
A SYSTEMS APPROACH
A SYSTEMS APPROACH
3
A SYSTEMS APPROACH
A SYSTEMS APPROACH
A SYSTEMS APPROACH

3 Internetworking

Nature seems . . . to reach many of her ends by long circuitous routes.

–Rudolph Lotze

In the previous chapter, we saw how to connect one node to another or to an existing network. Many technologies can be used to build "last-mile" links or to connect a modest number of nodes together, but how do we build networks of global scale? A single Ethernet can interconnect no more than 1024 hosts; a point-to-point link connects only two. Wireless networks are limited by the ranges of their radios. To build a global network, we need a way to interconnect these different types of links and networks. The concept of interconnecting different types of networks to build a large, global network is the core idea of the Internet and is often referred to as *internetworking*.

PROBLEM: NOT ALL NETWORKS ARE DIRECTLY CONNECTED

We can divide the internetworking problem up into a few subproblems. First of all, we need a way to interconnect links. Devices that interconnect links of the same type are often called *switches*, and these devices are the first topic of this chapter. A particularly important class of switches today are those used to interconnect Ethernet segments; these switches are also sometimes called *bridges*. The core job of a switch is to take packets that arrive on an input and *forward*

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

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

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

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

LAB 03: Switched LANs

3.1 SWITCHING AND BRIDGING

In the simplest terms, a switch is a mechanism that allows us to interconnect links to form a larger network. A switch is a multi-input, multi-output device that transfers packets from an input to one or more outputs. Thus, a switch adds the star topology (see Figure 3.1) to the point-to-point link, bus (Ethernet), and ring 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.

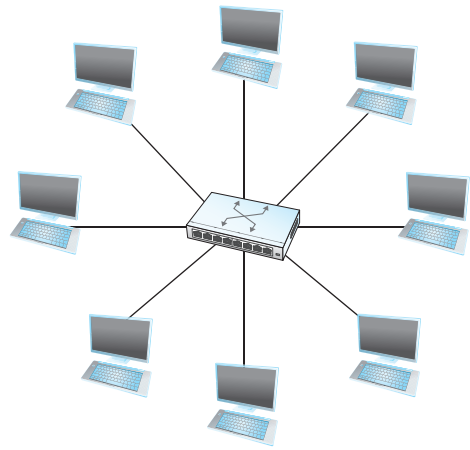


FIGURE 3.1 A switch provides a star topology.

- 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 segment to transmit continuously at 10 Mbps because they share the same transmission medium. Every host on a switched network has its own link to the switch, so it may be entirely possible for many hosts to transmit at the full link speed (bandwidth), provided that the switch is designed with enough aggregate capacity. Providing high aggregate throughput is one of the design goals for a switch; we return to this topic later. In general, switched networks are considered more *scalable* (i.e., more capable of growing to large numbers of nodes) than shared-media networks because of this ability to support many hosts at full speed.

A switch is connected to a set of links and, for each of these links, runs the appropriate data link protocol to communicate with the node at the other end of the link. A switch's primary job is to receive incoming packets on one of its links and to transmit them on some other link. This function is sometimes referred to as either *switching* or *forwarding,* and in

terms of the Open Systems Interconnection (OSI) architecture, it is the main function of the network layer.

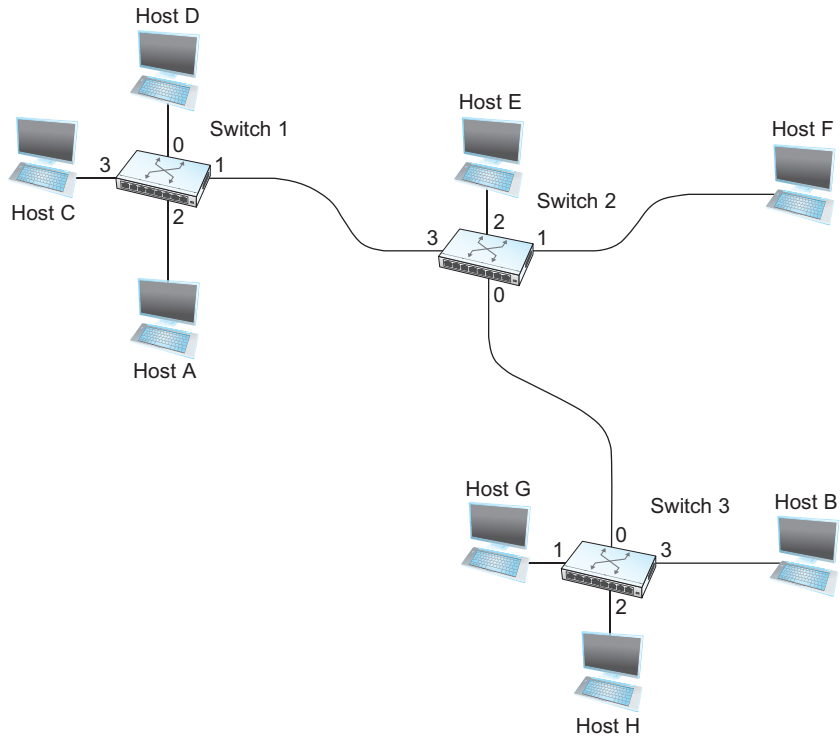
The question, then, is how does the switch decide which output link to place each packet on? The general answer is that it looks at the header of the packet for an identifier that it uses to make the decision. The details of how it uses this identifier vary, but there are two common approaches. The first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

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

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

3.1.1 Datagrams

The idea behind datagrams is incredibly simple: You just include in every packet enough information to enable any switch to decide how to get it to its destination. That is, every packet contains the complete destination address. Consider the example network illustrated in Figure 3.2, in which the hosts have addresses A, B, C, and so on. To decide how to forward a packet, a switch consults a *forwarding table* (sometimes called a *routing table*), an example of which is depicted in Table 3.1. This particular table shows the forwarding information that switch 2 needs to forward datagrams in the example network. It is pretty easy to figure out such a table when you have a complete map of a simple network like that depicted here; we could imagine a network operator configuring



■ FIGURE 3.2 Datagram forwarding: an example network.

Table 3.1 Forwarding Table for Switch 2

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

the tables statically. It is a lot harder to create the forwarding tables in large, complex networks with dynamically changing topologies and multiple paths between destinations. That harder problem is known as *routing* and is the topic of Section 3.3. We can think of routing as a process that takes place in the background so that, when a data packet turns up, we will have the right information in the forwarding table to be able to forward, or switch, the packet.

Datagram networks have the following characteristics:

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

This last fact is particularly important to the history of datagram networks. One of the important design goals of the Internet is robustness to failures, and history has shown it to be quite effective at meeting this goal.[1]

3.1.2 Virtual Circuit Switching

A second technique for packet switching, which differs significantly from the datagram model, uses the concept of a *virtual circuit* (VC).

[1]The oft-repeated claim that the ARPANET was built to withstand nuclear attack does not appear to be substantiated by those who actually worked on its design, but robustness to failure of individual components was certainly a goal.

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

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

- A *virtual circuit identifier* (VCI) that uniquely identifies the connection at this switch and which will be carried inside the header of the packets that belong to this connection
- An incoming interface on which packets for this VC arrive at the switch
- An outgoing interface in which packets for this VC leave the switch
- A potentially different VCI that will be used for outgoing packets

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

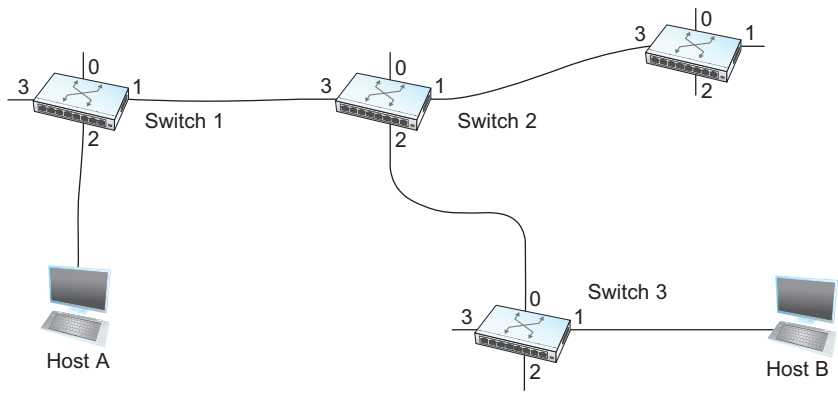


FIGURE 3.3 An example of a virtual circuit network.

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

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

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

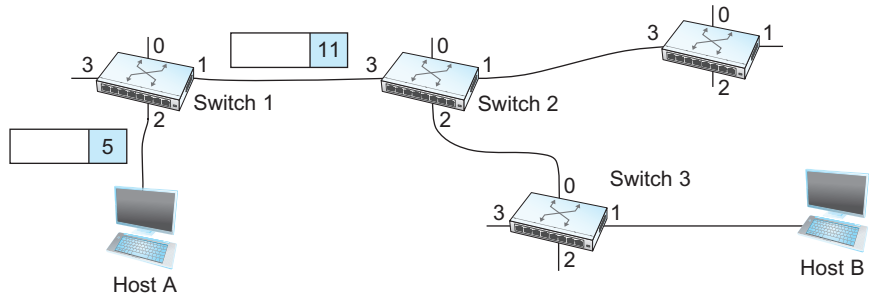
Let's assume that a network administrator wants to manually create a new virtual connection from host A to host B.[2] First, the administrator needs to identify a path through the network from A to B. In the example network of Figure 3.3, there is only one such path, but in general this may not be the case. The administrator then picks a VCI value that is currently unused on each link for the connection. For the purposes of our example, let's suppose that the VCI value 5 is chosen for the link from host A to switch 1, and that 11 is chosen for the link from switch 1 to switch 2. In that case, switch 1 needs to have an entry in its VC table configured as shown in Table 3.2.

Similarly, suppose that the VCI of 7 is chosen to identify this connection on the link from switch 2 to switch 3 and that a VCI of 4 is chosen for

[2]In practice, the process would likely be much more automated than described here, perhaps using some sort of graphical network management tool. The following steps illustrate the state that has to be established in any case.

Table 3.2 Virtual Circuit Table Entry for Switch 1

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



■ FIGURE 3.4 A packet is sent into a virtual circuit network.

the link from switch 3 to host B. In that case, switches 2 and 3 need to be configured with VC table entries as shown in Table 3.3. Note that the "outgoing" VCI value at one switch is the "incoming" VCI value at the next switch.

Once the VC tables have been set up, the data transfer phase can proceed, as illustrated in Figure 3.4. For any packet that it wants to send to host B, A puts the VCI value of 5 in the header of the packet and sends it to switch 1. Switch 1 receives any such packet on interface 2, and it uses the combination of the interface and the VCI in the packet header to find the appropriate VC table entry. As shown in Table 3.2, the table entry in this

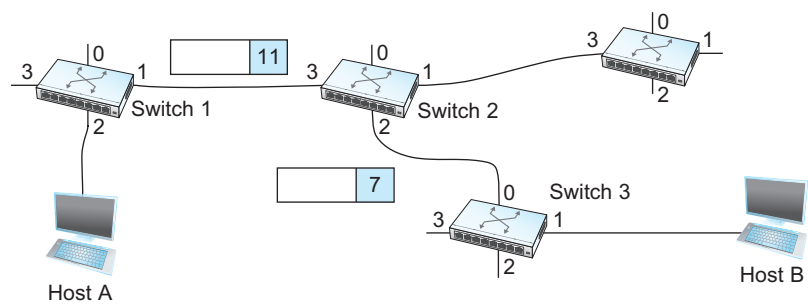
Table 3.3 Virtual Circuit Table Entries for Switches 2 and 3

VC Table Entry at Switch 2			
Incoming Interface	**Incoming VCI**	**Outgoing Interface**	**Outgoing VCI**
3	11	2	7
VC Table Entry at Switch 3			
Incoming Interface	**Incoming VCI**	**Outgoing Interface**	**Outgoing VCI**
0	7	1	4

case tells switch 1 to forward the packet out of interface 1 and to put the VCI value 11 in the header when the packet is sent. Thus, the packet will arrive at switch 2 on interface 3 bearing VCI 11. Switch 2 looks up interface 3 and VCI 11 in its VC table (as shown in Table 3.3) and sends the packet on to switch 3 after updating the VCI value in the packet header appropriately, as shown in Figure 3.5. This process continues until it arrives at host B with the VCI value of 4 in the packet. To host B, this identifies the packet as having come from host A.

In real networks of reasonable size, the burden of configuring VC tables correctly in a large number of switches would quickly become excessive using the above procedures. Thus, either a network management tool or some sort of signalling (or both) is almost always used, even when setting up "permanent" VCs. In the case of PVCs, signalling is initiated by the network administrator, while SVCs are usually set up using signalling by one of the hosts. We consider now how the same VC just described could be set up by signalling from the host.

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



■ **FIGURE 3.5** A packet makes its way through a virtual circuit network.

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

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

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

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

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

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

There are several things to note about virtual circuit switching:

- Since host A has to wait for the connection request to reach the far side of the network and return before it can send its first data packet, there is at least one round-trip time (RTT) of delay before data is sent.[3]
- While the connection request contains the full address for host B (which might be quite large, being a global identifier on the network), each data packet contains only a small identifier, which is only unique on one link. Thus, the per-packet overhead caused by the header is reduced relative to the datagram model.
- If a switch or a link in a connection fails, the connection is broken and a new one will need to be established. Also, the old one needs to be torn down to free up table storage space in the switches.
- The issue of how a switch decides which link to forward the connection request on has been glossed over. In essence, this is the same problem as building up the forwarding table for datagram forwarding, which requires some sort of *routing algorithm*. Routing is described in Section 3.3, and the algorithms described there are generally applicable to routing setup requests as well as datagrams.

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

[3]This is not strictly true. Some people have proposed "optimistically" sending a data packet immediately after sending the connection request. However, most current implementations wait for connection setup to complete before sending data.

networking technology. X.25 networks employ the following three-part strategy:

1. Buffers are allocated to each virtual circuit when the circuit is initialized.
2. The sliding window protocol (Section 2.5) is run between each pair of nodes along the virtual circuit, and this protocol is augmented with flow control to keep the sending node from over-running the buffers allocated at the receiving node.
3. The circuit is rejected by a given node if not enough buffers are available at that node when the connection request message is processed.

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

By comparison, a datagram network has no connection establishment phase, and each switch processes each packet independently, making it less obvious how a datagram network would allocate resources in a meaningful way. Instead, each arriving packet competes with all other packets for buffer space. If there are no free buffers, the incoming packet must be discarded. We observe, however, that even in a datagram-based network a source host often sends a sequence of packets to the same destination host. It is possible for each switch to distinguish among the set of packets it currently has queued, based on the source/destination pair, and thus for the switch to ensure that the packets belonging to each source/destination pair are receiving a fair share of the switch's buffers. We discuss this idea in much greater depth in Chapter 6.

In the virtual circuit model, we could imagine providing each circuit with a different *quality of service* (QoS). In this setting, the term *quality of service* is usually taken to mean that the network gives the user some kind of performance-related guarantee, which in turn implies that switches set aside the resources they need to meet this guarantee. For example, the switches along a given virtual circuit might allocate a percentage of each outgoing link's bandwidth to that circuit. As another example, a sequence of switches might ensure that packets belonging to a particular circuit not be delayed (queued) for more than a certain amount of time. We return to the topic of quality of service in Section 6.5.

Introduction to Congestion

One important issue that switch designers face is *contention*. Contention occurs when multiple packets have to be queued at a switch because they are competing for the same output link. We'll look at how switches deal with this issue in Section 1.4. You can think of contention as something that happens at the timescale of individual packet arrivals. Congestion, by contrast, happens at a slightly longer timescale, when a switch has so many packets queued that it runs out of buffer space and has to start dropping packets. We'll return to the topic of congestion in Chapter 6, after we have seen the transport protocol component of the network architecture. At this point, however, we observe that how you deal with congestion is related to the issue of whether your network uses virtual circuits or datagrams.

On the one hand, suppose that each switch allocates enough buffers to handle the packets belonging to each virtual circuit it supports, as is done in an X.25 network. In this case, the network has defined away the problem of congestion—a switch never encounters a situation in which it has more packets to queue than it has buffer space, since it does not allow the connection to be established in the first place unless it can dedicate enough resources to it to avoid this situation. The problem with this approach, however, is that it is extremely conservative—it is unlikely that all the circuits will need to use all of their buffers at the same time, and as a consequence the switch is potentially underutilized.

On the other hand, the datagram model seemingly invites congestion—you do not know that there is enough contention at a switch to cause congestion until you run out of buffers. At that point, it is too late to prevent the congestion, and your only choice is to try to recover from it. The good news, of course, is that you may be able to get better utilization out of your switches since you are not holding buffers in reserve for a worst-case scenario that is unlikely to happen.

As is quite often the case, nothing is strictly black and white—there are design advantages for defining congestion away (as the X.25 model does) and for doing nothing about congestion until after it happens (as the simple datagram model does). There are also intermediate points between these two extremes. We describe some of these design points in Chapter 6.

There have been a number of successful examples of virtual circuit technologies over the years, notably X.25, Frame Relay, and Asynchronous Transfer Mode (ATM). With the success of the Internet's connectionless model, however, none of them enjoys great popularity today. One of the most common applications of virtual circuits for many years was the construction of *virtual private networks* (VPNs), a subject discussed in Section 3.2.9. Even that application is now mostly supported using Internet-based technologies today.

Optical Switching

To a casual observer of the networking industry around the year 2000, it might have appeared that the most interesting sort of switching was optical switching. Indeed, optical switching did become an important technology in the late 1990s, due to a confluence of several factors. One factor was the commercial availability of dense wavelength division multiplexing (DWDM) equipment, which makes it possible to send a great deal of information down a single fiber by transmitting on a large number of optical wavelengths (or colors) at once. Thus, for example, one might send data on 100 or more different wavelengths, and each wavelength might carry as much as 10 Gbps of data.

A second factor was the commercial availability of optical amplifiers. Optical signals are attenuated as they pass through fiber, and after some distance (about 40 km or so) they need to be made stronger in some way. Before optical amplifiers, it was necessary to place *repeaters* in the path to recover the optical signal, convert it to a digital electronic signal, and then convert it back to optical again. Before you could get the data into a repeater, you would have to demultiplex it using a DWDM terminal. Thus, a large number of DWDM terminals would be needed just to drive a single fiber pair for a long distance. Optical amplifiers, unlike repeaters, are *analog* devices that boost whatever signal is sent along the fiber, even if it is sent on a hundred different wavelengths. Optical amplifiers therefore made DWDM gear much more attractive, because now a pair of DWDM terminals could talk to each other when separated by a distance of hundreds of kilometers. Furthermore, you could even upgrade the DWDM gear at the ends without touching the optical amplifiers in the middle of the path, because they will amplify 100 wavelengths as easily as 50 wavelengths.

With DWDM and optical amplifiers, it became possible to build optical networks of huge capacity. But at least one more type of device is needed to make these networks useful—the *optical switch*. Most so-called optical switches today actually perform their switching function electronically, and from an architectural point of view they have more in common with the circuit switches of the telephone network than the packet switches described in this chapter. A typical optical switch has a large number of interfaces that understand SONET framing and is able to cross-connect a SONET channel from an incoming interface to an outgoing interface. Thus, with an optical switch, it becomes possible to provide SONET channels from point A to point B via point C even if there is no direct fiber path from A to B—there just needs to be a path from A to C, a switch at C, and a path from C to B. In this respect, an optical switch bears some relationship to the switches in Figure 3.3, in that it creates the illusion of a connection between two points even when there is no direct physical connection between them. However, optical switches do not provide virtual circuits, they provide "real" circuits (e.g., a SONET channel). There are even some newer types of optical

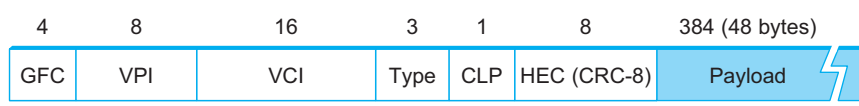
switches that use microscopic, electronically controlled mirrors to deflect all the light from one switch port to another, so that there could be an uninterrupted optical channel from point A to point B. The technology behind these devices is called *MEMS* (*microelectromechanical systems*).

We don't cover optical networking extensively in this book, in part because of space considerations. For many practical purposes, you can think of optical networks as a piece of the infrastructure that enables telephone companies to provide SONET links or other types of circuits where and when you need them. However, it is worth noting that many of the technologies that are discussed later in this book, such as routing protocols and multiprotocol label switching, do have application to the world of optical networking.

Asynchronous Transfer Mode (ATM)

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

There are a few aspects of ATM that are worth examining. The picture of the ATM packet format—more commonly called an ATM *cell*—in Figure 3.6 will illustrate the main points. We'll skip the generic flow control (GFC) bits, which never saw much use, and start with the 24 bits that are labelled VPI (virtual path identifier—8 bits) and VCI (virtual circuit identifier—16 bits). If you consider these bits together as a single 24-bit



■ **FIGURE 3.6** ATM cell format at the UNI.

field, they correspond to the virtual circuit identifier introduced above. The reason for breaking the field into two parts was to allow for a level of hierarchy: All the circuits with the same VPI could, in some cases, be treated as a group (a virtual path) and could all be switched together looking only at the VPI, simplifying the work of a switch that could ignore all the VCI bits and reducing the size of the VC table considerably.

Skipping to the last header byte we find an 8-bit cyclic redundancy check (CRC), known as the *header error check* (HEC). It uses the CRC-8 polynomial given in Section 2.4.3 and provides error detection and single-bit error correction capability on the cell header only. Protecting the cell header is particularly important because an error in the VCI will cause the cell to be misdelivered.

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

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

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

those packets into some sort of cell in order to switch them, as we'll see in Section 3.4.

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

As it turns out, 48 bytes was picked for the ATM cell payload as a compromise. There were good arguments for both larger and smaller cells, but 48 made almost no one happy—a power of two would certainly have been better for computers to work with.

3.1.3 Source Routing

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

There are various ways to implement source routing. One would be to assign a number to each output of each switch and to place that number in the header of the packet. The switching function is then very simple: For each packet that arrives on an input, the switch would read the port number in the header and transmit the packet on that output. However, since there will in general be more than one switch in the path between the sending and the receiving host, the header for the packet needs to contain enough information to allow every switch in the path to

Where Are They Now?

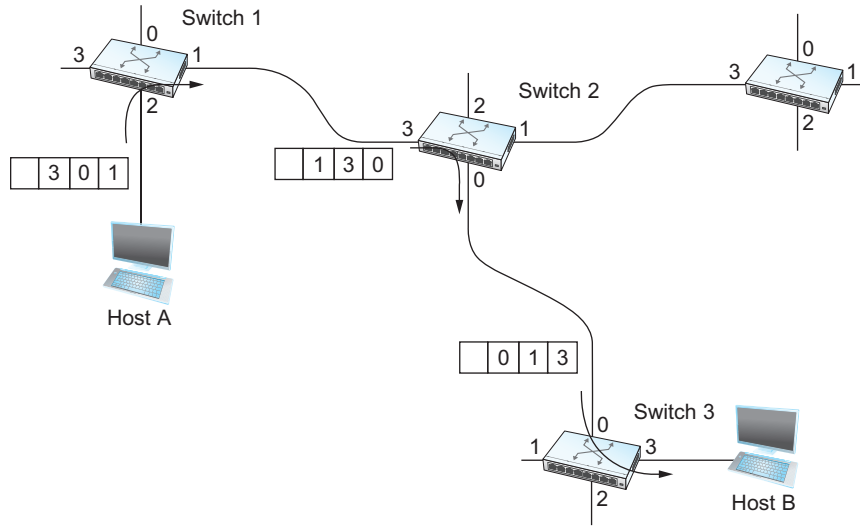
ATM

There was a period of time in the late 1980s and early 1990s when ATM seemed (to many people) poised to take over the world. The major telecommunication companies were supporting it, and the promise of high-speed networks that could smoothly integrate voice, video, and data onto a common network seemed compelling. Proponents of ATM referred to anything that used variable-length packets—technologies such as Ethernet and IP—as "legacy" technologies. Today, however, Ethernet and IP dominate, and ATM is viewed as yesterday's technology. You can still find pockets of ATM deployment, primarily as a way to get access to IP networks. Notably, a lot of Digital Subscriber Line (DSL) access networks were built using ATM, so some amount of broadband Internet access today is actually over ATM links, although this fact is completely hidden by the DSL modems, which take Ethernet frames and chop them into cells which are subsequently reassembled inside the access network.

There is room for debate as to why ATM didn't take over the world. One thing that seems fundamentally important in retrospect was that IP was well on its way to becoming completely entrenched by the time ATM appeared. Even though the Internet wasn't on the radar of a lot of people in the 1980s, it was already achieving global reach and the number of hosts connected was doubling every year. And, since the whole point of IP was to smoothly interconnect all sorts of different networks, when ATM appeared, rather than displace IP as its proponents imagined it might, ATM was quickly absorbed as just another network type over which IP could run. At that point, ATM was more directly in competition with Ethernet than with IP, and the arrival of inexpensive Ethernet switches and 100-Mbps Ethernet without expensive optics ensured that the Ethernet remained entrenched as a local area technology.

determine which output the packet needs to be placed on. One way to do this would be to put an ordered list of switch ports in the header and to rotate the list so that the next switch in the path is always at the front of the list. Figure 3.7 illustrates this idea.

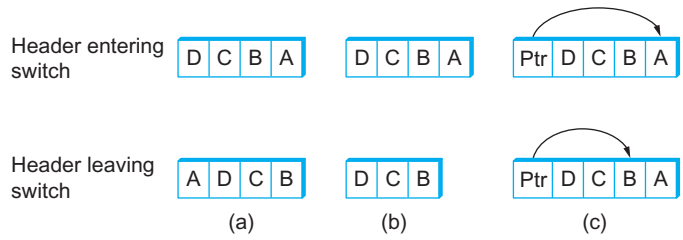
In this example, the packet needs to traverse three switches to get from host A to host B. At switch 1, it needs to exit on port 1, at the next switch it needs to exit at port 0, and at the third switch it needs to exit at port 3. Thus, the original header when the packet leaves host A contains the list of ports (3, 0, 1), where we assume that each switch reads the rightmost element of the list. To make sure that the next switch gets the appropriate information, each switch rotates the list after it has read its own entry. Thus, the packet header as it leaves switch 1 en route to switch 2 is now



■ **FIGURE 3.7** Source routing in a switched network (where the switch reads the rightmost number).

