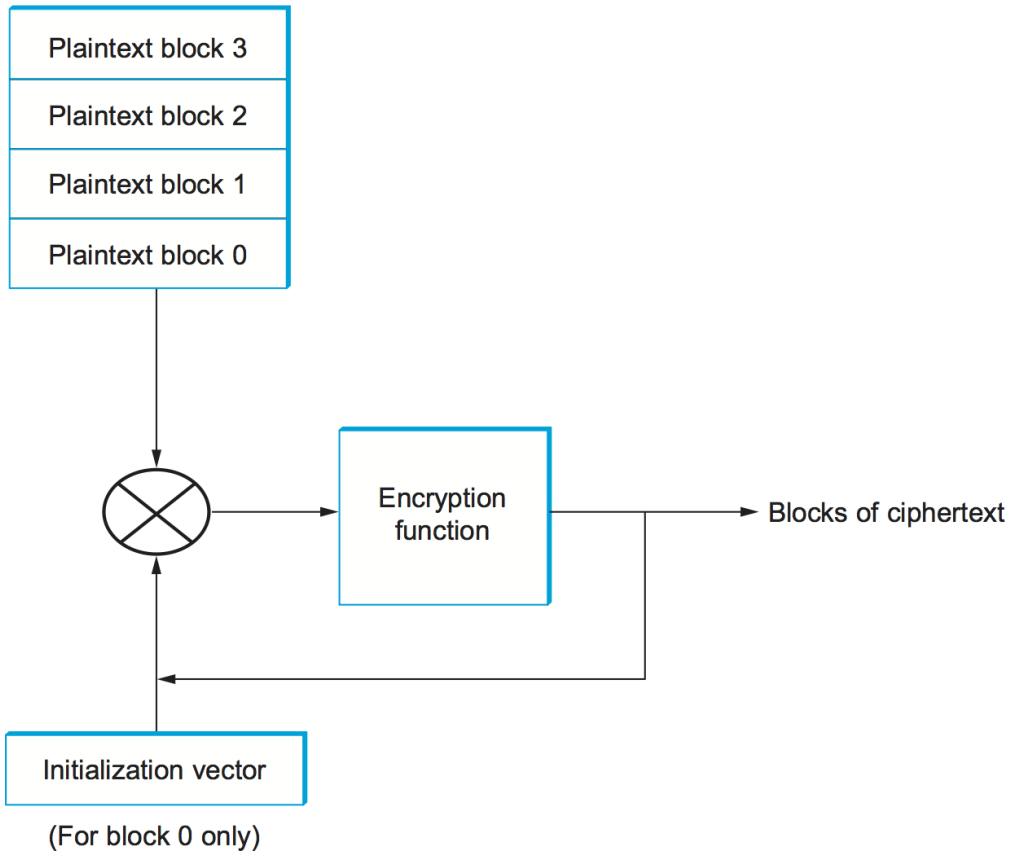


Figure 8.2.: Cipher Block Chaining.

8.2.2 Secret-Key Ciphers

In a secret-key cipher, both participants in a communication share the same key.[1] In other words, if a message is encrypted using a particular key, the same key is required for decrypting the message. If the cipher illustrated in Figure 8.1 were a secret-key cipher, then the encryption and decryption keys would be identical. Secret-key ciphers are also known as symmetric-key ciphers since the secret is shared with both

[1] We use the term *participant* for the parties involved in a secure communication since that is the term we have been using throughout the book to identify the two endpoints of a channel. In the security world, they are typically called *principals*.

participants. We'll take a look at the alternative, public-key ciphers, shortly. (Public-key cipers are known as also asymmetric-key ciphers, since as we'll soon se, the two participants use different keys.)

The U.S. National Institute of Standards and Technology (NIST) has issued standards for a series of secret-key ciphers. *Data Encryption Standard* (DES) was the first, and it has stood the test of time in that no cryptanalytic attack better than brute force search has been discovered. Brute force search, however, has gotten faster. DES's keys (56 independent bits) are now too small given current processor speeds. DES keys have 56 independent bits (although they have 64 bits in total; the last bit of every byte is a parity bit). As noted above, you would, on average, have to search half of the space of 2^{56} possible keys to find the right one, giving $2^{55} = 3.6 \times 10^{16}$ keys. That may sound like a lot, but such a search is highly parallelizable, so it's possible to throw as many computers at the task as you can get your hands on—and these days it's easy to lay your hands on thousands of computers. (Amazon will rent them to you for a few cents an hour.) By the late 1990s, it was already possible to recover a DES key after a few hours. Consequently, NIST updated the DES standard in 1999 to indicate that DES should only be used for legacy systems.

NIST also standardized the cipher *Triple DES* (3DES), which leverages the cryptanalysis resistance of DES while in effect increasing the key size. A 3DES key has 168 ($= 3 \times 56$) independent bits, and is used as three DES keys; let's call them DES-key1, DES-key2, and DES-key3. 3DES encryption of a block is performed by first DES encrypting the block using DES-key1, then DES *de*crypting the result using DES-key2, and finally DES encrypting that result using DES-key3. Decryption involves decrypting using DES-key3, then encrypting using DES-key2, then decrypting using DES-key1.

The reason 3DES encryption uses DES *de*cryption with DES-key2 is to interoperate with legacy DES systems. If a legacy DES system uses a single key, then a 3DES system can perform the same encryption function by using that key for each of DES-key1, DES-key2, and DES-key3; in the first two steps, we encrypt and then decrypt with the same key, producing the original plaintext, which we then encrypt again.

Although 3DES solves DES's key-length problem, it inherits some other shortcomings. Software implementations of DES/3DES are slow because it was originally designed by IBM for implementation in hardware. Also, DES/3DES uses a 64-bit block size; a larger block size is more efficient and more secure.

3DES is now being superseded by the *Advanced Encryption Standard* (AES) standard issued by NIST. The cipher underlying AES (with a few minor modifications) was originally named Rijndael (pronounced roughly like "Rhine dahl") based on the names of its inventors, Daemen and Rijmen. AES supports key lengths of 128, 192, or 256 bits, and the block length is 128 bits. AES permits fast implementations in both software and hardware. It doesn't require much memory, which makes it suitable for small mobile devices. AES has some mathematically proven security properties and, as of the time of writing, has not suffered from any significant successful attacks.

8.2.3 Public-Key Ciphers

An alternative to secret-key ciphers is public-key, ciphers. Instead of a single key shared by two participants, a public-key cipher uses a pair of related keys, one for encryption and a different one for decryption. The pair of keys is "owned" by just one participant. The owner keeps the decryption key secret so that only the owner can decrypt messages; that key is called the *private key*. The owner makes the encryption key public, so that anyone can encrypt messages for the owner; that key is called the *public key*. Obviously, for such a scheme to work, it must not be possible to deduce the private key from the public key. Consequently, any participant can get the public key and send an encrypted message to the owner of the keys, and only the owner has the private key necessary to decrypt it. This scenario is depicted in Figure 8.3.

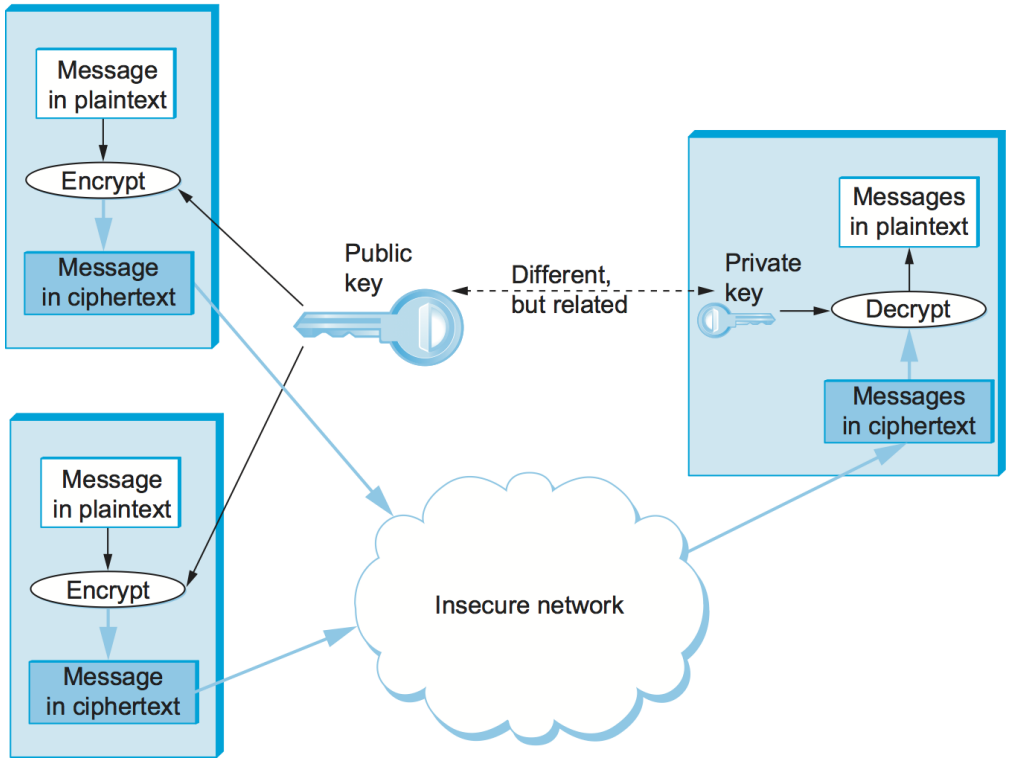


Figure 8.3.: Public-key encryption.

Because it is somewhat unintuitive, we emphasize that the public encryption key is useless for decrypting a message—you couldn't even decrypt a message that you yourself had just encrypted unless you had the private decryption key. If we think of keys as defining a communication channel between participants, then another difference between public-key and secret-key ciphers is the topology of the channels. A key for a secret-key cipher provides a channel that is two-way between two participants—each participant holds the same (symmetric) key that either one can use to encrypt or decrypt messages in either direction. A public/private key pair, in contrast, provides a channel that is one way and many-to-one: from everyone who has the public key to the unique owner of the private key, as illustrated in Figure 8.3.

An important additional property of public-key ciphers is that the private "decryption" key can be used with the encryption algorithm to encrypt messages so that they can only be decrypted using the public "encryption" key. This property clearly wouldn't be useful for confidentiality since anyone with the public key could decrypt such a message. (Indeed, for two-way confidentiality between two participants, each participant needs its own pair of keys, and each encrypts messages using the other's public key.) This property is, however, useful for authentication since it tells the receiver of such a message that it could only have been created by the owner of the keys (subject to certain assumptions that we will get into later). This is illustrated in Figure 8.4. It should be clear from the figure that anyone with the public key can decrypt the encrypted message, and, assuming that the result of the decryption matches the expected result, it can be concluded that the private key must have been used to perform the encryption. Exactly how this operation is used to provide authentication is the topic of a later section. As we will see, public-key ciphers are used primarily for authentication and to confidentially distribute secret (symmetric) keys, leaving the rest of confidentiality to secret-key ciphers.

A bit of interesting history: The concept of public-key ciphers was first published in 1976 by Diffie and

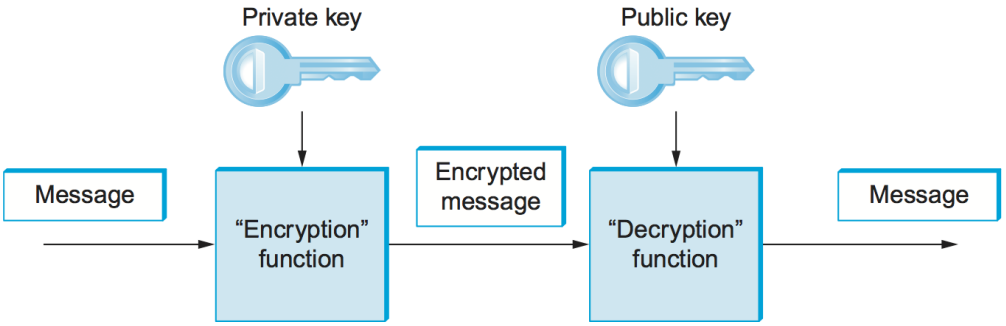


Figure 8.4.: Authentication using public keys.

Hellman. Subsequently, however, documents have come to light proving that Britain's Communications-Electronics Security Group had discovered public-key ciphers by 1970, and the U.S. National Security Agency (NSA) claims to have discovered them in the mid-1960s.

The best-known public-key cipher is RSA, named after its inventors: Rivest, Shamir, and Adleman. RSA relies on the high computational cost of factoring large numbers. The problem of finding an efficient way to factor numbers is one that mathematicians have worked on unsuccessfully since long before RSA appeared in 1978, and RSA's subsequent resistance to cryptanalysis has further bolstered confidence in its security. Unfortunately, RSA needs relatively large keys, at least 1024 bits, to be secure. This is larger than keys for secret-key ciphers because it is faster to break an RSA private key by factoring the large number on which the pair of keys is based than by exhaustively searching the key space.

Another public-key cipher is ElGamal. Like RSA, it relies on a mathematical problem, the discrete logarithm problem, for which no efficient solution has been found, and requires keys of at least 1024 bits. There is a variation of the discrete logarithm problem, arising when the input is an elliptic curve, that is thought to be even more difficult to compute; cryptographic schemes based on this problem are referred to as *elliptic curve cryptography*.

Public-key ciphers are, unfortunately, several orders of magnitude slower than secret-key ciphers. Consequently, secret-key ciphers are used for the vast majority of encryption, while public-key ciphers are reserved for use in authentication and session key establishment.

8.2.4 Authenticators

Encryption alone does not provide data integrity. For example, just randomly modifying a ciphertext message could turn it into something that decrypts into valid-looking plaintext, in which case the tampering would be undetectable by the receiver. Nor does encryption alone provide authentication. It is not much use to say that a message came from a certain participant if the contents of the message have been modified after that participant created it. In a sense, integrity and authentication are fundamentally inseparable.

An *authenticator* is a value, to be included in a transmitted message, that can be used to verify simultaneously the authenticity and the data integrity of a message. We will see how authenticators can be used in protocols. For now, we focus on the algorithms that produce authenticators.

You may recall that checksums and cyclic redundancy checks (CRCs) are pieces of information added to a message so the receiver detect when the message has been inadvertently modified by bit errors. A similar concept applies to authenticators, with the added challenge that the corruption of the message is likely to be

deliberately performed by someone who wants the corruption to go undetected. To support authentication, an authenticator includes some proof that whoever created the authenticator knows a secret that is known only to the alleged sender of the message; for example, the secret could be a key, and the proof could be some value encrypted using the key. There is a mutual dependency between the form of the redundant information and the form of the proof of secret knowledge. We discuss several workable combinations.

We initially assume that the original message need not be confidential—that a transmitted message will consist of the plaintext of the original message plus an authenticator. Later we will consider the case where confidentiality is desired.

One kind of authenticator combines encryption and a *cryptographic hash function*. Cryptographic hash algorithms are treated as public knowledge, as with cipher algorithms. A cryptographic hash function (also known as a *cryptographic checksum*) is a function that outputs sufficient redundant information about a message to expose any tampering. Just as a checksum or CRC exposes bit errors introduced by noisy links, a cryptographic checksum is designed to expose deliberate corruption of messages by an adversary. The value it outputs is called a *message digest* and, like an ordinary checksum, is appended to the message. All the message digests produced by a given hash have the same number of bits regardless of the length of the original message. Since the space of possible input messages is larger than the space of possible message digests, there will be different input messages that produce the same message digest, like collisions in a hash table.

An authenticator can be created by encrypting the message digest. The receiver computes a digest of the plaintext part of the message and compares that to the decrypted message digest. If they are equal, then the receiver would conclude that the message is indeed from its alleged sender (since it would have to have been encrypted with the right key) and has not been tampered with. No adversary could get away with sending a bogus message with a matching bogus digest because she would not have the key to encrypt the bogus digest correctly. An adversary could, however, obtain the plaintext original message and its encrypted digest by eavesdropping. The adversary could then (since the hash function is public knowledge) compute the digest of the original message and generate alternative messages looking for one with the same message digest. If she finds one, she could undetectably send the new message with the old authenticator. Therefore, security requires that the hash function have the *one-way* property: It must be computationally infeasible for an adversary to find any plaintext message that has the same digest as the original.

For a hash function to meet this requirement, its outputs must be fairly randomly distributed. For example, if digests are 128 bits long and randomly distributed, then you would need to try 2^{127} messages, on average, before finding a second message whose digest matches that of a given message. If the outputs are not randomly distributed—that is, if some outputs are much more likely than others—then for some messages you could find another message with the same digest much more easily than this, which would reduce the security of the algorithm. If you were instead just trying to find any *collision*—any two messages that produce the same digest—then you would need to compute the digests of only 2^{64} messages, on average. This surprising fact is the basis of the "birthday attack"—see the exercises for more details.

There have been several common cryptographic hash algorithms over the years, including Message Digest 5 (MD5) and the Secure Hash Algorithm (SHA) family. Weaknesses of MD5 and earlier versions of SHA have been known for some time, which led NIST to recommend using SHA-3 in 2015. generating an encrypted message digest, the digest encryption could use either a secret-key cipher or a public-key cipher. If a public-key cipher is used, the digest would be encrypted using the sender's private key (the one we normally think of as being used for decryption), and the receiver—or anyone else—could decrypt the digest using the sender's public key.

A digest encrypted with a public key algorithm but using the private key is called a *digital signature* because

it provides nonrepudiation like a written signature. The receiver of a message with a digital signature can prove to any third party that the sender really sent that message, because the third party can use the sender's public key to check for herself. (secret-key encryption of a digest does not have this property because only the two participants know the key; furthermore, since both participants know the key, the alleged receiver could have created the message herself.) Any public-key cipher can be used for digital signatures. *Digital Signature Standard* (DSS) is a digital signature format that has been standardized by NIST. DSS signatures may use any one of three public-key ciphers, one based on RSA, another on ElGamal, and a third called the *Elliptic Curve Digital Signature Algorithm*.

Another kind of authenticator is similar, but instead of encrypting a hash it uses a hash-like function that takes a secret value (known only to the sender and the receiver) as a parameter, as illustrated in Figure 8.5. Such a function outputs an authenticator called a *message authentication code* (MAC). The sender appends the MAC to her plaintext message. The receiver recomputes the MAC using the plaintext and the secret value and compares that recomputed MAC to the received MAC.

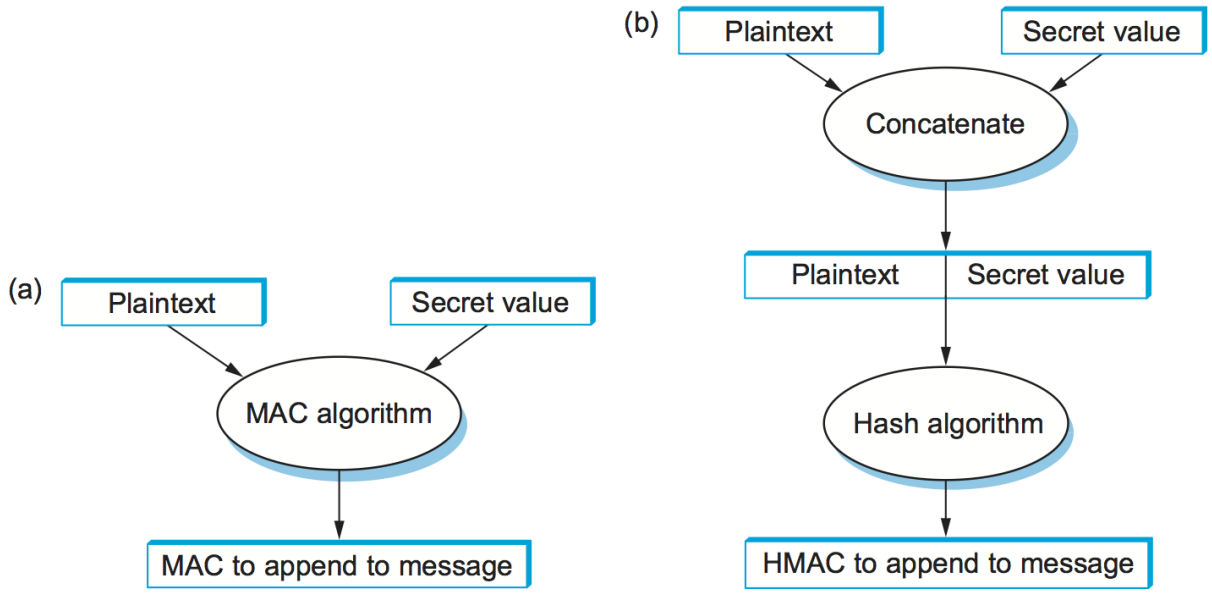


Figure 8.5.: Computing a MAC (a) versus computing an HMAC (b).

A common variation on MACs is to apply a cryptographic hash (such as MD5 or SHA-1) to the concatenation of the plaintext message and the secret value, as illustrated in Figure 8.5. The resulting digest is called a *hashed message authentication code* (HMAC) since it is essentially a MAC. The HMAC, but not the secret value, is appended to the plaintext Only a receiver who knows the secret value can compute the correct HMAC to compare with the received HMAC. If it weren't for the one-way property of the hash, an adversary might be able to find the input that generated the HMAC and compare it to the plaintext message to determine the secret value.

Up to this point, we have been assuming that the message wasn't confidential, so the original message could be transmitted as plaintext. To add confidentiality to a message with an authenticator, it suffices to encrypt the concatenation of the entire message including its authenticator—the MAC, HMAC, or encrypted digest. Remember that, in practice, confidentiality is implemented using secret-key ciphers because they are so much faster than public-key ciphers. Furthermore, it costs little to include the authenticator in the encryption, and it increases security. A common simplification is to encrypt the message with its (raw) digest, such that the digest is only encrypted once; in this case, the entire ciphertext message is considered

to be an authenticator.

Although authenticators may seem to solve the authentication problem, we will see in a later section that they are only the foundation of a solution. First, however, we address the issue of how participants obtain keys in the first place.

8.3 Key Predistribution

To use ciphers and authenticators, the communicating participants need to know what keys to use. In the case of a secret-key cipher, how does a pair of participants obtain the key they share? In the case of a public-key cipher, how do participants know what public key belongs to a certain participant? The answer differs depending on whether the keys are short-lived *session keys* or longer-lived *predistributed keys*.

A session key is a key used to secure a single, relatively short episode of communication: a session. Each distinct session between a pair of participants uses a new session key, which is always a secret key for speed. The participants determine what session key to use by means of a protocol—a session key establishment protocol. A session key establishment protocol needs its own security (so that, for example, an adversary cannot learn the new session key); that security is based on the longer-lived predistributed keys.

There are two primary motivations for this division of labor between session keys and predistributed keys:

- Limiting the amount of time a key is used results in less time for computationally intensive attacks, less ciphertext for cryptanalysis, and less information exposed should the key be broken.
- Public key ciphers are generally superior for authentication and session key establishment but too slow to use for encrypting entire messages for confidentiality.

This section explains how predistributed keys are distributed, and the next section will explain how session keys are then established. We henceforth use "Alice" and "Bob" to designate participants, as is common in the cryptography literature. Bear in mind that although we tend to refer to participants in anthropomorphic terms, we are more frequently concerned with the communication between software or hardware entities such as clients and servers that often have no direct relationship with any particular person.

8.3.1 Predistribution of Public Keys

The algorithms to generate a matched pair of public and private keys are publicly known, and software that does it is widely available. So, if Alice wanted to use a public-key cipher, she could generate her own pair of public and private keys, keep the private key hidden, and publicize the public key. But, how can she publicize her public key—assert that it belongs to her—in such a way that other participants can be sure it really belongs to her? Not via email or Web, because an adversary could forge an equally plausible claim that key x belongs to Alice when x really belongs to the adversary.

A complete scheme for certifying bindings between public keys and identities—what key belongs to whom—is called a *Public Key Infrastructure* (PKI). A PKI starts with the ability to verify identities and bind them to keys out of band. By "out of band," we mean something outside the network and the computers that comprise it, such as in the following If Alice and Bob are individuals who know each other, then they could get together in the same room and Alice could give her public key to Bob directly, perhaps on a

business card. If Bob is an organization, Alice the individual could present conventional identification, perhaps involving a photograph or fingerprints. If Alice and Bob are computers owned by the same company, then a system administrator could configure Bob with Alice's public key.

Establishing keys out of band doesn't sound like it would scale well, but it suffices to bootstrap a PKI. Bob's knowledge that Alice's key is x can be widely, scalably disseminated using a combination of digital signatures and a concept of trust. For example, suppose that you have received Bob's public key out of band and that you know enough about Bob to trust him on matters of keys and identities. Then Bob could send you a message asserting that Alice's key is x and—since you already know Bob's public key—you could authenticate the message as having come from Bob. (Remember that to digitally sign the statement Bob would append a cryptographic hash of it that has been encrypted using his private key.) Since you trust Bob to tell the truth, you would now know that Alice's key is x, even though you had never met her or exchanged a single message with her. Using digital signatures, Bob wouldn't even have to send you a message; he could simply create and publish a digitally signed statement that Alice's key is x. Such a digitally signed statement of a public key binding is called a *public key certificate*, or simply a certificate. Bob could send Alice a copy of the certificate, or post it on a website. If and when someone needs to verify Alice's public key, they could do so by getting a copy of the certificate, perhaps directly from Alice—as long as they trust Bob and know his public key. You can see how starting from a very small number of keys (in this case, just Bob's) you could build up a large set of trusted keys over time. Bob in this case is playing the role often referred to as a *certification authority* (CA), and much of today's Internet security depends on CAs. VeriSign is one well-known commercial CA. We return to this topic below.

One of the major standards for certificates is known as X.509. This standard leaves a lot of details open, but specifies a basic structure. A certificate clearly must include:

- The identity of the entity being certified
- The public key of the entity being certified
- The identity of the signer
- The digital signature
- A digital signature algorithm identifier (which cryptographic hash and which cipher)

An optional component is an expiration time for the certificate. We will see a particular use of this feature below.

Since a certificate creates a binding between an identity and a public key, we should look more closely at what we mean by "identity." For example, a certificate that says, "This public key belongs to John Smith," may not be terribly useful if you can't tell which of the thousands of John Smiths is being identified. Thus, certificates must use a well-defined name space for the identities being certified; for example, certificates are often issued for email addresses and DNS domains.

There are different ways a PKI could formalize the notion of trust. We discuss the two main approaches.

Certification Authorities

In this model of trust, trust is binary; you either trust someone completely or not at all. Together with certificates, this allows the building of *chains of trust*. If X certifies that a certain public key belongs to Y, and then Y goes on to certify that another public key belongs to Z, then there exists a chain of certificates from X to Z, even though X and Z may have never met. If you know X's key—and you trust X and Y—then

you can believe the certificate that gives Z's key. In other words, all you need is a chain of certificates, all signed by entities you trust, as long as it leads back to an entity whose key you already know.

A *certification authority* or *certificate authority* (CA) is an entity claimed (by someone) to be trustworthy for verifying identities and issuing public key certificates. There are commercial CAs, governmental CAs, and even free CAs. To use a CA, you must know its own key. You can learn that CA's key, however, if you can obtain a chain of CA-signed certificates that starts with a CA whose key you already know. Then you can believe any certificate signed by that new CA.

A common way to build such chains is to arrange them in a tree-structured hierarchy, as shown in Figure 8.6. If everyone has the public key of the root CA, then any participant can provide a chain of certificates to another participant and know that it will be sufficient to build a chain of trust for that participant.

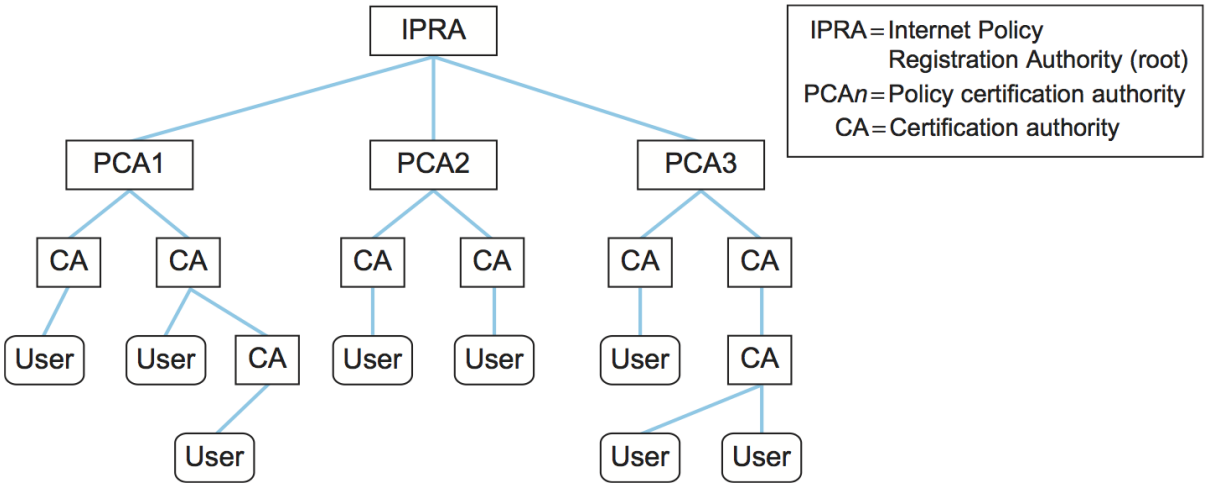


Figure 8.6.: Tree-structured certification authority hierarchy.

There are some significant issues with building chains of trust. Most importantly, even if you are certain that you have the public key of the root CA, you need to be sure that every CA from the root on down is doing its job properly. If just one CA in the chain is willing to issue certificates to entities without verifying their identities, then what looks like a valid chain of certificates becomes meaningless. For example, a root CA might issue a certificate to a second-tier CA and thoroughly verify that the name on the certificate matches the business name of the CA, but that second-tier CA might be willing to sell certificates to anyone who asks, without verifying their identity. This problem gets worse the longer the chain of trust. X.509 certificates provide the option of restricting the set of entities that the subject of a certificate is, in turn, trusted to certify.

There can be more than one root to a certification tree, and this is common in securing Web transactions today, for example. Web browsers such as Firefox and Internet Explorer come pre-equipped with certificates for a set of CAs; in effect, the browser's producer has decided these CAs and their keys can be trusted. A user can also add CAs to those that their browser recognizes as trusted. These certificates are accepted by Secure Socket Layer (SSL)/Transport Layer Security (TLS), the protocol most often used to secure Web transactions, which we discuss in a later section. (If you are curious, you can poke around in the preferences settings for your browser and find the "view certificates" option to see how many CAs your browser is configured to trust.)

Web of Trust

An alternative model of trust is the *web of trust* exemplified by Pretty Good Privacy (PGP), which is further discussed in a later section. PGP is a security system for email, so email addresses are the identities to which keys are bound and by which certificates are signed. In keeping with PGP's roots as protection against government intrusion, there are no CAs. Instead, every individual decides whom they trust and how much they trust them—in this model, trust is a matter of degree. In addition, a public key certificate can include a confidence level indicating how confident the signer is of the key binding claimed in the certificate, so a given user may have to have several certificates attesting to the same key binding before he is willing to trust it.

For example, suppose you have a certificate for Bob provided by Alice; you can assign a moderate level of trust to that certificate. However, if you have additional certificates for Bob that were provided by C and D, each of whom is also moderately trustworthy, that might considerably increase your level of confidence that the public key you have for Bob is valid. In short, PGP recognizes that the problem of establishing trust is quite a personal matter and gives users the raw material to make their own decisions, rather than assuming that they are all willing to trust in a single hierarchical structure of CAs. To quote Phil Zimmerman, the developer of PGP, "PGP is for people who prefer to pack their own parachutes."

PGP has become quite popular in the networking community, and PGP key-signing parties are a regular feature of various networking events, such as IETF meetings. At these gatherings, an individual can

- Collect public keys from others whose identity he knows.
- Provide his public key to others.
- Get his public key signed by others, thus collecting certificates that will be persuasive to an increasingly large set of people.
- Sign the public key of other individuals, thus helping them build up their set of certificates that they can use to distribute their public keys.
- Collect certificates from other individuals whom he trusts enough to sign keys.

Thus, over time, a user will collect a set of certificates with varying degrees of trust.

Certificate Revocation

One issue that arises with certificates is how to revoke, or undo, a certificate. Why is this important? Suppose that you suspect that someone has discovered your private key. There may be any number of certificates in the universe that assert that you are the owner of the public key corresponding to that private key. The person who discovered your private key thus has everything he needs to impersonate you: valid certificates and your private key. To solve this problem, it would be nice to be able to revoke the certificates that bind your old, compromised key to your identity, so that the impersonator will no longer be able to persuade other people that he is you.

The basic solution to the problem is simple enough. Each CA can issue a *certificate revocation list* (CRL), which is a digitally signed list of certificates that have been revoked. The CRL is periodically updated and made publicly available. Because it is digitally signed, it can just be posted on a website. Now, when Alice receives a certificate for Bob that she wants to verify, she will first consult the latest CRL issued by the CA. As long as the certificate has not been revoked, it is valid. Note that, if all certificates have unlimited life

spans, the CRL would always be getting longer, since you could never take a certificate off the CRL for fear that some copy of the revoked certificate might be used. For this reason, it is common to attach an expiration date to a certificate when it is issued. Thus, we can limit the length of time that a revoked certificate needs to stay on a CRL. As soon as its original expiration date is passed, it can be removed from the CRL.

8.3.2 Predistribution of Secret Keys

If Alice wants to use a secret-key cipher to communicate with Bob, she can't just pick a key and send it to him because, without already having a key, they can't encrypt this key to keep it confidential and they can't authenticate each other. As with public keys, some predistribution scheme is needed. Predistribution is harder for secret keys than for public keys for two obvious reasons:

- While only one public key per entity is sufficient for authentication and confidentiality, there must be a secret key for each pair of entities who wish to communicate. If there are N entities, that means N(N-1)/2 keys.
- Unlike public keys, secret keys must be kept secret.

In summary, there are a lot more keys to distribute, and you can't use certificates that everyone can read.

The most common solution is to use a *Key Distribution Center* (KDC). A KDC is a trusted entity that shares a secret key with each other entity. This brings the number of keys down to a more manageable N-1, few enough to establish out of band for some applications. When Alice wishes to communicate with Bob, that communication does not travel via the KDC. Rather, the KDC participates in a protocol that authenticates Alice and Bob—using the keys that the KDC already shares with each of them—and generates a new session key for them to use. Then Alice and Bob communicate directly using their session key. Kerberos is a widely used system based on this approach. We describe Kerberos (which also provides authentication) in the next section. The following subsection describes a powerful alternative.

8.3.3 Diffie-Hellman Key Exchange

Another approach to establishing a shared secret key is to use the Diffie-Hellman key exchange protocol, which works without using any predistributed keys. The messages exchanged between Alice and Bob can be read by anyone able to eavesdrop, and yet the eavesdropper won't know the secret key that Alice and Bob end up with.

Diffie-Hellman doesn't authenticate the participants. Since it is rarely useful to communicate securely without being sure whom you're communicating with, Diffie-Hellman is usually augmented in some way to provide authentication. One of the main uses of Diffie-Hellman is in the Internet Key Exchange (IKE) protocol, a central part of the IP Security (IPsec) architecture.

The Diffie-Hellman protocol has two parameters, p and g, both of which are public and may be used by all the users in a particular system. Parameter p must be a prime number. The integers $\bmod p$ (short for modulo p) are 0 through $p-1$, since $x \bmod p$ is the remainder after x is divided by p, and form what mathematicians call a *group* under multiplication. Parameter g (usually called a generator) must be a *primitive root* of p: For every number n from 1 through $p-1$ there must be some value k such that $n = g^k \bmod p$. For example, if p were the prime number 5 (a real system would use a much larger number), then we might choose 2 to be the generator g since:

$$1 = 2^0 \bmod p$$

$$2 = 2^1 \bmod p$$

$$3 = 2^3 \bmod p$$

$$4 = 2^2 \bmod p$$

Suppose Alice and Bob want to agree on a shared secret key. Alice and Bob, and everyone else, already know the values of *p* and *g*. Alice generates a random private value *a* and Bob generates a random private value *b*. Both *a* and *b* are drawn from the set of integers $\{1, \ldots, p-1\}$. Alice and Bob derive their corresponding public values—the values they will send to each other unencrypted—as follows. Alice's public value is

$$g^a \bmod p$$

and Bob's public value is

$$g^b \bmod p$$

They then exchange their public values. Finally, Alice computes

$$g^{ab} \bmod p = (g^b \bmod p)^a \bmod p$$

and Bob computes

$$g^{ba} \bmod p = (g^a \bmod p)^b \bmod p.$$

Alice and Bob now have $g^{ab} \bmod p$ (which is equal to $g^{ba} \bmod p$) as their shared secret key.

Any eavesdropper would know *p, g*, and the two public values $g^a \bmod p$ and $g^b \bmod p$. If only the eavesdropper could determine *a* or *b*, she could easily compute the resulting key. Determining *a* or *b* from that information is, however, computationally infeasible for suitably large *p,a,* and *b*; it is known as the *discrete logarithm problem*.

For example, using *p = 5* and *g = 2* from above, suppose Alice picks the random number *a = 3* and Bob picks the random number *b = 4*. Then Alice sends Bob the public value

$$2^3 \bmod 5 = 3$$

and Bob sends Alice the public value

$$2^4 \bmod 5 = 1$$

Alice is then able to compute

$$g^{ab} \bmod p = (2^b \bmod 5)^3 \bmod 5 = (1)^3 \bmod 5 = 1$$

by substituting Bob's public value for $(2^b \bmod 5)$. Similarly, Bob is able to compute

$$g^{ba} \bmod p = (g^a \bmod 5)^4 \bmod 5 = (3)^4 \bmod 5 = 1.$$

by substituting Alice's public value for $(2^a \bmod 5)$. Both Alice and Bob now agree that the secret key is 1.

There is the problem of Diffie-Hellman's lack of authentication. One attack that can take advantage of this is the *man-in-the-middle attack*. Suppose Mallory is an adversary with the ability to intercept messages.

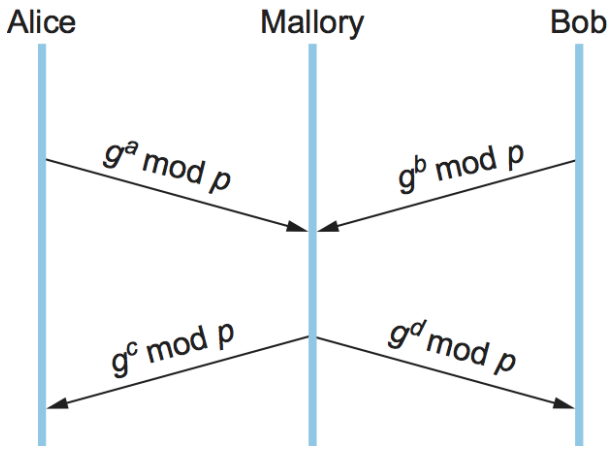


Figure 8.7.: A man-in-the-middle attack.

Mallory already knows p and g since they are public, and she generates random private values c and d to use with Alice and Bob, respectively. When Alice and Bob send their public values to each other, Mallory intercepts them and sends her own public values, as in Figure 8.7. The result is that Alice and Bob each end up unknowingly sharing a key with Mallory instead of each other.

A variant of Diffie-Hellman sometimes called *fixed Diffie-Hellman* supports authentication of one or both participants. It relies on certificates that are similar to public key certificates but instead certify the Diffie-Hellman public parameters of an entity. For example, such a certificate would state that Alice's Diffie-Hellman parameters are p, g, and $g^a \bmod p$ (note that the value of a would still be known only to Alice). Such a certificate would assure Bob that the other participant in Diffie-Hellman is Alice—or else the other participant won't be able to compute the secret key, because she won't know a. If both participants have certificates for their Diffie-Hellman parameters, they can authenticate each other. If just one has a certificate, then just that one can be authenticated. This is useful in some situations; for example, when one participant is a web server and the other is an arbitrary client, the client can authenticate the web server and establish a secret key for confidentiality before sending a credit card number to the web server.

8.4 Authentication Protocols

So far we described how to encrypt messages, build authenticators, predistribute the necessary keys. It might seem as if all we have to do to make a protocol secure is append an authenticator to every message and, if we want confidentiality, encrypt the message.

There are two main reasons why it's not that simple. First, there is the problem of a *replay attack*: an adversary retransmitting a copy of a message that was previously sent. If the message was an order you had placed on a website, for example, then the replayed message would appear to the website as though you had ordered more of the same. Even though it wasn't the original incarnation of the message, its authenticator would still be valid; after all, the message was created by you, and it wasn't modified. Clearly, we need a solution that ensures *originality*.

In a variation of this attack called a *suppress-replay attack*, an adversary might merely delay your message (by intercepting and later replaying it), so that it is received at a time when it is no longer appropriate. For example, an adversary could delay your order to buy stock from an auspicious time to a time when

you would not have wanted to buy. Although this message would in a sense be the original, it wouldn't be timely. So we also need to ensure *timeliness*. Originality and timeliness may be considered aspects of integrity. Ensuring them will in most cases require a nontrivial, back-and-forth protocol.

The second problem we have not yet solved is how to establish a session key. A session key is a secret-key cipher key generated on the fly and used for just one session. This too involves a nontrivial protocol.

