Direct Link Networks

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

—Winston Churchill

The simplest network possible is one in which two hosts are directly connected by some physical medium. The medium be a length of wire, a piece of optical fiber, or a medium (such as air or even free space) through which electromagnetic radiation (e.g., radio waves) can be transmitted. It may cover a small area (e.g., an office building) or a wide area (e.g., transcontinental). Connecting two or more nodes with a suitable medium is only the first step, however. There are five additional problems that must be addressed before the nodes can successfully exchange packets.

P R O B L E M

Physically Connecting Hosts

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

Although these five issues—encoding, framing, error detection, reliable delivery, and access mediation—can be discussed in the abstract, they are very real problems that are addressed in different ways by different networking technologies. This chapter considers these issues in the context of four specific network technologies: point-to-point links, carrier sense multiple access (CSMA) networks (of which Ethernet is the most

famous example), token rings (of which IEEE Standard 802.5 and FDDI are the most famous examples), and wireless networks (for which 802.11 is the most widespread standard[1]). The goal of this chapter is simultaneously to survey the available network technology and to explore these five fundamental issues.

Before tackling the specific issues of connecting hosts, this chapter begins by examining the building blocks that will be used: nodes and links. We then explore the first three issues—encoding, framing, and error detection—in the context of a simple point-to-point link. The techniques introduced in these three sections are general and therefore apply equally well to multiple-access networks. The problem of reliable delivery is considered next. Since link-level reliability is usually not implemented in shared-access networks, this discussion focuses on point-to-point links only. Finally, we address the media access problem in the context of CSMA, token rings, and wireless.

[1] Strictly speaking, 802.11 is a set of standards.

2.1 Hardware Building Blocks

As we saw in Chapter 1, networks are constructed from two classes of hardware building blocks: *nodes* and *links*. This statement is just as true for the simplest possible network—one in which a single point-to-point link connects a pair of nodes—as it is for a worldwide internet. This section gives a brief overview of what we mean by nodes and links and, in so doing, defines the underlying technology that we will assume throughout the rest of this book.

2.1.1 Nodes

Nodes are often general-purpose computers, like a desktop workstation, a multiprocessor, or a PC. For our purposes, let's assume it's a workstation-class machine. This workstation can serve as a host that users run application programs on, it might be used inside the network as a switch that forwards messages from one link to another, or it might be configured as a router that forwards internet packets from one network to another. In some cases, a network node—most commonly a switch or router inside the network, rather than a host—is implemented by special-purpose hardware. This is usually done for reasons of performance and cost: It is generally possible to build custom hardware that performs a particular function faster and cheaper than a general-purpose processor can perform it. When this happens, we will first describe the basic function being performed by the node as though this function is being implemented in software on a general-purpose workstation, and then explain why and how this functionality might instead be implemented by special hardware.

Although we could leave it at that, it is useful to know a little bit about what a workstation looks like on the inside. This information becomes particularly important when we become concerned about how well the network performs. Figure 2.1 gives a simple block diagram of the workstation-class machine we assume throughout this book.

Two aspects of the memory component are important to note. First, the memory on any given machine is finite. It may be 64 MB or it may be 1 GB, but it is not infinite. As pointed out in Section 1.2.2, this is important because memory turns out to be one of the two scarce resources in the network (the other is link bandwidth) that must be carefully managed if we are to provide a fair amount of network capacity to each user. Memory is a scarce resource because, on any node that forwards packets, those packets must be buffered in memory while waiting their turn to be transmitted over an outgoing link.

Second, while CPUs are becoming faster at an unbelievable pace, the same is not true of memory. Recent performance trends show processor speeds doubling every 18 months, but memory latency improving at a rate of only 7% each year. The relevance of this difference is that as a network node, a workstation runs at memory speeds, not processor speeds, to a first approximation. This means that the network software needs

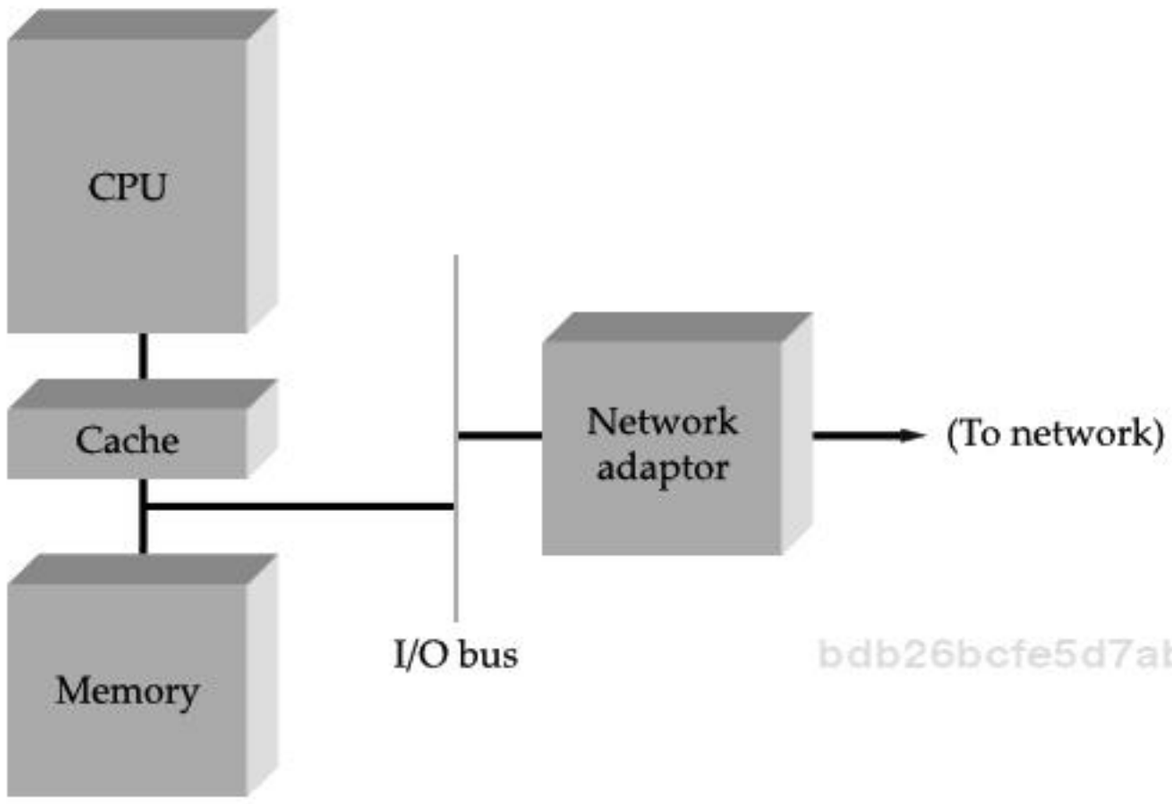

Figure 2.1 Example workstation architecture.

to be careful about how it uses memory and, in particular, about how many times it accesses memory as it processes each message. We do not have the luxury of being sloppy just because processors are becoming infinitely fast.