(1, 3, 0); switch 2 performs another rotation and sends out a packet with (0, 1, 3) in the header. Although not shown, switch 3 performs yet another rotation, restoring the header to what it was when host A sent it.

There are several things to note about this approach. First, it assumes that host A knows enough about the topology of the network to form a header that has all the right directions in it for every switch in the path. This is somewhat analogous to the problem of building the forwarding tables in a datagram network or figuring out where to send a setup packet in a virtual circuit network. Second, observe that we cannot predict how big the header needs to be, since it must be able to hold one word of information for every switch on the path. This implies that headers are probably of variable length with no upper bound, unless we can predict with absolute certainty the maximum number of switches through which a packet will ever need to pass. Third, there are some variations on this approach. For example, rather than rotate the header, each switch could just strip the first element as it uses it. Rotation has an advantage over stripping, however: Host B gets a copy of the complete header, which may help it figure out how to get back to host A. Yet another alternative is to have the header carry a pointer to the current "next port" entry, so that each switch just updates the pointer rather than rotating the header; this may be more efficient to implement. We show these three approaches in



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

Figure 3.8. In each case, the entry that this switch needs to read is A, and the entry that the next switch needs to read is B.

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

Source routes are sometimes categorized as "strict" or "loose." In a strict source route, every node along the path must be specified, whereas a loose source route only specifies a set of nodes to be traversed, without saying exactly how to get from one node to the next. A loose source route can be thought of as a set of waypoints rather than a completely specified route. The loose option can be helpful to limit the amount of information that a source must obtain to create a source route. In any reasonably large network, it is likely to be hard for a host to get the complete path information it needs to construct a correct strict source route to any destination. But both types of source routes do find application in certain scenarios, one of which is described in Section 4.3.

3.1.4 Bridges and LAN Switches

LAB 04: VLAN

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 that is used to forward packets between LANs (local area networks) such as Ethernets. Such switches are sometimes known by the obvious name of LAN switches; historically, they have also

been referred to as *bridges*, and they are very widely used in campus and enterprise networks.

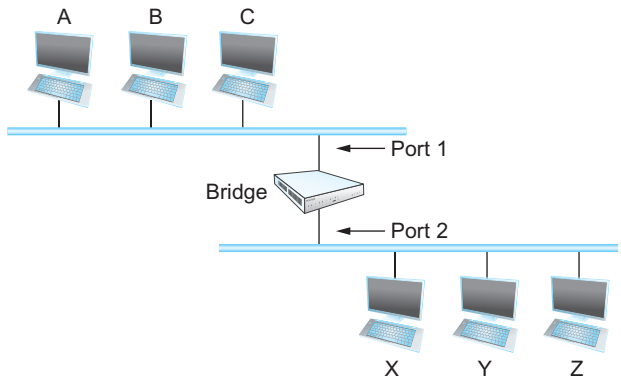
Suppose you have a pair of Ethernets that you want to interconnect. One approach you might try is to put a repeater between them, as described in Chapter 2. This would not be a workable solution, however, if doing so exceeded the physical limitations of the Ethernet. (Recall that no more than two repeaters between any pair of hosts and no more than a total of 2500 m in length are allowed.) An alternative would be to put a node with a pair of Ethernet adaptors between the two Ethernets and have the node forward frames from one Ethernet to the other. This node would differ from a repeater, which operates on bits, not frames, and just blindly copies the bits received on one interface to another. Instead, this node would fully implement the Ethernet's collision detection and media access protocols on each interface. Hence, the length and number-of-host restrictions of the Ethernet, which are all about managing collisions, would not apply to the combined pair of Ethernets connected in this way. This device operates in promiscuous mode, accepting all frames transmitted on either of the Ethernets, and forwarding them to the other.

The node we have just described is typically called a *bridge*, and a collection of LANs connected by one or more bridges is usually said to form an *extended LAN*. In their simplest variants, bridges simply accept LAN frames on their inputs and forward them out on all other outputs. This simple strategy was used by early bridges but has some pretty serious limitations as we'll see below. A number of refinements have been added over the years to make bridges an effective mechanism for interconnecting a set of LANs. The rest of this section fills in the more interesting details.

Note that a bridge meets our definition of a switch from the previous section: a multi-input, multi-output device, which transfers packets from an input to one or more outputs. And recall that this provides a way to increase the total bandwidth of a network. For example, while a single Ethernet segment might carry only 100 Mbps of total traffic, an Ethernet bridge can carry as much as $100n$ Mbps, where n is the number of ports (inputs and outputs) on the bridge.

Learning Bridges

The first optimization we can make to a bridge is to observe that it need not forward all frames that it receives. Consider the bridge in Figure 3.9. Whenever a frame from host A that is addressed to host B arrives on port 1,



■ FIGURE 3.9 Illustration of a learning bridge.

Table 3.4 Forwarding Table Maintained by a Bridge

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

there is no need for the bridge to forward the frame out over port 2. The question, then, is how does a bridge come to learn on which port the various hosts reside?

One option would be to have a human download a table into the bridge similar to the one given in Table 3.4. Then, whenever the bridge receives a frame on port 1 that is addressed to host A, it would not forward the frame out on port 2; there would be no need because host A would have already directly received the frame on the LAN connected to port 1. Anytime a frame addressed to host A was received on port 2, the bridge would forward the frame out on port 1.

No one actually builds bridges in which the table is configured by hand. Having a human maintain this table is too burdensome, and there is a

simple trick by which a bridge can learn this information for itself. The idea is for each bridge to inspect the *source* address in all the frames it receives. Thus, when host A sends a frame to a host on either side of the bridge, the bridge receives this frame and records the fact that a frame from host A was just received on port 1. In this way, the bridge can build a table just like Table 3.4.

Note that a bridge using such a table implements a version of the datagram (or connectionless) model of forwarding described in Section 3.1.1. Each packet carries a global address, and the bridge decides which output to send a packet on by looking up that address in a table.

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

Implementation

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

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

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

```
} BridgeEntry;

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

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

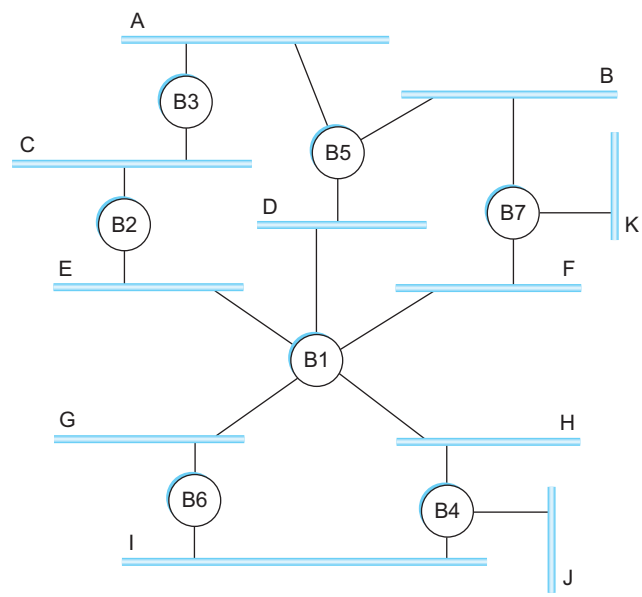
```
void
updateTable (MacAddr src, int inif)
{
    BridgeEntry       *b;

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

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

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

Spanning Tree Algorithm

The preceding strategy works just fine until the extended LAN has a loop in it, in which case it fails in a horrible way—frames potentially loop through the extended LAN forever. This is easy to see in the example depicted in Figure 3.10, where, for example, bridges B1, B4, and B6 form



■ **FIGURE 3.10** Extended LAN with loops.

a loop. Suppose that a packet enters bridge B4 from Ethernet J and that the destination address is one not yet in any bridge's forwarding table: B4 sends a copy of the packet out to Ethernets H and I. Now bridge B6 forwards the packet to Ethernet G, where B1 would see it and forward it back to Ethernet H; B4 still doesn't have this destination in its table, so it forwards the packet back to Ethernets I and J. There is nothing to stop this cycle from repeating endlessly, with packets looping in both directions among B1, B4, and B6.

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

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

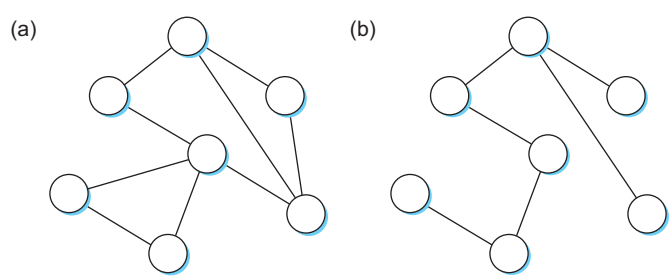


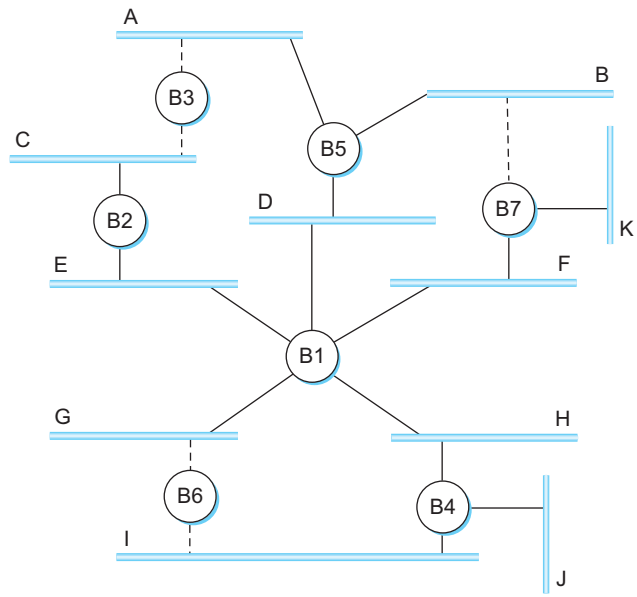
FIGURE 3.11 Example of (a) a cyclic graph; (b) a corresponding spanning tree.

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

The spanning tree algorithm, which was developed by Radia Perlman at the Digital Equipment Corporation, is a protocol used by a set of bridges to agree upon a spanning tree for a particular extended LAN. (The IEEE 802.1 specification for LAN bridges is based on this algorithm.) In practice, this means that each bridge decides the ports over which it is and is not willing to forward frames. In a sense, it is by removing ports from the topology that the extended LAN is reduced to an acyclic tree.[4] It is even possible that an entire bridge will not participate in forwarding frames, which seems kind of strange at first glance. The algorithm is dynamic, however, meaning that the bridges are always prepared to reconfigure themselves into a new spanning tree should some bridge fail, and so those unused ports and bridges provide the redundant capacity needed to recover from failures.

The main idea of the spanning tree is for the bridges to select the ports over which they will forward frames. The algorithm selects ports as follows. Each bridge has a unique identifier; for our purposes, we use the labels B1, B2, B3, and so on. The algorithm first elects the bridge with the smallest ID as the root of the spanning tree; exactly how this election takes place is described below. The root bridge always forwards frames out over all of its ports. Next, each bridge computes the shortest path to the root and notes which of its ports is on this path. This port is also selected as the bridge's preferred path to the root. Finally, all the bridges connected to a given LAN elect a single *designated* bridge that will be responsible for forwarding frames toward the root bridge. Each LAN's designated bridge is the one that is closest to the root. If two or more bridges are equally close to the root, then the bridges' identifiers are used to break ties, and

[4]Representing an extended LAN as an abstract graph is a bit awkward. Basically, you let both the bridges and the LANs correspond to the vertices of the graph and the ports correspond to the graph's edges. However, the spanning tree we are going to compute for this graph needs to span only those nodes that correspond to networks. It is possible that nodes corresponding to bridges will be disconnected from the rest of the graph. This corresponds to a situation in which all the ports connecting a bridge to various networks get removed by the algorithm.



■ **FIGURE 3.12** Spanning tree with some ports not selected.

the smallest ID wins. Of course, each bridge is connected to more than one LAN, so it participates in the election of a designated bridge for each LAN it is connected to. In effect, this means that each bridge decides if it is the designated bridge relative to each of its ports. The bridge forwards frames over those ports for which it is the designated bridge.

Figure 3.12 shows the spanning tree that corresponds to the extended LAN shown in Figure 3.10. In this example, B1 is the root bridge, since it has the smallest ID. Notice that both B3 and B5 are connected to LAN A, but B5 is the designated bridge since it is closer to the root. Similarly, both B5 and B7 are connected to LAN B, but in this case B5 is the designated bridge since it has the smaller ID; both are an equal distance from B1.

While it is possible for a human to look at the extended LAN given in Figure 3.10 and to compute the spanning tree given in Figure 3.12 according to the rules given above, the bridges in an extended LAN do not have the luxury of being able to see the topology of the entire network, let alone peek inside other bridges to see their ID. Instead, the bridges have to exchange configuration messages with each other and then decide

whether or not they are the root or a designated bridge based on these messages.

Specifically, the configuration messages contain three pieces of information:

1. The ID for the bridge that is sending the message
2. The ID for what the sending bridge believes to be the root bridge
3. The distance, measured in hops, from the sending bridge to the root bridge

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

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

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

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

When a bridge receives a configuration message indicating that it is not the root bridge—that is, a message from a bridge with a smaller ID—the bridge stops generating configuration messages on its own and instead only forwards configuration messages from other bridges, after first adding 1 to the distance field. Likewise, when a bridge receives a configuration message that indicates it is not the designated bridge for that port—that is, a message from a bridge that is closer to the root or equally far from the root but with a smaller ID—the bridge stops sending

configuration messages over that port. Thus, when the system stabilizes, only the root bridge is still generating configuration messages, and the other bridges are forwarding these messages only over ports for which they are the designated bridge. At this point, a spanning tree has been built, and all the bridges are in agreement on which ports are in use for the spanning tree. Only those ports may be used for forwarding data packets in the extended LAN.

Let's see how this works with an example. Consider what would happen in Figure 3.12 if the power had just been restored to the building housing this network, so that all the bridges boot at about the same time. All the bridges would start off by claiming to be the root. We denote a configuration message from node X in which it claims to be distance d from root node Y as (Y, d, X). Focusing on the activity at node B3, a sequence of events would unfold as follows:

1. B3 receives (B2, 0, B2).
2. Since $2 < 3$, B3 accepts B2 as root.
3. B3 adds one to the distance advertised by B2 (0) and thus sends (B2, 1, B3) toward B5.
4. Meanwhile, B2 accepts B1 as root because it has the lower ID, and it sends (B1, 1, B2) toward B3.
5. B5 accepts B1 as root and sends (B1, 1, B5) toward B3.
6. B3 accepts B1 as root, and it notes that both B2 and B5 are closer to the root than it is; thus, B3 stops forwarding messages on both its interfaces.

This leaves B3 with both ports not selected, as shown in Figure 3.12.

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

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

Broadcast and Multicast

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

Multicast can be implemented in exactly the same way, with each host deciding for itself whether or not to accept the message. This is exactly what is done in practice. Notice, however, that since not all the LANs in an extended LAN necessarily have a host that is a member of a particular multicast group, it is possible to do better. Specifically, the spanning tree algorithm can be extended to prune networks over which multicast frames need not be forwarded. Consider a frame sent to group M by a host on LAN A in Figure 3.12. If there is no host on LAN J that belongs to group M, then there is no need for bridge B4 to forward the frames over that network. On the other hand, not having a host on LAN H that belongs to group M does not necessarily mean that bridge B1 can avoid forwarding multicast frames onto LAN H. It all depends on whether or not there are members of group M on LANs I and J.

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

Note that, although the multicast extension just described has been proposed, it is not widely adopted. Instead, multicast is implemented in exactly the same way as broadcast on today's extended LANs.

Limitations of Bridges

The bridge-based solution just described is meant to be used in only a fairly limited setting—to connect a handful of similar LANs. The main

limitations of bridges become apparent when we consider the issues of scale and heterogeneity.

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

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

We can see how VLANs work with an example. Figure 3.13 shows four hosts on four different LAN segments. In the absence of VLANs, any broadcast packet from any host will reach all the other hosts. Now let's suppose that we define the segments connected to hosts W and X as being in one VLAN, which we'll call VLAN 100. We also define the segments that connect to hosts Y and Z as being in VLAN 200. To do this, we need to

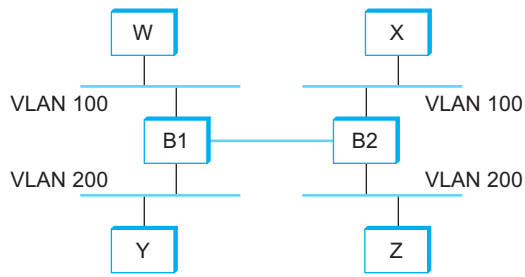


FIGURE 3.13 Two virtual LANs share a common backbone.

configure a VLAN ID on each port of bridges B1 and B2. The link between B1 and B2 is considered to be in both VLANs.

When a packet sent by host X arrives at bridge B2, the bridge observes that it came in a port that was configured as being in VLAN 100. It inserts a VLAN header between the Ethernet header and its payload. The interesting part of the VLAN header is the VLAN ID; in this case, that ID is set to 100. The bridge now applies its normal rules for forwarding to the packet, with the extra restriction that the packet may not be sent out an interface that is not part of VLAN 100. Thus, under no circumstances will the packet—even a broadcast packet—be sent out the interface to host Z, which is in VLAN 200. The packet is, however, forwarded on to bridge B1, which follows the same rules and thus may forward the packet to host W but not to host Y.

An attractive feature of VLANs is that it is possible to change the logical topology without moving any wires or changing any addresses. For example, if we wanted to make the segment that connects to host Z be part of VLAN 100 and thus enable X, W, and Z to be on the same virtual LAN, then we would just need to change one piece of configuration on bridge B2.

On the issue of heterogeneity, bridges are fairly limited in the kinds of networks they can interconnect. In particular, bridges make use of the network's frame header and so can support only networks that have exactly the same format for addresses. Thus, bridges can be used to connect Ethernets to Ethernets, token rings to token rings, and one 802.11 network to another. It's also possible to put a bridge between, say, an Ethernet and an 802.11 network, since both networks support the same 48-bit address format. However, bridges do not readily generalize to other kinds of networks with different addressing formats, such as ATM.[5]

Despite their limitations, bridges are a very important part of the complete networking picture. Their main advantage is that they allow multiple LANs to be transparently connected; that is, the networks can be connected without the end hosts having to run any additional protocols (or even be aware, for that matter). The one potential exception is when the hosts are expected to announce their membership in a multicast group, as described in Section 3.1.4.

[5]Ultimately there was quite a lot of work done to make ATM networks behave more like Ethernets, called *LAN Emulation*, to get around this limitation, but this is rarely seen today.

Notice, however, that this transparency can be dangerous. If a host or, more precisely, the application and transport protocol running on that host is programmed under the assumption that it is running on a single LAN, then inserting bridges between the source and destination hosts can have unexpected consequences. For example, if a bridge becomes congested, it may have to drop frames; in contrast, it is rare that a single Ethernet ever drops a frame. As another example, the latency between any pair of hosts on an extended LAN becomes both larger and more highly variable; in contrast, the physical limitations of a single Ethernet make the latency both small and predictable. As a final example, it is possible (although unlikely) that frames will be reordered in an extended LAN; in contrast, frame order is never shuffled on a single Ethernet. The bottom line is that it is never safe to design network software under the assumption that it will run over a single Ethernet segment. Bridges happen.


LAB 05:
Net Design

3.2 BASIC INTERNETWORKING (IP)

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

3.2.1 What Is an Internetwork?

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

principles of "lowercase *i*" internetworking, but we illustrate these ideas with real-world examples from the "big *I*" Internet.

Another piece of terminology that can be confusing is the difference between networks, subnetworks, and internetworks. We are going to avoid subnetworks (or subnets) altogether until Section 3.2.5. For now, we use *network* to mean either a directly connected or a switched network of the kind described in the previous section and the previous chapter. Such a network uses one technology, such as 802.11 or Ethernet. An *internetwork* is an interconnected collection of such networks. Sometimes, to avoid ambiguity, we refer to the underlying networks that we are interconnecting as *physical* networks. An internet is a *logical* network built out of a collection of physical networks. In this context, a collection of Ethernets connected by bridges or switches would still be viewed as a single network.

Figure 3.14 shows an example internetwork. An internetwork is often referred to as a "network of networks" because it is made up of lots of smaller networks. In this figure, we see Ethernets, a wireless network,

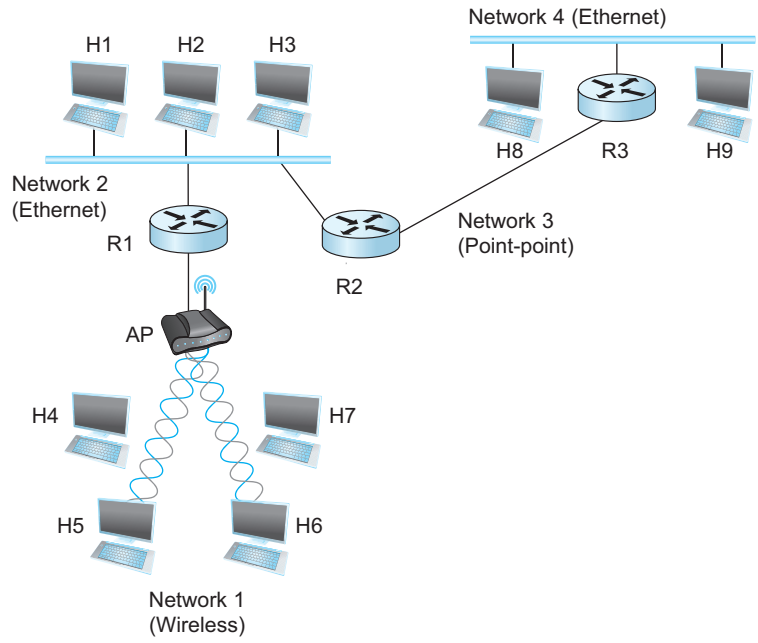


FIGURE 3.14 A simple internetwork. H*n* = host; R*n* = router.

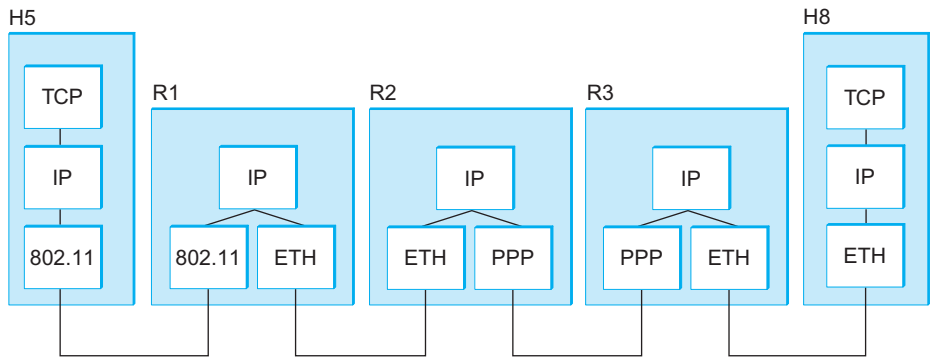


FIGURE 3.15 A simple internetwork, showing the protocol layers used to connect H5 to H8 in Figure 3.14. ETH is the protocol that runs over the Ethernet.

and a point-to-point link. Each of these is a single-technology network. The nodes that interconnect the networks are called *routers*. They are also sometimes called *gateways*, but since this term has several other connotations, we restrict our usage to router.

The *Internet Protocol* is the key tool used today to build scalable, heterogeneous internetworks. It was originally known as the Kahn-Cerf protocol after its inventors.[6] One way to think of IP is that it runs on all the nodes (both hosts and routers) in a collection of networks and defines the infrastructure that allows these nodes and networks to function as a single logical internetwork. For example, Figure 3.15 shows how hosts H5 and H8 are logically connected by the internet in Figure 3.14, including the protocol graph running on each node. Note that higher-level protocols, such as TCP and UDP, typically run on top of IP on the hosts.

Most of the rest of this chapter is about various aspects of IP. While it is certainly possible to build an internetwork that does not use IP—for example, Novell created an internetworking protocol called IPX, which was in turn based on the XNS internet designed by Xerox—IP is the most interesting case to study simply because of the size of the Internet. Said another way, it is only the IP Internet that has really faced the issue of scale. Thus, it provides the best case study of a scalable internetworking protocol.

[6]Robert Kahn and Vint Cerf received the A.M. Turing award, often referred to as the Nobel Prize of computer science, in 2005 for their design of IP.

3.2.2 Service Model

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

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

Datagram Delivery

The IP datagram is fundamental to the Internet Protocol. Recall from Section 3.1.1 that a datagram is a type of packet that happens to be sent in a connectionless manner over a network. Every datagram carries enough information to let the network forward the packet to its correct destination; there is no need for any advance setup mechanism to tell the network what to do when the packet arrives. You just send it, and the network makes its best effort to get it to the desired destination. The "best-effort" part means that if something goes wrong and the packet gets lost, corrupted, misdelivered, or in any way fails to reach its intended destination, the network does nothing—it made its best effort, and that is all it has to do. It does not make any attempt to recover from the failure. This is sometimes called an *unreliable* service.

Best-effort, connectionless service is about the simplest service you could ask for from an internetwork, and this is a great strength. For example, if you provide best-effort service over a network that provides a reliable service, then that's fine—you end up with a best-effort service that just happens to always deliver the packets. If, on the other hand, you had a reliable service model over an unreliable network, you would

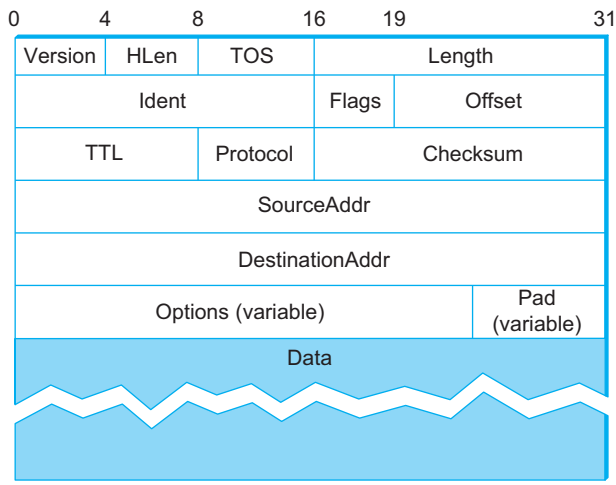
have to put lots of extra functionality into the routers to make up for the deficiencies of the underlying network. Keeping the routers as simple as possible was one of the original design goals of IP.