What these two issues have in common is authentication. If a message is not original and timely, then from a practical standpoint we want to consider it as not being authentic, not being from whom it claims to be. And, obviously, when you are arranging to share a new session key with someone, you want to know you are sharing it with the right person. Usually, authentication protocols establish a session key at the same time, so that at the end of the protocol Alice and Bob have authenticated each other and they have a new secret key to use. Without a new session key, the protocol would just authenticate Alice and Bob at one point in time; a session key allows them to efficiently authenticate subsequent messages. Generally, session key establishment protocols perform authentication. A notable exception is Diffie-Hellman, as described below, so the terms *authentication protocol* and *session key establishment protocol* are almost synonymous.

There is a core set of techniques used to ensure originality and timeliness in authentication protocols. We describe those techniques before moving on to particular protocols.

8.4.1 Originality and Timeliness Techniques

We have seen that authenticators alone do not enable us to detect messages that are not original or timely. One approach is to include a timestamp in the message. Obviously the timestamp itself must be tamper-proof, so it must be covered by the authenticator. The primary drawback to timestamps is that they require distributed clock synchronization. Since our system would then depend on synchronization, the clock synchronization itself would need to be defended against security threats, in addition to the usual challenges of clock synchronization. Another issue is that distributed clocks are synchronized to only a certain degree—a certain margin of error. Thus, the timing integrity provided by timestamps is only as good as the degree of synchronization.

Another approach is to include a *nonce*—a random number used only once—in the message. Participants can then detect replay attacks by checking whether a nonce has been used previously. Unfortunately, this requires keeping track of past nonces, of which a great many could accumulate. One solution is to combine the use of timestamps and nonces, so that nonces are required to be unique only within a certain span of time. That makes ensuring uniqueness of nonces manageable while requiring only loose synchronization of clocks.

Another solution to the shortcomings of timestamps and nonces is to use one or both of them in a *challenge-response* protocol. Suppose we use a timestamp. In a challenge-response protocol, Alice sends Bob a timestamp, challenging Bob to encrypt it in a response message (if they share a secret key) or digitally sign it in a response message (if Bob has a public key, as in Figure 8.8). The encrypted timestamp is like an authenticator that additionally proves timeliness. Alice can easily check the timeliness of the timestamp in a response from Bob since that timestamp comes from Alice's own clock—no distributed clock synchronization needed. Suppose instead that the protocol uses nonces. Then Alice need only keep track of those nonces for which responses are currently outstanding and haven't been outstanding too long; any purported response with an unrecognized nonce must be bogus.

The beauty of challenge-response, which might otherwise seem excessively complex, is that it combines timeliness and authentication; after all, only Bob (and possibly Alice, if it's a secret-key cipher) knows the

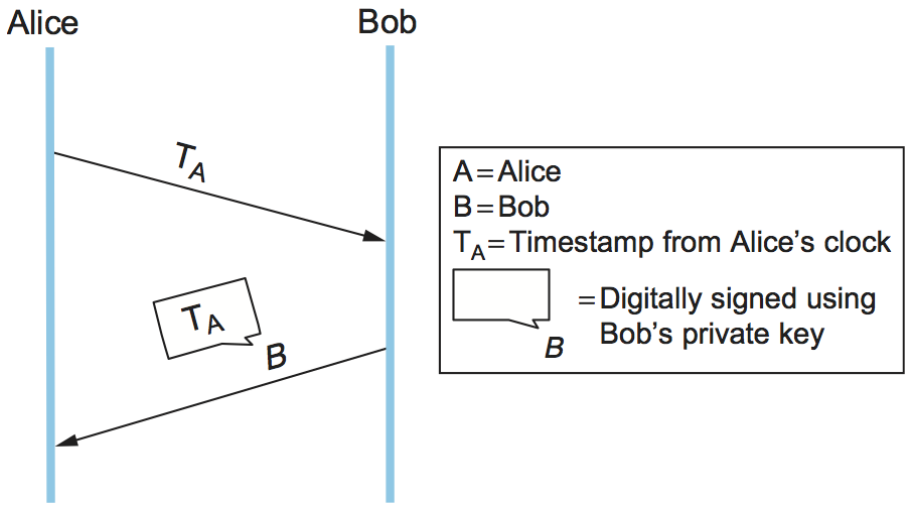


Figure 8.8.: A challenge-response protocol.

key necessary to encrypt the never before seen timestamp or nonce. Timestamps or nonces are used in most of the authentication protocols that follow.

8.4.2 Public-Key Authentication Protocols

In the following discussion, we assume that Alice and Bob's public keys have been predistributed to each other via some means such as a PKI. We mean this to include the case where Alice includes her certificate in her first message to Bob, and the case where Bob searches for a certificate about Alice when he receives her first message.

This first protocol (Figure 8.9) relies on Alice and Bob's clocks being synchronized. Alice sends Bob a message with a timestamp and her identity in plaintext plus her digital signature. Bob uses the digital signature to authenticate the message and the timestamp to verify its freshness. Bob sends back a message with a timestamp and his identity in plaintext, as well as a new session key encrypted (for confidentiality) using Alice's public key, all digitally signed. Alice can verify the authenticity and freshness of the message, so she knows she can trust the new session key. To deal with imperfect clock synchronization, the timestamps could be augmented with nonces.

The second protocol (Figure 8.10) is similar but does not rely on clock synchronization. In this protocol, Alice again sends Bob a digitally signed message with a timestamp and her identity. Because their clocks aren't synchronized, Bob cannot be sure that the message is fresh. Bob sends back a digitally signed message with Alice's original timestamp, his own new timestamp, and his identity. Alice can verify the freshness of Bob's reply by comparing her current time against the timestamp that originated with her. She then sends Bob a digitally signed message with his original timestamp and a new session key encrypted using Bob's public key. Bob can verify the freshness of the message because the timestamp came from his clock, so he knows he can trust the new session key. The timestamps essentially serve as convenient nonces, and indeed this protocol could use nonces instead.

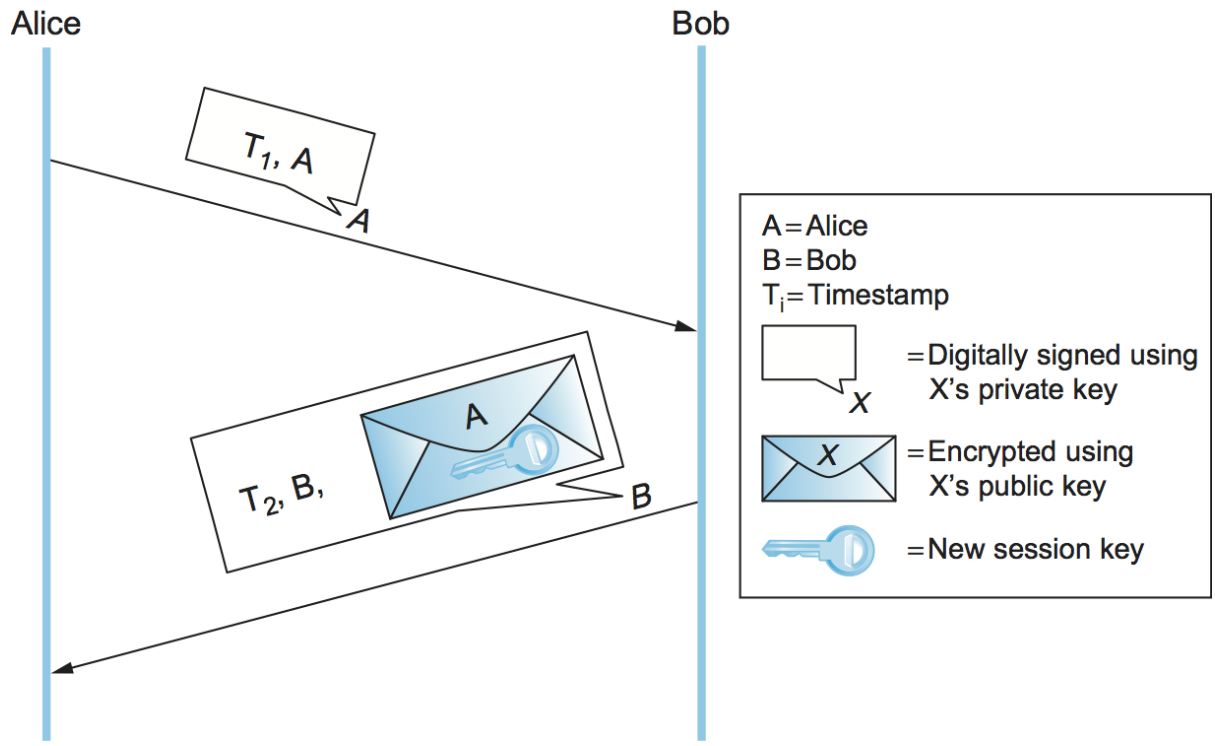


Figure 8.9.: A public-key authentication protocol that depends on synchronization.

8.4.3 Secret-Key Authentication Protocols

Only in fairly small systems is it practical to predistribute secret keys to every pair of entities. We focus here on larger systems, where each entity would have its own *master key* shared only with a Key Distribution Center (KDC). In this case, secret-key-based authentication protocols involve three parties: Alice, Bob, and a KDC. The end product of the authentication protocol is a session key shared between Alice and Bob that they will use to communicate directly, without involving the KDC.

The Needham-Schroeder authentication protocol is illustrated in Figure 8.11. Note that the KDC doesn't actually authenticate Alice's initial message and doesn't communicate with Bob at all. Instead, the KDC uses its knowledge of Alice's and Bob's master keys to construct a reply that would be useless to anyone other than Alice (because only Alice can decrypt it) and contains the necessary ingredients for Alice and Bob to perform the rest of the authentication protocol themselves.

The nonce in the first two messages is to assure Alice that the KDC's reply is fresh. The second and third messages include the new session key and Alice's identifier, encrypted together using Bob's master key. It is a sort of secret-key version of a public-key certificate; it is in effect a signed statement by the KDC (because the KDC is the only entity besides Bob who knows Bob's master key) that the enclosed session key is owned by Alice and Bob. Although the nonce in the last two messages is intended to assure Bob that the third message was fresh, there is a flaw in this reasoning.

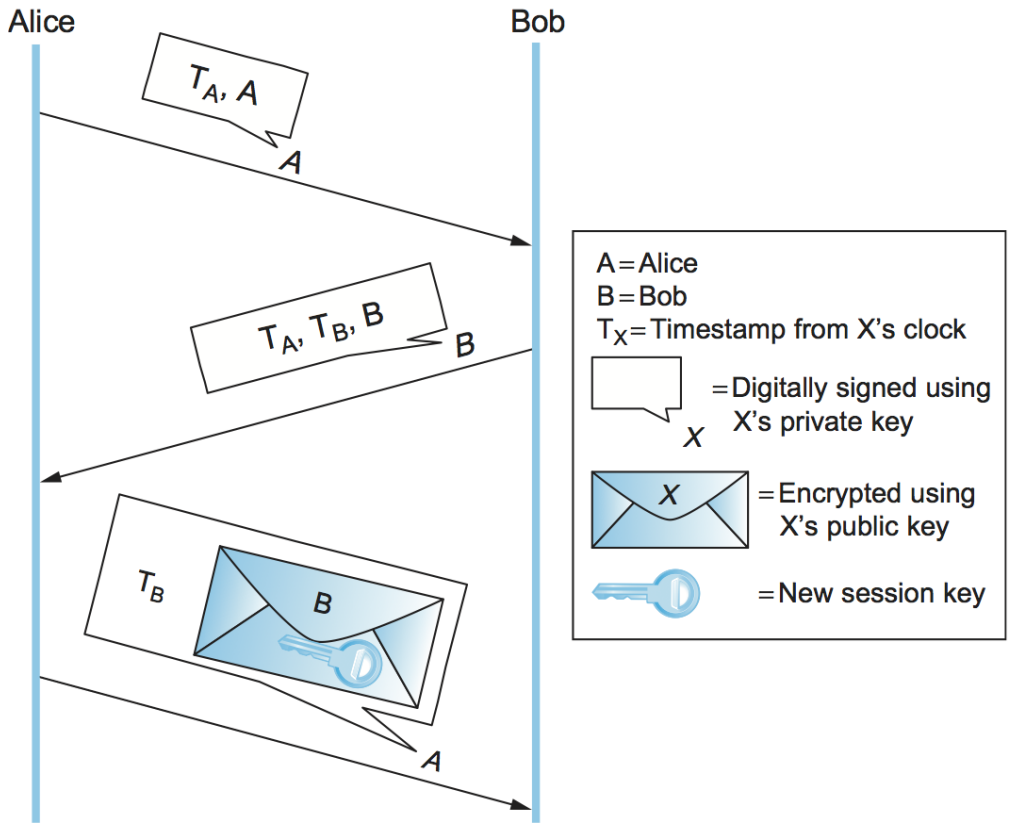


Figure 8.10.: A public-key authentication protocol that does not depend on synchronization. Alice checks her own timestamp against her own clock, and likewise for Bob.

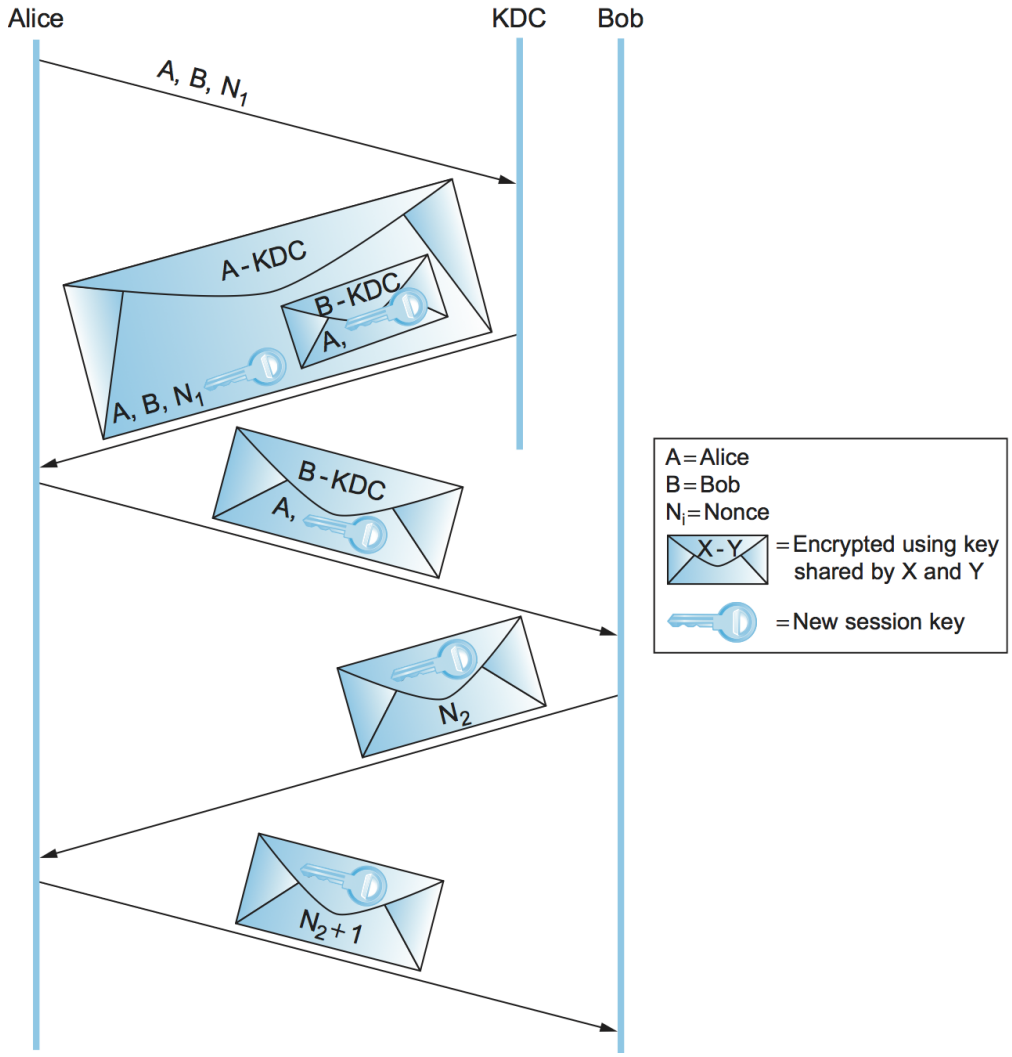


Figure 8.11.: The Needham-Schroeder authentication protocol.

Kerberos

Kerberos is an authentication system based on the Needham-Schroeder protocol and specialized for client/server environments. Originally developed at MIT, it has been standardized by the IETF and is available as both open source and commercial products. We will focus here on some of Kerberos's interesting innovations.

Kerberos clients are generally human users, and users authenticate themselves using passwords. Alice's master key, shared with the KDC, is derived from her password—if you know the password, you can compute the key. Kerberos assumes anyone can physically access any client machine; therefore, it is important to minimize the exposure of Alice's password or master key not just in the network but also on any machine where she logs in. Kerberos takes advantage of Needham-Schroeder to accomplish this. In Needham-Schroeder, the only time Alice needs to use her password is when decrypting the reply from the KDC. Kerberos client-side software waits until the KDC's reply arrives, prompts Alice to enter her password, computes the master key and decrypts the KDC's reply, and then erases all information about the password and master key to minimize its exposure. Also note that the only sign a user sees of Kerberos is when the user is prompted for a password.

In Needham-Schroeder, the KDC's reply to Alice plays two roles: It gives her the means to prove her identity (only Alice can decrypt the reply), and it gives her a sort of secret-key certificate or "ticket" to present to Bob—the session key and Alice's identifier, encrypted with Bob's master key. In Kerberos, those two functions—and the KDC itself, in effect—are split up (Figure 8.12). A trusted server called an Authentication Server (AS) plays the first KDC role of providing Alice with something she can use to prove her identity—not to Bob this time, but to a second trusted server called a Ticket Granting Server (TGS). The TGS plays the second KDC role, replying to Alice with a ticket she can present to Bob. The attraction of this scheme is that if Alice needs to communicate with several servers, not just Bob, then she can get tickets for each of them from the TGS without going back to the AS.

In the client/server application domain for which Kerberos is intended, it is reasonable to assume a degree of clock synchronization. This allows Kerberos to use timestamps and lifespans instead of Needham-Shroeder's nonces, and thereby eliminate the Needham-Schroeder security weakness. Kerberos supports a choice of hash functions and secret-key ciphers, allowing it to keep pace with the state-of-the-art in cryptographic algorithms.

8.5 Example Systems

We have now seen many of the components required to provide one or two aspects of security. These components include cryptographic algorithms, key predistribution mechanisms, and authentication protocols. In this section, we examine some complete systems that use these components.

These systems can be roughly categorized by the protocol layer at which they operate. Systems that operate at the application layer include Pretty Good Privacy (PGP), which provides electronic mail security, and Secure Shell (SSH), a secure remote login facility. At the transport layer, there is the IETF's Transport Layer Security (TLS) standard and the older protocol from which it derives, Secure Socket Layer (SSL). The IPsec (IP Security) protocols, as their name implies, operate at the IP (network) layer. 802.11i provides security at the link layer of wireless networks. This section describes the salient features of each of these approaches.

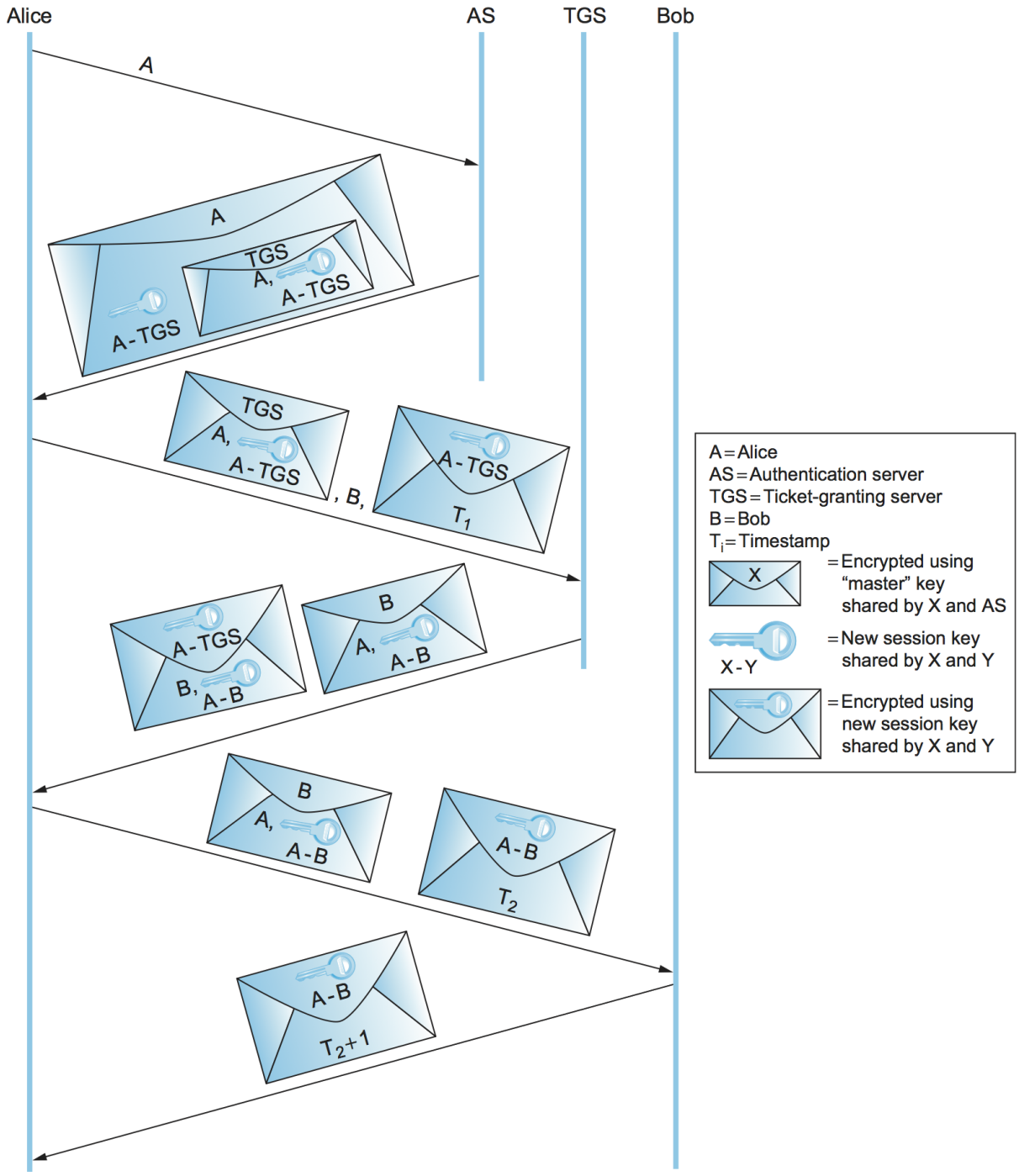


Figure 8.12.: Kerberos authentication.

You might reasonably wonder why security has to be provided at so many different layers. One reason is that different threats require different defensive measures, and this often translates into securing a different protocol layer. For example, if your main concern is with a person in the building next door snooping on your traffic as it flows between your laptop and your 802.11 access point, then you probably want security at the link layer. However, if you want to be really sure you are connected to your bank's website and preventing all the data that you send to the bank from being read by curious employees of some Internet service provider, then something that extends all the way from your machine to the bank's server—like the transport layer—may be the right place to secure the traffic. As is often the case, there is no one-size-fits-all solution.

The security systems described below have the ability to vary which cryptographic algorithms they use. The idea of making a security system algorithm independent is a very good one, because you never know when your favorite cryptographic algorithm might be proved to be insufficiently strong for your purposes. It would be nice if you could quickly change to a new algorithm without having to change the protocol specification or implementation. Note the analogy to being able to change keys without changing the algorithm; if one of your cryptographic algorithms turns out to be flawed, it would be great if your entire security architecture didn't need an immediate redesign.

8.5.1 Pretty Good Privacy (PGP)

Pretty Good Privacy (PGP) is a widely used approach to providing security for electronic mail. It provides authentication, confidentiality, data integrity, and nonrepudiation. Originally devised by Phil Zimmerman, it has evolved into an IETF standard known as OpenPGP. As we saw in a previous section, PGP is notable for using a "web of trust" model for distribution of keys rather than a tree-like hierarchy.

PGP's confidentiality and receiver authentication depend on the receiver of an email message having a public key that is known to the sender. To provide sender authentication and nonrepudiation, the sender must have a public key that is known by the receiver. These public keys are predistributed using certificates and a web-of-trust PKI. PGP supports RSA and DSS for public key certificates. These certificates may additionally specify which cryptographic algorithms are supported or preferred by the key's owner. The certificates provide bindings between email addresses and public keys.

Consider the following example of PGP being used to provide both sender authentication and confidentiality. Suppose Alice has a message to email to Bob. Alice's PGP application goes through the steps illustrated in Figure 8.13. First, the message is digitally signed by Alice; MD5, SHA-1, and the SHA-2 family are among the hashes that may be used in the digital signature. Her PGP application then generates a new session key for just this one message; AES and 3DES are among the supported secret-key ciphers. The digitally signed message is encrypted using the session key, then the session key itself is encrypted using Bob's public key and appended to the message. Alice's PGP application reminds her of the level of trust she had previously assigned to Bob's public key, based on the number of certificates she has for Bob and the trustworthiness of the individuals who signed the certificates. Finally, not for security but because email messages have to be sent in ASCII, a base64 encoding is applied to the message to convert it to an ASCII-compatible representation. Upon receiving the PGP message in an email, Bob's PGP application reverses this process step-by-step to obtain the original plaintext message and confirm Alice's digital signature—and reminds Bob of the level of trust he has in Alice's public key.

Email has particular characteristics that allow PGP to embed an adequate authentication protocol in this one-message data transmission protocol, avoiding the need for any prior message exchange (and sidestepping some of the complexities described in the previous section). Alice's digital signature suffices to authenticate

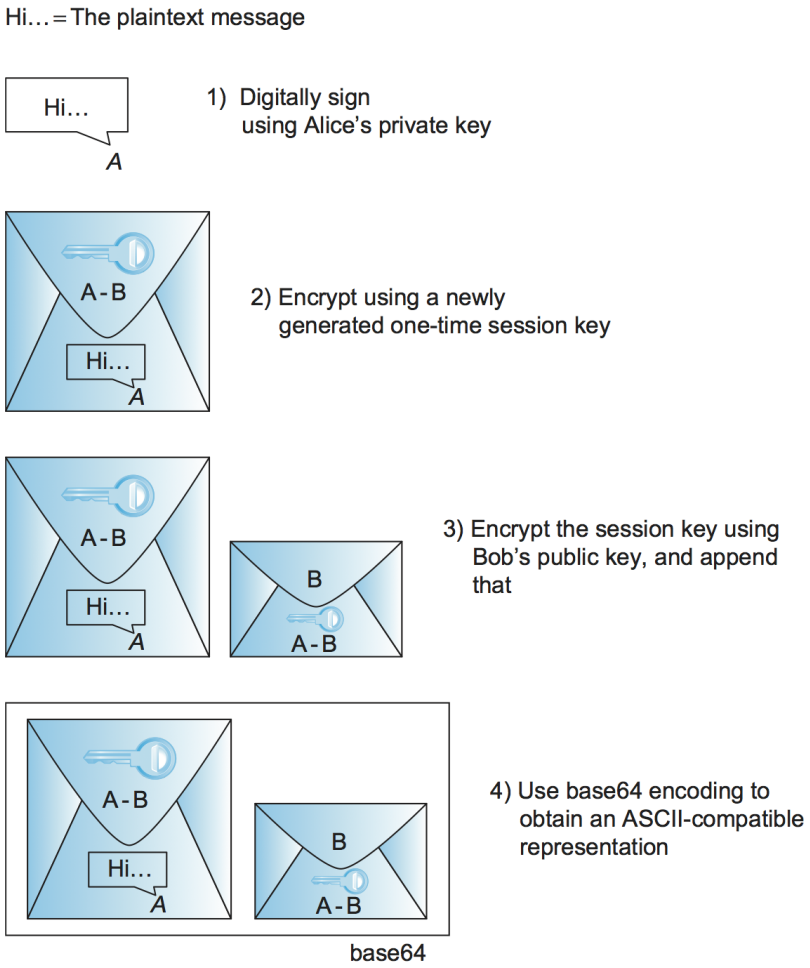


Figure 8.13.: PGP's steps to prepare a message for emailing from Alice to Bob.

her. Although there is no proof that the message is timely, legitimate email isn't guaranteed to be timely either. There is also no proof that the message is original, but Bob is an email user and probably a fault-tolerant human who can recover from duplicate emails (which, again, are not out of the question under normal operation anyway). Alice can be sure that only Bob could read the message because the session key was encrypted with his public key. Although this protocol doesn't prove to Alice that Bob is actually there and received the email, an authenticated email from Bob back to Alice could do this.

The preceding discussion gives a good example of why application-layer security mechanisms can be helpful. Only with a full knowledge of how the application works can you make the right choices about which attacks to defend against (like forged email) versus which to ignore (like delayed or replayed email).

8.5.2 Secure Shell (SSH)

The Secure Shell (SSH) protocol is used to provide a remote login service, replacing the less secure Telnet used in the early days of the Internet. (SSH can also be used to remotely execute commands and transfer files, but we will focus first on how SSH supports remote login.) SSH is most often used to provide strong client/server authentication/message integrity—where the SSH client runs on the user's desktop machine and the SSH server runs on some remote machine that the user wants to log into—but it also supports confidentiality. Telnet provides none of these capabilities. Note that "SSH" is often used to refer to both the SSH protocol and applications that use it; you need to figure out which from the context.

To better appreciate the importance of SSH on today's Internet, consider a couple of the scenarios where it is used. Telecommuters, for example, often subscribe to ISPs that offer high-speed fiber-to-the-home, and they use these ISPs (plus some chain of other ISPs) to reach machines operated by their employer. This means that when a telecommuter logs into a machine inside his employer's data center, both the passwords and all the data sent or received potentially passes through any number of untrusted networks. SSH provides a way to encrypt the data sent over these connections and to improve the strength of the authentication mechanism used to log in. (A similar situation occurs when said employee connects to work using the public Wi-Fi at Starbucks.) Another usage of SSH is remote login to a router, perhaps to change its configuration or read its log files; clearly, a network administrator wants to be sure that he can log into a router securely and that unauthorized parties can neither log in nor intercept the commands sent to the router or output sent back to the administrator.

The latest version of SSH, version 2, consists of three protocols:

- SSH-TRANS, a transport layer protocol
- SSH-AUTH, an authentication protocol
- SSH-CONN, a connection protocol

We focus on the first two, which are involved in remote login. We briefly discuss the purpose of SSH-CONN at the end of the section.

SSH-TRANS provides an encrypted channel between the client and server machines. It runs on top of a TCP connection. Any time a user uses an SSH application to log into a remote machine, the first step is to set up an SSH-TRANS channel between those two machines. The two machines establish this secure channel by first having the client authenticate the server using RSA. Once authenticated, the client and server establish a session key that they will use to encrypt any data sent over the channel. This high-level description skims over several details, including the fact that the SSH-TRANS protocol includes a negotiation of the

encryption algorithm the two sides are going to use. For example, AES is commonly selected. Also, SSH-TRANS includes a message integrity check of all data exchanged over the channel.

The one issue we can't skim over is how the client came to possess the server's public key that it needs to authenticate the server. Strange as it may sound, the server tells the client its public key at connection time. The first time a client connects to a particular server, the SSH application warns the user that it has never talked to this machine before and asks if the user wants to continue. Although it is a risky thing to do, because SSH is effectively not able to authenticate the server, users often say "yes" to this question. The SSH application then remembers the server's public key, and the next time the user connects to that same machine it compares this saved key with the one the server responds with. If they are the same, SSH authenticates the server. If they are different, however, the SSH application again warns the user that something is amiss, and the user is then given an opportunity to abort the connection. Alternatively, the prudent user can learn the server's public key through some out-of-band mechanism, save it on the client machine, and thus never take the "first time" risk.

Once the SSH-TRANS channel exists, the next step is for the user to actually log into the machine, or more specifically, authenticate himself or herself to the server. SSH allows three different mechanisms for doing this. First, since the two machines are communicating over a secure channel, it is OK for the user to simply send his or her password to the server. This is not a safe thing to do when using Telnet since the password would be sent in the clear, but in the case of SSH the password is encrypted in the SSH-TRANS channel. The second mechanism uses public-key encryption. This requires that the user has already placed his or her public key on the server. The third mechanism, called *host-based authentication*, basically says that any user claiming to be so-and-so from a certain set of trusted hosts is automatically believed to be that same user on the server. Host-based authentication requires that the client *host* authenticate itself to the server when they first connect; standard SSH-TRANS only authenticates the server by default.

The main thing you should take away from this discussion is that SSH is a fairly straightforward application of the protocols and algorithms we have seen throughout this chapter. However, what sometimes makes SSH a challenge to understand is all the keys a user has to create and manage, where the exact interface is operating system dependent. For example, the OpenSSH package that runs on most Unix machines supports a command that can be used to create public/private key pairs. These keys are then stored in various files in directory in the user's home directory. For example, file `~/.ssh/known_hosts` records the keys for all the hosts the user has logged into, file `~/.ssh/authorized_keys` contains the public keys needed to authenticate the user when he or she logs into this machine (i.e., they are used on the server side), and file contains the private keys needed to authenticate the user on remote machines (i.e., they are used on the client side).

Finally, SSH has proven so useful as a system for securing remote login, it has been extended to also support other applications, such as sending and receiving email. The idea is to run these applications over a secure "SSH tunnel." This capability is called *port forwarding*, and it uses the SSH-CONN protocol. The idea is illustrated in Figure 8.14, where we see a client on host A indirectly communicating with a server on host B by forwarding its traffic through an SSH connection. The mechanism is called *port forwarding* because when messages arrive at the well-known SSH port on the server, SSH first decrypts the contents and then "forwards" the data to the actual port at which the server is listening. This is just another sort of tunnel, which in this case happens to provide confidentiality and authentication. It's possible to provide a form of virtual private network (VPN) using SSH tunnels in this way.

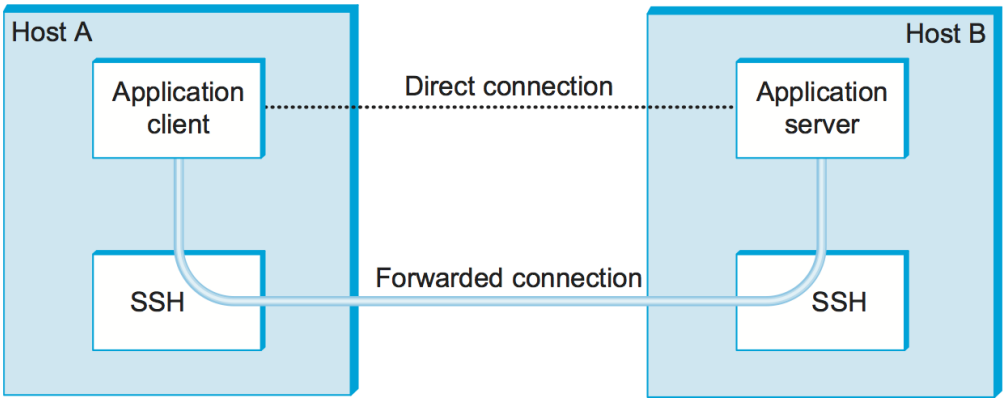


Figure 8.14.: Using SSH port forwarding to secure other TCP-based applications.

8.5.3 Transport Layer Security (TLS, SSL, HTTPS)

To understand the design goals and requirements for the Transport Layer Security (TLS) standard and the Secure Socket Layer (SSL) on which TLS is based, it is helpful to consider one of the main problems that they are intended to solve. As the World Wide Web became popular and commercial enterprises began to take an interest in it, it became clear that some level of security would be necessary for transactions on the Web. The canonical example of this is making purchases by credit card. There are several issues of concern when sending your credit card information to a computer on the Web. First, you might worry that the information would be intercepted in transit and subsequently used to make unauthorized purchases. You might also worry about the details of a transaction being modified, such as changing the purchase amount. And you would certainly like to know that the computer to which you are sending your credit card information is in fact one belonging to the vendor in question and not some other party. Thus, we immediately see a need for confidentiality, integrity, and authentication in Web transactions. The first widely used solution to this problem was SSL, originally developed by Netscape and subsequently the basis for the IETF's TLS standard.

The designers of SSL and TLS recognized that these problems were not specific to Web transactions (i.e., those using HTTP) and instead built a general-purpose protocol that sits between an application protocol such as HTTP and a transport protocol such as TCP. The reason for calling this "transport layer security" is that, from the application's perspective, this protocol layer looks just like a normal transport protocol except for the fact that it is secure. That is, the sender can open connections and deliver bytes for transmission, and the secure transport layer will get them to the receiver with the necessary confidentiality, integrity, and authentication. By running the secure transport layer on top of TCP, all of the normal features of TCP (reliability, flow control, congestion control, etc.) are also provided to the application. This arrangement of protocol layers is depicted in Figure 8.15.

When HTTP is used in this way, it is known as HTTPS (Secure HTTP). In fact, HTTP itself is unchanged. It simply delivers data to and accepts data from the SSL/TLS layer rather than TCP. For convenience, a default TCP port has been assigned to HTTPS (443). That is, if you try to connect to a server on TCP port 443, you will likely find yourself talking to the SSL/TLS protocol, which will pass your data through to HTTP provided all goes well with authentication and decryption. Although standalone implementations of SSL/TLS are available, it is more common for an implementation to be bundled with applications that need it, primarily web browsers.

In the remainder of our discussion of transport layer security, we focus on TLS. Although SSL and TLS

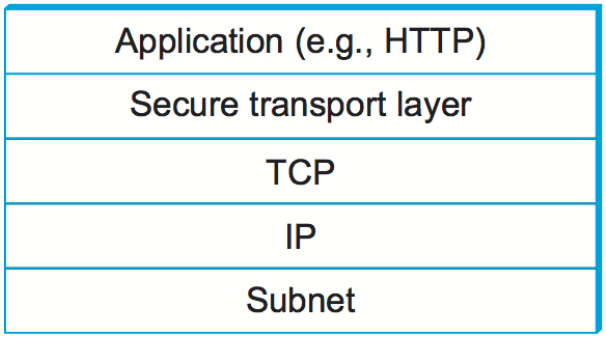


Figure 8.15.: Secure transport layer inserted between application and TCP layers.

are unfortunately not interoperable, they differ in only minor ways, so nearly all of this description of TLS applies to SSL.

Handshake Protocol

A pair of TLS participants negotiate at runtime which cryptography to use. The participants negotiate a choice of:

- Data integrity hash (MD5, SHA-1, etc.), used to implement HMACs
- secret-key cipher for confidentiality (among the possibilities are DES, 3DES, and AES)
- Session key establishment approach (among the possibilities are Diffie-Hellman, and public-key authentication protocols using DSS)

Interestingly, the participants may also negotiate the use of a compression algorithm, not because this offers any security benefits, but because it's easy to do when you're negotiating all this other stuff and you've already decided to do some expensive per-byte operations on the data.

In TLS, the confidentiality cipher uses two keys, one for each direction, and similarly two initialization vectors. The HMACs are likewise keyed with different keys for the two participants. Thus, regardless of the choice of cipher and hash, a TLS session requires effectively six keys. TLS derives all of them from a single shared *master secret*. The master secret is a 384-bit (48-byte) value that in turn is derived in part from the "session key" that results from TLS's session key establishment protocol.

The part of TLS that negotiates the choices and establishes the shared master secret is called the *handshake protocol*. (Actual data transfer is performed by TLS's *record protocol*.) The handshake protocol is at heart a session key establishment protocol, with a master secret instead of a session key. Since TLS supports a choice of approaches to session key establishment, these call for correspondingly different protocol variants. Furthermore, the handshake protocol supports a choice between mutual authentication of both participants, authentication of just one participant (this is the most common case, such as authenticating a website but not a user), or no authentication at all (anonymous Diffie-Hellman). Thus, the handshake protocol knits together several session key establishment protocols into a single protocol.

Figure 8.16 shows the handshake protocol at a high level. The client initially sends a list of the combinations of cryptographic algorithms that it supports, in decreasing order of preference. The server responds, giving the single combination of cryptographic algorithms it selected from those listed by the client. These mes-

sages also contain a *client nonce* and a *server nonce*, respectively, that will be incorporated in generating the master secret later.

At this point, the negotiation phase is complete. The server now sends additional messages based on the negotiated session key establishment protocol. That could involve sending a public-key certificate or a set of Diffie-Hellman parameters. If the server requires authentication of the client, it sends a separate message indicating that. The client then responds with its part of the negotiated key exchange protocol.

Now the client and server each have the information necessary to generate the master secret. The "session key" that they exchanged is not in fact a key, but instead what TLS calls a *pre-master secret*. The master secret is computed (using a published algorithm) from this pre-master secret, the client nonce, and the server nonce. Using the keys derived from the master secret, the client then sends a message that includes a hash of all the preceding handshake messages, to which the server responds with a similar message. This enables them to detect any discrepancies between the handshake messages they sent and received, such as would result, for example, if a man in the middle modified the initial unencrypted client message to weaken its choices of cryptographic algorithms.

