Foundation

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

—William Blake

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

PROBLEM
Building a Network

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

What distinguishes a computer network from these other types of networks? Probably the most important characteristic of a computer network is its generality. Computer networks are built primarily from general-purpose programmable hardware, and they are not optimized for a particular application like making phone calls or delivering television signals. Instead, they are able to carry many different types of data, and they support a wide, and ever-growing, range of applications. This chapter looks

at some typical applications of computer networks and discusses the requirements that a network designer who wishes to support such applications must be aware of.

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

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

1.1 Applications

Most people know the Internet through its applications: the World Wide Web, email, streaming audio and video, chat rooms, and music (file) sharing. The Web, for example, presents an intuitively simple interface. Users view pages full of textual and graphical objects, click on objects that they want to learn more about, and a corresponding new page appears. Most people are also aware that just under the covers, each selectable object on a page is bound to an identifier for the next page to be viewed. This identifier, called a Uniform Resource Locator (URL), is used to provide a way of identifying all the possible pages that can be viewed from your web browser. For example,

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

is the URL for a page providing information about one of this book's authors: the string http indicates that the HyperText Transfer Protocol (HTTP) should be used to download the page, www.cs.princeton.edu is the name of the machine that serves the page, and

/~llp/index.html

uniquely identifies Larry's home page at this site.

What most Web users are not aware of, however, is that by clicking on just one such URL, as many as 17 messages may be exchanged over the Internet, and this assumes the page itself is small enough to fit in a single message. This number includes up to six messages to translate the server name (www.cs.princeton.edu) into its Internet address (128.112.136.35), three messages to set up a Transmission Control Protocol (TCP) connection between your browser and this server, four messages for your browser to send the HTTP "get" request and the server to respond with the requested page (and for each side to acknowledge receipt of that message), and four messages to tear down the TCP connection. Of course, this does not include the millions of messages exchanged by Internet nodes throughout the day, just to let each other know that they exist and are ready to serve web pages, translate names to addresses, and forward messages toward their ultimate destination.

Another widespread application of the Internet is the delivery of "streaming" audio and video. While an entire video file could first be fetched from a remote machine and then played on the local machine, similar to the process of downloading and displaying a web page, this would entail waiting for the last second of the video file to be delivered before starting to look at it. Streaming video implies that the sender and the receiver are, respectively, the source and the sink for the video stream. That is, the source generates a video stream (perhaps using a video capture card), sends it across the Internet in messages, and the sink displays the stream as it arrives.

There are a variety of different classes of video applications. One class of video application is video-on-demand, which reads a preexisting movie from disk and transmits it over the network. Another kind of application is videoconferencing, which is in some ways the more challenging (and, for networking people, interesting) case because it has very tight timing constraints. Just as when using the telephone, the interactions among the participants must be timely. When a person at one end gestures, then that action must be displayed at the other end as quickly as possible. Too much delay makes the system unusable. Contrast this with video-on-demand where, if it takes several seconds from the time the user starts the video until the first image is displayed, the service is still deemed satisfactory. Also, interactive video usually implies that video is flowing in both directions, while a video-on-demand application is most likely sending video in only one direction.

One pioneering example of a videoconferencing tool, developed in the early and mid-1990s, is vic. Figure 1.1 shows the control panel for a vic session. vic is actually

Figure 1.1 The vic video application. This shot is from a 1995 release of the tool.

one of a suite of conferencing tools designed at Lawrence Berkeley Laboratory and UC Berkeley. The others include a whiteboard application (wb) that allows users to send sketches and slides to each other, a visual audio tool called vat, and a session directory (sdr) that is used to create and advertise videoconferences. All these tools run on Unix—hence their lowercase names—and are freely available on the Internet. Many similar tools are available for other operating systems. It is interesting to note that while video over the Internet is still considered to be in its relative infancy at the time of this writing (2006), that the tools to support video over IP have existed for well over a decade.

Although they are just two examples, downloading pages from the Web and participating in a videoconference demonstrate the diversity of applications that can be built on top of the Internet, and hint at the complexity of the Internet's design. Starting from the beginning, and addressing one problem at time, the rest of this book explains how to build a network that supports such a wide range of applications. Chapter 9 concludes the book by revisiting these two specific applications, as well as several others that have become popular on today's Internet.

1.2 Requirements

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

The first step is to identify the set of constraints and requirements that influence network design. Before getting started, however, it is important to understand that the expectations you have of a network depend on your perspective:

- An *application programmer* would list the services that his application needs, for example, a guarantee that each message the application sends will be delivered without error within a certain amount of time.
- A *network designer* would list the properties of a cost-effective design, for example, that network resources are efficiently utilized and fairly allocated to different users.

- A *network provider* would list the characteristics of a system that is easy to administer and manage, for example, in which faults can be easily isolated and where it is easy to account for usage.

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

1.2.1 Connectivity

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

Links, Nodes, and Clouds

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

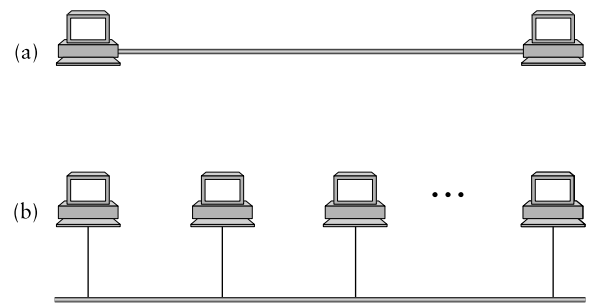

Figure 1.2 Direct links: (a) point-to-point; (b) multiple-access.

a given link supports point-to-point or multiple-access connectivity depends on how the node is attached to the link. It is also the case that multiple-access links are often limited in size, in terms of both the geographical distance they can cover and the number of nodes they can connect.

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

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

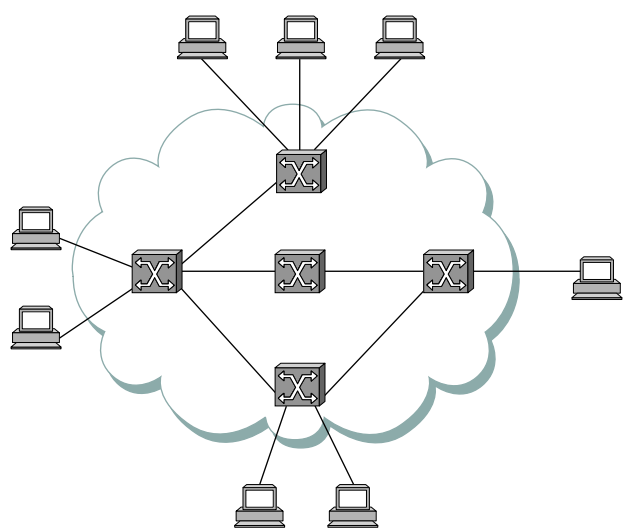

Figure 1.3 Switched network.

discrete blocks of data to each other. Think of these blocks of data as corresponding to some piece of application data such as a file, a piece of email, or an image. We call each block of data either a *packet* or a *message*, and for now we use these terms interchangeably; we discuss the reason they are not always the same in Section 1.2.2.

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

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

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

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

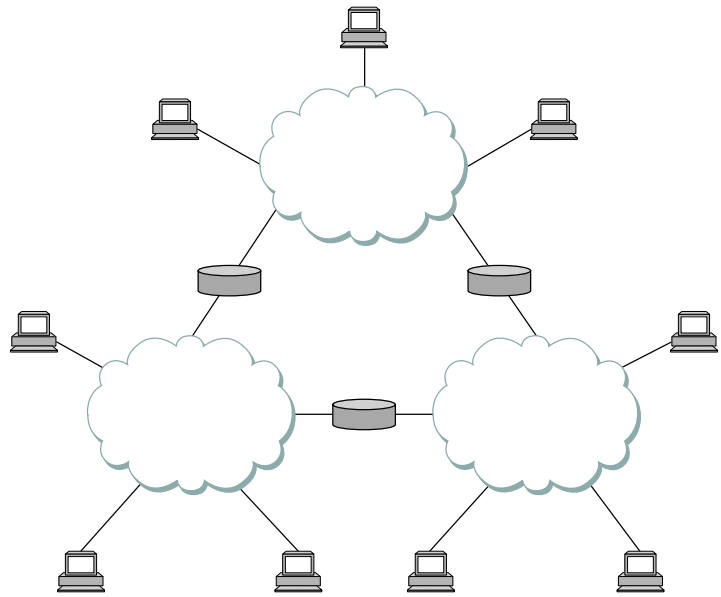

Figure 1.4 Interconnection of networks.

connected, then the switches and routers of the network use this address to decide how to forward the message toward the destination. The process of determining systematically how to forward messages toward the destination node based on its address is called *routing*.

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

The main idea to take away from this discussion is that we can define a *network* recursively as consisting of two or more nodes connected by a physical link, or as two or more networks connected by a node. In other words, a network can be constructed from a nesting of networks, where at the bottom level, the network is implemented by some physical medium. One of the key challenges in providing network connectivity is to define an address for each node that is reachable on the network (including support for broadcast and multicast connectivity), and to be able to use this address to route messages toward the appropriate destination node(s).

1.2.2 Cost-Effective Resource Sharing

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

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

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

To see how this might work, consider the simple network illustrated in Figure 1.5, where the three hosts on the left side of the network (senders S1–S3) are sending data to the three hosts on the right (receivers R1–R3) by sharing a switched network that contains only one physical link. (For simplicity, assume that host S1 is sending data to host R1, and so on.) In this situation, three flows of data—corresponding to the three pairs of hosts—are multiplexed onto a single physical link by switch 1 and then *demultiplexed* back into separate flows by switch 2. Note that we are being intentionally vague about

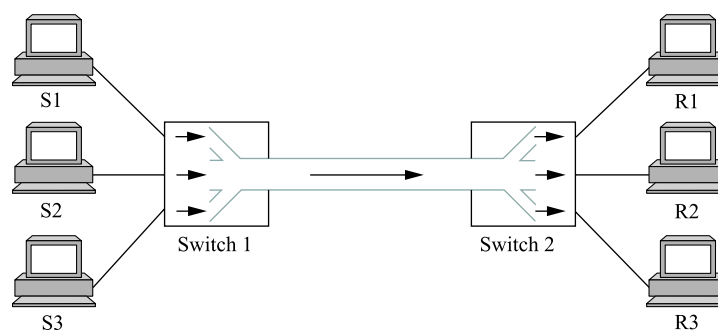

Figure 1.5 Multiplexing multiple logical flows over a single physical link.

exactly what a "flow of data" corresponds to. For the purposes of this discussion, assume that each host on the left has a large supply of data that it wants to send to its counterpart on the right.