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

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

Packet Format

Clearly, a key part of the IP service model is the type of packets that can be carried. The IP datagram, like most packets, consists of a header followed by a number of bytes of data. The format of the header is shown in Figure 3.16. Note that we have adopted a different style of representing packets than the one we used in previous chapters. This is because packet formats at the internetworking layer and above, where we will be focusing our attention



■ **FIGURE 3.16** IPv4 packet header.

for the next few chapters, are almost invariably designed to align on 32-bit boundaries to simplify the task of processing them in software. Thus, the common way of representing them (used in Internet Requests for Comments, for example) is to draw them as a succession of 32-bit words. The top word is the one transmitted first, and the leftmost byte of each word is the one transmitted first. In this representation, you can easily recognize fields that are a multiple of 8 bits long. On the odd occasion when fields are not an even multiple of 8 bits, you can determine the field lengths by looking at the bit positions marked at the top of the packet.

Looking at each field in the IP header, we see that the "simple" model of best-effort datagram delivery still has some subtle features. The Version field specifies the version of IP. The current version of IP is 4, and it is sometimes called *IPv4*.[7] Observe that putting this field right at the start of the datagram makes it easy for everything else in the packet format to be redefined in subsequent versions; the header processing software starts off by looking at the version and then branches off to process the rest of the packet according to the appropriate format. The next field, HLen, specifies the length of the header in 32-bit words. When there are no options, which is most of the time, the header is 5 words (20 bytes) long. The 8-bit TOS (type of service) field has had a number of different definitions over the years, but its basic function is to allow packets to be treated differently based on application needs. For example, the TOS value might determine whether or not a packet should be placed in a special queue that receives low delay. We discuss the use of this field (which has evolved somewhat over the years) in more detail in Sections 6.4.2 and 6.5.3.

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

Moving on to the third word of the header, the next byte is the TTL (time to live) field. Its name reflects its historical meaning rather than the

[7]The next major version of IP, which is discussed in Chapter 4, has the new version number 6 and is known as IPv6. The version number 5 was used for an experimental protocol called ST-II that was not widely used.

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

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

The Checksum is calculated by considering the entire IP header as a sequence of 16-bit words, adding them up using ones complement arithmetic, and taking the ones complement of the result. This is the IP checksum algorithm described in Section 2.4. Thus, if any bit in the header is corrupted in transit, the checksum will not contain the correct value upon receipt of the packet. Since a corrupted header may contain an error in the destination address—and, as a result, may have been misdelivered—it makes sense to discard any packet that fails the checksum. It should be noted that this type of checksum does not have the same strong error detection properties as a CRC, but it is much easier to calculate in software.

The last two required fields in the header are the SourceAddr and the DestinationAddr for the packet. The latter is the key to datagram delivery: Every packet contains a full address for its intended destination so that forwarding decisions can be made at each router. The source address is required to allow recipients to decide if they want to accept the packet and to enable them to reply. IP addresses are discussed in Section 3.2.3—for now, the important thing to know is that IP defines its own global address space, independent of whatever physical networks it runs over. As we will see, this is one of the keys to supporting heterogeneity.

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

Fragmentation and Reassembly

One of the problems of providing a uniform host-to-host service model over a heterogeneous collection of networks is that each network technology tends to have its own idea of how large a packet can be. For example, an Ethernet can accept packets up to 1500 bytes long, while FDDI (Fiber Distributed Data Interface) packets may be 4500 bytes long. This leaves two choices for the IP service model: Make sure that all IP datagrams are small enough to fit inside one packet on any network technology, or provide a means by which packets can be fragmented and reassembled when they are too big to go over a given network technology. The latter turns out to be a good choice, especially when you consider the fact that new network technologies are always turning up, and IP needs to run over all of them; this would make it hard to pick a suitably small bound on datagram size. This also means that a host will not send needlessly small packets, which wastes bandwidth and consumes processing resources by requiring more headers per byte of data sent. For example, two hosts connected to FDDI networks that are interconnected by a point-to-point link would not need to send packets small enough to fit on an Ethernet.

The central idea here is that every network type has a *maximum transmission unit* (MTU), which is the largest IP datagram that it can carry in a frame. Note that this value is smaller than the largest packet size on that network because the IP datagram needs to fit in the *payload* of the link-layer frame.[8]

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

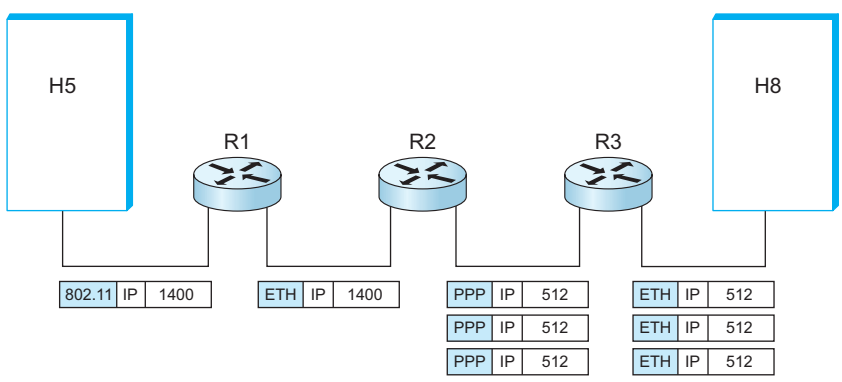
Fragmentation typically occurs in a router when it receives a datagram that it wants to forward over a network that has an MTU that is smaller

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

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

To see what this all means, consider what happens when host H5 sends a datagram to host H8 in the example internet shown in Figure 3.14. Assuming that the MTU is 1500 bytes for the two Ethernets and the 802.11 network, and 532 bytes for the point-to-point network, then a 1420-byte datagram (20-byte IP header plus 1400 bytes of data) sent from H5 makes it across the 802.11 network and the first Ethernet without fragmentation but must be fragmented into three datagrams at router R2. These three fragments are then forwarded by router R3 across the second Ethernet to the destination host. This situation is illustrated in Figure 3.17. This figure also serves to reinforce two important points:

1. Each fragment is itself a self-contained IP datagram that is transmitted over a sequence of physical networks, independent of the other fragments.
2. Each IP datagram is re-encapsulated for each physical network over which it travels.



■ **FIGURE 3.17** IP datagrams traversing the sequence of physical networks graphed in Figure 3.14.

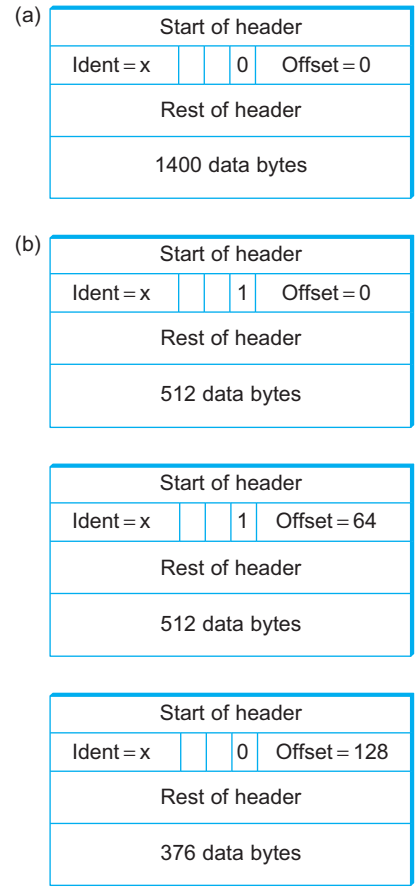


FIGURE 3.18 Header fields used in IP fragmentation: (a) unfragmented packet; (b) fragmented packets.

The fragmentation process can be understood in detail by looking at the header fields of each datagram, as is done in Figure 3.18. The unfragmented packet, shown at the top, has 1400 bytes of data and a 20-byte IP header. When the packet arrives at router R2, which has an MTU of 532 bytes, it has to be fragmented. A 532-byte MTU leaves 512 bytes for data after the 20-byte IP header, so the first fragment contains 512 bytes of data. The router sets the M bit in the Flags field (see Figure 3.16), meaning that there are more fragments to follow, and it sets the Offset to 0, since this fragment contains the first part of the original datagram. The data carried in the second fragment starts with the 513th byte of the original

data, so the Offset field in this header is set to 64, which is $512 \div 8$. Why the division by 8? Because the designers of IP decided that fragmentation should always happen on 8-byte boundaries, which means that the Offset field counts 8-byte chunks, not bytes. (We leave it as an exercise for you to figure out why this design decision was made.) The third fragment contains the last 376 bytes of data, and the offset is now $2 \times 512 \div 8 = 128$. Since this is the last fragment, the M bit is not set.

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

IP reassembly is far from a simple process. For example, if a single fragment is lost, the receiver will still attempt to reassemble the datagram, and it will eventually give up and have to garbage-collect the resources that were used to perform the failed reassembly.[9] For this reason, among others, IP fragmentation is generally considered a good thing to avoid. Hosts are now strongly encouraged to perform "path MTU discovery," a process by which fragmentation is avoided by sending packets that are small enough to traverse the link with the smallest MTU in the path from sender to receiver.

3.2.3 Global Addresses

In the above discussion of the IP service model, we mentioned that one of the things that it provides is an addressing scheme. After all, if you want to be able to send data to any host on any network, there needs to be a way of identifying all the hosts. Thus, we need a global addressing scheme—one in which no two hosts have the same address. Global uniqueness is the first property that should be provided in an addressing scheme.[10]

Ethernet addresses are globally unique, but that alone does not suffice for an addressing scheme in a large internetwork. Ethernet addresses are also *flat*, which means that they have no structure and provide very few clues to routing protocols. (In fact, Ethernet addresses do have a

[9] As we will see in Chapter 8, getting a host to tie up resources needlessly can be the basis of a denial-of-service attack.

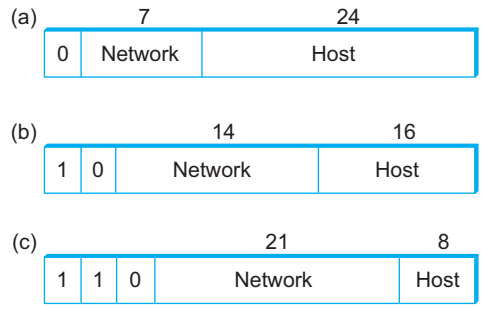
[10] For better or worse, global addressing isn't guaranteed anymore in the modern Internet, for a range of reasons, touched on in Section 4.1.

structure for the purposes of *assignment*—the first 24 bits identify the manufacturer—but this provides no useful information to routing protocols since this structure has nothing to do with network topology.) In contrast, IP addresses are *hierarchical*, by which we mean that they are made up of several parts that correspond to some sort of hierarchy in the internetwork. Specifically, IP addresses consist of two parts, usually referred to as a *network* part and a *host* part. This is a fairly logical structure for an internetwork, which is made up of many interconnected networks. The network part of an IP address identifies the network to which the host is attached; all hosts attached to the same network have the same network part in their IP address. The host part then identifies each host uniquely on that particular network. Thus, in the simple internetwork of Figure 3.14, the addresses of the hosts on network 1, for example, would all have the same network part and different host parts.

Note that the routers in Figure 3.14 are attached to two networks. They need to have an address on each network, one for each interface. For example, router R1, which sits between the wireless network and an Ethernet, has an IP address on the interface to the wireless network whose network part is the same as all the hosts on that network. It also has an IP address on the interface to the Ethernet that has the same network part as the hosts on that Ethernet. Thus, bearing in mind that a router might be implemented as a host with two network interfaces, it is more precise to think of IP addresses as belonging to interfaces than to hosts.

Now, what do these hierarchical addresses look like? Unlike some other forms of hierarchical address, the sizes of the two parts are not the same for all addresses. Originally, IP addresses were divided into three different classes, as shown in Figure 3.19, each of which defines different-sized network and host parts. (There are also class D addresses that specify a multicast group, discussed in Section 4.2, and class E addresses that are currently unused.) In all cases, the address is 32 bits long.

The class of an IP address is identified in the most significant few bits. If the first bit is 0, it is a class A address. If the first bit is 1 and the second is 0, it is a class B address. If the first two bits are 1 and the third is 0, it is a class C address. Thus, of the approximately 4 billion possible IP addresses, half are class A, one-quarter are class B, and one-eighth are class C. Each class allocates a certain number of bits for the network part of the address and the rest for the host part. Class A networks have 7 bits



■ **FIGURE 3.19** IP addresses: (a) class A; (b) class B; (c) class C.

for the network part and 24 bits for the host part, meaning that there can be only 126 class A networks (the values 0 and 127 are reserved), but each of them can accommodate up to $2^{24} - 2$ (about 16 million) hosts (again, there are two reserved values). Class B addresses allocate 14 bits for the network and 16 bits for the host, meaning that each class B network has room for 65,534 hosts. Finally, class C addresses have only 8 bits for the host and 21 for the network part. Therefore, a class C network can have only 256 unique host identifiers, which means only 254 attached hosts (one host identifier, 255, is reserved for broadcast, and 0 is not a valid host number). However, the addressing scheme supports 2^{21} class C networks.

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

Before we look at how IP addresses get used, it is helpful to look at some practical matters, such as how you write them down. By convention, IP addresses are written as four *decimal* integers separated by dots. Each integer represents the decimal value contained in 1 byte of the address, starting at the most significant. For example, the address of the computer on which this sentence was typed is 171.69.210.245.

It is important not to confuse IP addresses with Internet domain names, which are also hierarchical. Domain names tend to be ASCII

strings separated by dots, such as cs.princeton.edu. We will be talking about those in Section 9.3.1. The important thing about IP addresses is that they are what is carried in the headers of IP packets, and it is those addresses that are used in IP routers to make forwarding decisions.

3.2.4 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 Section 3.1 that *forwarding* is the process of taking a packet from an input and sending it out on the appropriate output, while *routing* is the process of building up the tables that allow the correct output for a packet to be determined. The discussion here focuses on forwarding; we take up routing in Section 3.3.

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

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

Forwarding IP datagrams can therefore be handled in the following way. A datagram is sent from a source host to a destination host, possibly passing through several routers along the way. Any node, whether it is a host or a router, first tries to establish whether it is connected to the same physical network as the destination. To do this, it compares the network part of the destination address with the network part of the address of each of its network interfaces. (Hosts normally have only one interface, while routers normally have two or more, since they are typically connected to two or more networks.) If a match occurs, then that means that the destination lies on the same physical network as the interface, and the packet can be directly delivered over that network. Section 3.2.6 explains some of the details of this process.

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

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

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

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

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

Let's see how this works in the example internetwork of Figure 3.14. First, suppose that H1 wants to send a datagram to H2. Since they are on the same physical network, H1 and H2 have the same network number in their IP address. Thus, H1 deduces that it can deliver the datagram directly to H2 over the Ethernet. The one issue that needs to be resolved is

how H1 finds out the correct Ethernet address for H2—this is the address resolution mechanism described in Section 3.2.6.

Bridges, Switches, and Routers

It is easy to become confused about the distinction between bridges, switches, and routers. There is good reason for such confusion, since at some level they all forward messages from one link to another. One distinction people make is based on layering: Bridges are link-level nodes (they forward frames from one link to another to implement an extended LAN), switches are network-level nodes (they forward packets from one link to another to implement a packet-switched network), and routers are internet-level nodes (they forward datagrams from one network to another to implement an internet).

The distinction between bridges and switches is now pretty much obsolete. For example, we have already seen that a multiport bridge is usually called an Ethernet switch or LAN switch. For this reason, bridges and switches are often grouped together as "layer 2 devices," where layer 2 in this context means "above the physical layer, below the internet layer."

Historically, there have been important distinctions between LAN switches (or bridges) and WAN switches (such as those based on ATM or Frame Relay). LAN switches traditionally depend on the spanning tree algorithm, while WAN switches generally run routing protocols that allow each switch to learn the topology of the whole network. This is an important distinction because knowing the whole network topology allows the switches to discriminate among different routes, while, in contrast, the spanning tree algorithm locks in a single tree over which messages are forwarded. It is also the case that the spanning tree approach does not scale as well. Again, this distinction is under threat as routing protocols from the wide area start to make their way into LAN switches.

What about switches and routers? Internally, they look quite similar (as the section on switch and router implementation will illustrate). The key distinction is the sort of packet they forward: IP datagrams in the case of routers and Layer 2 packets (Ethernet frames or ATM cells) in the case of switches.

One big difference between a network built from switches and the Internet built from routers is that the Internet is able to accommodate heterogeneity, whereas switched networks typically consists of homogeneous links. This support for 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.

Now suppose H5 wants to send a datagram to H8. Since these hosts are on different physical networks, they have different network numbers, so H5 deduces that it needs to send the datagram to a router. R1 is the only choice—the default router—so H1 sends the datagram over the wireless network to R1. Similarly, R1 knows that it cannot deliver a datagram directly to H8 because neither of R1's interfaces is on the same network as H8. Suppose R1's default router is R2; R1 then sends the datagram to R2 over the Ethernet. Assuming R2 has the forwarding table shown in Table 3.5, it looks up H8's network number (network 4) and forwards the datagram over the point-to-point network to R3. Finally, R3, since it is on the same network as H8, forwards the datagram directly to H8.

Table 3.5 Example Forwarding Table for Router R2 in Figure 3.14

NetworkNum	NextHop
1	R1
4	R3

Table 3.6 Complete Forwarding Table for Router R2 in Figure 3.14

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

Note that it is possible to include the information about directly connected networks in the forwarding table. For example, we could label the network interfaces of router R2 as interface 0 for the point-to-point link (network 3) and interface 1 for the Ethernet (network 2). Then R2 would have the forwarding table shown in Table 3.6.

Thus, for any network number that R2 encounters in a packet, it knows what to do. Either that network is directly connected to R2, in which case the packet can be delivered to its destination over that network, or the

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

The forwarding table used by R2 is simple enough that it could be manually configured. Usually, however, these tables are more complex and would be built up by running a routing protocol such as one of those described in Section 3.3. Also note that, in practice, the network numbers are usually longer (e.g., 128.96).

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

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.

3.2.5 Subnetting and Classless Addressing

The original intent of IP addresses was that the network part would uniquely identify exactly one physical network. It turns out that this approach has a couple of drawbacks. Imagine a large campus that has lots of internal networks and decides to connect to the Internet. For every network, no matter how small, the site needs at least a class C network address. Even worse, for any network with more than 255 hosts, they need a class B address. This may not seem like a big deal, and indeed it wasn't when the Internet was first envisioned, but there are only a finite

number of network numbers, and there are far fewer class B addresses than class Cs. Class B addresses tend to be in particularly high demand because you never know if your network might expand beyond 255 nodes, so it is easier to use a class B address from the start than to have to renumber every host when you run out of room on a class C network. The problem we observe here is address assignment inefficiency: A network with two nodes uses an entire class C network address, thereby wasting 253 perfectly useful addresses; a class B network with slightly more than 255 hosts wastes over 64,000 addresses.

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

Assigning many network numbers has another drawback that becomes apparent when you think about routing. Recall that the amount of state that is stored in a node participating in a routing protocol is proportional to the number of other nodes, and that routing in an internet consists of building up forwarding tables that tell a router how to reach different networks. Thus, the more network numbers there are in use, the bigger the forwarding tables get. Big forwarding tables add costs to routers, and they are potentially slower to search than smaller tables for a given technology, so they degrade router performance. This provides another motivation for assigning network numbers carefully.

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

so one entry in your forwarding table will suffice. Even if there are multiple points at which the campus is connected to the rest of the Internet, knowing how to get to one point in the campus network is still a good start.

The mechanism by which a single network number can be shared among multiple networks involves configuring all the nodes on each subnet with a *subnet mask*. With simple IP addresses, all hosts on the same network must have the same network number. The subnet mask enables us to introduce a *subnet number*; all hosts on the same physical network will have the same subnet number, which means that hosts may be on different physical networks but share a single network number. This concept is illustrated in Figure 3.20.

What subnetting means to a host is that it is now configured with both an IP address and a subnet mask for the subnet to which it is attached. For example, host H1 in Figure 3.21 is configured with an address of 128.96.34.15 and a subnet mask of 255.255.255.128. (All hosts on a given subnet are configured with the same mask; that is, there is exactly one subnet mask per subnet.) The bitwise AND of these two numbers defines the subnet number of the host and of all other hosts on the same subnet. In this case, 128.96.34.15 AND 255.255.255.128 equals 128.96.34.0, so this is the subnet number for the topmost subnet in the figure.

When the host wants to send a packet to a certain IP address, the first thing it does is to perform a bitwise AND between its own subnet mask and the destination IP address. If the result equals the subnet number of the sending host, then it knows that the destination host is on the same subnet and the packet can be delivered directly over the subnet.

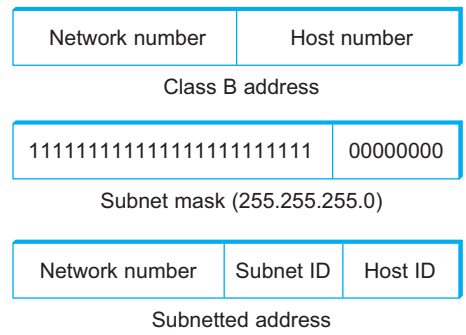
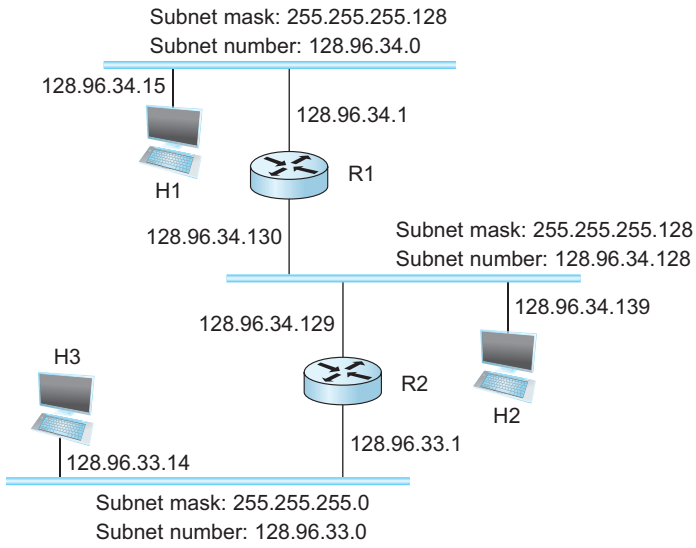


FIGURE 3.20 Subnet addressing.



■ **FIGURE 3.21** An example of subnetting.

If the results are not equal, the packet needs to be sent to a router to be forwarded to another subnet. For example, if H1 is sending to H2, then H1 ANDs its subnet mask (255.255.255.128) with the address for H2 (128.96.34.139) to obtain 128.96.34.128. This does not match the subnet number for H1 (128.96.34.0) so H1 knows that H2 is on a different subnet. Since H1 cannot deliver the packet to H2 directly over the subnet, it sends the packet to its default router R1.

The forwarding table of a router also changes slightly when we introduce subnetting. Recall that we previously had a forwarding table that consisted of entries of the form ⟨NetworkNum, NextHop⟩. To support subnetting, the table must now hold entries of the form ⟨SubnetNumber, SubnetMask, NextHop⟩. To find the right entry in the table, the router ANDs the packet's destination address with the SubnetMask for each entry in turn; if the result matches the SubnetNumber of the entry, then this is the right entry to use, and it forwards the packet to the next hop router indicated. In the example network of Figure 3.21, router R1 would have the entries shown in Table 3.7.

Continuing with the example of a datagram from H1 being sent to H2, R1 would AND H2's address (128.96.34.139) with the subnet mask of the first entry (255.255.255.128) and compare the result (128.96.34.128) with

Table 3.7 Example Forwarding Table with Subnetting for Figure 3.21

SubnetNumber	SubnetMask	NextHop
128.96.34.0	255.255.255.128	Interface 0
128.96.34.128	255.255.255.128	Interface 1
128.96.33.0	255.255.255.0	R2

the network number for that entry (128.96.34.0). Since this is not a match, it proceeds to the next entry. This time a match does occur, so R1 delivers the datagram to H2 using interface 1, which is the interface connected to the same network as H2.

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

```
D = destination IP address
for each forwarding table entry ⟨SubnetNumber, SubnetMask, NextHop⟩
  D1 = SubnetMask & D
  if D1 = SubnetNumber
    if NextHop is an interface
      deliver datagram directly to destination
    else
      deliver datagram to NextHop (a router)
```

Although not shown in this example, a default route would usually be included in the table and would be used if no explicit matches were found. We note in passing that a naive implementation of this algorithm—one involving repeated ANDing of the destination address with a subnet mask that may not be different every time, and a linear table search—would be very inefficient.

An important consequence of subnetting is that different parts of the internet see the world differently. From outside our hypothetical campus, routers see a single network. In the example above, routers outside the campus see the collection of networks in Figure 3.21 as just the network 128.96, and they keep one entry in their forwarding tables to tell them how to reach it. Routers within the campus, however, need to be able to route packets to the right subnet. Thus, not all parts of the internet see exactly the same routing information. This is an example of *aggregation*

of routing information, which is fundamental to scaling of the routing system. The next section shows how aggregation can be taken to another level.

Classless Addressing

Subnetting has a counterpart, sometimes called *supernetting*, but more often called *Classless Interdomain Routing* or CIDR, pronounced "cider." CIDR takes the subnetting idea to its logical conclusion by essentially doing away with address classes altogether. Why isn't subnetting alone sufficient? In essence, subnetting only allows us to split a classful address among multiple subnets, while CIDR allows us to coalesce several classful addresses into a single "supernet." This further tackles the address space inefficiency noted above, and does so in a way that keeps the routing system from being overloaded.