The workstation's network adaptor component connects the rest of the workstation to the link. More than just a physical connection, it is an active intermediary between node and link, with its own internal processor. Its role is to transmit data from the workstation onto the link, and receive data from the link, storing it for the workstation. The adaptor implements nearly all the networking functionality, to be discussed in the course of this chapter, that makes it possible to convey data over a dumb wire (or radio airwaves) between adaptors. For example, adaptors break data into frames that the link can transport, detect errors introduced as a frame travels over the link, and follow fairness rules that allow a link to be shared by multiple workstations.

A network adaptor can be thought of as having two main components: a bus interface that understands how to communicate with the host and a link interface that understands how to use the link. There must also be a communication path between these two components, over which incoming and outgoing data is passed. A simple block diagram of a network adaptor is depicted in Figure 2.2.

Different kinds of links require network adaptors with very different link interfaces. In this chapter we will see the tasks performed by the link interface for a variety of link technologies. Different I/O buses likewise require different adaptors. From the perspective of the host, however, bus interfaces tend to be similar to each other. Typically, the adaptor exports a *control status register (CSR)* that is readable and writable from the CPU. The CSR is typically located at some address in the memory, thereby making it possible

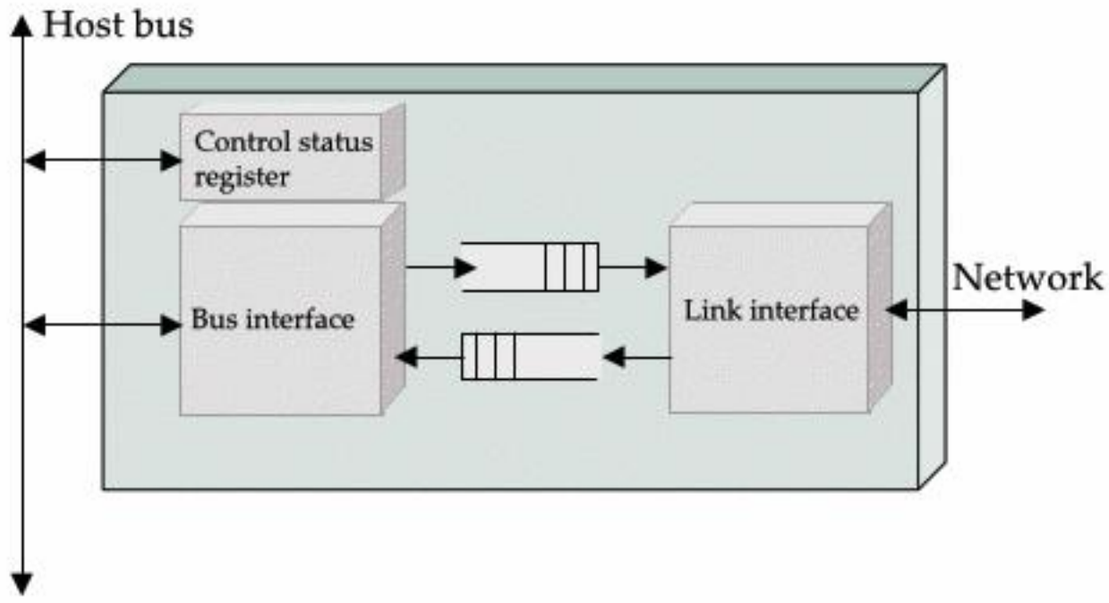

Figure 2.2 Block diagram of a typical network adaptor.

for the CPU to read and write just like any other memory location. Software on the host—a *device driver*—writes to the CSR to instruct it to transmit and/or receive data and reads from the CSR to learn the current state of the adaptor. To notify the host of an asynchronous event such as the reception of a frame, the adaptor *interrupts* the host.

One of the most important issues in network adaptor design is how bytes of data are transferred between the adaptor and the host memory. There are two basic mechanisms: *direct memory access (DMA)* and *programmed I/O (PIO)*. With DMA, the adaptor directly reads and writes the host's memory without any CPU involvement; the host simply gives the adaptor a memory address and the adaptor reads from (or writes to) it. With PIO, the CPU is directly responsible for moving data between the adaptor and the host memory: To send a frame, the CPU executes a tight loop that first reads a word from host memory and then writes it to the adaptor; to receive a frame, the CPU reads words from the adaptor and writes them to memory.

As noted earlier, host memory performance is often the limiting factor in network performance. Nowhere is this possibility more critical than at the host/adaptor interface. To help drive this point home, consider Figure 2.3. This diagram shows the bandwidth available between various components of a modern PC. While the I/O bus is fast enough to transfer frames between the network adaptor and host memory at gigabit rates, there are two potential problems.

Frames, Buffers, and Messages

As this section has suggested, the network adaptor is the place where the network comes in physical contact with the host. It also happens to be the place where three different worlds intersect: the network, the host architecture, and the host operating system. It turns out that each of these has a different terminology for talking about the same thing. It is important to recognize when this is happening.