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

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

The form of multiplexing that we make most use of in this book is called *statistical multiplexing*. Although the name is not all that helpful for understanding the concept, statistical multiplexing is really quite simple, with two key ideas. First, it is like STDM in that the physical link is shared over time—first data from one flow is transmitted over the physical link, then data from another flow is transmitted, and so on. Unlike STDM, however, data is transmitted from each flow on demand rather than during a predetermined time slot. Thus, if only one flow has data to send, it gets to transmit that data without waiting for its quantum to come around and thus without having to watch the quanta assigned to the other flows go by unused. It is this avoidance of idle time that gives packet switching its efficiency.

As defined so far, however, statistical multiplexing has no mechanism to ensure that all the flows eventually get their turn to transmit over the physical link. That is, once a flow begins sending data, we need some way to limit the transmission, so that the other flows can have a turn. To account for this need, statistical multiplexing defines an upper bound on the size of the block of data that each flow is permitted to transmit at a given

time. This limited-size block of data is typically referred to as a *packet*, to distinguish it from the arbitrarily large *message* that an application program might want to transmit. Because a packet-switched network limits the maximum size of packets, a host may not be able to send a complete message in one packet. The source may need to fragment the message into several packets, with the receiver reassembling the packets back into the original message.

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

The decision as to which packet to send next on a shared link can be made in a number of different ways. For example, in a network consisting of switches interconnected by links such as the one in Figure 1.5, the decision would be made by the switch that transmits packets onto the shared link. (As we will see later, not all packet-switched networks actually involve switches, and they may use other mechanisms to determine whose packet goes onto the link next.) Each switch in a packet-switched network makes this decision independently, on a packet-by-packet basis. One of the issues that faces a network designer is how to make this decision in a fair manner. For example, a switch could be designed to service packets on a first-in-first-out (FIFO) basis. Another approach would be to transmit the packets from each of the different flows that are currently sending data through the switch in a round-robin manner. This might be done to

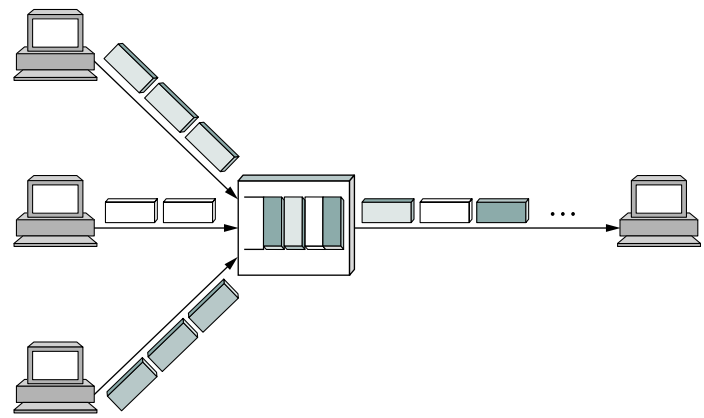

Figure 1.6 A switch multiplexing packets from multiple sources onto one shared link.

ensure that certain flows receive a particular share of the link's bandwidth, or that they never have their packets delayed in the switch for more than a certain length of time. A network that attempts to allocate bandwidth to particular flows is sometimes said to support *quality of service (QoS)*, a topic that we return to in Chapter 6.

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

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

1.2.3 Support for Common Services

While the previous section outlined the challenges involved in providing cost-effective connectivity among a group of hosts, it is overly simplistic to view a computer network as simply delivering packets among a collection of computers. It is more accurate to think of a network as providing the means for a set of application processes that are distributed over those computers to communicate. In other words, the next requirement of a computer network is that the application programs running on the hosts connected to the network must be able to communicate in a meaningful way.

When two application programs need to communicate with each other,

SANs, LANs, MANs, and WANs

One way to characterize networks is according to their size. Two well-known examples are local area networks (LANs) and wide area networks (WANs); the former typically extend less than 1 km, while the latter can be worldwide. Other networks are classified as metropolitan area networks (MANs), which usually span tens of kilometers. The reason such classifications are interesting is that the size of a network often has implications for the underlying technology that can be used, with a key factor being the amount of time it takes for data to

propagate from one end of the network to the other; we discuss this issue more in later chapters.

An interesting historical note is that the term "wide area network" was not applied to the first WANs because there was no other sort of network to differentiate them from. When computers were incredibly rare and expensive, there was no point in thinking about how to connect all the computers in the local area—there was only one computer in that area. Only as computers began to proliferate did LANs become necessary, and the term "WAN" was then introduced to describe the larger networks that interconnected geographically distant computers.

Another kind of network that we need to be aware of is a storage area network (SAN). SANs are usually confined to a single room and connect the various components of a large computing system, such as disk arrays and servers. For example, High Performance Parallel Interface (HiPPI) and Fiber Channel are two common SAN technologies used to connect massively parallel processors to scalable storage servers and data vaults. Although this book does not describe such networks in detail, they are worth knowing about because they are often at the leading edge in terms of performance, and because it is increasingly common to connect such networks into LANs and WANs.

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

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

The challenge is to recognize what functionality the channels should provide to application programs. For example,

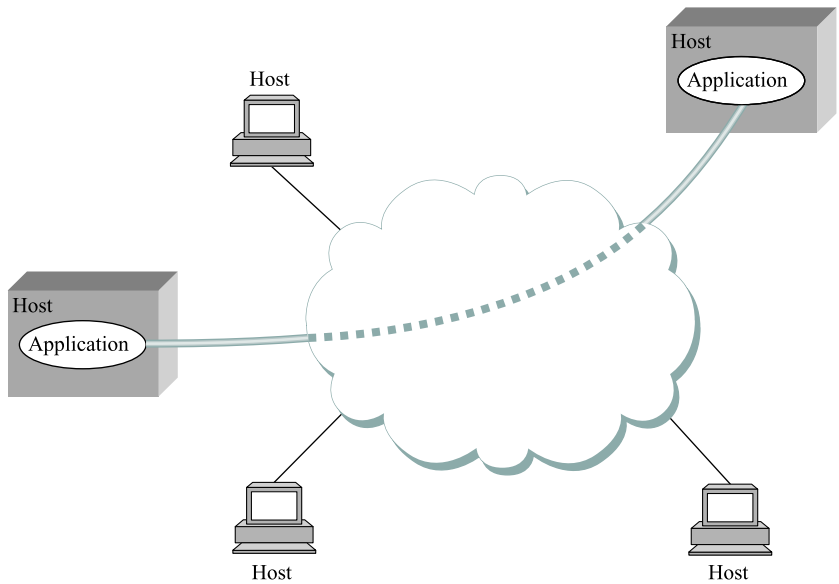

Figure 1.7 Processes communicating over an abstract channel.

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

Identifying Common Communication Patterns

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

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

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

Reading a file involves the client sending a small request message to a server and the server responding with a large message that contains the data in the file. Writing works in the opposite way—the client sends a large message containing the data to be written to the server, and the server responds with a small message confirming that the write to disk has taken place. A digital library, as exemplified by the World Wide Web, is another application that behaves in a similar way: a client process makes a request, and a server process responds by returning the requested data.

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

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

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

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

Reliability

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

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

The second class of failure is at the packet, rather than the bit, level; that is, a complete packet is lost by the network. One reason this can happen is that the packet contains an uncorrectable bit error and therefore has to be discarded. A more likely

reason, however, is that one of the nodes that has to handle the packet—for example, a switch that is forwarding it from one link to another—is so overloaded that it has no place to store the packet, and therefore is forced to drop it. This is the problem of congestion mentioned in Section 1.2.2. Less commonly, the software running on one of the nodes that handles the packet makes a mistake. For example, it might incorrectly forward a packet out on the wrong link, so that the packet never finds its way to the ultimate destination. As we will see, one of the main difficulties in dealing with lost packets is distinguishing between a packet that is indeed lost and one that is merely late in arriving at the destination.

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

The key idea to take away from this discussion is that defining useful channels involves both understanding the applications' requirements and recognizing the limitations of the underlying technology. The challenge is to fill in the gap between what the application expects and what the underlying technology can provide. This is sometimes called the *semantic gap.*

1.3 Network Architecture

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

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

It also introduces two of the most widely referenced architectures—the OSI architecture and the Internet architecture.

1.3.1 Layering and Protocols

When a system gets complex, the system designer introduces another level of abstraction. The idea of an abstraction is to define a unifying model that can capture some important aspect of the system, encapsulate this model in an object that provides an interface that can be manipulated by other components of the system, and hide the details of how the object is implemented from the users of the object. The challenge is to identify abstractions that simultaneously provide a service that proves useful in a large number of situations and that can be efficiently implemented in the underlying system. This is exactly what we were doing when we introduced the idea of a channel in the previous section: We were providing an abstraction for applications that hides the complexity of the network from application writers.

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

Layering provides two nice features. First, it decomposes the problem of building a network into more manageable components. Rather than implementing a monolithic piece of software that does everything you will ever want, you can implement several

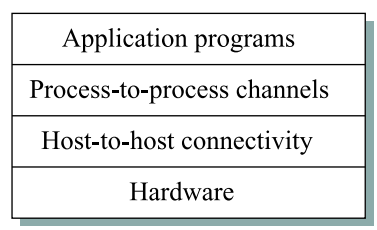

Figure 1.8 Example of a layered network system.

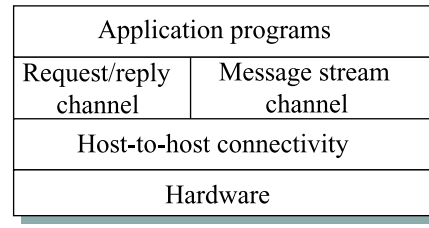

Figure 1.9 Layered system with alternative abstractions available at a given layer.

layers, each of which solves one part of the problem. Second, it provides a more modular design. If you decide that you want to add some new service, you may only need to modify the functionality at one layer, reusing the functions provided at all the other layers.

Thinking of a system as a linear sequence of layers is an oversimplification, however. Many times there are multiple abstractions provided at any given level of the system, each providing a different service to the higher layers but building on the same low-level abstractions. To see this, consider the two types of channels discussed in Section 1.2.3: One provides a request/reply service and one supports a message stream service. These two channels might be alternative offerings at some level of a multilevel networking system, as illustrated in Figure 1.9.

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

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

Second, a protocol defines a *peer interface* to its counterpart (peer) on another machine. This second interface defines the form and meaning of messages exchanged between protocol peers to implement the communication service. This would determine

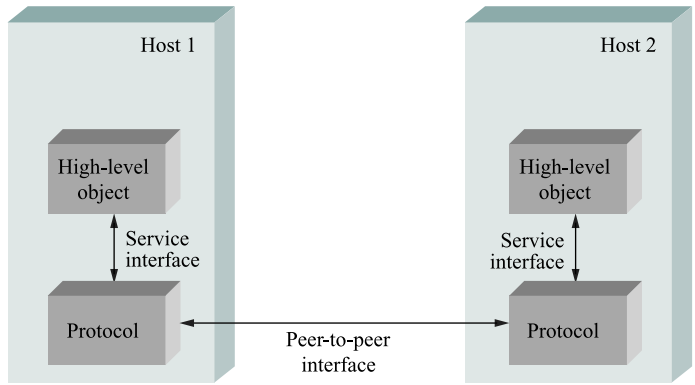

Figure 1.10 Service and peer interfaces.

the way in which a request/reply protocol on one machine communicates with its peer on another machine. In the case of HTTP, for example, the protocol specification defines in detail how a "GET" command is formatted, what arguments can be used with the command, and how a web server should respond when it receives such a command. (We will look more closely at this particular protocol in Section 9.1.2.)

