COMPUTER NETWORK
NOTES
SCAD ENGNEERING COLLEGE.

Applications

Most people know the Internet through its applications: the World Wide Web, email, streaming audio and video, chat rooms, and music (file) sharing. The Web, for example, presents an intuitively simple interface. Users view pages full of textual and graphical objects, 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 objection a page is bound to an identifier for the next page to be viewed. This identifier, called aUniform Resource Locator (URL), is used to provide a way of identifying all the possiblepages that can be viewed from your web browser. For example, http://www.cs.princeton.edu/~llp/index.htmlis the URL for a page providing information about one of this book's authors: the stringhttp indicates that the HyperText Transfer Protocol (HTTP) should be used to downloadthe page, www.cs.princeton.edu is the name of the machine that serves thepage, and

/~llp/index.html

uniquely identifies Larry's home page at this site.

What mostWeb users are not aware of, however, is that by clicking on just one suchURL, as many as 17 messages may be exchanged over the Internet, and this assumes

Applications

There are a variety of different classes of video applications. One class of video applicationis video-on-demand, which reads a preexisting movie from disk and transmitsit over the network. Another kind of application is videoconferencing, which is in someways the more challenging (and, for networking people, interesting) case because it hasvery tight timing constraints. Just as when using the telephone, the interactions amongthe participants must be timely. When a person at one end gestures, then that actionmust be displayed at the other end as quickly as possible. Too much delay makes thesystem unusable. Contrast this with video-on-demand where, if it takes several secondsfrom the time the user starts the video until the first image is displayed, the service is stilldeemed satisfactory. Also, interactive video usually implies that video is flowing in both, while a video-on-demand application is most likely sending video in only onedirection. One pioneering example of a videoconferencing tool, developed in the early andmid-1990s, is vic.

one of a suite of conferencing tools designed at Lawrence Berkeley Laboratory and UC Berkeley. The others include a whiteboard application (wb) that allows users to sendsketches and slides to eachother, a visual audio tool called vat, and a session directory (sdr) that is used to create and advertise videoconferences. All these tools run on Unix—hence their lowercase names—and are freely available on the Internet.Many similar toolsare available for other operating systems. It is interesting to note that while video over theInternet is still considered to be in its relative infancy at the time of this writing (2006), that the tools to support video over IP have existed for well over a decade. Although they are just two examples, downloading pages from theWeb and participatingin a videoconference demonstrate the diversity of applications that can be builton top of the Internet, and hint at the complexity of the Internet's design. Starting fromthe beginning, and addressing one problem at time, the rest of this book explains howto build a network that supports such a wide range of applications. Chapter 9 concludesthe book by revisiting these two specific applications, as well as several others that havebecome popular on today's Internet.

Requirements

We have just established an ambitious goal for ourselves: to understand how to build acomputer network from the ground upOur approach to accomplishing this goal willbe to start from first principles, and then ask the kinds of questions we would naturallyask if building an actual network. At each step, we will use today's protocols to illustratevarious design choices available to us, but we will not accept these existing artifacts asgospel. Instead, we will be asking (and answering) the question of *why* networks aredesigned the way they are. While it is tempting to settle for just

understanding the wayit's done today, it is important to recognize the underlying concepts because networks areonstantly changing as the technology evolves and new applications are invented. It is our experience that once you understand the fundamental ideas, any new protocol thatyou are confronted with will be relatively easy to digest.
The first step is to identify the set of constraints and requirements that influencenetwork design. Before getting started, however, it is important to understand that the expectations you have of a network depend on your perspective:

■ An *application programmer* would list the services that his application needs, forexample, a guarantee that each message the application sends will be deliveredwithout error within a certain amount of time.
■ 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.

■ A *network provider* would list the characteristics of a system that is easy to administerand manage, for example, in which faults can be easily isolated andwhere it is easy to account for usage. This section attempts to distill these different perspectives into a high-level to the major considerations that drive network design, and in doing so, identifiesthe challenges addressed throughout the rest of this book.

1.2.1 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 selectmachines. In fact, for reasons of privacy and security, many private (corporate) networkshave 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 away that allows them the potential to connect all the computers in the world. A systemthat is designed to support growth to an arbitrarily large size is said to *scale*. Using theInternet as a model, this book addresses the challenge of scalability.

Links, Nodes, and Clouds

Network connectivity occurs at many different levels. At the lowest level, a network canconsist of two or more computers directly connected by some physical medium, such asa coaxial cable or an optical fiber. We call such a physical medium a *link*, and we refer to the computers it connects as *nodes*. (Sometimes a node is a more specialized pieceof hardware rather than a computer, but we overlook that distinction for the purposesof this discussion.) As illustrated in Figure 1.2, physical links are sometimes limited to apair of nodes (such a link is said to be *point-to-point*), while in other cases, more than twonodes may share a single physical link (such a link is said to be *multiple-access*).

Requirements

discrete blocks of data to each other. Think of these blocks of data as corresponding tosome piece of application data such as a file, a piece of email, or an image. We call eachblock of data either a *packet* or a *message*, and for now we use these terms interchangeably; we discuss the reason they are not always the same in Section 1.2.2. Packet-switched networks typically use a strategy called *store-and-forward*. As thename suggests, each node in a store-and-forward network first receives a complete packetover some link, stores the packet in its internal memory, and then forwards the completepacket to the next node. In contrast, a circuit-switched network first establishes adedicated circuit across a sequence of links and then allows the source node to send astream of bits across this circuit to a destination node. The major reason for using packetswitching rather than circuit switching in a computer network is efficiency, discussed inthe 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 tostore and forward packets) and the nodes on the outside of the cloud that *use* the network (they are commonly called *hosts*, and they support users and run application programs). Also note that the cloud in Figure 1.3 is one of the most important icons of computernetworking. In general, we use a cloud to denote any type of network,

whether it is asingle point-to-point link, a multiple-access link, or a switched network. Thus, wheneveryou see a cloud used in a figure, you can think of it as a placeholder for any of thenetworking technologies covered in this book. A second way in which a set of computers can be indirectly connected is shown inFigure 1.4. In this situation, a set of independent networks (clouds) are interconnectedto form an *internetwork*, or internet for short. We adopt the Internet's ofreferring to a generic internetwork of networks as a lowercase *i* internet, and the currentlyoperational TCP/IP Internet as the capital *I* Internet. A node that is connected to two ormore networks is commonly called a *router* or *gateway*, and it plays much the same roleas a switch—it forwards messages from one network to another. Note that an internetcan itself be viewed as another kind of network, which means that an internet can bebuilt from an interconnection of internets. Thus, we can recursively build arbitrarilylarge networks by interconnecting clouds to form larger clouds.

Just because a set of hosts are directly or indirectly connected to each other does notmean that we have succeeded in providing host-to-host connectivity. The final requirementis that each node must be able to state which of the other nodes on the networkit wants to communicate with. This is done by assigning an *address* to each node. Anaddress is a byte string that identifies a node; that is, the network can use a node's addressto distinguish it from the other nodes connected to the network. When a sourcenode 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

Network Architecture

In case you hadn't noticed, the previous section established a pretty substantial set ofrequirements for network design—a computer network must provide general, costeffective, fair, and robust connectivity among a large number of computers. As if thisweren't enough, networks do not remain fixed at any single point in time, but mustevolve to accommodate changes in both the underlying technologies upon which theyare based as well as changes in the demands placed on them by application programs. Designing a network to meet these requirements is no small task.

To help deal with this complexity, network designers have developed generalblueprints—usually called *network architectures*—that guide the design and implementationof networks. This section defines more carefully what we mean by a network architectureby introducing the central ideas that are common to all network architectures

Layering and Protocols

When a system gets complex, the system designer introduces another level of abstraction. The idea of an abstraction is to define a unifying model that can capture some importantaspect of the system, encapsulate this model in an object that provides an interface thatcan be manipulated by other components of the system, and hide the details of howthe object is implemented from the users of the object. The challenge is to identifyabstractions that simultaneously provide a service that proves useful in a large numberof situations and that can be efficiently implemented in the underlying system. This isexactly 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 ofthe network from application writers.

Abstractions naturally lead to layering, especially in network systems. The generalidea is that you start with the services offered by the underlying hardware, and thenadd a sequence of layers, each providing a higher (more abstract) level of service. Theservices provided at the high layers are implemented in terms of the services provided bythe 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 abstractionsandwiched between the application program and the underlying hardware, as illustratedin Figure 1.8. The layer immediately above the hardware in this case might provide hostto-host connectivity, abstracting away the fact that there may be an arbitrarily complexnetwork topology between any two hosts. The next layer up builds on the available hostto-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 nice features. First, it decomposes the problem of buildinga network into more manageable components. 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.

For example, a request/reply protocol would support operations by which an applicationan send and receive messages. An implementation of the HTTP protocol could supportan operation to fetch a page of hypertext from a remote server. An application such asa web browser would invoke such an operation whenever the browser needs to obtain anew page, for example, when the user clicks on a link in the currently displayed page. Second, a protocol defines a *peer interface* to its counterpart

(peer) on anothermachine. This second interface defines the form and meaning of messages exchangedbetween 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.

Consider what happens in Figure 1.11 when one of the application programs sends amessage to its peer by passing the message to protocol RRP. From RRP's perspective, themessage it is given by the application is an uninterpreted string of bytes. RRP does notcare that these bytes represent an array of integers, an email message, a digital image, orwhatever; it is simply charged with sending them to its peer. However, RRP must communicatecontrol information to its peer, instructing it how to handle the message whenit 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 usedamong peers to communicate with each other. As the name suggests, headers are usuallyattached to the front of a message. In some cases, however, this peer-to-peer controlinformation is sent at the end of the message, in which case it is called a *trailer*. Theexact format for the header attached by RRP is defined by its protocol specification. Therest of the message—that is, the data being transmitted on behalf of the application—iscalled the message's *body* or *payload*. We say that the application's data is *encapsulated* inthe new message created by protocol 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 wenow assume that HHP sends the message to its peer over some network, then whenthe message arrives at the destination host, it is processed in the opposite order: HHPfirst interprets the HHP header at the front of the message (i.e., takes whatever actionis 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 theRRP header that its peer attached, and passes the body of the message (but not theRRP header) up to the application program. The message passed up from RRP to theapplication on host 2 is exactly the same message as the application passed down to RRPon host 1; the application does not see any of the headers that have been attached to it toimplement the lower-level communication services.

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

Multiplexing and Demultiplexing

Recall from Section 1.2.2 that a fundamental idea of packet switching is to multiplexmultiple flows of data over a single physical link. This same idea applies up and downthe protocol graph, not just to switching nodes. In Figure 1.11, for example, we canthink of RRP as implementing a logical communication channel, with messages from**26 1 Foundation**two different applications multiplexed over this channel at the source host and thendemultiplexed back to the appropriate application at the destination host.

Practically speaking, all this means is that the header that RRP attaches to its messagescontains 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 sourcehost, RRP includes the appropriate demux key in its header. When the message is deliveredto RRP on the destination host, it strips its header, examines the demux key, anddemultiplexes the message to the correct application. RRP is not unique in its support for multiplexing; nearly every protocol implementsthis mechanism. For example, HHP has its own demux key to determine whichmessages to pass up to RRP and which to pass up to MSP. However, there is no uniformagreement among protocols—even those within a single network architecture—on exactlywhat constitutes a demux key. Some protocols use an 8-bit field (meaning they cansupport only 256 high-level protocols), and others use 16- or 32-bit fields. Also, someprotocols have a single demultiplexing field in their header, while others have pair ofdemultiplexing fields. In the former case, the same demux key is used on both sides ofthe

communication, while in the latter case, each side uses a different key to identify thehigh-level protocol (or application program) to which the message is to be delivered.

OSI Architecture

The ISO was one of the first organizations to formally define a common way to connectcomputers. Their architecture, called the *Open Systems Interconnection (OSI)* architectureand illustrated in Figure 1.13, defines a partitioning of network functionality into sevenlayers, where one or more protocols implement the functionality assigned to a givenlayer. In this sense, the schematic given in Figure 1.13 is not a protocol graph, per se, but rather a *reference model* for a protocol graph. The ISO, usually in conjunction witha second standards organization known as the International Telecommunications Union (ITU),1 publishes a series of protocol specifications based on the OSI architecture. This is sometimes called the "X dot" series since the protocols are given names like X.25, X.400, X.500, and so on. Starting at the bottom and working up, the *physical* layer handles the transmissionof raw bits over a communications link. The *data link* layer then collects a stream of bitsinto a larger aggregate called a *frame*. Network adaptors, along with device drivers runningin the node's OS, typically implement the data link level. This means that frames, not raw bits, are actually delivered to hosts. The *network* layer handles routing amongnodes within a packet-switched network. At this layer, the unit of data exchanged among is typically called a *packet* rather than a frame, although they are fundamentally

The lower three layers are implemented on all network nodes, includingswitches within the network and hosts connected along the exterior of the network. The*transport* layer then implements what we have up to this point been calling a process-toprocesschannel. Here, the unit of data exchanged is commonly called a *message* ratherthan a packet or a frame. The transport layer and higher layers typically run only on theend hosts and not on the intermediate switches or routers.