From the network's perspective, the adaptor transmits *frames* from the

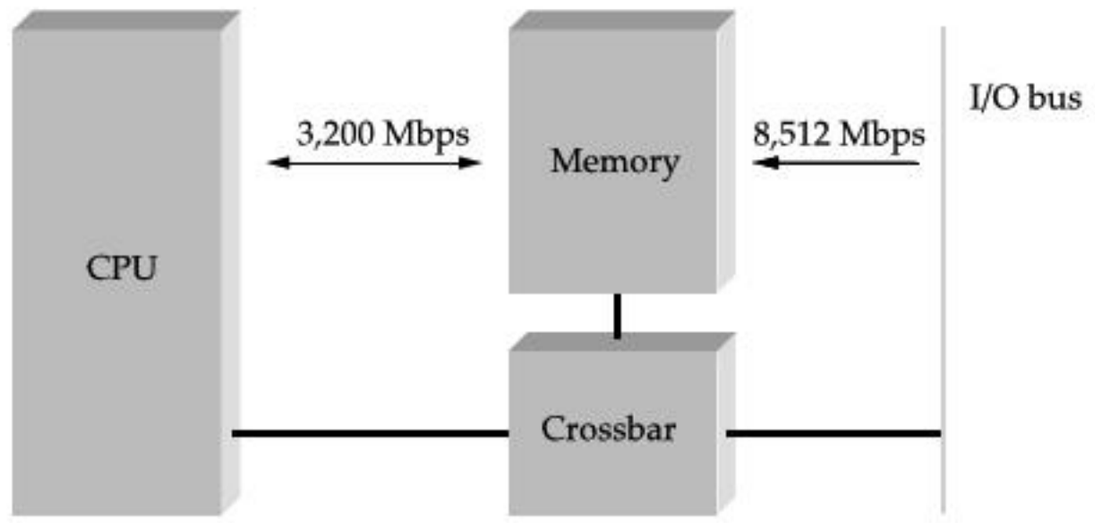

Figure 2.3 Memory bandwidth on a modern PC-class machine.

The first is that the advertised I/O bus speed corresponds to its peak bandwidth; it is the product of the bus's width and clock speed (e.g., a 64-bit-wide bus running at 133 MHz has a peak transfer rate of 8,512 Mbps). The real limitation is the size of the data block that is being transferred across the I/O bus, since there is a certain amount of overhead involved in each bus transfer. On some architectures, for example, it takes 8 clock cycles to acquire the bus for the purpose of transferring data from the adaptor to host memory. This overhead is independent of the number of data bytes transferred. Thus, if you want to transfer a 64-byte payload across the I/O bus—this happens to be the size of a minimum Ethernet packet—then the whole transfer takes 16 cycles: 8 cycles to acquire the bus and 8 cycles to transfer the data. (The bus is 64 bits wide, which means that it can transfer 8 bytes during each clock cycle; 64 bytes divided by 8 bytes per cycle equals 8 cycles.) This means that the maximum sustained bandwidth you can achieve for such packets is only half the peak (i.e., 4,256 Mbps).

The second problem is that the memory/CPU bandwidth, which in this example is 3,200 MBps (25.6 Gbps), is the same order of magnitude as the bandwidth of the I/O bus. Fortunately, this is a measured number rather than an advertised peak rate. The ramification is that while it is possible to deliver frames across the I/O bus and into memory and then to load the data from memory into the CPU's registers at network bandwidths, it is impractical for the device driver, operating

host and receives *frames* into the host. From the perspective of the host architecture, each frame is received into or transmitted from a *buffer*, which is simply a region of main memory of some length and starting at some address. Finally, from the operating system's perspective, a *message* is an abstract object that holds network frames. Messages are implemented by a data structure that includes pointers to different memory locations (buffers). We saw an example of a message data structure in Chapter 1.

system, and application to go to memory multiple times for each word of data in a network packet, possibly because it needs to copy the data from one buffer to another. In particular, if the memory/CPU path is crossed n times, then it might be the case that the bandwidth your application sees is $3{,}200/n$ MBps. (The performance might be better if the data is cached, but often caches don't help with data arriving from the network.) For example, if the various software layers need to copy the message from one buffer to another four times—not an uncommon situation—then the application might see a throughput of 800 MBps (6,400 Mbps), less than the 8,512 Mbps that the I/O bus can support.

As an aside, it is important to recognize that there are many parallels between moving a message to and from memory and moving a message across a network. In particular, the effective throughput of the memory system is defined by the same two formulas given in Section 1.5.

$$\text{Throughput} = \text{TransferSize}/\text{TransferTime}$$

$$\text{TransferTime} = \text{RTT} + 1/\text{Bandwidth} \times \text{TransferSize}$$

In the case of the memory system, however, the transfer size corresponds to how big a unit of data we can move across the bus in one transfer (i.e., cache line versus small cells versus large message), and the RTT corresponds to the memory latency, that is, whether the memory is on-chip cache, off-chip cache, or main memory. Just as in the case of the network, the larger the transfer size and the smaller the latency, the better the effective throughput. Also similar to a network, the effective memory throughput does not necessarily equal the peak memory bandwidth (i.e., the bandwidth that can be achieved with an infinitely large transfer).

The main point of this discussion is that we must be aware of the limits memory bandwidth places on network performance. If carefully designed, the system can work around these limits. For example, it is possible to integrate the buffers used by the device driver, the operating system, and the application in a way that minimizes data copies. The system also needs to be aware of when data is brought into cache, so it can perform all necessary operations on the data before it gets bumped from the cache. The details of how this is accomplished are beyond the scope of this book, but can be found in papers referenced at the end of the chapter.

Finally, there is a second important lesson lurking in this discussion: when the network isn't performing as well as you think it should, it's not always the network's fault. In many cases, the actual bottleneck in the system is one of the machines connected to the network. For example, when it takes a long time for a web page to appear on your browser, it might be network congestion, but it's just as likely the case that the server at

the other end of the network—which may be trying to serve many users at the same time as you—can't keep up with the workload.

2.1.2 Links

Network links are implemented on a variety of different physical media, including twisted pair (the wire that your phone connects to), coaxial cable (the wire that your TV connects to), optical fiber (the medium most commonly used for high-bandwidth, long-distance links), and space (the stuff that radio waves, microwaves, and infrared beams propagate through). Whatever the physical medium, it is used to propagate *signals*. These signals are actually electromagnetic waves traveling at the speed of light. (The speed of light is, however, medium dependent—electromagnetic waves traveling through copper and fiber do so at about two-thirds the speed of light in a vacuum.)