To see how the issues of address space efficiency and scalability of the routing system are coupled, consider the hypothetical case of a company whose network has 256 hosts on it. That is slightly too many for a Class C address, so you would be tempted to assign a class B. However, using up a chunk of address space that could address 65,535 to address 256 hosts has an efficiency of only $256/65{,}535 = 0.39\%$. Even though subnetting can help us to assign addresses carefully, it does not get around the fact that any organization with more than 255 hosts, or an expectation of eventually having that many, wants a class B address.

The first way you might deal with this issue would be to refuse to give a class B address to any organization that requests one unless they can show a need for something close to 64K addresses, and instead giving them an appropriate number of class C addresses to cover the expected number of hosts. Since we would now be handing out address space in chunks of 256 addresses at a time, we could more accurately match the amount of address space consumed to the size of the organization. For any organization with at least 256 hosts, we can guarantee an address utilization of at least 50%, and typically much more.

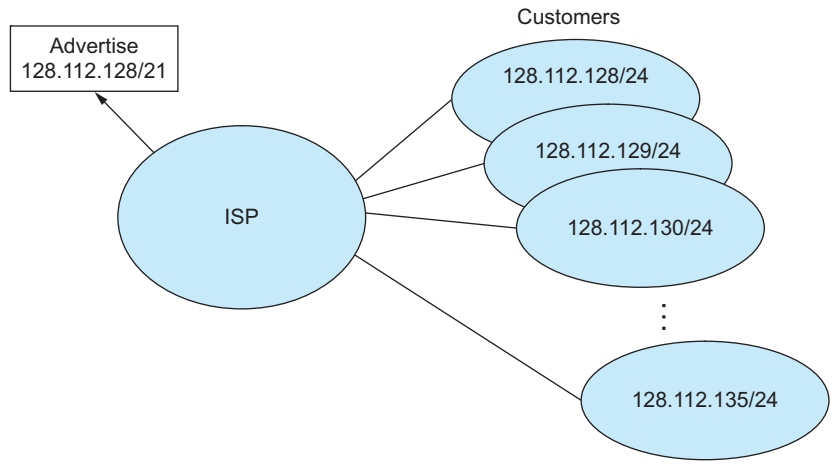
This solution, however, raises a problem that is at least as serious: excessive storage requirements at the routers. If a single site has, say, 16 class C network numbers assigned to it, that means every Internet backbone router needs 16 entries in its routing tables to direct packets to that site. This is true even if the path to every one of those networks is the same. If we had assigned a class B address to the site, the same routing

information could be stored in one table entry. However, our address assignment efficiency would then be only $16 \times 255/65{,}536 = 6.2\%$.

CIDR, therefore, tries to balance the desire to minimize the number of routes that a router needs to know against the need to hand out addresses efficiently. To do this, CIDR helps us to *aggregate* routes. That is, it lets us use a single entry in a forwarding table to tell us how to reach a lot of different networks. As noted above it does this by breaking the rigid boundaries between address classes. To understand how this works, consider our hypothetical organization with 16 class C network numbers. Instead of handing out 16 addresses at random, we can hand out a block of *contiguous* class C addresses. Suppose we assign the class C network numbers from 192.4.16 through 192.4.31. Observe that the top 20 bits of all the addresses in this range are the same (11000000 00000100 0001). Thus, what we have effectively created is a 20-bit network number—something that is between a class B network number and a class C number in terms of the number of hosts that it can support. In other words, we get both the high address efficiency of handing out addresses in chunks smaller than a class B network, and a single network prefix that can be used in forwarding tables. Observe that, for this scheme to work, we need to hand out blocks of class C addresses that share a common prefix, which means that each block must contain a number of class C networks that is a power of two.

CIDR requires a new type of notation to represent network numbers, or *prefixes* as they are known, because the prefixes can be of any length. The convention is to place a /X after the prefix, where X is the prefix length in bits. So, for the example above, the 20-bit prefix for all the networks 192.4.16 through 192.4.31 is represented as 192.4.16/20. By contrast, if we wanted to represent a single class C network number, which is 24 bits long, we would write it 192.4.16/24. Today, with CIDR being the norm, it is more common to hear people talk about "slash 24" prefixes than class C networks. Note that representing a network address in this way is similar to the ⟨mask, value⟩ approach used in subnetting, as long as masks consist of contiguous bits starting from the most significant bit (which in practice is almost always the case).

The ability to aggregate routes at the edge of the network as we have just seen is only the first step. Imagine an Internet service provider network, whose primary job is to provide Internet connectivity to a large number of corporations and campuses (customers). If we assign prefixes



■ **FIGURE 3.22** Route aggregation with CIDR.

to the customers in such a way that many different customer networks connected to the provider network share a common, shorter address prefix, then we can get even greater aggregation of routes. Consider the example in Figure 3.22. Assume that eight customers served by the provider network have each been assigned adjacent 24-bit network prefixes. Those prefixes all start with the same 21 bits. Since all of the customers are reachable through the same provider network, it can advertise a single route to all of them by just advertising the common 21-bit prefix they share. And it can do this even if not all the 24-bit prefixes have been handed out, as long as the provider ultimately *will* have the right to hand out those prefixes to a customer. One way to accomplish that is to assign a portion of address space to the provider in advance and then to let the network provider assign addresses from that space to its customers as needed. Note that, in contrast to this simple example, there is no need for all customer prefixes to be the same length.

IP Forwarding Revisited

In all our discussion of IP forwarding so far, we have assumed that we could find the network number in a packet and then look up that number in a forwarding table. However, now that we have introduced CIDR, we need to reexamine this assumption. CIDR means that prefixes may be of any length, from 2 to 32 bits. Furthermore, it is sometimes possible

to have prefixes in the forwarding table that "overlap," in the sense that some addresses may match more than one prefix. For example, we might find both 171.69 (a 16-bit prefix) and 171.69.10 (a 24-bit prefix) in the forwarding table of a single router. In this case, a packet destined to, say, 171.69.10.5 clearly matches both prefixes. The rule in this case is based on the principle of "longest match"; that is, the packet matches the longest prefix, which would be 171.69.10 in this example. On the other hand, a packet destined to 171.69.20.5 would match 171.69 and *not* 171.69.10, and in the absence of any other matching entry in the routing table 171.69 would be the longest match.

The task of efficiently finding the longest match between an IP address and the variable-length prefixes in a forwarding table has been a fruitful field of research in recent years, and the Further Reading section of this chapter provides some references. The most well-known algorithm uses an approach known as a *PATRICIA tree*, which was actually developed well in advance of CIDR.

3.2.6 Address Translation (ARP)

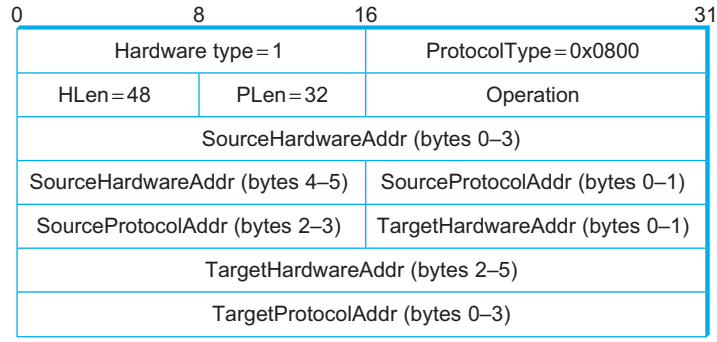
In the previous section we talked about how to get IP datagrams to the right physical network but glossed over the issue of how to get a datagram to a particular host or router on that network. The main issue is that IP datagrams contain IP addresses, but the physical interface hardware on the host or router to which you want to send the datagram only understands the addressing scheme of that particular network. Thus, we need to translate the IP address to a link-level address that makes sense on this network (e.g., a 48-bit Ethernet address). We can then encapsulate the IP datagram inside a frame that contains that link-level address and send it either to the ultimate destination or to a router that promises to forward the datagram toward the ultimate destination.

One simple way to map an IP address into a physical network address is to encode a host's physical address in the host part of its IP address. For example, a host with physical address 00100001 01001001 (which has the decimal value 33 in the upper byte and 81 in the lower byte) might be given the IP address 128.96.33.81. While this solution has been used on some networks, it is limited in that the network's physical addresses can be no more than 16 bits long in this example; they can be only 8 bits long on a class C network. This clearly will not work for 48-bit Ethernet addresses.

A more general solution would be for each host to maintain a table of address pairs; that is, the table would map IP addresses into physical addresses. While this table could be centrally managed by a system administrator and then copied to each host on the network, a better approach would be for each host to dynamically learn the contents of the table using the network. This can be accomplished using the Address Resolution Protocol (ARP). The goal of ARP is to enable each host on a network to build up a table of mappings between IP addresses and link-level addresses. Since these mappings may change over time (e.g., because an Ethernet card in a host breaks and is replaced by a new one with a new address), the entries are timed out periodically and removed. This happens on the order of every 15 minutes. The set of mappings currently stored in a host is known as the ARP cache or ARP table.

ARP takes advantage of the fact that many link-level network technologies, such as Ethernet, support broadcast. If a host wants to send an IP datagram to a host (or router) that it knows to be on the same network (i.e., the sending and receiving 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 adds the information contained in this response to its ARP table.

The query message also includes the IP address and link-layer address of the sending host. Thus, when a host broadcasts a query message, each host on the network can learn the sender's link-level and IP addresses and place that information in its ARP table. However, not every host adds this information to its ARP table. If the host already has an entry for that host in its table, it "refreshes" this entry; that is, it resets the length of time until it discards the entry. If that host is the target of the query, then it adds the information about the sender to its table, even if it did not already have an entry for that host. This is because there is a good chance that the source host is about to send it an application-level message, and it may eventually have to send a response or ACK back to the source; it will need the source's physical address to do this. If a host is not the target and does not already have an entry for the source in its ARP table, then it does not



■ **FIGURE 3.23** ARP packet format for mapping IP addresses into Ethernet addresses.

add an entry for the source. This is because there is no reason to believe that this host will ever need the source's link-level address; there is no need to clutter its ARP table with this information.

Figure 3.23 shows the ARP packet format for IP-to-Ethernet address mappings. In fact, ARP can be used for lots of other kinds of mappings—the major differences are in the address sizes. In addition to the IP and link-layer addresses of both sender and target, the packet contains

- A HardwareType field, which specifies the type of physical network (e.g., Ethernet)
- A ProtocolType field, which specifies the higher-layer protocol (e.g., IP)
- HLen ("hardware" address length) and PLen ("protocol" address length) fields, which specify the length of the link-layer address and higher-layer protocol address, respectively
- An Operation field, which specifies whether this is a request or a response
- The source and target hardware (Ethernet) and protocol (IP) addresses

Note that the results of the ARP process can be added as an extra column in a forwarding table like the one in Table 4.1. Thus, for example, when R2 needs to forward a packet to network 2, it not only finds that the next hop is R1, but also finds the MAC address to place on the packet to send it to R1.

We have now seen the basic mechanisms that IP provides for dealing with both heterogeneity and scale. On the issue of heterogeneity, IP begins by defining a best-effort service model that makes minimal assumptions about the underlying networks; most notably, this service model is based on unreliable datagrams. IP then makes two important additions to this starting point: (1) a common packet format (fragmentation/reassembly is the mechanism that makes this format work over networks with different MTUs) and (2) a global address space for identifying all hosts (ARP is the mechanism that makes this global address space work over networks with different physical addressing schemes). On the issue of scale, IP uses hierarchical aggregation to reduce the amount of information needed to forward packets. Specifically, IP addresses are partitioned into network and host components, with packets first routed toward the destination network and then delivered to the correct host on that network.

3.2.7 Host Configuration (DHCP)

In Section 2.6, 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, not only must be unique on a given internetwork but also must reflect the structure of the internetwork. As noted above, they contain a network part and a host part, and the network part must be the same for all hosts on the same network. Thus, it is not possible for the IP address to be configured once into a host when it is manufactured, since that would imply that the manufacturer knew which hosts were going to end up on which networks, and it would mean that a host, once connected to one network, could never move to another. For this reason, IP addresses need to be reconfigurable.

In addition to an IP address, there are some other pieces of information a host needs to have before it can start sending packets. The most notable of these is the address of a default router—the place to which it can send packets whose destination address is not on the same network as the sending host.

Most host operating systems provide a way for a system administrator, or even a user, to manually configure the IP information needed by

a host; however, there are some obvious drawbacks to such manual configuration. One is that it is simply a lot of work to configure all the hosts in a large network directly, especially when you consider that such hosts are not reachable over a network until they are configured. Even more importantly, the configuration process is very error prone, since it is necessary to ensure that every host gets the correct network number and that no two hosts receive the same IP address. For these reasons, automated configuration methods are required. The primary method uses a protocol known as the *Dynamic Host Configuration Protocol* (DHCP).

DHCP relies on the existence of a DHCP server that is responsible for providing configuration information to hosts. There is at least one DHCP server for an administrative domain. At the simplest level, the DHCP server can function just as a centralized repository for host configuration information. Consider, for example, the problem of administering addresses in the internetwork of a large company. DHCP saves the network administrators from having to walk around to every host in the company with a list of addresses and network map in hand and configuring each host manually. Instead, the configuration information for each host could be stored in the DHCP server and automatically retrieved by each host when it is booted or connected to the network. However, the administrator would still pick the address that each host is to receive; he would just store that in the server. In this model, the configuration information for each host is stored in a table that is indexed by some form of unique client identifier, typically the hardware address (e.g., the Ethernet address of its network adaptor).

A more sophisticated use of DHCP saves the network administrator from even having to assign addresses to individual hosts. In this model, the DHCP server maintains a pool of available addresses that it hands out to hosts on demand. This considerably reduces the amount of configuration an administrator must do, since now it is only necessary to allocate a range of IP addresses (all with the same network number) to each network.

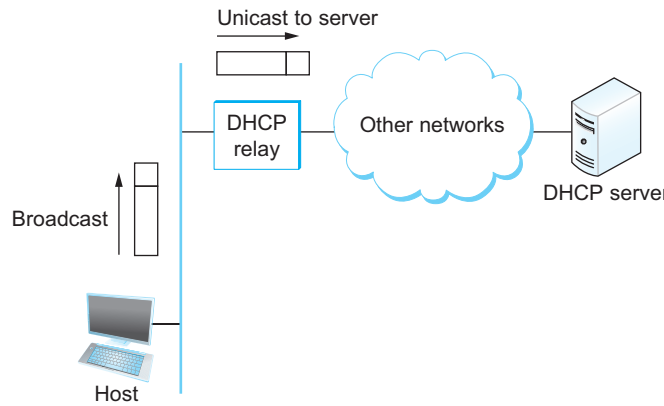
Since the goal of DHCP is to minimize the amount of manual configuration required for a host to function, it would rather defeat the purpose if each host had to be configured with the address of a DHCP server. Thus, the first problem faced by DHCP is that of server discovery.

To contact a DHCP server, a newly booted or attached host sends a DHCPDISCOVER message to a special IP address (255.255.255.255) that

is an IP broadcast address. This means it will be received by all hosts and routers on that network. (Routers do not forward such packets onto other networks, preventing broadcast to the entire Internet.) In the simplest case, one of these nodes is the DHCP server for the network. The server would then reply to the host that generated the discovery message (all the other nodes would ignore it). However, it is not really desirable to require one DHCP server on every network, because this still creates a potentially large number of servers that need to be correctly and consistently configured. Thus, DHCP uses the concept of a *relay agent*. There is at least one relay agent on each network, and it is configured with just one piece of information: the IP address of the DHCP server. When a relay agent receives a DHCPDISCOVER message, it unicasts it to the DHCP server and awaits the response, which it will then send back to the requesting client. The process of relaying a message from a host to a remote DHCP server is shown in Figure 3.24.

Figure 3.25 shows the format of a DHCP message. The message is actually sent using a protocol called the *User Datagram Protocol* (UDP) that runs over IP. UDP is discussed in detail in the next chapter, but the only interesting thing it does in this context is to provide a demultiplexing key that says, "This is a 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



■ **FIGURE 3.24** A DHCP relay agent receives a broadcast DHCPDISCOVER message from a host and sends a unicast DHCPDISCOVER to the DHCP server.

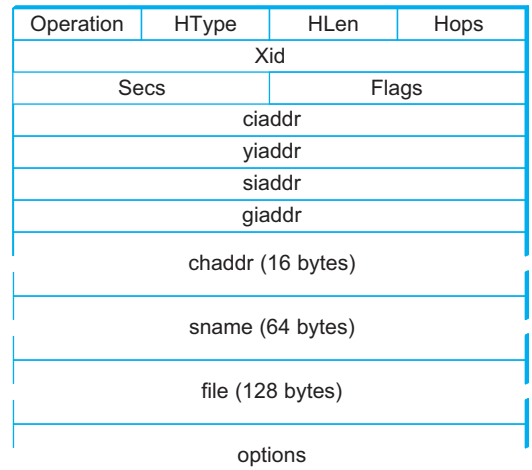


FIGURE 3.25 DHCP packet format.

trying to obtain configuration information, the client puts its hardware address (e.g., its Ethernet address) in the chaddr field. The DHCP server replies by filling in the yiaddr ("your" IP address) field and sending it to the client. Other information such as the default router to be used by this client can be included in the options field.

In the case where DHCP dynamically assigns IP addresses to hosts, it is clear that hosts cannot keep addresses indefinitely, as this would eventually cause the server to exhaust its address pool. At the same time, a host cannot be depended upon to give back its address, since it might have crashed, been unplugged from the network, or been turned off. Thus, DHCP allows addresses to be leased for some period of time. Once the lease expires, the server is free to return that address to its pool. A host with a leased address clearly needs to renew the lease periodically if in fact it is still connected to the network and functioning correctly.

DHCP illustrates an important aspect of scaling: the scaling of network management. While discussions of scaling often focus on keeping the state in network devices from growing too fast, it is important to pay attention to growth of network management complexity. By allowing network managers to configure a range of IP addresses per network rather than one IP address per host, DHCP improves the manageability of a network.

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.

3.2.8 Error Reporting (ICMP)

The next issue is how the Internet treats errors. While IP is perfectly willing to drop datagrams when the going gets tough—for example, when a router does not know how to forward the datagram or when one fragment of a datagram fails to arrive at the destination—it does not necessarily fail silently. IP is always configured with a companion protocol, known as the *Internet Control Message Protocol* (ICMP), that defines a collection of error messages that are sent back to the source host whenever a router or host is unable to process an IP datagram successfully. For example, ICMP defines error messages indicating that the destination host is unreachable (perhaps due to a link failure), that the reassembly process failed, that the TTL had reached 0, that the IP header checksum failed, and so on.

ICMP also defines a handful of control messages that a router can send back to a source host. One of the most useful control messages, called an *ICMP-Redirect*, tells the source host that there is a better route to the destination. ICMP-Redirects are used in the following situation. Suppose a host is connected to a network that has two routers attached to it, called *R1* and *R2*, where the host uses R1 as its default router. Should R1 ever receive a datagram from the host, where based on its forwarding table it knows that R2 would have been a better choice for a particular destination address, it sends an ICMP-Redirect back to the host, instructing it to use R2 for all future datagrams addressed to that destination. The host then adds this new route to its forwarding table.

ICMP also provides the basis for two widely used debugging tools, ping and traceroute. ping uses ICMP echo messages to determine if a node is reachable and alive. traceroute uses a slightly non-intuitive technique to determine the set of routers along the path to a destination, which is the topic for one of the exercises at the end of this chapter.

3.2.9 Virtual Networks and Tunnels

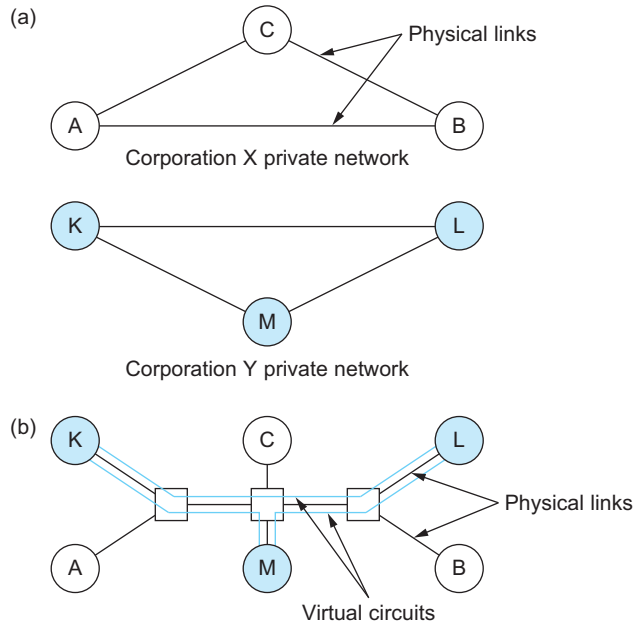
We conclude our introduction to IP by considering an issue you might not have anticipated, but one that is becoming increasingly important.

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 the usually the goal in the Internet—everybody wants to be able to send email to everybody, and the creator of a new website wants to reach the widest possible audience. However, there are many situations where more controlled connectivity is required. An important example of such a situation is the *virtual private network* (VPN).

The term *VPN* is heavily overused and definitions vary, but intuitively we can define a VPN by considering first the idea of a private network. Corporations with many sites often build private networks by leasing transmission lines from the phone companies and using those lines to interconnect sites. In such a network, communication is restricted to take place only among the sites of that corporation, which is often desirable for security reasons. To make a private network *virtual*, the leased transmission lines—which are not shared with any other corporations—would be replaced by some sort of shared network. A virtual circuit (VC) is a very reasonable replacement for a leased line because it still provides a logical point-to-point connection between the corporation's sites. For example, if corporation X has a VC from site A to site B, then clearly it can send packets between sites A and B. But there is no way that corporation Y can get its packets delivered to site B without first establishing its own virtual circuit to site B, and the establishment of such a VC can be administratively prevented, thus preventing unwanted connectivity between corporation X and corporation Y.

Figure 3.26(a) shows two private networks for two separate corporations. In Figure 3.26(b) they are both migrated to a virtual circuit network. The limited connectivity of a real private network is maintained, but since the private networks now share the same transmission facilities and switches we say that two virtual private networks have been created.

In Figure 3.26, a virtual circuit network (using Frame Relay or ATM, for example) is used to provide the controlled connectivity among sites. It is also possible to provide a similar function using an IP network—an internetwork—to provide the connectivity. However, we cannot just connect the various corporations' sites to a single internetwork because that would provide connectivity between corporation X and corporation Y, which we wish to avoid. To solve this problem, we need to introduce a new concept, the *IP tunnel*.

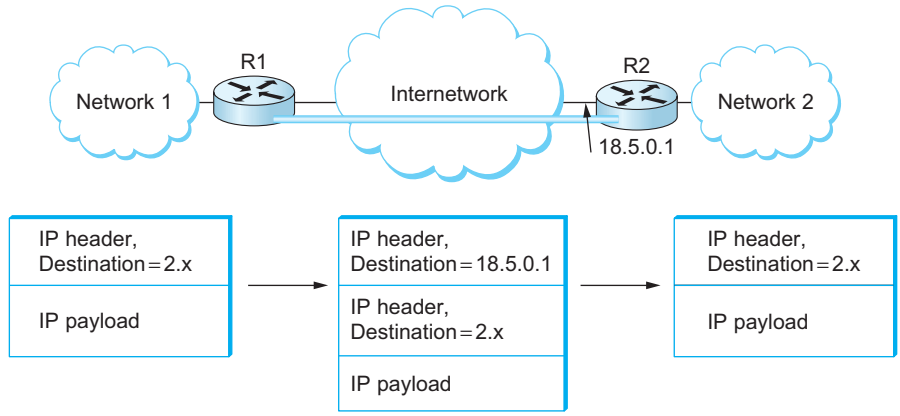


■ **FIGURE 3.26** An example of virtual private networks: (a) two separate private networks; (b) two virtual private networks sharing common switches.

We can think of an IP tunnel as a virtual point-to-point link between a pair of nodes that are actually separated by an arbitrary number of networks. The virtual link is created within the router at the entrance to the tunnel by providing it with the IP address of the router at the far end of the tunnel. Whenever the router at the entrance of the tunnel wants to send a packet over this virtual link, it encapsulates the packet inside an IP datagram. The destination address in the IP header is the address of the router at the far end of the tunnel, while the source address is that of the encapsulating router.

In the forwarding table of the router at the entrance to the tunnel, this virtual link looks much like a normal link. Consider, for example, the network in Figure 3.27. A tunnel has been configured from R1 to R2 and assigned a virtual interface number of 0. The forwarding table in R1 might therefore look like Table 3.8.

R1 has two physical interfaces. Interface 0 connects to network 1; interface 1 connects to a large internetwork and is thus the default for all traffic



■ **FIGURE 3.27** A tunnel through an internetwork. 18.5.0.1 is the address of R2 that can be reached from R1 across the internetwork.

Table 3.8 Forwarding Table for Router R1 in Figure 3.27

NetworkNum	NextHop
1	Interface 0
2	Virtual interface 0
Default	Interface 1

that does not match something more specific in the forwarding table. In addition, R1 has a virtual interface, which is the interface to the tunnel. Suppose R1 receives a packet from network 1 that contains an address in network 2. The forwarding table says this packet should be sent out virtual interface 0. In order to send a packet out this interface, the router takes the packet, adds an IP header addressed to R2, and then proceeds to forward the packet as if it had just been received. R2's address is 18.5.0.1; since the network number of this address is 18, not 1 or 2, a packet destined for R2 will be forwarded out the default interface into the internetwork.

Once the packet leaves R1, it looks to the rest of the world like a normal IP packet destined to R2, and it is forwarded accordingly. All the routers in the internetwork forward it using normal means, until it arrives at R2. When R2 receives the packet, it finds that it carries its own address, so it

removes the IP header and looks at the payload of the packet. What it finds is an inner IP packet whose destination address is in network 2. R2 now processes this packet like any other IP packet it receives. Since R2 is directly connected to network 2, it forwards the packet on to that network. Figure 3.27 shows the change in encapsulation of the packet as it moves across the network.

While R2 is acting as the endpoint of the tunnel, there is nothing to prevent it from performing the normal functions of a router. For example, it might receive some packets that are not tunneled, but that are addressed to networks that it knows how to reach, and it would forward them in the normal way.