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

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

In this example, suppose that the file access program on host 1 wants to send a message to its peer on host 2 using the communication service offered by protocol RRP. In this case, the file application asks RRP to send the message on its behalf. To communicate with its peer, RRP then invokes the services of HHP, which in turn transmits the message to its peer on the other machine. Once the message has arrived at protocol HHP

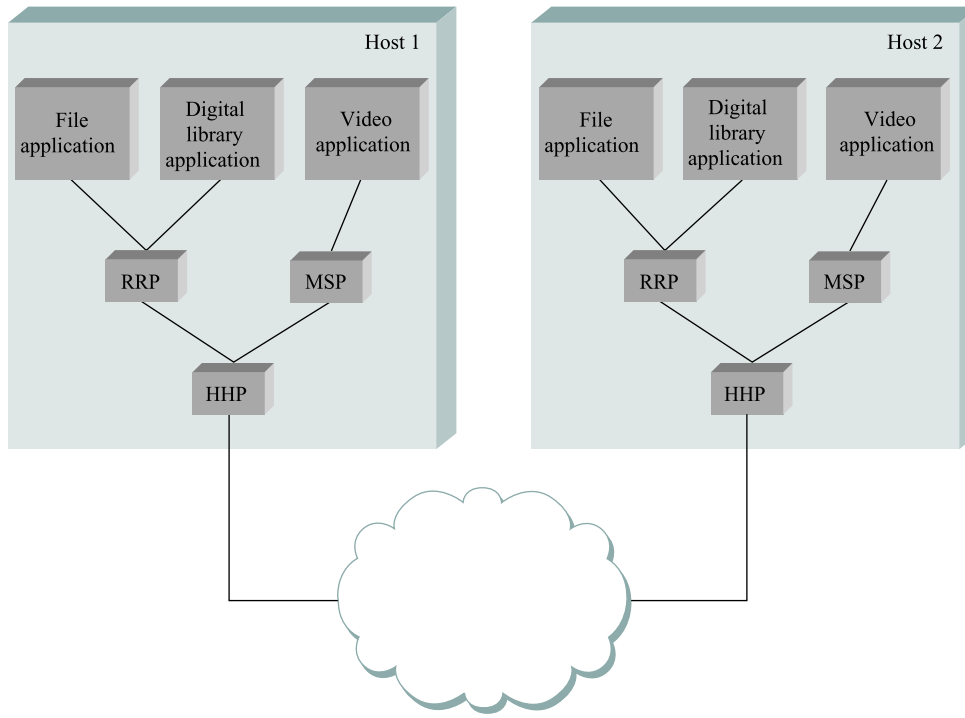

Figure 1.11 Example of a protocol graph.

on host 2, HHP passes the message up to RRP, which in turn delivers the message to the file application. In this particular case, the application is said to employ the services of the *protocol stack* RRP/HHP.

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

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

Encapsulation

Consider what happens in Figure 1.11 when one of the application programs sends a message to its peer by passing the message to protocol RRP. From RRP's perspective, the message it is given by the application is an uninterpreted string of bytes. RRP does not care that these bytes represent an array of integers, an email message, a digital image, or whatever; it is simply charged with sending them to its peer. However, RRP must communicate control information to its peer, instructing it how to handle the message when it is received. RRP does this by attaching a *header* to the message. Generally speaking, a header is a small data structure—from a few bytes to a few dozen bytes—that is used among peers to communicate with each other. As the name suggests, headers are usually attached to the front of a message. In some cases, however, this peer-to-peer control information is sent at the end of the message, in which case it is called a *trailer*. The exact format for the header attached by RRP is defined by its protocol specification. The rest of the message—that is, the data being transmitted on behalf of the application—is called the message's *body* or *payload*. We say that the application's data is *encapsulated* in the new message created by protocol RRP.

This process of encapsulation is then repeated at each level of the protocol graph; for example, HHP encapsulates RRP's message by attaching a header of its own. If we now assume that HHP sends the message to its peer over some network, then when the message arrives at the destination host, it is processed in the opposite order: HHP first interprets the HHP header at the front of the message (i.e., takes whatever action is appropriate given the contents of the header), and passes the body of the message (but not the HHP header) up to RRP, which takes whatever action is indicated by the RRP header that its peer attached, and passes the body of the message (but not the RRP header) up to the application program. The message passed up from RRP to the application on host 2 is exactly the same message as the application passed down to RRP on host 1; the application does not see any of the headers that have been attached to it to implement the lower-level communication services. This whole process is illustrated in

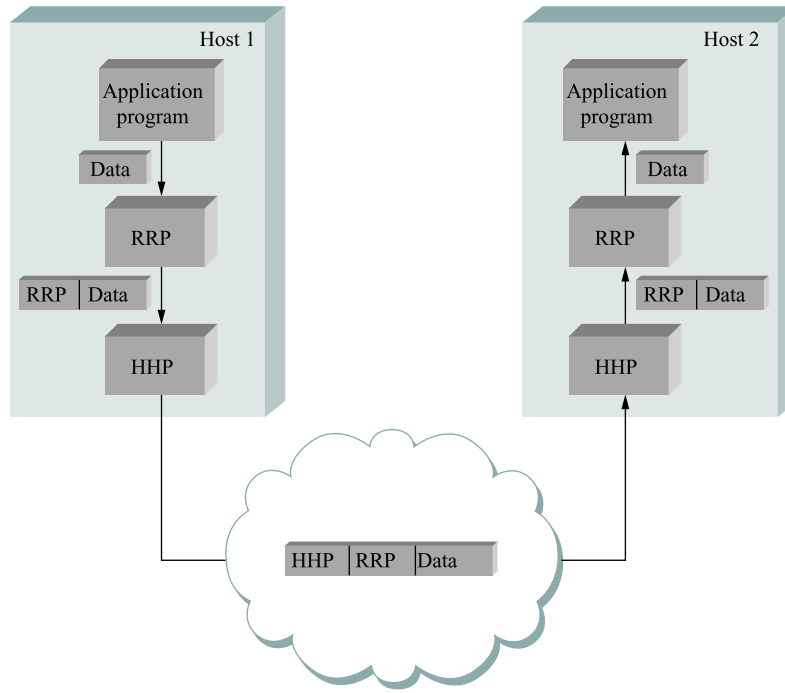

Figure 1.12 High-level messages are encapsulated inside of low-level messages.

Figure 1.12. Note that in this example, nodes in the network (e.g., switches and routers) may inspect the HHP header at the front of the message.

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

Multiplexing and Demultiplexing

Recall from Section 1.2.2 that a fundamental idea of packet switching is to multiplex multiple flows of data over a single physical link. This same idea applies up and down the protocol graph, not just to switching nodes. In Figure 1.11, for example, we can think of RRP as implementing a logical communication channel, with messages from

two different applications multiplexed over this channel at the source host and then demultiplexed back to the appropriate application at the destination host.

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

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

1.3.2 OSI Architecture

The ISO was one of the first organizations to formally define a common way to connect computers. Their architecture, called the *Open Systems Interconnection (OSI)* architecture and illustrated in Figure 1.13, defines a partitioning of network functionality into seven layers, where one or more protocols implement the functionality assigned to a given layer. In this sense, the schematic given in Figure 1.13 is not a protocol graph, per se, but rather a *reference model* for a protocol graph. The ISO, usually in conjunction with a second standards organization known as the International Telecommunications Union (ITU),[1] publishes a series of protocol specifications based on the OSI architecture. This series is sometimes called the "X dot" series since the protocols are given names like X.25, X.400, X.500, and so on.

Starting at the bottom and working up, the *physical* layer handles the transmission of raw bits over a communications link. The *data link* layer then collects a stream of bits into a larger aggregate called a *frame*. Network adaptors, along with device drivers running in the node's OS, typically implement the data link level. This means that frames, not raw bits, are actually delivered to hosts. The *network* layer handles routing among nodes within a packet-switched network. At this layer, the unit of data exchanged among nodes is typically called a *packet* rather than a frame, although they are fundamentally

[1] A subcommittee of the ITU on telecommunications (ITU-T) replaces an earlier subcommittee of the ITU, which was known by its French name, Comité Consultatif International de Télégraphique et Téléphonique (CCITT).

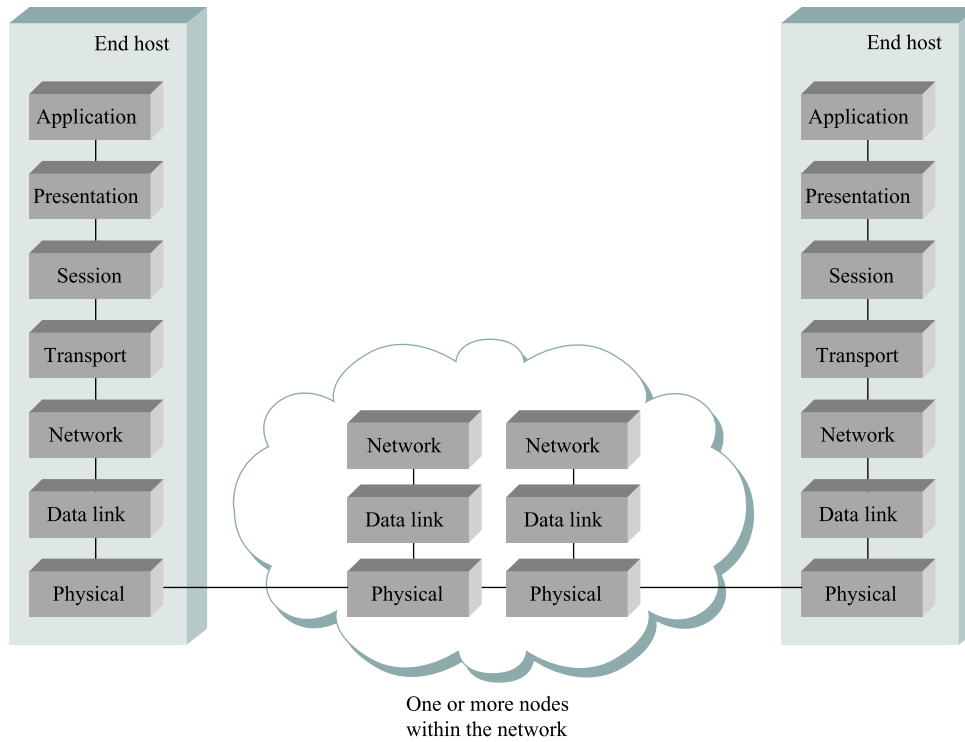

Figure 1.13 OSI network architecture.