One important property of an electromagnetic wave is the *frequency*, measured in hertz, with which the wave oscillates. The distance between a pair of adjacent maxima or minima of a wave, typically measured in meters, is called the wave's *wavelength*. Since all electromagnetic waves travel at the speed of light, that speed divided by the wave's frequency is equal to its wavelength. We have already seen the example of a voice-grade telephone line, which carries continuous electromagnetic signals ranging between 300 and 3,300 Hz; a 300-Hz wave traveling through copper would have a wavelength of

$$\begin{aligned}&\mathsf{SpeedOfLightInCopper} \div \mathsf{Frequency}\\ &= 2/3 \times 3 \times 10^8 \div 300\\ &= 667 \times 10^3 \text{ meters}\end{aligned}$$

Generally, electromagnetic waves span a much wider range of frequencies, ranging from radio waves, to infrared light, to visible light, to X-rays and gamma rays. Figure 2.4 depicts the electromagnetic spectrum and shows which media are commonly used to carry which frequency bands.

So far we understand a link to be a physical medium carrying signals in the form of electromagnetic waves. Such links provide the foundation for transmitting all sorts of information, including the kind of data we are interested in transmitting—binary data (1s and 0s). We say that the binary data is *encoded* in the signal. The problem of encoding binary data onto electromagnetic signals is a complex topic. To help make the topic more manageable, we can think of it as being divided into two layers. The lower layer is concerned with *modulation*—varying the frequency, amplitude, or phase of the signal to effect the transmission of information. A simple example of modulation is to vary the power (amplitude) of a single wavelength. Intuitively, this is equivalent to turning a light on and off. Because the issue of modulation is secondary to our discussion of links as a

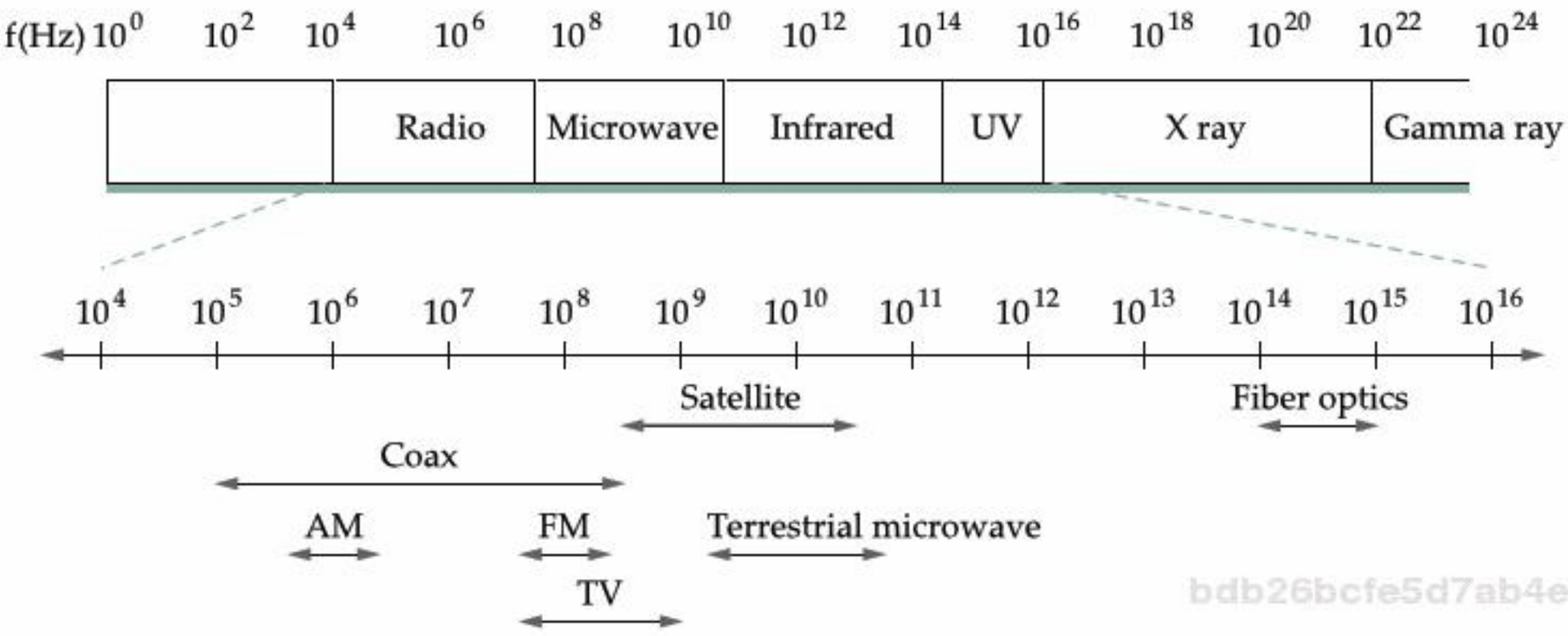

Figure 2.4 Electromagnetic spectrum.

building block for computer networks, we simply assume that it is possible to transmit a pair of distinguishable signals—think of them as a "high" signal and a "low" signal—and we consider only the upper layer, which is concerned with the much simpler problem of encoding binary data onto these two signals. Section 2.2 discusses such encodings.

Another attribute of a link is how many bitstreams can be encoded on it at a given time. If the answer is only one, then the nodes connected to the link must share access to the link. This is the case for the multiple-access links described in Sections 2.6 and 2.7. For point-to-point links, however, it is often the case that two bitstreams can be simultaneously transmitted over the link at the same time, one going in each direction. Such a link is said to be *full-duplex*. A point-to-point link that supports data flowing in only one direction at a time—such a link is called *half-duplex*—requires that the two nodes connected to the link alternate using it. For the purposes of this book, we assume that all point-to-point links are full-duplex.

The only other property of a link that we are interested in at this stage is a very pragmatic one—how do you go about getting one? The answer depends on how far the link needs to reach, how much money you have to spend, and whether or not you know how to operate earth-moving equipment. The following is a survey of different link types you might use to build a computer network.

Cables

If the nodes you want to connect are in the same room, in the same building, or even on the same site (e.g., a campus), then you can buy a piece of cable and physically string it between the nodes. Exactly what type of cable you choose to install depends on the technology you plan to use to transmit data over the link; we'll see several examples later in this chapter. For now, a list of the common cable (fiber) types is given in Table 2.1.