You might wonder why anyone would want to go to all the trouble of creating a tunnel and changing the encapsulation of a packet as it goes across an internetwork. One reason is security, which we will discuss in more detail in Chapter 8. Supplemented with encryption, a tunnel can become a very private sort of link across a public network. Another reason may be that R1 and R2 have some capabilities that are not widely available in the intervening networks, such as multicast routing. By connecting these routers with a tunnel, we can build a virtual network in which all the routers with this capability appear to be directly connected. This in fact is how the MBone (multicast backbone) is built, as we will see in Section 4.2. A third reason to build tunnels is to carry packets from protocols other than IP across an IP network. As long as the routers at either end of the tunnel know how to handle these other protocols, the IP tunnel looks to them like a point-to-point link over which they can send non-IP packets. Tunnels also provide a mechanism by which we can force a packet to be delivered to a particular place even if its original header—the one that gets encapsulated inside the tunnel header—might suggest that it should go somewhere else. We will see an application of this when we consider mobile hosts in Section 4.4.2. Thus, we see that tunneling is a powerful and quite general technique for building virtual links across internetworks.

Tunneling does have its downsides. One is that it increases the length of packets; this might represent a significant waste of bandwidth for short packets. Longer packets might be subject to fragmentation, which has its own set of drawbacks. There may also be performance implications for the routers at either end of the tunnel, since they need to do more work than normal forwarding as they add and remove the tunnel header.

Finally, there is a management cost for the administrative entity that is responsible for setting up the tunnels and making sure they are correctly handled by the routing protocols.

3.3 ROUTING

So far in this chapter we have assumed that the switches and routers have enough knowledge of the network topology so they can choose the right port onto which each packet should be output. In the case of virtual circuits, routing is an issue only for the connection request packet; all subsequent packets follow the same path as the request. In datagram networks, including IP networks, routing is an issue for every packet. In either case, a switch or router needs to be able to look at a destination address and then to determine which of the output ports is the best choice to get a packet to that address. As we saw in Section 3.1.1, the switch makes this decision by consulting a forwarding table. The fundamental problem of routing is how switches and routers acquire the information in their forwarding tables.

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 on complex 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 prefix to an outgoing interface and some MAC information, such as the Ethernet address of the next hop. The routing table, on the other hand, is the table that is built up by the routing algorithms as a precursor to building the forwarding table. It generally contains mappings from network prefixes to next hops. It may also contain information about how this

Table 3.9 Example Rows from (a) Routing and (b) Forwarding Tables

(a)

Prefix/Length	Next Hop
18/8	171.69.245.10

(b)

Prefix/Length	Interface	MAC Address
18/8	if0	8:0:2b:e4:b:1:2

information was learned, so that the router will be able to decide when it should discard some information.

Whether the routing table and forwarding table are actually separate data structures is something of an implementation choice, but there are numerous reasons to keep them separate. For example, the forwarding table needs to be structured to optimize the process of looking up an address when forwarding a packet, while the routing table needs to be optimized for the purpose of calculating changes in topology. In many cases, the forwarding table may even be implemented in specialized hardware, whereas this is rarely if ever done for the routing table. Table 3.9 provides an example of a row from each sort of table. In this case, the routing table tells us that network prefix 18/8 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 "not so much." They are designed for networks of fairly modest size—up to a few hundred nodes, in practice. However, the solutions we describe do serve as a building block for a hierarchical routing infrastructure that is used in the Internet today. Specifically, the protocols described in this section are

collectively known as *intradomain* routing protocols, or *interior gateway protocols* (IGPs). To understand these terms, we need to define a routing *domain*. A good working definition is an internetwork in which all the routers are under the same administrative control (e.g., a single university campus, or the network of a single Internet Service Provider). The relevance of this definition will become apparent in the next chapter when we look at *interdomain* routing protocols. For now, the important thing to keep in mind is that we are considering the problem of routing in the context of small to midsized networks, not for a network the size of the Internet.

3.3.1 Network as a Graph

Routing is, in essence, a problem of graph theory. Figure 3.28 shows a graph representing a network. The nodes of the graph, labeled A through F, may be hosts, switches, routers, or networks. For our initial discussion, we will focus on the case where the nodes are routers. The edges of the graph correspond to the network links. Each edge has an associated *cost*, which gives some indication of the desirability of sending traffic over that link. A discussion of how edge costs are assigned is given in Section 3.3.4.[11]

The basic problem of routing is to find the lowest-cost path between any two nodes, where the cost of a path equals the sum of the costs of all the edges that make up the path. For a simple network like the one in Figure 3.28, you could imagine just calculating all the shortest paths and

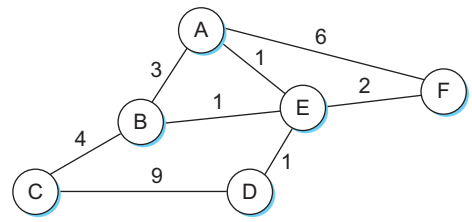


FIGURE 3.28 Network represented as a graph.

[11] In the example networks (graphs) used throughout this chapter, we use undirected edges and assign each edge a single cost. This is actually a slight simplification. It is more accurate to make the edges directed, which typically means that there would be a pair of edges between each node—one flowing in each direction, and each with its own edge cost.

loading them into some nonvolatile storage on each node. Such a static approach has several shortcomings:

- It does not deal with node or link failures.
- It does not consider the addition of new nodes or links.
- It implies that edge costs cannot change, even though we might reasonably wish to have link costs change over time (e.g., assigning high cost to a link that is heavily loaded).

For these reasons, routing is achieved in most practical networks by running routing protocols among the nodes. These protocols provide a distributed, dynamic way to solve the problem of finding the lowest-cost path in the presence of link and node failures and changing edge costs. Note the word *distributed* in the previous sentence; it is difficult to make centralized solutions scalable, so all the widely used routing protocols use distributed algorithms.[12]

The distributed nature of routing algorithms is one of the main reasons why this has been such a rich field of research and development—there are a lot of challenges in making distributed algorithms work well. For example, distributed algorithms raise the possibility that two routers will at one instant have different ideas about the shortest path to some destination. In fact, each one may think that the other one is closer to the destination and decide to send packets to the other one. Clearly, such packets will be stuck in a loop until the discrepancy between the two routers is resolved, and it would be good to resolve it as soon as possible. This is just one example of the type of problem routing protocols must address.

To begin our analysis, we assume that the edge costs in the network are known. We will examine the two main classes of routing protocols: *distance vector* and *link state*. In Section 3.3.4, we return to the problem of calculating edge costs in a meaningful way.

3.3.2 Distance-Vector (RIP)

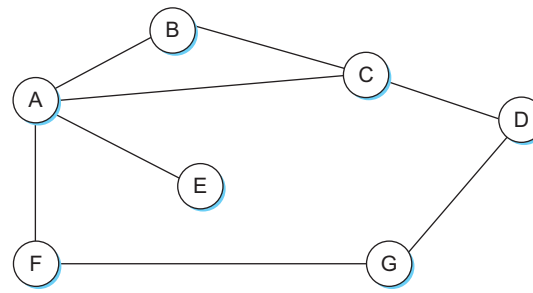
The idea behind the distance-vector algorithm is suggested by its name.[13] Each node constructs a one-dimensional array (a vector) containing the "distances" (costs) to all other nodes and distributes that vector to its

[12]This widely held assumption, however, has been re-examined in recent years—see the Further Reading section.

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

immediate neighbors. The starting assumption for distance-vector routing is that each node knows the cost of the link to each of its directly connected neighbors. These costs may be provided when the router is configured by a network manager. A link that is down is assigned an infinite cost.

To see how a distance-vector routing algorithm works, it is easiest to consider an example like the one depicted in Figure 3.29. In this example, the cost of each link is set to 1, so that a least-cost path is simply the one with the fewest hops. (Since all edges have the same cost, we do not show the costs in the graph.) We can represent each node's knowledge about the distances to all other nodes as a table like Table 3.10. Note that each



■ FIGURE 3.29 Distance-vector routing: an example network.

Table 3.10 Initial Distances Stored at Each Node (Global View)

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

node knows only the information in one row of the table (the one that bears its name in the left column). The global view that is presented here is not available at any single point in the network.

We may consider each row in Table 3.10 as a list of distances from one node to all other nodes, representing the current beliefs of that node. Initially, each node sets a cost of 1 to its directly connected neighbors and ∞ to all other nodes. Thus, A initially believes that it can reach B in one hop and that D is unreachable. The routing table stored at A reflects this set of beliefs and includes the name of the next hop that A would use to reach any reachable node. Initially, then, A's routing table would look like Table 3.11.

The next step in distance-vector routing is that every node sends a message to its directly connected neighbors containing its personal list of distances. For example, node F tells node A that it can reach node G at a cost of 1; A also knows it can reach F at a cost of 1, so it adds these costs to get the cost of reaching G by means of F. This total cost of 2 is less than the current cost of infinity, so A records that it can reach G at a cost of 2 by going through F. Similarly, A learns from C that D can be reached from C at a cost of 1; it adds this to the cost of reaching C (1) and decides that D can be reached via C at a cost of 2, which is better than the old cost of infinity. At the same time, A learns from C that B can be reached from C at a cost of 1, so it concludes that the cost of reaching B via C is 2. Since this is worse than the current cost of reaching B (1), this new information is ignored.

Table 3.11 Initial Routing Table at Node A

Destination	Cost	NextHop
B	1	B
C	1	C
D	∞	—
E	1	E
F	1	F
G	∞	—

At this point, A can update its routing table with costs and next hops for all nodes in the network. The result is shown in Table 3.12.

In the absence of any topology changes, it takes only a few exchanges of information between neighbors before each node has a complete routing table. The process of getting consistent routing information to all the nodes is called *convergence*. Table 3.13 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

Table 3.12 Final Routing Table at Node A

Destination	Cost	NextHop
B	1	B
C	1	C
D	2	C
E	1	E
F	1	F
G	2	F

Table 3.13 Final Distances Stored at Each Node (Global View)

Information Stored at Node	Distance to Reach Node						
	A	B	C	D	E	F	G
A	0	1	1	2	1	1	2
B	1	0	1	2	2	2	3
C	1	1	0	1	2	2	2
D	2	2	1	0	3	2	1
E	1	2	2	3	0	2	3
F	1	2	2	2	2	0	1
G	2	3	2	1	3	1	0

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 several minutes. The second mechanism, sometimes called a *triggered* update, happens whenever a node notices a link failure or receives an update from one of its neighbors that causes it to change one of the routes in its routing table. Whenever a node's routing table changes, it sends an update to its neighbors, which may lead to a change in their tables, causing them to send an update to their neighbors.

Now consider what happens when a link or node fails. The nodes that notice first send new lists of distances to their neighbors, and normally the system settles down fairly quickly to a new state. As to the question of how a node detects a failure, there are a couple of different answers. In one approach, a node continually tests the link to another node by sending a control packet and seeing if it receives an acknowledgment. In another approach, a node determines that the link (or the node at the other end of the link) is down if it does not receive the expected periodic routing update for the last few update cycles.

To understand what happens when a node detects a link failure, consider what happens when F detects that its link to G has failed. First, F sets its new distance to G to infinity and passes that information along to A. Since A knows that its 2-hop path to G is through F, A would also set its distance to G to infinity. However, with the next update from C, A would learn that C has a 2-hop path to G. Thus, A would know that it could reach G in 3 hops through C, which is less than infinity, and so A would update its table accordingly. When it advertises this to F, node F would learn that it can reach G at a cost of 4 through A, which is less than infinity, and the system would again become stable.

Unfortunately, slightly different circumstances can prevent the network from stabilizing. Suppose, for example, that the link from A to E goes down. In the next round of updates, A advertises a distance of infinity to E, but B and C advertise a distance of 2 to E. Depending on the exact timing of events, the following might happen: Node B, upon hearing that E can be reached in 2 hops from C, concludes that it can reach E in 3 hops and advertises this to A; node A concludes that it can reach E in 4 hops and advertises this to C; node C concludes that it can reach E in 5 hops; and so on. This cycle stops only when the distances reach some number that is large enough to be considered infinite. In the meantime, none of the nodes actually knows that E is unreachable, and the routing tables for the network do not stabilize. This situation is known as the *count to infinity* problem.

There are several partial solutions to this problem. The first one is to use some relatively small number as an approximation of infinity. For example, we might decide that the maximum number of hops to get across a certain network is never going to be more than 16, and so we could pick 16 as the value that represents infinity. This at least bounds the amount of time that it takes to count to infinity. Of course, it could also present a problem if our network grew to a point where some nodes were separated by more than 16 hops.

One technique to improve the time to stabilize routing is called *split horizon*. The idea is that when a node sends a routing update to its neighbors, it does not send those routes it learned from each neighbor back to that neighbor. For example, if B has the route (E, 2, A) in its table, then it knows it must have learned this route from A, and so whenever B sends a routing update to A, it does not include the route (E, 2) in that update. In a stronger variation of split horizon, called *split horizon with poison reverse*, B actually sends that route back to A, but it puts negative information in the route to ensure that A will not eventually use B to get to E. For example, B sends the route (E, ∞) to A. The problem with both of these techniques is that they only work for routing loops that involve two nodes. For larger routing loops, more drastic measures are called for. Continuing the above example, if B and C had waited for a while after hearing of the link failure from A before advertising routes to E, they would have found that neither of them really had a route to E. Unfortunately, this approach delays the convergence of the protocol; speed of convergence is one of the key advantages of its competitor, link-state routing, the subject of Section 3.3.3.

Implementation

The code that implements this algorithm is very straightforward; we give only some of the basics here. Structure Route defines each entry in the routing table, and constant MAX_TTL specifies how long an entry is kept in the table before it is discarded.

```
#define MAX_ROUTES      128     /* maximum size of routing table */
#define MAX_TTL         120     /* time (in seconds) until route expires */

typedef struct {
    NodeAddr    Destination;    /* address of destination */
    NodeAddr    NextHop;        /* address of next hop */
    int         Cost;           /* distance metric */
    u_short     TTL;            /* time to live */
} Route;

int      numRoutes = 0;
Route    routingTable[MAX_ROUTES];
```

The routine that updates the local node's routing table based on a new route is given by mergeRoute. Although not shown, a timer function periodically scans the list of routes in the node's routing table, decrements the TTL (time to live) field of each route, and discards any routes that have a time to live of 0. Notice, however, that the TTL field is reset to MAX_TTL any time the route is reconfirmed by an update message from a neighboring node.

```
void
mergeRoute (Route *new)
{
    int i;

    for (i = 0; i < numRoutes; ++i)
    {
        if (new->Destination == routingTable[i].Destination)
        {
            if (new->Cost + 1 < routingTable[i].Cost)
            {
                /* found a better route: */
                break;
            } else if (new->NextHop == routingTable[i].NextHop) {
```

```
                    /* metric for current next-hop may have
                       changed: */
                    break;
                } else {
                    /* route is uninteresting---just ignore
                      it */
                    return;
                }
            }
        }
        if (i == numRoutes)
        {
            /* this is a completely new route; is there room
               for it? */
            if (numRoutes < MAXROUTES)
            {
                ++numRoutes;
            } else {
                /* can't fit this route in table so give up */
                return;
            }
        }
        routingTable[i] = *new;
        /* reset TTL */
        routingTable[i].TTL = MAX_TTL;
        /* account for hop to get to next node */
        ++routingTable[i].Cost;

}
```

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

```
void
updateRoutingTable (Route *newRoute, int numNewRoutes)
{
    int i;

    for (i=0; i < numNewRoutes; ++i)
    {
```

```
            mergeRoute(&newRoute[i]);
        }
    }
```

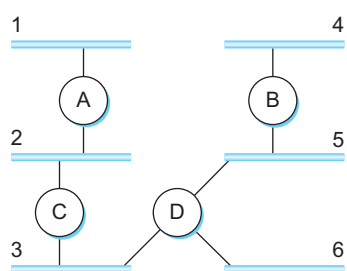
Routing Information Protocol (RIP)

One of the more widely used routing protocols in IP networks is the Routing Information Protocol (RIP). Its widespread use in the early days of IP was due in no small part to the fact that it was distributed along with the popular Berkeley Software Distribution (BSD) version of Unix, from which many commercial versions of Unix were derived. It is also extremely simple. RIP is the canonical example of a routing protocol built on the distance-vector algorithm just described.

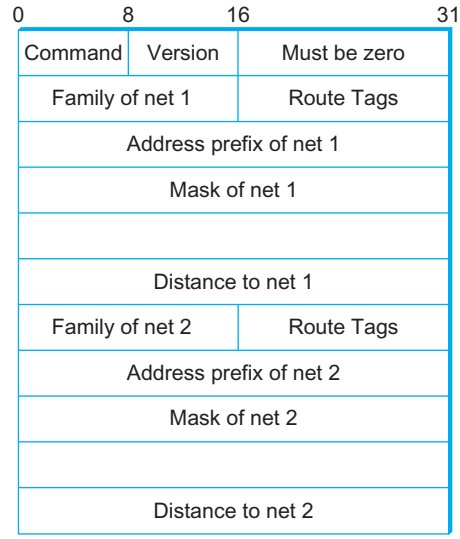
Routing protocols in internetworks differ very slightly from the idealized graph model described above. In an internetwork, the goal of the routers is to learn how to forward packets to various *networks*. Thus, rather than advertising the cost of reaching other routers, the routers advertise the cost of reaching networks. For example, in Figure 3.30, router C would advertise to router A the fact that it can reach networks 2 and 3 (to which it is directly connected) at a cost of 0, networks 5 and 6 at cost 1, and network 4 at cost 2.

We can see evidence of this in the RIP (version 2) packet format in Figure 3.31. The majority of the packet is taken up with ⟨address, mask, distance⟩ triples. However, the principles of the routing algorithm are just the same. For example, if router A learns from router B that network X can be reached at a lower cost via B than via the existing next hop in the routing table, A updates the cost and next hop information for the network number accordingly.

RIP is in fact a fairly straightforward implementation of distance-vector routing. Routers running RIP send their advertisements every 30 seconds;



■ **FIGURE 3.30** Example network running RIP.



■ **FIGURE 3.31** RIPv2 packet format.

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—that is the reason for the Family part of the advertisements. RIP version 2 (RIPv2) also introduced the subnet masks described in Section 3.2.5, whereas RIP version 1 worked with the old classful addresses of IP.

As we will see below, it is possible to use a range of different metrics or costs for the links in a routing protocol. RIP takes the simplest approach, with all link costs being equal to 1, just as in our example above. Thus, it always tries to find the minimum hop route. Valid distances are 1 through 15, with 16 representing infinity. This also limits RIP to running on fairly small networks—those with no paths longer than 15 hops.

LAB 07: OSPF

3.3.3 Link State (OSPF)

Link-state routing is the second major class of intradomain routing protocol. The starting assumptions for link-state routing are rather similar to those for distance-vector routing. Each node is assumed to be capable of finding out the state of the link to its neighbors (up or down) and the cost of each link. Again, we want to provide each node with enough information to enable it to find the least-cost path to any destination. The basic

idea behind link-state protocols is very simple: Every node knows how to reach its directly connected neighbors, and if we make sure that the totality of this knowledge is disseminated to every node, then every node will have enough knowledge of the network to build a complete map of the network. This is clearly a sufficient condition (although not a necessary one) for finding the shortest path to any point in the network. Thus, link-state routing protocols rely on two mechanisms: reliable dissemination of link-state information, and the calculation of routes from the sum of all the accumulated link-state knowledge.

Reliable Flooding

Reliable flooding is the process of making sure that all the nodes participating in the routing protocol get a copy of the link-state information from all the other nodes. As the term *flooding* suggests, the basic idea is for a node to send its link-state information out on all of its directly connected links; each node that receives this information then forwards it out on all of *its* links. This process continues until the information has reached all the nodes in the network.

More precisely, each node creates an update packet, also called a *link-state packet* (LSP), which contains the following information:

- The ID of the node that created the LSP
- A list of directly connected neighbors of that node, with the cost of the link to each one
- A sequence number
- A time to live for this packet

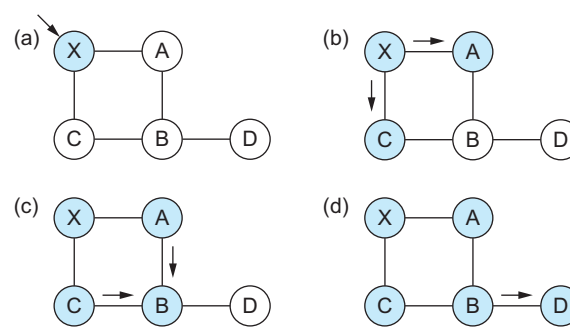
The first two items are needed to enable route calculation; the last two are used to make the process of flooding the packet to all nodes reliable. Reliability includes making sure that you have the most recent copy of the information, since there may be multiple, contradictory LSPs from one node traversing the network. Making the flooding reliable has proven to be quite difficult. (For example, an early version of link-state routing used in the ARPANET caused that network to fail in 1981.)

Flooding works in the following way. First, the transmission of LSPs between adjacent routers is made reliable using acknowledgments and retransmissions just as in the reliable link-layer protocol described in Section 2.5. However, several more steps are necessary to reliably flood an LSP to all nodes in a network.

Consider a node X that receives a copy of an LSP that originated at some other node Y. Note that Y may be any other router in the same routing domain as X. X checks to see if it has already stored a copy of an LSP from Y. If not, it stores the LSP. If it already has a copy, it compares the sequence numbers; if the new LSP has a larger sequence number, it is assumed to be the more recent, and that LSP is stored, replacing the old one. A smaller (or equal) sequence number would imply an LSP older (or not newer) than the one stored, so it would be discarded and no further action would be needed. If the received LSP was the newer one, X then sends a copy of that LSP to all of its neighbors except the neighbor from which the LSP was just received. The fact that the LSP is not sent back to the node from which it was received helps to bring an end to the flooding of an LSP. Since X passes the LSP on to all its neighbors, who then turn around and do the same thing, the most recent copy of the LSP eventually reaches all nodes.

Figure 3.32 shows an LSP being flooded in a small network. Each node becomes shaded as it stores the new LSP. In Figure 3.32(a) the LSP arrives at node X, which sends it to neighbors A and C in Figure 3.32(b). A and C do not send it back to X, but send it on to B. Since B receives two identical copies of the LSP, it will accept whichever arrived first and ignore the second as a duplicate. It then passes the LSP onto D, which has no neighbors to flood it to, and the process is complete.

Just as in RIP, each node generates LSPs under two circumstances. Either the expiry of a periodic timer or a change in topology can cause a node to generate a new LSP. However, the only topology-based reason for a node to generate an LSP is if one of its directly connected links or



■ **FIGURE 3.32** Flooding of link-state packets: (a) LSP arrives at node X; (b) X floods LSP to A and C; (c) A and C flood LSP to B (but not X); (d) flooding is complete.

immediate neighbors has gone down. The failure of a link can be detected in some cases by the link-layer protocol. The demise of a neighbor or loss of connectivity to that neighbor can be detected using periodic "hello" packets. Each node sends these to its immediate neighbors at defined intervals. If a sufficiently long time passes without receipt of a "hello" from a neighbor, the link to that neighbor will be declared down, and a new LSP will be generated to reflect this fact.

One of the important design goals of a link-state protocol's flooding mechanism is that the newest information must be flooded to all nodes as quickly as possible, while old information must be removed from the network and not allowed to circulate. In addition, it is clearly desirable to minimize the total amount of routing traffic that is sent around the network; after all, this is just overhead from the perspective of those who actually use the network for their applications. The next few paragraphs describe some of the ways that these goals are accomplished.

One easy way to reduce overhead is to avoid generating LSPs unless absolutely necessary. This can be done by using very long timers—often on the order of hours—for the periodic generation of LSPs. Given that the flooding protocol is truly reliable when topology changes, it is safe to assume that messages saying "nothing has changed" do not need to be sent very often.

To make sure that old information is replaced by newer information, LSPs carry sequence numbers. Each time a node generates a new LSP, it increments the sequence number by 1. Unlike most sequence numbers used in protocols, these sequence numbers are not expected to wrap, so the field needs to be quite large (say, 64 bits). If a node goes down and then comes back up, it starts with a sequence number of 0. If the node was down for a long time, all the old LSPs for that node will have timed out (as described below); otherwise, this node will eventually receive a copy of its own LSP with a higher sequence number, which it can then increment and use as its own sequence number. This will ensure that its new LSP replaces any of its old LSPs left over from before the node went down.

LSPs also carry a time to live. This is used to ensure that old link-state information is eventually removed from the network. A node always decrements the TTL of a newly received LSP before flooding it to its neighbors. It also "ages" the LSP while it is stored in the node. When the TTL reaches 0, the node refloods the LSP with a TTL of 0, which is interpreted by all the nodes in the network as a signal to delete that LSP.

Route Calculation

Once a given node has a copy of the LSP from every other node, it is able to compute a complete map for the topology of the network, and from this map it is able to decide the best route to each destination. The question, then, is exactly how it calculates routes from this information. The solution is based on a well-known algorithm from graph theory—Dijkstra's shortest-path algorithm.

We first define Dijkstra's algorithm in graph-theoretic terms. Imagine that a node takes all the LSPs it has received and constructs a graphical representation of the network, in which N denotes the set of nodes in the graph, $l(i,j)$ denotes the nonnegative cost (weight) associated with the edge between nodes $i, j \in N$ and $l(i,j) = \infty$ 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 nodes incorporated so far by the algorithm, and $C(n)$ denotes the cost of the path from s to each node n. Given these definitions, the algorithm is defined as follows:

$M = \{s\}$
for each n in $N - \{s\}$
  $C(n) = l(s,n)$
while $(N \neq M)$
  $M = M \cup \{w\}$ such that $C(w)$ is the minimum for all w in $(N - M)$
  for each n in $(N - M)$
    $C(n) = \text{MIN}(C(n), C(w) + l(w,n))$

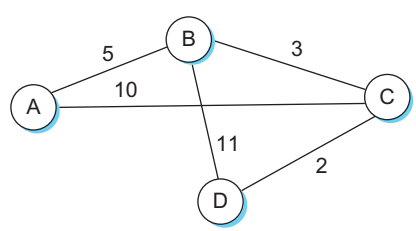
Basically, the algorithm works as follows. We start with M containing this node s and then initialize the table of costs (the $C(n)$s) to other nodes using the known costs to directly connected nodes. We then look for the node that is reachable at the lowest cost (w) and add it to M. Finally, we update the table of costs by considering the cost of reaching nodes through w. In the last line of the algorithm, we choose a new route to node n that goes through node w if the total cost of going from the source to w and then following the link from w to n is less than the old route we had to n. This procedure is repeated until all nodes are incorporated in M.