Record Protocol

Within a session established by the handshake protocol, TLS's record protocol adds confidentiality and integrity to the underlying transport service. Messages handed down from the application layer are:

1. Fragmented or coalesced into blocks of a convenient size for the following steps
2. Optionally compressed
3. Integrity-protected using an HMAC
4. Encrypted using a secret-key cipher
5. Passed to the transport layer (normally TCP) for transmission

The record protocol uses an HMAC as an authenticator. The HMAC uses whichever hash algorithm (MD5, SHA-1, etc.) was negotiated by the participants. The client and server have different keys to use when computing HMACs, making them even harder to break. Furthermore, each record protocol message is assigned a sequence number, which is included when the HMAC is computed—even though the sequence number is never explicit in the message. This implicit sequence number prevents replays or reorderings of messages. This is needed because, although TCP can deliver sequential, unduplicated messages to the layer above it under normal assumptions, those assumptions do not include an adversary that can intercept TCP messages, modify messages, or send bogus ones. On the other hand, it is TCP's delivery guarantees that make it possible for TLS to rely on a legitimate TLS message having the next implicit sequence number in order.

Another interesting feature of the TLS protocol is the ability to resume a session. To understand the original motivation for this, it is helpful to understand how HTTP originally mades use of TCP connections. (The details of HTTP are presented in the next chapter.) Each HTTP operation, such as getting a page from a server, required a new TCP connection to be opened. Retrieving a single page with a number of embedded graphical objects might take many TCP connections. Opening a TCP connection requires a three-way handshake before data transmission can start. Once the TCP connection is ready to accept data, the client would then need to start the TLS handshake protocol, taking at least another two round-trip times (and

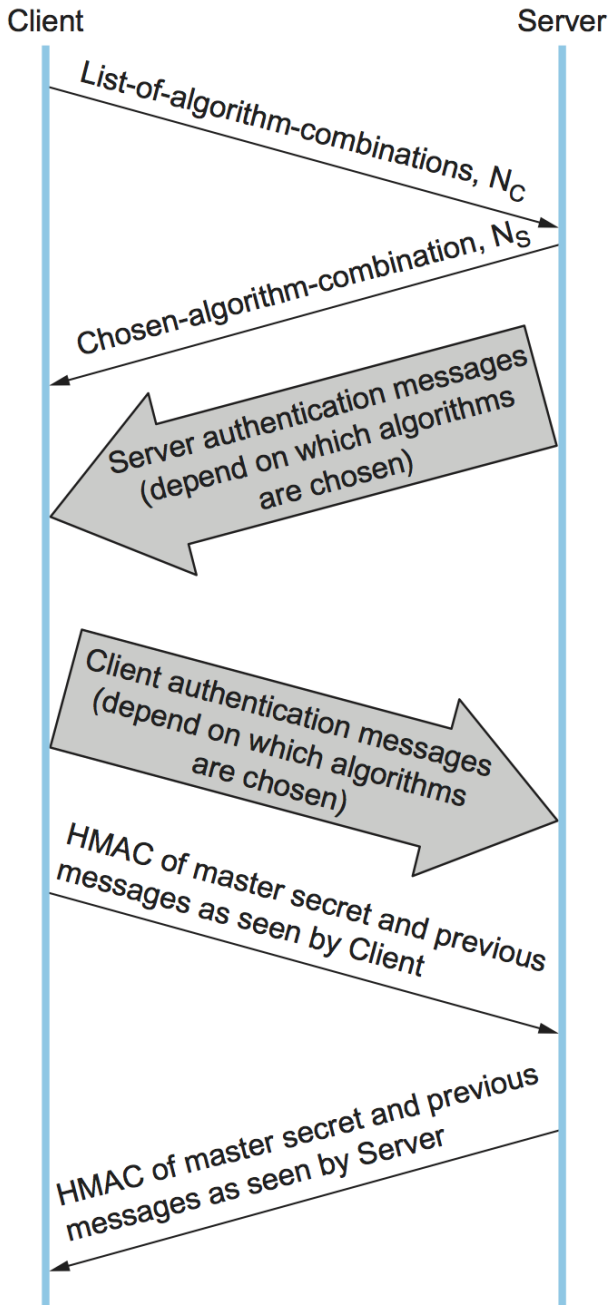


Figure 8.16.: Handshake protocol to establish TLS session.

consuming some amount of processing resources and network bandwidth) before actual application data could be sent. The resumption capability of TLS was designed to alleviate this problem.

The idea of session resumption is to optimize away the handshake in those cases where the client and the server have already established some shared state in the past. The client simply includes the session ID from a previously established session in its initial handshake message. If the server finds that it still has state for that session, and the resumption option was negotiated when that session was originally created, then the server can reply to the client with an indication of success, and data transmission can begin using the algorithms and parameters previously negotiated. If the session ID does not match any session state cached at the server, or if resumption was not allowed for the session, then the server will fall back to the normal handshake process.

The reason the preceeding discussion emphasized the *original* motivation is that having to do a TCP handshake for every embedded object in a web page led to so much overhead, independent of TLS, that HTTP was eventually optimized to support *persistent connections* (also discussed in the next chapter). Because optimizing HTTP mitigated the value of session resumption in TLS (plus the realization that reusing the same session IDs and master secret key in a series of resumed sessions is a security risk), TLS changed its approach to resumption in the latest version (1.3).

In TLS 1.3, the client sends an opaque, server-encrypted *session ticket* to the server upon resumption. This ticket contains all the information required to resume the session. The same master secret is used across handshakes, but the default behavior is to perform a session key exchange upon resumption.

Key Takeaway

We call attention to this change in TLS because it illustrates the challenge of knowing which layer should solve a given problem. In isolation, session resumption as implemented in the earlier version of TLS seems like a good idea, but it needs to be considered in the context of the dominate use case, which is HTTP. Once the overhead of doing multiple TCP connections was addressed by HTTP, the equation for how resumption should be implemented by TLS changed. The bigger lesson is that we need to avoid rigid thinking about the right layer to implement a given function—the answer changes over time as the network evolves—where a holistic/cross-layer analysis is required to get the design right. *[Next]*

8.5.4 IP Security (IPsec)

Probably the most ambitious of all the efforts to integrate security into the Internet happens at the IP layer. Support for IPsec, as the architecture is called, is optional in IPv4 but mandatory in IPv6.

IPsec is really a framework (as opposed to a single protocol or system) for providing all the security services discussed throughout this chapter. IPsec provides three degrees of freedom. First, it is highly modular, allowing users (or more likely, system administrators) to select from a variety of cryptographic algorithms and specialized security protocols. Second, IPsec allows users to select from a large menu of security properties, including access control, integrity, authentication, originality, and confidentiality. Third, IPsec can be used to protect narrow streams (e.g., packets belonging to a particular TCP connection being sent between a pair of hosts) or wide streams (e.g., all packets flowing between a pair of routers).

When viewed from a high level, IPsec consists of two parts. The first part is a pair of protocols that implement the available security services. They are the Authentication Header (AH), which provides access

control, connectionless message integrity, authentication, and antireplay protection, and the Encapsulating Security Payload (ESP), which supports these same services, plus confidentiality. AH is rarely used so we focus on ESP here. The second part is support for key management, which fits under an umbrella protocol known as the Internet Security Association and Key Management Protocol (ISAKMP).

The abstraction that binds these two pieces together is the *security association* (SA). An SA is a simplex (one-way) connection with one or more of the available security properties. Securing a bidirectional communication between a pair of hosts—corresponding to a TCP connection, for example—requires two SAs, one in each direction. Although IP is a connectionless protocol, security depends on connection state information such as keys and sequence numbers. When created, an SA is assigned an ID number called a *security parameters index* (SPI) by the receiving machine. A combination of this SPI and the destination IP addresses uniquely identifies an SA. An ESP header includes the SPI so the receiving host can determine which SA an incoming packet belongs to and, hence, what algorithms and keys to apply to the packet.

SAs are established, negotiated, modified, and deleted using ISAKMP. It defines packet formats for exchanging key generation and authentication data. These formats aren't terribly interesting because they provide a framework only—the exact form of the keys and authentication data depends on the key generation technique, the cipher, and the authentication mechanism that is used. Moreover, ISAKMP does not specify a particular key exchange protocol, although it does suggest the Internet Key Exchange (IKE) as one possibility, and IKE v2 is what is used in practice.

ESP is the protocol used to securely transport data over an established SA. In IPv4, the ESP header follows the IP header; in IPv6, it is an extension header. Its format uses both a header and a trailer, as shown in Figure 8.17. The `SPI` field lets the receiving host identify the security association to which the packet belongs. The `SeqNum` field protects against replay attacks. The packet's `PayloadData` contains the data described by the `NextHdr` field. If confidentiality is selected, then the data is encrypted using whatever cipher was associated with the SA. The `PadLength` field records how much padding was added to the data; padding is sometimes necessary because, for example, the cipher requires the plaintext to be a multiple of a certain number of bytes or to ensure that the resulting ciphertext terminates on a 4-byte boundary. Finally, the `AuthenticationData` carries the authenticator.

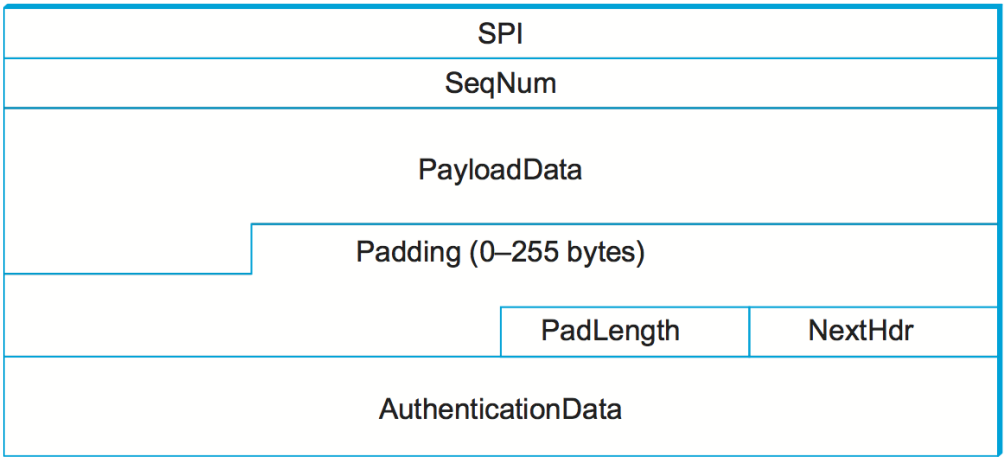


Figure 8.17.: IPSec's ESP format.

IPsec supports a *tunnel mode* as well as the more straightforward *transport mode*. Each SA operates in one or the other mode. In a transport mode SA, ESP's payload data is simply a message for a higher layer such as UDP or TCP. In this mode, IPsec acts as an intermediate protocol layer, much like SSL/TLS does between

TCP and a higher layer. When an ESP message is received, its payload is passed to the higher level protocol.

In a tunnel mode SA, however, ESP's payload data is itself an IP packet, as in Figure 8.18. The source and destination of this inner IP packet may be different from those of the outer IP packet. When an ESP message is received, its payload is forwarded on as a normal IP packet. The most common way to use the ESP is to build an "IPsec tunnel" between two routers, typically firewalls. For example, a corporation wanting to link two sites using the Internet could open a pair of tunnel-mode SAs between a router at one site and a router at the other site. An IP packet outgoing from one site would, at the outgoing router, become the payload of an ESP message sent to the other site's router. The receiving router would unwrap the payload IP packet and forward it on to its true destination.

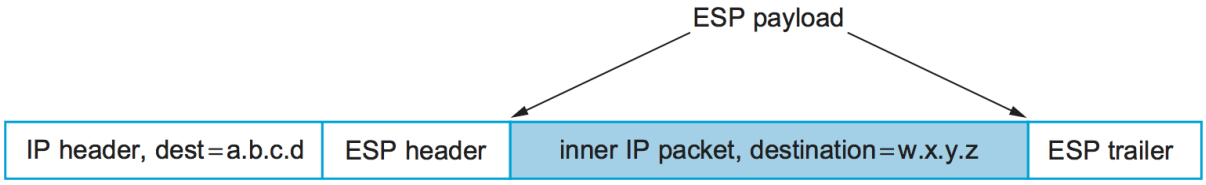


Figure 8.18.: An IP packet with a nested IP packet encapsulated using ESP in tunnel mode. Note that the inner and outer packets have different addresses.

These tunnels may also be configured to use ESP with confidentiality and authentication, thus preventing unauthorized access to the data that traverses this virtual link and ensuring that no spurious data is received at the far end of the tunnel. Furthermore, tunnels can provide traffic confidentiality, since multiplexing multiple flows through a single tunnel obscures information about how much traffic is flowing between particular endpoints. A network of such tunnels can be used to implement an entire virtual private network. Hosts communicating over a VPN need not even be aware that it exists.

8.5.5 Wireless Security (802.11i)

Wireless links are particularly exposed to security threats due to the lack of any physical security on the medium. While the convenience of 802.11 has prompted widespread acceptance of the technology, lack of security has been a recurring problem. For example, it is all too easy for an employee of a corporation to connect an 802.11 access point to the corporate network. Since radio waves pass through most walls, if the access point lacks the correct security measures, an attacker can now gain access to the corporate network from outside the building. Similarly, a computer with a wireless network adaptor inside the building could connect to an access point outside the building, potentially exposing it to attack, not to mention the rest of the corporate network if that same computer has, say, an Ethernet connection as well.

Consequently, there has been considerable work on securing Wi-Fi links. Somewhat surprisingly, one of the early security techniques developed for 802.11, known as Wired Equivalent Privacy (WEP), turned out to be seriously flawed and quite easily breakable.

The IEEE 802.11i standard provides authentication, message integrity, and confidentiality to 802.11 (Wi-Fi) at the link layer. *WPA3* (Wi-Fi Protected Access 3) is often used as a synonym for 802.11i, although it is technically a trademark of the Wi-Fi Alliance that certifies product compliance with 802.11i.

For backward compatibility, 802.11i includes definitions of first-generation security algorithms—including WEP—that are now known to have major security flaws. We will focus here on 802.11i's newer, stronger algorithms.

802.11i authentication supports two modes. In either mode, the end result of successful authentication is a shared Pairwise Master Key. *Personal mode*, also known as *Pre-Shared Key (PSK) mode*, provides weaker security but is more convenient and economical for situations like a home 802.11 network. The wireless device and the Access Point (AP) are preconfigured with a shared *passphrase*—essentially a very long password—from which the Pairwise Master Key is cryptographically derived.

802.11i's stronger authentication mode is based on the IEEE 802.1X framework for controlling access to a LAN, which uses an Authentication Server (AS) as in Figure 8.19. The AS and AP must be connected by a secure channel and could even be implemented as a single box, but they are logically separate. The AP forwards authentication messages between the wireless device and the AS. The protocol used for authentication is called the *Extensible Authentication Protocol* (EAP). EAP is designed to support multiple authentication methods—smart cards, Kerberos, one-time passwords, public key authentication, and so on—as well as both one-sided and mutual authentication. So EAP is better thought of as an authentication framework than a protocol. Specific EAP-compliant protocols, of which there are many, are called *EAP methods*. For example, EAP-TLS is an EAP method based on TLS authentication.

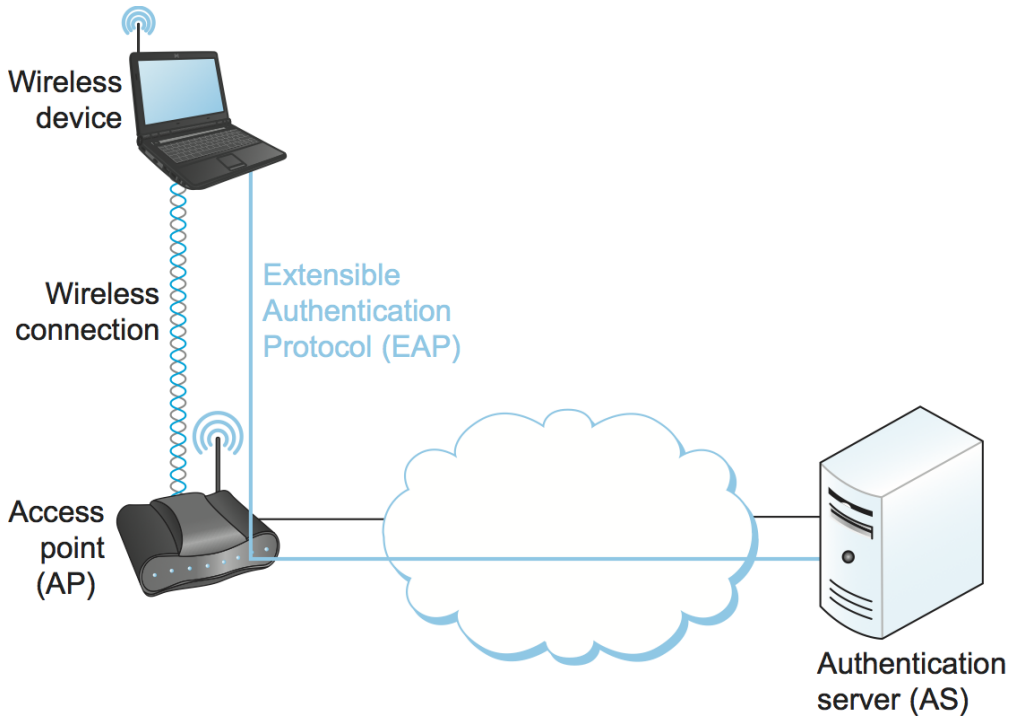


Figure 8.19.: Use of an Authentication Server in 802.11i.

802.11i does not place any restrictions on what the EAP method can use as a basis for authentication. It does, however, require an EAP method that performs *mutual* authentication, because not only do we want to prevent an adversary from accessing the network via our AP, we also want to prevent an adversary from fooling our wireless devices with a bogus, malicious AP. The end result of a successful authentication is a Pairwise Master Key shared between the wireless device and the AS, which the AS then conveys to the AP.

One of the main differences between the stronger AS-based mode and the weaker personal mode is that the former readily supports a unique key per client. This in turn makes it easier to change the set of clients that can authenticate themselves (e.g., to revoke access to one client) without needing to change the secret stored in every client.

With a Pairwise Master Key in hand, the wireless device and the AP execute a session key establishment protocol called the 4-way handshake to establish a Pairwise Transient Key. This Pairwise Transient Key is really a collection of keys that includes a session key called a *Temporal Key*. This session key is used by the protocol, called *CCMP*, that provides 802.11i's data confidentiality and integrity.

CCMP stands for CTR (Counter Mode) with CBC-MAC (Cipher-Block Chaining with Message Authentication Code) Protocol. CCMP uses AES in counter mode to encrypt for confidentiality. Recall that in counter mode encryption successive values of a counter are incorporated into the encryption of successive blocks of plaintext.

CCMP uses a Message Authentication Code (MAC) as an authenticator. The MAC algorithm is based on CBC, even though CCMP doesn't use CBC in the confidentiality encryption. In effect, CBC is performed without transmitting any of the CBC-encrypted blocks, solely so that the last CBC-encrypted block can be used as a MAC (only its first 8 bytes are actually used). The role of initialization vector is played by a specially constructed first block that includes a 48-bit packet number—a sequence number. (The packet number is also incorporated in the confidentiality encryption and serves to expose replay attacks.) The MAC is subsequently encrypted along with the plaintext in order to prevent birthday attacks, which depend on finding different messages with the same authenticator.

8.5.6 Firewalls

Whereas much of this chapter has focused on the uses of cryptography to provide such security features as authentication and confidentiality, there is a whole set of security issues that are not readily addressed by cryptographic means. For example, worms and viruses spread by exploiting bugs in operating systems and application programs (and sometimes human gullibility as well), and no amount of cryptography can help you if your machine has unpatched vulnerabilities. So other approaches are often used to keep out various forms of potentially harmful traffic. Firewalls are one of the most common ways to do this.

A firewall is a system that typically sits at some point of connectivity between a site it protects and the rest of the network, as illustrated in Figure 8.20. It is usually implemented as an "appliance" or part of a router, although a "personal firewall" may be implemented on an end-user machine. Firewall-based security depends on the firewall being the only connectivity to the site from outside; there should be no way to bypass the firewall via other gateways, wireless connections, or dial-up connections. The wall metaphor is somewhat misleading in the context of networks since a great deal of traffic passes through a firewall. One way to think of a firewall is that by default it blocks traffic unless that traffic is specifically allowed to pass through. For example, it might filter out all incoming messages except those addresses to a particular set of IP addresses or to particular TCP port numbers.

In effect, a firewall divides a network into a more-trusted zone internal to the firewall and a less-trusted zone external to the firewall. This is useful if you do not want external users to access a particular host or service within your site. Much of the complexity comes from the fact that you want to allow different kinds of access to different external users, ranging from the general public, to business partners, to remotely located members of your organization. A firewall may also impose restrictions on outgoing traffic to prevent certain attacks and to limit losses if an adversary succeeds in getting access inside the firewall.

The location of a firewall also often happens to be the dividing line between globally addressable regions and those that use local addresses. Hence, Network Address Translation (NAT) functionality and firewall functionality often are found in the same device, even though they are logically separate.

Firewalls may be used to create multiple *zones of trust*, such as a hierarchy of increasingly trusted zones. A

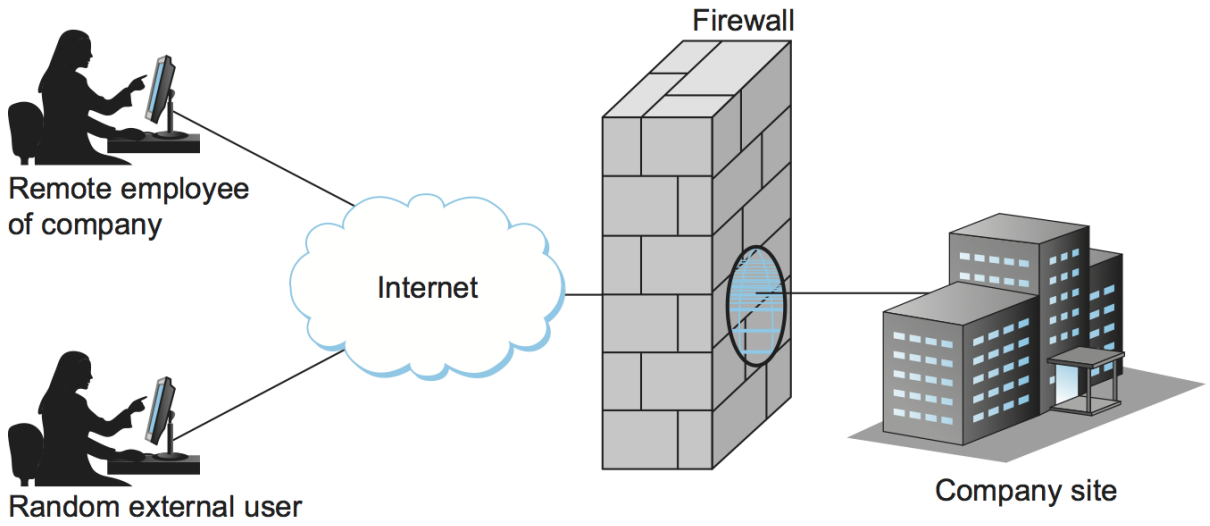


Figure 8.20.: A firewall filters packets flowing between a site and the rest of the Internet.

common arrangement involves three zones of trust: the internal network, the *DMZ* ("demilitarized zone"); and the rest of the Internet. The DMZ is used to hold services such as DNS and email servers that need to be accessible to the outside. Both the internal network and the outside world can access the DMZ, but hosts in the DMZ cannot access the internal network; therefore, an adversary who succeeds in compromising a host in the exposed DMZ still cannot access the internal network. The DMZ can be periodically restored to a clean state.

Firewalls filter based on IP, TCP, and UDP information, among other things. They are configured with a table of addresses that characterize the packets they will, and will not, forward. By addresses, we mean more than just the destination's IP address, although that is one possibility. Generally, each entry in the table is a 4-tuple: It gives the IP address and TCP (or UDP) port number for both the source and destination.

For example, a firewall might be configured to filter out (not forward) all packets that match the following description:

```
(192.12.13.14, 1234, 128.7.6.5, 80)
```

This pattern says to discard all packets from port 1234 on host 192.12.13.14 addressed to port 80 on host 128.7.6.5. (Port 80 is the well-known TCP port for HTTP.) Of course, it's often not practical to name every source host whose packets you want to filter, so the patterns can include wildcards. For example,

```
(*,  *, 128.7.6.5, 80)
```

says to filter out all packets addressed to port 80 on 128.7.6.5, regardless of what source host or port sent the packet. Notice that address patterns like these require the firewall to make forwarding/filtering decisions based on level 4 port numbers, in addition to level 3 host addresses. It is for this reason that network layer firewalls are sometimes called *level 4 switches*.

In the preceding discussion, the firewall forwards everything except where specifically instructed to filter out certain kinds of packets. A firewall could also filter out everything unless explicitly instructed to forward it, or use a mix of the two strategies. For example, instead of blocking access to port 80 on host 128.7.6.5, the firewall might be instructed to only allow access to port 25 (the SMTP mail port) on a particular mail server,

such as

```
(*,  *, 128.19.20.21, 25)
```

but to block all other traffic. Experience has shown that firewalls are very frequently configured incorrectly, allowing unsafe access. Part of the problem is that filtering rules can overlap in complex ways, making it hard for a system administrator to correctly express the intended filtering. A design principle that maximizes security is to configure a firewall to discard all packets other than those that are explicitly allowed. Of course, this means that some valid applications might be accidentally disabled; presumably users of those applications eventually notice and ask the system administrator to make the appropriate change.

Many client/server applications dynamically assign a port to the client. If a client inside a firewall initiates access to an external server, the server's response would be addressed to the dynamically assigned port. This poses a problem: How can a firewall be configured to allow an arbitrary server's response packet but disallow a similar packet for which there was no client request? This is not possible with a *stateless firewall*, which evaluates each packet in isolation. It requires a *stateful firewall*, which keeps track of the state of each connection. An incoming packet addressed to a dynamically assigned port would then be allowed only if it is a valid response in the current state of a connection on that port.

Modern firewalls also understand and filter based on many specific application-level protocols such as HTTP, Telnet, or FTP. They use information specific to that protocol, such as URLs in the case of HTTP, to decide whether to discard a message.

Strengths and Weaknesses of Firewalls

At best, a firewall protects a network from undesired access from the rest of the Internet; it cannot provide security to legitimate communication between the inside and the outside of the firewall. In contrast, the cryptography-based security mechanisms described in this chapter are capable of providing secure communication between any participants anywhere. This being the case, why are firewalls so common? One reason is that firewalls can be deployed unilaterally, using mature commercial products, while cryptography-based security requires support at both endpoints of the communication. A more fundamental reason for the dominance of firewalls is that they encapsulate security in a centralized place, in effect factoring security out of the rest of the network. A system administrator can manage the firewall to provide security, freeing the users and applications inside the firewall from security concerns—at least some kinds of security concerns.

Unfortunately, firewalls have serious limitations. Since a firewall does not restrict communication between hosts that are inside the firewall, the adversary who does manage to run code internal to a site can access all local hosts. How might an adversary get inside the firewall? The adversary could be a disgruntled employee with legitimate access, or the adversary's software could be hidden in some software installed from a CD or downloaded from the Web. It might be possible to bypass the firewall by using wireless communication or dial-up connections.

Another problem is that any parties granted access through your firewall, such as business partners or externally located employees, become a security vulnerability. If their security is not as good as yours, then an adversary could penetrate your security by penetrating their security.

On of the most serious problems for firewalls is their vulnerability to the exploitation of bugs in machines inside the firewall. Such bugs are discovered regularly, so a system administrator has to constantly monitor announcements of them. Administrators frequently fail to do so, since firewall security breaches routinely exploit security flaws that have been known for some time and have straightforward solutions.

Malware (for "malicious software") is the term for software that is designed to act on a computer in ways concealed from and unwanted by the computer's user. Viruses, worms, and spyware are common types of malware. (*Virus* is sometimes used synonymously with *malware*, but we will use it in the narrower sense in which it refers to only a particular kind of malware.) Malware code need not be natively executable object code; it could as well be interpreted code such as a script or an executable macro such as those used by Microsoft Word.

Viruses and *worms* are characterized by the ability to make and spread copies of themselves; the difference between them is that a worm is a complete program that replicates itself, while a virus is a bit of code that is inserted (and inserts copies of itself) into another piece of software or a file, so that it is executed as part of the execution of that piece of software or as a result of opening the file. Viruses and worms typically cause problems such as consuming network bandwidth as mere side effects of attempting to spread copies of themselves. Even worse, they can also deliberately damage a system or undermine its security in various ways. They could, for example, install a *backdoor*—software that allows remote access to the system without the normal authentication. This could lead to a firewall exposing a service that should be providing its own authentication procedures but has been undermined by a backdoor.

Spyware is software that, without authorization, collects and transmits private information about a computer system or its users. Usually spyware is secretly embedded in an otherwise useful program and is spread by users deliberately installing copies. The problem for firewalls is that the transmission of the private information looks like legitimate communication.

A natural question to ask is whether firewalls (or cryptographic security) could keep malware out of a system in the first place. Most malware is indeed transmitted via networks, although it may also be transmitted via portable storage devices such as CDs and memory sticks. Certainly this is one argument in favor of the "block everything not explicitly allowed" approach taken by many administrators in their firewall configurations.

One approach that is used to detect malware is to search for segments of code from known malware, sometimes called a *signature*. This approach has its own challenges, as cleverly designed malware can tweak its representation in various ways. There is also a potential impact on network performance to perform such detailed inspection of data entering a network. Cryptographic security cannot eliminate the problem either, although it does provide a means to authenticate the originator of a piece of software and detect any tampering, such as when a virus inserts a copy of itself.

Related to firewalls are systems known as *intrusion detection systems* (IDS) and *intrusion prevention systems* (IPS). These systems try to look for anomalous activity, such as an unusually large amount of traffic targeting a given host or port number, for example, and generate alarms for network managers or perhaps even take direct action to limit a possible attack. While there are commercial products in this space today, it is still a developing field.

Perspective: Blockchain and a Decentralized Internet

Probably without giving it much thought, users have invested enormous trust in the applications they use, especially those like Facebook and Google that not only store their personal photos and videos, but also manage their identity (i.e., provide *Single Sign On* for other web applications). This is troubling to many people, which has sparked interest in *decentralized platforms*, systems for which users do not have to trust a third-party. Such systems often build on top of a cryptocurrency like Bitcoin, not for its monetary value, but because cryptocurrency is itself based on a decentralized technology (called a *blockchain*) that no single

organization controls. It's easy to be distracted by all the hype, but a blockchain is essentially a decentralized log (ledger) that anyone can write a "fact" to, and later prove to the world that that fact was recorded.

Blockstack is an open source implementation of a decentralized platform, including the blockchain, but more interestingly, it has been used to implement a self-sovereign identity service for Internet applications. A self-sovereign identity service is a type of identity service that is *administratively decentralized*: it has no distinct service operator, and no single principal can control who can create an identity and who cannot. Blockstack uses a commodity public blockchain to build a replicated identity database log. When this database log is replayed by a Blockstack node, it produces the same view of all identities in the system as every other Blockstack node reading the same view of the underlying blockchain. Anyone can register an identity in Blockstack by appending to the blockchain.

Instead of requiring users to place trust in a distinct set of identity providers, Blockstack's identity protocol instead asks users to trust that the majority of the decision-making nodes in the blockchain (called *miners*) will preserve the order of writes (called *transactions*). The underlying blockchain provides a cryptocurrency to incentivize miners to do this. Under normal operation, miners stand to earn the most cryptocurrency by participating honestly. This allows Blockstack's database log to remain secure against tampering without a distinct service operator. An adversary who wishes to tamper with the log must compete against the majority of miners to produce an alternative transaction history in the underlying blockchain that the blockchain peer network will accept as the canonical write history.

The protocol for reading and appending to the Blockstack identity database log operates at a logical layer above the blockchain. Blockchain transactions are data frames for identity database log entries. A client appends to the identity database log by sending a blockchain transaction that embeds the database log entry, and a client reads the log back by extracting the log entries from blockchain transactions in the blockchain-given order. This makes it possible to implement the database log "on top" of any blockchain.

Identities in Blockstack are distinguished by user-chosen names. Blockstack's identity protocol binds a name to a public key and to some *routing state* (described below). It ensures that names are globally unique by assigning them on a first-come first-serve basis.

Names are registered in a two-step process—one to bind the client's public key to the *salted hash* of the name, and one to reveal the name itself. The two-step process is necessary to prevent front-running—only the client that signed the name hash may reveal the name, and only the client that calculated the salted hash can reveal the preimage. Once a name is registered, only the owner of the name's private key can transfer or revoke the name, or update its routing state.

Each name in Blockstack has an associated piece of routing state that contains one or more URLs that point to where the user's identity information can be found online. This data is too big and expensive to store on the blockchain directly, so instead Blockstack implements a layer of indirection: the hash of the routing state is written to the identity database log, and Blockstack peers implement a gossip network for disseminating and authenticating the routing state. Each peer maintains a full copy of the routing state.

Putting it all together, Figure 8.21 shows how resolving a name to its corresponding identity state works. Given a name, a client first queries a Blockstack peer for the corresponding public key and routing state (Step 1). Once it has the routing state, the client obtains the identity data by resolving the URL(s) contained within it and authenticates the identity information by verifying that it is signed by the name's public key (Step 2).

Broader Perspective

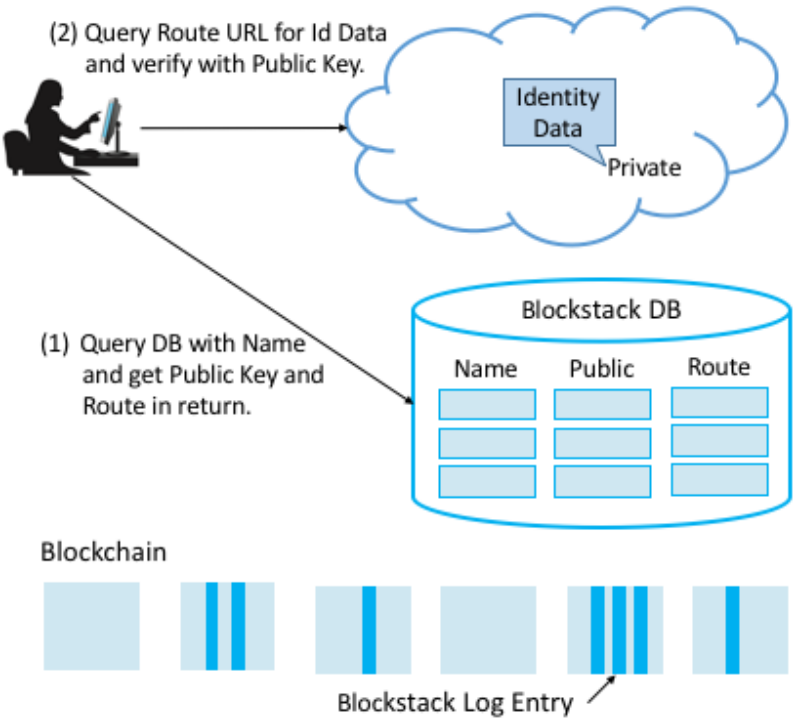


Figure 8.21.: Decentralized identity management built on a blockchain foundation.

To continue reading about the cloudification of the Internet, see *Perspective: The Cloud is the New Internet*.

To learn more about Blockstack and decentralizing the Internet, we recommend: Blockstack: A New Internet for Decentralized Applications, October 2017.

CHAPTER

NINE

APPLICATIONS

> Now this is not the end. It is not even the beginning of the end. But it is, perhaps, the end of the beginning.
>
> —*Winston Churchill*

Problem: Applications Need Their Own Protocols

We started this book by talking about application programs—everything from web browsers to videoconferencing tools—that people want to run over computer networks. In the intervening chapters, we have developed, one building block at a time, the networking infrastructure needed to make such applications possible. We have now come full circle, back to network applications. These applications are part network protocol (in the sense that they exchange messages with their peers on other machines) and part traditional application program (in the sense that they interact with the windowing system, the file system, and ultimately the user). This chapter explores some popular network applications available today.

Looking at applications drives home the *systems approach* that we have emphasized throughout this book. That is, the best way to build effective networked applications is to understand the building blocks that a network can provide and how those blocks can interact with each other. Thus, for example, a particular networked application might need to make use of a reliable transport protocol, authentication and privacy mechanisms, and resource allocation capabilities of the underlying network. Applications often work best when the application developer knows how to make the best use of these facilities (and there are also plenty of counter-examples of applications making poor use of available networking capabilities). Applications typically need their own protocols, too, in many cases using the same principles that we have seen in our prior examination of lower layer protocols. Thus, our focus in this chapter is on how to put together the ideas and techniques already described to build effective networked applications. Said another way, if you ever imagine yourself writing a network application, then you will by definition also become a protocol designer (and implementer).

We proceed by examining a variety of familiar, and not so familiar, network applications. These range from exchanging email and surfing the Web, to integrating applications across businesses, to multimedia applications like videoconferencing, to managing a set of network elements, to emerging peer-to-peer and content distribution networks. This list is by no means exhaustive, but it does serve to illustrate many of the key principles of designing and building applications. Applications need to pick and choose the appropriate building blocks that are available at other layers either inside the network or in the host protocol stacks and then augment those underlying services to provide the precise communication service required by the application.

9.1 Traditional Applications

We begin our discussion of applications by focusing on two of the most popular—the World Wide Web and email. Broadly speaking, both of these applications use the request/reply paradigm—users send requests to servers, which then respond accordingly. We refer to these as "traditional" applications because they typify the sort of applications that have existed since the early days of computer networks (although the Web is a lot newer than email but has its roots in file transfers that predated it). By contrast, later sections will look at a class of applications that have become popular more recently: streaming applications (e.g., multimedia applications like video and audio) and various overlay-based applications. (Note that there is a bit of a blurring between these classes, as you can of course get access to streaming multimedia data over the Web, but for now we'll focus on the general usage of the Web to request pages, images, etc.)

Before taking a close look at each of these applications, there are three general points that we need to make. The first is that it is important to distinguish between application *programs* and application *protocols*. For example, the HyperText Transport Protocol (HTTP) is an application protocol that is used to retrieve Web pages from remote servers. Many different application programs—that is, web clients like Internet Explorer, Chrome, Firefox, and Safari—provide users with a different look and feel, but all of them use the same HTTP protocol to communicate with web servers over the Internet. Indeed, it is the fact that the protocol is published and standardized that enables application programs developed by many different companies and individuals to interoperate. That is how so many browsers are able to interoperate with all the web servers (of which there are also many varieties).

This section looks at two very widely used, standardized application protocols:

- Simple Mail Transfer Protocol (SMTP) is used to exchange electronic mail.
- HyperText Transport Protocol (HTTP) is used to communicate between web browsers and web servers.

Second, we observe that many application layer protocols, including HTTP and SMTP, have a companion protocol that specifies the format of the data that can be exchanged. This is one reason WHY these protocols are relatively simple: Much of the complexity is managed in this companion standard. For example, SMTP is a protocol for exchanging electronic mail messages, but RFC 822 and Multipurpose Internet Mail Extensions (MIME) define the format of email messages. Similarly, HTTP is a protocol for fetching Web pages, but HyperText Markup Language (HTML) is a companion specification that defines the basic form of those pages.

Finally, since the application protocols described in this section follow the same request/reply communication pattern, you might expect that they would be built on top of a Remote Procedure Call (RPC) transport protocol. This is not the case, however, as they are instead implemented on top of TCP. In effect, each protocol reinvents a simple RPC-like mechanism on top of a reliable transport protocol (TCP). We say "simple" because each protocol is not designed to support arbitrary remote procedure calls of the sort discussed in an earlier chapter, but is instead designed to send and respond to a specific set of request messages. Interestingly, the approach taken by HTTP has proven quite powerful, which has led to it being adopted widely by the *Web Services* architecture, with general RPC mechanisms built *on top of HTTP* rather than the other way around. More on this topic at the end of this section.

9.1.1 Electronic Mail (SMTP, MIME, IMAP)

Email is one of the oldest network applications. After all, what could be more natural than wanting to send a message to the user at the other end of a cross-country link you just managed to get running? It's the 20th century's version of "*Mr. Watson, come here... I want to see you.*" Surprisingly, the pioneers of the ARPANET had not really envisioned email as a key application when the network was created—remote access to computing resources was the main design goal—but it turned out to be the Internet's original killer app.

As noted above, it is important (1) to distinguish the user interface (i.e., your mail reader) from the underlying message transfer protocols (such as SMTP or IMAP), and (2) to distinguish between this transfer protocol and a companion standard (RFC 822 and MIME) that defines the format of the messages being exchanged. We start by looking at the message format.

Message Format

RFC 822 defines messages to have two parts: a *header* and a *body*. Both parts are represented in ASCII text. Originally, the body was assumed to be simple text. This is still the case, although RFC 822 has been augmented by MIME to allow the message body to carry all sorts of data. This data is still represented as ASCII text, but because it may be an encoded version of, say, a JPEG image, it's not necessarily readable by human users. More on MIME in a moment.