Cable	Typical Bandwidths	Distances
Category 5 twisted pair	10–100 Mbps	100m
Thin-net coax	10–100 Mbps	200m
Thick-net coax	10–100 Mbps	500m
Multimode fiber	100 Mbps	2 km
Single-mode fiber	0.1–10 Gbps	40 km

Table 2.1 Common types of cables and fibers available for local links.

Service	Bandwidth
DS1	1.544 Mbps
DS3	44.736 Mbps
STS-1	51.840 Mbps
STS-3	155.250 Mbps
STS-12	622.080 Mbps
STS-24	1.244160 Gbps
STS-48	2.488320 Gbps

Table 2.2 Common bandwidths available from the carriers.

Of these, Category 5 (Cat-5) twisted pair—it uses a thicker gauge than the twisted pair you find in your home—is quickly becoming the within-building norm. Because of the difficulty and cost in pulling new cable through a building, every effort is made to make new technologies use existing cable; Gigabit Ethernet, for example, has been designed to run over Cat-5 wiring. Fiber is typically used to connect buildings at a site.

Leased Lines

If the two nodes you want to connect are on opposite sides of the country, or even across town, then it is not practical to install the link yourself. Until recently, your only option was to lease a dedicated link from the telephone company. Table 2.2 gives the common "leased line" services that can be obtained from the average phone company. Again, more details are given throughout this chapter.

While these bandwidths appear somewhat arbitrary, there is actually some method to the madness. DS1 and DS3 (they are also sometimes called T1 and T3, respectively) are relatively old technologies that were originally defined for copper-based transmission

media. DS1 is equal to the aggregation of 24 digital voice circuits of 64 Kbps each, and DS3 is equal to 28 DS1 links. All the STS-*N* links are for optical fiber (STS stands for Synchronous Transport Signal). STS-1 is the base link speed, and each STS-*N* has *N* times the bandwidth of STS-1. An STS-*N* link is also sometimes called an OC-*N* link (OC stands for optical carrier). The difference between STS and OC is subtle: The former refers to the *electrical* transmission on the devices connected to the link, and the latter refers to the actual *optical* signal that is propagated over the fiber.

More recently, many providers, both traditional telephone companies and some upstart competitors, have started to offer a range of alternatives to leased lines based on Ethernet, a technology we will discuss in detail in Section 2.6. One consequence of this development is a proliferation of different access speeds beyond those in Table 2.2.

Keep in mind that the phone company does not implement the "link" we just ordered as a single, unbroken piece of cable or fiber. Instead, it implements the link on its own network. Although the telephone network has historically looked much different from the kind of network described in this book—it was built primarily to provide a voice service and used circuit-switching technology—the current trend is toward the style of packet-switched networking described in this book. This is not surprising—the potential market for carrying data, voice, and video on one packet-switched network is huge.

In any case, whether the link is physical or a logical connection through the telephone network, the problem of building a computer network on top of a collection of such links remains the same. So, we will proceed as though each link is implemented by a single cable/fiber, and only when we are done will we worry about whether we have just built a computer network on top of the underlying telephone network, or the computer network we have just built could itself serve as the backbone for the telephone network.

Last-Mile Links

If you can't afford a dedicated leased line—they range in price from several hundred dollars a month for a DS1 link across the United States to "if you have to ask, you can't afford it"—then there are less expensive options available. We call these "last-mile" links because they often span the last mile from the home to a network service provider. These services, which are summarized in Table 2.3, typically connect a home to an existing network. This means they are probably not suitable for use in building a complete network from scratch, but if you've already succeeded in building a network—and "you" happen to be either the telephone company or the cable company—then you can use these links to reach millions of customers.

The first option is a conventional modem over POTS. Today it is possible to buy a modem that transmits data at 56 Kbps over a standard voice-grade line for less than a hundred dollars. The technology is already at its bandwidth limit, however, which

Service	Bandwidth
POTS	28.8–56 Kbps
ISDN	64–128 Kbps
xDSL	128 Kbps–100 Mbps
CATV	1–40 Mbps

Table 2.3 Common services available to connect your home.

led to the development of the second option: Integrated Services Digital Network (ISDN). An ISDN connection includes two 64-Kbps channels, one that can be used to transmit data and another that can be used for digitized voice. (A device that encodes analog voice into a digital ISDN link is called a CODEC, for *coder/decoder*.) When the voice channel is not in use, it can be combined with the data channel to support up to 128 Kbps of data bandwidth.

For many years ISDN was viewed as the future for modest bandwidth into the home. ISDN, however, has now been largely overtaken by two newer technologies: digital subscriber line (xDSL) and cable modems. The former is actually a collection of technologies that are able to transmit data at high speeds over the standard twisted pair lines that currently come into most homes in the United States (and many other places). The one in most widespread use today is asymmetric digital subscriber line (ADSL). As its name implies, ADSL provides a different bandwidth from the subscriber to the telephone company's central office (upstream) than it does from the central office to the subscriber (downstream). The exact bandwidth depends on the length of the line running from the subscriber to the central office. This line is called the *local loop*, as illustrated in Figure 2.5, and runs over

Shannon's Theorem Meets Your Modem

There has been an enormous body of work done in the related areas of signal processing and information theory, studying everything from how signals degrade over distance to how much data a given signal can effectively carry. The most notable piece of work in this area is a formula known as *Shannon's theorem*. Simply stated, Shannon's theorem gives an upper bound to the capacity of a link, in terms of bits per second (bps), as a function of the signal-to-noise ratio of the link, measured in decibels (dB).

Shannon's theorem can be used to determine the data rate at which a modem can be expected to transmit binary data over a voice-grade phone line without suffering from too high an error rate. For example, we assume that a voice-grade phone connection

existing copper. Downstream bandwidths range from 1.544 Mbps (18,000 feet) to 8.448 Mbps (9,000 feet), while upstream bandwidths range from 16 to 640 Kbps.