In practice, each switch computes its routing table directly from the LSPs it has collected using a realization of Dijkstra's algorithm called the *forward search* algorithm. Specifically, each switch maintains two lists,

known as Tentative and Confirmed. Each of these lists contains a set of entries of the form (Destination, Cost, NextHop). The algorithm works as follows:

1. Initialize the Confirmed list with an entry for myself; this entry has a cost of 0.
2. For the node just added to the Confirmed list in the previous step, call it node Next and select its LSP.
3. For each neighbor (Neighbor) of Next, calculate the cost (Cost) to reach this Neighbor as the sum of the cost from myself to Next and from Next to Neighbor.
   (a) If Neighbor is currently on neither the Confirmed nor the Tentative list, then add (Neighbor, Cost, NextHop) to the Tentative list, where NextHop is the direction I go to reach Next.
   (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 3.33. Note that, unlike our previous example, this network has a range of different edge costs. Table 3.14 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



■ FIGURE 3.33 Link-state routing: an example network.

Table 3.14 Steps for Building Routing Table for Node D (Figure 3.33)

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.

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.

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.

The difference between the distance-vector and link-state algorithms can be summarized as follows. In distance-vector, each node talks only to its directly connected neighbors, but it tells them everything it has learned (i.e., distance to all nodes). In link-state, each node talks to all other nodes, but it tells them only what it knows for sure (i.e., only the state of its directly connected links).

The Open Shortest Path First Protocol (OSPF)

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

- *Authentication of routing messages*—One feature of distributed routing algorithms is that they disperse information from one node to many other nodes, and the entire network can thus be impacted by bad information from one node. For this reason, it's a good idea to be sure that all the nodes taking part in the protocol can be trusted. Authenticating routing messages helps achieve this. Early versions of OSPF used a simple 8-byte password for authentication. This is not a strong enough form of authentication to prevent dedicated malicious users, but it alleviates some problems caused by misconfiguration or casual attacks. (A similar form of authentication was added to RIP in version 2.) Strong cryptographic authentication 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. We examine areas in detail in Section 4.1.1.
- *Load balancing*—OSPF allows multiple routes to the same place to be assigned the same cost and will cause traffic to be distributed evenly over those routes, thus making better use of available network capacity.

There are several different types of OSPF messages, but all begin with the same header, as shown in Figure 3.34. 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

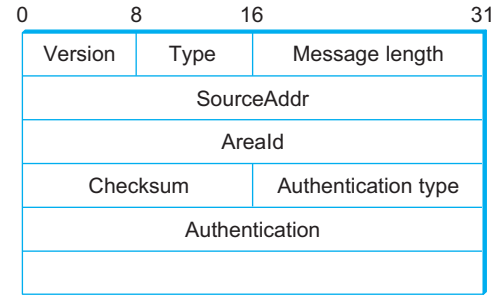


FIGURE 3.34 OSPF header format.

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 that 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 the link-state advertisement (LSA). One message may contain many LSAs. We provide a few details of the LSA here.

Like any internetwork routing protocol, OSPF must provide information about how to reach networks. Thus, OSPF must provide a little more information than the simple graph-based protocol described above. Specifically, a router running OSPF may generate link-state packets that advertise one or more of the networks that are directly connected to that router. In addition, a router that is connected to another router by some link must advertise the cost of reaching that router over the link. These two types of advertisements are necessary to enable all the routers in a domain to determine the cost of reaching all networks in that domain and the appropriate next hop for each network.

Figure 3.35 shows the packet format for a type 1 link-state advertisement. Type 1 LSAs advertise the cost of links between routers. Type 2

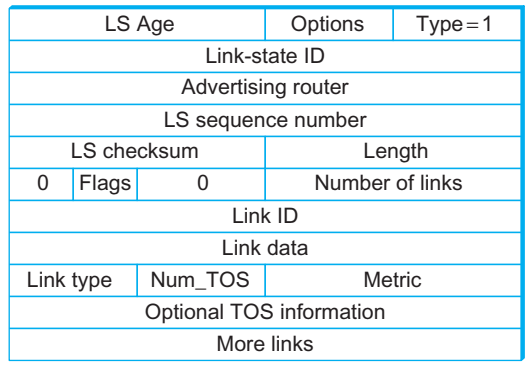


FIGURE 3.35 OSPF link-state advertisement.

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 it is not necessary to recompute a checksum every time LS Age is incremented. Length is the length in bytes of the complete LSA.

Now we get to the actual link-state information. This is made a little complicated by the presence of TOS (type of service) information. Ignoring that for a moment, each link in the LSA is represented by a Link ID, some Link Data, and a metric. The first two of these fields identify the link;

a common way to do this would be to use the router ID of the router at the far end of the link as the Link ID and then use the Link Data to disambiguate among multiple parallel links if necessary. The metric is of course the cost of the link. Type tells us something about the link—for example, if it is a point-to-point link.

The TOS information is present to allow OSPF to choose different routes for IP packets based on the value in their TOS field. Instead of assigning a single metric to a link, it is possible to assign different metrics depending on the TOS value of the data. For example, if we had a link in our network that was very good for delay-sensitive traffic, we could give it a low metric for the TOS value representing low delay and a high metric for everything else. OSPF would then pick a different shortest path for those packets that had their TOS field set to that value. It is worth noting that, at the time of writing, this capability has not been widely deployed.[14]

3.3.4 Metrics

The preceding discussion assumes that link costs, or metrics, are known when we execute the routing algorithm. In this section, we look at some ways to calculate link costs that have proven effective in practice. One example that we have seen already, which is quite reasonable and very simple, is to assign a cost of 1 to all links—the least-cost route will then be the one with the fewest hops. Such an approach has several drawbacks, however. First, it does not distinguish between links on a latency basis. Thus, a satellite link with 250-ms latency looks just as attractive to the routing protocol as a terrestrial link with 1-ms latency. Second, it does not distinguish between routes on a capacity basis, making a 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;

[14] Note also that the meaning of the TOS field has changed since the OSPF specification was written. This topic is discussed in Section 6.5.3.

the original mechanism used distance vector while the later version used link state.) The following discussion traces the evolution of the ARPANET routing metric and, in so doing, explores the subtle aspects of the problem.

The original ARPANET routing metric measured the number of packets that were queued waiting to be transmitted on each link, meaning that a link with 10 packets queued waiting to be transmitted was assigned a larger cost weight than a link with 5 packets queued for transmission. Using queue length as a routing metric did not work well, however, since queue length is an artificial measure of load—it moves packets toward the shortest queue rather than toward the destination, a situation all too familiar to those of us who hop from line to line at the grocery store. Stated more precisely, the original ARPANET routing mechanism suffered from the fact that it did not take either the bandwidth or the latency of the link into consideration.

A second version of the ARPANET routing algorithm, 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 and captured the link's bandwidth and latency, respectively. Notice that in this case, DepartTime − ArrivalTime represents the amount of time the packet was delayed (queued) in the node due to load. If the ACK did not arrive, but instead the packet timed out, then DepartTime was reset to the time the packet was *retransmitted*. In this case, DepartTime − ArrivalTime captures the reliability of the link—the more frequent the retransmission of packets, the less reliable the link, and the more we want to avoid it. Finally, the weight assigned to each link was derived from the average delay experienced by the packets recently sent over that link.

Although an improvement over the original mechanism, this approach also had a lot of problems. Under light load, it worked reasonably well,

since the two static factors of delay dominated the cost. Under heavy load, however, a congested link would start to advertise a very high cost. This caused all the traffic to move off that link, leaving it idle, so then it would advertise a low cost, thereby attracting back all the traffic, and so on. The effect of this instability was that, under heavy load, many links would in fact spend a great deal of time being idle, which is the last thing you want under heavy load.

Another problem was that the range of link values was much too large. For example, a heavily loaded 9.6-kbps link could look 127 times more costly than a lightly loaded 56-kbps link. This means that the routing algorithm would choose a path with 126 hops of lightly loaded 56-kbps links in preference to a 1-hop 9.6-kbps path. While shedding some traffic from an overloaded line is a good idea, making it look so unattractive that it loses all its traffic is excessive. Using 126 hops when 1 hop will do is in general a bad use of network resources. Also, satellite links were unduly penalized, so that an idle 56-kbps satellite link looked considerably more costly than an idle 9.6-kbps terrestrial link, even though the former would give better performance for high-bandwidth applications.

A third approach, called the "revised ARPANET routing metric," addressed these problems. The major changes were to compress the dynamic range of the metric considerably, to account for the link type, and to smooth the variation of the metric with time.

The smoothing was achieved by several mechanisms. First, the delay measurement was transformed to a link utilization, and this number was averaged with the last reported utilization to suppress sudden changes. Second, there was a hard limit on how much the metric could change from one measurement cycle to the next. By smoothing the changes in the cost, the likelihood that all nodes would abandon a route at once is greatly reduced.

The compression of the dynamic range was achieved by feeding the measured utilization, the link type, and the link speed into a function that is shown graphically in Figure 3.36. 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.

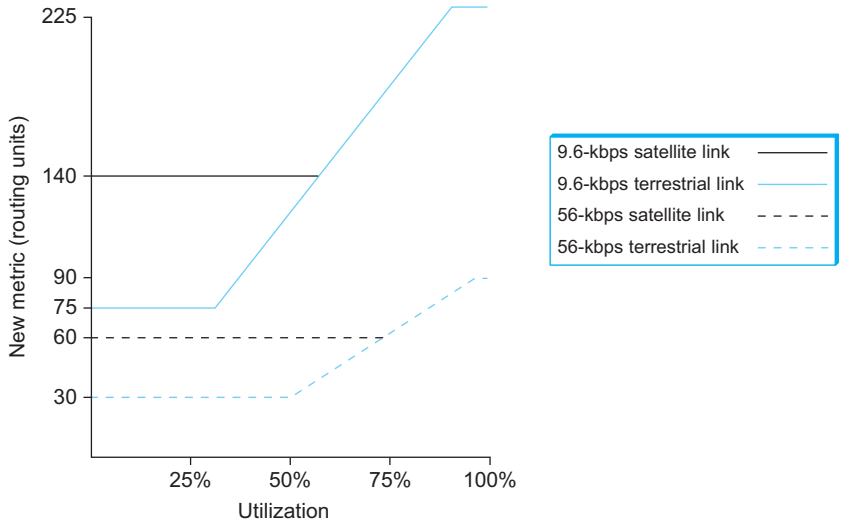


FIGURE 3.36 Revised ARPANET routing metric versus link utilization.

- A high-speed satellite link is more attractive than a low-speed terrestrial link.
- Cost is a function of link utilization only at moderate to high loads.

All of these factors mean that a link is much less likely to be universally abandoned, since a threefold increase in cost is likely to make the link unattractive for some paths while letting it remain the best choice for others. The slopes, offsets, and breakpoints for the curves in Figure 3.36 were arrived at by a great deal of trial and error, and they were carefully tuned to provide good performance.

We end our discussion of routing metrics with a dose of reality. In the majority of real-world network deployments at the time of writing, metrics change rarely if at all and only under the control of a network administrator, not automatically as was described above. The reason for this is partly that conventional wisdom now holds that dynamically changing metrics are too unstable, even though this probably need not be true. Perhaps more significantly, many networks today lack the great disparity of link speeds and latencies that prevailed in the ARPANET. Thus, static metrics are the norm. One common approach to setting metrics is to use a constant multiplied by (1/link_bandwidth).

Monitoring Routing Behavior

Given the complexity of routing packets through a network of the scale of the Internet, we might wonder how well the system works. We know it works some of the time because we are able to connect to sites all over the world. We suspect it doesn't work all the time, though, because sometimes we are unable to connect to certain sites. The real problem is determining what part of the system is at fault when our connections fail: Has some routing machinery failed to work properly, is the remote server too busy, or has some link or machine simply gone down?

This is really an issue of network management, and while there are tools that system administrators use to keep tabs on their own networks—for example, see the Simple Network Management Protocol (SNMP) described in Section 9.3.2—it is a largely unresolved problem for the Internet as a whole. In fact, the Internet has grown so large and complex that, even though it is constructed from a collection of man-made, largely deterministic parts, we have come to view it almost as a living organism or natural phenomenon that is to be studied. That is, we try to understand the Internet's dynamic behavior by performing experiments on it and proposing models that explain our observations.

An excellent example of this kind of study has been conducted by Vern Paxson. Paxson used the Unix traceroute tool to study 40,000 end-to-end routes between 37 Internet sites in 1995. He was attempting to answer questions about how routes fail, how stable routes are over time, and whether or not they are symmetric. Among other things, Paxson found that the likelihood of a user encountering a serious end-to-end routing problem was 1 in 30, and that such problems usually lasted about 30 seconds. He also found that two-thirds of the Internet's routes persisted for days or weeks, and that about one-third of the time the route used to get from host A to host B included at least one different routing domain than the route used to get from host B to host A. Paxson's overall conclusion was that Internet routing was becoming less and less predictable over time.

3.4 IMPLEMENTATION AND PERFORMANCE

So far, we have talked about what switches and routers must do without discussing how to do it. There is a very simple way to build a switch or router: Buy a general-purpose processor and equip it with a number of network interfaces. Such a device, running suitable software, can receive packets on one of its interfaces, perform any of the switching or forwarding functions described above, and send packets out another of its

interfaces. This is, in fact, a popular way to build experimental routers and switches when you want to be able to do things like develop new routing protocols because it offers extreme flexibility and a familiar programming environment. It is also not too far removed from the architecture of many commercial mid- to low-end routers.

3.4.1 Switch Basics

Switches and routers use similar implementation techniques, so we'll start this section by looking at those common techniques, then move on to look at the specific issues affecting router implementation in Section 3.4.4. For most of this section, we'll use the word *switch* to cover both types of devices, since their internal designs are so similar (and it's tedious to say "switch or router" all the time).

Figure 3.37 shows a processor with three network interfaces used as a switch. The figure shows a path that a packet might take from the time it arrives on interface 1 until it is output on interface 2. We have assumed here that the processor has a mechanism to move data directly from an interface to its main memory without having to be directly copied by the CPU, a technique called *direct memory access* (DMA). Once the packet is in memory, the CPU examines its header to determine which interface the packet should be sent out on. It then uses DMA to move the packet out to the appropriate interface. Note that Figure 3.37 does not show the packet

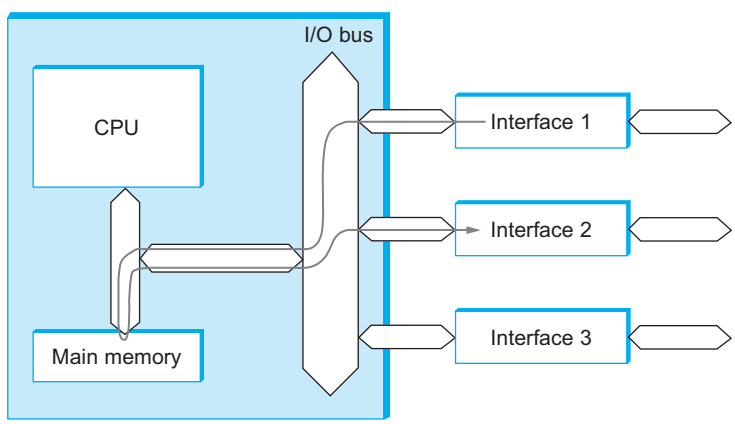


FIGURE 3.37 A general-purpose processor used as a packet switch.

going to the CPU because the CPU inspects only the header of the packet; it does not have to read every byte of data in the packet.

The main problem with using a general-purpose processor as a switch is that its performance is limited by the fact that all packets must pass through a single point of contention: In the example shown, each packet crosses the I/O bus twice and is written to and read from main memory once. The upper bound on aggregate throughput of such a device (the total sustainable data rate summed over all inputs) is, thus, either half the main memory bandwidth or half the I/O bus bandwidth, whichever is less. (Usually, it's the I/O bus bandwidth.) For example, a machine with a 133-MHz, 64-bit-wide I/O bus can transmit data at a peak rate of a little over 8 Gbps. Since forwarding a packet involves crossing the bus twice, the actual limit is 4 Gbps—enough to build a switch with a fair number of 100-Mbps Ethernet ports, for example, but hardly enough for a high-end router in the core of the Internet.

Moreover, this upper bound also assumes that moving data is the only problem—a fair approximation for long packets but a bad one when packets are short. In the latter case, the cost of processing each packet—parsing its header and deciding which output link to transmit it on—is likely to dominate. Suppose, for example, that a processor can perform all the necessary processing to switch 2 million packets each second. This is sometimes called the packet per second (pps) rate. (This number is representative of what is achievable on an inexpensive PC.) If the average packet is short, say, 64 bytes, this would imply

$$\begin{aligned}\text{Throughput} &= \text{pps} \times (\text{BitsPerPacket}) \\ &= 2 \times 10^6 \times 64 \times 8 \\ &= 1024 \times 10^6\end{aligned}$$

that is, a throughput of about 1 Gbps—substantially below the range that users are demanding from their networks today. Bear in mind that this 1 Gbps would be shared by all users connected to the switch, just as the bandwidth of a single (unswitched) Ethernet segment is shared among all users connected to the shared medium. Thus, for example, a 20-port switch with this aggregate throughput would only be able to cope with an average data rate of about 50 Mbps on each port.

To address this problem, hardware designers have come up with a large array of switch designs that reduce the amount of contention and provide

high aggregate throughput. Note that some contention is unavoidable: If every input has data to send to a single output, then they cannot all send it at once. However, if data destined for different outputs is arriving at different inputs,then a well-designed switch will be able to move data from inputs to outputs in parallel, thus increasing the aggregate throughput.

Defining Throughput

It turns out to be difficult to define precisely the throughput of a switch. Intuitively, we might think that if a switch has n inputs that each support a link speed of s_i, then the throughput would just be the sum of all the s_i. This is actually the best possible throughput that such a switch could provide, but in practice almost no real switch can guarantee that level of performance. One reason for this is simple to understand. Suppose that, for some period of time, all the traffic arriving at the switch needed to be sent to the same output. As long as the bandwidth of that output is less than the sum of the input bandwidths, then some of the traffic will need to be either buffered or dropped. With this particular traffic pattern, the switch could not provide a sustained throughput higher than the link speed of that one output. However, a switch might be able to handle traffic arriving at the full link speed on all inputs if it is distributed across all the outputs evenly; this would be considered optimal.

Another factor that affects the performance of switches is the size of packets arriving on the inputs. For an ATM switch, this is normally not an issue because all "packets" (cells) are the same length. But, for Ethernet switches or IP routers, packets of widely varying sizes are possible. Some of the operations that a switch must perform have a constant overhead per packet, so a switch is likely to perform differently depending on whether all arriving packets are very short, very long, or mixed. For this reason, routers or switches that forward variable-length packets are often characterized by a *packet per second* rate as well as a throughput in bits per second. The pps rate is usually measured with minimum-sized packets.

The first thing to notice about this discussion is that the throughput of the switch is a function of the traffic to which it is subjected. One of the things that switch designers spend a lot of their time doing is trying to come up with traffic models that approximate the behavior of real data traffic. It turns out that it is extremely difficult to achieve accurate models. There are several elements to a traffic model. The main ones are (1) when the packets arrive, (2) what outputs they are destined for, and (3) how big they are.

Traffic modeling is a well-established science that has been extremely successful in the world of telephony, enabling telephone companies to engineer their networks to carry expected loads quite efficiently. This is partly because the way people use the phone network does not change that much

over time: The frequency with which calls are placed, the amount of time taken for a call, and the tendency of everyone to make calls on Mother's Day have stayed fairly constant for many years. By contrast, the rapid evolution of computer communications, where a new application like BitTorrent can change the traffic patterns almost overnight, has made effective modeling of computer networks much more difficult. Nevertheless, there are some excellent books and articles on the subject that we list at the end of the chapter.

To give you a sense of the range of throughputs that designers need to be concerned about, a single rack router used in the core of the Internet at the time of writing might support 16 OC-768 links for a throughput of approximately 640 Gbps. A 640-Gbps switch, if called upon to handle a steady stream of 64-byte packets, would need a packet per second rate of

$$640 \times 10^9 \div (64 \times 8) = 1.25 \times 10^9 \text{ pps}$$

3.4.2 Ports

Most switches look conceptually similar to the one shown in Figure 3.38. They consist of a number of *input* and *output ports* and a *fabric*. There is usually at least one control processor in charge of the whole switch that communicates with the ports either directly or, as shown here, via the switch fabric. The ports communicate with the outside world. They may contain fiber optic receivers and lasers, buffers to hold packets that are waiting to be switched or transmitted, and often a significant amount of other circuitry that enables the switch to function. The fabric has a very

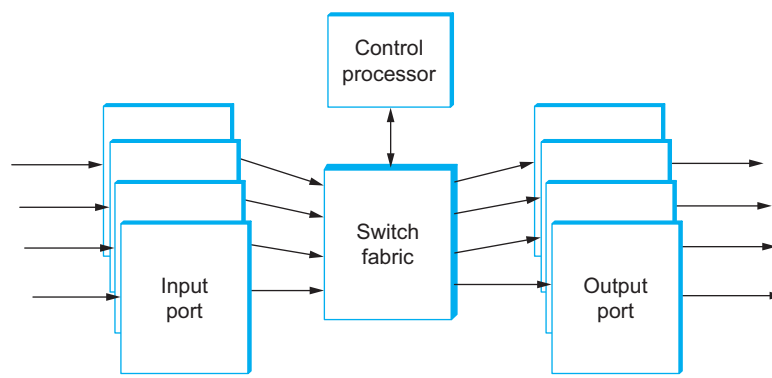


FIGURE 3.38 A 4 × 4 switch.

simple and well-defined job: When presented with a packet, deliver it to the right output port.

One of the jobs of the ports, then, is to deal with the complexity of the real world in such a way that the fabric can do its relatively simple job. For example, suppose that this switch is supporting a virtual circuit model of communication. In general, the virtual circuit mapping tables described in Section 3.1.2 are located in the ports. The ports maintain lists of virtual circuit identifiers that are currently in use, with information about what output a packet should be sent out on for each VCI and how the VCI needs to be remapped to ensure uniqueness on the outgoing link. Similarly, the ports of an Ethernet switch store tables that map between Ethernet addresses and output ports (bridge forwarding tables as described in Section 3.1.4). In general, when a packet is handed from an input port to the fabric, the port has figured out where the packet needs to go, and either the port sets up the fabric accordingly by communicating some control information to it, or it attaches enough information to the packet itself (e.g., an output port number) to allow the fabric to do its job automatically. Fabrics that switch packets by looking only at the information in the packet are referred to as *self-routing*, since they require no external control to route packets. An example of a self-routing fabric is discussed below.

The input port is the first place to look for performance bottlenecks. The input port has to receive a steady stream of packets, analyze information in the header of each one to determine which output port (or ports) the packet must be sent to, and pass the packet on to the fabric. The type of header analysis that it performs can range from a simple table lookup on a VCI to complex matching algorithms that examine many fields in the header. This is the type of operation that sometimes becomes a problem when the average packet size is very small. Consider, for example, 64-byte packets arriving on a port connected to an OC-48 (2.48 Gbps) link. Such a port needs to process packets at a rate of

$$2.48 \times 10^9 \div (64 \times 8) = 4.83 \times 10^6 \text{ pps}$$

In other words, when small packets are arriving as fast as possible on this link (the worst-case scenario that most ports are engineered to handle), the input port has approximately 200 nanoseconds to process each packet.

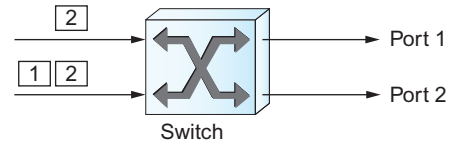


FIGURE 3.39 Simple illustration of head-of-line blocking.

Another key function of ports is buffering. Observe that buffering can happen in either the input or the output port; it can also happen within the fabric (sometimes called *internal buffering*). Simple input buffering has some serious limitations. Consider an input buffer implemented as a FIFO. As packets arrive at the switch, they are placed in the input buffer. The switch then tries to forward the packets at the front of each FIFO to their appropriate output port. However, if the packets at the front of several different input ports are destined for the same output port at the same time, then only one of them can be forwarded;[15] the rest must stay in their input buffers.

The drawback of this feature is that those packets left at the front of the input buffer prevent other packets further back in the buffer from getting a chance to go to their chosen outputs, even though there may be no contention for those outputs. This phenomenon is called *head-of-line blocking*. A simple example of head-of-line blocking is given in Figure 3.39, where we see a packet destined for port 1 blocked behind a packet contending for port 2. It can be shown that when traffic is uniformly distributed among outputs, head-of-line blocking limits the throughput of an input-buffered switch to 59% of the theoretical maximum (which is the sum of the link bandwidths for the switch). Thus, the majority of switches use either pure output buffering or a mixture of internal and output buffering. Those that do rely on input buffers use more advanced buffer management schemes to avoid head-of-line blocking.

Buffers actually perform a more complex task than just holding onto packets that are waiting to be transmitted. Buffers are the main source of delay in a switch, and also the place where packets are most likely to

[15] For a simple input-buffered switch, exactly one packet at a time can be sent to a given output port. It is possible to design switches that can forward more than one packet to the same output at once, at a cost of higher switch complexity, but there is always some upper limit on the number.

get dropped due to lack of space to store them. The buffers therefore are the main place where the quality of service characteristics of a switch are determined. For example, if a certain packet has been sent along a VC that has a guaranteed delay, it cannot afford to sit in a buffer for very long. This means that the buffers, in general, must be managed using packet scheduling and discard algorithms that meet a wide range of QoS requirements. We talk more about these issues in Chapter 6.