The message header is a series of `<CRLF>`-terminated lines. (`<CRLF>` stands for carriage-return plus line-feed, which are a pair of ASCII control characters often used to indicate the end of a line of text.) The header is separated from the message body by a blank line. Each header line contains a type and value separated by a colon. Many of these header lines are familiar to users, since they are asked to fill them out when they compose an email message; for example, the header identifies the message recipient, and the header says something about the purpose of the message. Other headers are filled in by the underlying mail delivery system. Examples include (when the message was transmitted), (what user sent the message), and (each mail server that handled this message). There are, of course, many other header lines; the interested reader is referred to RFC 822.

RFC 822 was extended in 1993 (and updated quite a few times since then) to allow email messages to carry many different types of data: audio, video, images, PDF documents, and so on. MIME consists of three basic pieces. The first piece is a collection of header lines that augment the original set defined by RFC 822. These header lines describe, in various ways, the data being carried in the message body. They include (the version of MIME being used), (a human-readable description of what's in the message, analogous to the line), (the type of data contained in the message), and (how the data in the message body is encoded).

The second piece is definitions for a set of content types (and subtypes). For example, MIME defines two different still image types, denoted and , each with the obvious meaning. As another example, refers to simple text you might find in a vanilla 822-style message, while denotes a message that contains "marked up" text (text using special fonts, italics, etc.). As a third example, MIME defines an `application` type, where the subtypes correspond to the output of different application programs (e.g., `application/postscript` and `application/msword`).

MIME also defines a `multipart` type that says how a message carrying more than one data type is structured. This is like a programming language that defines both base types (e.g., integers and floats) and compound types (e.g., structures and arrays). One possible `multipart` subtype is `mixed`, which says

that the message contains a set of independent data pieces in a specified order. Each piece then has its own header line that describes the type of that piece.

The third piece is a way to encode the various data types so they can be shipped in an ASCII email message. The problem is that, for some data types (a JPEG image, for example), any given 8-bit byte in the image might contain one of 256 different values. Only a subset of these values are valid ASCII characters. It is important that email messages contain only ASCII, because they might pass through a number of intermediate systems (gateways, as described below) that assume all email is ASCII and would corrupt the message if it contained non-ASCII characters. To address this issue, MIME uses a straightforward encoding of binary data into the ASCII character set. The encoding is called `base64`. The idea is to map every three bytes of the original binary data into four ASCII characters. This is done by grouping the binary data into 24-bit units and breaking each such unit into four 6-bit pieces. Each 6-bit piece maps onto one of 64 valid ASCII characters; for example, 0 maps onto A, 1 maps onto B, and so on. If you look at a message that has been encoded using the base64 encoding scheme, you'll notice only the 52 upper- and lowercase letters, the 10 digits 0 through 9, and the special characters + and /. These are the first 64 values in the ASCII character set.

As one aside, so as to make reading mail as painless as possible for those who still insist on using text-only mail readers, a MIME message that consists of regular text only can be encoded using 7-bit ASCII. There's also a readable encoding for mostly ASCII data.

Putting this all together, a message that contains some plain text, a JPEG image, and a PostScript file would look something like this:

```
MIME-Version: 1.0
Content-Type: multipart/mixed;
boundary="-------417CA6E2DE4ABCAFBC5"
From: Alice Smith <Alice@cisco.com>
To: Bob@cs.Princeton.edu
Subject: promised material
Date: Mon, 07 Sep 1998 19:45:19 -0400

---------417CA6E2DE4ABCAFBC5
Content-Type: text/plain; charset=us-ascii
Content-Transfer-Encoding: 7bit

Bob,

Here is the jpeg image and draft report I promised.

--Alice

---------417CA6E2DE4ABCAFBC5
Content-Type: image/jpeg
Content-Transfer-Encoding: base64
... unreadable encoding of a jpeg figure
---------417CA6E2DE4ABCAFBC5
Content-Type: application/postscript; name="draft.ps"
Content-Transfer-Encoding: 7bit
... readable encoding of a PostScript document
```

In this example, the line in the message header says that this message contains various pieces, each denoted

by a character string that does not appear in the data itself. Each piece then has its own and lines.

Message Transfer

For many years, the majority of email was moved from host to host using only SMTP. While SMTP continues to play a central role, it is now just one email protocol of several, Internet Message Access Protocol (IMAP) and Post Office Protocol (POP) being two other important protocols for retrieving mail messages. We'll begin our discussion by looking at SMTP, and move on to IMAP below.

To place SMTP in the right context, we need to identify the key players. First, users interact with a *mail reader* when they compose, file, search, and read their email. Countless mail readers are available, just like there are many web browsers to choose from. In the early days of the Internet, users typically logged into the machine on which their *mailbox* resided, and the mail reader they invoked was a local application program that extracted messages from the file system. Today, of course, users remotely access their mailbox from their laptop or smartphone; they do not first log into the host that stores their mail (a mail server). A second mail transfer protocol, such as POP or IMAP, is used to remotely download email from a mail server to the user's device.

Second, there is a *mail daemon* (or process) running on each host that holds a mailbox. You can think of this process, also called a *message transfer agent* (MTA), as playing the role of a post office: Users (or their mail readers) give the daemon messages they want to send to other users, the daemon uses SMTP running over TCP to transmit the message to a daemon running on another machine, and the daemon puts incoming messages into the user's mailbox (where that user's mail reader can later find them). Since SMTP is a protocol that anyone could implement, in theory there could be many different implementations of the mail daemon. It turns out, though, that there are only a few popular implementations, with the old `sendmail` program from Berkeley Unix and `postfix` being the most widespread.

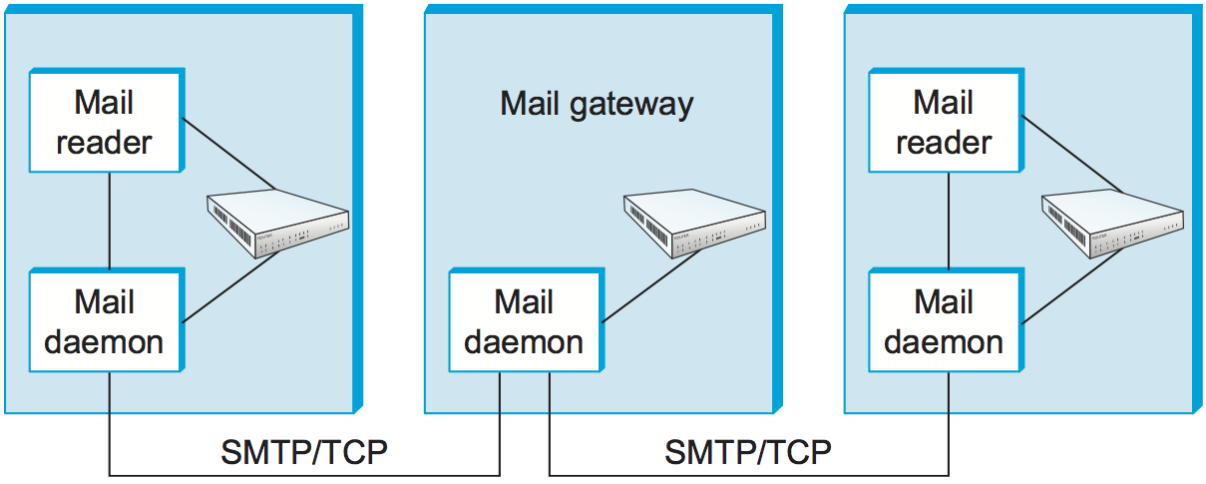


Figure 9.1.: Sequence of mail gateways store and forward email messages.

While it is certainly possible that the MTA on a sender's machine establishes an SMTP/TCP connection to the MTA on the recipient's mail server, in many cases the mail traverses one or more *mail gateways* on its route from the sender's host to the receiver's host. Like the end hosts, these gateways also run a message transfer agent process. It's not an accident that these intermediate nodes are called *gateways* since their job is to store and forward email messages, much like an "IP gateway" (which we have referred to as a *router*)

stores and forwards IP datagrams. The only difference is that a mail gateway typically buffers messages on disk and is willing to try retransmitting them to the next machine for several days, while an IP router buffers datagrams in memory and is only willing to retry transmitting them for a fraction of a second. Figure 9.1 illustrates a two-hop path from the sender to the receiver.

Why, you might ask, are mail gateways necessary? Why can't the sender's host send the message to the receiver's host? One reason is that the recipient does not want to include the specific host on which he or she reads email in his or her address. Another is scale: In large organizations, it's often the case that a number of different machines hold the *mailboxes* for the organization. For example, mail delivered to `bob@cs.princeton.edu` is first sent to a mail gateway in the CS Department at Princeton (that is, to the host named `cs.princeton.edu`), and then forwarded—involving a second connection—to the specific machine on which Bob has a mailbox. The forwarding gateway maintains a database that maps users into the machine on which their mailbox resides; the sender need not be aware of this specific name. (The list of header lines in the message will help you trace the mail gateways that a given message traversed.) Yet another reason, particularly true in the early days of email, is that the machine that hosts any given user's mailbox may not always be up or reachable, in which case the mail gateway holds the message until it can be delivered.

Independent of how many mail gateways are in the path, an independent SMTP connection is used between each host to move the message closer to the recipient. Each SMTP session involves a dialog between the two mail daemons, with one acting as the client and the other acting as the server. Multiple messages might be transferred between the two hosts during a single session. Since RFC 822 defines messages using ASCII as the base representation, it should come as no surprise to learn that SMTP is also ASCII based. This means it is possible for a human at a keyboard to pretend to be an SMTP client program.

SMTP is best understood by a simple example. The following is an exchange between sending host `cs.princeton.edu` and receiving host `cisco.com`. In this case, user Bob at Princeton is trying to send mail to users Alice and Tom at Cisco. Extra blank lines have been added to make the dialog more readable.

```
HELO cs.princeton.edu
250 Hello daemon@mail.cs.princeton.edu [128.12.169.24]

MAIL FROM:<Bob@cs.princeton.edu>
250 OK

RCPT TO:<Alice@cisco.com>
250 OK

RCPT TO:<Tom@cisco.com>
550 No such user here

DATA
354 Start mail input; end with <CRLF>.<CRLF>
Blah blah blah...
...etc. etc. etc.
<CRLF>.<CRLF>
250 OK

QUIT
221 Closing connection
```

As you can see, SMTP involves a sequence of exchanges between the client and the server. In each exchange,

the client posts a command (e.g., QUIT) and the server responds with a code (e.g., 221). The server also returns a human-readable explanation for the code (e.g.,). In this particular example, the client first identifies itself to the server with the HELO command. It gives its domain name as an argument. The server verifies that this name corresponds to the IP address being used by the TCP connection; you'll notice the server states this IP address back to the client. The client then asks the server if it is willing to accept mail for two different users; the server responds by saying "yes" to one and "no" to the other. Then the client sends the message, which is terminated by a line with a single period (".") on it. Finally, the client terminates the connection.

There are, of course, many other commands and return codes. For example, the server can respond to a client's RCPT command with a 251 code, which indicates that the user does not have a mailbox on this host, but that the server promises to forward the message onto another mail daemon. In other words, the host is functioning as a mail gateway. As another example, the client can issue a VRFY operation to verify a user's email address, but without actually sending a message to the user.

The only other point of interest is the arguments to the MAIL and RCPT operations; for example, FROM:<Bob@cs.princeton.edu> and TO:<Alice@cisco.com>, respectively. These look a lot like 822 header fields, and in some sense they are. What actually happens is that the mail daemon parses the message to extract the information it needs to run SMTP. The information it extracts is said to form an *envelope* for the message. The SMTP client uses this envelope to parameterize its exchange with the SMTP server. One historical note: The reason sendmail became so popular is that no one wanted to reimplement this message parsing function. While today's email addresses look pretty tame (e.g., Bob@cs.princeton.edu), this was not always the case. In the days before everyone was connected to the Internet, it was not uncommon to see email addresses of the form user%host@site!neighbor.

Mail Reader

The final step is for the user to actually retrieve his or her messages from the mailbox, read them, reply to them, and possibly save a copy for future reference. The user performs all these actions by interacting with a mail reader. As pointed out earlier, this reader was originally just a program running on the same machine as the user's mailbox, in which case it could simply read and write the file that implements the mailbox. This was the common case in the pre-laptop era. Today, most often the user accesses his or her mailbox from a remote machine using yet another protocol, such as POP or IMAP. It is beyond the scope of this book to discuss the user interface aspects of the mail reader, but it is definitely within our scope to talk about the access protocol. We consider IMAP, in particular.

IMAP is similar to SMTP in many ways. It is a client/server protocol running over TCP, where the client (running on the user's desktop machine) issues commands in the form of <CRLF>-terminated ASCII text lines and the mail server (running on the machine that maintains the user's mailbox) responds in kind. The exchange begins with the client authenticating him- or herself and identifying the mailbox he or she wants to access. This can be represented by the simple state transition diagram shown in Figure 9.2. In this diagram, LOGIN and LOGOUT are example commands that the client can issue, while OK is one possible server response. Other common commands include and EXPUNGE, with the obvious meanings. Additional server responses include NO (client does not have permission to perform that operation) and BAD (command is ill formed).

When the user asks to FETCH a message, the server returns it in MIME format and the mail reader decodes it. In addition to the message itself, IMAP also defines a set of message *attributes* that are exchanged as part of other commands, independent of transferring the message itself. Message attributes include information

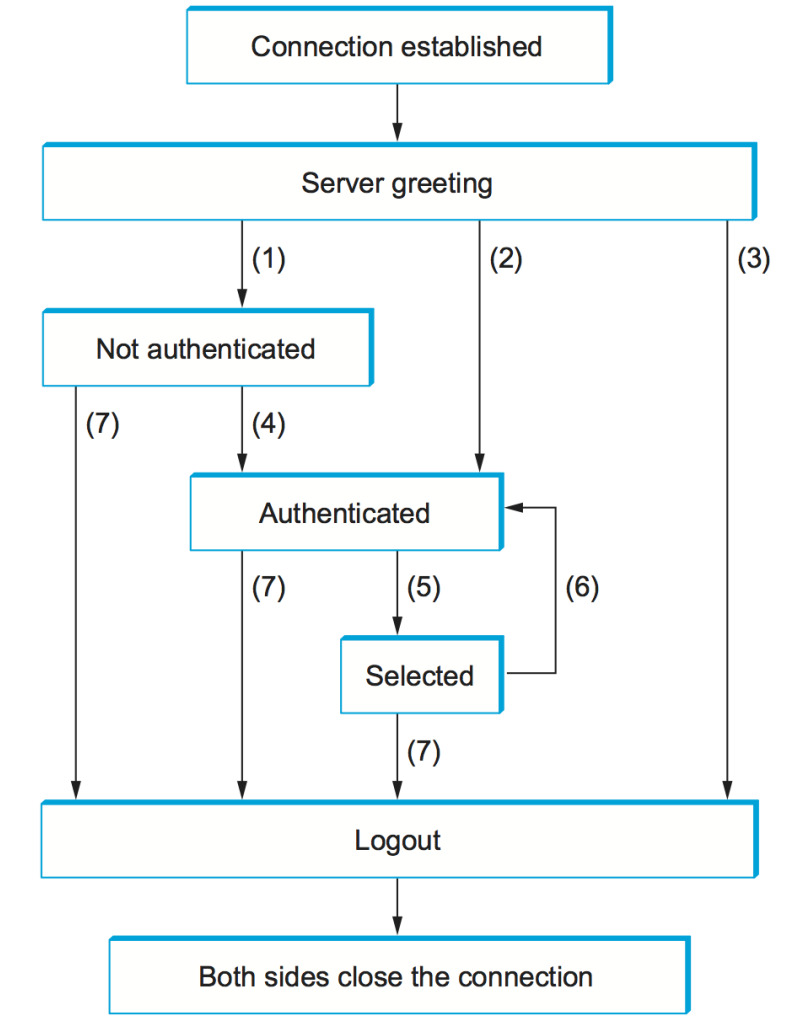


Figure 9.2.: IMAP state transition diagram.

like the size of the message and, more interestingly, various *flags* associated with the message (e.g., and `Recent`). These flags are used to keep the client and server synchronized; that is, when the user deletes a message in the mail reader, the client needs to report this fact to the mail server. Later, should the user decide to expunge all deleted messages, the client issues an `EXPUNGE` command to the server, which knows to actually remove all earlier deleted messages from the mailbox.

Finally, note that when the user replies to a message, or sends a new message, the mail reader does not forward the message from the client to the mail server using IMAP, but it instead uses SMTP. This means that the user's mail server is effectively the first mail gateway traversed along the path from the desktop to the recipient's mailbox.

9.1.2 World Wide Web (HTTP)

The World Wide Web has been so successful and has made the Internet accessible to so many people that sometimes it seems to be synonymous with the Internet. In fact, the design of the system that became the Web started around 1989, long after the Internet had become a widely deployed system. The original goal of the Web was to find a way to organize and retrieve information, drawing on ideas about hypertext—interlinked documents—that had been around since at least the 1960s.[1] The core idea of hypertext is that one document can link to another document, and the protocol (HTTP) and document language (HTML) were designed to meet that goal.

One helpful way to think of the Web is as a set of cooperating clients and servers, all of whom speak the same language: HTTP. Most people are exposed to the Web through a graphical client program or web browser like Safari, Chrome, Firefox, or Internet Explorer. Figure 9.3 shows the Safari browser in use, displaying a page of information from Princeton University.

Clearly, if you want to organize information into a system of linked documents or objects, you need to be able to retrieve one document to get started. Hence, any web browser has a function that allows the user to obtain an object by opening a URL. Uniform Resource Locators (URLs) are so familiar to most of us by now that it's easy to forget that they haven't been around forever. They provide information that allows objects on the Web to be located, and they look like the following:

```
http://www.cs.princeton.edu/index.html
```

If you opened that particular URL, your web browser would open a TCP connection to the web server at a machine called `www.cs.princeton.edu` and immediately retrieve and display the file called `index.html`. Most files on the Web contain images and text, and many have other objects such as audio and video clips, pieces of code, etc. They also frequently include URLs that point to other files that may be located on other machines, which is the core of the "hypertext" part of HTTP and HTML. A web browser has some way in which you can recognize URLs (often by highlighting or underlining some text) and then you can ask the browser to open them. These embedded URLs are called *hypertext links*. When you ask your web browser to open one of these embedded URLs (e.g., by pointing and clicking on it with a mouse), it will open a new connection and retrieve and display a new file. This is called *following a link*. It thus becomes very easy to hop from one machine to another around the network, following links to all sorts of information. Once you have a means to embed a link in a document and allow a user to follow that link to get another document, you have the basis of a hypertext system.

[1] A short history of the Web provided by the World Wide Web consortium traces its roots to a 1945 article describing links between microfiche documents.

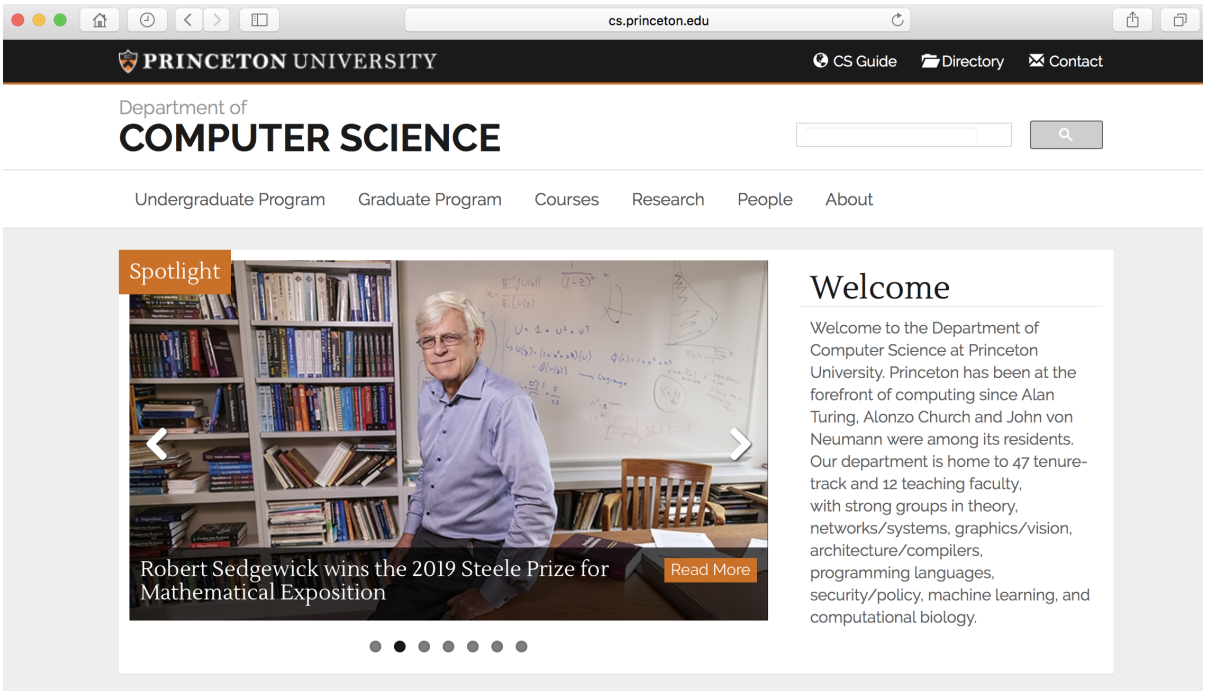


Figure 9.3.: The Safari web browser.

When you ask your browser to view a page, your browser (the client) fetches the page from the server using HTTP running over TCP. Like SMTP, HTTP is a text-oriented protocol. At its core, HTTP is a request/response protocol, where every message has the general form

```
START_LINE <CRLF>
MESSAGE_HEADER <CRLF>
<CRLF>
MESSAGE_BODY <CRLF>
```

where, as before, `<CRLF>` stands for carriage-return+line-feed. The first line (`START_LINE`) indicates whether this is a request message or a response message. In effect, it identifies the "remote procedure" to be executed (in the case of a request message), or the *status* of the request (in the case of a response message). The next set of lines specifies a collection of options and parameters that qualify the request or response. There are zero or more of these `MESSAGE_HEADER` lines—the set is terminated by a blank line—each of which looks like a header line in an email message. HTTP defines many possible header types, some of which pertain to request messages, some to response messages, and some to the data carried in the message body. Instead of giving the full set of possible header types, though, we just give a handful of representative examples. Finally, after the blank line comes the contents of the requested message (`MESSAGE_BODY`); this part of the message is where a server would place the requested page when responding to a request, and it is typically empty for request messages.

Why does HTTP run over TCP? The designers didn't have to do it that way, but TCP does provide a pretty good match to what HTTP needs, particularly by providing reliable delivery (who wants a Web page with missing data?), flow control, and congestion control. However, as we'll see below, there are a few issues that can arise from building a request/response protocol on top of TCP, especially if you ignore the subtleties of the interactions between the application and transport layer protocols.

Request Messages

The first line of an HTTP request message specifies three things: the operation to be performed, the Web page the operation should be performed on, and the version of HTTP being used. Although HTTP defines a wide assortment of possible request operations—including *write* operations that allow a Web page to be posted on a server—the two most common operations are `GET` (fetch the specified Web page) and `HEAD` (fetch status information about the specified Web page). The former is obviously used when your browser wants to retrieve and display a Web page. The latter is used to test the validity of a hypertext link or to see if a particular page has been modified since the browser last fetched it. The full set of operations is summarized in Table 9.1. As innocent as it sounds, the `POST` command enables much mischief (including spam) on the Internet.

Table 9.1.: HTTP Request Operations.

Operation	Description
OPTIONS	Request information about available options
GET	Retrieve document identified in URL
HEAD	Retrieve metainformation about document identified in URL
POST	Give information (e.g., annotation) to server
PUT	Store document under specified URL
DELETE	Delete specified URL
TRACE	Loopback request message
CONNECT	For use by proxies

For example, the `START_LINE`

```
GET http://www.cs.princeton.edu/index.html
HTTP/1.1
```

says that the client wants the server on host to return the page named . This particular example uses an *absolute* URL. It is also possible to use a *relative* identifier and specify the host name in one of the `MESSAGE_HEADER` lines; for example,

```
GET index.html HTTP/1.1
Host: www.cs.princeton.edu
```

Here, `Host` is one of the possible `MESSAGE_HEADER` fields. One of the more interesting of these is , which gives the client a way to conditionally request a Web page—the server returns the page only if it has been modified since the time specified in that header line.

Response Messages

Like request messages, response messages begin with a single `START_LINE`. In this case, the line specifies the version of HTTP being used, a three-digit code indicating whether or not the request was successful, and a text string giving the reason for the response. For example, the `START_LINE`

```
HTTP/1.1 202 Accepted
```

indicates that the server was able to satisfy the request, while

```
HTTP/1.1 404 Not Found
```

indicates that it was not able to satisfy the request because the page was not found. There are five general types of response codes, with the first digit of the code indicating its type. Table 9.2 summarizes the five types of codes.

Table 9.2.: Five Types of HTTP Result Codes.

Code	Type	Example Reasons
1xx	Informational	request received, continuing process
2xx	Success	action successfully received, understood, and accepted
3xx	Redirection	further action must be taken to complete the request
4xx	Client Error	request contains bad syntax or cannot be fulfilled
5xx	Server Error	server failed to fulfill an apparently valid request

As with the unexpected consequences of the `POST` request message, it is sometimes surprising how various response messages are used in practice. For example, request redirection (specifically code 302) turns out to be a powerful mechanism that plays a big role in Content Distribution Networks (CDNs) by redirecting requests to a nearby cache.

Also similar to request messages, response messages can contain one or more `MESSAGE_HEADER` lines. These lines relay additional information back to the client. For example, the `Location` header line specifies that the requested URL is available at another location. Thus, if the Princeton CS Department Web page had moved from to , for example, then the server at the original address might respond with

```
HTTP/1.1 301 Moved Permanently
Location: http://www.princeton.edu/cs/index.html
```

In the common case, the response message will also carry the requested page. This page is an HTML document, but since it may carry nontextual data (e.g., a GIF image), it is encoded using MIME (see the previous section). Certain of the `MESSAGE_HEADER` lines give attributes of the page contents, including (number of bytes in the contents), `Expires` (time at which the contents are considered stale), and (time at which the contents were last modified at the server).

Uniform Resource Identifiers

The URLs that HTTP uses as addresses are one type of *Uniform Resource Identifier* (URI). A URI is a character string that identifies a resource, where a resource can be anything that has identity, such as a document, an image, or a service.

The format of URIs allows various more specialized kinds of resource identifiers to be incorporated into the URI space of identifiers. The first part of a URI is a *scheme* that names a particular way of identifying a certain kind of resource, such as `mailto` for email addresses or `file` for file names. The second part of a URI, separated from the first part by a colon, is the *scheme-specific part*. It is a resource identifier consistent with the scheme in the first part, as in the URIs `mailto:santa@northpole.org` and `file:///C:/foo.html`.

A resource doesn't have to be retrievable or accessible. We saw an example of this in an earlier chapter—extensible markup language (XML) namespaces are identified by URIs that look an awful lot like URLs, but strictly speaking they are not *locators* because they don't tell you how to locate something; they just provide a globally unique identifier for the namespace. There is no requirement that you can retrieve anything at the URI given as the target namespace of an XML document. We'll see another example of a URI that is not a URL in a later section.

TCP Connections

The original version of HTTP (1.0) established a separate TCP connection for each data item retrieved from the server. It's not too hard to see how this was a very inefficient mechanism: connection setup and teardown messages had to be exchanged between the client and server even if all the client wanted to do was verify that it had the most recent copy of a page. Thus, retrieving a page that included some text and a dozen icons or other small graphics would result in 13 separate TCP connections being established and closed. Figure 9.4 shows the sequence of events for fetching a page that has just a single embedded object. Colored lines indicate TCP messages, while black lines indicate the HTTP requests and responses. (Some of the TCP ACKs are not shown to avoid cluttering the picture.) You can see two round trip times are spent setting up TCP connections while another two (at least) are spent getting the page and image. As well as the latency impact, there is also processing cost on the server to handle the extra TCP connection establishment and termination.

To overcome this situation, HTTP version 1.1 introduced *persistent connections*—the client and server can exchange multiple request/response messages over the same TCP connection. Persistent connections have many advantages. First, they obviously eliminate the connection setup overhead, thereby reducing the load on the server, the load on the network caused by the additional TCP packets, and the delay perceived by the user. Second, because a client can send multiple request messages down a single TCP connection, TCP's congestion window mechanism is able to operate more efficiently. This is because it's not necessary to go through the slow start phase for each page. Figure 9.5 shows the transaction from Figure 9.4 using a persistent connection in the case where the connection is already open (presumably due to some prior access of the same server).

Persistent connections do not come without a price, however. The problem is that neither the client nor server necessarily knows how long to keep a particular TCP connection open. This is especially critical on the server, which might be asked to keep connections opened on behalf of thousands of clients. The solution is that the server must time out and close a connection if it has received no requests on the connection for a period of time. Also, both the client and server must watch to see if the other side has elected to close the connection, and they must use that information as a signal that they should close their side of the connection as well. (Recall that both sides must close a TCP connection before it is fully terminated.) Concerns about this added complexity may be one reason why persistent connections were not used from the outset, but today it is widely accepted that the benefits of persistent connections more than offset the drawbacks.

While 1.1 is still widely supported, a new version (2.0) was formally approved by the IETF in 2015. Known as HTTP/2, the new version is backwards compatible with 1.1 (i.e,. it adopts the same syntax for header fields, status codes, and URIs), but it adds two new features.

The first is to make it easier for web servers to *minify* the information they send back to web browsers. If you look closely at the makeup of the HTML in a typical web page, you will find a plethora of references to other bits-and-pieces (e.g., images, scripts, style files) that the browser needs to render the page. Rather than force the client to request these bits-and-pieces (technically known as *resources*) in subsequent requests,

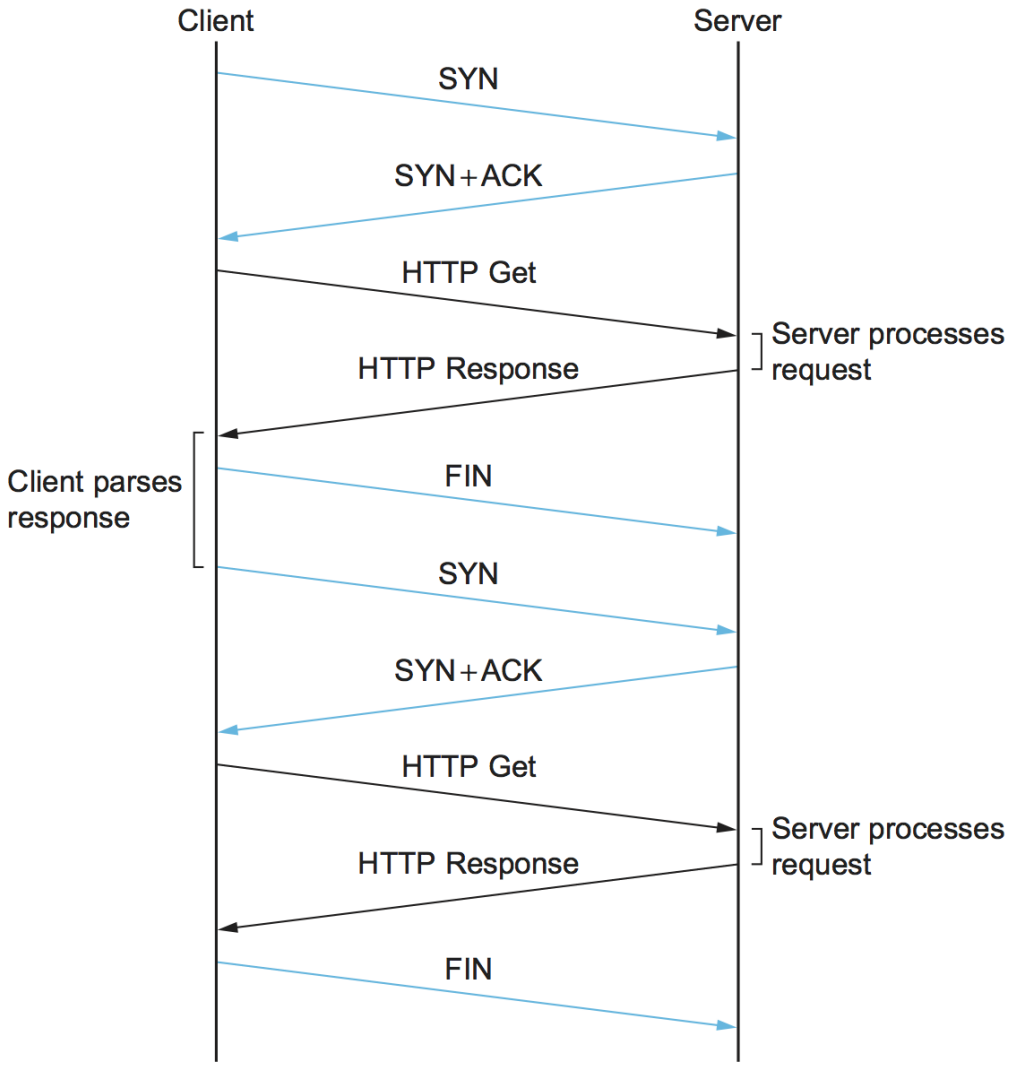


Figure 9.4.: HTTP 1.0 behavior.

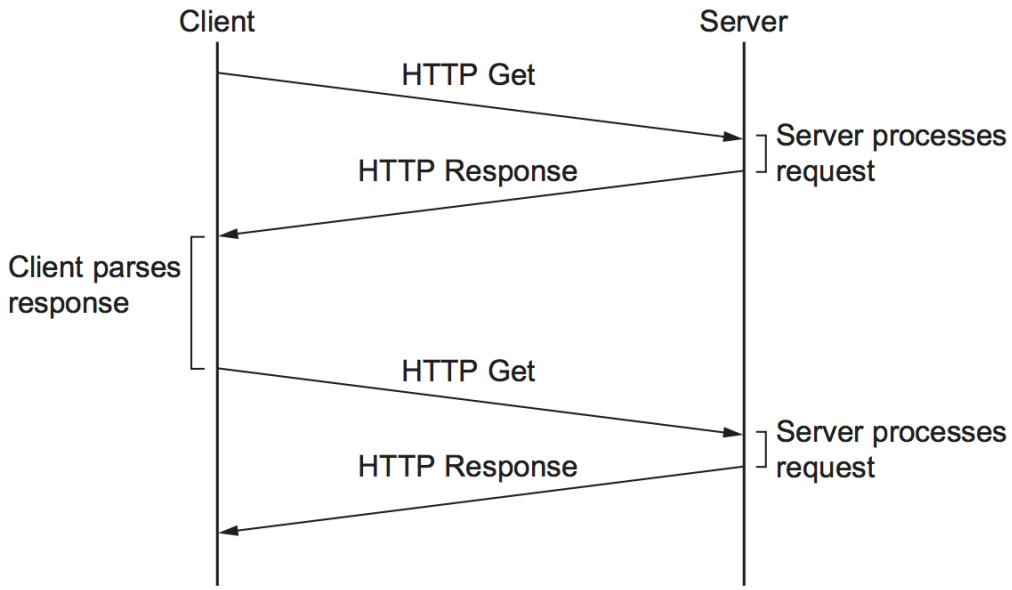


Figure 9.5.: HTTP 1.1 behavior with persistent connections.

HTTP/2 provides a means for the server to bundle the required resources and proactively *push* them to the client without incurring the round-trip delay of forcing the client to request them. This feature is coupled with a compression mechanism that reduces the number of bytes that need to be pushed. The whole goal is to minimize the latency an end-user experiences from the moment they click on a hyperlink until the selected page is fully rendered.

The second big advance of HTTP/2 is to multiplex several requests on a single TCP connection. This goes beyond what version 1.1 supports—allowing a *sequence* of requests to reuse a TCP connection—by permitting these requests to overlap with each other. The way HTTP/2 does this should sound familiar: it defines a *channel* abstraction (technically, the channels are called *streams*), permits multiple concurrent streams to be active at a given time (each labeled with a unique *stream id*), and limits each stream to one active request/reply exchange at a time.

Caching

An important implementation strategy that makes the web more usable is to cache Web pages. Caching has many benefits. From the client's perspective, a page that can be retrieved from a nearby cache can be displayed much more quickly than if it has to be fetched from across the world. From the server's perspective, having a cache intercept and satisfy a request reduces the load on the server.

Caching can be implemented in many different places. For example, a user's browser can cache recently accessed pages and simply display the cached copy if the user visits the same page again. As another example, a site can support a single site-wide cache. This allows users to take advantage of pages previously downloaded by other users. Closer to the middle of the Internet, Internet Service Providers (ISPs) can cache pages.[2] Note that, in the second case, the users within the site most likely know what machine is caching

[2] There are quite a few issues with this sort of caching, ranging from the technical to the regulatory. One example of a technical challenge is the effect of *asymmetric paths*, when the request to the server and the response to the client do not follow the same sequence of router hops.

pages on behalf of the site, and they configure their browsers to connect directly to the caching host. This node is sometimes called a *proxy*. In contrast, the sites that connect to the ISP are probably not aware that the ISP is caching pages. It simply happens to be the case that HTTP requests coming out of the various sites pass through a common ISP router. This router can peek inside the request message and look at the URL for the requested page. If it has the page in its cache, it returns it. If not, it forwards the request to the server and watches for the response to fly by in the other direction. When it does, the router saves a copy in the hope that it can use it to satisfy a future request.

No matter where pages are cached, the ability to cache Web pages is important enough that HTTP has been designed to make the job easier. The trick is that the cache needs to make sure it is not responding with an out-of-date version of the page. For example, the server assigns an expiration date (the `Expires` header field) to each page it sends back to the client (or to a cache between the server and client). The cache remembers this date and knows that it need not reverify the page each time it is requested until after that expiration date has passed. After that time (or if that header field is not set) the cache can use the `HEAD` or conditional `GET` operation (`GET` with header line) to verify that it has the most recent copy of the page. More generally, there are a set of *cache directives* that must be obeyed by all caching mechanisms along the request/response chain. These directives specify whether or not a document can be cached, how long it can be cached, how fresh a document must be, and so on. We'll look at the related issue of CDNs—which are effectively distributed caches—in a later section.

9.1.3 Web Services

So far we have focused on interactions between a human and a web server. For example, a human uses a web browser to interact with a server, and the interaction proceeds in response to input from the user (e.g., by clicking on links). However, there is increasing demand for direct computer-to-computer interaction. And, just as the applications described above need protocols, so too do the applications that communicate directly with each other. We conclude this section by looking at the challenges of building large numbers of application-to-application protocols and some of the proposed solutions.

Much of the motivation for enabling direct application-to-application communication comes from the business world. Historically, interactions between enterprises—businesses or other organizations—have involved some manual steps such as filling out an order form or making a phone call to determine whether some product is in stock. Even within a single enterprise it is common to have manual steps between software systems that cannot interact directly because they were developed independently. Increasingly, such manual interactions are being replaced with direct application-to-application interaction. An ordering application at enterprise A would send a message to an order fulfillment application at enterprise B, which would respond immediately indicating whether the order can be filled. Perhaps, if the order cannot be filled by B, the application at A would immediately order from another supplier or solicit bids from a collection of suppliers.

Here is a simple example of what we are talking about. Suppose you buy a book at an online retailer like Amazon. Once your book has been shipped, Amazon could send you the tracking number in an email, and then you could head over to the website for the shipping company—`http://www.fedex.com`, perhaps—and track the package. However, you can also track your package directly from the Amazon.com website. In order to make this happen, Amazon has to be able to send a query to FedEx, in a format that FedEx understands, interpret the result, and display it in a Web page that perhaps contains other information about your order. Underlying the user experience of getting all the information about the order served up at once on the Amazon.com Web page is the fact that Amazon and FedEx had to have a protocol for

exchanging the information needed to track packages—call it the Package Tracking Protocol. It should be clear that there are so many potential protocols of this type that we'd better have some tools to simplify the task of specifying them and building them.

Network applications, even those that cross organization boundaries, are not new—email and web browsing cross such boundaries. What is new about this problem is the scale. Not scale in the size of the network, but scale in the number of different kinds of network applications. Both the protocol specifications and the implementations of those protocols for traditional applications like electronic mail and file transfer have typically been developed by a small group of networking experts. To enable the vast number of potential network applications to be developed quickly, it was necessary to come up with some technologies that simplify and automate the task of application protocol design and implementation.

Two architectures have been advocated as solutions to this problem. Both architectures are called *Web Services*, taking their name from the term for the individual applications that offer a remotely accessible service to client applications to form network applications. The terms used as informal shorthand to distinguish the two Web Services architectures are *SOAP* and *REST*. We will discuss the technical meanings of those terms shortly.

The SOAP architecture's approach to the problem is to make it feasible, at least in theory, to generate protocols that are customized to each network application. The key elements of the approach are a framework for protocol specification, software toolkits for automatically generating protocol implementations from the specifications, and modular partial specifications that can be reused across protocols.

The REST architecture's approach to the problem is to regard individual Web Services as World Wide Web resources—identified by URIs and accessed via HTTP. Essentially, the REST architecture is just the Web architecture. The Web architecture's strengths include stability and a demonstrated scalability (in the network-size sense). It could be considered a weakness that HTTP is not well suited to the usual procedural or operation-oriented style of invoking a remote service. REST advocates argue, however, that rich services can nonetheless be exposed using a more data-oriented or document-passing style for which HTTP is well suited.