There is less agreement about the definition of the top three layers. Skipping aheadto the top (seventh) layer, we find the *application* layer. Application layer protocols includethings like the File Transfer Protocol (FTP), which defines a protocol by whichfile transfer applications can interoperate. Below that, the *presentation* layer is concernedwith the format of data exchanged between peers, for example, whether an integer is 16, 32, or 64 bits long and whether the most significant byte is transmitted first or last, orhow a video stream is formatted. Finally, the *session* layer provides a name space that isused to tie together the potentially different transport streams that are part of a single

1.5 Performance

Up to this point, we have focused primarily on the functional aspects of a network. Likeany computer system, however, computer networks are also expected to perform well. This is because the effectiveness of computations distributed over the network oftendepends directly on the efficiency with which the network delivers the computation'sdata. While the old programming adage "first get it right and then make it fast" is validin many settings, in networking it is usually 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 thenumber of bits that can be transmittedover the network in a certain period oftime. For example, a network might havea bandwidth of 10 million bits/second (Mbps), meaning that it is able to deliver10 million bits every second. It issometimes useful to think of bandwidthin terms of how long it takes to transmiteach bit of data. On a 10-Mbps network, for example, it takes 0.1 microsecond (μs) to transmit each bit. While you can talk about the bandwidthof the network as a whole, sometimesyou want to be more precise, focusing, for example, on the bandwidth**Bandwidth and Throughput***Bandwidth* and *throughput* are twoof the most confusing terms used innetworking. While we could try togive you a precise

definition of eachterm, it is important that you knowhow other people might use them andfor you to be aware that they are oftenused interchangeably. First of all, bandwidth is literally a measure of thewidth of a frequency band. For example, a voice-grade telephone line supportsa frequency band ranging from300 to 3,300 Hz; it is said to have abandwidth of 3,300 Hz − 300 Hz =3,000 Hz. If you see the word "bandwidth" used in a situation in whichit is being measured in hertz, then itprobably refers to the range of signalsthat can be accommodated. When we talk about the bandwidthof a communication link, wenormally refer to the number of bitsper second that can be transmittedon the link. We might say that thebandwidth of an Ethernet is 10 Mbps. A useful distinction might be made, however, between the bandwidth thatis available on the link and the numberof bits per second that we can actuallytransmit over the link in practice. We tend to use the word "throughput" to refer to the *measured performance*of a system. Thus, because ofof a single physical link or a logical processto-process channel. At the physical level, bandwidth is constantly improving, withno end in sight. Intuitively, if you thinka second of time as a distance youcould measure with a ruler, and bandwidthas how many bits fit in that distance, then you can think of each bitas a pulse of some width. For example, each bit on a 1-Mbps link is 1 μswide, while each bit on a 2-Mbps linkis 0.5 μs wide, as illustrated in Figure1.19. The more sophisticated thetransmitting and receiving technology, thenarrower each bit can become, and thus, the higher the bandwidth. For logicalprocess-to-process channels, bandwidthis also influenced by other factors, includinghow many times the softwarethat implements the channel has to handle, and possibly transform, each bit of data.

Third, there may be queuing delays insidethe network, since packet switches generally need to store packets for some time beforeforwarding them on an outbound link, as discussed in Section 1.2.2. So, we could definethe total latency as

Latency = Propagation + Transmit + QueuePropagation = Distance/SpeedOfLight
Transmit = Size/Bandwidth
where Distance is the length of the wire over which the data will travel, Speed-OfLight 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 themessage contains only one bit and we are talking

How Big Is a Mega?

There are several pitfalls you need tobe aware of when working with thecommon units of networking—MB, Mbps, KB, and Kbps. The first is todistinguish carefully between bits andbytes. Throughout this book, we alwaysuse a lowercase *b* for bits and acapital *B* for bytes. The second is tobe sure you are using the appropriatedefinition of mega (M) and kilo (K). *Mega*, for example, can mean either2^{20} or 10^6. Similarly, *kilo* can be either2^{10} or 10^3. What is worse, in networkingwe typically use both definitions. Here's why. Network bandwidth, which isoften specified in terms of Mbps, istypically governed by the speed of theclock that paces the transmission ofthe bits. A clock that is running at10 MHz is used to transmit bits at10 Mbps. Because the *mega* in MHzmeans 10^6 hertz, Mbps is usually alsodefined as 10^6 bits per second. (Similarly, Kbps is 10^3 bits per second.) On the other hand, when we talkabout a message that we want to transmit, we often give its size in kilobytes. The delay × bandwidth product isimportant to know when constructinghigh-performance networks because it correspondsto how many bits the sendermust transmit before the first bit arriveswait for a signal—the sender will not fullyutilize the network. Note that most of the time we areinterested in the RTT scenario, which wesimply refer to as the delay × bandwidthproduct, without explicitly saying that this

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 illustrated in Figure 2.11. Recallfrom Chapter 1 that we are focusing on packet-switched networks, which means thatblocks of data (called frames at this level), not bitstreams, are exchanged between nodes. It is the network adaptor that enables the nodes to exchange frames. When node Awishes to transmit a frame to node B, it tells its adaptor to transmit a frame from thenode's memory. This results in a sequence of bits being sent over the link. The adaptoron node B then collects together the sequence of bits arriving on the link and depositsthe corresponding frame in B's memory. Recognizing exactly what set of bits constitutea frame—that is, determining where the frame begins and ends—is the central challengefaced by the adaptor. are several ways to address the framing problem. This section uses severaldifferent protocols to illustrate the various points in the design space. Note that while weframing in the context of point-to-point links, the problem is a fundamental onethat must also be addressed in multiple-access networks like Ethernet and token rings.

2.3.1 Byte-Oriented Protocols (PPP)

One of the oldest approaches to framing—it has its roots in connecting terminals tomainframes—is to view each frame as a collection of bytes (characters) rather than acollection of bits. Such a *byte-oriented* approach is exemplified by older protocols such asthe Binary Synchronous Communication (BISYNC) protocol developed by IBM in thelate 1960s, and the Digital Data Communication Message Protocol (DDCMP) used inDigital Equipment Corporation's DECNET. The more recent and widely used Point-to-Point Protocol (PPP) provides another example of this approach.
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. BISYNC uses special characters known as *sentinel characters* to indicate whereframes start and end. The beginning of a frame is denoted by sending a special SYN (synchronization) character. The data portion of the frame is then contained betweentwo more special characters: STX (start of text) and ETX (end of text). The SOH (startof header) field serves much the same purpose as the STX field. The problem with thesentinel approach, of course, is that the ETX character might appear in the data portionof the frame. BISYNC overcomes this problem by "escaping" the ETX character bypreceding it with a data-link-escape (DLE) character whenever it appears in the body ofa frame; the DLE character is also escaped (by preceding it with an extra DLE) in theframe body. (C programmers may notice that this is analogous to the way a quotationmark is escaped by the backslash when it occurs inside a string.) This approach is oftencalled *character stuffing* because extra characters are inserted in the data portion of theframe. The frame format also includes a field labeled cyclic redundancy check (CRC) thatis used to detect

transmission errors; various algorithms for error detection are presentedin Section 2.4. Finally, the frame contains additional header fields that are used for, among other things, the link-level reliable delivery algorithm.

Byte-Counting Approach

As every Computer Sciences 101 studentknows, the alternative to detecting theend of a file with a sentinel value is toinclude the number of items in the fileat the beginning of the file. The sameis true in framing—the number of bytescontained in a frame can be included

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 approachsimply as *clock-based framing*. SONET was first proposed by Bell CommunicationsResearch (Bellcore), and then developed under the American National StandardsInstitute (ANSI) for digital transmission over optical fiber; it has since been adopted bythe ITU-T. Who standardized what and when is not the interesting issue, though. Thething to remember about SONET is that it is the dominant standard for long-distancetransmission of data over optical networks. An important point to make about SONET before we go any further is that thefull specification is substantially larger than this book. Thus, the following discussionwill necessarily cover only the high points of the standard. Also, SONET addresses boththe framing problem and the encoding problem. It also addresses a problem that is veryimportant for phone companies—the multiplexing of several low-speed links onto onehigh-speed link. We begin with framing and discuss the other issues following. As with the previously discussed framing schemes, a SONET frame has some specialinformation that tells the receiver where the frame starts and ends. However, that isabout as far as the similarities go. Notably, no bit stuffing is used, so that a frame's lengthdoes not depend on the data being sent. So the question to ask is, "How does the receiverknow where each frame starts and ends?" We consider this question for the lowest-speedSONET link, which is known as STS-1 and runs at 51.84 Mbps. An STS-1 frame isshown in Figure 2.16. It is arranged as nine rows of 90 bytes each, and the first 3 bytesof each row are overhead, with the rest being available for data that is being transmittedover the link. The first 2 bytes of the frame contain a special bit pattern, and it is thesebytes that enable the receiver to determine where the frame starts. However, since bitstuffing is not used, there is no reason why this pattern will not occasionally turn up inthe payload portion of the frame. To guard against this, the receiver looks for the specialbit pattern consistently, hoping to see it appearing once every 810 bytes, since each frameis $9 \times 90 = 810$ bytes long. When the special pattern turns up in the right place enoughtimes, the receiver concludes that it is in sync and can then interpret the frame correctly. One of the things we are not describing due to the complexity of SONET is thedetailed use of all the other overhead bytes. Part of this complexity can be attributed tothe 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 weare instead focusing on the fact that we can lease a SONET link from them and then use

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 approachsimply as *clock-based framing*. SONET was first proposed by Bell CommunicationsResearch (Bellcore), and then developed under the American National StandardsInstitute (ANSI) for digital transmission over optical fiber; it has since been adopted bythe ITU-T. Who standardized what and when is not the interesting issue, though. Thething to remember about SONET is that it is the dominant standard for long-distancetransmission of data over optical networks. An important point to make about SONET before we go any further is that thefull specification is substantially larger than this book. Thus, the following discussionwill necessarily cover only the high points of the standard. Also, SONET addresses boththe 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 onehigh-speed link. We begin with framing and discuss the other issues following. As with the previously discussed framing schemes, a SONET frame has some specialinformation that tells the receiver where the frame

starts and ends. However, that isabout 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 receiverknow where each frame starts and ends?" We consider this question for the lowest-speedSONET link, which is known as STS-1 and runs at 51.84 Mbps. An STS-1 frame isshown in Figure 2.16. It is arranged as nine rows of 90 bytes each, and the first 3 bytesof each row are overhead, with the rest being available for data that is being transmittedover the link. The first 2 bytes of the frame contain a special bit pattern, and it is thesebytes that enable the receiver to determine where the frame starts. However, since bitstuffing is not used, there is no reason why this pattern will not occasionally turn up inthe payload portion of the frame. To guard against this, the receiver looks for the specialbit pattern consistently, hoping to see it appearing once every 810 bytes, since each frameis $9 \times 90 = 810$ bytes long. When the special pattern turns up in the right place enoughtimes, the receiver concludes that it is in sync and can then interpret the frame correctly. One of the things we are not describing due to the complexity of SONET is thedetailed use of all the other overhead bytes. Part of this complexity can be attributed tothe 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 weare instead focusing on the fact that we can lease a SONET link from them and then usethis link to build our own packet-switched network.) Additional complexity comes fromthe 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 isused for maintenance.

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 arerare, especially on optical links, some mechanism is needed to detect these errors so thatcorrective action can be taken. Otherwise, the end user is left wondering why the C programthat 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 twonotable examples that were developed for use in punch card readers and when storingdata on magnetic disks and in early core memories. This section describes some of theerror detection techniques most commonly used in networking. Detecting errors is only one part of the problem. The other part is correcting errorsonce detected. There are two basic approaches that can be taken when the recipient ofa message detects an error. One is to notify the sender that the message was corruptedso that the sender can retransmit a copy of the message. If bit errors are rare, then in allprobability the retransmitted copy will be error free. Alternatively, there are some typesof error detection algorithms that allow the recipient to reconstruct the correct messageeven after it has been corrupted; such algorithms rely on *error correcting codes*, discussedbelow. One of the most common techniques for detecting transmission errors is a known as the *cyclic redundancy check (CRC)*. It is used in nearly all the link-levelprotocols discussed in the previous section—for example, HDLC, DDCMP—as well asin the CSMA and token ring protocols described later in this chapter. Section 2.4.3 outlinesthe basic CRC algorithm. Before discussing that approach, we consider two simplerschemes that are also widely used: *two-dimensional parity* and *checksums*. The former isused by the BISYNC protocol when it is transmitting ASCII characters (CRC is used asthe error code when BISYNC is used to transmit EBCDIC), and the latter is used byseveral Internet protocols. The basic idea behind any error detection scheme is to add redundant informationto 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 areidentical 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 redundantbits for an n-bit message. Second, many errors will go undetected—any error thathappens to corrupt the same bit positions in the first and second copies of the message. Fortunately, we can do a lot better than this simple scheme. In general, we canprovide quite strong error detection capability while sending only k redundant bits for an n-bit message, where $k \_ n$. On an Ethernet, for example, a frame carrying up to12,000 bits (1,500 bytes) of data requires only a 32-bit CRC code,

or as it is commonlyexpressed, uses CRC-32. Such a code will catch the overwhelming majority of errors, aswe will see below. We say that the extra bits we send are redundant because they add no new informationto the message. Instead, they are derived directly from the original message usingsome well-defined algorithm. Both the sender and the receiver know exactly what thatalgorithm is. The sender applies the algorithm to the message to generate the redundantbits.