3.4.3 Fabrics

While there has been an abundance of impressive research conducted on the design of efficient and scalable fabrics, it is sufficient for our purposes here to understand only the high-level properties of a switch fabric. A switch fabric should be able to move packets from input ports to output ports with minimal delay and in a way that meets the throughput goals of the switch. That usually means that fabrics display some degree of parallelism. A high-performance fabric with n ports can often move one packet from each of its n ports to one of the output ports at the same time. A sample of fabric types includes the following:

- *Shared Bus*—This is the type of "fabric" found in a conventional processor used as a switch, as described above. Because the bus bandwidth determines the throughput of the switch, high-performance switches usually have specially designed busses rather than the standard busses found in PCs.
- *Shared Memory*—In a shared memory switch, packets are written into a memory location by an input port and then read from memory by the output ports. Here it is the memory bandwidth that determines switch throughput, so wide and fast memory is typically used in this sort of design. A shared memory switch is similar in principle to the shared bus switch, except it usually uses a specially designed, high-speed memory bus rather than an I/O bus.
- *Crossbar*—A crossbar switch is a matrix of pathways that can be configured to connect any input port to any output port. Figure 3.40 shows a 4×4 crossbar switch. The main problem with crossbars is that, in their simplest form, they require each output port to be able to accept packets from all inputs at once, implying that each port would have a memory bandwidth equal to the total switch throughput. In reality, more complex designs are typically

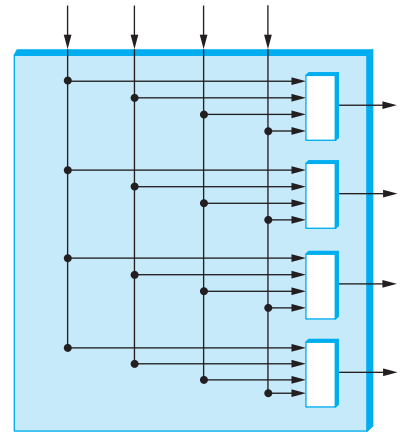
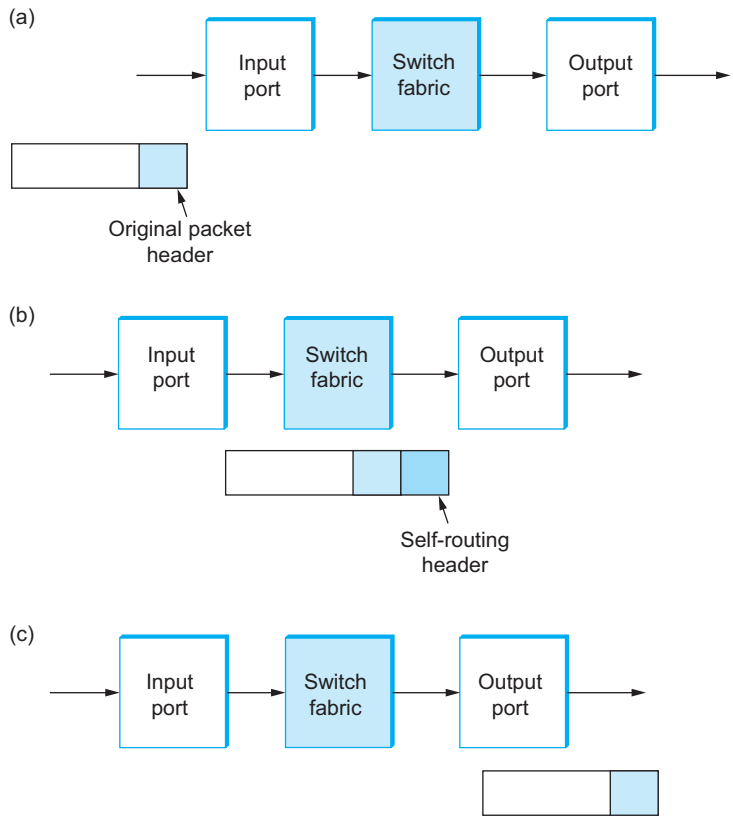


FIGURE 3.40 A 4×4 crossbar switch.

used to address this issue (see, for example, the Knockout switch and McKeown's virtual output-buffered approach in the Further Reading section.)

- *Self-routing*—As noted above, self-routing fabrics rely on some information in the packet header to direct each packet to its correct output. Usually a special "self-routing header" is appended to the packet by the input port after it has determined which output the packet needs to go to, as illustrated in Figure 3.41; this extra header is removed before the packet leaves the switch. Self-routing fabrics are often built from large numbers of very simple 2×2 switching elements interconnected in regular patterns, such as the *banyan* switching fabric shown in Figure 3.42. For some examples of self-routing fabric designs, see the Further Reading section at the end of this chapter.

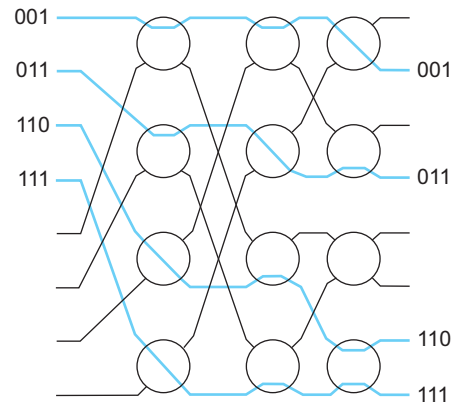
Self-routing fabrics are among the most scalable approaches to fabric design, and there has been a wealth of research on the topic, some of which is listed in the Further Reading section. Many self-routing fabrics resemble the one shown in Figure 3.42, consisting of regularly interconnected 2×2 switching elements. For example, the 2×2 switches in the banyan network perform a simple task: They look at 1 bit in each self-routing header and route packets toward the upper output if it is zero or toward the lower output if it is one. Obviously, if two packets arrive at a



■ **FIGURE 3.41** A self-routing header is applied to a packet at input to enable the fabric to send the packet to the correct output, where it is removed: (a) Packet arrives at input port; (b) input port attaches self-routing header to direct packet to correct output; (c) self-routing header is removed at output port before packet leaves switch.

banyan element at the same time and both have the bit set to the same value, then they want to be routed to the same output and a collision will occur. Either preventing or dealing with these collisions is a main challenge for self-routing switch design. The banyan network is a clever arrangement of 2×2 switching elements that routes all packets to the correct output without collisions if the packets are presented in ascending order.

We can see how this works in an example, as shown in Figure 3.42, where the self-routing header contains the output port number encoded in binary. The switch elements in the first column look at the most significant bit of the output port number and route packets to the top if that bit



■ **FIGURE 3.42** Routing packets through a banyan network. The 3-bit numbers represent values in the self-routing headers of four arriving packets.

is a 0 or the bottom if it is a 1. Switch elements in the second column look at the second bit in the header, and those in the last column look at the least significant bit. You can see from this example that the packets are routed to the correct destination port without collisions. Notice how the top outputs from the first column of switches all lead to the top half of the network, thus getting packets with port numbers 0 to 3 into the right half of the network. The next column gets packets to the right quarter of the network, and the final column gets them to the right output port. The clever part is the way switches are arranged to avoid collisions. Part of the arrangement includes the "perfect shuffle" wiring pattern at the start of the network. To build a complete switch fabric around a banyan network would require additional components to sort packets before they are presented to the banyan. The Batcher-banyan switch design is a notable example of such an approach. The Batcher network, which is also built from a regular interconnection of 2×2 switching elements, sorts packets into descending order. On leaving the Batcher network, the packets are then ready to be directed to the correct output, with no risk of collisions, by the banyan network.

One of the interesting things about switch design is the wide range of different types of switches that can be built using the same basic technology. For example, Ethernet switches, ATM switches, and Internet routers, discussed below, have all been built using designs such as those outlined in this section.

3.4.4 Router Implementation

We have now seen a variety of ways to build a switch, ranging from a general-purpose processor with a suitable number of network interfaces to some sophisticated hardware designs. In general, the same range of options is available for building routers, many of which look something like Figure 3.43. The control processor is responsible for running the routing protocols discussed above, among other things, and generally acts as the central point of control of the router. The switching fabric transfers packets from one port to another, just as in a switch; and the ports provide a range of functionality to allow the router to interface to links of various types (e.g., Ethernet, SONET).

A few points are worth noting about router design and how it differs from switch design. First, routers must be designed to handle variable-length packets, a constraint that does not apply to ATM switches but is certainly applicable to Ethernet or Frame Relay switches. It turns out that many high-performance routers are designed using a switching fabric that is cell based. In such cases, the ports must be able to convert variable-length packets into cells and back again. This is known as *segmentation and re-assembly* (SAR), a problem also faced by network adaptors for ATM networks.

Another consequence of the variable length of IP datagrams is that it can be harder to characterize the performance of a router than a switch that forwards only cells. Routers can usually forward a certain number of packets per second, and this implies that the total throughput in *bits*

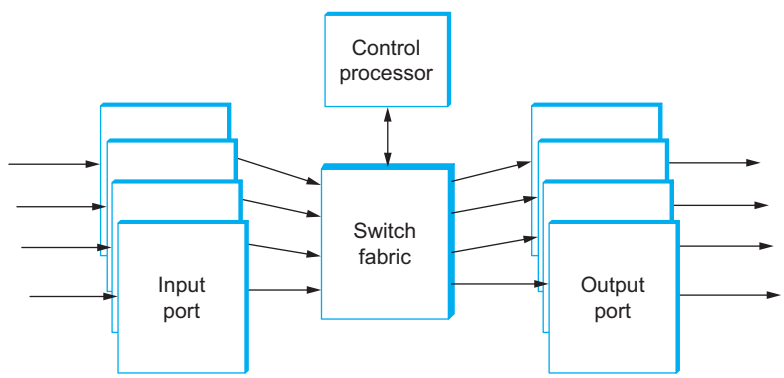


FIGURE 3.43 Block diagram of a router.

per second depends on packet size. Router designers generally have to make a choice as to what packet length they will support at *line rate*. That is, if pps (packets per second) is the rate at which packets arriving on a particular port can be forwarded, and linerate is the physical speed of the port in bits per second, then there will be some packetsize in bits such that:

$$\text{packetsize} \times \text{pps} = \text{linerate}$$

This is the packet size at which the router can forward at line rate; it is likely to be able to sustain line rate for longer packets but not for shorter packets. Sometimes a designer might decide that the right packet size to support is 40 bytes, since that is the minimum size of an IP packet that has a TCP header attached. Another choice might be the expected *average* packet size, which can be determined by studying traces of network traffic. For example, measurements of the Internet backbone suggest that the average IP packet is around 300 bytes long. However, such a router would fall behind and perhaps start dropping packets when faced with a long sequence of short packets, which is statistically likely from time to time and also very possible if the router is subject to an active attack (see Chapter 8). Design decisions of this type depend heavily on cost considerations and the intended application of the router.

When it comes to the task of forwarding IP packets, routers can be broadly characterized as having either a *centralized* or *distributed* forwarding model. In the centralized model, the IP forwarding algorithm, outlined earlier in this chapter, is done in a single processing engine that handles the traffic from all ports. In the distributed model, there are several processing engines, perhaps one per port, or more often one per line card, where a line card may serve one or more physical ports. Each model has advantages and disadvantages. All things being equal, a distributed forwarding model should be able to forward more packets per second through the router as a whole, because there is more processing power in total. But a distributed model also complicates the software architecture, because each forwarding engine typically needs its own copy of the forwarding table, and thus it is necessary for the control processor to ensure that the forwarding tables are updated consistently and in a timely manner.

Another aspect of router implementation that is significantly different from that of switches is the IP forwarding algorithm itself. In bridges and

most ATM switches, the forwarding algorithm simply involves looking up a fixed-length identifier (MAC address or VCI) in a table, finding the correct output port in the table, and sending the packet to that port. We have already seen in Section 3.2.4 that the IP forwarding algorithm is a little more complicated than that, in part because the relevant number of bits that need to be examined when forwarding a packet is not fixed but variable, typically ranging from 8 bits to 32 bits.

Because of the relatively high complexity of the IP forwarding algorithm, there have been periods of time when it seemed IP routers might be running up against fundamental upper limits of performance. However, as we discuss in the Further Reading section of this chapter, there have been many innovative approaches to IP forwarding developed over the years, and at the time of writing there are commercial routers that can forward 40 Gbps of IP traffic *per interface*. By combining many such high-performance IP forwarding engines with the sort of very scalable switch fabrics discussed in Section 3.4, it has now become possible to build routers with many terabits of total throughput. That is more than enough to see us through the next few years of growth in Internet traffic.

Another technology of interest in the field of router implementation is the *network processor*. A network processor is intended to be a device that is just about as programmable as a standard PC processor, but that is more highly optimized for networking tasks. For example, a network processor might have instructions that are particularly well suited to performing lookups on IP addresses, or calculating checksums on IP datagrams. Such devices could be used in routers and other networking devices (e.g., firewalls).

One of the interesting and ongoing debates about network processors is whether they can do a better job than the alternatives. For example, given the continuous and remarkable improvements in performance of conventional processors, and the huge industry that drives those improvements, can network processors keep up? And can a device that strives for generality do as good a job as a custom-designed application-specific integrated circuit (ASIC) that does nothing except, say, IP forwarding? Part of the answer to questions like these depends on what you mean by "do a better job." For example, there will always be trade-offs to be made between cost of hardware, time to market, performance, power consumption, and flexibility—the ability to change the features

supported by a router after it is built. We will see in later chapters just how diverse the requirements for router functionality can be. It is safe to assume that a wide range of router designs will exist for the foreseeable future and that network processors will have some role to play.

3.5 SUMMARY

This chapter has begun to look at some of the issues involved in building scalable and heterogeneous networks by using switches and routers to interconnect links and networks. The most common use of switching is the interconnection of LANs, especially Ethernet segments. LAN switches, or bridges, use techniques such as source address learning to improve forwarding efficiency and spanning tree algorithms to avoid looping. These switches are extensively used in data centers, campuses, and corporate networks.

To deal with heterogeneous networks, the Internetworking Protocol (IP) was invented and forms the basis of today's routers. IP tackles heterogeneity by defining a simple, common service model for an internetwork, which is based on the best-effort delivery of IP datagrams. An important part of the service model is the global addressing scheme, which enables any two nodes in an internetwork to uniquely identify each other for the purposes of exchanging data. The IP service model is simple enough to be supported by any known networking technology, and the ARP mechanism is used to translate global IP addresses into local link-layer addresses.

A crucial aspect of the operation of an internetwork is the determination of efficient routes to any destination in the internet. Internet routing algorithms solve this problem in a distributed fashion; this chapter introduced the two major classes of algorithms—link-state and distance-vector—along with examples of their application (RIP and OSPF).

Both switches and routers need to forward packets from inputs to outputs at a high rate and, in some circumstances, grow to a large size to accommodate hundreds or thousands of ports. Building switches that both scale and offer high performance at acceptable cost is complicated by the problem of contention; as a consequence, switches and routers often employ special-purpose hardware rather than being built from general-purpose workstations.

The Internet has without doubt been an enormous success, and it can be easy to forget that there was ever a time when it didn't exist. However, the inventors of the Internet developed it in part because the networks available at the time, such as the circuit-switched telephone network, were not well suited to the things they wanted to do. Now that the Internet is an established artifact, just as the telephone network was in the 1960s, it is reasonable to ask: What comes after the Internet?

No one knows the answer to that question at the moment, but some significant research efforts are underway to try to enable some sort of "Future Internet." While it is difficult to imagine that today's Internet will be replaced by something new any time soon (after all, the telephone network is still around, although increasingly its traffic is moving onto the Internet), thinking beyond the constraints of incrementally deployable tweaks to today's Internet could enable some new innovations that we would otherwise miss. It is popular to talk about "clean slate" research in this context—such research looks at what might be possible if we *could* start from scratch, postponing deployment considerations for later.

WHAT'S NEXT: THE FUTURE INTERNET

For example, what if we assumed that every node in the Internet was mobile? We would probably start with a different way of identifying nodes—rather than an IP address, which includes information about what network the node is currently attached to, we might use some other form of identifier. Or, as another example, we might consider a different trust model than the one built into the current Internet. When the Internet was originally developed, it seemed reasonable to assume that every host should be able to send to every other host by default, but today in the world of spammers, phishers, and denial-of-service attacks, a different trust model—with more limited initial capabilities for newly connected or unknown nodes perhaps—might be considered. These two examples illustrate cases where, knowing today some things that were not apparent in the '70s (like the importance of mobility and security to networking), we might want to come up with a very different design for an internetwork.

A couple of points can be made here. First, you should not assume that the Internet is "done." Its architecture is inherently flexible and it will continue to evolve. We will see some examples of its evolution in the next chapter. The other point is that there's more than one way to do networking research: Developing incrementally deployable ideas is great, but in the words of Internet pioneer David Clark, "To conceive the future, it helps to let go of the present."

FURTHER READING

The seminal paper on bridges, in particular the spanning tree algorithm, is the article by Perlman below. Not surprisingly, countless papers have been written on various aspects of the Internet; the paper by Cerf and Kahn is the one that originally introduced the TCP/IP architecture and is worth reading for its historical perspective. Finally, McKeown's paper, one of many on switch design, describes an approach to switch design that uses cells internally but has been used commercially as the basis for high-performance routers forwarding variable-length IP packets.

- Perlman, R. An algorithm for distributed computation of spanning trees in an extended LAN. *Proceedings of the Ninth Data Communications Symposium*, pages 44–53, September 1985.
- Cerf, V., and R. Kahn. A protocol for packet network intercommunication. *IEEE Transactions on Communications* COM-22(5):637–648, May 1974.
- McKeown, N. The *i*SLIP scheduling algorithm for input-queued switches. *IEEE Transactions on Networking* 7(2):188–201, April 1999.

A good general overview of bridges and routers can be found in another work by Perlman [Per00]. There is a wealth of papers on ATM; Turner [Tur85], an ATM pioneer, was one of the first to propose the use of a cell-based network for integrated services.

Many of the techniques and protocols that are central to today's Internet are described in requests for comments (RFCs): Subnetting is described in Mogul and Postel [MP85], CIDR is described in Fuller and Li [FL06], RIPv2 is defined in Malkin [Mal98], and OSPF is defined in Moy [Moy98]. The OSPF specification, at over 200 pages, is one of the longer RFCs around, but it contains an unusual wealth of detail about

how to implement a protocol. The reasons to avoid IP fragmentation are examined in Kent and Mogul [KM87] and the Path MTU discovery technique is described in Mogul and Deering [MD90].

A forward-looking paper about research on the future Internet was written by Clark et al. [CPB+05]. This paper is related to the ongoing research efforts around a future Internet for which we provide live references below.

Literally thousands of papers have been published on switch architectures. One early paper that explains Batcher networks well is, not surprisingly, one by Batcher himself [Bat68]. Sorting networks are explained by Drysdale and Young [DY75], and an interesting form of cross-bar switch is described by Yeh et al. [YHA87]. Giacopelli et al. [GHMS91] describe the "Sunshine" switch, which provides insights into the important role of traffic analysis in switch design. In particular, the Sunshine designers were among the first to realize that cells were likely to arrive at a switch in bursts and thus were able to factor correlated arrivals into their design. A good overview of the performance of different switching fabrics can be found in Robertazzi [Rob93]. An example of the design of a switch based on variable-length packets can be found in Gopal and Guerin [GG94].

There has been a lot of work aimed at developing algorithms that can be used by routers to do fast lookup of IP addresses. (Recall that the problem is that the router needs to match the longest prefix in the forwarding table.) PATRICIA trees are one of the first algorithms applied to this problem [Mor68]. More recent work is reported in [DBCP97], [WVTP97], [LS98], [SVSM98], and [EVD04]. For an overview of how algorithms like these can be used to build a high-speed router, see Partridge et al. [Par98].

Optical networking is a rich field in its own right, with its own journals, conferences, etc. We recommend Ramaswami et al. [RS01] as a good introductory text in that field.

An excellent text to read if you want to learn about the mathematical analysis of network performance is by Kleinrock [Kle75], one of the pioneers of the ARPANET. Many papers have been published on the applications of queuing theory to packet switching. We recommend the article by Paxson and Floyd [PF94] as a significant contribution focused on the Internet, and one by Leland et al. [LTWW94], a paper that introduces the important concept of "long-range dependence" and shows the inadequacy of many traditional approaches to traffic modeling.

Finally, we recommend the following live references:

- http://www.nets-find.net/: A website of the U.S. National Science Foundation that covers the "Future Internet Design" research program.
- http://www.geni.net/: A site describing the GENI networking testbed, which has been created to enable some of the "clean slate" research described above.

EXERCISES

1. Using the example network given in Figure 3.44, give the virtual circuit tables for all the switches after each of the following connections is established. Assume that the sequence of connections is cumulative; that is, the first connection is still up when the second connection is established, and so on. Also assume that the VCI assignment always picks the lowest unused

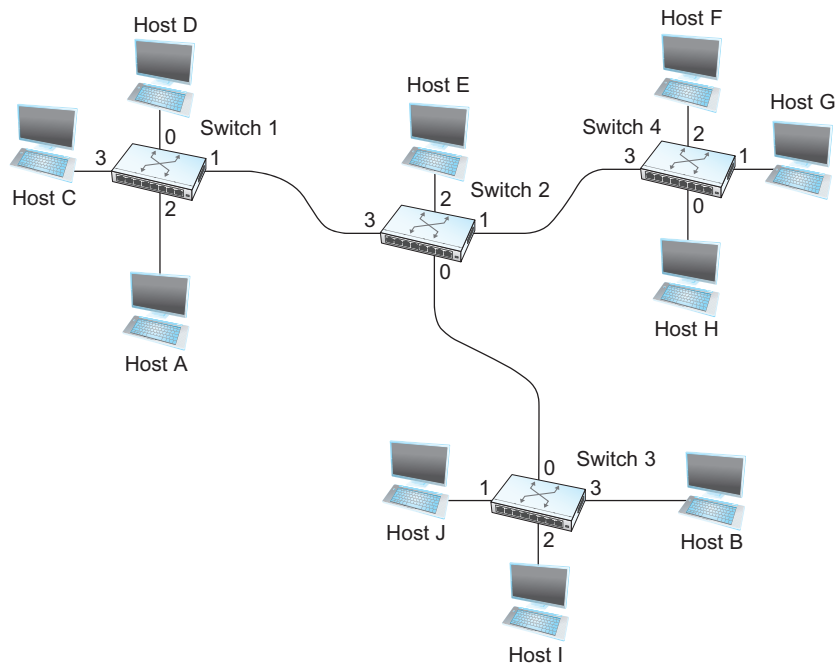


FIGURE 3.44 Example network for Exercises 1 and 2.

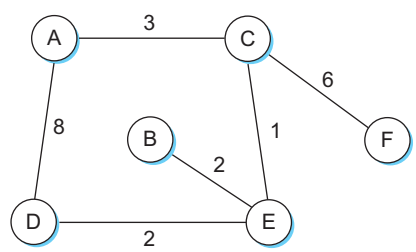
VCI on each link, starting with 0, and that a VCI is consumed for both directions of a virtual circuit.

(a) Host A connects to host C.
(b) Host D connects to host B.
(c) Host D connects to host I.
(d) Host A connects to host B.
(e) Host F connects to host J.
(f) Host H connects to host A.

2. Using the example network given in Figure 3.44, give the virtual circuit tables for all the switches after each of the following connections is established. Assume that the sequence of connections is cumulative; that is, the first connection is still up when the second connection is established, and so on. Also assume that the VCI assignment always picks the lowest unused VCI on each link, starting with 0, and that a VCI is consumed for both directions of a virtual circuit.

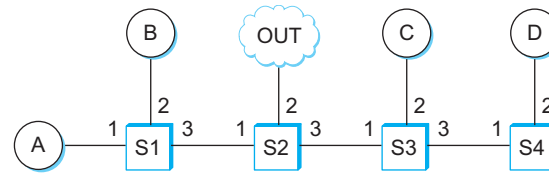
(a) Host D connects to host H.
(b) Host B connects to host G.
(c) Host F connects to host A.
(d) Host H connects to host C.
(e) Host I connects to host E.
(f) Host H connects to host J.

3. For the network given in Figure 3.45, give the datagram forwarding table for each node. The links are labeled with relative costs; your tables should forward each packet via the lowest-cost path to its destination.



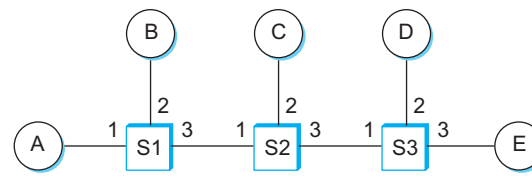
■ **FIGURE 3.45** Network for Exercise 3.

4. Give forwarding tables for switches S1 to S4 in Figure 3.46. Each switch should have a default routing entry, chosen to forward packets with unrecognized destination addresses toward OUT. Any specific-destination table entries duplicated by the default entry should then be eliminated.



■ **FIGURE 3.46** Diagram for Exercise 4.

5. Consider the virtual circuit switches in Figure 3.47. Table 3.15 lists, for each switch, what ⟨port, VCI⟩ (or ⟨VCI, interface⟩) pairs are connected to what other. Connections are bidirectional. List all endpoint-to-endpoint connections.



■ **FIGURE 3.47** Diagram for Exercise 5.

6. In the source routing example of Section 3.1.3, the address received by B is not reversible and doesn't help B know how to reach A. Propose a modification to the delivery mechanism that does allow for reversibility. Your mechanism should *not* require giving all switches globally unique names.

7. Propose a mechanism that virtual circuit switches might use so that if one switch loses all its state regarding connections then a sender of packets along a path through that switch is informed of the failure.

8. Propose a mechanism that might be used by datagram switches so that if one switch loses all or part of its forwarding table affected senders are informed of the failure.

Table 3.15 VCI Tables for Switches in Figure 3.47

Switch S1			
Port	**VCI**	**Port**	**VCI**
1	2	3	1
1	1	2	3
2	1	3	2
Switch S2			
Port	**VCI**	**Port**	**VCI**
1	1	3	3
1	2	3	2
Switch S3			
Port	**VCI**	**Port**	**VCI**
1	3	2	1
1	2	2	2

9. The virtual circuit mechanism described in Section 3.1.2 assumes that each link is point-to-point. Extend the forwarding algorithm to work in the case that links are shared-media connections (e.g., Ethernet).

10. Suppose, in Figure 3.2, that a new link has been added, connecting switch 3 port 1 (where G is now) and switch 1 port 0 (where D is now); neither switch is informed of this link. Furthermore, switch 3 mistakenly thinks that host B is reached via port 1.

(a) What happens if host A attempts to send to host B, using datagram forwarding?