An alternative technology that has yet to be widely deployed—very high data rate digital subscriber line (VDSL)—is symmetric, with data rates ranging from 12.96 to 55.2 Mbps. VDSL runs over much shorter distances—1,000 to 4,500 feet—which means that it will not typically reach from the home to the central office. Instead, the telephone company would have to put VDSL transmission hardware in neighborhoods, with some other technology (e.g., STS-N running over fiber) connecting the neighborhood to the central office, as illustrated in Figure 2.6. This is sometimes called "fiber to the neighborhood" (contrasting with more ambitious schemes such as "fiber to the home" and "fiber to the curb").

Cable modems are an alternative to the various types of DSL. As the name suggests, this technology uses the cable TV (CATV) infrastructure, which currently reaches 95% of the households in the United States. (Only 65% of U.S. homes actually subscribe.) In this approach, some subset of the available CATV channels are made available for transmitting digital data, where a single CATV channel has a bandwidth of 6 MHz. CATV, like ADSL, is used in an asymmetric way, with downstream rates much greater than upstream rates. The technology is currently able to achieve 40 Mbps downstream on a single CATV channel, with 100 Mbps as the theoretical capacity. The upstream rate

supports a frequency range of 300 to 3,300 Hz.

Shannon's theorem is typically given by the following formula:

$$C = B\log_2(1 + S/N)$$

where C is the achievable channel capacity measured in hertz, B is the bandwidth of the line (3,300 Hz − 300 Hz = 3,000 Hz), S is the average signal power, and N is the average noise power. The signal-to-noise ratio (S/N) is usually expressed in decibels, related as follows:

$$dB = 10 \times \log_{10}(S/N)$$

Assuming a typical decibel ratio of 30 dB, this means that $S/N = 1{,}000$. Thus, we have

$$C = 3{,}000 \times \log_2(1001)$$

which equals approximately 30 Kbps, roughly the limit of a 28.8-Kbps modem.

Given this fundamental limit, why is it possible to buy 56-Kbps modems at any electronics store? One reason is that such rates depend on improved line quality, that is, a higher signal-to-noise ratio than 30 dB. Another reason is that changes within the phone system have largely eliminated analog lines that are bandwidth limited to 3,300 Hz.

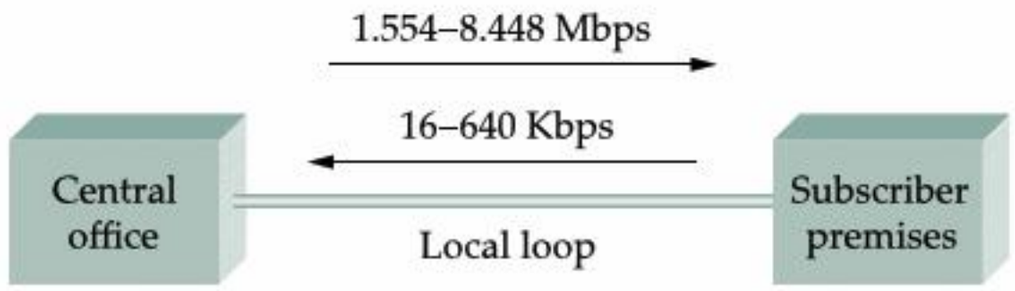

Figure 2.5 ADSL connects the subscriber to the central office via the local loop.

Figure 2.6 VDSL connects the subscriber to the optical network that reaches the neighborhood.

is roughly half the downstream rate (i.e., 20 Mbps) due to a 1,000-fold decrease in the signal-to-noise ratio. It is also the case that fewer CATV channels are dedicated to upstream traffic than to downstream traffic. Unlike DSL, the bandwidth is shared among all subscribers in a neighborhood (a fact that led to some amusing advertising from DSL providers). This means that some method for arbitrating access to the shared medium—similar to the 802 standards described later in this chapter—needs to be used. Finally, like DSL, it is unlikely that cable modems will be used to connect arbitrary node A at one site to arbitrary node B at some other site. Instead, cable modems are seen as a means to connect node A in your home to the cable company, with the cable company then defining what the rest of the network looks like.

Wireless Links

Up to this point we have discussed links that channel signals through a physical medium like a wire or optical fiber. Wireless links transmit electromagnetic signals—radio, microwave, infrared, or even visible light—through space, even through vacuum. Wireless communication is used not only by computer networks, of course. The ability to exchange signals without physical connectivity is what makes mobile communication devices possible. The further ability to broadcast that signal, and the fact that the hardware and power burden is primarily on the transmitting end, makes wireless communication well-suited to television and radio broadcasting.

Because wireless links all share the same wire, so to speak, the challenge is to share it efficiently, without unduly interfering with each other. Most of this sharing is accomplished by dividing the "wire" along the dimensions of frequency and space. Exclusive use of a particular frequency in a particular geographic area may be allocated to an individual

entity such as a corporation. It is feasible to limit the area covered by an electromagnetic signal because such signals weaken, or *attenuate*, with the distance from their origin. To reduce the area covered by your signal, reduce the power of your transmitter.

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

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

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

A second spread spectrum technique, called *direct sequence*, adds redundancy for greater tolerance of interference. Each bit of data is represented by multiple bits in the transmitted signal so that, if some of the transmitted bits are damaged by interference, there is usually enough redundancy to recover the original bit. For each bit the sender wants to transmit, it actually sends the exclusive-OR of that bit and n random bits. As with frequency hopping, the sequence of random bits is generated by a pseudorandom number generator known to both the sender and the receiver. The transmitted values, known as an *n-bit chipping code*, spread the signal across a frequency band that is n times wider than the frame would have otherwise required. Figure 2.7 gives an example of a 4-bit chipping sequence.