Two-Dimensional Parity

Two-dimensional parity is exactly what the name suggests. It is based on "simple" (onedimensional parity, which usually involves adding one extra bit to a 7-bit code to balancethe number of 1s in the byte. For example, odd parity sets the eighth bit to 1 if neededto give an odd number of 1s in the byte, and even parity sets the eighth bit to 1 ifneeded to give an even number of 1s in the byte. Two-dimensional parity does a similarcalculation for each bit position across each of the bytes contained in the frame. Thisresults in an extra parity byte for the entire frame, in addition to a parity bit for each byte. Figure 2.19 illustrates how two-dimensional even parity works for an example framecontaining 6 bytes of data. Notice that the third bit of the parity byte is 1 since there isan odd number of 1s in the third bit across the 6 bytes in the frame. It can be shown
two-dimensional parity catches all 1-, 2-, and 3-bit errors, and most 4-bit errors. Inthis case, we have added 14 bits of redundant information to a 42-bit message, and yetwe have stronger protection against common errors than the "repetition code" describedabove.

Internet Checksum Algorithm

A second approach to error detection is exemplified by the Internet checksum. Althoughit is not used at the link level, it nevertheless provides the same sort of functionality asCRCs and parity, so we discuss it here. We will see examples of its use in Sections 4.1, 5.1, and 5.2. The idea behind the Internet checksum is very simple—you add up all the wordsthat are transmitted and then transmit the result of that sum. The result is called thechecksum. The receiver performs the same calculation on the received data and comparesthe result with the received checksum. If any transmitted data, including the checksumitself, is corrupted, then the results will not match, so the receiver knows that an erroroccurred You can imagine many different variations on the basic idea of a checksum. Theexact scheme used by the Internet protocols works as follows. Consider the data beingchecksummed as a sequence of 16-bit integers. Add them together using 16-bit onescomplement arithmetic (explained below) and then take the ones complement of theresult. That 16-bit number is the checksum. In ones complement arithmetic, a negative integer $-x$ is represented as the complementof x, that is, each bit of x is inverted. When adding numbers in ones complementarithmetic, 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-bitintegers: +5 is 0101, so −5 is 1010; +3 is 0011, so −3 is 1100. If we add 1010 and1100 ignoring the carry, we get 0110. In ones complement arithmetic, the fact that thisoperation 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 invertingthe bits in 1000), as we would expect.