Custom Application Protocols (WSDL, SOAP)

The architecture informally referred to as SOAP is based on *Web Services Description Language* (WSDL) and *SOAP*.[3] Both of these standards are issued by the World Wide Web Consortium (W3C). This is the architecture that people usually mean when they use the term Web Services without any preceding qualifier. As these standards are still evolving, our discussion here is effectively a snapshot.

WSDL and SOAP are frameworks for specifying and implementing application protocols and transport protocols, respectively. They are generally used together, although that is not strictly required. WSDL is used to specify application-specific details such as what operations are supported, the formats of the application data to invoke or respond to those operations, and whether an operation involves a response. SOAP's role is to make it easy to define a transport protocol with exactly the desired semantics regarding protocol features such as reliability and security.

Both WSDL and SOAP consist primarily of a protocol specification language. Both languages are based on XML with an eye toward making specifications accessible to software tools such as stub compilers and directory services. In a world of many custom protocols, support for automating generation of implementations is crucial to avoid the effort of manually implementing each protocol. Support software generally

[3] Although the name *SOAP* originated as an acronym, it officially no longer stands for anything.

takes the form of toolkits and application servers developed by third-party vendors, which allows developers of individual Web Services to focus more on the business problem they need to solve (such as tracking the package purchased by a customer).

Defining Application Protocols

WSDL has chosen a procedural *operation* model of application protocols. An abstract Web Service interface consists of a set of named operations, each representing a simple interaction between a client and the Web Service. An operation is analogous to a remotely callable procedure in an RPC system. An example from W3C's WSDL Primer is a hotel reservation Web Service with two operations, CheckAvailability and MakeReservation.

Each operation specifies a *Message Exchange Pattern* (MEP) that gives the sequence in which the messages are to be transmitted, including the fault messages to be sent when an error disrupts the message flow. Several MEPs are predefined, and new custom MEPs can be defined, but it appears that in practice only two MEPs are being used: **In-Only** (a single message from client to service) and **In-Out** (a request from client and a corresponding reply from service). These patterns should be very familiar, and suggest that the costs of supporting MEP flexibility perhaps outweigh the benefits.

MEPs are templates that have placeholders instead of specific message types or formats, so part of the definition of an operation involves specifying which message formats to map into the placeholders in the pattern. Message formats are not defined at the bit level that is typical of protocols we have discussed. They are instead defined as an abstract data model using XML. XML Schema provides a set of primitive data types and ways to define compound data types. Data that conforms to an XML Schema-defined format—its abstract data model—can be concretely represented using XML, or it can use another representation, such as the "binary" representation Fast Infoset.

WSDL nicely separates the parts of a protocol that can be specified abstractly—operations, MEPs, abstract message formats—from the parts that must be concrete. WSDL's concrete part specifies an underlying protocol, how MEPs are mapped onto it, and what bit-level representation is used for messages on the wire. This part of a specification is known as a *binding*, although it is better described as an implementation, or a mapping onto an implementation. WSDL has predefined bindings for HTTP and SOAP-based protocols, with parameters that allow the protocol designer to fine-tune the mapping onto those protocols. There is a framework for defining new bindings, but SOAP protocols dominate.

A crucial aspect of how WSDL mitigates the problem of specifying large numbers of protocols is through reuse of what are essentially specification modules. The WSDL specification of a Web Service may be composed of multiple WSDL documents, and individual WSDL documents may also be used in other Web Service specifications. This modularity makes it easier to develop a specification and easier to ensure that, if two specifications are supposed to have some elements that are identical (for example, so that they can be supported by the same tool), then those elements are indeed identical. This modularity, together with WSDL's defaulting rules, also helps keep specifications from becoming overwhelmingly verbose for human protocol designers.

WSDL modularity should be familiar to anyone who has developed moderately large pieces of software. A WSDL document need not be a complete specification; it could, for example, define a single message format. The partial specifications are uniquely identified using XML Namespaces; each WSDL document specifies the URI of a *target namespace*, and any new definitions in the document are named in the context of that namespace. One WSDL document can incorporate components of another by *including* the second

document if both share the same target namespace or *importing* it if the target namespaces differ.

Defining Transport Protocols

Although SOAP is sometimes called a protocol, it is better thought of as a framework for defining protocols. As the SOAP 1.2 specification explains, "SOAP provides a simple messaging framework whose core functionality is concerned with providing extensibility." SOAP uses many of the same strategies as WSDL, including message formats defined using XML Schema, bindings to underlying protocols, Message Exchange Patterns, and reusable specification elements identified using XML namespaces.

SOAP is used to define transport protocols with exactly the features needed to support a particular application protocol. SOAP aims to make it feasible to define many such protocols by using reusable components. Each component captures the header information and logic that go into implementing a particular feature. To define a protocol with a certain set of features, just compose the corresponding components. Let's look more closely at this aspect of SOAP.

SOAP 1.2 introduced a *feature* abstraction, which the specification describes thus:

> *A SOAP feature is an extension of the SOAP messaging framework. Although SOAP poses no constraints on the potential scope of such features, example features may include "reliability," "security," "correlation," "routing," and message exchange patterns (MEPs) such as request/response, one-way, and peer-to-peer conversations.*

A SOAP feature specification must include:

- A URI that identifies the feature
- The state information and processing, abstractly described, that is required at each SOAP node to implement the feature
- The information to be relayed to the next node
- (If the feature is a MEP) the life cycle and temporal/causal relationships of the messages exchanged—for example, responses follow requests and are sent to the originator of the request

Note that this formalization of the concept of a protocol feature is rather low level; it is almost a design.

Given a set of features, there are two strategies for defining a SOAP protocol that will implement them. One is by layering: binding SOAP to an underlying protocol in such a way as to derive the features. For example, we could obtain a request/response protocol by binding SOAP to HTTP, with a SOAP request in an HTTP request and a SOAP reply in an HTTP response. Because this is such a common example, it happens that SOAP has a predefined binding to HTTP; new bindings may be defined using the SOAP Protocol Binding Framework.

The second and more flexible way to implement features involves *header blocks*. A SOAP message consists of an Envelope, which contains a Header that contains header blocks, and a Body, which contains the payload destined for the ultimate receiver. This message structure is illustrated in Figure 9.6.

It should be a familiar notion by now that certain header information corresponds to particular features. A digital signature is used to implement authentication, a sequence number is used for reliability, and a checksum is used to detect message corruption. A SOAP header block is intended to encapsulate the header information that corresponds to a particular feature. The correspondence is not always one-to-one since multiple header blocks could be involved in a single feature, or a single header block could be used in

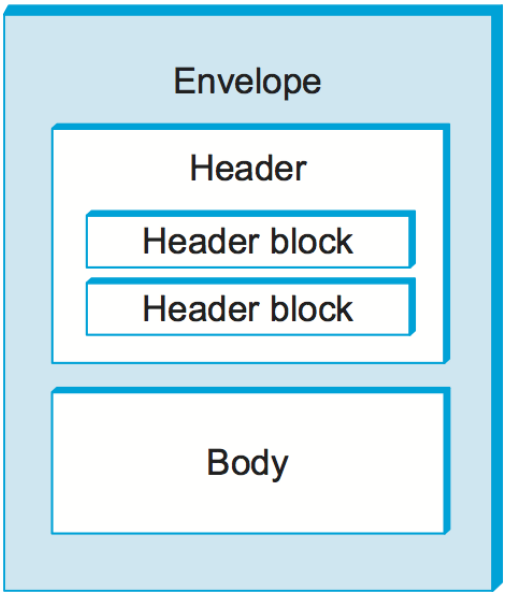


Figure 9.6.: SOAP message structure.

multiple features. A *SOAP module* is a specification of the syntax and the semantics of one or more header blocks. Each module is intended to provide one or more features and must declare the features it implements.

The goal behind SOAP modules is to be able to compose a protocol with a set of features by simply including each of the corresponding module specifications. If your protocol is required to have at-most-once semantics and authentication, include the corresponding modules in your specification. This represents a novel approach to modularizing protocol services, an alternative to the protocol layering we have seen throughout this book. It is a bit like flattening a series of protocol layers into a single protocol, but in a structured way. It remains to be seen how well SOAP features and modules, introduced in version 1.2 of SOAP, will work in practice. The main weakness of this scheme is that modules may well interfere with each other. A module specification is required to specify any *known* interactions with other SOAP modules, but clearly that doesn't do much to alleviate the problem. On the other hand, a core set of features and modules that provides the most important properties may be small enough to be well known and well understood.

Standardizing Web Services Protocols

As we've said, WSDL and SOAP aren't protocols; they are standards for *specifying* protocols. For different enterprises to implement Web Services that interoperate with each other, it is not enough to agree to use WSDL and SOAP to define their protocols; they must agree on—standardize—specific protocols. For example, you could imagine that online retailers and shipping companies might like to standardize a protocol by which they exchange information, along the lines of the simple package tracking example at the start of this section. This standardization is crucial for tool support as well as interoperability. And, yet, different network applications in this architecture must necessarily differ in at least the message formats and operations they use.

This tension between standardization and customization is tackled by establishing partial standards called *profiles*. A profile is a set of guidelines that narrow or constrain choices available in WSDL, SOAP, and other standards that may be referenced in defining a protocol. They may at the same time resolve ambiguities or

gaps in those standards. In practice, a profile often formalizes an emerging *de facto* standard.

The broadest and most widely adopted profile is known as the *WS-I Basic Profile*. It was proposed by the Web Services Interoperability Organization (WS-I), an industry consortium, while WSDL and SOAP are specified by the World Wide Web Consortium (W3C). The Basic Profile resolves some of the most basic choices faced in defining a Web Service. Most notably it requires that WSDL be bound exclusively to SOAP and SOAP be bound exclusively to HTTP and use the HTTP POST method. It also specifies which versions of WSDL and SOAP must be used.

The *WS-I Basic Security Profile* adds security constraints to the Basic Profile by specifying how the SSL/TLS layer is to be used and requiring conformance to *WS-Security* (Web Services Security). WS-Security specifies how to use various existing techniques such as X.509 public key certificates and Kerberos to provide security features in SOAP protocols.

WS-Security is just the first of a growing suite of SOAP-level standards established by the industry consortium OASIS (Organization for the Advancement of Structured Information Standards). The standards known collectively as *WS-** include WS-Reliability, WS-ReliableMessaging, WS-Coordination, and WS-AtomicTransaction.

A Generic Application Protocol (REST)

The WSDL/SOAP Web Services architecture is based on the assumption that the best way to integrate applications across networks is via protocols that are customized to each application. That architecture is designed to make it practical to specify and implement all those protocols. In contrast, the REST Web Services architecture is based on the assumption that the best way to integrate applications across networks is by re-applying the model underlying the World Wide Web architecture. This model, articulated by Web architect Roy Fielding, is known as *REpresentational State Transfer* (REST). There is no need for a new REST architecture for Web Services—the existing Web architecture is sufficient, although a few extensions are probably necessary. In the Web architecture, individual Web Services are regarded as resources identified by URIs and accessed via HTTP—a single generic application protocol with a single generic addressing scheme.

Where WSDL has user-defined operations, REST uses the small set of available HTTP methods, such as `GET` and `POST` (see Table 9.1). So how can these simple methods provide an interface to a rich Web Service? By employing the REST model, in which the complexity is shifted from the protocol to the payload. The payload is a representation of the abstract state of a resource. For example, a `GET` could return a representation of the current state of the resource, and a `POST` could send a representation of a desired state of the resource.

The representation of a resource state is abstract; it need not resemble how the resource is actually implemented by a particular Web Service instance. It is not necessary to transmit a complete resource state in each message. The size of messages can be reduced by transmitting just the parts of a state that are of interest (e.g., just the parts that are being modified). And, because Web Services share a single protocol and address space with other web resources, parts of states can be passed by reference—by URI—even when they are other Web Services.

This approach is best summarized as a data-oriented or document-passing style, as opposed to a procedural style. Defining an application protocol in this architecture consists of defining the document structure (i.e., the state representation). XML and the lighter-weight JavaScript Object Notation (JSON) are the most frequently used presentation languages for this state. Interoperability depends on agreement, between a Web

Service and its clients, on the state representation. Of course, the same is true in the SOAP architecture; a Web Service and its client have to be in agreement on payload format. The difference is that in the SOAP architecture interoperability additionally depends on agreement on the protocol; in the REST architecture, the protocol is always HTTP, so that source of interoperability problems is eliminated.

One of the selling features of REST is that it leverages the infrastructure that has been deployed to support the Web. For example, Web proxies can enforce security or cache information. Existing content distribution networks (CDNs) can be used to support RESTful applications.

In contrast with WSDL/SOAP, the Web has had time for standards to stabilize and to demonstrate that it scales very well. It also comes with some security in the form of Secure Socket Layer (SSL)/Transport Layer Security (TLS). The Web and REST may also have an advantage in evolvability. Although the WSDL and SOAP *frameworks* are highly flexible with regard to what new features and bindings can go into the definition of a protocol, that flexibility is irrelevant once the protocol is defined. Standardized protocols such as HTTP are designed with a provision for being extended in a backward-compatible way. HTTP's own extensibility takes the form of headers, new methods, and new content types. Protocol designers using WSDL/SOAP need to design such extensibility into each of their custom protocols. Of course, the designers of state representations in a REST architecture also have to design for evolvability.

An area where WSDL/SOAP may have an advantage is in adapting or wrapping previously written, "legacy" applications to conform to Web Services. This is an important point since most Web Services will be based on legacy applications for the near future at least. These applications usually have a procedural interface that maps more easily into WSDL's operations than REST states. The REST versus WSDL/SOAP competition may very well hinge on how easy or difficult it turns out to be to devise REST-style interfaces for individual Web Services. We may find that some Web Services are better served by WSDL/SOAP and others by REST.

The online retailer Amazon, as it happens, was an early adopter (2002) of Web Services. Interestingly, Amazon made its systems publicly accessible via *both* of the Web Services architectures, and according to some reports a substantial majority of developers use the REST interface. Of course, this is just one data point and may well reflect factors specific to Amazon.

From Web Services to Cloud Services

If Web Services is what we call it when the web server that implements my application sends a request to the web server that implements your application, then what do we call it when we both put our applications in the cloud so that they can support scalable workloads? We can call both of them *Cloud Services* if we want to, but is that a distinction without a difference? It depends.

Moving a server process from a physical machine running in my machine room into a virtual machine running in a cloud provider's datacenter shifts responsibility for keeping the machine running from my system admin to the cloud provider's operations team, but the application is still designed according to the Web Services architecture. On the other hand, if the application is designed from scratch to run on a scalable cloud platform, for example by adhering to the *micro-services architecture*, then we say the application is *cloud native*. So the important distinction is cloud native versus legacy web services deployed in the cloud.

We briefly saw the micro-services architecture in Chapter 5 when describing gRPC, and although it's difficult to definitively declare micro-services superior to web services, the current trend in industry almost certainly favors the former. More interesting, perhaps, is the ongoing debate about REST+Json versus gRPC+Protbufs as the preferred RPC mechanism for implementing micro-services. Keeping in mind that both run on top of HTTP, we leave it as an exercise for the reader to pick a side and defend it.

9.2 Multimedia Applications

Just like the traditional applications described in the previous section, multimedia applications such as telephony and videoconferencing need their own protocols. Much of the initial experience in designing protocols for multimedia applications came from the MBone tools—applications such as `vat` and `vic` that were developed for use on the MBone, an overlay network that supports IP multicast to enable multiparty conferencing. (More on overlay networks including the MBone in the next section.) Initially, each application implemented its own protocol (or protocols), but it became apparent that many multimedia applications have common requirements. This ultimately led to the development of a number of general-purpose protocols for use by multimedia applications.

We have already seen a number of protocols that multimedia applications use. The Real-Time Transport Protocol (RTP) provides many of the functions that are common to multimedia applications such as conveying timing information and identifying the coding schemes and media types of an application.

The Resource Reservation Protocol (RSVP) can be used to request the allocation of resources in the network so that the desired quality of service (QoS) can be provided to an application. We will see how resource allocation interacts with other aspects of multimedia applications later in this section.

In addition to these protocols for multimedia transport and resource allocation, many multimedia applications also need a signalling or *session control* protocol. For example, suppose that we wanted to be able to make telephone calls across the Internet (Voice over IP, or VoIP). We would need some mechanism to notify the intended recipient of such a call that we wanted to talk to her, such as by sending a message to some multimedia device that would cause it to make a ringing sound. We would also like to be able to support features like call forwarding, three-way calling, etc. The Session Initiation Protocol (SIP) and H.323 are examples of protocols that address the issues of session control; we begin our discussion of multimedia applications by examining these protocols.

9.2.1 Session Control and Call Control (SDP, SIP, H.323)

To understand some of the issues of session control, consider the following problem. Suppose you want to hold a videoconference at a certain time and make it available to a wide number of participants. Perhaps you have decided to encode the video stream using the MPEG-2 standard, to use the multicast IP address 224.1.1.1 for transmission of the data, and to send it using RTP over UDP port number 4000. How would you make all that information available to the intended participants? One way would be to put all that information in an email and send it out, but ideally there should be a standard format and protocol for disseminating this sort of information. The IETF has defined protocols for just this purpose. The protocols that have been defined include

- Session Description Protocol (SDP)
- Session Announcement Protocol (SAP)
- Session Initiation Protocol (SIP)
- Simple Conference Control Protocol (SCCP)

You might think that this is a lot of protocols for a seemingly simple task, but there are many aspects of the problem and several different situations in which it must be addressed. For example, there is a difference between announcing the fact that a certain conference session is going to be made available on the MBone

(which would be done using SDP and SAP) and trying to make an Internet phone call to a certain user at a particular time (which could be done using SDP and SIP). In the former case, you could consider your job done once you have sent all the session information in a standard format to a well-known multicast address. In the latter, you would need to locate one or more users, get a message to them announcing your desire to talk (analogous to ringing their phone), and perhaps negotiate a suitable audio encoding among all parties. We will look first at SDP, which is common to many applications, then at SIP, which is widely used for a number of interactive applications such as Internet telephony.

Session Description Protocol (SDP)

The Session Description Protocol (SDP) is a rather general protocol that can be used in a variety of situations and is typically used in conjunction with one or more other protocols (e.g., SIP). It conveys the following information:

- The name and purpose of the session
- Start and end times for the session
- The media types (e.g., audio, video) that comprise the session
- Detailed information required to receive the session (e.g., the multicast address to which data will be sent, the transport protocol to be used, the port numbers, the encoding scheme)

SDP provides this information formatted in ASCII using a sequence of lines of text, each of the form "." An example of an SDP message will illustrate the main points.

```
v=0
o=larry 2890844526 2890842807 IN IP4 128.112.136.10
s=Networking 101
i=A class on computer networking
u=http://www.cs.princeton.edu/
e=larry@cs.princeton.edu
c=IN IP4 224.2.17.12/127
t=2873397496 2873404696
m=audio 49170 RTP/AVP 0
m=video 51372 RTP/AVP 31
m=application 32416 udp wb
```

Note that SDP, like HTML, is fairly easy for a human to read but has strict formatting rules that make it possible for machines to interpret the data unambiguously. For example, the SDP specification defines all the possible information types that are allowed to appear, the order in which they must appear, and the format and reserved words for every type that is defined.

The first thing to notice is that each information type is identified by a single character. For example, the line tells us that "version" has the value zero; that is, this message is formatted according to version zero of SDP. The next line provides the "origin" of the session which contains enough information to uniquely identify the session. `larry` is a username of the session creator, and is the IP address of his computer. The number following `larry` is a session identifier that is chosen to be unique to that machine. This is followed by a "version" number for the SDP announcement; if the session information was updated by a later message, the version number would be increased.

The next three lines (`i`, `s` and `u`) provide the session name, a session description, and a session Uniform Resource Identifier (URI, as described earlier in this chapter)—information that would be helpful to a user in deciding whether to participate in this session. Such information could be displayed in the user interface of a session directory tool that shows current and upcoming events that have been advertised using SDP. The next line (`e=...`) contains an email address of a person to contact regarding the session. Figure 9.7 shows a screen shot of a (now archaic) session directory tool called `sdr` along with the descriptions of several sessions that had been announced at the time the picture was taken.

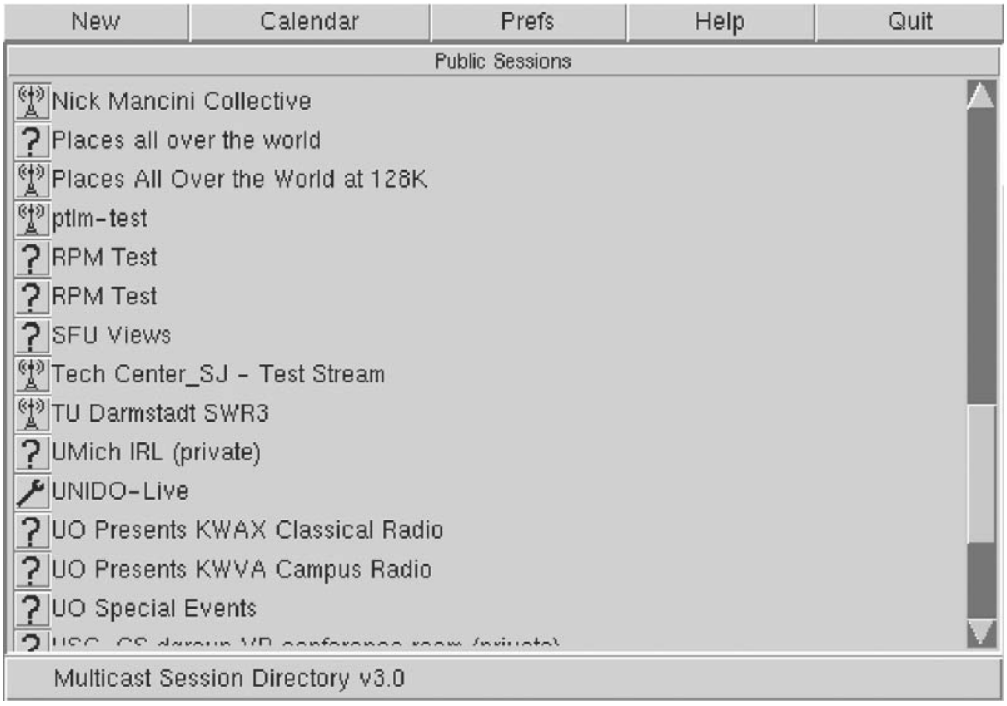


Figure 9.7.: A session directory tool displays information extracted from SDP messages.

Next we get to the technical details that would enable an application program to participate in the session. The line beginning `c=...` provides the IP multicast address to which data for this session will be sent; a user would need to join this multicast group to receive the session. Next we see the start and end times for the session (encoded as integers according to the Network Time Protocol). Finally, we get to the information about the media for this session. This session has three media types available—audio, video, and a shared whiteboard application known as "`wb`." For each media type there is one line of information formatted as follows:

```
m=<media> <port> <transport> <format>
```

The media types are self-explanatory, and the port numbers in each case are UDP ports. When we look at the "transport" field, we can see that the `wb` application runs directly over UDP, while the audio and video are transported using "RTP/AVP." This means that they run over RTP and use the *application profile* known as *AVP*. That application profile defines a number of different encoding schemes for audio and video; we can see in this case that the audio is using encoding 0 (which is an encoding using an 8-kHz sampling rate and 8 bits per sample) and the video is using encoding 31, which represents the H.261 encoding scheme. These "magic numbers" for the encoding schemes are defined in the RFC that defines the AVP profile; it is also possible to describe nonstandard coding schemes in SDP.

Finally, we see a description of the "wb" media type. All the encoding information for this data is specific to the `wb` application, and so it is sufficient just to provide the name of the application in the "format" field. This is analogous to putting `application/wb` in a MIME message.

Now that we know how to describe sessions, we can look at how they can be initiated. One way in which SDP is used is to announce multimedia conferences, by sending SDP messages to a well-known multicast address. The session directory tool shown in Figure 9.7 would function by joining that multicast group and displaying information that it gleans from received SDP messages. SDP is also used in the delivery of entertainment video of IP (often called IPTV) to provide information about the video content on each TV channel.

SDP also plays an important role in conjunction with the Session Initiation Protocol (SIP). With the widespread adoption of Voice over IP (i.e., the support of telephony-like applications over IP networks) and IP-based video conferencing, SIP is now one of the more important members of the Internet protocol suite.

SIP

SIP is an application layer protocol that bears a certain resemblance to HTTP, being based on a similar request/response model. However, it is designed with rather different sorts of applications in mind and thus provides quite different capabilities than HTTP. The capabilities provided by SIP can be grouped into five categories:

- User location—Determining the correct device with which to communicate to reach a particular user
- User availability—Determining if the user is willing or able to take part in a particular communication session
- User capabilities—Determining such items as the choice of media and coding scheme to use
- Session setup—Establishing session parameters such as port numbers to be used by the communicating parties
- Session management—A range of functions including transferring sessions (e.g., to implement "call forwarding") and modifying session parameters

Most of these functions are easy enough to understand, but the issue of location bears some further discussion. One important difference between SIP and, say, HTTP, is that SIP is primarily used for human-to-human communication. Thus, it is important to be able to locate individual *users*, not just machines. And, unlike email, it's not good enough just to locate a server that the user will be checking on at some later date and dump the message there—we need to know where the user is right now if we want to be able to communicate with him in real time. This is further complicated by the fact that a user might choose to communicate using a range of different devices, such as using his desktop PC when he's in the office and using a handheld device when traveling. Multiple devices might be active at the same time and might have widely different capabilities (e.g., an alphanumeric pager and a PC-based video "phone"). Ideally, it should be possible for other users to be able to locate and communicate with the appropriate device at any time. Furthermore, the user must be able to have control over when, where, and from whom he receives calls.

To enable a user to exercise the appropriate level of control over his calls, SIP introduces the notion of a proxy. A SIP proxy can be thought of as a point of contact for a user to which initial requests for communication with him are sent. Proxies also perform functions on behalf of callers. We can see how proxies work

best through an example.

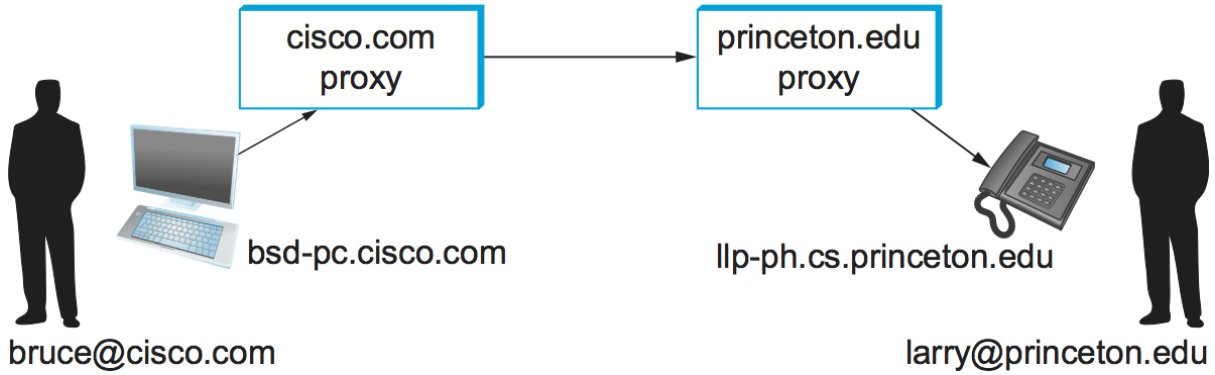


Figure 9.8.: Establishing communication through SIP proxies.

Consider the two users in Figure 9.8. The first thing to notice is that each user has a name in the format `user@domain`, very much like an email address. When user Bruce wants to initiate a session with Larry, he sends his initial SIP message to the local proxy for his domain, `cisco.com`. Among other things, this initial message contains a *SIP URI*—these are a form of uniform resource identifier which look like this:

```
SIP:larry@princeton.edu
```

A SIP URI provides complete identification of a user, but (unlike a URL) does not provide his location, since that may change over time. We will see shortly how the location of a user can be determined.

Upon receiving the initial message from Bruce, the proxy looks at the SIP URI and deduces that this message should be sent to the proxy. For now, we assume that the proxy has access to some database that enables it to obtain a mapping from the name to the IP address of one or more devices at which Larry currently wishes to receive messages. The proxy can therefore forward the message on to Larry's chosen device(s). Sending the message to more than one device is called *forking* and may be done either in parallel or in series (e.g., send it to his mobile phone if he doesn't answer the phone at his desk).

The initial message from Bruce to Larry is likely to be a SIP `invite` message, which looks something like the following:

```
INVITE sip:larry@princeton.edu SIP/2.0
Via: SIP/2.0/UDP bsd-pc.cisco.com;branch=z9hG4bK433yte4
To: Larry <sip:larry@princeton.edu>
From: Bruce <sip:bruce@cisco.com>;tag=55123
Call-ID: xy745jj210re3@bsd-pc.cisco.com
CSeq: 271828 INVITE
Contact: <sip:bruce@bsd-pc.cisco.com>
Content-Type: application/sdp
Content-Length: 142
```

The first line identifies the type of function to be performed (`invite`); the resource on which to perform it, the called party (`sip:larry@princeton.edu`); and the protocol version (2.0). The subsequent header lines probably look somewhat familiar because of their resemblance to the header lines in an email message. SIP defines a large number of header fields, only some of which we describe here. Note that the header in this example identifies the device from which this message originated. The and headers describe the contents of the message following the header, just as in a MIME-encoded email message. In this case,

the content is an SDP message. That message would describe such things as the type of media (audio, video, etc.) that Bruce would like to exchange with Larry and other properties of the session such as codec types that he supports. Note that the field in SIP provides the capability to use any protocol for this purpose, although SDP is the most common.

Returning to the example, when the `invite` message arrives at the proxy, not only does the proxy forward the message on toward `princeton.edu`, but it also responds to the sender of the `invite`. Just as in HTTP, all responses have a response code, and the organization of codes is similar to that for HTTP. In Figure 9.9 we can see a sequence of SIP messages and responses.

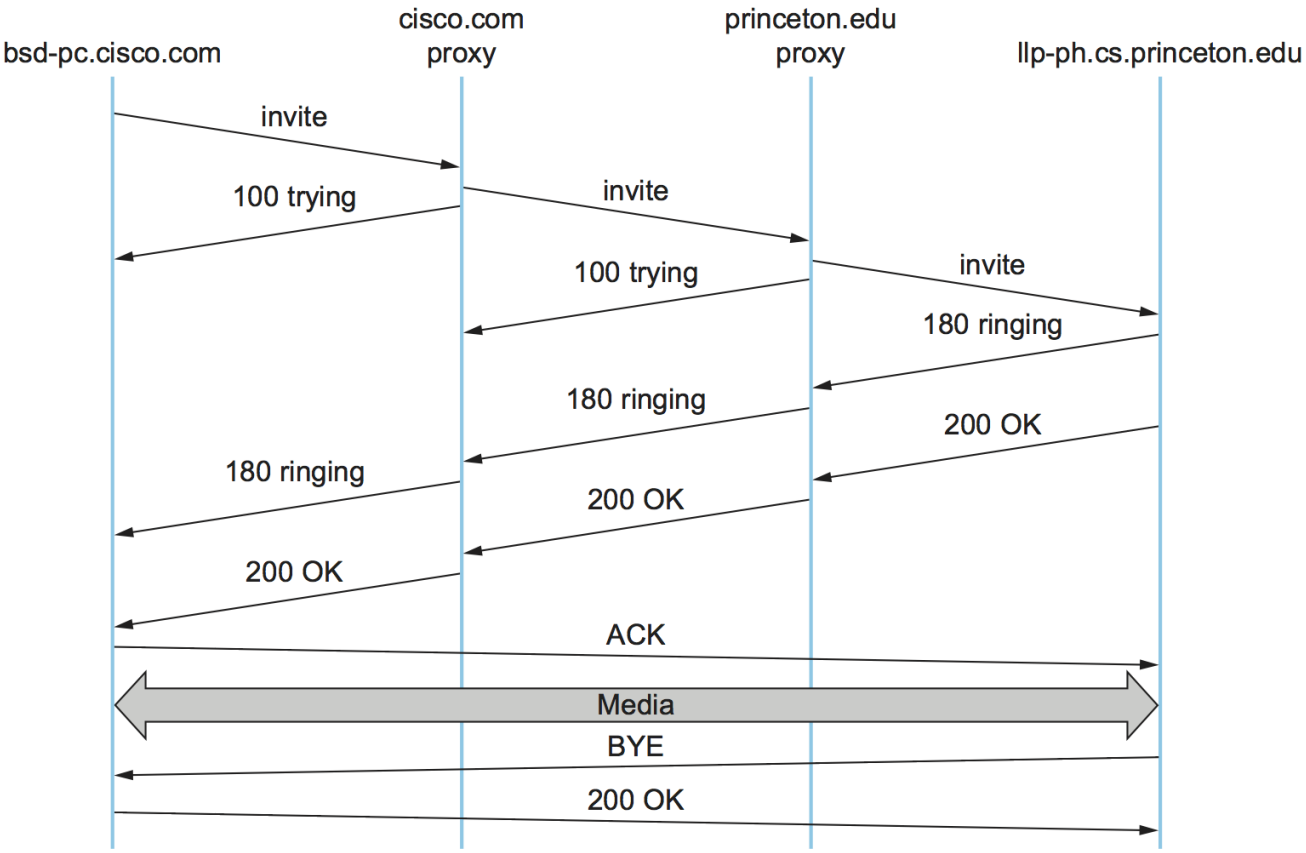


Figure 9.9.: Message flow for a basic SIP session.

The first response message in this figure is the provisional response `100 trying`, which indicates that the message was received without error by the caller's proxy. Once the `invite` is delivered to Larry's phone, it alerts Larry and responds with a `180 ringing` message. The arrival of this message at Bruce's computer is a sign that it can generate a "ringtone." Assuming Larry is willing and able to communicate with Bruce, he could pick up his phone, causing the message `200 OK` to be sent. Bruce's computer responds with an `ACK`, and media (e.g., an RTP-encapsulated audio stream) can now begin to flow between the two parties. Note that at this point the parties know each others' addresses, so the `ACK` can be sent directly, bypassing the proxies. The proxies are now no longer involved in the call. Note that the media will therefore typically take a different path through the network than the original signalling messages. Furthermore, even if one or both of the proxies were to crash at this point, the call could continue on normally. Finally, when one party wishes to end the session, it sends a `BYE` message, which elicits a `200 OK` response under normal circumstances.

There are a few details that we have glossed over. One is the negotiation of session characteristics. Perhaps Bruce would have liked to communicate using both audio and video but Larry's phone only supports audio. Thus, Larry's phone would send an SDP message in its `200 OK` describing the properties of the session that will be acceptable to Larry and the device, considering the options that were proposed in Bruce's `invite`. In this way, mutually acceptable session parameters are agreed to before the media flow starts.

The other big issue we have glossed over is that of locating the correct device for Larry. First, Bruce's computer had to send its `invite` to the `cisco.com` proxy. This could have been a configured piece of information in the computer, or it could have been learned by DHCP. Then the `cisco.com` proxy had to find the `princeton.ed` proxy. This could be done using a special sort of DNS lookup that would return the IP address of the SIP proxy for the domain. (We'll discuss how DNS can do this in the next section.) Finally, the `princeton.ed` proxy had to find a device on which Larry could be contacted. Typically, a proxy server has access to a location database that can be populated in several ways. Manual configuration is one option, but a more flexible option is to use the *registration* capabilities of SIP.

A user can register with a location service by sending a SIP `register` message to the "registrar" for his domain. This message creates a binding between an "address of record" and a "contact address." An "address of record" is likely to be a SIP URI that is the well-known address for the user (e.g., `sip:larry@princeton.edu`) and the "contact address" will be the address at which the user can currently be found (e.g., `sip:larry@llp-ph.cs.princeton.edu`). This is exactly the binding that was needed by the proxy `princeton.edu` in our example.

Note that a user may register at several locations and that multiple users may register at a single device. For example, one can imagine a group of people walking into a conference room that is equipped with an IP phone and all of them registering on it so that they can receive calls on that phone.

SIP is a very rich and flexible protocol that can support a wide range of complex calling scenarios as well as applications that have little or nothing to do with telephony. For example, SIP supports operations that enable a call to be routed to a "music-on-hold" server or a voicemail server. It is also easy to see how it could be used for applications like instant messaging, and standardization of SIP extensions for such purposes is ongoing.

H.323

The International Telecommunication Union (ITU) has also been very active in the call control area, which is not surprising given its relevance to telephony, the traditional realm of that body. Fortunately, there has been considerable coordination between the IETF and the ITU in this instance, so that the various protocols are somewhat interoperable. The major ITU recommendation for multimedia communication over packet networks is known as *H.323*, which ties together many other recommendations, including H.225 for call control. The full set of recommendations covered by H.323 runs to many hundreds of pages, and the protocol is known for its complexity, so it is only possible to give a brief overview of it here.

H.323 is popular as a protocol for Internet telephony, including video calls, and we consider that class of application here. A device that originates or terminates calls is known as an H.323 terminal; this might be a workstation running an Internet telephony application, or it might be a specially designed "appliance"—a telephone-like device with networking software and an Ethernet port, for example. H.323 terminals can talk to each other directly, but the calls are frequently mediated by a device known as a *gatekeeper*. Gatekeepers perform a number of functions such as translating among the various address formats used for phone calls and controlling how many calls can be placed at a given time to limit the bandwidth used by the H.323

applications. H.323 also includes the concept of a *gateway*, which connects the H.323 network to other types of networks. The most common use of a gateway is to connect an H.323 network to the public switched telephone network (PSTN) as illustrated in Figure 9.10. This enables a user running an H.323 application on a computer to talk to a person using a conventional phone on the public telephone network. One useful function performed by the gatekeeper is to help a terminal find a gateway, perhaps choosing among several options to find one that is relatively close to the ultimate destination of the call. This is clearly useful in a world where conventional phones greatly outnumber PC-based phones. When an H.323 terminal makes a call to an endpoint that is a conventional phone, the gateway becomes the effective endpoint for the H.323 call and is responsible for performing the appropriate translation of both signalling information and the media stream that need to be carried over the telephone network.

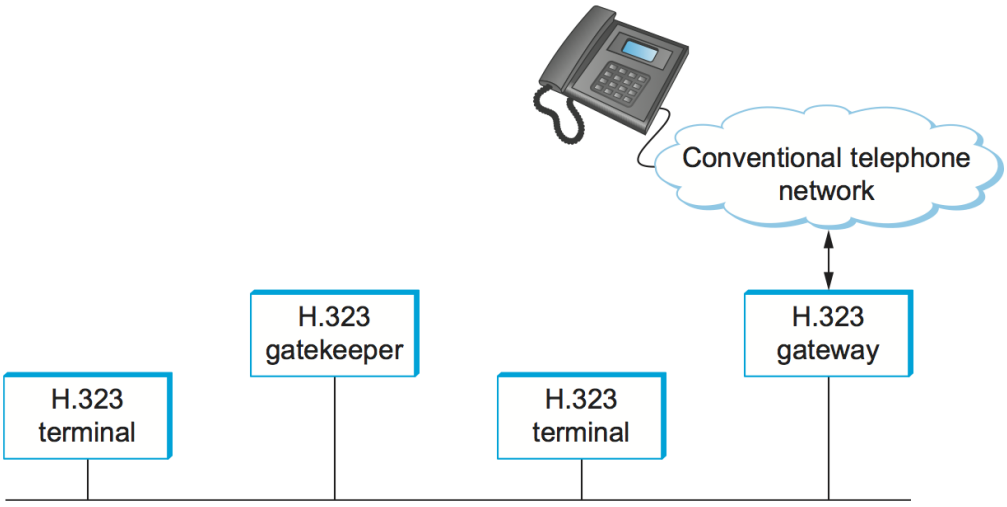


Figure 9.10.: Devices in an H.323 network.

An important part of H.323 is the H.245 protocol, which is used to negotiate the properties of the call, somewhat analogously to the use of SDP described above. H.245 messages might list a number of different audio codec standards that it can support; the far endpoint of the call would reply with a list of its own supported codecs, and the two ends could pick a coding standard that they can both live with. H.245 can also be used to signal the UDP port numbers that will be used by RTP and Real-Time Control Protocol (RTCP) for the media stream (or streams—a call might include both audio and video, for example) for this call. Once this is accomplished, the call can proceed, with RTP being used to transport the media streams and RTCP carrying the relevant control information.

9.2.2 Resource Allocation for Multimedia Applications

As we have just seen, session control protocols like SIP and H.323 can be used to initiate and control communication in multimedia applications, while RTP provides transport-level functions for the data streams of the applications. A final piece of the puzzle in getting multimedia applications to work is making sure that suitable resources are allocated inside the network to ensure that the quality of service needs of the application are met. We presented a number of methods for resource allocation in an earlier chapter. The motivation for developing these technologies was largely for the support of multimedia applications. So how do applications take advantage of the underlying resource allocation capabilities of the network?

It is worth noting that many multimedia applications run successfully over "best-effort" networks, such as

the public Internet. The wide array of commercial VOIP services (such as Skype) are a testimony to the fact that you only have to worry about resource allocation when resources are not abundant—and in many parts of today's Internet, resource abundance is the norm.

A protocol like RTCP can help applications in best-effort networks, by giving the application detailed information about the quality of service that is being delivered by the network. Recall that RTCP carries information about the loss rate and delay characteristics between participants in a multimedia application. An application can use this information to change its coding scheme—changing to a lower bitrate codec, for example, when bandwidth is scarce. Note that, while it might be tempting to change to a codec that sends additional, redundant information when loss rates are high, this is frowned upon; it is analogous to *increasing* the window size of TCP in the presence of loss, the exact opposite of what is required to avoid congestion collapse.