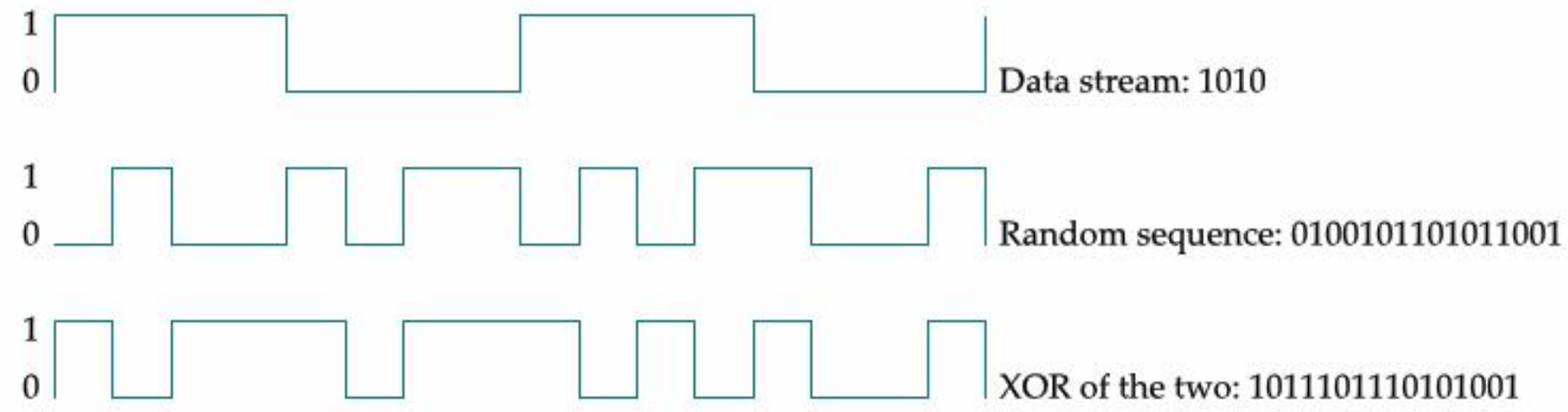

Figure 2.7 Example 4-bit chipping sequence.

It is these license-exempt frequencies, with their spread spectrum techniques and limited range, that are used by 802.11 (Wi-Fi), 802.15.1 (Bluetooth), and some flavors of 802.16 (WiMAX). These technologies are discussed further in Section 2.8.

Different parts of the electromagnetic spectrum have different properties, making some better suited to communication, and some less so. For example, some can penetrate buildings and some cannot. Governments regulate only the prime communication portion: the radio and microwave ranges. As demand for prime spectrum increases, there is great interest in the spectrum that will become available when analog television is phased out in favor of digital. There is also an effort to devise a wireless network technology that could squeeze into the currently unused frequencies that separate television channels without interfering with them.

Among the unregulated spectra are the infrared and visual light ranges, which cannot penetrate walls. Infrared is widely used in remote controls for television and the like. Increasingly, it is also used in a variety of short-range data exchange applications, based on standards established by the Infrared Data Association (IrDA). For example, IrDA has defined a standard for "Point & Pay," using infrared to conduct a financial transaction between a handheld device, such as a mobile phone or PDA, and a stationary financial terminal such as a cash register. Visual light or infrared can also be focused by a laser to provide a high-bandwidth link between two stationary points—even though no mobility is involved—in situations where a wired link is less practical for some reason. For example, two buildings belonging to the same organization but separated by a busy highway could communicate using lasers.

2.2 Encoding (NRZ, NRZI, Manchester, 4B/5B)

The first step in turning nodes and links into usable building blocks is to understand how to connect them in such a way that bits can be transmitted from one node to another. As mentioned in the preceding section, signals propagate over physical links. The task,

therefore, is to encode the binary data that the source node wants to send into the signals that the links are able to carry, and then to decode the signal back into the corresponding binary data at the receiving node. We ignore the details of modulation and assume we are working with two discrete signals: high and low. In practice, these signals might correspond to two different voltages on a copper-based link, or two different power levels on an optical link.

As we have said, most of the functions discussed in this chapter are performed by a network adaptor—a piece of hardware that connects a node to a link. The network adaptor contains a signalling component that actually encodes bits into signals at the sending node and decodes signals into bits at the receiving node. Thus, as illustrated in Figure 2.8, signals travel over a link between two signalling components, and bits flow between network adaptors.

Let's return to the problem of encoding bits onto signals. The obvious thing to do is to map the data value 1 onto the high signal and the data value 0 onto the low signal. This is exactly the mapping used by an encoding scheme called, cryptically enough, *nonreturn to zero* (NRZ). For example, Figure 2.9 schematically depicts the NRZ-encoded signal (bottom) that corresponds to the transmission of a particular sequence of bits (top).

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

The second problem is that frequent transitions from high to low and vice versa are necessary to enable *clock recovery*. Intuitively, the clock recovery problem is that both the encoding and the decoding processes are driven by a clock—every clock cycle the sender transmits a bit and

Bit Rates and Baud Rates

Many people use the terms *bit rate* and *baud rate* interchangeably, even though as we see with the Manchester encoding, they are not the same thing. While the Manchester encoding is an example of a case in which a link's baud rate is greater than its bit rate, it is also possible to have a bit rate that is greater than the baud rate. This would imply that more than one bit is encoded on each pulse sent over the link.

To see how this might happen, suppose you could transmit four

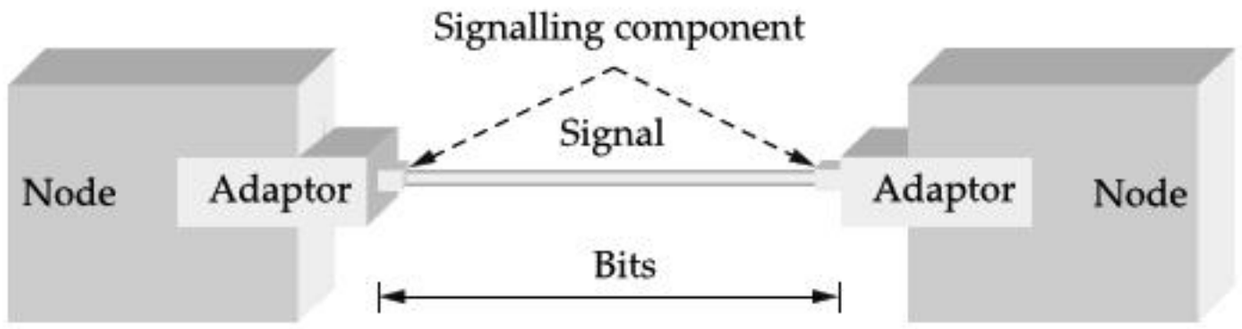

Figure 2.8 Signals travel between signalling components; bits flow between adaptors.