The following routine gives a straightforward implementation of the Internet'schecksum algorithm. The count argument gives the length of buf measured in16-bit units. The routine assumes that buf has already been padded with 0s to a 16-bitboundary.

```
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, ratherthan the twos complement that is used in most machines. Note the if statement insidethe while loop. If there is a carry into the top 16 bits of sum, then we increment sumjust as in the previous example. Compared to our repetition code, this algorithm scores well for using a small numberof redundant bits—only 16 for a message of any length—but it does not score extremelywell for strength of error detection. For example, a pair of single-bit errors, oneof which increments a word and one of which decrements another word by the sameamount, will go undetected. The reason for using an algorithm like this in spite of itsrelatively weak protection against errors (compared to a CRC, for example) is simple: This algorithm is much easier to implement in software. Experience in the ARPANETsuggested that a checksum of this form was adequate. One reason it is adequate is thatthis checksum is the last line of defense in an end-to-end protocol; the majority of errorsare picked up by stronger error detection algorithms, such as CRCs, at the link level. **96 2 Direct Link Networks**

Cyclic Redundancy Check

It should be clear by now that a major goal in designing error detection algorithms is tomaximize 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. Forexample, a 32-bit CRC gives strong protection against common bit errors in messagesthat are thousands of bytes long. The theoretical foundation of the cyclic redundancycheck is rooted in a branch of mathematics called finite fields. While this may sounddaunting, 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 representedby a polynomial by using the value of each bit in the message as the coefficient

Simple Probability Calculations

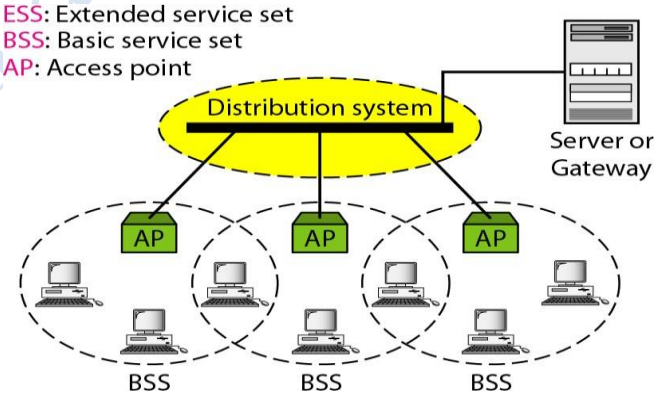
When dealing with network errorsand other unlikely (we hope) events, we often have use for simple backof-the-envelope probability estimates. A useful approximation here is thatif two independent events have *small*probabilities p and q, then the probabilityof either event is $p + q$; the exactanswer is $1 - (1 - p)(1 - q) = p + q - pq$. For $p = q = .01$, this estimateis .02, while the exact value is.0199. For a simple application of this, suppose that the per-bit error rate ona link is 1 in 10^7. Now suppose we areinterested in estimating the probabilityof at least one bit in a 10,000-bitpacket being errored. Using the aboveapproximation repeatedly over all thebits, we can say that we are interestedin the probability of either the firstbit being errored, or the second bit, or the third, and so on. Assuming biterrors are all independent (which theyaren't), we can therefore estimate thatfor each term in the polynomial, startingwith the most significant bit to representthe highest-order term. For example, an 8-bit message consisting of the bits10011010 corresponds to the polynomial$M(x) = 1 \times x^7 + 0 \times x^6 + 0 \times x^5 + 1 \times x^4 + 1 \times x^3 + 0 \times x^2 + 1 \times x^1 + 0 \times x^0$

$= x^7 + x^4 + x^3 + x^1$

We can thus think of a sender and areceiver as exchanging polynomials witheach other. For the purposes of calculating aCRC, a sender and receiver have to agreeon a *divisor* polynomial, $C(x)$. $C(x)$ isa polynomial of degree k. For example, suppose $C(x) = x^3 + x^2 + 1$. In thiscase, $k = 3$. The answer to the question "Where did $C(x)$ come from?" is, in most practical cases, "You look it upin a bookIn fact, the choice of $C(x)$ has a significant impact on what typesof errors can be reliably detected

Ethernet (802.3)

The Ethernet is easily the most successful local area networking technology of the 20yearsDeveloped in the mid-1970s by researchers at the Xerox Palo Alto Research Center (PARCthe Ethernet working example of the more general carrier sense, multipleaccess with collision detect (CSMA/CD) local area network technology. As indicated by the CSMA name, the Ethernet is a multiple-access network, meaningthat a set of nodes send and receive 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 idleand a busy link, and "collision detect" means that a node listens as it transmits and cantherefore detect when a frame it is transmitting has interfered (collided) with a frametransmitted by another nodeThe Ethernet has its roots in an early packet radio network, called Aloha, developedat the University of Hawaii to support computer communication across the HawaiianIslands. Like the Aloha network, the fundamental problem faced by the Ethernet is howto mediate access to a shared medium fairly and efficiently (in Aloha the medium wasthe atmosphere, while in Ethernet the medium is a coax cable). That is, the core idea inboth Aloha and the Ethernet is an algorithm that controls when each node can transmit. Digital Equipment Corporation and Intel Corporation joined Xerox to define a10-Mbps Ethernet standard in 1978. This standard then formed the basis for IEEEstandard 802.3. With one exception that we will see in Section 2.6.2, it is fair to viewthe 1978 Ethernet standard as a proper subset of the 802.3 standard; 802.3 additionallydefines a much wider collection of physical media over which Ethernet can operate, andmore recently, it has been extended to include a 100-Mbps version called Fast Ethernet, and a 1,000-Mbps version called Gigabit Ethernet. The rest of this section focuses on the10-Mbps Ethernet since it is typically used in multiple-access mode, and we are interestedin how multiple hosts share a single link. Both 100- and 1,000-Mbps Ethernets aredesigned to be used in full-duplex, point-to-point configurations, which means that theyare typically used in switched networks, as described in the next chapter.



Physical Properties

An Ethernet segment is implemented on a coaxial cable of up to 500 m. This cable is similarto the type used for cable TV, except that it typically has an impedance of 50 ohmsinstead of cable TV's 75 ohms. Hosts connect to an Ethernet segment by tapping intoit; taps must be at least 2.5 m apart. A *transceiver*—a small device directly attached tothe tap—detects when the line is idle and drives the signal when the host is transmitting. It also receives incoming signals. The transceiver is, in turn, connected to an Ethernetadaptor, which is plugged into the host.

All the logic that makes up the Et**Figure 2.27 Ethernet transceiver and adaptor.** tocol, as described in this section, is implemented in the adaptor (not the transceiver). This configuration is shown in Figure 2.27. Multiple Ethernet segments can be joined together by

repeaters. A repeater is a devicethat forwards digital signals, much like an amplifier forwards analog signals. in the same way as you would with 10Base5 cable. With 10Base2, a T-joint is splicedinto the cable. In effect, 10Base2 is used to daisy-chain a set of hosts together. With10BaseT, the common configuration is to have several point-to-point segments comingout of a multiway repeater, sometimes called a *hub*, as illustrated in Figure 2.29.

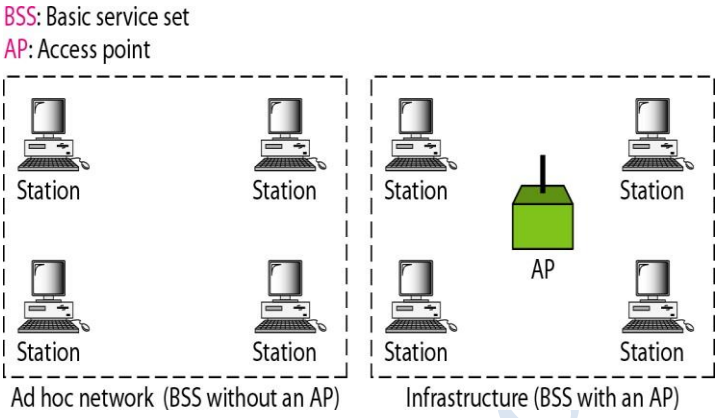


Figure 2.29 Ethernet hub.

Multiple100-Mbps Ethernet segments can also be connected by a hub, but the same is not trueof 1,000-Mbps segments. 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 connectedin a star configuration by a hub, data transmitted by any one host on that Ethernetreaches all the other hosts. This is the good news. The bad news is that all these hosts arecompeting for access to the same link, and as a consequence, they are said to be in thesame *collision domain.*

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

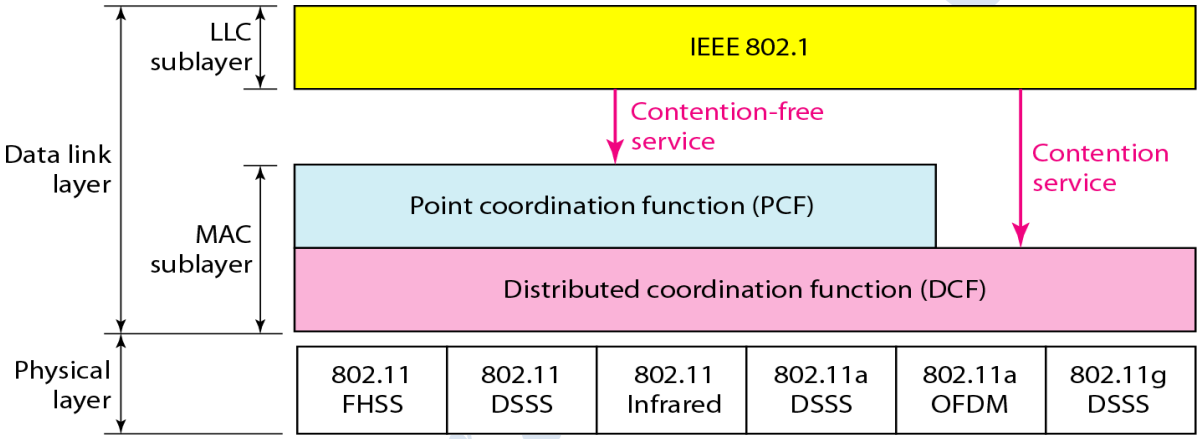
Finally, each frame includes a 32-bit CRC. Like the HDLC protocoldescribed in Section 2.3.2, the Ethernet is a bit-oriented framing protocol. Notethat from the host's perspective, an Ethernet frame has a 14-byte header: two 6-byte addressesand a 2-byte type field. The sending adaptor attaches the preamble, CRC, andpostamble before transmitting, and the receiving adaptor removes them. The frame format just described is taken from the Digital-Intel-Xerox Ethernetstandard. The 802.3 frame format is exactly the same, except it substitutes a 16-bit length**Figure 2.30 Ethernet frame format.** field for the 16-bit type field. 802.3 is usually paired with an encapsulation standard thatdefines a type field used to demultiplex incoming frames. This type field is the first thingin the data portion of the 802.3 frames, that is, it immediately follows the 802.3 header. Fortunately, since the Ethernet standard has avoided using any type values less than 1,500 (the maximum length found in an 802.3 header), and the type and length fields are inthe same location in the header, it is possible for a single device to accept both formats, and for the device driver running on the host to interpret the last 16 bits of the headeras either a type or a length. In practice, most hosts follow the Digital-Intel-Xerox formatand interpret this field as the frame's type.

Addresses

Each host on an Ethernet—in fact, every Ethernet host in the world—has a unique Ethernetaddress. Technically, the address belongs to the adaptor, not the host; it is usuallyburned into ROM. Ethernet addresses are typically printed in a form humans can readas a sequence of six numbers separated by colons. Each number corresponds to 1 byteof the 6-byte address and is given by a pair of hexadecimal digits, one for each of the4-bit nibbles in the byte; leading 0s are dropped. For example,

8:0:2b:e4:b1:2 is thehuman-readable representation of Ethernet address00001000 00000000 00101011 11100100 10110001 00000010To ensure that every adaptor gets a unique address, each manufacturer of Ethernetdevices is allocated a different prefix that must be prepended to the address on everyadaptor they build. For example, Advanced Micro Devices has been assigned the 24-bitprefix x080020 (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 tothat Ethernet. Each adaptor recognizes those frames addressed to its address and passesonly 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 thenormal mode.)
Similarly, an address that has the first bit set to 1 but is not thebroadcast address is called a *multicast* address. A given host can program its adaptor toaccept some set of multicast addresses. Multicast addresses are used to send messages tosome subset of the hosts on an Ethernet (e.g., all file servers).

■ 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 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 frameimmediately; there is no negotiation with the other adaptors. The upper bound of1,500 bytes in the message means that the adaptor can occupy the line for only a fixedlength of time. When an adaptor has a frame to send and the line is busy, it waits for the line to goidle and then transmits immediately.[4] The Ethernet is said to be a *1-persistent* protocolbecause an adaptor with a frame to send transmits with probability 1 whenever a busyline 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 behindchoosing a $p < 1$ is that there might be multiple adaptors waiting for the busy line tobecome 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 threeadaptors can be waiting to transmit and the odds are that only one will begin transmittingwhen the line becomes idle. Despite this reasoning, an Ethernet adaptor always transmitsimmediately after noticing that the network has become idle and has been very effectivein doing so. To complete the story about *p*-persistent protocols for the case when $p < 1$, youmight wonder how long a sender that loses the coin flip (i.e., decides to defer) has to waitbefore it can transmit. The answer for the Aloha network, which originally developed thisstyle of protocol, was to divide time intodiscrete slots, with

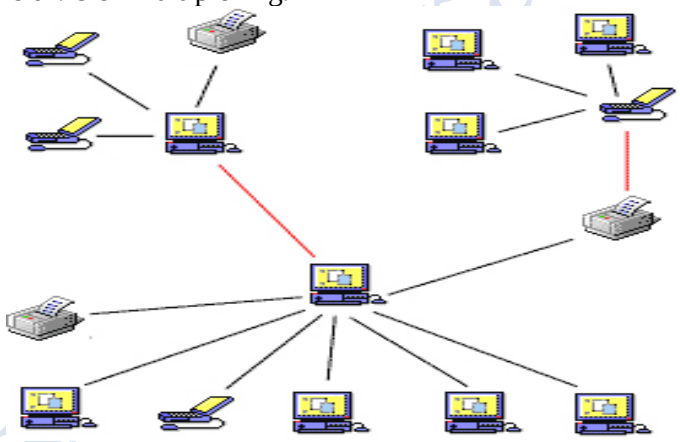
each slot corresponding tothe length of time it takes to transmit a full frame. Whenever a node has a frame tosend and it senses an empty (idle) slot, it transmits with probability *p* and defers until the next slot with probability $q = 1 - p$. If that next slot is also empty, the node againdecides to transmit or defer, with probabilities *p* and *q*, respectively. If that next slot isnot empty—that is, some other station has decided to transmit—then the node simplywaits 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

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, Bluetootcan be used to connect a mobile phone to a headset, or a notebook compuer to a printer. Roughly speaking, Bluetooth is a more convenient alternative to connecting twodevices with a wire. In such applications, it is not necessary to provide much range orbandwidth. This is fortunate for some of the target battery-powered devices, since it isimportant that they not consume much power.

Bluetooth operates in the license-exempt band at 2.45 GHz. It has a range of onlyabout 10 m. For this reason, and because the communicating devices typically belong toone individual or group, Bluetooth is sometimes categorized as a personal area network(PAN). Version 2.0 provides speeds up to 2.1 Mbps. Power consumption is low.Bluetooth is specified by an industry consortium called the Bluetooth Special Interes Group. It specifies an entire suite of protocols, going beyond the link layer to defineapplication 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 profilegives a mobile computer access to a wired LAN in the manner of 802.11, although thiswas not Bluetooth's original goal. The but excludes the application protocols.

The basic Bluetooth network configuration, called a *piconet*, consists of a master deviceand up to seven slave devices, as in Figure 2.40. Any communication is between themaster and a slave; the slaves do not communicate directly with each other. Because slaveshave a simpler role, their Bluetooth hardware and software can be simpler and cheaper.Since Bluetooth operates in an license-exempt band, it is required to use a spreadspectrum technique (as discussed in Section 2.1.2) to deal with possible interferencein the band. It uses frequency hopping with 79 *channels* (frequencies), using each for625 μm at a time. This provides a natural time slot for Bluetooth to use for synchronous time division multiplexing.



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 theprevious slot, thereby preventing any contention between the slave devices.A slave device can be *parked*: set to an inactive, low-power state. A parked devicecannot communicate on the piconet; it can only be reactivated by the master. A piconetcan have up to 255 parked devices in addition to its active slave devices.

ZigBee is a newer technology that competes with Bluetooth to some extent. Devisedby the ZigBee alliance and standardized as IEEE 802.15.4, it is designed for situationswhere the bandwidth requirements are low and power consumption must bevery low to give very long battery life. It is also

intended to be simpler and cheaper thanBluetooth, making it financially feasible to incorporate in cheaper devices such as a wallswitch that wirelessly communicates with a ceiling-mounted fan.

2.8.2 Wi-Fi (802.11)

This section takes a closer look at a specific technology centered around the emergingIEEE 802.11 standard, also known as *Wi-Fi*.6Wi-Fi is technically a trademark, owned a trade group called the Wi-Fi alliance, that certifies product compliance with 802.11.Like its Ethernet and token ring siblings, 802.11 is designed for use in a limited geographicalarea (homes, office buildings, campuses), and its primary challenge is to mediateaccess to a shared communication medium—in this case, signals propagating throughspace. 802.11 supports additional features (e.g., time-bounded services, power management,and security mechanisms), but we focus our discussion on its base functionality.

Physical Properties

802.11 runs over six different physical layer protocols (so far). Five are based on spreadspectrum radio, and one on diffused infrared (and is of historical interest only at thispoint). The fastest runs at a maximum of 54 Mbps.The original 802.11 standard defined two radio-based physical layers standards,one using frequency hopping (over 79 1-MHz-wide frequency bandwidths) and theother usingdirect sequence (with an 11-bit chipping sequence). Both provide up to2 Mbps. Thn physical layer standard 802.11b was added. Using a variant of directsequence, 82.11b provides up to 11 Mbps. These three standards run in the licenseexempt2.4 GHz freuency band of the electromagnetic spectrum. Then came 802.11a,which delivers up to 54 Mbps using a variant of FDM called *orthogonal frequency divisionmultiplexing (OFDM)*. 802.11a runs in the license-exempt 5-GHz band. On one hand,this band is less used, so there is less interference. On the other hand, there is more absorptionof the signal and it is limited to almost line of sight. The most recent standardis 802.11g, which is backward compatible with 802.11b (and returns to the 2.4-GHzband). 802.11g uses OFDM and delivers up to 54 Mbps. It is common for commercialproducts to support all three of 802.11a, 802.11b, and 802.11g, which not only ensurescompatibility with any device that supports any one of the standards, but also makes itpossible for two such products to choose the highest bandwidth option for a particular environment.

Collision Avoidance

At first glance, it might seem that a wireless protocol would follow the same algorithm asthe Ethernet—wait until the link becomes idle before transmitting and back off shoulda collision occur—and to a first approximation, this is what 802.11 does. The additionalcomplication for wireless is that, while a node on an Ethernet receives every other node'stransmissions, a node on an 802.11 network may be too far from certain other nodes toreceive their transmissions (and vice versa).

Consider the situation depicted in Figure 2.41, where A and C are both withinrange of B but not each other. Suppose both A and C want to communicate with B andso they each send it a frame. A and C are unaware of each other since their signals donot carry that far. These two frames collide with each other at B, but unlike an Ethernet, field contains three subfields of interest (not shown): a 6-bit Type field that indicateswhether the frame carries data, is an RTS or CTS frame, or is being used by the scanningalgorithm; and a pair of 1-bit fields—called ToDS and FromDS—that are described below.

The peculiar thing about the 802.11 frame format is that it contains four, ratherthan two, addresses. How these addresses are interpreted depends on the settings of theToDS and FromDS bits in the frame's Control field. This is to account for the possibilitythat the frame had to be forwarded across the distribution system, which wouldmean that the original sender is not necessarily the same as the most recent transmittingnode. Similar reasoning applies to the destination address. In the simplest case, whenone 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 DSbits are set to 1, indicating that the message went from a wireless node onto the distributionsystem, and then from the distribution system to another wireless node. With both bits set, Addr1 identifies the ultimate destination, Addr2 identifies the immediatesender (the one that forwarded the frame from the distribution system to the ultimate destination), Addr3 identifies the intermediate destination (the one that accepted therame from a wireless node and forwarded it across the distribution system), and Addr4identifies the original

source. In terms of the example given in Figure 2.43, Addr1corresponds to E, Addr2 identifies AP-3, Addr3 corresponds to AP-1, and Addr4identifies A.

2.8.3 WiMAX (802.16)

WiMAX, which stands for Worldwide Interoperability for Microwave Access, was designedby theWiMAX Forum and standardized as IEEE 802.16. It was originally conceivedas a last-mile technology (Section 2.1.2). InWiMAX's case that "mile" is typically1 to 6 miles, with a maximum of about 30 miles, leading to WiMAX being classifiedas a metropolitan area network (MAN). In keeping with a last-mile role, WiMAX doesnot incorporate mobility at the time of this writing, although efforts to add mobility are nearing completion as IEEE 802.16e. Also in keeping with the last-mile niche,WiMAX'sclient systems, called *subscriber stations*, are assumed to be not end-user computing devices,but rather systems that multiplex all the communication of the computing devicesbeing used in a particular building. WiMAX provides up to 70 Mbps to a single subscriber station.In order to adapt to different frequency bands and different conditions, WiMAXdefines several physical layer protocols. The original WiMAX physical layer protocolis designed to use frequencies in the 10- to 66-GHz range. In this range waves travel in straight lines, so communication is limited to line-of-sight (LOS). A WiMAX basestation uses multiple antennas pointed in different directions; the area covered.