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

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

application. For example, it might manage an audio stream and a video stream that are being combined in a teleconferencing application.

1.3.3 Internet Architecture

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

While the seven-layer OSI model can, with some imagination, be applied to the Internet, a four-layer model is often used instead. At the lowest level are a wide variety of network protocols, denoted NET_1, NET_2, and so on. In practice, these protocols are implemented by a combination of hardware (e.g., a network adaptor) and software (e.g., a network device driver). For example, you might find Ethernet or Fiber Distributed

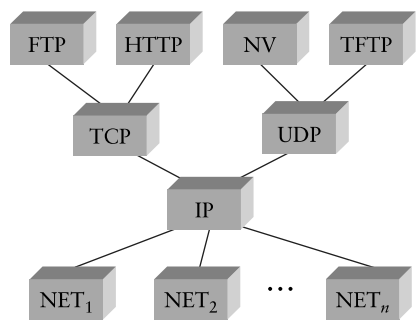

Figure 1.14 Internet protocol graph.

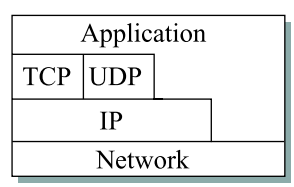

Figure 1.15 Alternative view of the Internet architecture. The "Network" layer shown here is sometimes referred to as the "subnetwork" or "link" layer.

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

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

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

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

A final attribute of the Internet architecture (or more accurately, of the IETF culture) is that in order for a new protocol to be officially included in the architecture, there

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

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

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

1.4 Implementing Network Software

Network architectures and protocol specifications are essential things, but a good blueprint is not enough to explain the phenomenal success of the Internet: The number of computers connected to the Internet has roughly doubled every 12 to 18 months since 1981, and is now estimated at 350 million; the number of people that use the Internet is estimated at 1 billion; and it is believed that the number of bits transmitted over the Internet, which has also grown exponentially, surpassed the corresponding figure for the voice phone system sometime in 2001.

What explains the success of the Internet? There are certainly many contributing factors (including a good architecture), but one thing that has made the Internet such a runaway success is the fact that so much of its functionality is provided by software running in general-purpose computers. The significance of this is that new functionality can be added readily with "just a small matter of programming." As a result, new applications and services—electronic commerce, videoconferencing, and packet telephony, to name a few—have been showing up at a phenomenal pace.

A related factor is the massive increase in computing power available in commodity machines. Although computer networks have always been capable in principle of transporting any kind of information, such as digital voice samples, digitized images, and so on, this potential was not particularly interesting if the computers sending and receiving that data were too slow to do anything useful with the information. Virtually all of today's computers are capable of playing back digitized voice at full speed and can display video at a speed and resolution that is useful for some (but by no means all) applications.

Thus, today's networks have begun to support multimedia, and their support for it will only improve as computing hardware becomes faster.

The point to take away from this is that knowing how to implement network software is an essential part of understanding computer networks. With this in mind, this section first introduces some of the issues involved in implementing an application program on top of a network, and then goes on to identify the issues involved in implementing the protocols running within the network. In many respects, network applications and network protocols are very similar—the way an application engages the services of the network is pretty much the same as the way a high-level protocol invokes the services of a low-level protocol. As we will see later in the section, however, there are a couple of important differences.

1.4.1 Application Programming Interface (Sockets)

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

Although each operating system is free to define its own network API (and most have), over time certain of these APIs have become widely supported; that is, they have been ported to operating systems other than their native system. This is what has happened with the *socket interface* originally provided by the Berkeley distribution of Unix, which is now supported in virtually all popular operating systems. The advantage of industry-wide support for a single API is that applications can be easily ported from one OS to another, and that developers can easily write applications for multiple OSs. It is important to keep in mind, however, that application programs typically interact with many parts of the OS other than the network; for example, they read and write files, fork concurrent processes, and output to the graphical display. Just because two systems support the same network API does not mean that their file system, process, or graphic interfaces are the same. Still, understanding a widely adopted API like Unix sockets gives us a good place to start.

Before describing the socket interface, it is important to keep two concerns separate in your mind. Each protocol provides a certain set of *services*, and the API provides a *syntax* by which those services can be invoked in this particular OS. The implementation is then responsible for mapping the tangible set of operations and objects defined by the API onto the abstract set of services defined by the protocol. If you have done a good job of defining the interface, then it will be possible to use the syntax of the interface to

invoke the services of many different protocols. Such generality was certainly a goal of the socket interface, although it's far from perfect.

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

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

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

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

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

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

The bind operation, as its name suggests, binds the newly created socket to the specified address. This is the network address of the *local* participant—the server. Note that, when used with the Internet protocols, address is a data structure that includes both the IP address of the server and a TCP port number. (As we will see in Chapter 5, ports are used to indirectly identify processes. They are a form of *demux keys* as defined in Section 1.3.1.) The port number is usually some well-known number specific to the service being offered; for example, web servers commonly accept connections on port 80.

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

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

```
int connect(int socket, struct sockaddr *address, intaddr_len)
```

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

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

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

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

1.4.2 Example Application

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

Client

We start with the client side, which takes the name of the remote machine as an argument. It calls the Unix utility gethostbyname to translate this name into the remote

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

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

#define SERVER_PORT 5432
#define MAX_LINE 256

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

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

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

```
  }

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

  /* active open */
  if ((s = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
    perror("simplex-talk: socket");
    exit(1);
  }
  if (connect(s, (struct sockaddr *)&sin, sizeof(sin)) < 0) {
    perror("simplex-talk: connect");
    close(s);
    exit(1);
  }
  /* main loop: get and send lines of text */
  while (fgets(buf, sizeof(buf), stdin)) {
    buf[MAX_LINE-1] = '\0';
    len = strlen(buf) + 1;
    send(s, buf, len, 0);
  }
}
```

Server

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

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

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

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

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

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

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

1.4.3 Protocol Implementation Issues

As mentioned at the beginning of this section, the way application programs interact with the underlying network is similar to the way a high-level protocol interacts with a low-level protocol. For example, TCP needs an interface to send outgoing messages to IP, and IP needs to be able to deliver incoming messages to TCP. This is exactly the service interface introduced in Section 1.3.1.

Since we already have a network API (e.g., sockets), we might be tempted to use this same interface between every pair of protocols in the protocol stack. Although certainly an option, in practice the socket interface is not used in this way. The reason is that there are inefficiencies built into the socket interface that protocol implementers are not willing to tolerate. Application programmers tolerate them because they simplify their programming task, and because the inefficiency only has to be tolerated once, but protocol implementers are often obsessed with performance and must worry about getting a message through several layers of protocols. The rest of this section discusses the two primary differences between the network API and the protocol-to-protocol interface found lower in the protocol graph.

Process Model

Most operating systems provide an abstraction called a *process*, or alternatively, a *thread*. Each process runs largely independently of other processes, and the OS is responsible for making sure that resources, such as address space and CPU cycles, are allocated to all the current processes. The process abstraction makes it fairly straightforward to have a lot of things executing concurrently on one machine; for example, each user application might execute in its own process, and various things inside the OS might execute as other processes. When the OS stops one process from executing on the CPU and starts up another one, we call the change a *context switch*.

When designing the network subsystem, one of the first questions to answer is, "Where are the processes?" There are essentially two choices, as illustrated in Figure 1.16. In the first, which we call the *process-per-protocol* model, each protocol is implemented by a separate process. This means that as a message moves up or down the protocol stack, it is passed from one process/protocol to another—the process that implements protocol i processes the message, then passes it to protocol $i - 1$, and so on. How one process/protocol passes a message to the next process/protocol depends on the support the host OS provides for interprocess communication. Typically, there is a simple mechanism for enqueuing a message with a process. The important point, however, is that a context switch is required at each level of the protocol graph—typically a time-consuming operation.

The alternative, which we call the *process-per-message* model, treats each protocol as a static piece of code and associates the processes with the messages. That is, when a

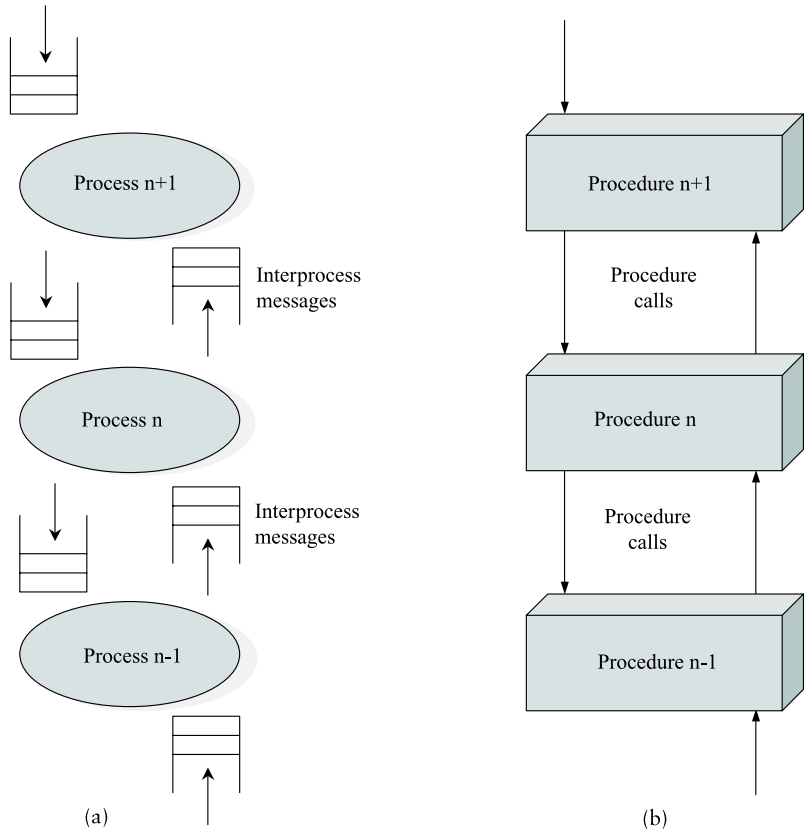

Figure 1.16 Alternative process models: (a) process-per-protocol; (b) process-per-message.

message arrives from the network, the OS dispatches a process that it makes responsible for the message as it moves up the protocol graph. At each level, the procedure that implements that protocol is invoked, which eventually results in the procedure for the next protocol being invoked, and so on. For outbound messages, the application's process invokes the necessary procedure calls until the message is delivered. In both directions, the protocol graph is traversed in a sequence of procedure calls.