As discussed in an earlier chapter, Differentiated Services (DiffServ) can be used to provide fairly basic and scalable resource allocation to applications. A multimedia application can set the differentiated services code point (DSCP) in the IP header of the packets that it generates in an effort to ensure that both the media and control packets receive appropriate quality of service. For example, it is common to mark voice media packets as "EF" (expedited forwarding) to cause them to be placed in a low-latency or priority queue in routers along the path, while the call signalling (e.g., SIP) packets are often marked with some sort of "AF" (assured forwarding) to enable them to be queued separately from best-effort traffic and thus reduce their risk of loss.

Of course, it only makes sense to mark the packets inside the sending host or appliance if network devices such as routers pay attention to the DSCP. In general, routers in the public Internet ignore the DSCP, providing best-effort service to all packets. However, enterprise or corporate networks have the ability to use DiffServ for their internal multimedia traffic, and frequently do so. Also, even residential users of the Internet can often improve the quality of VOIP or other multimedia applications just by using DiffServ on the outbound direction of their Internet connections, as illustrated in Figure 9.11. This is effective because of the asymmetry of many broadband Internet connections: If the outbound link is substantially slower (i.e., more resource constrained) than the inbound, then resource allocation using DiffServ on that link may be enough to make all the difference in quality for latency- and loss-sensitive applications.

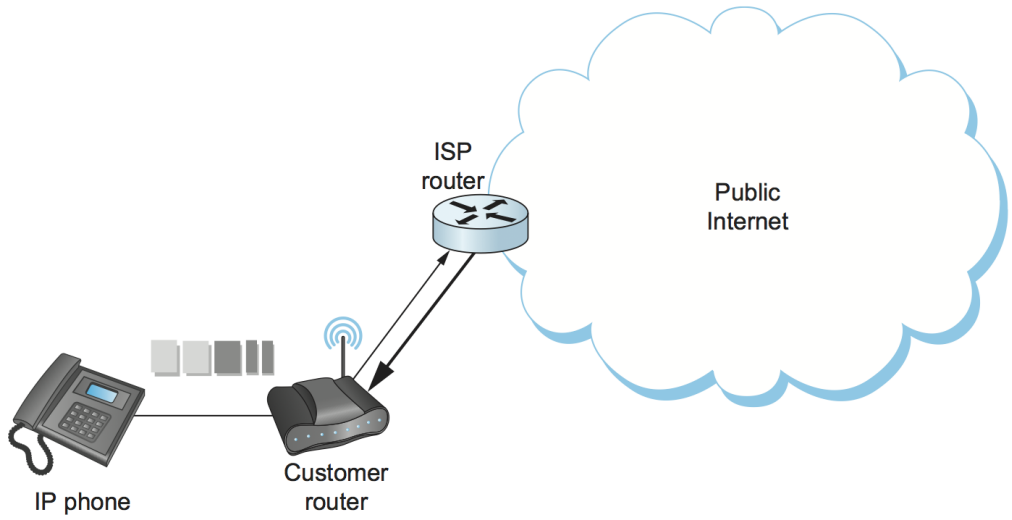


Figure 9.11.: Differentiated Services applied to a VOIP application. DiffServ queueing is applied only on the upstream link from customer router to ISP.

While DiffServ is appealing for its simplicity, it is clear that it cannot meet the needs of applications under all conditions. For example, suppose the upstream bandwidth in Figure 9.11 is only 100 kbps, and the customer attempts to place two VOIP calls, each with a 64-kbps codec. Clearly the upstream link is now more than 100% loaded, which will lead to large queueing delays and lost packets. No amount of clever queueing in the customer's router can fix that.

The characteristics of many multimedia applications are such that, rather than try to squeeze too many calls into a too-narrow pipe, it would be better to block one call while allowing another to proceed. That is, it is better to have one person carrying on a conversation successfully while another hears a busy signal than to have both callers experiencing unacceptable audio quality at the same time. We sometimes refer to such applications as having a *steep utility curve*, meaning that the utility (usefulness) of the application drops rapidly as the quality of service provided by the network degrades. Multimedia applications often have this property, whereas many traditional applications do not. Email, for example, continues to work quite well even if delays run into the hours.

Applications with steep utility curves are often well suited to some form of admission control. If you cannot be sure that sufficient resources will always be available to support the offered load of the applications, then admission control provides a way to say "no" to some applications while allowing others to get the resources they need.

We saw one way to do admission control using RSVP in an earlier chapter, and we will return to that shortly, but multimedia applications that use session control protocols provide some other admission control options. The key point to observe here is that session control protocols like SIP or H.323 often involve some sort of message exchange between an endpoint and another entity (SIP proxy or H.323 gatekeeper) at the beginning of a call or session. This can provide a handy means to say "no" to a new call for which sufficient resources are not available.

As an example, consider the network in Figure 9.12. Suppose the wide area link from the branch office to the head office has enough bandwidth to accommodate three VOIP calls simultaneously using 64-kbps codecs. Each phone already needs to communicate with the local SIP proxy or H.323 gatekeeper when it begins to place a call, so it is easy enough for the proxy/gatekeeper to send back a message that tells the IP phone to play a busy signal if that link is already fully loaded. The proxy or gatekeeper can even deal with the possibility that a particular IP phone might be making multiple calls at the same time and that different codec speeds might be used. However, this scheme will work only if no other device can overload the link without first talking to the gatekeeper or proxy. DiffServ queueing can be used to ensure that, for example, a PC engaged in a file transfer doesn't interfere with the VOIP calls. But, suppose some VOIP application that doesn't first talk to the gatekeeper or proxy is enabled in the remote office. Such an application, if it can get its packets marked appropriately and in the same queue as the existing VOIP traffic, can clearly drive the link to the point of overload with no feedback from the proxy or gatekeeper.

Another problem with the approach just described is that it depends on the gatekeeper or proxy having knowledge of the path that each application will use. In the simple topology of Figure 9.12 this isn't a big issue, but in more complex networks it can quickly become unmanageable. We only need to imagine the case where the remote office has two different connections to the outside world to see that we are asking the proxy or gatekeeper to understand not just SIP or H.323 but also routing, link failures, and current network conditions. This can quickly become unmanageable.

We refer to the sort of admission control just described as *off-path*, in the sense that the device making admission control decisions does not sit on the data path where resources need to be allocated. The obvious alternative is *on-path* admission control, and the standard example of a protocol that does on-path admission control in IP networks is the Resource Reservation Protocol (RSVP). We saw in an earlier chapter how

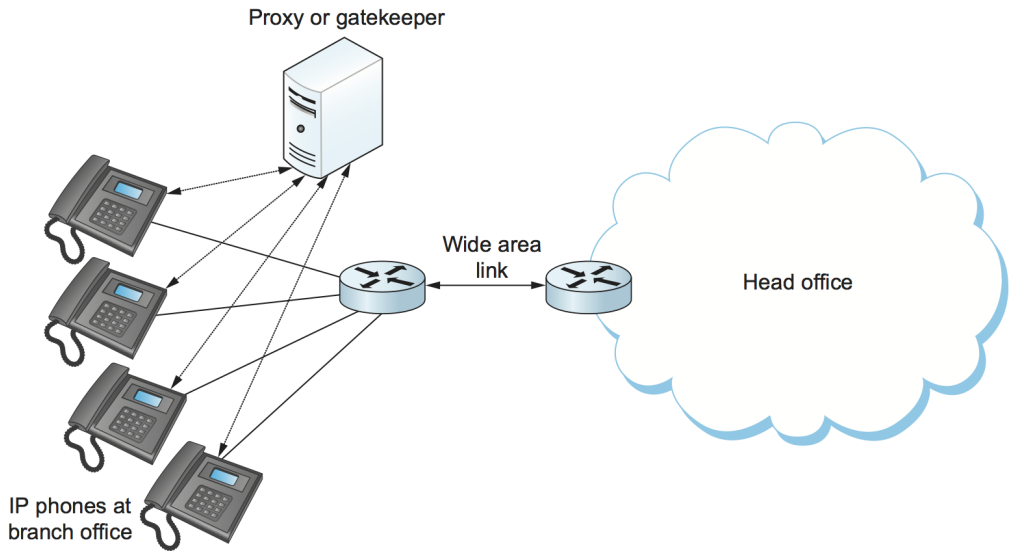


Figure 9.12.: Admission control using session control protocol.

RSVP can be used to ensure that sufficient resources are allocated along a path, and it is straightforward to use RSVP in applications like those described in this section. The one detail that still needs to be filled in is how the admission control protocol interacts with the session control protocol.

Coordinating the actions of an admission control (or resource reservation) protocol and a session control protocol is not rocket science, but it does require some attention to details. As an example, consider a simple telephone call between two parties. Before you can make a reservation, you need to know how much bandwidth the call is going to use, which means you need to know what codecs are to be used. That implies you need to do some of the session control first, to exchange information about the codecs supported by the two phones. However, you can't do *all* the session control first, because you wouldn't want the phone to ring before the admission control decision had been made, in case admission control failed. Figure 9.13 illustrates this situation where SIP is used for session control and RSVP is used to make the admission control decision (successfully in this case).

The main thing to notice here is the interleaving of session control and resource allocation tasks. Solid lines represent SIP messages, dashed lines represent RSVP messages. Note that SIP messages are transmitted direction from phone to phone in this example (i.e., we have not shown any SIP proxies), whereas the RSVP messages are also processed by the routers in the middle as the check for sufficient resources to admit the call.

We begin with an initial exchange of codec information in the first two SIP messages (recall that SDP is used to list available codecs, among other things). `PRACK` is a "provisional acknowledgment." Once these messages have been exchanged, RSVP `PATH` messages, which contain a description of the amount of resources that will be required, can be sent as the first step in reserving resources in both directions of the call. Next, `RESV` messages can be sent back to actually reserve the resources. Once a `RESV` is received by the initiating phone, it can send an updated SDP message reporting the fact that resources have been reserved in one direction. When the called phone has received both that message and the `RESV` from the other phone, it can start to ring and tell the other phone that resources are now reserved in both directions (with the SDP message) and also notify the calling phone that it is ringing. From here on, normal SIP signalling and media flow, similar to that shown in Figure 9.9, proceeds.

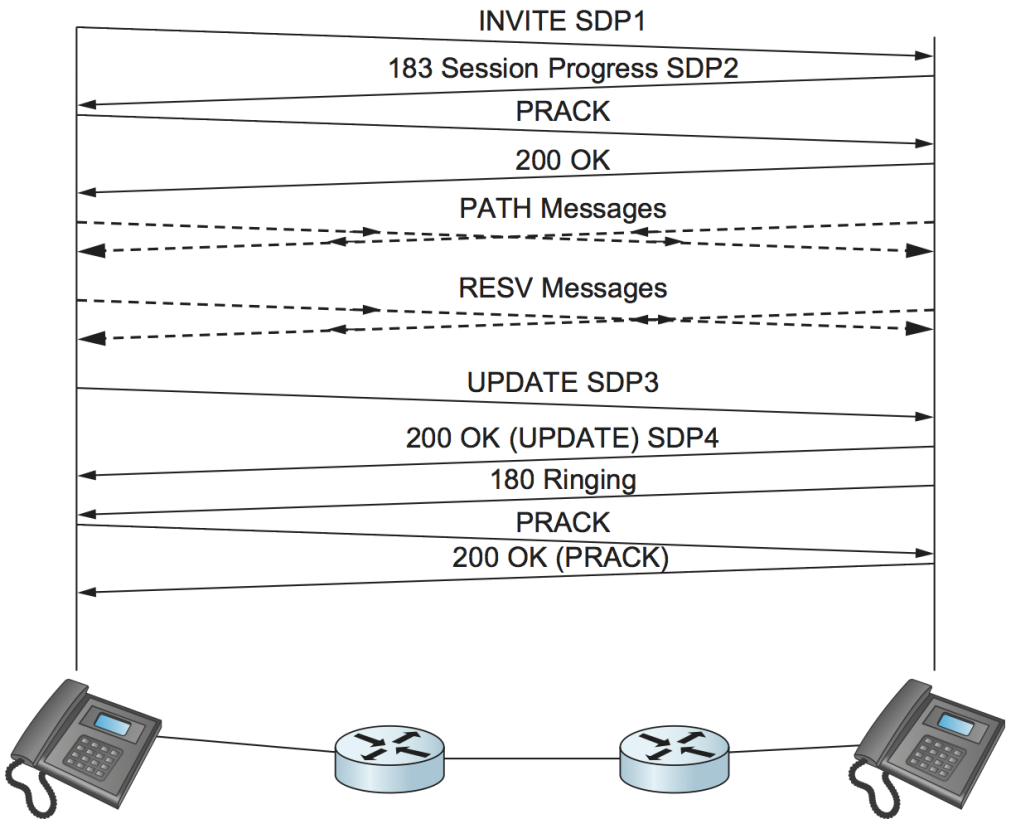


Figure 9.13.: Coordination of SIP signalling and resource reservation.

Again we see how building applications requires us to understand the interaction between different building blocks (SIP and RSVP, in this case). The designers of SIP actually made some changes to the protocol to enable this interleaving of functions between protocols with different jobs, hence our repeated emphasis in this book on focusing on complete systems rather than just looking at one layer or component in isolation from the other parts of the system.

9.3 Infrastructure Applications

There are some protocols that are essential to the smooth running of the Internet but that don't fit neatly into the strictly layered model. One of these is the Domain Name System (DNS)—not an application that users normally invoke directly, but rather a service that almost all other applications depend upon. This is because the name service is used to translate host names into host addresses; the existence of such an application allows the users of other applications to refer to remote hosts by name rather than by address. In other words, a name service is usually used by other applications, rather than by humans.

A second critical function is network management, which although not so familiar to the average user, is performed most often by the people that operate the network on behalf of users. Network management is widely considered one of the hard problems of networking and continues to be the focus of much innovation. We'll look at some of the issues and approaches to the problem below.

9.3.1 Name Service (DNS)

In most of this book, we have been using addresses to identify hosts. While perfectly suited for processing by routers, addresses are not exactly user friendly. It is for this reason that a unique *name* is also typically assigned to each host in a network. Already in this section we have seen application protocols like HTTP using names such as `www.princeton.edu`. We now describe how a naming service can be developed to map user-friendly names into router-friendly addresses. Name services are sometimes called *middleware* because they fill a gap between applications and the underlying network.

Host names differ from host addresses in two important ways. First, they are usually of variable length and mnemonic, thereby making them easier for humans to remember. (In contrast, fixed-length numeric addresses are easier for routers to process.) Second, names typically contain no information that helps the network locate (route packets toward) the host. Addresses, in contrast, sometimes have routing information embedded in them; *flat* addresses (those not divisible into component parts) are the exception.

Before getting into the details of how hosts are named in a network, we first introduce some basic terminology. First, a *name space* defines the set of possible names. A name space can be either *flat* (names are not divisible into components) or *hierarchical* (Unix file names are an obvious example). Second, the naming system maintains a collection of *bindings* of names to values. The value can be anything we want the naming system to return when presented with a name; in many cases, it is an address. Finally, a *resolution mechanism* is a procedure that, when invoked with a name, returns the corresponding value. A *name server* is a specific implementation of a resolution mechanism that is available on a network and that can be queried by sending it a message.

Because of its large size, the Internet has a particularly well-developed naming system in place—the Domain Name System (DNS). We therefore use DNS as a framework for discussing the problem of naming hosts. Note that the Internet did not always use DNS. Early in its history, when there were only a few hundred hosts on the Internet, a central authority called the *Network Information Center* (NIC) maintained a flat

table of name-to-address bindings; this table was called `HOSTS.TXT`.[1] Whenever a site wanted to add a new host to the Internet, the site administrator sent email to the NIC giving the new host's name/address pair. This information was manually entered into the table, the modified table was mailed out to the various sites every few days, and the system administrator at each site installed the table on every host at the site. Name resolution was then simply implemented by a procedure that looked up a host's name in the local copy of the table and returned the corresponding address.

It should come as no surprise that the approach to naming did not work well as the number of hosts in the Internet started to grow. Therefore, in the mid-1980s, the Domain Naming System was put into place. DNS employs a hierarchical namespace rather than a flat name space, and the "table" of bindings that implements this name space is partitioned into disjoint pieces and distributed throughout the Internet. These subtables are made available in name servers that can be queried over the network.

What happens in the Internet is that a user presents a host name to an application program (possibly embedded in a compound name such as an email address or URL), and this program engages the naming system to translate this name into a host address. The application then opens a connection to this host by presenting some transport protocol (e.g., TCP) with the host's IP address. This situation is illustrated (in the case of sending email) in Figure 9.14. While this picture makes the name resolution task look simple enough, there is a bit more to it, as we shall see.

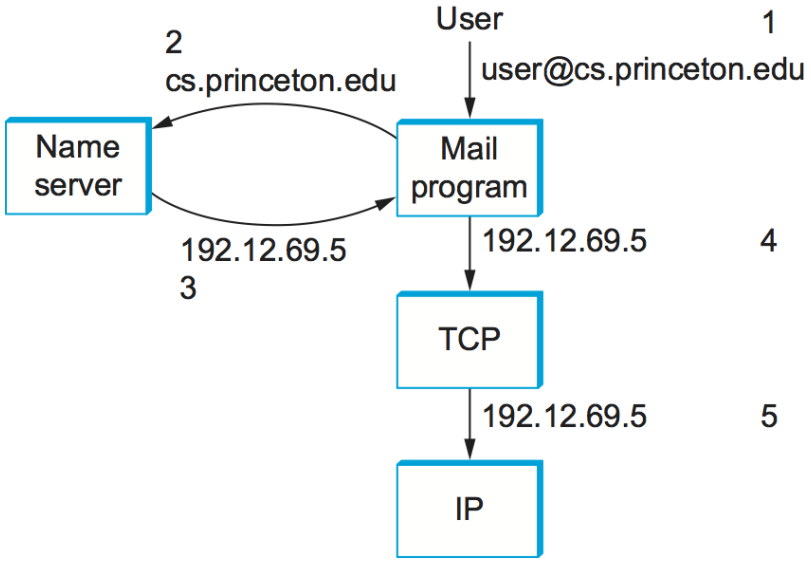


Figure 9.14.: Names translated into addresses, where the numbers 1 to 5 show the sequence of steps in the process.

Domain Hierarchy

DNS implements a hierarchical name space for Internet objects. Unlike Unix file names, which are processed from left to right with the naming components separated with slashes, DNS names are processed from right to left and use periods as the separator. (Although they are processed from right to left, humans still

[1] Believe it or not, there was also a paper book (like a phone book) published periodically that listed all the machines connected to the Internet and all the people that had an Internet email account.

read domain names from left to right.) An example domain name for a host is `cicada.cs.princeton.edu`. Notice that we said domain names are used to name Internet "objects." What we mean by this is that DNS is not strictly used to map host names into host addresses. It is more accurate to say that DNS maps domain names into values. For the time being, we assume that these values are IP addresses; we will come back to this issue later in this section.

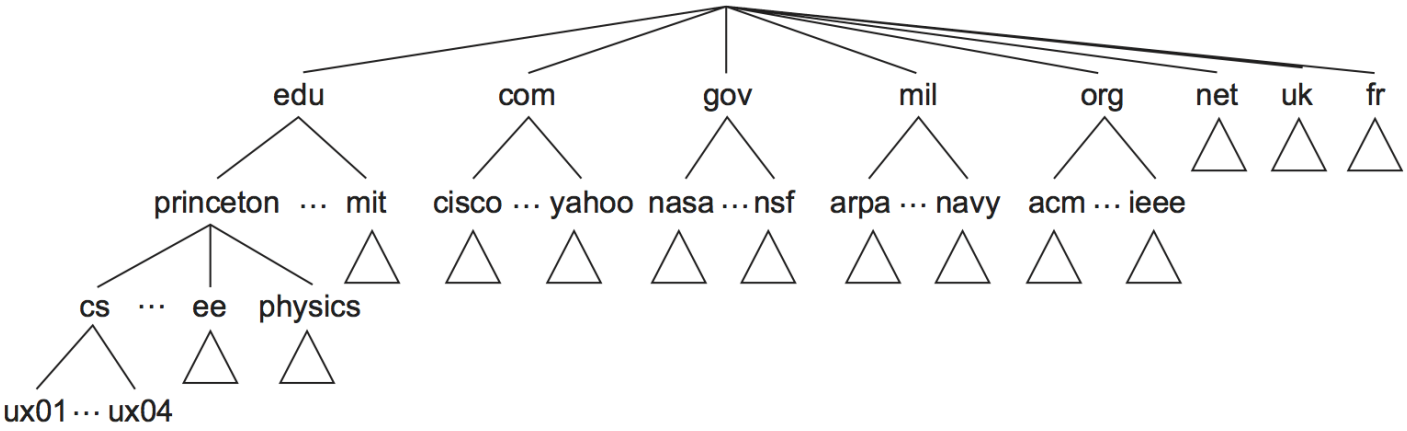


Figure 9.15.: Example of a domain hierarchy.

Like the Unix file hierarchy, the DNS hierarchy can be visualized as a tree, where each node in the tree corresponds to a domain, and the leaves in the tree correspond to the hosts being named. Figure 9.15 gives an example of a domain hierarchy. Note that we should not assign any semantics to the term *domain* other than that it is simply a context in which additional names can be defined.[2]

There was actually a substantial amount of discussion that took place when the domain name hierarchy was first being developed as to what conventions would govern the names that were to be handed out near the top of the hierarchy. Without going into that discussion in any detail, notice that the hierarchy is not very wide at the first level. There are domains for each country, plus the "big six" domains: `.edu`, `.com`, `.gov`, `.mil`, `.org`, and `.net`. These six domains were all originally based in the United States (where the Internet and DNS were invented); for example, only U.S.-accredited educational institutions can register an `.edu` domain name. In recent years, the number of top-level domains has been expanded, partly to deal with the high demand for `.com` domains names. The newer top-level domains include `.biz`, `.coop`, and `.info`. There are now over 1200 top-level domains.

Name Servers

The complete domain name hierarchy exists only in the abstract. We now turn our attention to the question of how this hierarchy is actually implemented. The first step is to partition the hierarchy into subtrees called *zones*. Figure 9.16 shows how the hierarchy given in Figure 9.15 might be divided into zones. Each zone can be thought of as corresponding to some administrative authority that is responsible for that portion of the hierarchy. For example, the top level of the hierarchy forms a zone that is managed by the Internet Corporation for Assigned Names and Numbers (ICANN). Below this is a zone that corresponds to Princeton University. Within this zone, some departments do not want the responsibility of managing the hierarchy

[2] Confusingly, the word *domain* is also used in Internet routing, where it means something different than it does in DNS, being roughly equivalent to the term *autonomous system*.

(and so they remain in the university-level zone), while others, like the Department of Computer Science, manage their own department-level zone.

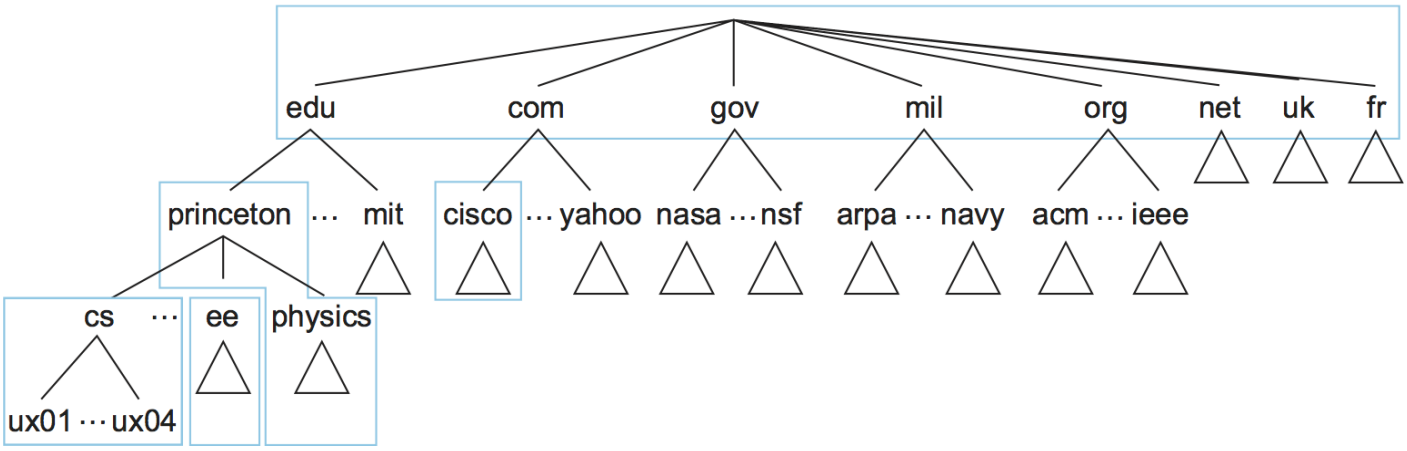


Figure 9.16.: Domain hierarchy partitioned into zones.

The relevance of a zone is that it corresponds to the fundamental unit of implementation in DNS—the name server. Specifically, the information contained in each zone is implemented in two or more name servers. Each name server, in turn, is a program that can be accessed over the Internet. Clients send queries to name servers, and name servers respond with the requested information. Sometimes the response contains the final answer that the client wants, and sometimes the response contains a pointer to another server that the client should query next. Thus, from an implementation perspective, it is more accurate to think of DNS as being represented by a hierarchy of name servers rather than by a hierarchy of domains, as illustrated in Figure 9.17.

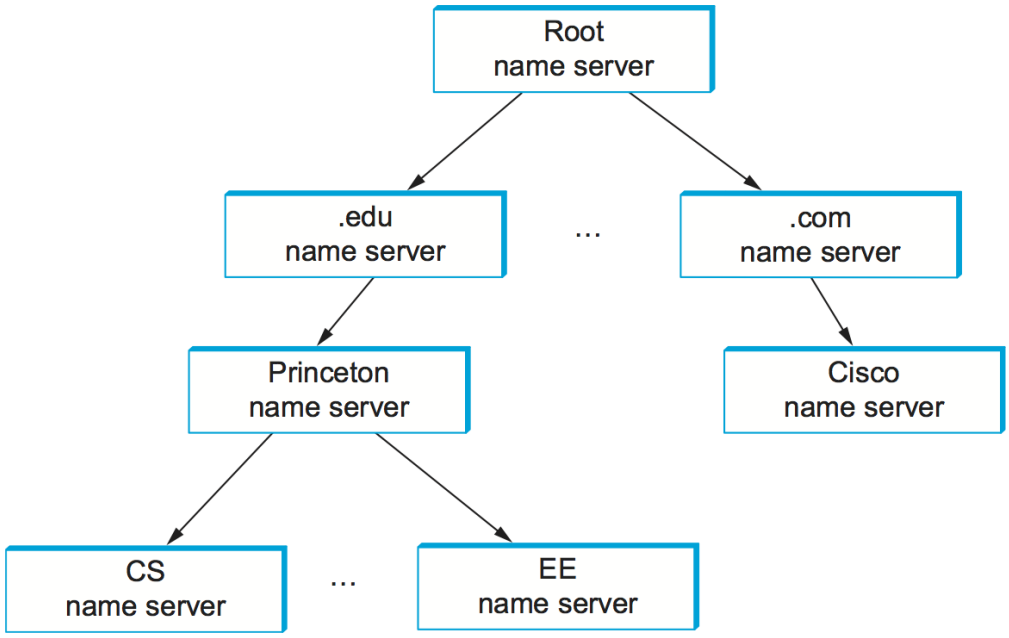


Figure 9.17.: Hierarchy of name servers.

Note that each zone is implemented in two or more name servers for the sake of redundancy; that is, the

information is still available even if one name server fails. On the flip side, a given name server is free to implement more than one zone.

Each name server implements the zone information as a collection of *resource records*. In essence, a resource record is a name-to-value binding or, more specifically, a 5-tuple that contains the following fields:

```
(Name, Value, Type, Class, TTL)
```

The `Name` and `Value` fields are exactly what you would expect, while the `Type` field specifies how the `Value` should be interpreted. For example, indicates that the `Value` is an IP address. Thus, `A` records implement the name-to-address mapping we have been assuming. Other record types include:

- `NS`—The `Value` field gives the domain name for a host that is running a name server that knows how to resolve names within the specified domain.
- `CNAME`—The `Value` field gives the canonical name for a particular host; it is used to define aliases.
- `MX`—The `Value` field gives the domain name for a host that is running a mail server that accepts messages for the specified domain.

The `Class` field was included to allow entities other than the NIC to define useful record types. To date, the only widely used `Class` is the one used by the Internet; it is denoted `IN`. Finally, the time-to-live (`TTL`) field shows how long this resource record is valid. It is used by servers that cache resource records from other servers; when the `TTL` expires, the server must evict the record from its cache.

To better understand how resource records represent the information in the domain hierarchy, consider the following examples drawn from the domain hierarchy given in Figure 9.15. To simplify the example, we ignore the `TTL` field and we give the relevant information for only one of the name servers that implement each zone.

First, a root name server contains an `NS` record for each top-level domain (TLD) name server. This identifies a server that can resolve queries for this part of the DNS hierarchy (`.edu` and `.com`in this example). It also has `A` records that translates these names into the corresponding IP addresses. Taken together, these two records effectively implement a pointer from the root name server to one of the TLD servers.

```
(edu, a3.nstld.com, NS, IN)
(a3.nstld.com, 192.5.6.32, A, IN)
(com, a.gtld-servers.net, NS, IN)
(a.gtld-servers.net, 192.5.6.30, A, IN)
...
```

Moving our way down the hierarchy by one level, the server has records for domains like this:

```
(princeton.edu, dns.princeton.edu, NS, IN)
(dns.princeton.edu, 128.112.129.15, A, IN)
...
```

In this case, we get an `NS` record and an `A` record for the name server that is responsible for the `princeton.edu` part of the hierarchy. That server might be able to directly resolve some queries (e.g., for`email.princeton.edu`) while it would redirect others to a server at yet another layer in the hierarchy (e.g., for a query about `penguins.cs.princeton.edu`).

```
(email.princeton.edu, 128.112.198.35, A, IN)
(penguins.cs.princeton.edu, dns1.cs.princeton.edu, NS, IN)
(dns1.cs.princeton.edu, 128.112.136.10, A, IN)
...
```

Finally, a third-level name server, such as the one managed by domain `cs.princeton.edu`, contains `A` records for all of its hosts. It might also define a set of aliases (`CNAME` records) for each of those hosts. Aliases are sometimes just convenient (e.g., shorter) names for machines, but they can also be used to provide a level of indirection. For example,`www.cs.princeton.edu` is an alias for the host named `coreweb.cs.princeton.edu`.This allows the site's web server to move to another machine without affecting remote users; they simply continue to use the alias without regard for what machine currently runs the domain's web server. The mail exchange (`MX`) records serve the same purpose for the email application—they allow an administrator to change which host receives mail on behalf of the domain without having to change everyone's email address.

```
(penguins.cs.princeton.edu, 128.112.155.166, A, IN)
(www.cs.princeton.edu, coreweb.cs.princeton.edu, CNAME, IN)
coreweb.cs.princeton.edu, 128.112.136.35, A, IN)
(cs.princeton.edu, mail.cs.princeton.edu, MX, IN)
(mail.cs.princeton.edu, 128.112.136.72, A, IN)
...
```

Note that, although resource records can be defined for virtually any type of object, DNS is typically used to name hosts (including servers) and sites. It is not used to name individual people or other objects like files or directories; other naming systems are typically used to identify such objects. For example, X.500 is an ISO naming system designed to make it easier to identify people. It allows you to name a person by giving a set of attributes: name, title, phone number, postal address, and so on. X.500 proved too cumbersome—and, in some sense, was usurped by powerful search engines now available on the Web—but it did eventually evolve into the Lightweight Directory Access Protocol (LDAP). LDAP is a subset of X.500 originally designed as a PC front end to X.500. Today, widely used, mostly at the enterprise level, as a system for learning information about users.

Name Resolution

Given a hierarchy of name servers, we now consider the issue of how a client engages these servers to resolve a domain name. To illustrate the basic idea, suppose the client wants to resolve the name `penguins.cs.princeton.edu` relative to the set of servers given in the previous subsection. The client could first send a query containing this name to one of the root servers (as we'll see below, this rarely happens in practice but will suffice to illustrate the basic operation for now). The root server, unable to match the entire name, returns the best match it has—the `NS` record for `edu` which points to the TLD server `a3.nstld.com`. The server also returns all records that are related to this record, in this case, the `A` record for `a3.nstld.com`. The client, having not received the answer it was after, next sends the same query to the name server at IP host `192.5.6.32`. This server also cannot match the whole name and so returns the `NS` and corresponding `A` records for the `princeton.edu` domain. Once again, the client sends the same query as before to the server at IP host `128.112.129.15`, and this time gets back the `NS` record and corresponding `A` record for the `cs.princeton.edu` domain. This time, the server that can fully resolve the query has been reached. A final query to the server at `128.112.136.10` yields the `A` record for `penguins.cs.princeton.edu`, and the client learns that the corresponding IP address is `128.112.155.166`.

This example still leaves a couple of questions about the resolution process unanswered. The first question is how did the client locate the root server in the first place, or, put another way, how do you resolve the name of the server that knows how to resolve names? This is a fundamental problem in any naming system, and the answer is that the system has to be bootstrapped in some way. In this case, the name-to-address mapping for one or more root servers is well known; that is, it is published through some means outside the naming system itself.

In practice, however, not all clients know about the root servers. Instead, the client program running on each Internet host is initialized with the address of a *local* name server. For example, all the hosts in the Department of Computer Science at Princeton know about the server on `dns1.cs.princeton.edu`. This local name server, in turn, has resource records for one or more of the root servers, for example:

```
('root', a.root-servers.net, NS, IN)
(a.root-servers.net, 198.41.0.4, A, IN)
```

Thus, resolving a name actually involves a client querying the local server, which in turn acts as a client that queries the remote servers on the original client's behalf. This results in the client/server interactions illustrated in Figure 9.18. One advantage of this model is that all the hosts in the Internet do not have to be kept up-to-date on where the current root servers are located; only the servers have to know about the root. A second advantage is that the local server gets to see the answers that come back from queries that are posted by all the local clients. The local server *caches* these responses and is sometimes able to resolve future queries without having to go out over the network. The `TTL` field in the resource records returned by remote servers indicates how long each record can be safely cached. This caching mechanism can be used further up the hierarchy as well, reducing the load on the root and TLD servers.

The second question is how the system works when a user submits a partial name (e.g., `penguins`) rather than a complete domain name (e.g., `penguins.cs.princeton.edu`). The answer is that the client program is configured with the local domain in which the host resides (e.g., `cs.princeton.edu`), and it appends this string to any simple names before sending out a query.

Key Takeaway

Just to make sure we are clear, we have now seen three different levels of identifiers—domain names, IP addresses, and physical network addresses—and the mapping of identifiers at one level into identifiers at another level happens at different points in the network architecture. First, users specify domain names when interacting with the application. Second, the application engages DNS to translate this name into an IP address; it is the IP address that is placed in each datagram, not the domain name. (As an aside, this translation process involves IP datagrams being sent over the Internet, but these datagrams are addressed to a host that runs a name server, not to the ultimate destination.) Third, IP does forwarding at each router, which often means that it maps one IP address into another; that is, it maps the ultimate destination's address into the address for the next hop router. Finally, IP engages the Address Resolution Protocol (ARP) to translate the next hop IP address into the physical address for that machine; the next hop might be the ultimate destination or it might be an intermediate router. Frames sent over the physical network have these physical addresses in their headers. *[Next]*

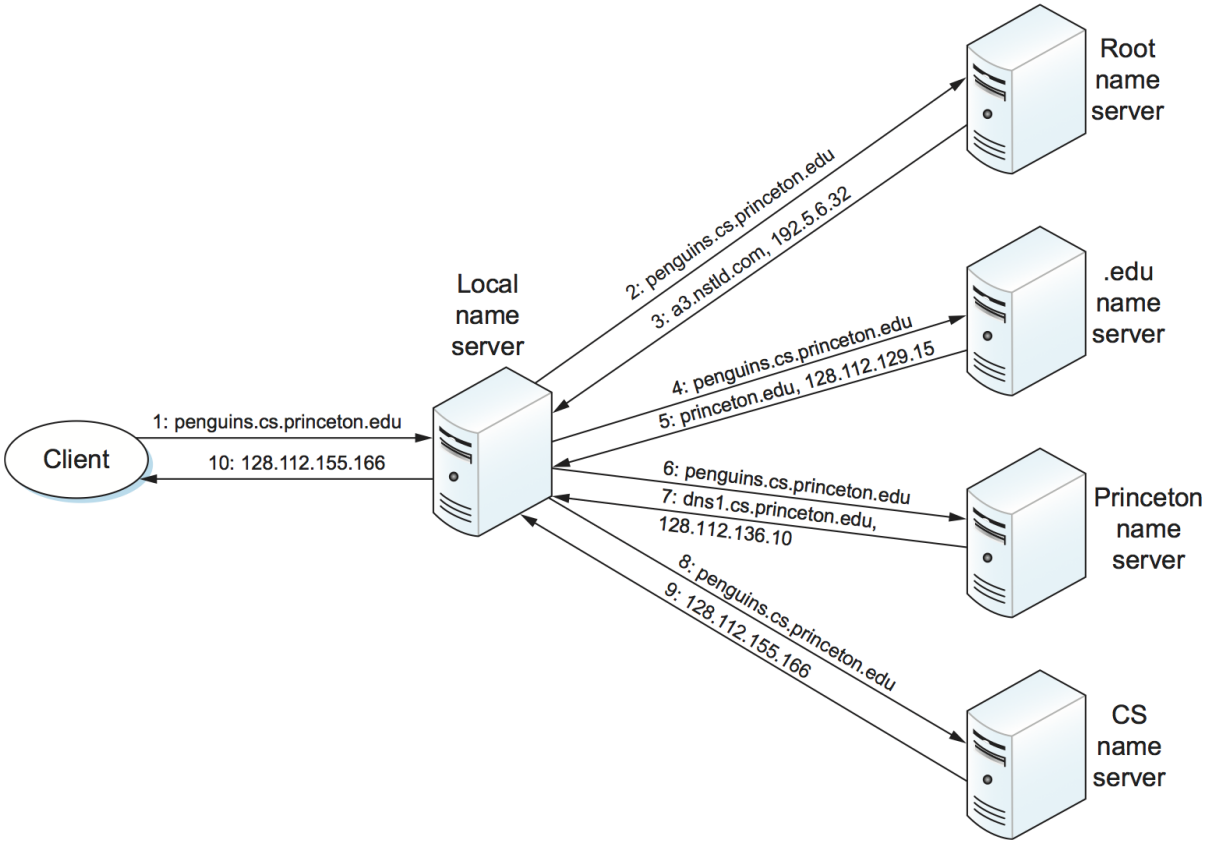


Figure 9.18.: Name resolution in practice, where the numbers 1 to 10 show the sequence of steps in the process.

9.3.2 Network Management (SNMP, OpenConfig)

A network is a complex system, both in terms of the number of nodes that are involved and in terms of the suite of protocols that can be running on any one node. Even if you restrict yourself to worrying about the nodes within a single administrative domain, such as a campus, there might be dozens of routers and hundreds—or even thousands—of hosts to keep track of. If you think about all the state that is maintained and manipulated on any one of those nodes—address translation tables, routing tables, TCP connection state, and so on—then it is easy to become overwhelmed by the prospect of having to manage all of this information.

It is easy to imagine wanting to know about the state of various protocols on different nodes. For example, you might want to monitor the number of IP datagram reassemblies that have been aborted, so as to determine if the timeout that garbage collects partially assembled datagrams needs to be adjusted. As another example, you might want to keep track of the load on various nodes (i.e., the number of packets sent or received) so as to determine if new routers or links need to be added to the network. Of course, you also have to be on the watch for evidence of faulty hardware and misbehaving software.

What we have just described is the problem of network management, an issue that pervades the entire network architecture. Since the nodes we want to keep track of are distributed, our only real option is to use the network to manage the network. This means we need a protocol that allows us to read and write various pieces of state information on different network nodes. The following describes two approaches.

SNMP

A widely used protocol for network management is SNMP (*Simple Network Management Protocol*). SNMP is essentially a specialized request/reply protocol that supports two kinds of request messages: `GET` and `SET`. The former is used to retrieve a piece of state from some node, and the latter is used to store a new piece of state in some node. (SNMP also supports a third operation, `GET-NEXT`, which we explain below.) The following discussion focuses on the `GET` operation, since it is the one most frequently used.

SNMP is used in the obvious way. An operator interacts with a client program that displays information about the network. This client program usually has a graphical interface. You can think of this interface as playing the same role as a web browser. Whenever the operator selects a certain piece of information that he or she wants to see, the client program uses SNMP to request that information from the node in question. (SNMP runs on top of UDP.) An SNMP server running on that node receives the request, locates the appropriate piece of information, and returns it to the client program, which then displays it to the user.

There is only one complication to this otherwise simple scenario: Exactly how does the client indicate which piece of information it wants to retrieve, and, likewise, how does the server know which variable in memory to read to satisfy the request? The answer is that SNMP depends on a companion specification called the *management information base* (MIB). The MIB defines the specific pieces of information—the MIB *variables*—that you can retrieve from a network node.