2.8.4 Cell Phone Technologies

Cell phone technology seems an obvious approach to mobile computer communication,and indeed data services based on cellular standards are commercially available. One drawback is the cost to users, due in part to cellular's use of licensed spectrum (which has historically been sold off to cellular phone operators for astronomical sums). The frequency bands that are used for cellular telephones (and now for cellular data) vary around the world. In Europe, for example, the main bands for cellular phones are at 900 and 1,800 MHz. In North America, 850- and 1,900-MHz bands are used. Thisglobal variation in spectrum usage creates problems for users who want to travel fromone part of the world to another, and has created a market for phones that can operateat multiple frequencies (e.g., a tri-band phone can operate at three of the four frequencybands mentioned above). That problem, however, pales in comparison to the proliferation of incompatible standards that have plagued the cellular communication business. Only recently have some signs of convergence on a small set of standards appeared. And finally, there is the problem that most cellular technology was designed for voice communication, and is only now starting to support moderately high-bandwidth data communication.Like 802.11 and WiMAX, cellular technology relies on the use of base stations that are part of a wired network. The geographic area served by a base station's antenna is called a *cell*. A base station 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, a mobile phone could potentially communicate with multiple base stations. This is somewhat similar to the 802.11 picture the phone is in communication with, and under the control of, just one base station. As the phone begins to leave a cell, it moves into an area of overlap with one or more other cells. The current base station senses the weakening signal from the phone, and gives control of the phone to whichever base station is receiving the strongest signal from it.If the phone is involved in a call at the time, the call must be transferred to the new base station in what is called a *handoff*. As we noted above, there is not one unique standard for cellular, but rather a collection of competing technologies that support data traffic in different ways and deliver different speeds. These technologies are loosely categorized by "generation." The first generation (1G) was analog, and thus of limited interest from a data communications perspective. Most of the cell phone technology currently deployed is considered second generation (2G) or "2.5G" (not quite worthy of being called 3G, but more advanced than 2G). The 2G and later technologies are digital. The most widely deployed 2G technology is referred to as GSM—the Global System for Mobile Communications, which is used in more than 200 countries. North America, however, is a late adopter of GSM, which helped prolong the proliferation of competing standards. 2G technologies use one of two approacheharing a limited amount of spectrum between simultaneous calls. One way is a combination of FDM and TDM. The spectrum available is divided into disjoint frequency bands, and each band is subdivided into time slots. A given call is allocated every *n*th slot in one of the bands. Theother approach is code division multiple access (CDMA). CDMA does not divide the channel in either time or frequency, but rather uses different chipping codes to distinguish the transmissions of different cellphone users. (See Section 2.1.2 for a

discussionof chipping codes.).The 2G and later cell phone technologies use compression algorithms tailored to human speech to compress voice data to about 8 Kbps without losing quality. Since 2G technologies focus on voice communication, they provide connections with just enoughbandwidth for that compressed speech—not enough for a decent data link. One of the
first cellular data standards to gain widespread adoption is the General Packet Radio
Service (GPRS), which is part of the GSM set of standards and is often referred to as a

Switching and Forwarding

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, multioutput device, which transfers packets from an input to one or more outputs. Thus, a switch adds the star topology to the point-to-point link, bus (Ethernet), and ring (802.5, 802.17, and FDDI) topologies established in the last chapter. 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. 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 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. is one of the design goals for a switch; we return to this topic below. In general, switchednetworks are considered more *scalable* (i.e., more capable of growing to large numbers ofnodes) than shared-media networks because of this ability to support many hosts at full speed.

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 OSI architecture, it is the main function of the network layer.

The question then is, how does the switch decide on which output port to place each packet? 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.

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 onlyspecifies a set of nodes to be traversed, without saying exactly how to get from one nodeto the next. A loose source route can be thought of as a set of waypoints rather thana completely specified route. The loose option can be helpful to limit the amount ofinformation 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 needsto correctly construct a strict source route to any destination.

Bridges and LAN Switches

Having discussed some of the basic ideas behind switching, we now focus more closely on some specific switching technologies. We begin by considering a class of switch thatare even some newer types of optical switch that use microscopic, electronicallycontrolled mirrors to deflect allthe light from one switch port to another,so that there could be an uninterrupted optical channel from pointA to point B. The technology behind these devices is called MEMS (MicroelectromechanicalSystems).We don’t cover optical networking extensively in this book, in partbecause of space considerations One approach you might try is to put a repeater between them, as described in Chapter 2. This would not be a workable solution, however, if doing so exceededthe physical limitations of the Ethernet.(Recall that no more than four repeaters between any pair of hosts and no more than a total of 2,500 m in length are allowed.) An alternative would be to put a node between the two Ethernets and have the node forward frames from one Ethernet to the other. This node would be inpromiscuous mode, accepting all frames transmitted on either of the Ethernets, so it could forward them to the other.The node we have just described is typically called a *bridge*, and a collection of LANs connected by one or more devices.

Internetworking

Simple Internetworking (IP)

In the previous chapter, 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 toscale and to handle heterogeneity. In this chapter, we explore some ways to go beyondthe limitations of bridged networks, enabling us to build large, highly heterogeneousnetworks with reasonably efficient routing. We refer to such networks as *internetworks.*In the following sections, we make a steady progression toward larger and larger internetworks.We start with the basic functionality of the currently deployed version of the Internet Protocol (IP), and then we examine various techniques that have been developed
to extend the scalability of the Internet in Section 4.3. This discussion culminates with adescription of IP version 6 (IPv6), also known as the next generation IP. Before delving into the details of an internetworking protocol, however, let’s consider more carefully what the word “internetwork” means.

What Is an Internetwork?

We use the term “internetwork,” or sometimes just “internet” with a lowercase *i*, to refer to aarbitrary collection of networks interconnected to provide some sort of host-tohost packet delivery service. For example, a corporation with many sites might constructa private internetwork by interconnecting the LANs at their different sites with pointto-point links leased from the phone company. When we are talking about the widelyused, 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 thisbook, 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 Section 4.3. For now, we use *network* to mean either a directly connectedor a switched network of the kind that was discussed in the last two chapters. Such a network uses one technology, such as 802.5, Ethernet, or

ATM. 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, acollection of Ethernets connected by bridges or switches would still be viewed as a single network.
Figure 4.1 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, an FDDI ring, and a point-to-point link. Each of these is a singletechnology network. The nodes that interconnect the networks are called *routers*.

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 servicecan 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 provideguaranteed delivery of every packet in 1 ms or less if there were underlying networktechnologies that could arbitrarily delay packets. The philosophy used in defining the IP service model, therefore, was to make it undemanding enough that just about anynetworktechnology 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). 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 from Section 3.1.1 that a datagram is a type of packet that happens to be sent in a connectionless manner overa network. Every datagram carries enough information to let the network forward thepacket to its correct destination; there is no need for any advance setup mechanism totell 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 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, aswe shall see in Section 4.3, additional flexibility has been needed, and some innovative ways to provide it are now in use. Because one of these techniques actually removes the distinction between address classes, the addressing scheme just described is now knownas "classful" addressing to distinguish it from the newer "classless" approach.Before we look at how IP addresses get used, it is helpful to look at some practicalmatters, 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 valuecontained in 1 byte of the address, starting at the most significant. For example, theaddress 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, whichare also hierarchical. Domain names tend to be ASCII strings separated by dots, suchas cs.princeton.edu. We will be talking about those in Section 9.1.3. 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.

4.1.4 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 Chapter 3 that *forwarding* is the process of takinga packet from an input and sending it out on the appropriate output, while *routing* isthe 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.
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 datagramis sent from a source host to a destination host, possibly passing through 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. heterogeneity is one of the key reasons why the Internet is so widely deployed. It is also the fact that IP runs *over* virtually every other protocol (including ATM and Ethernet) that now causes those protocols to be viewed as layer 2 technologies.

▲

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.

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 ahost'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 and81 in the lower byte) might be given the IP address128.96.33.81. 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 of the table using the network. This can be accomplished using the Address

ARP takes advantage of the fact that many link-level network technologies, such as Ethernet and token ring, 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 node 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 addsthe information contained in this response to its ARP table. The query message also includes the IP address and link-layer address of the sendinghost. Thus, when a host broadcasts a query message, each host on the network can learn the sender's link-level nd 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 theinformation about the sender to its table, even if it did not already have an entry for thathost. This is because there is a good chance that the source host is about to send it anapplication-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 targetand 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.

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); the same administrative entity. The division of the ATM network into a number of LISs also improves scalability by limiting the number of nodes that must be supported by a single ARP server. The basic job of an ARP server is to enable nodes on a LIS to resolve IP addresses to ATM addresses without using broadcast. Each node in the LIS must be configuredwith the ATM address of the ARP server so that it can establish a VC to the server whenit boots. Once it has a VC to the server, the node sends a registration message to the ARP server that contains both the IP and ATM addresses of the registering node. Thus,the ARP server builds up a complete database of all the _IP address, ATM address_pairs. Once this is in place, any node that wants to send a packet to some IP address can ask the ARP server to provide the corresponding ATM address. Once this is received, the sending node can use ATM signalling to set up a VC to that ATM address, and then send the packet. Just like conventional ARP, a cache of IP-to-ATM address mappings can be maintained. In addition, the node can keep a VC established to that ATM destination aslong as there is enough traffic flowing to justify it, thus avoiding the delay of setting up the VC again when the next packet arrives.

An interesting consequence of the classical IP over ATM model is that two nodes on the same ATM network cannot establish a direct VC between themselves if they are on different subnets. This would violate the rule that communication from one subnet to another must pass through a router. For example, host H1 and host H2 in Figure 4.8 cannot establish a direct VC under the classical model. Instead, each needs to have a VCto router R. The simple explanation for this rule is that IP routing is known to work wellwhen that rule is obeyed, as it is in non-ATM networks. New techniques to work around that rule have been developed, but they have introduced considerable complexity and problems of robustness.

▲

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-effortservice model that makes minimal assumptions about the underlying networks; most notably, this service model is based on unreliable datagrams. IP then makes two importantadditions 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.

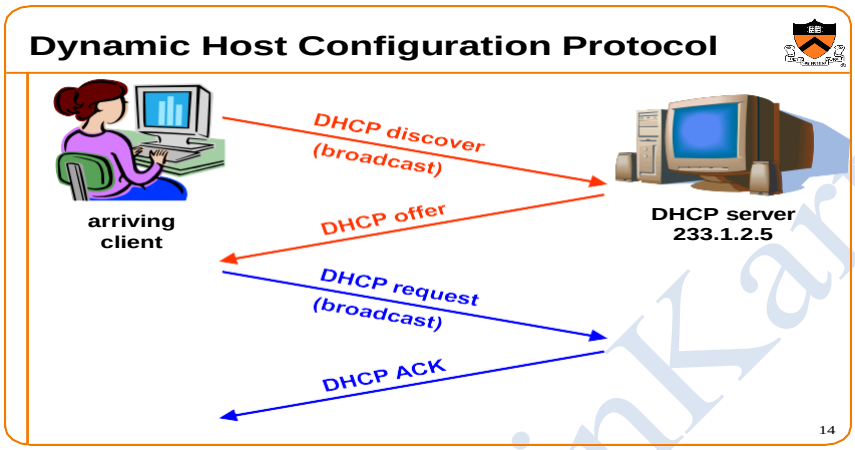
Host Configuration (DHCP)

we observed that 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, must be not only 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 hostwhen 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 auser, 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 ensurethat 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 internet work 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 juststore 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 agentreceives 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. 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 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 from growing too fast, it is important to pay attention to growth of network management complexity. By allowing network managers to configure a range of IP addresses pernetwork rather than one IP address per host, DHCP improves the manageability of a network.
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.

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.

4.1.8 Virtual Networks and Tunnels

We conclude our introduction to IP by considering an issue you might not have anticipated, but one that is becoming 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)*.

Routing

In both this and the previous 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 the packet's destination address and then to determine which of the output ports is the best choice to get the packet to that address.

▲ We restate an important distinction, which is often neglected, between *forwarding* and *routing*. Forwarding consists of taking a packet, looking at its destination address, consulting a table, and sending the packet in a direction determined by that table. We saw several examples of forwarding in the preceding section. Routing is the process by which forwarding tables are built. We also note that forwarding is a relatively simple and well-defined process performed locally at a node, whereas routing depends oncomplex distributed algorithms that have continued to evolve throughout the history of networking.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 number 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 numbers 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 a network number when forwarding a packet, while the routing table needs to be optimized for the purpose of calculating changes in topology. In some cases, the forwarding table may even be implemented in specialized hardware, whereas this is rarely if ever done for the routing table. Table 4.4 provides an example of a row from each sort of table. In this case, the routing table tells us that network number 10 is to be reached by a next hop router with the IP address 171.69.245.10, while the forwarding table 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 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 no. They are designed for networks of fairly modest size—fewer than a 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 router 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 section when we look at *interdomain* routing protocols.

Distance Vector (RIP)

The idea behind the distance-vector algorithm is suggested by its name:7 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. A link that is down is assigned an infinite cost. To see how a distance-vector routing algorithm works, it is easiest to consider an example like the one depicted in Figure 4.14. 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

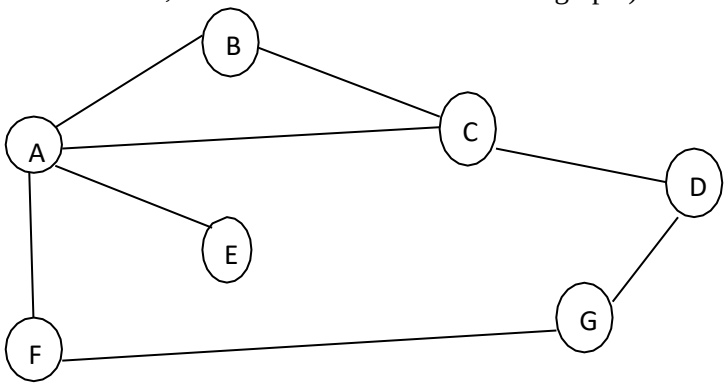


Figure 4.14 Distance-vector routing: an example network.

7The other common name for this class of algorithm is Bellman-Ford, after its inventors.

Information Stored at Node	Distance to Reach Node 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

Table 4.5 Initial distances stored at each node (global view).

Destination	Cost	Next Hop
B	1	B
C	1	C
D	∞	—
E	1	E
F	1	F
G	∞	—

Table 4.6 Initial routing table at node A.

knowledge about the distances to all other nodes as a table like the one given in Table 4.5. Note that each node only knows 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 4.5 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 thatDis 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 4.6.

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

Destination	Cost	Next Hop
B	1	B
C	1	C
D	2	C
E	1	E
F	1	F
G	2	F

Table 4.7 Final routing table at node A.

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. In the absence of any topology changes, it only takes 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 4.8 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. 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

Stored at Node	Information
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

Table 4.8 Final distances stored at each node (global view).

several minutes. The second mechanism, sometimes called a *triggered* update, happens whenever a node receives an update from one of its neighbors that causes it to change one of the routes in its routing table. That is, 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 2One 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*, Bactually 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

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 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)
```

```
{
++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]);
}
}
```