Although the process-per-protocol model is sometimes easier to think about—I implement my protocol in my process, and you implement your protocol in your process—the process-per-message model is generally more efficient for a simple reason: A procedure call is an order of magnitude more efficient than a context switch on most

computers. The former model requires the expense of a context switch at each level, while the latter model costs only a procedure call per level.

Message Buffers

A second inefficiency of the socket interface is that the application process provides the buffer that contains the outbound message when calling send, and similarly it provides the buffer into which an incoming message is copied when invoking the receive operation. This forces the topmost protocol to copy the message from the application's buffer into a network buffer, and vice versa, as shown in Figure 1.17. It turns out that copying data from one buffer to another is one of the most expensive things a protocol implementation can do. This is because while processors are becoming faster at an incredible pace, memory is not getting faster as quickly as processors are. Relative to processors, memory is getting slower.

Instead of copying message data from one buffer to another at each layer in the protocol stack, most network subsystems define an abstract data type for messages that is shared by all protocols in the protocol graph. Not only does this abstraction permit messages to be passed up and down the protocol graph without copying, but it usually provides copy-free ways of manipulating messages in other ways, such as adding and stripping headers, fragmenting large messages into a set of small messages, and reassembling a collection of small messages into a single large message. The exact form of this message abstraction differs from OS to OS, but it generally involves a linked-list of pointers to message buffers, similar to the one shown in Figure 1.18. We leave it as an exercise for the reader to define a general copy-free message abstraction.

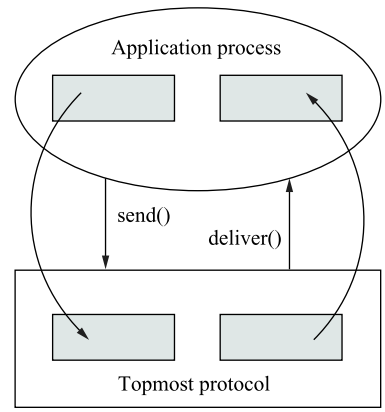

Figure 1.17 Copying incoming/outgoing messages between application buffer and network buffer.

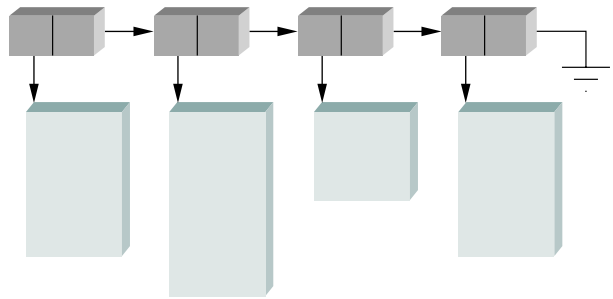

Figure 1.18 Example message data structure.

1.5 Performance

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

1.5.1 Bandwidth and Latency

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

While you can talk about the bandwidth of the network as a whole, sometimes you want to be more precise, focusing, for example, on the bandwidth

Bandwidth and Throughput

Bandwidth and *throughput* are two of the most confusing terms used in networking. While we could try to give you a precise definition of each term, it is important that you know how other people might use them and for you to be aware that they are often used interchangeably. First of all, bandwidth is literally a measure of the width of a frequency band. For example, a voice-grade telephone line supports a frequency band ranging from

300 to 3,300 Hz; it is said to have a bandwidth of 3,300 Hz − 300 Hz = 3,000 Hz. If you see the word "bandwidth" used in a situation in which it is being measured in hertz, then it probably refers to the range of signals that can be accommodated.

When we talk about the bandwidth of a communication link, we normally refer to the number of bits per second that can be transmitted on the link. We might say that the bandwidth of an Ethernet is 10 Mbps. A useful distinction might be made, however, between the bandwidth that is available on the link and the number of bits per second that we can actually transmit over the link in practice. We tend to use the word "throughput" to refer to the *measured performance* of a system. Thus, because of of a single physical link or a logical process-to-process channel. At the physical level, bandwidth is constantly improving, with no end in sight. Intuitively, if you think of a second of time as a distance you could measure with a ruler, and bandwidth as how many bits fit in that distance, then you can think of each bit as a pulse of some width. For example, each bit on a 1-Mbps link is 1 μs wide, while each bit on a 2-Mbps link is 0.5 μs wide, as illustrated in Figure 1.19. The more sophisticated the transmitting and receiving technology, the narrower each bit can become, and thus, the higher the bandwidth. For logical process-to-process channels, bandwidth is also influenced by other factors, including how many times the software that implements the channel has to handle, and possibly transform, each bit of data.

The second performance metric, latency, corresponds to how long it takes a message to travel from one end of a network to the other. (As with bandwidth, we could be focused on the latency of a single link or an end-to-end channel.) Latency is measured strictly in terms of time. For example, a transcontinental network might have a latency of

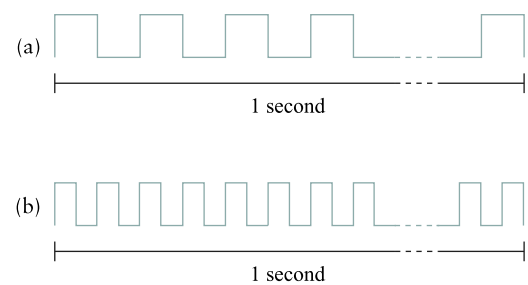

Figure 1.19 Bits transmitted at a particular bandwidth can be regarded as having some width: (a) bits transmitted at 1 Mbps (each bit 1 μs wide); (b) bits transmitted at 2 Mbps (each bit 0.5 μs wide).

24 milliseconds (ms); that is, it takes a message 24 ms to travel from one end of North America to the other. There are many situations in which it is more important to know how long it takes to send a message from one end of a network to the other and back, rather than the one-way latency. We call this the *round-trip time (RTT)* of the network.

We often think of latency as having three components. First, there is the speed-of-light propagation delay. This delay occurs because nothing, including a bit on a wire, can travel faster than the speed of light. If you know the distance between two points, you can calculate the speed-of-light latency, although you have to be careful because light travels across different mediums at different speeds: It travels at 3.0×10^8 m/s in a vacuum, 2.3×10^8 m/s in a cable, and 2.0×10^8 m/s in a fiber. Second, there is the amount of time it takes to transmit a unit of data. This is a function of the network bandwidth and the size of the packet in which the data is carried. Third, there may be queuing delays inside the network, since packet switches generally need to store packets for some time before forwarding them on an outbound link, as discussed in Section 1.2.2. So, we could define the total latency as

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

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

Bandwidth and latency combine to define the performance characteristics of a given link or channel. Their relative importance, however, depends on the application.

various inefficiencies of implementation, a pair of nodes connected by a link with a bandwidth of 10 Mbps might achieve a throughput of only 2 Mbps. This would mean that an application on one host could send data to the other host at 2 Mbps.

Finally, we often talk about the bandwidth *requirements* of an application. This is the number of bits per second that it needs to transmit over the network to perform acceptably. For some applications, this might be "whatever I can get"; for others, it might be some fixed number (preferably no more than the available link bandwidth); and for others, it might be a number that varies with time. We will provide more on this topic later in this section.

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

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

Figure 1.20 gives you a sense of how latency or bandwidth can dominate performance in different circumstances. The graph shows how long it takes to move objects of

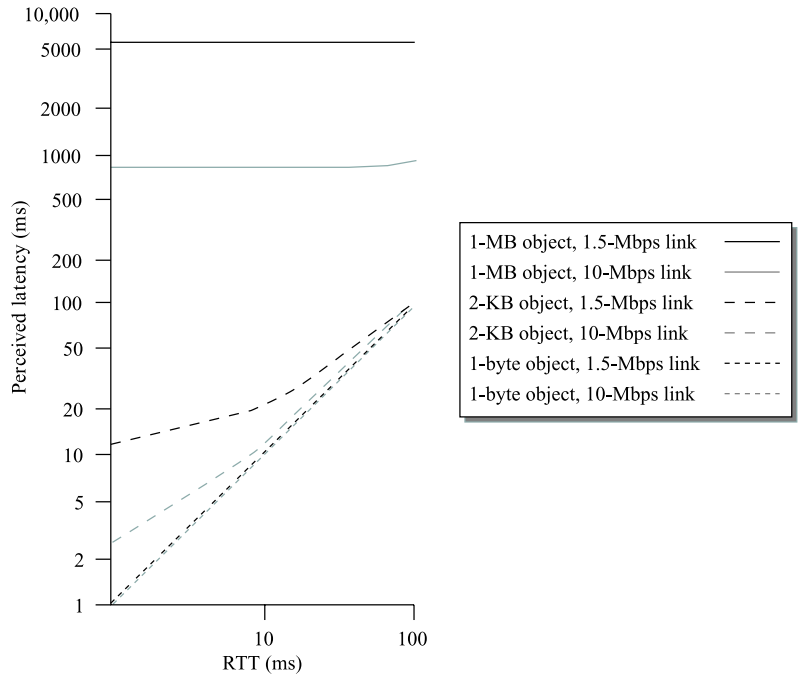

Figure 1.20 Perceived latency (response time) versus round-trip time for various object sizes and link speeds.

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

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

As an aside, computers are becoming so fast that when we connect them to networks, it is sometimes useful to think, at least figuratively, in terms of *instructions per mile*. Consider what happens when a computer that is able to execute 1 billion instructions per second sends a message out on a channel with a 100-ms RTT. (To make the math easier, assume that the message covers a distance of 5,000 miles.) If that computer sits idle the full 100 ms waiting for a reply message, then it has forfeited the ability to execute 100 million instructions, or 20,000 instructions per mile. It had better have been worth going over the network to justify this waste.

1.5.2 Delay × Bandwidth Product

It is also useful to talk about the product of these two metrics, often called the *delay* × *bandwidth product*. Intuitively, if we think of a channel between a pair of processes as a hollow pipe (see Figure 1.21), where the latency corresponds to the length of the pipe and the bandwidth gives the diameter of the pipe, then the delay × bandwidth product gives the volume of the pipe—the maximum number of bits that could be in transit through the pipe at any given instant. Said another way, if latency (measured in time) corresponds to the length of the pipe, then given the width of each bit (also measured in time), you can calculate how many bits fit in the pipe. For example, a transcontinental channel with a one-way latency of 50 ms and a bandwidth of 45 Mbps is able to hold

$$\begin{aligned} &50 \times 10^{-3} \text{ sec} \times 45 \times 10^{6} \text{ bits/sec} \\ &= 2.25 \times 10^{6} \text{ bits} \end{aligned}$$

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

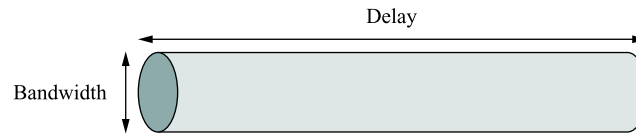

Figure 1.21 Network as a pipe.