The current version of MIB, called MIB-II, organizes variables into different *groups*. You will recognize that most of the groups correspond to one of the protocols described in this book, and nearly all of the variables defined for each group should look familiar. For example:

- System—General parameters of the system (node) as a whole, including where the node is located, how long it has been up, and the system's name

- Interfaces—Information about all the network interfaces (adaptors) attached to this node, such as the physical address of each interface and how many packets have been sent and received on each interface
- Address translation—Information about the Address Resolution Protocol, and in particular, the contents of its address translation table
- IP—Variables related to IP, including its routing table, how many datagrams it has successfully forwarded, and statistics about datagram reassembly; includes counts of how many times IP drops a datagram for one reason or another
- TCP—Information about TCP connections, such as the number of passive and active opens, the number of resets, the number of timeouts, default timeout settings, and so on; per-connection information persists only as long as the connection exists
- UDP—Information about UDP traffic, including the total number of UDP datagrams that have been sent and received.

There are also groups for Internet Control Message Protocol (ICMP) and SNMP itself.

Returning to the issue of the client stating exactly what information it wants to retrieve from a node, having a list of MIB variables is only half the battle. Two problems remain. First, we need a precise syntax for the client to use to state which of the MIB variables it wants to fetch. Second, we need a precise representation for the values returned by the server. Both problems are addressed using Abstract Syntax Notation One (ASN.1).

Consider the second problem first. As we already saw in a previous chapter, ASN.1/Basic Encoding Rules (BER) defines a representation for different data types, such as integers. The MIB defines the type of each variable, and then it uses ASN.1/BER to encode the value contained in this variable as it is transmitted over the network. As far as the first problem is concerned, ASN.1 also defines an object identification scheme. The MIB uses this identification system to assign a globally unique identifier to each MIB variable. These identifiers are given in a "dot" notation, not unlike domain names. For example, 1.3.6.1.2.1.4.3 is the unique ASN.1 identifier for the IP-related MIB variable `ipInReceives`; this variable counts the number of IP datagrams that have been received by this node. In this example, the 1.3.6.1.2.1 prefix identifies the MIB database (remember, ASN.1 object IDs are for all possible objects in the world), the 4 corresponds to the IP group, and the final 3 denotes the third variable in this group.

Thus, network management works as follows. The SNMP client puts the ASN.1 identifier for the MIB variable it wants to get into the request message, and it sends this message to the server. The server then maps this identifier into a local variable (i.e., into a memory location where the value for this variable is stored), retrieves the current value held in this variable, and uses ASN.1/BER to encode the value it sends back to the client.

There is one final detail. Many of the MIB variables are either tables or structures. Such compound variables explain the reason for the SNMP `GET-NEXT` operation. This operation, when applied to a particular variable ID, returns the value of that variable plus the ID of the next variable, for example, the next item in the table or the next field in the structure. This aids the client in "walking through" the elements of a table or structure.

OpenConfig

SNMP is still widely used and has historically been "the" management protocol for switches and routers, but there has recently been growing attention paid to more flexible and powerful ways to manage networks. There isn't yet complete agreement on an industry-wide standard, but a consensus about the general approach is starting to emerge. We describe one example, called *OpenConfig*, that is both getting a lot of traction and illustrates many of the key ideas that are being pursued.

The general strategy is to automate network management as much as possible, with the goal of getting the error-prone human out of the loop. This is sometimes called *zero-touch* management, and it implies two things have to happen. First, whereas historically operators used tools like SNMP to *monitor* the network, but had to log into any misbehaving network device and use a command line interface (CLI) to fix the problem, zero-touch management implies that we also need to *configure* the network programmatically. In other words, network management is equal parts reading status information and writing configuration information. The goal is to build a closed control loop, although there will always be scenarios where the operator has to be alerted that manual intervention is required.

Second, whereas historically the operator had to configure each network device individually, all the devices have to be configured in a consistent way if they are going to function correctly as a network. As a consequence, zero-touch also implies that the operator should be able to declare their network-wide *intent*, with the management tool being smart enough to issue the necessary per-device configuration directives in a globally consistent way.

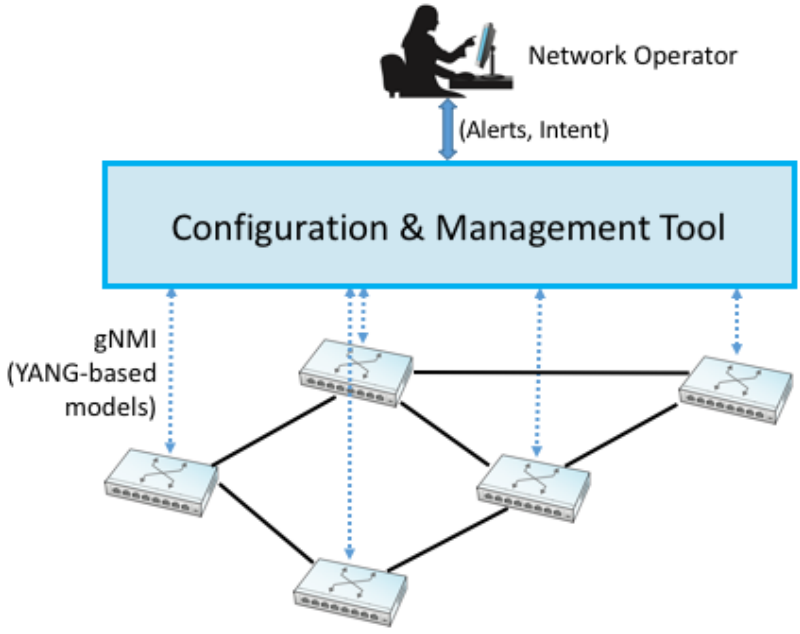


Figure 9.19.: Operator manages a network through a configuration and management tool, which in turn programmatically interacts with the underlying network devices (e.g., using gNMI as the transport protocol and YANG to specify the schema for the data being exchanged).

Figure 9.19 gives a high-level depiction of this idealized approach to network management. We say "idealized" because achieving true zero-touch management is still more aspirational than reality. But progress is being made. For example, new management tools are starting to leverage standard protocols like HTTP to monitor and configure network devices. This is a positive step because it gets us out of the business of

creating yet another request/reply protocol and lets us focus on creating smarter management tools, perhaps by taking advantage of Machine Learning algorithms to determine if something is amiss.

In the same way HTTP is starting to replace SNMP as the protocol for talking to network devices, there is a parallel effort to replace the MIB with a new standard for what status information various types of devices can report, *plus* what configuration information those same devices are able to respond to. Agreeing to a single standard for configuration is inherently challenging because every vendor claims their device is special, unlike any of the devices their competitors sell. (That is to say, the challenge is not entirely technical.)

The general approach is to allow each device manufacturer to publish a *data model* that specifies the configuration knobs (and available monitoring data) for its product, and limit standardization to the modeling language. The leading candidate is YANG, which stands for *Yet Another Next Generation*, a name chosen to poke fun at how often a do-over proves necessary. YANG can be viewed as a restricted version of XSD, which you may recall is a language for defining a schema (model) for XML. That is, YANG defines the structure of the data. But unlike XSD, YANG is not XML-specific. It can instead be used in conjunction with different over-the-wire message formats, including XML, but also ProtoBufs and JSON.

What's important about going in this direction is that the data model that defines the semantics of the variables available to be read and written in a programmatic form (i.e., it's not just text in a standards specification). It's not a free-for-all with each vendor defining a unique model since the network operators that buy network hardware have a strong incentive to drive the models for similar devices towards convergence. YANG makes the process of creating, using, and modifying models more programmable, and hence, adaptable to this process.

This is where OpenConfig comes in. It uses YANG as its modeling language, but has also established a process for driving the industry towards common models. OpenConfig is officially agnostic as to the RPC mechanism used to communicate with network devices, but one approach it is actively pursuing is called gNMI (*gRPC Network Management Interface*). As you might guess from its name, gNMI uses gRPC, which you may recall, runs on top of HTTP. This means gNMI also adopts ProtoBufs as the way it specifies the data actually communicated over the HTTP connection. Thus, as depicted in Figure 9.19, gNMI is intended as a standard management interface for network devices. What's not standardized is the richness of the management tool's ability to automate, or the exact form of the operator-facing interface. Like any application that is trying to serve a need and support more features than the alternatives, there is still much room for innovation in tools for network management.

For completeness, we note that NETCONF is another of the post-SNMP protocols for communicating configuration information to network devices. OpenConfig works with NETCONF, but our reading of the tea leaves points to gNMI as the future.

We conclude by emphasizing that a seachange is underway. While listing SNMP and OpenConfig in the title to this section suggests they are equivalent, it is more accurate to say that each is "what we call" these two approaches, but the approaches are quite different. On the one hand, SNMP is really just a transport protocol, analogous to gNMI in the OpenConfig world. It historically enabled monitoring devices, but had virtually nothing to say about configuring devices. (The latter has historically required manual intervention.) On the other hand, OpenConfig is primarily an effort to define a common set of data models for network devices, roughly similar to the role MIB plays in the SNMP world, except OpenConfig is (1) model-based, using YANG, and (2) equally focused on monitoring and configuration.

9.4 Overlay Networks

From its inception, the Internet has adopted a clean model, in which the routers inside the network are responsible for forwarding packets from source to destination, and application programs run on the hosts connected to the edges of the network. The client/server paradigm illustrated by the applications discussed in the first two sections of this chapter certainly adhere to this model.

In the last few years, however, the distinction between *packet forwarding* and *application processing* has become less clear. New applications are being distributed across the Internet, and in many cases these applications make their own forwarding decisions. These new hybrid applications can sometimes be implemented by extending traditional routers and switches to support a modest amount of application-specific processing. For example, so-called *level-7 switches* sit in front of server clusters and forward HTTP requests to a specific server based on the requested URL. However, *overlay networks* are quickly emerging as the mechanism of choice for introducing new functionality into the Internet.

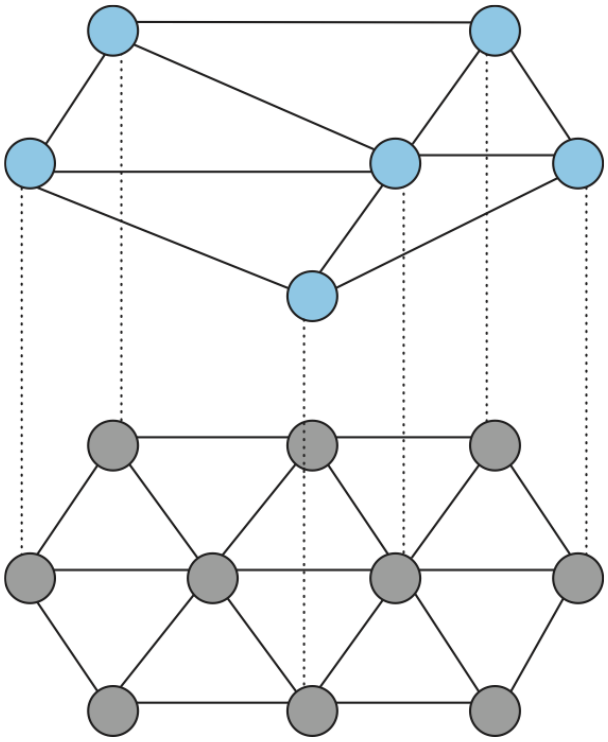


Figure 9.20.: Overlay network layered on top of a physical network.

You can think of an overlay as a logical network implemented on top of some underlying network. By this definition, the Internet started out as an overlay network on top of the links provided by the old telephone network. Figure 9.20 depicts an overlay implemented on top of an underlying network. Each node in the overlay also exists in the underlying network; it processes and forwards packets in an application-specific way. The links that connect the overlay nodes are implemented as tunnels through the underlying network. Multiple overlay networks can exist on top of the same underlying network—each implementing its own application-specific behavior—and overlays can be nested, one on top of another. For example, all of the example overlay networks discussed in this section treat today's Internet as the underlying network.

We have already seen examples of tunneling, for example, to implement virtual private networks (VPNs).

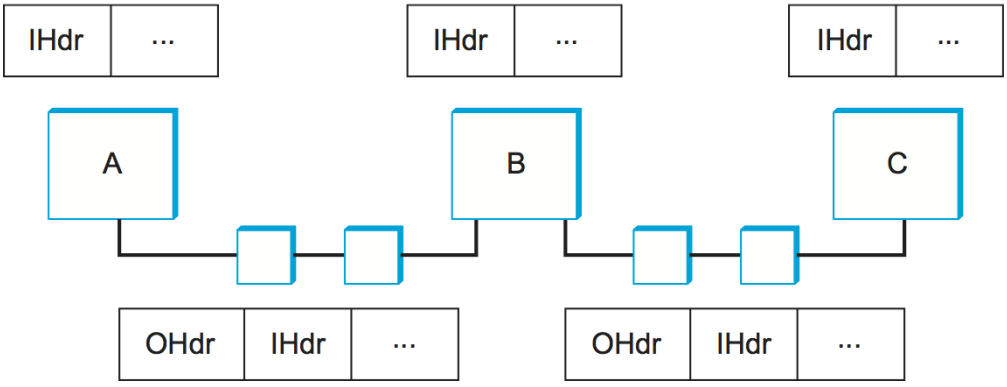


Figure 9.21.: Overlay nodes tunnel through physical nodes.

As a brief refresher, the nodes on either end of a tunnel treat the multi-hop path between them as a single logical link, the nodes that are tunneled through forward packets based on the outer header, never aware that the end nodes have attached an inner header. Figure 9.21 shows three overlay nodes (A, B, and C) connected by a pair of tunnels. In this example, overlay node B might make a forwarding decision for packets from A to C based on the inner header (`IHdr`), and then attach an outer header (`OHdr`) that identifies C as the destination in the underlying network. Nodes A, B, and C are able to interpret both the inner and outer header, whereas the intermediate routers understand only the outer header. Similarly, A, B, and C have addresses in both the overlay network and the underlying network, but they are not necessarily the same; for example, their underlying address might be a 32-bit IP address, while their overlay address might be an experimental 128-bit address. In fact, the overlay need not use conventional addresses at all but may route based on URLs, domain names, an XML query, or even the content of the packet.

9.4.1 Routing Overlays

The simplest kind of overlay is one that exists purely to support an alternative routing strategy; no additional application-level processing is performed at the overlay nodes. You can view a virtual private network (VPN) as an example of a routing overlay, but one that doesn't so much define an alternative strategy or algorithm as it does alternative routing table entries to be processed by the standard IP forwarding algorithm. In this particular case, the overlay is said to use "IP tunnels," and the ability to utilize these VPNs is supported in many commercial routers.

Suppose, however, you wanted to use a routing algorithm that commercial router vendors were not willing to include in their products. How would you go about doing it? You could simply run your algorithm on a collection of end hosts, and tunnel through the Internet routers. These hosts would behave like routers in the overlay network: As hosts they are probably connected to the Internet by only one physical link, but as a node in the overlay they would be connected to multiple neighbors via tunnels.

Since overlays, almost by definition, are a way to introduce new technologies independent of the standardization process, there are no standard overlays we can point to as examples. Instead, we illustrate the general idea of routing overlays by describing several experimental systems that have been built by network researchers.

Experimental Versions of IP

Overlays are ideal for deploying experimental versions of IP that you hope will eventually take over the world. For example, IP multicast started off as an extension to IP and even today is not enabled in many Internet routers. The MBone (multicast backbone) was an overlay network that implemented IP multicast on top of the unicast routing provided by the Internet. A number of multimedia conference tools were developed for and deployed on the Mbone. For example, IETF meetings—which are a week long and attract thousands of participants—were for many years broadcast over the MBone. (Today, the wide availability of commercial conferencing tools have replaced the MBone-based approach.)

Like VPNs, the MBone used both IP tunnels and IP addresses, but unlike VPNs, the MBone implemented a different forwarding algorithm—forwarding packets to all downstream neighbors in the shortest path multicast tree. As an overlay, multicast-aware routers tunnel through legacy routers, with the hope that one day there will be no more legacy routers.

The 6-BONE was a similar overlay that was used to incrementally deploy IPv6. Like the MBone, the 6-BONE used tunnels to forward packets through IPv4 routers. Unlike the MBone, however, 6-BONE nodes did not simply provide a new interpretation of IPv4's 32-bit addresses. Instead, they forwarded packets based on IPv6's 128-bit address space. The 6-BONE also supported IPv6 multicast. (Today, commercial routers support IPv6, but again, overlays are a valuable approach while a new technology is being evaluated and tuned.)

End System Multicast

Although IP multicast is popular with researchers and certain segments of the networking community, its deployment in the global Internet has been limited at best. In response, multicast-based applications like videoconferencing have recently turned to an alternative strategy, called *end system multicast*. The idea of end system multicast is to accept that IP multicast will never become ubiquitous and to instead let the end hosts that are participating in a particular multicast-based application implement their own multicast trees.

Before describing how end system multicast works, it is important to first understand that, unlike VPNs and the MBone, end system multicast assumes that only Internet hosts (as opposed to Internet routers) participate in the overlay. Moreover, these hosts typically exchange messages with each other through UDP tunnels rather than IP tunnels, making it easy to implement as regular application programs. This makes it possible to view the underlying network as a fully connected graph, since every host in the Internet is able to send a message to every other host. Abstractly, then, end system multicast solves the following problem: Starting with a fully connected graph representing the Internet, the goal is to find the embedded multicast tree that spans all the group members.

Note that there is a simpler version of this problem, enabled by the ready availability of cloud-hosted VMs around the world. The multicast-aware "end systems" can be VMs running at multiple sites. As these sites are well-known and relatively fixed, it's possible to construct a static multicast tree in the cloud, and have the actual end-hosts simply connect to the nearest cloud location. But for the sake of completeness, the following describes the approach in its full glory.

Since we take the underlying Internet to be fully connected, a naive solution would be to have each source directly connected to each member of the group. In other words, end system multicast could be implemented by having each node send unicast messages to every group member. To see the problem in doing this, especially compared to implementing IP multicast in routers, consider the example topology in Figure

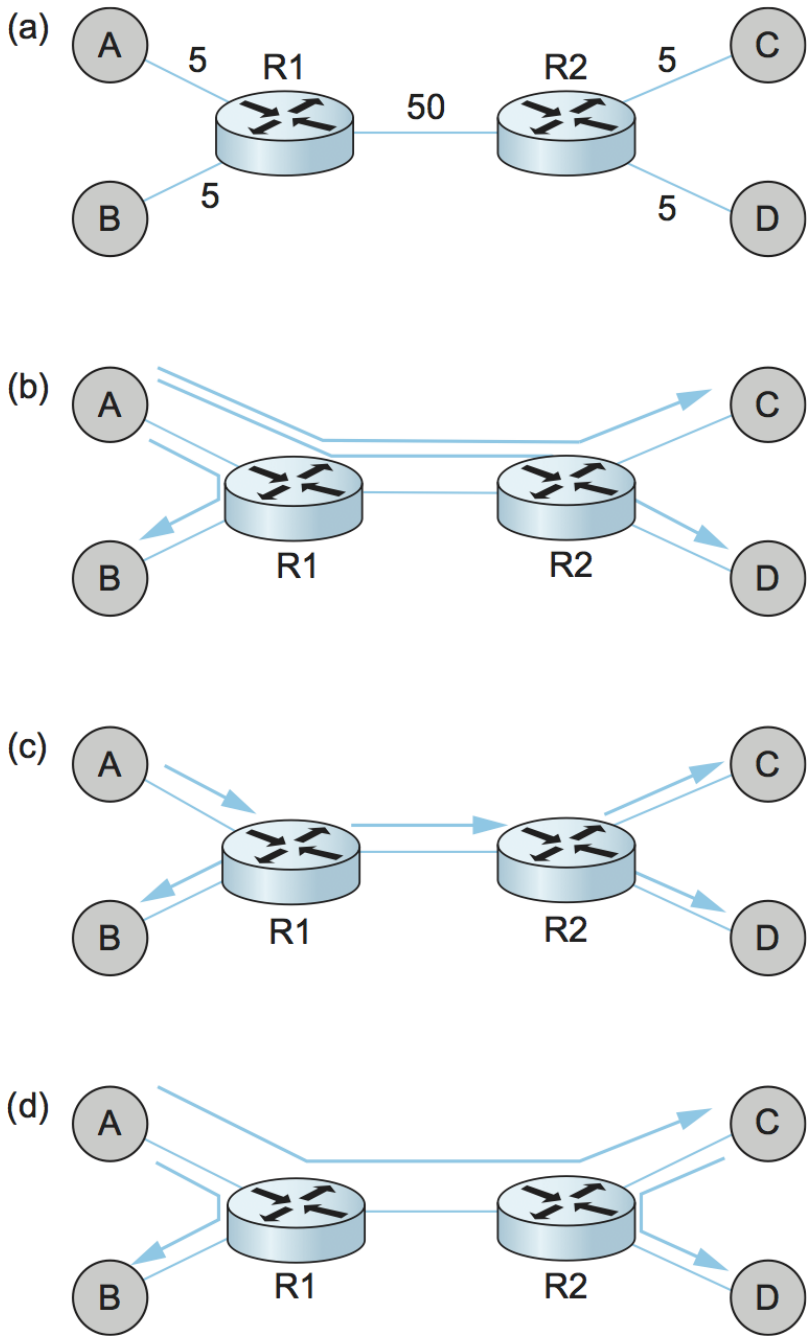


Figure 9.22.: Alternative multicast trees mapped onto a physical topology.

9.22. Figure 9.22 depicts an example physical topology, where R1 and R2 are routers connected by a low-bandwidth transcontinental link; A, B, C, and D are end hosts; and link delays are given as edge weights. Assuming A wants to send a multicast message to the other three hosts, Figure 9.22 shows how naive unicast transmission would work. This is clearly undesirable because the same message must traverse the link A-R1 three times, and two copies of the message traverse R1-R2. Figure 9.22 depicts the IP multicast tree constructed by the Distance Vector Multicast Routing Protocol (DVMRP). Clearly, this approach eliminates the redundant messages. Without support from the routers, however, the best one can hope for with end system multicast is a tree similar to the one shown in Figure 9.22. End system multicast defines an architecture for constructing this tree.

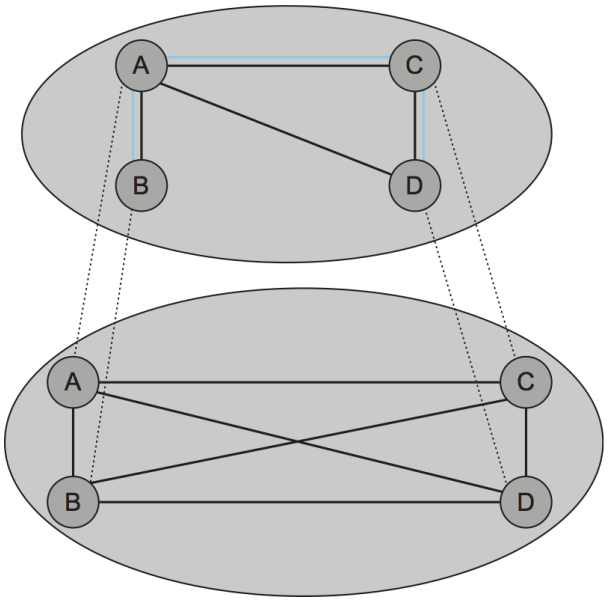


Figure 9.23.: Multicast tree embedded in an overlay network.

The general approach is to support multiple levels of overlay networks, each of which extracts a subgraph from the overlay below it, until we have selected the subgraph that the application expects. For end system multicast, in particular, this happens in two stages: First we construct a simple *mesh* overlay on top of the fully connected Internet, and then we select a multicast tree within this mesh. The idea is illustrated in Figure 9.23, again assuming the four end hosts A, B, C, and D. The first step is the critical one: Once we have selected a suitable mesh overlay, we simply run a standard multicast routing algorithm (e.g., DVMRP) on top of it to build the multicast tree. We also have the luxury of ignoring the scalability issue that Internet-wide multicast faces since the intermediate mesh can be selected to include only those nodes that want to participate in a particular multicast group.

The key to constructing the intermediate mesh overlay is to select a topology that roughly corresponds to the physical topology of the underlying Internet, but we have to do this without anyone telling us what the underlying Internet actually looks like since we are running only on end hosts and not routers. The general strategy is for the end hosts to measure the roundtrip latency to other nodes and decide to add links to the mesh only when they like what they see. This works as follows.

First, assuming a mesh already exists, each node exchanges the list of all other nodes it believes is part of the mesh with its directly connected neighbors. When a node receives such a membership list from a neighbor, it incorporates that information into its membership list and forwards the resulting list to its neighbors. This information eventually propagates through the mesh, much as in a distance vector routing protocol.

When a host wants to join the multicast overlay, it must know the IP address of at least one other node already in the overlay. It then sends a "join mesh" message to this node. This connects the new node to the mesh by an edge to the known node. In general, the new node might send a join message to multiple current nodes, thereby joining the mesh by multiple links. Once a node is connected to the mesh by a set of links, it periodically sends "keep alive" messages to its neighbors, letting them know that it still wants to be part of the group.

When a node leaves the group, it sends a "leave mesh" message to its directly connected neighbors, and this information is propagated to the other nodes in the mesh via the membership list described above. Alternatively, a node can fail or just silently decide to quit the group, in which case its neighbors detect that it is no longer sending "keep alive" messages. Some node departures have little effect on the mesh, but should a node detect that the mesh has become partitioned due to a departing node, it creates a new edge to a node in the other partition by sending it a "join mesh" message. Note that multiple neighbors can simultaneously decide that a partition has occurred in the mesh, leading to multiple cross-partition edges being added to the mesh.

As described so far, we will end up with a mesh that is a subgraph of the original fully connected Internet, but it may have suboptimal performance because (1) initial neighbor selection adds random links to the topology, (2) partition repair might add edges that are essential at the moment but not useful in the long run, (3) group membership may change due to dynamic joins and departures, and (4) underlying network conditions may change. What needs to happen is that the system must evaluate the value of each edge, resulting in new edges being added to the mesh and existing edges being removed over time.

To add new edges, each node *i* periodically probes some random member *j* that it is not currently connected to in the mesh, measures the round-trip latency of edge *(i,j)*, and then evaluates the utility of adding this edge. If the utility is above a certain threshold, link *(i,j)* is added to the mesh. Evaluating the utility of adding edge *(i,j)* might look something like this:

```
EvaluateUtility(j)
    utility = 0
    for each member m not equal to i
        CL = current latency to node m along route through mesh
        NL = new latency to node m along mesh if edge (i,j) is added}
        if (NL < CL) then
            utility += (CL - NL)/CL
    return utility
```

Deciding to remove an edge is similar, except each node *i* computes the cost of each link to current neighbor *j* as follows:

```
EvaluateCost(j)
    Cost[i,j] = number of members for which i uses j as next hop
    Cost[j,i] = number of members for which j uses i as next hop
    return max(Cost[i,j], Cost[j,i])
```

It then picks the neighbor with the lowest cost, and drops it if the cost falls below a certain threshold.

Finally, since the mesh is maintained using what is essentially a distance vector protocol, it is trivial to run DVMRP to find an appropriate multicast tree in the mesh. Note that, although it is not possible to prove that the protocol just described results in the optimum mesh network, thereby allowing DVMRP to select the best possible multicast tree, both simulation and extensive practical experience suggests that it does a good job.

Resilient Overlay Networks

Another function that can be performed by an overlay is to find alternative routes for traditional unicast applications. Such overlays exploit the observation that the triangle inequality does not hold in the Internet. Figure 9.24 illustrates what we mean by this. It is not uncommon to find three sites in the Internet—call them A, B, and C—such that the latency between A and B is greater than the sum of the latencies from A to C and from C to B. That is, sometimes you would be better off indirectly sending your packets via some intermediate node than sending them directly to the destination.

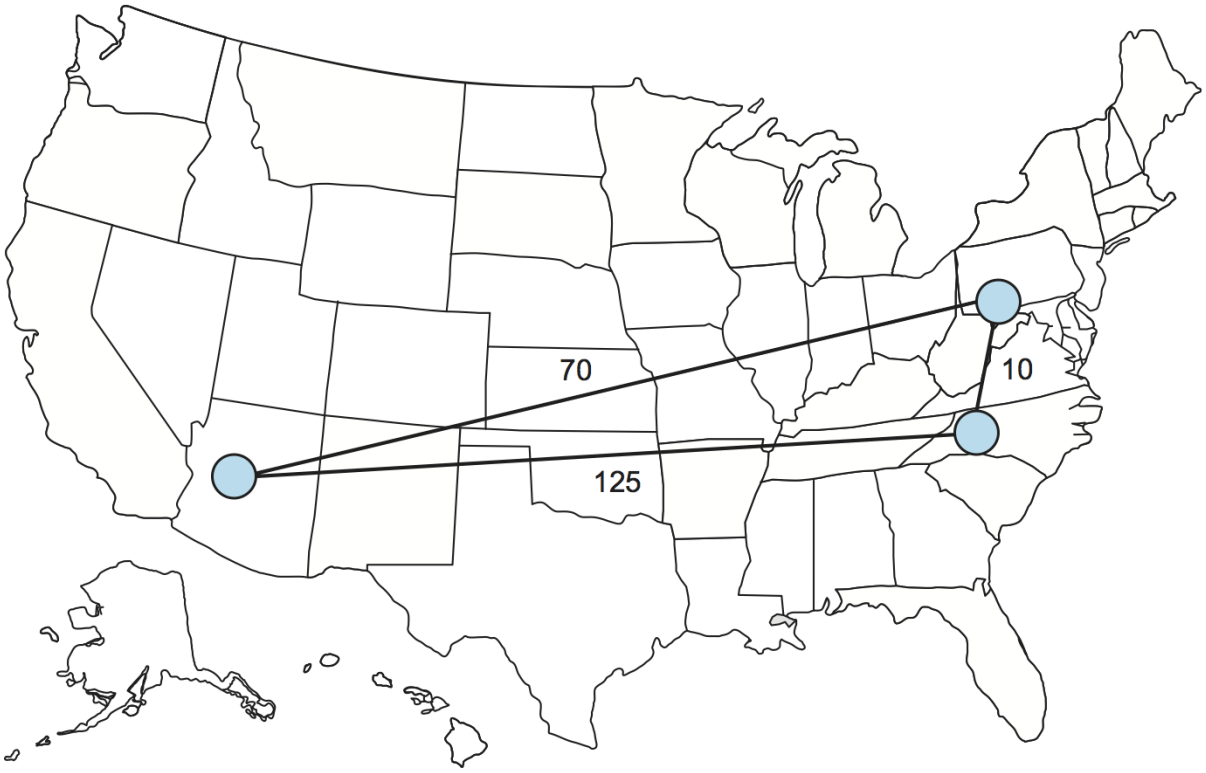


Figure 9.24.: The triangle inequality does not necessarily hold in networks.

How can this be? Well, the Border Gateway Protocol (BGP) never promised that it would find the *shortest* route between any two sites; it only tries to find *some* route. To make matters more complex, BGP's routes are heavily influenced by policy issues, such as who is paying whom to carry their traffic. This often happens, for example, at peering points between major backbone ISPs. In short, that the triangle inequality does not hold in the Internet should not come as a surprise.

How do we exploit this observation? The first step is to realize that there is a fundamental tradeoff between the scalability and optimality of a routing algorithm. On the one hand, BGP scales to very large networks, but often does not select the best possible route and is slow to adapt to network outages. On the other hand, if you were only worried about finding the best route among a handful of sites, you could do a much better job of monitoring the quality of every path you might use, thereby allowing you to select the best possible route at any moment in time.

An experimental overlay, called the Resilient Overlay Network (RON), did exactly this. RON scaled to only a few dozen nodes because it used an $N \times N$ strategy of closely monitoring (via active probes) three aspects of path quality—latency, available bandwidth, and loss probability—between every pair of sites. It was then

able to both select the optimal route between any pair of nodes, and rapidly change routes should network conditions change. Experience showed that RON was able to deliver modest performance improvements to applications, but more importantly, it recovered from network failures much more quickly. For example, during one 64-hour period in 2001, an instance of RON running on 12 nodes detected 32 outages lasting over 30 minutes, and it was able to recover from all of them in less than 20 seconds on average. This experiment also suggested that forwarding data through just one intermediate node is usually sufficient to recover from Internet failures.

Since RON was not designed to be a scalable approach, it is not possible to use RON to help random host A communicate with random host B; A and B have to know ahead of time that they are likely to communicate and then join the same RON. However, RON seems like a good idea in certain settings, such as when connecting a few dozen corporate sites spread across the Internet or allowing you and 50 of your friends to establish your own private overlay for the sake of running some application. (Today, this idea is put to practice with the marketing name *Software-Defined WAN*, or *SD-WAN*.) The real question, though, is what happens when everyone starts to run their own RON. Does the overhead of millions of RONs aggressively probing paths swamp the network, and does anyone see improved behavior when many RONs compete for the same paths? These questions are still unanswered.

Key Takeaway

All of these overlays illustrate a concept that is central to computer networks in general: *virtualization*. That is, it is possible to build a virtual network from abstract (logical) resources on top of a physical network constructed from physical resources. Moreover, it is possible to stack these virtualized networks on top of each other and for multiple virtual network to coexist at the same level. Each virtual network, in turn, provides new capabilities that are of value to some set of users, applications, or higher-level networks. *[Next]*

9.4.2 Peer-to-Peer Networks

Music-sharing applications like Napster and KaZaA introduced the term "peer-to-peer" into the popular vernacular. But what exactly does it mean for a system to be "peer-to-peer"? Certainly in the context of sharing MP3 files it means not having to download music from a central site, but instead being able to access music files directly from whoever in the Internet happens to have a copy stored on their computer. More generally then, we could say that a peer-to-peer network allows a community of users to pool their resources (content, storage, network bandwidth, disk bandwidth, CPU), thereby providing access to a larger archival store, larger video/audio conferences, more complex searches and computations, and so on than any one user could afford individually.

Quite often, attributes like *decentralized* and *self-organizing* are mentioned when discussing peer-to-peer networks, meaning that individual nodes organize themselves into a network without any centralized co-ordination. If you think about it, terms like these could be used to describe the Internet itself. Ironically, however, Napster was not a true peer-to-peer system by this definition since it depended on a central registry of known files, and users had to search this directory to find what machine offered a particular file. It was only the last step—actually downloading the file—that took place between machines that belong to two users, but this is little more than a traditional client/server transaction. The only difference is that the server is owned by someone just like you rather than a large corporation.

So we are back to the original question: What's interesting about peer-to-peer networks? One answer

is that both the process of locating an object of interest and the process of downloading that object onto your local machine happen without your having to contact a centralized authority, and at the same time the system is able to scale to millions of nodes. A peer-to-peer system that can accomplish these two tasks in a decentralized manner turns out to be an overlay network, where the nodes are those hosts that are willing to share objects of interest (e.g., music and other assorted files), and the links (tunnels) connecting these nodes represent the sequence of machines that you have to visit to track down the object you want. This description will become clearer after we look at two examples.

Gnutella

Gnutella is an early peer-to-peer network that attempted to distinguish between exchanging music (which likely violates somebody's copyright) and the general sharing of files (which must be good since we've been taught to share since the age of two). What's interesting about Gnutella is that it was one of the first such systems to not depend on a centralized registry of objects. Instead, Gnutella participants arrange themselves into an overlay network similar to the one shown in Figure 9.25. That is, each node that runs the Gnutella software (i.e., implements the Gnutella protocol) knows about some set of other machines that also run the Gnutella software. The relationship "A and B know each other" corresponds to the edges in this graph. (We'll talk about how this graph is formed in a moment.)

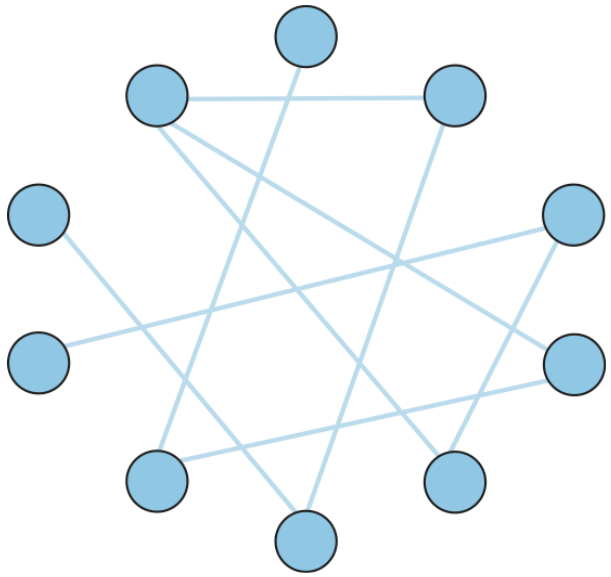


Figure 9.25.: Example topology of a gnutella peer-to-peer network.

Whenever the user on a given node wants to find an object, Gnutella sends a QUERY message for the object—for example, specifying the file's name—to its neighbors in the graph. If one of the neighbors has the object, it responds to the node that sent it the query with a QUERY RESPONSE message, specifying where the object can be downloaded (e.g., an IP address and TCP port number). That node can subsequently use GET or PUT messages to access the object. If the node cannot resolve the query, it forwards the QUERY message to each of its neighbors (except the one that sent it the query), and the process repeats. In other words, Gnutella floods the overlay to locate the desired object. Gnutella sets a TTL on each query so this flood does not continue indefinitely.

In addition to the TTL and query string, each QUERY message contains a unique query identifier (QID),

but it does not contain the identity of the original message source. Instead, each node maintains a record of the QUERY messages it has seen recently: both the QID and the neighbor that sent it the QUERY. It uses this history in two ways. First, if it ever receives a QUERY with a QID that matches one it has seen recently, the node does not forward the QUERY message. This serves to cut off forwarding loops more quickly than the TTL might have done. Second, whenever the node receives a QUERY RESPONSE from a downstream neighbor, it knows to forward the response to the upstream neighbor that originally sent it the QUERY message. In this way, the response works its way back to the original node without any of the intermediate nodes knowing who wanted to locate this particular object in the first place.

Returning to the question of how the graph evolves, a node certainly has to know about at least one other node when it joins a Gnutella overlay. The new node is attached to the overlay by at least this one link. After that, a given node learns about other nodes as the result of QUERY RESPONSE messages, both for objects it requested and for responses that just happen to pass through it. A node is free to decide which of the nodes it discovers in this way that it wants to keep as a neighbor. The Gnutella protocol provides PING and PONG messages by which a node probes whether or not a given neighbor still exists and that neighbor's response, respectively.

It should be clear that Gnutella as described here is not a particularly clever protocol, and subsequent systems have tried to improve upon it. One dimension along which improvements are possible is in how queries are propagated. Flooding has the nice property that it is guaranteed to find the desired object in the fewest possible hops, but it does not scale well. It is possible to forward queries randomly, or according to the probability of success based on past results. A second dimension is to proactively replicate the objects, since the more copies of a given object there are, the easier it should be to find a copy. Alternatively, one could develop a completely different strategy, which is the topic we consider next.

Structured Overlays

At the same time file sharing systems started fighting to fill the void left by Napster, the research community began to explore an alternative design for peer-to-peer networks. We refer to these networks as *structured*, to contrast them with the essentially random (unstructured) way in which a Gnutella network evolves. Unstructured overlays like Gnutella employ trivial overlay construction and maintenance algorithms, but the best they can offer is unreliable, random search. In contrast, structured overlays are designed to conform to a particular graph structure that allows reliable and efficient (probabilistically bounded delay) object location, in return for additional complexity during overlay construction and maintenance.

If you think about what we are trying to do at a high level, there are two questions to consider: (1) How do we map objects onto nodes, and (2) How do we route a request to the node that is responsible for a given object? We start with the first question, which has a simple statement: How do we map an object with name *x* into the address of some node *n* that is able to serve that object? While traditional peer-to-peer networks have no control over which node hosts object *x*, if we could control how objects get distributed over the network, we might be able to do a better job of finding those objects at a later time.

A well-known technique for mapping names into an address is to use a hash table, so that

$$hash(x) \rightarrow n$$

implies object *x* is first placed on node *n*, and at a later time a client trying to locate *x* would only have to perform the hash of *x* to determine that it is on node *n*. A hash-based approach has the nice property that it tends to spread the objects evenly across the set of nodes, but straightforward hashing algorithms suffer

from a fatal flaw: How many possible values of *n* should we allow? (In hashing terminology, how many buckets should there be?) Naively, we could decide that there are, say, 101 possible hash values, and we use a modulo hash function; that is,

```
hash(x)
    return x % 101
```

Unfortunately, if there are more than 101 nodes willing to host objects, then we can't take advantage of all of them. On the other hand, if we select a number larger than the largest possible number of nodes, then there will be some values of *x* that will hash into an address for a node that does not exist. There is also the not-so-small issue of translating the value returned by the hash function into an actual IP address.