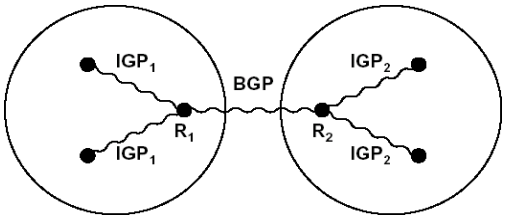
Routing Information Protocol (RIP)

One of the most widely used routing protocols in IP networks is the Routing Information Protocol (RIP). Its widespread use is 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 for

```
{
/* this is a completely new route;
is there room for it? */
if (numRoutes < MAXROUTES)
{
++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;
}
```

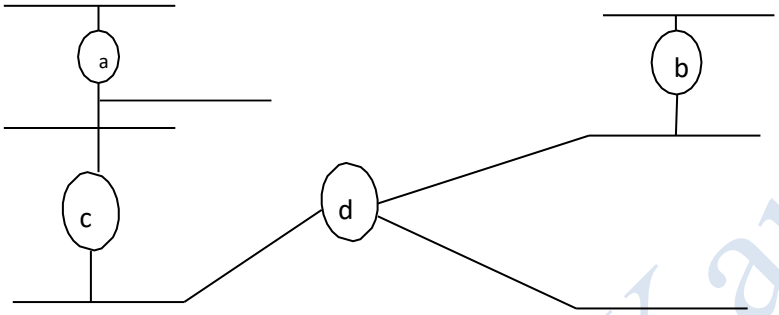
Finally, the procedure updateRoutingTable is the main routine that calls mergeRoute to incorporate all the routes contained in a routing update that is received from a neighboring node.

```
void
updateRoutingTable (Route *newRoute, int numNewRoutes)
{
int i;
for (i=0; i < numNewRoutes; ++i)
{
mergeRoute(&newRoute[i]);
}
}
```



Routing Information Protocol (RIP)

One of the most widely used routing protocols in IP networks is the Routing Information Protocol (RIP). Its widespread use is 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 descbed above.



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 families, not just IP.The network-address part of the advertisements is actually represented as a _family,address_ pair. RIP version 2 (RIPv2) also has some features related to scalability that we will discuss in the next section. 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.

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 neighbours (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, with

each node that receives this information forwarding 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), that 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 described in Section 2.5. However, there are several more steps needed 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.

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 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 solutionis 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, and $l(i, j)$ denotes the nonnegative cost (weight) associated with the edge between nodes $i, j \in N$, and $l(i, j)=\infty$ 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 nodesincorporated 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 $(N \_= M)$
$M = M \cup \{w\}$ 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 $C(n)$s) to other nodes using the known costs to directly connected nodes. We then look for the node that is reachable at the lowest 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, 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.

(a) If Neighbor is currently not on either the Confirmed or the Tentative list, then add (Neighbor, Cost, NextHop) to the Tentative list, where NextHop is the direction I go to reach Next.

(b) 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.

This will become a lot easier to understand when we look at an example. Consider the network depicted in Figure 4.18. Note that, unlike our previous example, this network has a range of different edge costs. Table 4.9 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.

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.

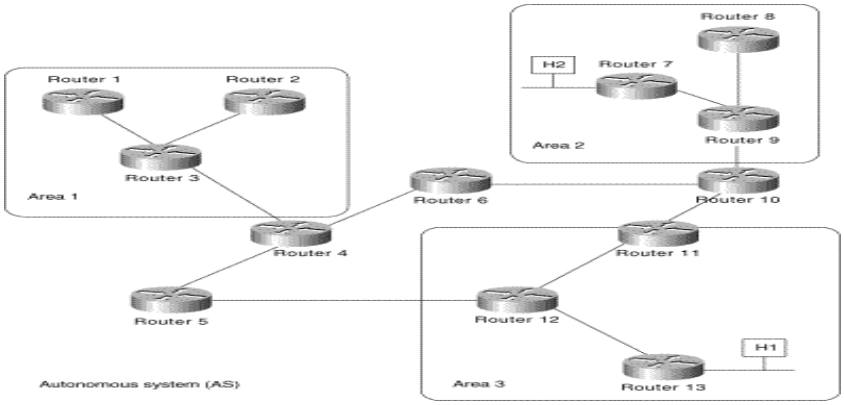
3 (D,0,-)
(C,2,C)
(B,11,B) Put lowest-cost member of Tentative (C) onto Confirmed list. Next, examine LSP of newly confirmed member (C).
4 (D,0,-)
(C,2,C)
(B,5,C)
(A,12,C)
Cost to reach B through C is 5, so replace (B,11,B). C's LSP tells us that we can reach A at cost 12.
5 (D,0,-)
(C,2,C)
(B,5,C)
(A,12,C) 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.

Open Shortest Path First Protocol (OSPF)

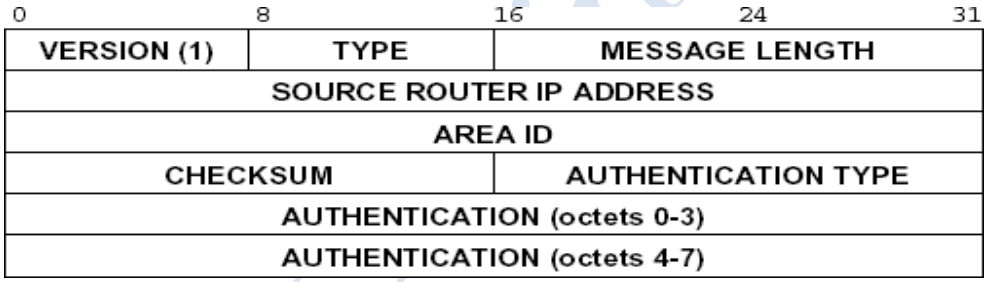
One of the most widely used link-state routing protocols is OSPF. The first word, "Open," refers to the fact that it is an open, nonproprietary standard, created under the auspices of the IETF. The "SPF" part comes from an alternative name for link-state routing. OSPF adds quite a number of features to the basic link-state algorithm described above, including the following:

■ Authentication of routing messages: This is a nice feature, since it is all too common for some misconfigured host to decide that it can reach every host in the universe at a cost of 0. When the host advertises this fact, every router in the surrounding neighborhood updates its forwarding tables to point to that host, and said host receives a vast amount of data that, in reality, it has no idea what to do with. It typically drops it all, bringing the network to a halt. Such disasters can be averted in many cases by requiring routing updates to be authenticated. 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 many problems caused by misconfiguration. (A similar form of authentication was added to RIP in version 2.) Strong cryptographic authentication of the sort discussed in Section 8.3 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. There are several different types of OSPF messages, but all begin with the same header, as shown in Figure 4.19. 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 (see Section 2.4). The Authentication type is 0 if no authentication is used; otherwise it may be 1, implying a simple password is used, or 2, which indicates that a cryptographic authentication checksum, of the sort described in Section 8.3, 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 known as the linkstate 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 4.20 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 Section 2.4 and 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 that 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 type of service (TOS) 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.

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 9.6-Kbps link look just as good as a 45-Mbps 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 5packets queued for transmission. Using queue length as a routing metric did not workwell, 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 A second version of the ARPANET routing algorithm, sometimes called the "new routing mechanism," 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

$$\text{Delay} = (\text{DepartTime} - \text{ArrivalTime}) + \text{TransmissionTime} + \text{Latency}$$

where TransmissionTime and Latency were statically defined for the link an 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 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
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 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

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 tens of thousands of networks connected to it. 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.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 4.24 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 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. 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 protocols 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. 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 paidmoney 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 ASs 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 inter domain routing system.To make the problem harder, I need to be able to implement such a policy without any help from other ASs, and in the face of possible misconfiguration or malicious behaviour by other ASs. There have been two major interdomain routing protocols in the recent history of the Internet. The first was the Exterior Gateway Protocol (EGP). EGP had a number of limitations, perhaps the most severe of which was that it constrained the topology of the Internet rather significantly. EGP basically forced a treelike topology onto the Internet, or to be more precise, it was design d when the Internet had a treelike topology, such as that illustrated in Figure 4.24. EGP 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 is the Border Gateway Protocol (BGP), which is in its fourth version at the time of this writing (BGP-4). BGP is also known for being rather complex. This section presents the highlights of BGP-4. As a starting position, BGP assumes that the Internet is an arbitrarily interconnected set of ASs. This model is clearly general enough to accommodate nontreestructured internetworks, like the simplified picture of today's multibackbone Internetshown .Unlike the simple tree-structured Internet. today's Internet consists of an interconnection of multiple backbone networks (they are usually called *service provider networks*, and they are operated by private companies rather than the government), and sites are connected to each other in arbitrary ways. Some large corporations connect directly to one or more of the backbones, while others connect to smaller, nonbackbone service providers. Many service providers exist mainly to provide service to "consumers" (i.e., individuals with PCs in their homes), and these providers must also connect to the backbone providers. Often many providers arrange to interconnect with each other at a single "peering point." In short, it is hard to discern much structure at all in today's Internet.

Given this rough sketch of the Internet, if we define *local traffic* as traffic that originates at or terminates on nodes within an AS, and *transit traffic* as traffic that passes through an AS, we can classify ASs into three 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 4.29 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; for example, the large corporation.

■ *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.

Whereas the discussion of routing in Section 4.2 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 ASs 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, intradomain focuses on finding the best, nonlooping, *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 A second 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 they choose to assign metrics to paths. This means that it is impossible to calculate meaningful path costs for a path that crosses multiple ASs. A cost of 1,000 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 third challenge involves the issue of trust. Provider A might be unwilling tobelieve 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 closely 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." When configuring BGP, the administrator of each AS picks at least one node to be a "BGP speaker," which is essentially a spokesperson for the entire AS. That BGP speaker establishes BGP sessions to other BGP speakers in other ASs. These sessions are used to exchange reachability information among ASs. In addition to the BGP speakers, the AS has one or more border gateways, which need not be the same as the speakers. The border gateways are the routers through which packets enter and leave the AS. In our simple example in routers R2 and R4 would be border gateways. Note that we have avoided using the word "gateway" until this point because it tends to be confusing. We can't avoid it here, given the name of the protocol we are describing. The important point to understand here is that, in the context of interdomain routing, a border gateway is simply an IP router that is charged with the task of forwarding packets between ASs. BGP does not belong to either of the two main classes of routing protocols (distance-vector and link-state protocols) described. In addition to advertising paths, BGP speakers need to be able to cancel previously advertised paths if a critical link or node on a path goes down. 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 (Note that the fields in this figure are multiples of 16 bits, unlike other packet formats in this chapter.) One point to note about BGP-4 is that it was designed to cope with the classless addresses described in Section 4.3.2. This means that the "networks" that are advertised in BGP are actually prefixes of any length. Thus, the updates contain both the prefix itself and its length in bits. When writing these down, it is common to write prefix/length. For example, a CIDR prefix that begins 192.4.16 and is 20 bits long would be written as 192.4.16/20. A final point to note is that BGP is defined to run on top of TCP, the reliable transport protocol described in Section 5.2. 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 "keep alive" message that says, in effect "I'm still here and nothing has changed." If that router were to crash, it would stop sending the keep alives, and the other routers that had learned routes from it would know that those routes were no longer valid. We will not delve further into the details of BGP-4, except to point out that allthe protocol does is specify how reachability information should be exchanged amongautonomous systems. BGP speakers obtain enough information by this exchange to calculate loop-free routes to all reachable networks, but how they choose the "best" routes is largely left to the policies of the AS.