How Big Is a Mega?

There are several pitfalls you need to be aware of when working with the common units of networking—MB, Mbps, KB, and Kbps. The first is to distinguish carefully between bits and bytes. Throughout this book, we always use a lowercase *b* for bits and a capital *B* for bytes. The second is to be sure you are using the appropriate definition of mega (M) and kilo (K). *Mega*, for example, can mean either 2^{20} or 10^6. Similarly, *kilo* can be either 2^{10} or 10^3. What is worse, in networking we typically use both definitions. Here's why.

Network bandwidth, which is often specified in terms of Mbps, is typically governed by the speed of the clock that paces the transmission of the bits. A clock that is running at 10 MHz is used to transmit bits at 10 Mbps. Because the *mega* in MHz means 10^6 hertz, Mbps is usually also defined as 10^6 bits per second. (Similarly, Kbps is 10^3 bits per second.) On the other hand, when we talk about a message that we want to transmit, we often give its size in kilobytes.

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

Note that most of the time we are interested in the RTT scenario, which we simply refer to as the delay × bandwidth product, without explicitly saying that this

Link Type	Bandwidth (Typical)	Distance (Typical)	Round-trip Delay	Delay × BW
Dial-up	56 Kbps	10 km	87 μs	5 bits
Wireless LAN	54 Mbps	50 m	0.33 μs	18 bits
Satellite	45 Mbps	35,000 km	230 ms	10 Mb
Cross-country fiber	10 Gbps	4,000 km	40 ms	400 Mb

Table 1.1 Sample delay × bandwidth products.

product is multiplied by two. Again, whether the "delay" in "delay × bandwidth" means one-way latency or RTT is made clear by the context. Table 1.1 shows some examples of delay × bandwidth products for some typical network links.

1.5.3 High-Speed Networks

The bandwidths available on today's networks are increasing at a dramatic rate, and there is eternal optimism that network bandwidth will continue to improve. This causes network designers to start thinking about what happens in the limit, or stated another way, what is the impact on network design of having infinite bandwidth available.

Although high-speed networks bring a dramatic change in the bandwidth available to applications, in many respects their impact on how we think about networking comes in what does *not* change as bandwidth increases: the speed of light. To quote Scotty from *Star Trek*, "You cannae change the laws of physics." In other words, "high speed" does not mean that latency improves at the same rate as band-

Because messages are stored in the computer's memory, and memory is typically measured in powers of two, the *K* in KB is usually taken to mean 2^{10}. (Similarly, MB usually means 2^{20}.) When you put the two together, it is not uncommon to talk about sending a 32-KB message over a 10-Mbps channel, which should be interpreted to mean $32 \times 2^{10} \times 8$ bits are being transmitted at a rate of 10×10^6 bits per second. This is the interpretation we use throughout the book, unless explicitly stated otherwise.

The good news is that many times we are satisfied with a back-of-the-envelope calculation, in which case it is perfectly reasonable to pretend that a byte has 10 bits in it (making it easy to convert between bits and bytes) and that 10^6 is really equal to 2^{20} (making it easy to convert between the two definitions of mega). Notice that the first approximation introduces a 20% error, while the latter introduces only a 5% error.

To help you in your quick-and-dirty calculations, 100 ms is a reasonable number to use for a cross-country round-trip time—at least when the country in question is the United States—and 1 ms is a good approximation of an RTT across a local area network. In the case of the former, we increase the 48-ms round-trip time implied by the speed of light over a fiber to 100 ms because there are, as we have said, other sources of delay, such as the queueing time in the switches inside the network. You can also be sure that the path taken by the fiber between two points will not be a straight line.

width; the transcontinental RTT of a 1-Gbps link is the same 100 ms as it is for a 1-Mbps link.

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

Figure 1.22 illustrates the difference between the two networks. In effect, the 1-MB file looks like a stream of data that needs to be transmitted across a 1-Mbps network, while it looks like a single packet on a 1-Gbps network. To help drive this point home, consider that a 1-MB file is to a 1-Gbps network what a 1-KB *packet* is to a 1-Mbps network.

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

Perhaps the best way to understand the relationship between throughput and latency is to return to basics. The effective end-to-end throughput that can be achieved over a network is given by the simple relationship

```
Throughput = TransferSize/TransferTime
```

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

```
TransferTime = RTT + 1/Bandwidth x TransferSize
```

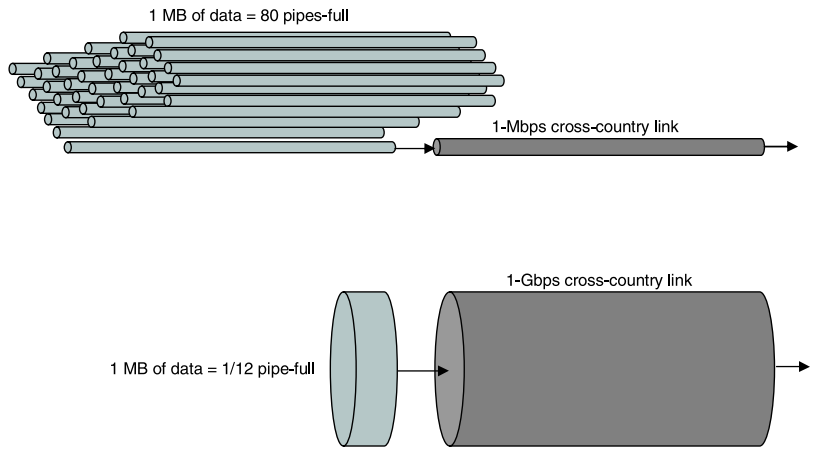

Figure 1.22 Relationship between bandwidth and latency. A 1-MB file would fill the 1-Mbps link 80 times, but only fill the 1-Gbps link 1/12 of one time.

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

$$1 \text{ MB}/108 \text{ ms} = 74.1 \text{ Mbps}$$

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

1.5.4 Application Performance Needs

The discussion in this section has taken a network-centric view of performance; that is, we have talked in terms of what a given link or channel will support. The unstated assumption has been that application programs have simple needs—they want as much bandwidth as the network can provide. This is certainly true of the aforementioned digital library program that is retrieving a 25-MB image; the more bandwidth that is available, the faster the program will be able to return the image to the user.

However, some applications are able to state an upper limit on how much bandwidth they need. Video applications are a prime example. Suppose one wants to stream a video image; that is one-quarter the size of a standard TV image; that is, it has a resolution of 352 by 240 pixels. If each pixel is represented by 24 bits of information, as would be the case for 24-bit color, then the size of each frame would be

$$(352 \times 240 \times 24)/8 = 247.5 \text{ KB}$$

If the application needs to support a frame rate of 30 frames per second, then it might request a throughput rate of 75 Mbps. The ability of the network to provide more bandwidth is of no interest to such an application because it has only so much data to transmit in a given period of time.

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

The key issue is the time interval over which the average is computed. Suppose that this example video application can be compressed down to the point that it needs only 2 Mbps, on average. If it transmits 1 Mb in a 1-second interval and 3 Mb in the following 1-second interval, then over the 2-second interval it is transmitting at an average rate of 2 Mbps; however, this will be of little consolation to a channel that was engineered to support no more than 2 Mb in any one second. Clearly, just knowing the average bandwidth needs of an application will not always suffice.

Generally, however, it is possible to put an upper bound on how large a burst an application like this is likely to transmit. A burst might be described by some peak rate that is maintained for some period of time. Alternatively, it could be described as the number of bytes that can be sent at the peak rate before reverting to the average rate or some lower rate. If this peak rate is higher than the available channel capacity, then the excess data will have to be buffered somewhere, to be transmitted later. Knowing how big of a burst might be sent allows the network designer to allocate sufficient buffer capacity to hold the burst. We will return to the subject of describing bursty traffic accurately in Chapter 6.

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

Figure 1.23 Network-induced jitter.

Consider the situation in which the source sends a packet once every 33 ms, as would be the case for a video application transmitting frames 30 times a second. If the packets arrive at the destination spaced out exactly 33 ms apart, then we can deduce that the delay experienced by each packet in the network was exactly the same. If the spacing between when packets arrive at the destination—sometimes called the *interpacket gap*—is variable, however, then the delay experienced by the sequence of packets must have also been variable, and the network is said to have introduced jitter into the packet stream, as shown in Figure 1.23. Such variation is generally not introduced in a single physical link, but it can happen when packets experience different queuing delays in a multihop packet-switched network. This queuing delay corresponds to the Queue component of latency defined earlier in this section, which varies with time.

To understand the relevance of jitter, suppose that the packets being transmitted over the network contain video frames, and in order to display these frames on the screen the receiver needs to receive a new one every 33 ms. If a frame arrives early, then it can simply be saved by the receiver until it is time to display it. Unfortunately, if a frame arrives late, then the receiver will not have the frame it needs in time to update the screen, and the video quality will suffer; it will not be smooth. Note that it is not necessary to eliminate jitter, only to know how bad it is. The reason for this is that if the receiver knows the upper and lower bounds on the latency that a packet can experience, it can delay the time at which it starts playing back the video (i.e., displays the first frame) long enough to ensure that in the future it will always have a frame to display when it needs it. The receiver delays the frame, effectively smoothing out the jitter, by storing it in a buffer. We return to the topic of jitter in Chapter 6.

1.6 Summary

Computer networks like the Internet have experienced enormous growth over the past decade and are now positioned to provide a wide range of services—remote file access, digital libraries, videoconferencing—to hundreds of millions of users. Much of this growth can be attributed to the general-purpose nature of computer networks, and in

particular to the ability to add new functionality to the network by writing software that runs on affordable, high-performance computers. With this in mind, the overriding goal of this book is to describe computer networks in such a way that when you finish reading it, you should feel that if you had an army of programmers at your disposal, you could actually build a fully-functional computer network from the ground up. This chapter lays the foundation for realizing this goal.

The first step we have taken toward this goal is to carefully identify exactly what we expect from a network. For example, a network must first provide cost-effective connectivity among a set of computers. This is accomplished through a nested interconnection of nodes and links, and by sharing this hardware base through the use of statistical multiplexing. This results in a packet-switched network, on top of which we then define a collection of process-to-process communication services.

The second step is to define a layered architecture that will serve as a blueprint for our design. The central objects of this architecture are network protocols. Protocols both provide a communication service to higher-level protocols and define the form and meaning of messages exchanged with their peers running on other machines. We have briefly surveyed two of the most widely used architectures: the OSI architecture and the Internet architecture. This book most closely follows the Internet architecture, both in its organization and as a source of examples.

The third step is to implement the network's protocols and application programs, usually in software. Both protocols and applications need an interface by which they invoke the services of other protocols in the network subsystem. The socket interface is the most widely used interface between application programs and the network subsystem, but a slightly different interface is typically used within the network subsystem.