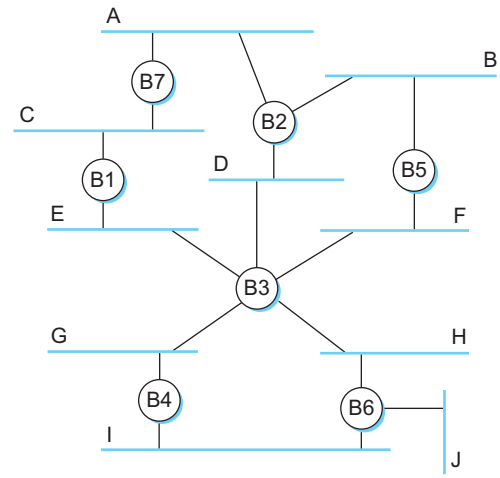
(b) What happens if host A attempts to connect to host B, using the virtual circuit setup mechanism discussed in the text?

11. Give an example of a working virtual circuit whose path traverses some link twice. Packets sent along this path should *not*, however, circulate indefinitely.

12. In Section 3.1.2, each switch chose the VCI value for the incoming link. Show that it is also possible for each switch to choose the VCI value for the outbound link and that the same VCI values will be

chosen by each approach. If each switch chooses the outbound VCI, is it still necessary to wait one RTT before data is sent?

13. Given the extended LAN shown in Figure 3.48, indicate which ports are not selected by the spanning tree algorithm.



■ **FIGURE 3.48** Network for Exercises 13 and 14.

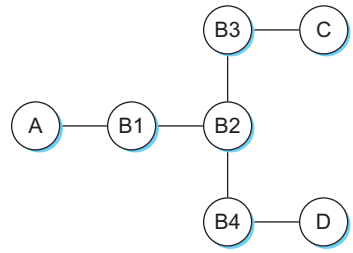
14. Given the extended LAN shown in Figure 3.48, assume that bridge B1 suffers catastrophic failure. Indicate which ports are not selected by the spanning tree algorithm after the recovery process and a new tree has been formed.

15. Consider the arrangement of learning bridges shown in Figure 3.49. Assuming all are initially empty, give the forwarding tables for each of the bridges B1 to B4 after the following transmissions:

- A sends to C.
- C sends to A.
- D sends to C.

Identify ports with the unique neighbor reached directly from that port; that is, the ports for B1 are to be labeled "A" and "B2."

16. As in the previous problem, consider the arrangement of learning bridges shown in Figure 3.49. Assuming all are initially empty, give



■ **FIGURE 3.49** Network for Exercises 15 and 16.

the forwarding tables for each of the bridges B1 to B4 after the following transmissions:

- D sends to C.
- C sends to D.
- A sends to C.

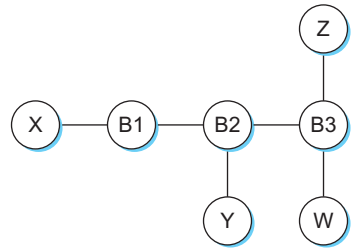
17. Consider hosts X, Y, Z, W and learning bridges B1, B2, B3, with initially empty forwarding tables, as in Figure 3.50.

(a) Suppose X sends to W. Which bridges learn where X is? Does Y's network interface see this packet?

(b) Suppose Z now sends to X. Which bridges learn where Z is? Does Y's network interface see this packet?

(c) Suppose Y now sends to X. Which bridges learn where Y is? Does Z's network interface see this packet?

(d) Finally, suppose W sends to Y. Which bridges learn where W is? Does Z's network interface see this packet?



■ **FIGURE 3.50** Diagram for Exercise 17.

18. Give the spanning tree generated for the extended LAN shown in Figure 3.51, and discuss how any ties are resolved.

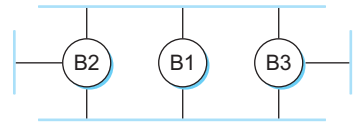


FIGURE 3.51 Extended LAN for Exercise 18.

19. Suppose learning bridges B1 and B2 form a loop as shown in Figure 3.52, and do *not* implement the spanning tree algorithm. Each bridge maintains a single table of ⟨*address, interface*⟩ pairs.

(a) What will happen if M sends to L?

(b) Suppose a short while later L replies to M. Give a sequence of events that leads to one packet from M and one packet from L circling the loop in opposite directions.

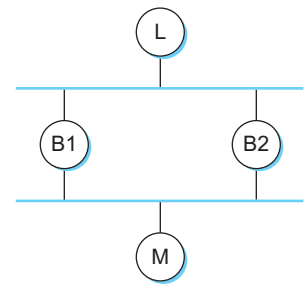


FIGURE 3.52 Loop for Exercises 19 and 20.

20. Suppose that M in Figure 3.52 sends to itself (this normally would never happen). State what would happen, assuming:

(a) The bridges' learning algorithm is to install (or update) the new ⟨*sourceaddress, interface*⟩ entry *before* searching the table for the destination address.

(b) The new source address was installed *after* destination address lookup.

21. Consider the extended LAN of Figure 3.10. What happens in the spanning tree algorithm if bridge B1 does not participate and

(a) Simply forwards all spanning tree algorithm messages?

(b) Drops all spanning tree messages?

22. Suppose some repeaters (hubs), rather than bridges, are connected into a loop.
    (a) What will happen when somebody transmits?
    (b) Why would the spanning tree mechanism be difficult or impossible to implement for repeaters?
    (c) Propose a mechanism by which repeaters might detect loops and shut down some ports to break the loop. Your solution is not required to work 100% of the time.

23. Suppose a bridge has two of its ports on the same network. How might the bridge detect and correct this?

24. What percentage of an ATM link's total bandwidth is consumed by the ATM cell headers? Ignore padding to fill cells or ATM adaptation layer headers.

25. Cell switching methods (like ATM) essentially always use virtual circuit switching rather than datagram forwarding. Give a specific argument why this is so (consider the preceding question).

26. Suppose a workstation has an I/O bus speed of 800 Mbps and memory bandwidth of 2 Gbps. Assuming direct memory access (DMA) is used to move data in and out of main memory, how many interfaces to 100-Mbps Ethernet links could a switch based on this workstation handle?

27. Suppose a workstation has an I/O bus speed of 1 Gbps and memory bandwidth of 2 Gbps. Assuming DMA is used to move data in and out of main memory, how many interfaces to 100-Mbps Ethernet links could a switch based on this workstation handle?

28. Suppose a switch is built using a computer workstation and that it can forward packets at a rate of 500,000 packets per second, regardless (within limits) of size. Assume the workstation uses direct memory access (DMA) to move data in and out of its main memory, which has a bandwidth of 2 Gbps, and that the I/O bus has a bandwidth of 1 Gbps. At what packet size would the bus bandwidth become the limiting factor?

29. Suppose that a switch is designed to have both input and output FIFO buffering. As packets arrive on an input port they are inserted

at the tail of the FIFO. The switch then tries to forward the packets at the head of each FIFO to the tail of the appropriate output FIFO.

(a) Explain under what circumstances such a switch can lose a packet destined for an output port whose FIFO is empty.

(b) What is this behavior called?

(c) Assume that the FIFO buffering memory can be redistributed freely. Suggest a reshuffling of the buffers that avoids the above problem, and explain why it does so.

☆ **30.** A stage of an $n \times n$ banyan network consists of $(n/2)$ 2×2 switching elements. The first stage directs packets to the correct half of the network, the next stage to the correct quarter, and so on, until the packet is routed to the correct output. Derive an expression for the number of 2×2 switching elements needed to make an $n \times n$ banyan network. Verify your answer for $n = 8$.

☆ **31.** Describe how a Batcher network works. (See the Further Reading section.) Explain how a Batcher network can be used in combination with a banyan network to build a switching fabric.

32. Suppose a 10-Mbps Ethernet hub (repeater) is replaced by a 10-Mbps switch, in an environment where all traffic is between a single server and N "clients." Because all traffic must still traverse the server–switch link, nominally there is no improvement in bandwidth.

(a) Would you expect *any* improvement in bandwidth? If so, why?

(b) What other advantages and drawbacks might a switch offer versus a hub?

33. What aspect of IP addresses makes it necessary to have one address per network interface, rather than just one per host? In light of your answer, why does IP tolerate point-to-point interfaces that have nonunique addresses or no addresses?

34. Why does the Offset field in the IP header measure the offset in 8-byte units? (Hint: Recall that the Offset field is 13 bits long.)

35. Some signalling errors can cause entire ranges of bits in a packet to be overwritten by all 0s or all 1s. Suppose all the bits in the packet, including the Internet checksum, are overwritten. Could a packet with all 0s or all 1s be a legal IPv4 packet? Will the Internet checksum catch that error? Why or why not?

36. Suppose a TCP message that contains 1024 bytes of data and 20 bytes of TCP header is passed to IP for delivery across two networks interconnected by a router (i.e., it travels from the source host to a router to the destination host). The first network has an MTU of 1024 bytes; the second has an MTU of 576 bytes. Each network's MTU gives the size of the largest IP datagram that can be carried in a link-layer frame. Give the sizes and offsets of the sequence of fragments delivered to the network layer at the destination host. Assume all IP headers are 20 bytes.

37. Path MTU is the smallest MTU of any link on the current path (route) between two hosts. Assume we could discover the path MTU of the path used in the previous exercise, and that we use this value as the MTU for all the path segments. Give the sizes and offsets of the sequence of fragments delivered to the network layer at the destination host.

38. Suppose an IP packet is fragmented into 10 fragments, each with a 1% (independent) probability of loss. To a reasonable approximation, this means there is a 10% chance of losing the whole packet due to loss of a fragment. What is the probability of net loss of the whole packet if the packet is transmitted twice,

(a) Assuming all fragments received must have been part of the same transmission?

(b) Assuming any given fragment may have been part of either transmission?

(c) Explain how use of the Ident field might be applicable here.

39. Suppose the fragments of Figure 3.18(b) all pass through another router onto a link with an MTU of 380 bytes, not counting the link header. Show the fragments produced. If the packet were originally fragmented for this MTU, how many fragments would be produced?

40. What is the maximum bandwidth at which an IP host can send 576-byte packets without having the Ident field wrap around within 60 seconds? Suppose that IP's maximum segment lifetime (MSL) is 60 seconds; that is, delayed packets can arrive up to 60 seconds late but no later. What might happen if this bandwidth were exceeded?

41. Why do you think IPv4 has fragment reassembly done at the endpoint, rather than at the next router? Why do you think IPv6 abandoned fragmentation entirely? (Hint: Think about the differences between IP-layer fragmentation and link-layer fragmentation).

42. Having ARP table entries time out after 10 to 15 minutes is an attempt at a reasonable compromise. Describe the problems that can occur if the timeout value is too small or too large.

43. IP currently uses 32-bit addresses. If we could redesign IP to use the 6-byte MAC address instead of the 32-bit address, would we be able to eliminate the need for ARP? Explain why or why not.

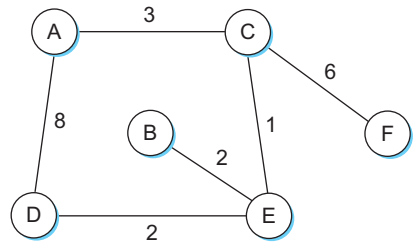
44. Suppose hosts A and B have been assigned the same IP address on the same Ethernet, on which ARP is used. B starts up after A. What will happen to A's existing connections? Explain how "self-ARP" (querying the network on start-up for one's own IP address) might help with this problem.

45. Suppose an IP implementation adheres literally to the following algorithm on receipt of a packet, P, destined for IP address D:

```
if (⟨Ethernet address for D is in ARP cache⟩)
    ⟨send P⟩
else
    ⟨send out an ARP query for D⟩
    ⟨put P into a queue until the response comes back⟩
```

    (a) If the IP layer receives a burst of packets destined for D, how might this algorithm waste resources unnecessarily?
    (b) Sketch an improved version.
    (c) Suppose we simply drop P, after sending out a query, when cache lookup fails. How would this behave? (Some early ARP implementations allegedly did this.)

46. For the network shown in Figure 3.53, give global distance–vector tables like those of Tables 3.10 and 3.13 when
    (a) Each node knows only the distances to its immediate neighbors.
    (b) Each node has reported the information it had in the preceding step to its immediate neighbors.
    (c) Step (b) happens a second time.



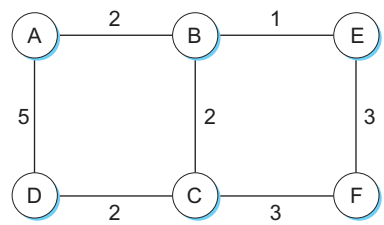
■ **FIGURE 3.53** Network for Exercises 46, 48, and 54.

47. For the network given in Figure 3.54, give global distance–vector tables like those of Tables 3.10 and 3.13 when

(a) Each node knows only the distances to its immediate neighbors.

(b) Each node has reported the information it had in the preceding step to its immediate neighbors.

(c) Step (b) happens a second time.



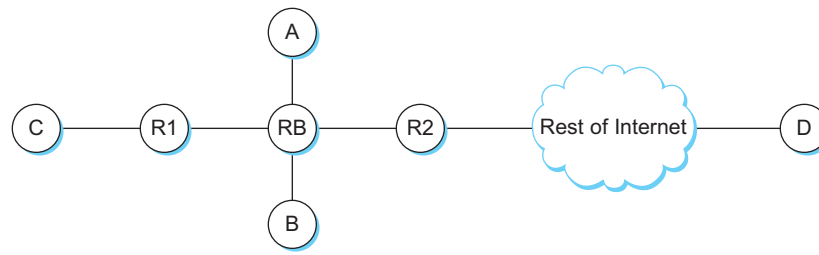
■ **FIGURE 3.54** Network for Exercise 47.

48. For the network given in Figure 3.53, show how the *link-state* algorithm builds the routing table for node D.

49. Use the Unix utility traceroute (Windows tracert) to determine how many hops it is from your host to other hosts in the Internet (e.g., cs.princeton.edu or www.cisco.com). How many routers do you traverse just to get out of your local site? Read the man page or other documentation for traceroute and explain how it is implemented.

50. What will happen if traceroute is used to find the path to an unassigned address? Does it matter if the network portion or only the host portion is unassigned?

51. A site is shown in Figure 3.55. R1 and R2 are routers; R2 connects to the outside world. Individual LANs are Ethernets. RB is a *bridge-router*; it routes traffic addressed to it and acts as a bridge for other traffic. Subnetting is used inside the site; ARP is used on each subnet. Unfortunately, host A has been misconfigured and doesn't use subnets. Which of B, C, and D can A reach?



■ **FIGURE 3.55** Site for Exercise 51.

52. Suppose we have the forwarding tables shown in Table 3.16 for nodes A and F, in a network where all links have cost 1. Give a diagram of the smallest network consistent with these tables.

Table 3.16 Forwarding Tables for Exercise 52

A

Node	Cost	Nexthop
B	1	B
C	2	B
D	1	D
E	2	B
F	3	D

F

Node	Cost	Nexthop
A	3	E
B	2	C
C	1	C
D	2	E
E	1	E

53. Suppose we have the forwarding tables shown in Table 3.17 for nodes A and F, in a network where all links have cost 1. Give a diagram of the smallest network consistent with these tables.

Table 3.17 Forwarding Tables for Exercise 53

A

Node	Cost	Nexthop
B	1	B
C	1	C
D	2	B
E	3	C
F	2	C

F

Node	Cost	Nexthop
A	2	C
B	3	C
C	1	C
D	2	C
E	1	E

54. For the network in Figure 3.53, suppose the forwarding tables are all established as in Exercise 46 and then the C–E link fails. Give:
    (a) The tables of A, B, D, and F after C and E have reported the news.
    (b) The tables of A and D after their next mutual exchange.
    (c) The table of C after A exchanges with it.

55. Suppose a router has built up the routing table shown in Table 3.18. The router can deliver packets directly over interfaces 0 and 1, or it can forward packets to routers R2, R3, or R4. Describe what the router does with a packet addressed to each of the following destinations:
    (a) 128.96.39.10
    (b) 128.96.40.12
    (c) 128.96.40.151
    (d) 192.4.153.17
    (e) 192.4.153.90

Table 3.18 Routing Table for Exercise 55

SubnetNumber	SubnetMask	NextHop
128.96.39.0	255.255.255.128	Interface 0
128.96.39.128	255.255.255.128	Interface 1
128.96.40.0	255.255.255.128	R2
192.4.153.0	255.255.255.192	R3
⟨default⟩		R4

✓ **56.** Suppose a router has built up the routing table shown in Table 3.19. The router can deliver packets directly over interfaces 0 and 1, or it can forward packets to routers R2, R3, or R4. Assume the router does the longest prefix match. Describe what the router does with a packet addressed to each of the following destinations:

(a) 128.96.171.92

(b) 128.96.167.151

(c) 128.96.163.151

(d) 128.96.169.192

(e) 128.96.165.121

Table 3.19 Routing Table for Exercise 56

SubnetNumber	SubnetMask	NextHop
128.96.170.0	255.255.254.0	Interface 0
128.96.168.0	255.255.254.0	Interface 1
128.96.166.0	255.255.254.0	R2
128.96.164.0	255.255.252.0	R3
⟨default⟩		R4

☆ **57.** Consider the simple network in Figure 3.56, in which A and B exchange distance-vector routing information. All links have cost 1. Suppose the A–E link fails.

■ **FIGURE 3.56** Simple network for Exercise 57.

(a) Give a sequence of routing table updates that leads to a routing loop between A and B.

(b) Estimate the probability of the scenario in (a), assuming A and B send out routing updates at random times, each at the same average rate.

(c) Estimate the probability of a loop forming if A broadcasts an updated report within 1 second of discovering the A–E failure, and B broadcasts every 60 seconds uniformly.

58. Consider the situation involving the creation of a routing loop in the network of Figure 3.29 when the A–E link goes down. List *all* sequences of table updates among A, B, and C, pertaining to destination E, that lead to the loop. Assume that table updates are done one at a time, that the split-horizon technique is observed by all participants, and that A sends its initial report of E's unreachability to B before C. You may ignore updates that don't result in changes.

59. Suppose a set of routers all use the split-horizon technique; we consider here under what circumstances it makes a difference if they use poison reverse in addition.

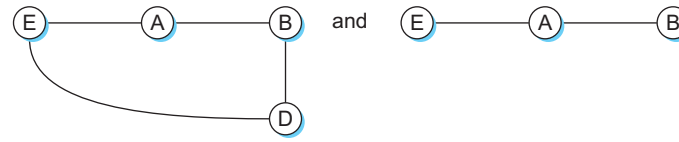
(a) Show that poison reverse makes no difference in the evolution of the routing loop in the two examples described in Section 3.3.2, given that the hosts involved use split horizon.

(b) Suppose split-horizon routers A and B somehow reach a state in which they forward traffic for a given destination X toward each other. Describe how this situation will evolve with and without the use of poison reverse.

(c) Give a sequence of events that leads A and B to a looped state as in (b), even if poison reverse is used. (Hint: Suppose B and A connect through a very slow link. They each reach X through a third node, C, and simultaneously advertise their routes to each other.)

60. *Hold down* is another distance–vector loop-avoidance technique, whereby hosts ignore updates for a period of time until link failure news has had a chance to propagate. Consider the networks in Figure 3.57, where all links have cost 1 except E–D, with cost 10. Suppose that the E–A link breaks and B reports its loop-forming E route to A immediately afterwards (this is the false route, via A).

Specify the details of a hold-down interpretation, and use this to describe the evolution of the routing loop in both networks. To what extent can hold down prevent the loop in the EAB network without delaying the discovery of the alternative route in the EABD network?



■ **FIGURE 3.57** Networks for Exercise 60.

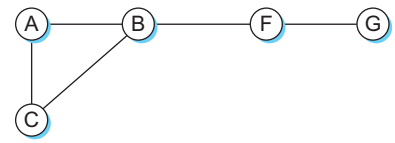
61. Consider the network in Figure 3.58, using link-state routing. Suppose the B–F link fails, and the following then occur in sequence:

(a) Node H is added to the right side with a connection to G.

(b) Node D is added to the left side with a connection to C.

(c) A new link, D–A, is added.

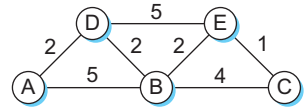
The failed B–F link is now restored. Describe what link-state packets will flood back and forth. Assume that the initial sequence number at all nodes is 1, that no packets time out, and that both ends of a link use the same sequence number in their LSP for that link, greater than any sequence number used before.



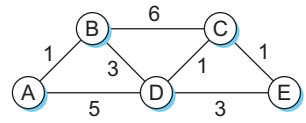
■ **FIGURE 3.58** Network for Exercise 61.

62. Give the steps as in Table 3.14 in the forward search algorithm as it builds the routing database for node A in the network shown in Figure 3.59.

63. Give the steps as in Table 3.14 in the forward search algorithm as it builds the routing database for node A in the network shown in Figure 3.60.



■ **FIGURE 3.59** Network for Exercise 62.



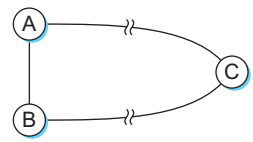
■ **FIGURE 3.60** Network for Exercise 63.

64. Suppose that nodes in the network shown in Figure 3.61 participate in link-state routing, and C receives contradictory LSPs: One from A arrives claiming the A–B link is down, but one from B arrives claiming the A–B link is up.

(a) How could this happen?

(b) What should C do? What can C expect?

Do not assume that the LSPs contain any synchronized timestamp.



■ **FIGURE 3.61** Network for Exercise 64.

65. Suppose IP routers learned about IP networks and subnets the way Ethernet learning bridges learn about hosts: by noting the appearance of new ones and the interface by which they arrive. Compare this with existing distance–vector router learning

(a) For a leaf site with a single attachment to the Internet.

(b) For internal use at an organization that did not connect to the Internet.

Assume that routers only receive new-network notices from other routers and that the originating routers receive their IP network information via configuration.

66. IP hosts that are not designated routers are *required* to drop packets misaddressed to them, even if they would otherwise be able to forward them correctly. In the absence of this requirement, what would happen if a packet addressed to IP address A were inadvertently broadcast at the link layer? What other justifications for this requirement can you think of?

67. Read the man page or other documentation for the Unix/Windows utility netstat. Use netstat to display the current IP routing table on your host. Explain the purpose of each entry. What is the practical minimum number of entries?

68. An organization has been assigned the prefix 212.1.1/24 (class C) and wants to form subnets for four departments, with hosts as follows:

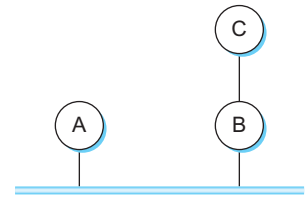
A	75 hosts
B	35 hosts
C	20 hosts
D	18 hosts

There are 148 hosts in all.

(a) Give a possible arrangement of subnet masks to make this possible.

(b) Suggest what the organization might do if department D grows to 32 hosts.

69. Suppose hosts A and B are on an Ethernet LAN with IP network address 200.0.0/24. It is desired to attach a host C to the network via a direct connection to B (see Figure 3.62). Explain how to do this with subnets; give sample subnet assignments. Assume that



■ **FIGURE 3.62** Network for Exercise 69.

an additional network prefix is not available. What does this do to the size of the Ethernet LAN?

70. An alternative method for connecting host C in Exercise 69 is to use *proxy ARP* and routing: B agrees to route traffic to and from C and also answers ARP queries for C received over the Ethernet.

(a) Give all packets sent, with physical addresses, as A uses ARP to locate and then send one packet to C.

(b) Give B's routing table. What peculiarity must it contain?

71. Suppose two subnets share the same physical LAN; hosts on each subnet will see the other subnet's broadcast packets.

(a) How will DHCP fare if two servers, one for each subnet, coexist on the shared LAN? What problems might [*do!*] arise?

(b) Will ARP be affected by such sharing?

72. Table 3.20 is a routing table using CIDR. Address bytes are in hexadecimal. The notation "/12" in C4.50.0.0/12 denotes a netmask with 12 leading 1 bits: FF.F0.0.0. Note that the last three entries cover every address and thus serve in lieu of a default route. State to what next hop the following will be delivered:

(a) C4.5E.13.87

(b) C4.5E.22.09

(c) C3.41.80.02

(d) 5E.43.91.12

(e) C4.6D.31.2E

(f) C4.6B.31.2E

Table 3.20 Routing Table for Exercise 72

Net/MaskLength	Nexthop
C4.50.0.0/12	A
C4.5E.10.0/20	B
C4.60.0.0/12	C
C4.68.0.0/14	D
80.0.0.0/1	E
40.0.0.0/2	F
00.0.0.0/2	G

73. Table 3.21 is a routing table using CIDR. Address bytes are in hexadecimal. The notation "/12" in C4.50.0.0/12 denotes a netmask with 12 leading 1 bits: FF.F0.0.0. State to what next hop the following will be delivered:

(a) C4.4B.31.2E

(b) C4.5E.05.09

(c) C4.4D.31.2E

(d) C4.5E.03.87

(e) C4.5E.7F.12

(f) C4.5E.D1.02

Table 3.21 Routing Table for Exercise 73

Net/MaskLength	Nexthop
C4.5E.2.0/23	A
C4.5E.4.0/22	B
C4.5E.C0.0/19	C
C4.5E.40.0/18	D
C4.4C.0.0/14	E
C0.0.0.0/2	F
80.0.0.0/1	G

74. An ISP that has authority to assign addresses from a /16 prefix (an old class B address) is working with a new company to allocate it a portion of address space based on CIDR. The new company needs IP addresses for machines in three divisions of its corporate network: Engineering, Marketing, and Sales. These divisions plan to grow as follows: Engineering has 5 machines as of the start of year 1 and intends to add 1 machine every week, Marketing will never need more than 16 machines, and Sales needs 1 machine for every 2 clients. As of the start of year 1, the company has no clients, but the sales model indicates that, by the start of year 2, the company will have 6 clients and each week thereafter will get one new client with probability 60%, will lose one client with probability 20%, or will maintain the same number with probability 20%.

(a) What address range would be required to support the company's growth plans for at least 7 years if Marketing uses all 16 of its addresses and the Sales and Engineering plans behave as expected?

(b) How long would this address assignment last? At the time when the company runs out of address space, how would the addresses be assigned to the three groups?

(c) If, instead of using CIDR addressing, it was necessary to use old-style classful addresses, what options would the new company have in terms of getting address space?

75. Propose a lookup algorithm for an IP forwarding table containing prefixes of varying lengths that does not require a linear search of the entire table to find the longest match.