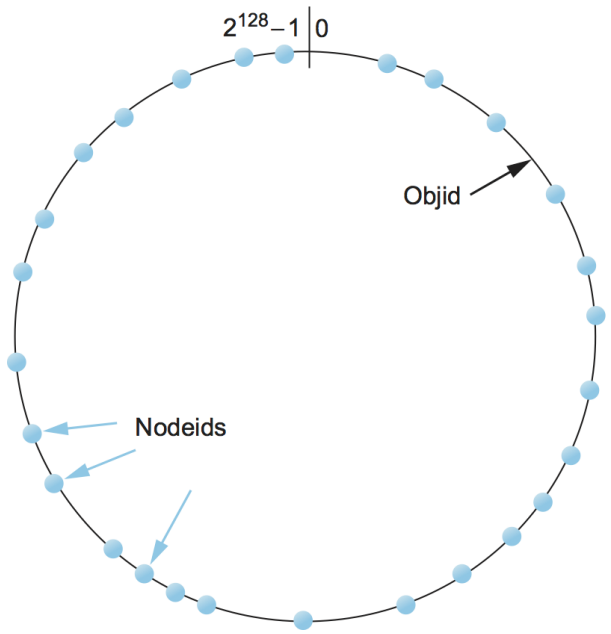


Figure 9.26.: Both nodes and objects map (hash) onto the ID space, where objects are maintained at the nearest node in this space.

To address these issues, structured peer-to-peer networks use an algorithm known as *consistent hashing*, which hashes a set of objects *x* uniformly across a large ID space. Figure 9.26 visualizes a 128-bit ID space as a circle, where we use the algorithm to place both objects

$$hash(ObjectName) \rightarrow Objid$$

and nodes

$$hash(IPAddr) \rightarrow Nodeid$$

onto this circle. Since a 128-bit ID space is enormous, it is unlikely that an object will hash to exactly the same ID as a machine's IP address hashes to. To account for this unlikelihood, each object is maintained on the node whose ID is *closest*, in this 128-bit space, to the object ID. In other words, the idea is to use a high-quality hash function to map both nodes and objects into the same large, sparse ID space; you then map objects to nodes by numerical proximity of their respective identifiers. Like ordinary hashing, this distributes objects fairly evenly across nodes, but, unlike ordinary hashing, only a small number of objects have to move when a node (hash bucket) joins or leaves.

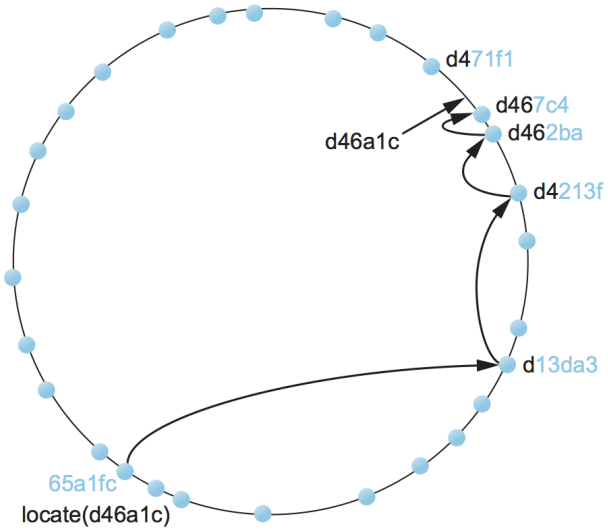


Figure 9.27.: Objects are located by routing through the peer-to-peer overlay network.

We now turn to the second question—how does a user that wants to access object x know which node is closest in x's ID in this space? One possible answer is that each node keeps a complete table of node IDs and their associated IP addresses, but this would not be practical for a large network. The alternative, which is the approach used by structured peer-to-peer networks, is to *route a message to this node!* In other words, if we construct the overlay in a clever way—which is the same as saying that we need to choose entries for a node's routing table in a clever way—then we find a node simply by routing toward it. Collectively, this approach is sometimes called a *distributed hash table* (DHT), since conceptually, the hash table is distributed over all the nodes in the network.

Figure 9.27 illustrates what happens for a simple 28-bit ID space. To keep the discussion as concrete as possible, we consider the approach used by a particular peer-to-peer network called *Pastry*. Other systems work in a similar manner.

Suppose you are at the node with id `65a1fc` (hex) and you are trying to locate the object with ID `d46a1c`. You realize that your ID shares nothing with the object's, but you know of a node that shares at least the prefix `d`. That node is closer than you in the 128-bit ID space, so you forward the message to it. (We do not give the format of the message being forwarded, but you can think of it as saying "locate object `d46a1c`.") Assuming node `d13da3` knows of another node that shares an even longer prefix with the object, it forwards the message on. This process of moving closer in ID-space continues until you reach a node that knows of no closer node. This node is, by definition, the one that hosts the object. Keep in mind that as we logically move through "ID space" the message is actually being forwarded, node to node, through the underlying Internet.

Each node maintains a both routing table (more below) and the IP addresses of a small set of numerically larger and smaller node IDs. This is called the node's *leaf set*. The relevance of the leaf set is that, once a message is routed to any node in the same leaf set as the node that hosts the object, that node can directly forward the message to the ultimate destination. Said another way, the leaf set facilitates correct and efficient delivery of a message to the numerically closest node, even though multiple nodes may exist that share a maximal length prefix with the object ID. Moreover, the leaf set makes routing more robust because any of the nodes in a leaf set can route a message just as well as any other node in the same set. Thus, if one node is unable to make progress routing a message, one of its neighbors in the leaf set may be able to. In summary,

the routing procedure is defined as follows:

```
Route(D)
    if D is within range of my leaf set
        forward to numerically closest member in leaf set
    else
        let l = length of shared prefix
        let d = value of l-th digit in D's address
        if RouteTab[l,d] exists
            forward to RouteTab[l,d]
        else
            forward to known node with at least as long a shared prefix
            and numerically closer than this node
```

The routing table, denoted `RouteTab`, is a two-dimensional array. It has a row for every hex digit in an ID (there such 32 digits in a 128-bit ID) and a column for every hex value (there are obviously 16 such values). Every entry in row *i* shares a prefix of length *i* with this node, and within this row the entry in column *j* has the hex value *j* in the *i+1*-th position. Figure 9.28 shows the first three rows of an example routing table for node `65a1fcx`, where *x* denotes an unspecified suffix. This figure shows the ID prefix matched by every entry in the table. It does not show the actual value contained in this entry—the IP address of the next node to route to.

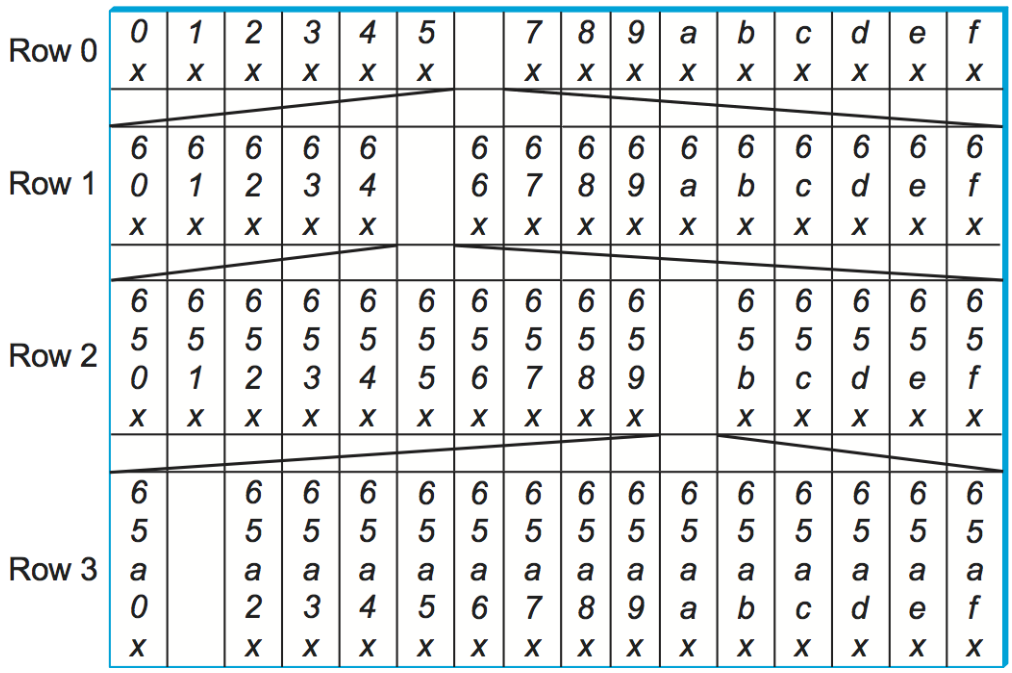


Figure 9.28.: Example routing table at the node with ID 65alcx

Adding a node to the overlay works much like routing a "locate object message" to an object. The new node must know of at least one current member. It asks this member to route an "add node message" to the node numerically closest to the ID of the joining node, as shown in Figure 9.29. It is through this routing process that the new node learns about other nodes with a shared prefix and is able to begin filling out its routing table. Over time, as additional nodes join the overlay, existing nodes also have the option of including information about the newly joined node in their routing tables. They do this when the new node adds a

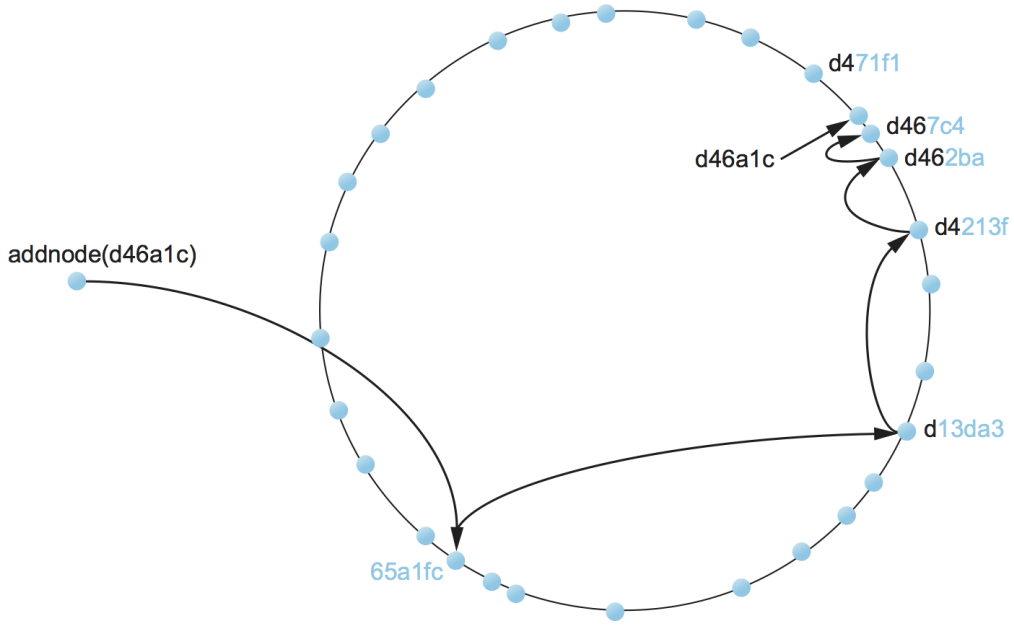


Figure 9.29.: Adding a node to the network.

longer prefix than they currently have in their table. Neighbors in the leaf sets also exchange routing tables with each other, which means that over time routing information propagates through the overlay.

The reader may have noticed that although structured overlays provide a probabilistic bound on the number of routing hops required to locate a given object—the number of hops in Pastry is bounded by $log_{16}N$, where N is the number of nodes in the overlay—each hop may contribute substantial delay. This is because each intermediate node may be at a random location in the Internet. (In the worst case, each node is on a different continent!) In fact, in a world-wide overlay network using the algorithm as described above, the expected delay of each hop is the average delay among all pairs of nodes in the Internet! Fortunately, one can do much better in practice. The idea is to choose each routing table entry such that it refers to a nearby node in the underlying physical network, among all nodes with an ID prefix that is appropriate for the entry. It turns out that doing so achieves end-to-end routing delays that are within a small factor of the delay between source and destination node.

Finally, the discussion up to this point has focused on the general problem of locating objects in a peer-to-peer network. Given such a routing infrastructure, it is possible to build different services. For example, a file sharing service would use file names as object names. To locate a file, you first hash its name into a corresponding object ID and then route a "locate object message" to this ID. The system might also replicate each file across multiple nodes to improve availability. Storing multiple copies on the leaf set of the node to which a given file normally routes would be one way of doing this. Keep in mind that even though these nodes are neighbors in the ID space, they are likely to be physically distributed across the Internet. Thus, while a power outage in an entire city might take down physically close replicas of a file in a traditional file system, one or more replicas would likely survive such a failure in a peer-to-peer network.

Services other than file sharing can also be built on top of distributed hash tables. Consider multicast applications, for example. Instead of constructing a multicast tree from a mesh, one could construct the tree from edges in the structured overlay, thereby amortizing the cost of overlay construction and maintenance across several applications and multicast groups.

BitTorrent

BitTorrent is a peer-to-peer file sharing protocol devised by Bram Cohen. It is based on replicating the file or, rather, replicating segments of the file, which are called *pieces*. Any particular piece can usually be downloaded from multiple peers, even if only one peer has the entire file. The primary benefit of BitTorrent's replication is avoiding the bottleneck of having only one source for a file. This is particularly useful when you consider that any given computer has a limited speed at which it can serve files over its uplink to the Internet, often quite a low limit due to the asymmetric nature of most broadband networks. The beauty of BitTorrent is that replication is a natural side effect of the downloading process: As soon as a peer downloads a particular piece, it becomes another source for that piece. The more peers downloading pieces of the file, the more piece replication occurs, distributing the load proportionately, and the more total bandwidth is available to share the file with others. Pieces are downloaded in random order to avoid a situation where peers find themselves lacking the same set of pieces.

Each file is shared via its own independent BitTorrent network, called a *swarm*. (A swarm could potentially share a set of files, but we describe the single file case for simplicity.) The lifecycle of a typical swarm is as follows. The swarm starts as a singleton peer with a complete copy of the file. A node that wants to download the file joins the swarm, becoming its second member, and begins downloading pieces of the file from the original peer. In doing so, it becomes another source for the pieces it has downloaded, even if it has not yet downloaded the entire file. (In fact, it is common for peers to leave the swarm once they have completed their downloads, although they are encouraged to stay longer.) Other nodes join the swarm and begin downloading pieces from multiple peers, not just the original peer. See Figure 9.30.

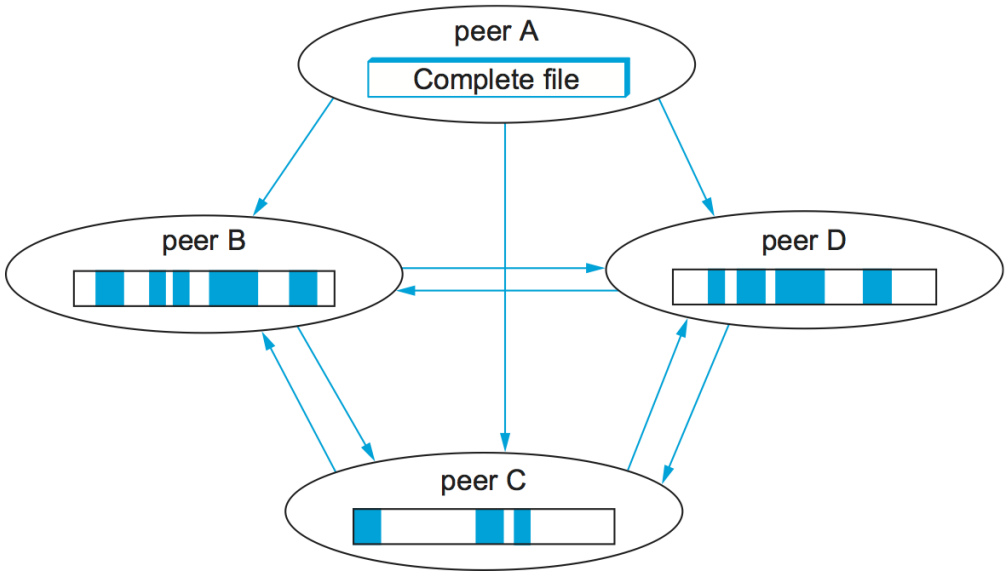


Figure 9.30.: Peers in a BitTorrent swarm download from other peers that may not yet have the complete file.

If the file remains in high demand, with a stream of new peers replacing those who leave the swarm, the swarm could remain active indefinitely; if not, it could shrink back to include only the original peer until new peers join the swarm.

Now that we have an overview of BitTorrent, we can ask how requests are routed to the peers that have a given piece. To make requests, a would-be downloader must first join the swarm. It starts by downloading

a file containing meta-information about the file and swarm. The file, which may be easily replicated, is typically downloaded from a web server and discovered by following links from Web pages. It contains:

- The target file's size
- The piece size
- SHA-1 hash values precomputed from each piece
- The URL of the swarm's *tracker*

A tracker is a server that tracks a swarm's current membership. We'll see later that BitTorrent can be extended to eliminate this point of centralization, with its attendant potential for bottleneck or failure.

The would-be downloader then joins the swarm, becoming a peer, by sending a message to the tracker giving its network address and a peer ID that it has generated randomly for itself. The message also carries a SHA-1 hash of the main part of the file, which is used as a swarm ID.

Let's call the new peer P. The tracker replies to P with a partial list of peers giving their IDs and network addresses, and P establishes connections, over TCP, with some of these peers. Note that P is directly connected to just a subset of the swarm, although it may decide to contact additional peers or even request more peers from the tracker. To establish a BitTorrent connection with a particular peer after their TCP connection has been established, P sends P's own peer ID and swarm ID, and the peer replies with its peer ID and swarm ID. If the swarm IDs don't match, or the reply peer ID is not what P expects, the connection is aborted.

The resulting BitTorrent connection is symmetric: Each end can download from the other. Each end begins by sending the other a bitmap reporting which pieces it has, so each peer knows the other's initial state. Whenever a downloader (D) finishes downloading another piece, it sends a message identifying that piece to each of its directly connected peers, so those peers can update their internal representation of D's state. This, finally, is the answer to the question of how a download request for a piece is routed to a peer that has the piece, because it means that each peer knows which directly connected peers have the piece. If D needs a piece that none of its connections has, it could connect to more or different peers (it can get more from the tracker) or occupy itself with other pieces in hopes that some of its connections will obtain the piece from their connections.

How are objects—in this case, pieces—mapped onto peer nodes? Of course each peer eventually obtains all the pieces, so the question is really about which pieces a peer has at a given time before it has all the pieces or, equivalently, about the order in which a peer downloads pieces. The answer is that they download pieces in random order, to keep them from having a strict subset or superset of the pieces of any of their peers.

The BitTorrent described so far utilizes a central tracker that constitutes a single point of failure for the swarm and could potentially be a performance bottleneck. Also, providing a tracker can be a nuisance for someone who would like to make a file available via BitTorrent. Newer versions of BitTorrent additionally support "trackerless" swarms that use a DHT-based implementation. BitTorrent client software that is trackerless capable implements not just a BitTorrent peer but also what we'll call a *peer finder* (the BitTorrent terminology is simply *node*), which the peer uses to find peers.

Peer finders form their own overlay network, using their own protocol over UDP to implement a DHT. Furthermore, a peer finder network includes peer finders whose associated peers belong to different swarms. In other words, while each swarm forms a distinct network of BitTorrent peers, a peer finder network instead spans swarms.

Peer finders randomly generate their own finder IDs, which are the same size (160 bits) as swarm IDs. Each finder maintains a modest table containing primarily finders (and their associated peers) whose IDs

are close to its own, plus some finders whose IDs are more distant. The following algorithm ensures that finders whose IDs are close to a given swarm ID are likely to know of peers from that swarm; the algorithm simultaneously provides a way to look them up. When a finder F needs to find peers from a particular swarm, it sends a request to the finders in its table whose IDs are close to that swarm's ID. If a contacted finder knows of any peers for that swarm, it replies with their contact information. Otherwise, it replies with the contact information of the finders in its table that are close to the swarm, so that F can iteratively query those finders.

After the search is exhausted, because there are no finders closer to the swarm, F inserts the contact information for itself and its associated peer into the finders closest to the swarm. The net effect is that peers for a particular swarm get entered in the tables of the finders that are close to that swarm.

The above scheme assumes that F is already part of the finder network, that it already knows how to contact some other finders. This assumption is true for finder installations that have run previously, because they are supposed to save information about other finders, even across executions. If a swarm uses a tracker, its peers are able to tell their finders about other finders (in a reversal of the peer and finder roles) because the BitTorrent peer protocol has been extended to exchange finder contact information. But, how can a newly installed finder discover other finders? The files for trackerless swarms include contact information for one or a few finders, instead of a tracker URL, for just that situation.

An unusual aspect of BitTorrent is that it deals head-on with the issue of fairness, or good "network citizenship." Protocols often depend on the good behavior of individual peers without being able to enforce it. For example, an unscrupulous Ethernet peer could get better performance by using a backoff algorithm that is more aggressive than exponential backoff, or an unscrupulous TCP peer could get better performance by not cooperating in congestion control.

The good behavior that BitTorrent depends on is peers uploading pieces to other peers. Since the typical BitTorrent user just wants to download the file as quickly as possible, there is a temptation to implement a peer that tries to download all the pieces while doing as little uploading as possible—this is a bad peer. To discourage bad behavior, the BitTorrent protocol includes mechanisms that allow peers to reward or punish each other. If a peer is misbehaving by not nicely uploading to another peer, the second peer can *choke* the bad peer: It can decide to stop uploading to the bad peer, at least temporarily, and send it a message saying so. There is also a message type for telling a peer that it has been unchoked. The choking mechanism is also used by a peer to limit the number of its active BitTorrent connections, to maintain good TCP performance. There are many possible choking algorithms, and devising a good one is an art.

9.4.3 Content Distribution Networks

We have already seen how HTTP running over TCP allows web browsers to retrieve pages from web servers. However, anyone who has waited an eternity for a Web page to return knows that the system is far from perfect. Considering that the backbone of the Internet is now constructed from 40-Gbps links, it's not obvious why this should happen. It is generally agreed that when it comes to downloading Web pages there are four potential bottlenecks in the system:

- *The first mile.* The Internet may have high-capacity links in it, but that doesn't help you download a Web page any faster when you're connected by a 1.5Mbps DSL line or a poorly performing wireless link.
- *The last mile.* The link that connects the server to the Internet can be overloaded by too many requests, even if the aggregate bandwidth of that link is quite high.

- *The server itself.* A server has a finite amount of resources (CPU, memory, disk bandwidth, etc.) and can be overloaded by too many concurrent requests.
- *Peering points.* The handful of ISPs that collectively implement the backbone of the Internet may internally have high-bandwidth pipes, but they have little motivation to provide high-capacity connectivity to their peers. If you are connected to ISP A and the server is connected to ISP B, then the page you request may get dropped at the point where A and B peer with each other.

There's not a lot anyone except you can do about the first problem, but it is possible to use replication to address the remaining problems. Systems that do this are often called *Content Distribution Networks* (CDNs). Akamai operates what is probably the best-known CDN.

The idea of a CDN is to geographically distribute a collection of *server surrogates* that cache pages normally maintained in some set of *backend servers*. Thus, rather than having millions of users wait forever to contact when a big news story breaks—such a situation is known as a *flash crowd*—it is possible to spread this load across many servers. Moreover, rather than having to traverse multiple ISPs to reach , if these surrogate servers happen to be spread across all the backbone ISPs, then it should be possible to reach one without having to cross a peering point. Clearly, maintaining thousands of surrogate servers all over the Internet is too expensive for any one site that wants to provide better access to its Web pages. Commercial CDNs provide this service for many sites, thereby amortizing the cost across many customers.

Although we call them surrogate servers, in fact, they can just as correctly be viewed as caches. If they don't have a page that has been requested by a client, they ask the backend server for it. In practice, however, the backend servers proactively replicate their data across the surrogates rather than wait for surrogates to request it on demand. It's also the case that only static pages, as opposed to dynamic content, are distributed across the surrogates. Clients have to go to the backend server for any content that either changes frequently (e.g., sports scores and stock quotes) or is produced as the result of some computation (e.g., a database query).

Having a large set of geographically distributed servers does not fully solve the problem. To complete the picture, CDNs also need to provide a set of *redirectors* that forward client requests to the most appropriate server, as shown in Figure 9.31. The primary objective of the redirectors is to select the server for each request that results in the best *response time* for the client. A secondary objective is for the system as a whole to process as many requests per second as the underlying hardware (network links and web servers) is able to support. The average number of requests that can be satisfied in a given time period—known as the *system throughput*—is primarily an issue when the system is under heavy load, such as when a flash crowd is accessing a small set of pages or a Distributed Denial of Service (DDoS) attacker is targeting a particular site, as happened to CNN, Yahoo, and several other high-profile sites in February 2000.

CDNs use several factors to decide how to distribute client requests. For example, to minimize response time, a redirector might select a server based on its *network proximity*. In contrast, to improve the overall system throughput, it is desirable to evenly *balance* the load across a set of servers. Both throughput and response time are improved if the distribution mechanism takes *locality* into consideration; that is, it selects a server that is likely to already have the page being requested in its cache. The exact combination of factors that should be employed by a CDN is open to debate. This section considers some of the possibilities.

Mechanisms

As described so far, a redirector is just an abstract function, although it sounds like what something a router might be asked to do since it logically forwards a request message much like a router forwards packets. In

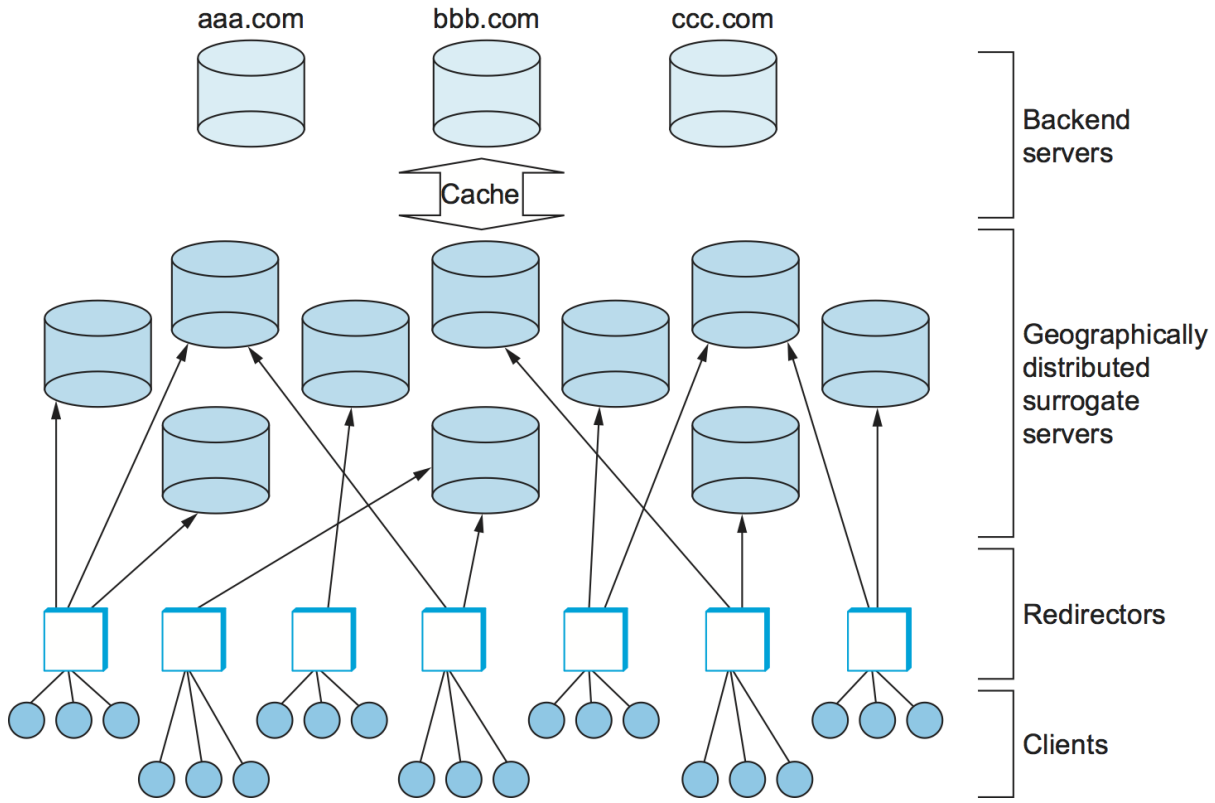


Figure 9.31.: Components in a Content Distribution Network (CDN).

fact, there are several mechanisms that can be used to implement redirection. Note that for the purpose of this discussion we assume that each redirector knows the address of every available server. (From here on, we drop the "surrogate" qualifier and talk simply in terms of a set of servers.) In practice, some form of out-of-band communication takes place to keep this information up-to-date as servers come and go.

First, redirection could be implemented by augmenting DNS to return different server addresses to clients. For example, when a client asks to resolve the name , the DNS server could return the IP address of a server hosting CNN's Web pages that is known to have the lightest load. Alternatively, for a given set of servers, it might just return addresses in a round-robin fashion. Note that the granularity of DNS-based redirection is usually at the level of a site (e.g.,) rather than a specific URL (e.g.,). However, when returning an embedded link, the server can rewrite the URL, thereby effectively pointing the client at the most appropriate server for that specific object.

Commercial CDNs essentially use a combination of URL rewriting and DNS-based redirection. For scalability reasons, the high-level DNS server first points to a regional-level DNS server, which replies with the actual server address. In order to respond to changes quickly, the DNS servers tweak the TTL of the resource records they return to a very short period, such as 20 seconds. This is necessary so clients don't cache results and thus fail to go back to the DNS server for the most recent URL-to-server mapping.

Another possibility is to use the HTTP redirect feature: The client sends a request message to a server, which responds with a new (better) server that the client should contact for the page. Unfortunately, server-based redirection incurs an additional round-trip time across the Internet, and, even worse, servers can be vulnerable to being overloaded by the redirection task itself. Instead, if there is a node close to the client (e.g., a local Web proxy) that is aware of the available servers, then it can intercept the request message and

instruct the client to instead request the page from an appropriate server. In this case, either the redirector would need to be on a choke point so that all requests leaving the site pass through it, or the client would have to cooperate by explicitly addressing the proxy (as with a classical, rather than transparent, proxy).

At this point you may be wondering what CDNs have to do with overlay networks, and while viewing a CDN as an overlay is a bit of a stretch, they do share one very important trait in common. Like an overlay node, a proxy-based redirector makes an application-level routing decision. Rather than forward a packet based on an address and its knowledge of the network topology, it forwards HTTP requests based on a URL and its knowledge of the location and load of a set of servers. Today's Internet architecture does not support redirection directly—where by "directly" we mean the client sends the HTTP request to the redirector, which forwards to the destination—so instead redirection is typically implemented indirectly by having the redirector return the appropriate destination address and the client contacts the server itself.

Policies

We now consider some example policies that redirectors might use to forward requests. Actually, we have already suggested one simple policy—round-robin. A similar scheme would be to simply select one of the available servers at random. Both of these approaches do a good job of spreading the load evenly across the CDN, but they do not do a particularly good job of lowering the client-perceived response time.

It's obvious that neither of these two schemes takes network proximity into consideration, but, just as importantly, they also ignore locality. That is, requests for the same URL are forwarded to different servers, making it less likely that the page will be served from the selected server's in-memory cache. This forces the server to retrieve the page from its disk, or possibly even from the backend server. How can a distributed set of redirectors cause requests for the same page to go to the same server (or small set of servers) without global coordination? The answer is surprisingly simple: All redirectors use some form of hashing to deterministically map URLs into a small range of values. The primary benefit of this approach is that no inter-redirector communication is required to achieve coordinated operation; no matter which redirector receives a URL, the hashing process produces the same output.

So what makes for a good hashing scheme? The classic *modulo* hashing scheme—which hashes each URL modulo the number of servers—is not suitable for this environment. This is because should the number of servers change, the modulo calculation will result in a diminishing fraction of the pages keeping their same server assignments. While we do not expect frequent changes in the set of servers, the fact that the addition of new servers into the set will cause massive reassignment is undesirable.

An alternative is to use the same *consistent hashing* algorithm discussed in the previous section. Specifically, each redirector first hashes every server into the unit circle. Then, for each URL that arrives, the redirector also hashes the URL to a value on the unit circle, and the URL is assigned to the server that lies closest on the circle to its hash value. If a node fails in this scheme, its load shifts to its neighbors (on the unit circle), so the addition or removal of a server only causes local changes in request assignments. Note that unlike the peer-to-peer case, where a message is routed from one node to another in order to find the server whose ID is closest to the objects, each redirector knows how the set of servers map onto the unit circle, so they can each, independently, select the "nearest" one.

This strategy can easily be extended to take server load into account. Assume the redirector knows the current load of each of the available servers. This information may not be perfectly up-to-date, but we can imagine the redirector simply counting how many times it has forwarded a request to each server in the last few seconds and using this count as an estimate of that server's current load. Upon receiving a URL, the

redirector hashes the URL plus each of the available servers and sorts the resulting values. This sorted list effectively defines the order in which the redirector will consider the available servers. The redirector then walks down this list until it finds a server whose load is below some threshold. The benefit of this approach compared to plain consistent hashing is that server order is different for each URL, so if one server fails its load is distributed evenly among the other machines. This approach is the basis for the Cache Array Routing Protocol (CARP) and is shown in pseudocode below.

```
SelectServer(URL, S)
    for each server s in server set S
        weight[s] = hash(URL, address[s])
    sort weight
    for each server s in decreasing order of weight
        if Load(s) < threshold then
            return s
       return server with highest weight
```

As the load increases, this scheme changes from using only the first server on the sorted list to spreading requests across several servers. Some pages normally handled by busy servers will also start being handled by less busy servers. Since this process is based on aggregate server load rather than the popularity of individual pages, servers hosting some popular pages may find more servers sharing their load than servers hosting collectively unpopular pages. In the process, some unpopular pages will be replicated in the system simply because they happen to be primarily hosted on busy servers. At the same time, if some pages become extremely popular, it is conceivable that all of the servers in the system could be responsible for serving them.

Finally, it is possible to introduce network proximity into the equation in at least two different ways. The first is to blur the distinction between server load and network proximity by monitoring how long a server takes to respond to requests and using this measurement as the "server load" parameter in the preceding algorithm. This strategy tends to prefer loaded servers over distant/heavily loaded servers. A second approach is to factor proximity into the decision at an earlier stage by limiting the candidate set of servers considered by the above algorithms (S) to only those that are nearby. The harder problem is deciding which of the potentially many servers are suitably close. One approach would be to select only those servers that are available on the same ISP as the client. A slightly more sophisticated approach would be to look at the map of autonomous systems produced by BGP and select only those servers within some number of hops from the client as candidate servers. Finding the right balance between network proximity and server cache locality is a subject of ongoing research.

Perspective: The Cloud is the New Internet

As we saw at the end of *Section 9.1*, there has been a migration of traditional Internet applications like email and web servers from machines running on-premises to VMs running in commodity clouds. This corresponds to a shift in terminology (from "Web Services" to "Cloud Services") and in many of the underlying technologies being used (from Virtual Machines to Cloud Native micro-services). But the Cloud's impact on how network applications are implemented today is even bigger than this migration suggests. It is the combination of commodity clouds and overlay networks (similar to those described in *Section 9.4*) that may eventually have the most impact.

The biggest thing an overlay-based application needs to be effective is a wide footprint, that is, many points-of-presence around the world. IP routers are widely deployed, so if you have permission to use a set of them

as the underlying nodes in your overlay network, then you're good-to-go. But that's not going to happen, as there are exactly zero network operators or enterprise administrators that are willing to let random people load overlay software onto their routers.

Your next choice might be to crowdsource hosting sites for your overlay software. Depending on the kindness of strangers works if you all share a common goal, like downloading free music, but it's difficult for a new overlay application to go viral, and even if it does, making sure there is sufficient capacity at any given time to carry all the traffic your application generates is often problematic. It sometimes works for free services, but not any application you might hope to monetize.

If only there were a way to pay someone for the right to load and run your software on servers spread all over the world. Of course, that's exactly what commodity clouds like Amazon AWS, Microsoft Azure, and the Google Cloud Platform provide. To many, the cloud offers a seemingly unlimited number of servers, but it's actually just as important—if not more important—where these servers are located. As we discussed at the end of *Chapter 4*, they are widely distributed across 150+ well-connected sites.

Suppose, for example, that you want to stream a collection of live video or audio channels to millions of users, or you want to support thousands of video conferencing sessions, each of which connects a dozen widely distributed participants. In both cases, you construct an overlay multicast tree (one per video channel in the first example, and one per conference session in the second example), with the overlay nodes in the tree located at some combination of those 150 cloud sites. Then you allow the end-users, from their general-purpose web browsers or purpose-built smartphone apps, connect to the multicast tree(s) of their choice. If you need to store some of the video/audio content to play at a later time (e.g., to support time shifting) then you might also buy some storage capacity at some or all of those cloud sites, effectively building your own Content Distribution Network.

Taking the long view, while the Internet was originally conceived as a pure communication service, with arbitrary compute-and-storage applications allowed to flourish around the edges, today application software is for all practical purposes embedded within (distributed across) the network, and it is increasingly difficult to tell where the Internet stops and the Cloud starts. This blending will only continue to deepen as the cloud moves closer and closer to the edge (e.g., to thousands of sites where access networks are anchored) and the economies-of-scale drive the hardware devices used to build Internet/Cloud sites increasingly towards commonality.

Broader Perspective

To remind yourself of why the cloudification of the Internet is important, see *Perspective: Feature Velocity*.

ABOUT THIS BOOK

This site contains source text for *Computer Networks: A Systems Approach*, now available under terms of the Creative Commons (CC BY 4.0) license. The community is invited to contribute corrections, improvements, updates, and new material under the same terms.

Like many open source software projects, this one has been seeded with once restricted content: the 5th edition of *Peterson and Davie*, copyrighted by Elsevier. Our hope is that open sourcing this material will both make it widely available and serve as an attractor for new content: updating what's already there, expanding it to cover new topics, and augmenting the text with additional teaching collateral.

We will initially play an editorial role (curating and wordsmithing) for contributions that come back, but our plan is to share ownership of the project with others committed to its success.

And if you make use of this work, the attribution should include the following information:

Title: *Computer Networks: A Systems Approach*
Authors: Larry Peterson and Bruce Davie
Copyright: Elsevier, 2012
Source: https://github.com/SystemsApproach
License: CC BY 4.0

Read the Book

An online version of the book is published at https://book.systemsapproach.org. You can also find a PDF version here.

To track progress and receive notices about new versions, you can follow the project on Facebook and Twitter. To read a running commentary on how the Internet is evolving, follow the Systems Approach Blog.

Releases and Editions

We release ever-changing open source content rather than publish fixed books, although you can roughly equate v6.0 with a 6th Edition. Read the Preface to find out what's new in this version. Note that Morgan

Kaufmann (Elsevier) plans to publish a 6th edition of their textbook based on a fork of v6.0, but going forward, open source releases found here will not necessarily stay in sync with any future published editions.

In general, `master` contains a coherent and internally consistent version of the material. (If it were code, the book would build and run.) New content under development is checked into branches until it can be merged into `master` without breaking self-consistency. The web version of the book available at https://book.systemsapproach.org is periodically generated from `master`, and corresponds to a typical maintenance release, although we do not bother to tag it as such (e.g., 6.0.1).

Minor releases (e.g., v6.1) are tagged whenever there is sufficient new content to justify the effort. This happens quarterly, give-or-take, and is primarily to create a snapshot so that everyone in a course can know they are using the same version. The off-line formats (e.g., pdf, epub) are also generated with every minor release.

Build the Book

The source content is organized as a git repository per chapter, each of which focuses on a major networking topic (e.g., *Internetworking*, *Congestion Control*). A "root" repo (this one) contains the top-level files needed to assemble a full book.

To build a web-viewable version, you first need to download the source:

```
mkdir ~/systemsapproach
cd ~/systemsapproach
git clone https://github.com/systemsapproach/book.git
cd book
git submodule init
git submodule update
```

The build process is stored in the `Makefile` and requires Python be installed. The `Makefile` will create a virtualenv (`doc_venv`) which installs the documentation generation toolset.

To generate HTML in `_build/html`, run `make html`.

To get a live reload in your browser (refreshes on file save), run `make reload`.

To check the formatting of the book, run `make lint`.

To see the other available output formats, run `make`.

How to Contribute

We hope that if you use this material, you are also willing to contribute back to it. If you are new to open source, you might check out this How to Contribute to Open Source guide. Among other things, you'll learn about posting *Issues* that you'd like to see addressed, and issuing *Pull Requests* to merge your improvements back into GitHub.

If you do want to contribute either patches or new material, you will need to sign a Contributor Licensing Agreement (CLA). You'll be prompted to sign the CLA the first time you make a pull request.

The CLA is pretty straightforward: it establishes that (a) you have the right to contribute what you're contributing, and (b) what you contribute is available to everyone else under the same CC BY terms as the existing content. The CLA is a little unusual in that it explicitly calls out Elsevier's rights (which are the same as everyone's), but this does signal their intent to continue publishing textbooks based on the material.

You should also familiarize yourself with the guidelines for contributing.

If you'd like to contribute and are looking for something that needs attention, see the current Project Board. We'd also like to expand the set of topics/chapters beyond the initial set inherited from the 5th edition, so if you have ideas, we'd love to hear from you. Send email to `discuss@systemsapproach.org`, or better yet, join the forum.

Finally, in as much as this is an on-going effort, we will try to record and track our progress. For now, think of this as a poor-man's release notes. Additional information about work-in-progress can be found in the wiki.

Join Us

We hope you've gotten value out of *Computer Networks: A Systems Approach* over the years, and we're eager to have you join us in this new venture.

Larry Peterson & Bruce Davie
November 2019