Finally, the network as a whole must offer high performance, where the two performance metrics we are most interested in are latency and throughput. As we will see in later chapters, it is the product of these two metrics—the so-called delay × bandwidth product—that often plays a critical role in protocol design.

OPEN ISSUE

Ubiquitous Networking

There is little doubt that computer networks are becoming an integral part of the everyday lives of vast numbers of people. What began over 35 years ago as experimental systems like the ARPANET—connecting mainframe computers over long-distance telephone lines—has turned into big business. And where there is big business, there are lots of players. In this case, there is the computing industry, which has become

increasingly involved in supporting packet-switched networking products; the telephone carriers, which recognize the market for carrying all sorts of data, not just voice; and the cable TV industry, which in parts of the world involved in both the delivery of "content" (e.g. video-on-demand) and the provision of high-speed residential connections to the Internet. And this list does not even include the many players involved in delivery of services over the Internet such as voiceover IP (VoIP) and electronic commerce.

Assuming that the goal is ubiquitous networking—to bring the network into every household—the first problem that must be addressed is how to establish the necessary physical links. The most widely discussed options in most parts of the world make use of either the existing cable TV facilities or the copper pairs used to deliver telephone service. Fiber to the home, or to the apartment building, which not long ago looked like a pipe dream, is gathering momentum in some areas. There have also been developments in the technology to deliver network connectivity over power lines, and, as we will see in the next chapter, there is now an abundance of wireless networking technologies. Increasingly this is leading to an expectation that access to the Internet is available everywhere, not just in the workplace or at home.

How the struggle between the computer companies, the telephone companies, the cable industry, and other stakeholders in the networking business will play out in the marketplace is anyone's guess. (If we knew the answer, we'd be charging a lot more for this book.) All we know is that there are many technical obstacles—issues of connectivity, levels of service, performance, reliability, security, and fairness—that stand between the current state-of-the-art and the sort of global, ubiquitous, heterogeneous network that we believe is possible and desirable. It is these challenges that are the focus of this book.

FURTHER READING

Computer networks are not the first communication-oriented technology to have found their way into the everyday fabric of our society. For example, the early part of this century saw the introduction of the telephone, and then during the 1950s television became widespread. When considering the future of networking—how widely it will spread and how we will use it—it is instructive to study this history. Our first reference is a good starting point for doing this (the entire issue is devoted to the first 100 years of telecommunications).

The second and third papers are the seminal papers on the OSI and Internet architectures, respectively. The Zimmerman paper introduces the OSI architecture, and the Clark paper is a retrospective. The final two papers are not specific to networking, but present viewpoints that capture the "systems approach" of this book. The Saltzer et al. paper motivates and describes one of the most widely applied rules of network architecture—the *end-to-end argument*. The paper by Mashey describes the thinking be-

hind RISC architectures; as we will soon discover, making good judgments about where to place functionality in a complex system is what system design is all about.

- Pierce, J. "Telephony—A Personal View." *IEEE Communications* 22(5):116–120, May 1984.
- Zimmerman, H. "OSI Reference Model—The ISO Model of Architecture for Open Systems Interconnection." *IEEE Transactions on Communications* COM-28(4):425–432, April 1980.
- Clark, D. "The Design Philosophy of the DARPA Internet Protocols." *Proceedings of the SIGCOMM '88 Symposium*, pp. 106–114, August 1988.
- Saltzer, J., D. Reed, and D. Clark. "End-to-End Arguments in System Design." *ACM Transactions on Computer Systems* 2(4):277–288, November 1984.
- Mashey, J. "RISC, MIPS, and the Motion of Complexity." *UniForum 1986 Conference Proceedings*, pp. 116–124, 1986.

Several texts offer an introduction to computer networking: Stallings gives an encyclopedic treatment of the subject, with an emphasis on the lower levels of the OSI hierarchy [Sta07]; Tanenbaum uses the OSI architecture as an organizational model [Tan03]; Comer gives an overview of the Internet architecture [Com00]; and Bertsekas and Gallager discuss networking from a performance modeling perspective [BG92].

To put computer networking into a larger context, two books—one dealing with the past and the other looking toward the future—are must reading. The first is Holzmann and Pehrson's *The Early History of Data Networks* [HP95]. Surprisingly, many of the ideas covered in the book you are now reading were invented during the 1700s. The second is *Realizing the Information Future: The Internet and Beyond*, a book prepared by the Computer Science and Telecommunications Board of the National Research Council [NRC94].

To follow the history of the Internet from its beginning, the reader is encouraged to peruse the Internet's *Request for Comments (RFC)* series of documents. These documents, which include everything from the TCP specification to April Fools' jokes, are retrievable at http://www.ietf.org/rfc.html. For example, the protocol specifications for TCP, UDP, and IP are available in RFC 793, 768, and 791, respectively.

To gain a better appreciation for the Internet philosophy and culture, two references are recommended; both are also quite entertaining. Padlipsky gives a good description of the early days, including a pointed comparison of the Internet and OSI architectures [Pad85]. For an account of what really happens behind the scenes at the Internet Engineering Task Force, we recommend Boorsook's article [Boo95].

There are a wealth of articles discussing various aspects of protocol implementations. A good starting point is to understand two complete protocol implementation environments: the Stream mechanism from System V Unix [Rit84] and the *x*-kernel [HP91]. In addition, [LMKQ89] and [SW95] describe the widely used Berkeley Unix implementation of TCP/IP.

More generally, there is a large body of work addressing the issue of structuring and optimizing protocol implementations. Clark was one of the first to discuss the relationship between modular design and protocol performance [Cla82]. Later papers then introduce the use of upcalls in structuring protocol code [Cla85] and study the processing overheads in TCP [CJRS89]. Finally, [WM87] describes how to gain efficiency through appropriate design and implementation choices.

Several papers have introduced specific techniques and mechanisms that can be used to improve protocol performance. For example, [HMPT89] describes some of the mechanisms used in the *x*-kernel, [MD93] discusses various implementations of demultiplexing tables, [VL87] introduces the timing-wheel mechanism used to manage protocol events, and [DP93] describes an efficient buffer management strategy. Also, the performance of protocols running on parallel processors—locking is a key issue in such environments—is discussed in [BG93] and [NYKT94].

Because many aspects of protocol implementation depend on an understanding of the basics of operating systems, we recommend Finkel [Fin88], Bic and Shaw [BS88], and Tanenbaum [Tan01] for an introduction to OS concepts.

Finally, we conclude the Further Reading section of each chapter with a set of live references; that is, URLs for locations on the World Wide Web where you can learn more about the topics discussed in that chapter. Since these references are live, it is possible that they will not remain active for an indefinite period of time. For this reason, we limit the set of live references at the end of each chapter to sites that either export software, provide a service, or report on the activities of an ongoing working group or standardization body. In other words, we only give URLs for the kinds of material that cannot easily be referenced using standard citations. For this chapter, we include four live references:

- http://www.mkp.com/pd4e: Information about this book, including supplements, addenda, and so on.
- http://www.acm.org/sigcomm/sos.html: Status of various networking standards, including those of the IETF, ISO, and IEEE.
- http://www.ietf.org/: Information about the IETF and its working groups.
- http://edas.info/S.cgi?search=1: Searchable bibliography of network-related research papers.

EXERCISES

1 Use anonymous FTP to connect to ftp.isi.edu (directory in-notes), and retrieve the RFC index. Also retrieve the protocol specifications for TCP, IP, and UDP.

2 Look up the website

http://www.cs.princeton.edu/nsg

Here you can read about current network research underway at Princeton University and see a picture of author Larry Peterson. Follow links to find a biography of author Bruce Davie.

3 Use a Web search tool to locate useful, general, and noncommercial information about the following topics: MBone, ATM, MPEG, IPv6, and Ethernet.

4 The Unix utility whois can be used to find the domain name corresponding to an organization, or vice versa. Read the man page documentation for whois and experiment with it. Try whois princeton.edu and whois princeton, for starters. As an alternative, explore the whois interface at http://www.internic.net/whois.html.

5 Calculate the total time required to transfer a 1,000-KB file in the following cases, assuming an RTT of 100 ms, a packet size of 1-KB data, and an initial $2 \times$ RTT of "handshaking" before data is sent.

(a) The bandwidth is 1.5 Mbps, and data packets can be sent continuously.

(b) The bandwidth is 1.5 Mbps, but after we finish sending each data packet we must wait one RTT before sending the next.

(c) The bandwidth is "infinite," meaning that we take transmit time to be zero, and up to 20 packets can be sent per RTT.

(d) The bandwidth is infinite, and during the first RTT we can send one packet (2^{1-1}), during the second RTT we can send two packets (2^{2-1}), during the third we can send four (2^{3-1}), and so on. (A justification for such an exponential increase will be given in Chapter 6.)

✓ **6** Calculate the total time required to transfer a 1.5 MB file in the following cases, assuming a RTT of 80 ms, a packet size of 1 KB data, and an initial $2 \times$ RTT of "handshaking" before data is sent.

(a) The bandwidth is 10 Mbps, and data packets can be sent continuously.

(b) The bandwidth is 10 Mbps, but after we finish sending each data packet we must wait one RTT before sending the next.

(c) The link allows infinitely fast transmit, but limits bandwidth such that only 20 packets can be sent per RTT.

(d) Zero transmit time as in (c), but during the first RTT we can send one packet, during the second RTT we can send two packets, during the third we can send four $= 2^{3-1}$, and so on. (A justification for such an exponential increase will be given in Chapter 6.)

7 Consider a point-to-point link 2 km in length. At what bandwidth would propagation delay (at a speed of 2×10^8 m/sec) equal transmit delay for 100-byte packets? What about 512-byte packets?

✓ **8** Consider a point-to-point link 50 km in length. At what bandwidth would propagation delay (at a speed of 2×10^8 m/sec) equal transmit delay for 100-byte packets? What about 512-byte packets?

9 What properties of postal addresses would be likely to be shared by a network addressing scheme? What differences might you expect to find? What properties of telephone numbering might be shared by a network addressing scheme?

10 One property of addresses is that they are unique; if two nodes had the same address it would be impossible to distinguish between them. What other properties might be useful for network addresses to have? Can you think of any situations in which network (or postal or telephone) addresses might *not* be unique?

11 Give an example of a situation in which multicast addresses might be beneficial.

12 What differences in traffic patterns account for the fact that STDM is a cost-effective form of multiplexing for a voice telephone network and FDM is a cost-effective form of multiplexing for television and radio networks, yet we reject both as not being cost-effective for a general-purpose computer network?

13 How "wide" is a bit on a 1-Gbps link? How long is a bit in copper wire, where the speed of propagation is 2.3×10^8 m/s?

14 How long does it take to transmit x KB over a y-Mbps link? Give your answer as a ratio of x and y.

15 Suppose a 100-Mbps point-to-point link is being set up between Earth and a new lunar colony. The distance from the moon to Earth is approximately 385,000 km, and data travels over the link at the speed of light—3×10^8 m/s.