Routing Areas

As if we didn't already have enough hierarchy, link-state intradomain routing protocols provide a means 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 intradomain routing protocols.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. 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 Section 4.2.3. 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 backbone to 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 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 that 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 trade-off between scalability and optimality of routing. The use of areas forces all packets travelling 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.

▲

This illustrates an important principle in network design. There is frequently a trade-off between some sort of optimality and scalability. When hierarchy is introduced, information is hidden from some nodes in the network, hindering their ability to make perfectly optimal decisions. However, information hiding is essential to scalability, 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 perfect optimality. 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 ofthe 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.

IP Version 6 (IPv6)

In many respects, the motivation for a new version of IP is the same as the motivationfor the techniques described so far in this section: to deal with scaling problems caused by the Internet's massive growth. Subnetting and CIDR have helped to contain the rate at which the Internet address space is being consumed (the address depletion problem) and have also helped to control the growth of routing table information needed in the Internet's routers (the routing information problem). However, there will come a point at which these techniques are no longer adequate. In particular, it is virtually impossible to achieve 100% address utilization efficiency, so the address space will be exhausted well before the 4 billionth host is connected to the Internet. Even if we were able to use all 4 billion addresses, it's not too hard to imagine ways that that number could be exhausted, such as the assignment of IP addresses to mobile phones, televisions, or other household appliances. All of these possibilities argue that a bigger address space than that provided by 32 bits will eventually be needed.

Support for real-time services;

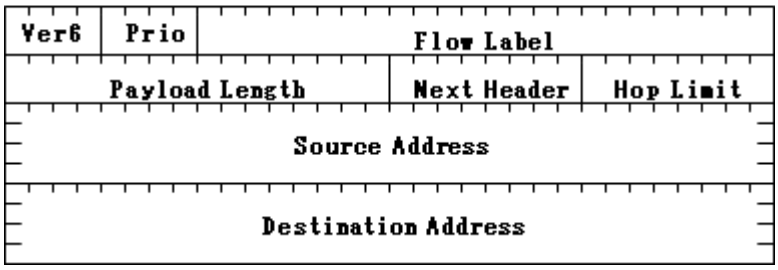
■ Security support;

■ 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.

IPv6 Header format



In addition to the wish list, one absolutely nonnegotiable feature for IPng 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. Thus, there will probably be a long transition period in which some hosts and routers will run IPv4 only, some will run IPv4 and IPv6, and some will run IPv6 only.

The IETF appointed a committee called the IPng Directorate to collect all the inputs on IPng requirements and to evaluate proposals for a protocol to become IPng. Over the life of this committee there were a number of proposals, some of which mergedwith other proposals, and eventually one was chosen by the Directorate to be the basis for IPng. That proposal was called Simple Internet Protocol Plus (SIPP). SIPP originally called for a doubling of the IP address size to 64 bits. When the Directorate selected SIPP, they stipulated several changes, one of which was another doubling of the address to 128 bits (16 bytes). It was around this time that the version number 6 was assigned.

The rest of this section describes some of the main features of IPv6. At the time of this writing, most of the key specifications for IPv6 are Proposed or Draft Standards in the IETF.

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 notaddress assignment efficiency. Based on the most pessimistic estimates of efficiency drawn from this study, the IPv6 address space is predicted to provide over 1,500 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. 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.

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 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.

Prefix	Use
00. . . 0 (128 bits)	Unspecified
00. . . 1 (128 bits)	Loopback
1111 1111	Multicast addresses
1111 1110 10	Link local unicast
1111 1110 11	Site local unicast
Everything else	Global unicast

Table 4.11 Address prefix assignments for IPv6.

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. Since there are two types of IPv6 addresses that contain an embedded IPv4 address, these 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. Using this scheme, an IPv6 address might look like.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. One tricky situation could occur when a subscriber is connected to more than one provider.Which prefix should the subscriber use for his 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.

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
.

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.

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 hostof IPv6, therefore, is to provide support for autoconfiguration, sometimes referred to as "plug-and-play" operation.

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 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

Other Features

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 have just described—autoconfiguration and source-directed routing. IPv6 includes several

additional features, most of which are covered elsewhere in this book— mobility is discussed in and a new service model proposed for the Internet is described. 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.

Multicast

Multiaccess networks like Ethernet and token rings implement multicast in hardware. There are, however, applications that need a broader multicasting capability that is effective at the scale of internetworks. 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, or software updates to multiple hosts. 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 androuters. Another problem is that the application would have to keep track of all the IP addresses to send to. For many, perhaps most, applications, that set of IP addresses could be constantly changing, for example, as listeners tune into an Internet radio station and other listeners turn it off. 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 multiaccess networks like Ethernet and token rings as we saw in Chapter 2. 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 the focus of this chapter 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 *sourcespecific 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 *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 LAN to determine which groups are still of interest to the attached hosts.

4.4.1 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 (see Table 4.11) 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. Thus, there are 28 bits of possible multicast addresses 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 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.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.

Multicast Routing (DVMRP, PIM, MSDP)

A router's unicast forwarding tables indicate, for any IP address, which link to use to forwardthe unicast packet. To support multicast, a router must additionally have multicastforwarding 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, which we discussed in Section 4.2.2 for 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 multicastgroup. 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 th 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 themulticast.
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

A multicast forwarding tree is built as a result of routers sending Join messages to the RP. PIM-SM allows two types of tree 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 4.38(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 (* meaning "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 4.38(a).
As more routers send Joins toward the RP, they cause new branches to be added to the tree, as illustrated in Figure 4.38(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, 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 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, 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 (Section 5.2) 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 whenevera 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,nd the IPaddress of the originating RP.f 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
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.

■ Since 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

The Fate of Multicast Protocols

A number of IP multicast protocols have fallen by the wayside since the 1991 publication of Steve Deering's doctoral thesis, "Multicast Routing in a Datagram Network." In most cases, their downfall had something to do with scaling. The most successful early multicast protocol was DVMRP, which we discussed at the start of the section. The *Multicast Open Shortest Path First (MOSPF)* protocol was based on the OSPF unicast routing protocol. PIM dense mode (PIMDM) has some similarity to DVMRP, in that it also uses a flood-and-prune approach; at the same time it is like PIM-SM in being independent of the unicast routing protocol used. All of these protocols are more appropriate to a "dense" domain (i.e., one with a high proportion of routers interested in the multicast). These protocols all appeared relatively early in the history of multicast, before some of the scaling challenges were fully apparent. Although they would still make sense within a domain for multicast groups expected to be of "dense" interest, they are rarely used today, in part because 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
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.

Multiprotocol Label Switching

We conclude our discussion of IP by describing an idea that was originally viewed as a way to improve the performance of the Internet. The idea, called *multiprotocol label switching (MPLS)*, tries to combine some of the properties of virtual circuits with the flexibility and robustness of datagrams. On the one hand, MPLS is very much associated 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.

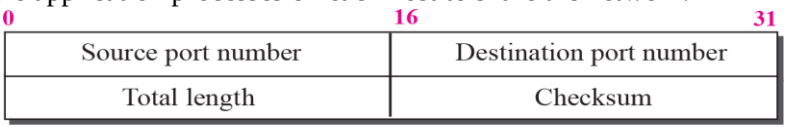
UNTT 4
TRANSORT LAYER

OVERVIEW OF TRANSORT LAYER

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. 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-desirable 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 demulti plexing 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 UDP and TCP protocols, 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. This discussion is capped off with a description of two widely used RPC protocols, SunRPC and DCE-RPC. 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. The protocol that is most widely used for this purpose is the Real-time Transport Protocol (RTP), which we examine here. Finally, the chapter concludes with a section that discusses the performance of the different transport protocols.

Simple Demultiplexer (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.



b. Header format

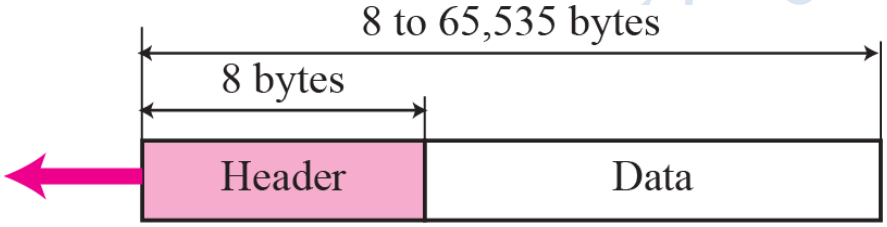
Aside from this requirement, the transport protocol adds no other functionality to the best-effort service provided by the underlying network. The Internet's User Datagram Protocol (UDP) is an

example of such a transport protocol. The only interesting issue in such a protocol is the form of the address used toidentify 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, often called a *port* or *mailbox*. 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 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. In fact, 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 (it was contained in the message header) and can reply to it.

Simple Demultiplexer (UDP)

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 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 wellknown port 517, and so on.



a. UDP user datagram

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. An alternative strategy is to generalize this idea, so that there is only a single wellknown

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 that tells 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 a little more work than to simply demultiplex 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 the current

Internet, but it will become mandatory with IPv6.) UDP computes its checksum over 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.) UDP uses the same checksum algorithm as IP. 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.

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 (TCP) 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.

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 (TCP) 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.

End-to-End Issues

At the heart of TCP is the sliding window algorithm. Even though this is the same basic algorithm we saw in Section 2.5.2, because TCP runs over the Internet rather than a 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 sliding window algorithm presented in Section 2.5.2 runs over a single physical link that always connects the same two computers, TCP supports logical connections etween 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 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.)

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 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 Acknowledgment, 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. The Acknowledgment 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 Acknowledgment and AdvertisedWindow.

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. The ACK flag is set any time the Acknowledgment 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 indicateswhere 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.

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.) 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).1 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.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*). then responds with a single segment that both acknowledges the client's sequence number (Flags = ACK, Ack = *x* + 1) and states its own 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 that each side acknowledges a sequence number that is one larger than the one sent is that the Acknowledgment field actually identifies the "next sequence number expected," there by 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 sequencenumber 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. 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. Now let's trace the typical transitions taken through the diagram 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, 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 Acknowledgment 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. 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.

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.

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 Last-ByteWritten. Clearly,

LastByteAcked ≤ LastByteSent

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 Last- ByteRcvd, whereas if data has arrived out of order, 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 Section 2.5.2; 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 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. **398 5 End-to-End Protocols**

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, that

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.

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 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.