(a) Calculate the minimum RTT for the link.

(b) Using the RTT as the delay, calculate the delay × bandwidth product for the link.

(c) What is the significance of the delay × bandwidth product computed in (b)?

(d) A camera on the lunar base takes pictures of Earth and saves them in digital format to disk. Suppose Mission Control on Earth wishes to download the most current image, which is 25 MB. What is the minimum amount of time that will elapse between when the request for the data goes out and the transfer is finished?

✓ **16** Suppose a 128-Kbps point-to-point link is set up between Earth and a rover on Mars. The distance from Earth to Mars (when they are closest together) is approximately 55 Gm, and data travels over the link at the speed of light—3×10^8 m/sec.

(a) Calculate the minimum RTT for the link.

(b) Calculate the delay × bandwidth product for the link.

(c) A camera on the rover takes pictures of its surroundings and sends these to Earth. How quickly after a picture is taken can it reach Mission Control on Earth? Assume that each image is 5 MB in size.

17 For each of the following operations on a remote file server, discuss whether they are more likely to be delay sensitive or bandwidth sensitive.

(a) Open a file.

(b) Read the contents of a file.

(c) List the contents of a directory.

(d) Display the attributes of a file.

18 Calculate the latency (from first bit sent to last bit received) for the following:

(a) A 10-Mbps Ethernet with a single store-and-forward switch in the path, and a packet size of 5,000 bits. Assume that each link introduces a propagation delay of 10 μs, and that the switch begins retransmitting immediately after it has finished receiving the packet.

(b) Same as (a) but with three switches.

(c) Same as (a) but assume the switch implements "cut-through" switching: it is able to begin retransmitting the packet after the first 200 bits have been received.

✓ **19** Calculate the latency (from first bit sent to last bit received) for:

(a) A 1-Gbps Ethernet with a single store-and-forward switch in the path, and a packet size of 5,000 bits. Assume that each link introduces a propagation delay of 10 μs and that the switch begins retransmitting immediately after it has finished receiving the packet.

(b) Same as (a) but with three switches.

(c) Same as (b) but assume the switch implements cut-through switching: it is able to begin retransmitting the packet after the first 128 bits have been received.

20 Calculate the effective bandwidth for the following cases. For (a) and (b) assume there is a steady supply of data to send; for (c) simply calculate the average over 12 hours.

(a) A 10-Mbps Ethernet through three store-and-forward switches as in Exercise 18(b). Switches can send on one link while receiving on the other.

(b) Same as (a) but with the sender having to wait for a 50-byte acknowledgment packet after sending each 5,000-bit data packet.

(c) Overnight (12-hour) shipment of 100 compact discs (650 MB each).

21 Calculate the bandwidth $\times$ delay product for the following links. Use one-way delay, measured from first bit sent to first bit received.

(a) A 10-Mbps Ethernet with a delay of 10 μs.

Figure 1.24 Diagram for Exercise 22.

(b) A 10-Mbps Ethernet with a single store-and-forward switch like that of Exercise 18(a), packet size 5,000 bits, and 10 μs per link propagation delay.

(c) A 1.5-Mbps T1 link, with a transcontinental one-way delay of 50 ms.

(d) A 1.5-Mbps T1 link through a satellite in geosynchronous orbit, 35,900-km high. The only delay is speed-of-light propagation delay.

22 Hosts A and B are each connected to a switch S via 10-Mbps links as in Figure 1.24. The propagation delay on each link is 20 μs. S is a store-and-forward device; it begins retransmitting a received packet 35 μs after it has finished receiving it. Calculate the total time required to transmit 10,000 bits from A to B.

(a) As a single packet.

(b) As two 5,000-bit packets sent one right after the other.

23 Suppose a host has a 1-MB file that is to be sent to another host. The file takes 1 second of CPU time to compress 50%, or 2 seconds to compress 60%.

(a) Calculate the bandwidth at which each compression option takes the same total compression + transmission time.

(b) Explain why latency does not affect your answer.

24 Suppose that a certain communications protocol involves a per-packet overhead of 100 bytes for headers and framing. We send 1 million bytes of data using this protocol; however, one data byte is corrupted and the entire packet containing it is thus lost. Give the total number of overhead + loss bytes for packet data sizes of 1,000, 5,000, 10,000, and 20,000 bytes. Which size is optimal?

25 Assume you wish to transfer an n B file along a path composed of the source, destination, seven point-to-point links, and five switches. Suppose each link has a propagation delay of 2 ms, bandwidth of 4 Mbps, and that the switches support both circuit and packet switching. Thus, you can either break the file

up into 1-KB packets, or set up a circuit through the switches and send the file as one contiguous bitstream. Suppose that packets have 24 B of packet header information and 1,000 B of payload, that store-and-forward packet processing at each switch incurs a 1-ms delay after the packet had been completely received, that packets may be sent continuously without waiting for acknowledgments, and that circuit setup requires a 1-KB message to make one round-trip on the path incurring a 1-ms delay at each switch after the message has been completely received. Assume switches introduce no delay to data traversing a circuit. You may also assume that file size is a multiple of 1,000 B.

(a) For what file size n B is the total number of bytes sent across the network less for circuits than for packets?

(b) For what file size n B is the total latency incurred before the entire file arrives at the destination less for circuits than for packets?

(c) How sensitive are these results to the number of switches along the path? To the bandwidth of the links? To the ratio of packet size to packet header size?

(d) How accurate do you think this model of the relative merits of circuits and packets is? Does it ignore important considerations that discredit one or the other approach? If so, what are they?

26 Consider a network with a ring topology, link bandwidths of 100 Mbps, and propagation speed 2×10^8 m/s. What would the circumference of the loop be to exactly contain one 250-byte packet, assuming nodes do not introduce delay? What would the circumference be if there was a node every 100 m, and each node introduced 10 bits of delay?

27 Compare the channel requirements for voice traffic with the requirements for the real-time transmission of music, in terms of bandwidth, delay, and jitter. What would have to improve? By approximately how much? Could any channel requirements be relaxed?

28 For the following, assume that no data compression is done; this would in practice almost never be the case. For (a)–(c), calculate the bandwidth necessary for transmitting in real time:

(a) Video at a resolution of 640×480, 3 bytes/pixel, 30 frames/second.

(b) 160×120 video, 1 byte/pixel, 5 frames/second.

(c) CD-ROM music, assuming one CD holds 75 minutes' worth and takes 650 MB.

(d) Assume a fax transmits an 8 × 10-inch black-and-white image at a resolution of 72 pixels per inch. How long would this take over a 14.4-Kbps modem?

✓ **29** For the following, as in the previous problem, assume that no data compression is done. Calculate the bandwidth necessary for transmitting in real time:

(a) HDTV high-definition video at a resolution of 1,920 × 1,080, 24 bits/pixel, 30 frames/sec.

(b) Plain old telephone service (POTS) voice audio of 8-bit samples at 8 KHz.

(c) GSM mobile voice audio of 260-bit samples at 50 Hz.

(d) HDCD high-definition audio of 24-bit samples at 88.2 kHz.

30 Discuss the relative performance needs of the following applications, in terms of average bandwidth, peak bandwidth, latency, jitter, and loss tolerance:

(a) File server.

(b) Print server.

(c) Digital library.

(d) Routine monitoring of remote weather instruments.

(e) Voice.

(f) Video monitoring of a waiting room.

(g) Television broadcasting.

31 Suppose a shared medium M offers to hosts $A_1, A_2, \ldots, A_N$ in round-robin fashion an opportunity to transmit one packet; hosts that have nothing to send immediately relinquish M. How does this differ from STDM? How does network utilization of this scheme compare with STDM?

★ **32** Consider a simple protocol for transferring files over a link. After some initial negotiation, A sends data packets of size 1 KB to B; B then replies with an acknowledgment. A always waits for each ACK before sending the next data

packet; this is known as *stop-and-wait*. Packets that are overdue are presumed lost and are retransmitted.

(a) In the absence of any packet losses or duplications, explain why it is not necessary to include any "sequence number" data in the packet headers.

(b) Suppose that the link can lose occasional packets, but that packets that do always arrive in the order sent. Is a 2-bit sequence number (that is, N mod 4) enough for A and B to detect and resend any lost packets? Is a 1-bit sequence number enough?

(c) Now suppose that the link can deliver out of order, and that sometimes a packet can be delivered as much as 1 minute after subsequent packets. How does this change the sequence number requirements?

★ **33** Suppose hosts A and B are connected by a link. Host A continuously transmits the current time from a high-precision clock, at a regular rate, fast enough to consume all the available bandwidth. Host B reads these time values and writes them each paired with its own time from a local clock synchronized with A's. Give qualitative examples of B's output assuming the link has

(a) High bandwidth, high latency, low jitter.

(b) Low bandwidth, high latency, high jitter.

(c) High bandwidth, low latency, low jitter, occasional lost data.

For example, a link with zero jitter, a bandwidth high enough to write on every other clock tick, and a latency of 1 tick might yield something like (0000, 0001), (0002, 0003), (0004, 0005).

34 Obtain and build the simplex-talk sample socket program shown in the text. Start one server and one client in separate windows. While the first client is running, start 10 other clients that connect to the same server; these other clients should most likely be started in the background with their input redirected from a file. What happens to these 10 clients? Do their connect()s fail, or time out, or succeed? Do any other calls block? Now let the first client exit. What happens? Try this with the server value MAX_PENDING set to 1 as well.

35 Modify the simplex-talk socket program so that each time the client sends a line to the server, the server sends the line back to the client. The client (and server) will now have to make alternating calls to recv() and send().

36 Modify the simplex-talk socket program so that it uses UDP as the transport protocol, rather than TCP. You will have to change SOCK_STREAM to SOCK_DGRAM in both client and server. Then, in the server, remove the calls to listen() and accept(), and replace the two nested loops at the end with a single loop that calls recv() with socket s. Finally, see what happens when two such UDP clients simultaneously connect to the same UDP server, and compare this to the TCP behavior.

37 Investigate the different options and parameters one can set for a TCP connection. (Do "man tcp" on Unix.) Experiment with various parameter settings to see how they effect TCP performance.

38 The Unix utility ping can be used to find the RTT to various Internet hosts. Read the man page for ping, and use it to find the RTT to www.cs.princeton.edu in New Jersey and www.cisco.com in California. Measure the RTT values at different times of day, and compare the results. What do you think accounts for the differences?

39 The Unix utility traceroute, or its Windows equivalent tracert, can be used to find the sequence of routers through which a message is routed. Use this to find the path from your site to some others. How well does the number of hops correlate with the RTT times from ping? How well does the number of hops correlate with geographical distance?

40 Use traceroute, above, to map out some of the routers within your organization (or to verify none are used).