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
```



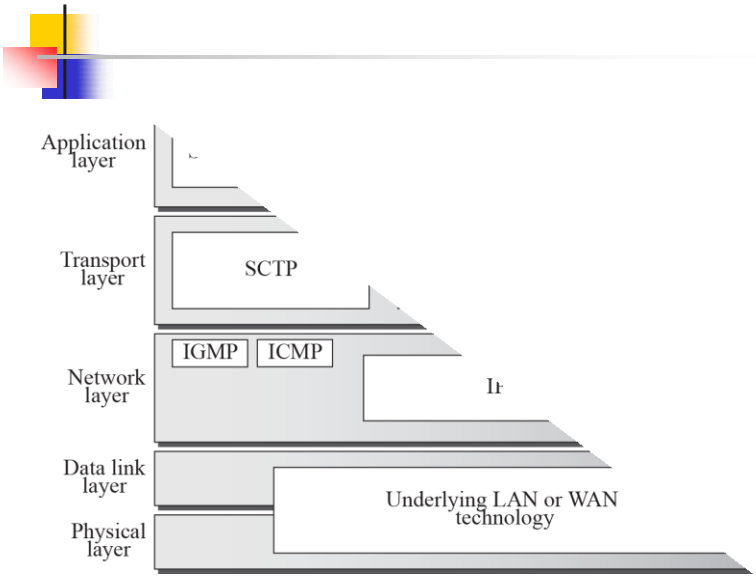
```
send a full segment
else
if there is unACKed data in flight
buffer the new data until an ACK arrives
else
send all the new data now
```

In other words, it's always OK to send a full segment if the window allows. It's also alright 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.

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.

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.



TCP/IP Protocol Suite

CONGESTON CONTROL

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 overflows and packets have to be dropped. When such drops become common events, 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 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; that is, 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.

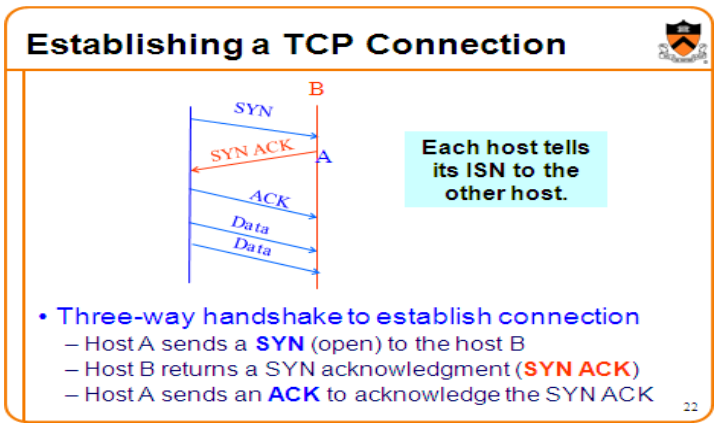
Congestion Control and Resource Allocation

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.

6.3 TCP Congestion Control

This section describes the predominant example of end-to-end congestion control in usetoday, 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 VanJacobson, 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 *selfclocking*. 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 these 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.



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 unacknowledged data allowed is now the minimum of the congestion window and the advertised window. Thus, using the variables defined in Section 5.2.4, 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.

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 × (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 MSS/CongestionWindow. 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.

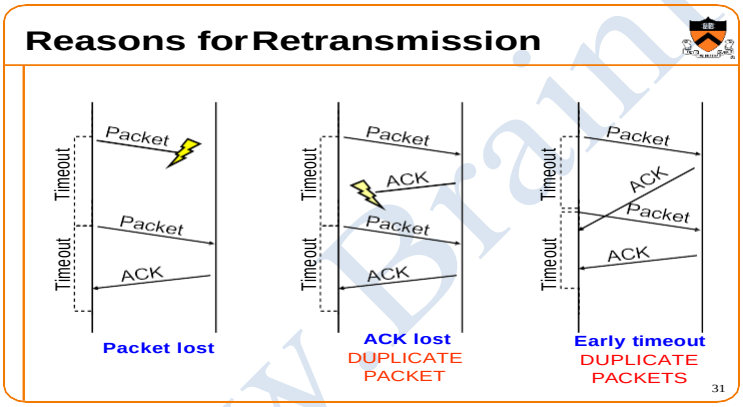
```
{
u_int cw = state->CongestionWindow;
u_int incr = state->maxseg;
if (cw > state->CongestionThreshold)
incr = incr * incr / cw;
state->CongestionWindow = MIN(cw + incr, TCP_MAXWIN);
}
```

CongestionWindow

1 A timeout happens, causing the congestion window to be divided by 2, dropping it from approximately 22 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

Fast Retransmit and Fast Recovery

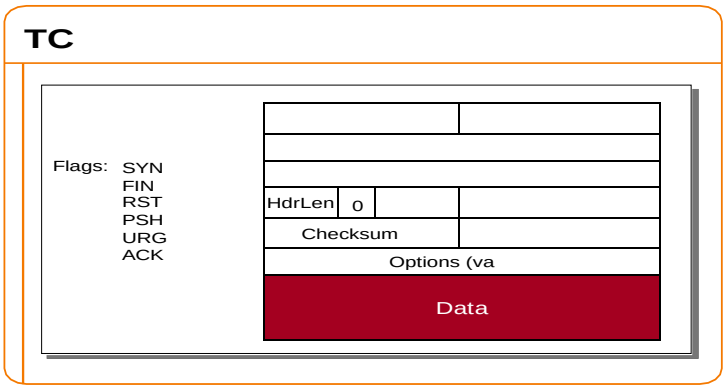
The mechanisms described so far were part of the original proposal to add congestion control to TCP. It was soon discovered,however, that the coarse-grained implementation of TCP timeouts led to long periods of time during which the connection went dead while waiting for a timer to expire. Because of 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.

Congestion-Avoidance Mechanisms

It is important to understand that 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, but one that has not yet been widely adopted, 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*.



This section describes three different congestion-avoidance mechanisms. The first two take a similar approach: They put a small amount of additional functionality intothe router to assist the end node in the anticipation of congestion. The third mechanism is very different from the first two: It attempts to avoid congestion purely from the end nodes.

6.4.1 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 6.14 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 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.

6.4.2 Random Early Detection (RED)

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 etects 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. Note that RED could easily be adapted to work with an explicit feedback scheme simply by *marking* a packet instead of *dropping* it, as discussed in the sidebar on Explicit Congestion Notification That is, Avg-Len is computed as

$$\text{AvgLen} = (1-\text{Weight}) \times \text{AvgLen} + \text{Weight} \times \text{SampleLen}$$

where $0 < \text{Weight} < 1$ and Sample-Len 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.

Quality of Service

For many years, packet-switched networks have offered the promise of supporting multimedia applications, that is, those that combine audio, video, and data. After all, once digitized, audio and video information become just another form of data—a stream of bits to be transmitted. One obstacle to the fulfillment of this promise has been the need for higher-bandwidth links. Recently, however, improvements in coding have reduced the bandwidth needs of audio and video applications, while at the same time link speeds have increased.

Congestion Control and Resource Allocation

There is more to transmitting audio and video over a network than just providing sufficient bandwidth, however. 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 *realtime 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.

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 nonreal-time. The latter are sometimes called “traditional data” applications, since they havetraditionally been the major applications found on data networks. They include most popular applications like Telnet, FTP, email, web browsing, and so on. All of these applications can work without guarantees of timely delivery of data. Another term for this nonreal-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 Telnet to more asynchronous ones like email, with interactive bulk transfers like FTP in the middle.

Real-Time Audio Example

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 that 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 tobuffer up some amount of data in reserve, thereby always providing a store of packetswaiting 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.

To get a better appreciation of how variable network delay can be, 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, 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

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. Toward 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.

Finally, 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.

UNIT 5
APPLICATION LAYER

Traditional Applications

We begin our discussion of applications by focusing on two of the most popular—the World Wide Web and email. We then turn to the domain name system (DNS)—not an application that users normally invoke explicitly, but nevertheless an application that all other applications depend upon. This is because the name server 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 server is usually used by other applications rather than by humans. Our finalexample in this section is network management, which although not so familiar to the average user, is the application of choice for system administrators. All of these application classes 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. By contrast, later sections will look at a class of applications that have become feasible only relatively recently: streaming applications (e.g., multimedia applications like video and audio) and various overlay-based applications. 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. There can be many different application programs—that is, web

clients like Internet Explorer, Netscape, Firefox, and Safari—that 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. This section focuses on four application protocols:

■ SMTP: Simple Mail Transfer Protocol is used to exchange electronic mail.
■ HTTP: HyperText Transport Protocol is used to communicate between web browsers and web servers.
■ DNS: Domain Name System protocol is used to query name servers and send the responses. (As we will see, DNS refers to rather more than just a protocol.)
■ SNMP: Simple NetworkManagement Protocol is used to query (and sometimes modify) the state of remote network nodes. The second point is that since all of the application protocols described in this section follow the same request/reply communication pattern, we would expect that they are all built on top of an RPC transport protocol. This is not the case, however, as they are all implemented on top of either TCP or UDP.

All these protocols except DNS have a companion protocol that specifies the format of the data that can be exchanged. This is one reason these protocols are relatively simple: Much of the complexity is managed in this companion document. For example, SMTP is a protocol for exchanging electronic mail messages, but RFC 822 (this specification has no other name) 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 form of those pages. Finally, SNMP is a protocol for querying a network node, but management information base (MIB) defines the variables that can be queried.

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? In fact, 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 a surprisingly successful application. Out of this work evolved the Internet's email system, which is now used by hundreds of millions of people every day. As with all the applications described in this section, the place to start in understanding how email works is to (1) distinguish the user interface (i.e., your mail reader) from the underlying message transfer protocol (in this case, SMTP), and (2) to distinguish between this transfer protocol and a companion protocol (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 + line-feed, which are a pair of ASCII control characters.

Applications

For example, the To: header identifies the message recipient, and the Subject: header says something about the purpose of the message. Other headers are filled in by the underlying mail delivery system. Examples include Date: (when the message was transmitted), From: (what user sent the message), and Received: (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 again in 1996) to allow email messages to carry many different types of data: audio, video, images, Word 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 MIME-Version: (the version of MIME being used), Content-Description: (a human-readable description of what's in the message, analogous to the Subject: line), Content-Type: (the type of data contained in the message), and Content-Transfer-Encoding (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 image/gif and image/jpeg, each with the obvious meaning. As another example, text/plain refers to simple text you might find in a vanilla 822-style message, while text/richtext 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.
ome 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's 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 Content-Type 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 Content-Type and Content-Transfer-Encoding lines.

Message Transfer

Next, we look at SMTP—the protocol used to transfer messages from one host to another.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. There are countless mail readers available, just like there are many web browsers to choose from. In fact, most web browsers now include a mail reader. Second, there is a *mail daemon* (or process) running on each host. You can think of this process as playing the role of a post office: 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 it). 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 the mail daemon running on most hosts is derived from the sendmail program originally implemented on While it is certainly possible that the sendmail program on a sender's machine establishes SMTP/TCP connection to the sendmail program on the recipient's machine, 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 sendmail 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.

Mail Reader

The final step is for the user to actually retrieve 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. In many cases, this reader is just a program running on the same machine as the user's mailbox resides, in which case it simply reads and writes the file that implements the mailbox. In other cases, the user accesses her mailbox from a remote machine using yet another protocol, such as the Post Office Protocol (POP) or the Internet Message Access Protocol (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 herself, and identifying the mailbox she wants to access.

LOGIN, AUTHENTICATE, SELECT, EXAMINE, CLOSE,and LOGOUT are example commands that the client can issue, while OK is one possible server response. Other common commands include FETCH, STORE, DELETE, and EXPUNGE, with the obvious meanings. Additional server responses include NO (client does not have permission to perform that operation) and BAD (command is illformed).

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 like the size of the message, but more interestingly, various *flags* associated with the message, such as Seen, Answered, Deleted, 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 alldeleted 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.

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. 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.

When you select 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, each HTTP 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 specify 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 typically empty for request messages.

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.

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

Table 9.1 HTTP request operations.

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 www.cs.princeton.edu to return the page named index.html. 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 If-Modified-Since, 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.

example, the START_LINE

```
HTTP/1.1 202 Accepted
```

indicates that the server was able to satisfy the request,

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

Table 9.2 Five types of HTTP result codes.

```
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. 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 http://www.cs.princeton.edu/index.html to http://www.princeton.edu/cs/index.html, 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 Section 9.1.1). Certain MESSAGE_HEADER lines give attributes of the page contents, including Content-Length (number of bytes in the contents), Expires (time at which the contents are considered stale), and Last- Modified (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 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. Even human beings and corporations can be resources. A more concrete example is the mid scheme for message IDs. Hence, URIs are not always some kind of address for locating the resource; they can be purely identifiers. Furthermore, a URI qualifies as a URL only if it is *intended* to be used to locate the resource. Even if a particular URI appears to be an address, such as a URI that uses the http scheme, the URI is not considered a URL unless it is intended to be used to locate the resource. For example, XML namespaces are identified by URIs that use the http scheme but are nonetheless not URLs since there is no requirement that the URI give the location of any resource related to the namespace.

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.

The most important improvement in the latest version of HTTP (1.1) is to allow *persistent connections*—the client and server can exchange multiple request/response messages over the same TCP connection. Persistent connections have two 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.

9.1.3 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 *namespace* defines the set of possible names. A namespace can be either *flat* (names are not divisible into components), or it can be *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, node in the tree corresponds to a domain, and the leaves in the tree correspond to the hosts being named.

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 The Name and Value fields are exactly what you would expect, while the Type field specifies how the Value should be interpreted. For example, Type = A 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 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. 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 a3.nstld.com server has records for .edu 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 _

...

The mail exchange (MX) records serve the same purpose for the email application—it allows 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.

Network Management (SNMP)

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—for example, address translation tables, routing tables, TCP connection state, and so on—then it is easy to become depressed about 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 possibly write, various pieces of state information on different network nodes. The most widely used protocol for this purpose is the Simple Network Management Protocol (SNMP). 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.

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 10 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, or how many packets have been sent and received on each interface.

■ Address translation: information about the Address Resolution Protocol (ARP), 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 ICMP, EGP, and SNMP itself. The tenth group is used by different media.

Web Services

Most of the applications that we have examined so far involve interaction between a human and a machine. 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 ing 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 due to being 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 fulfilment 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. In the business world, enabling applications to interact directly with each other is called business-to-business (B2B) integration when the applications are at different enterprises, and enterprise application integration (EAI) when they are within the same enterprise.

Network applications, even those that cross organization boundaries, are not new—we have just seen some examples in the preceding section. 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 protocols' 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 EAI and B2B 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.

Here is a simple example of what we are talking about. Suppose you buy a book at an online retailer like Amazon.com. 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.

Custom Application Protocols (WSDL, SOAP)

The architecture informally referred to as SOAP is based on *Web Services Description Language (WSDL)* and *SOAP*.2 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. As these j and security. Both WSDL and SOAP consist primarily of a protocol specification language. Both languages are based on XML (Section 7.1.3) 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 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 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 Schema (Section 7.1.3). 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

Defining Transport Protocols

SOAP uses many of the same strategies as WSDL, including message formats defined using XML Schema, bindings to underlying protocols, MEPs, 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. Of course it's not quite that simple. 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 (e.g., 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 more interesting 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 that contains the payload destined for the ultimate receiver.

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 applying the model underlying the World Wide Web architecture (Section 9.1.2). 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 suitable, 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. signers 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 about 80% of their usage over several years has been via the REST interface. Of course this is just one data point and may well reflect factors specific to Amazon.