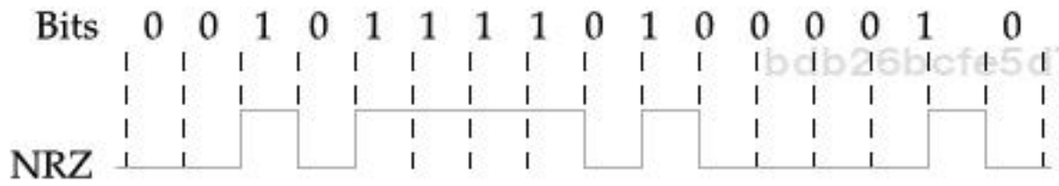

Figure 2.9 NRZ encoding of a bitstream.

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

One approach that addresses this problem, called *nonreturn to zero inverted (NRZI)*, has the sender make a transition from the current signal to encode a 1 and stay at the current signal to encode a 0.

distinguished signals over a link rather than just two. On an analog link, for example, these four signals might correspond to four different frequencies. Given four different signals, it is possible to encode two bits of information on each signal. That is, the first signal means 00, the second signal means 01, and so on. Now, a sender (receiver) that is able to transmit (detect) 1,000 pulses per second would be able to send (receive) 2,000 bits of information per second. That is, it would be a 1,000-baud/2,000-bps link.

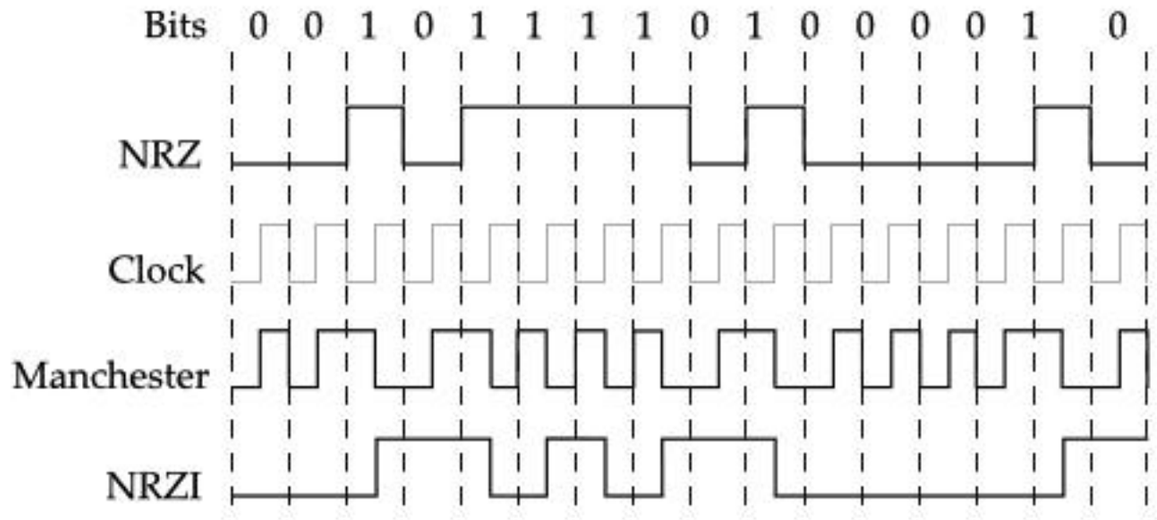

Figure 2.10 Different encoding strategies.

This solves the problem of consecutive 1s, but obviously does nothing for consecutive 0s. NRZI is illustrated in Figure 2.10. An alternative, called *Manchester encoding*, does a more explicit job of merging the clock with the signal by transmitting the exclusive-OR of the NRZ-encoded data and the clock. (Think of the local clock as an internal signal that alternates from low to high; a low/high pair is considered one clock cycle.) The Manchester encoding is also illustrated in Figure 2.10. Observe that the Manchester encoding results in 0 being encoded as a low-to-high transition and 1 being encoded as a high-to-low transition. Because both 0s and 1s result in a transition to the signal, the clock can be effectively recovered at the receiver. (There is also a variant of the Manchester encoding, called *differential Manchester*, in which a 1 is encoded with the first half of the signal equal to the last half of the previous bit's signal and a 0 is encoded with the first half of the signal opposite to the last half of the previous bit's signal.)

The problem with the Manchester encoding scheme is that it doubles the rate at which signal transitions are made on the link, which means that the receiver has half the time to detect each pulse of the signal. The rate at which the signal changes is called the link's *baud rate*. In the case of the Manchester encoding, the bit rate is half the baud rate, so the encoding is considered only 50% efficient. Keep in mind that if the receiver had been able to keep up with the faster baud rate required by the Manchester encoding in Figure 2.10, then both NRZ and NRZI could have been able to transmit twice as many bits in the same time period.

A final encoding that we consider, called *4B/5B*, attempts to address the inefficiency of the Manchester encoding without suffering from the problem of having extended durations of high or low signals. The idea of 4B/5B is to insert extra bits into the bitstream so as to break up long sequences of 0s or 1s. Specifically, every 4 bits of actual data are encoded in a 5-bit code that is then transmitted to the receiver; hence the name 4B/5B. The 5-bit codes are selected in such a way that each one has no more than one leading 0

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

Table 2.4 4B/5B encoding.

and no more than two trailing 0s. Thus, when sent back-to-back, no pair of 5-bit codes results in more than three consecutive 0s being transmitted. The resulting 5-bit codes are then transmitted using the NRZI encoding, which explains why the code is only concerned about consecutive 0s—NRZI already solves the problem of consecutive 1s. Note that the 4B/5B encoding results in 80% efficiency.

Table 2.4 gives the 5-bit codes that correspond to each of the 16 possible 4-bit data symbols. Notice that since 5 bits are enough to encode 32 different codes, and we are using only 16 of these for data, there are 16 codes left over that we can use for other purposes. Of these, code **11111** is used when the line is idle, code **00000** corresponds to when the line is dead, and **00100** is interpreted to mean halt. Of the remaining 13 codes, 7 of them are not valid because they violate the "one leading 0, two trailing 0s," rule, and the other 6 represent various control symbols. As we will see later in this chapter, some framing protocols (e.g., FDDI) make use of these control symbols.

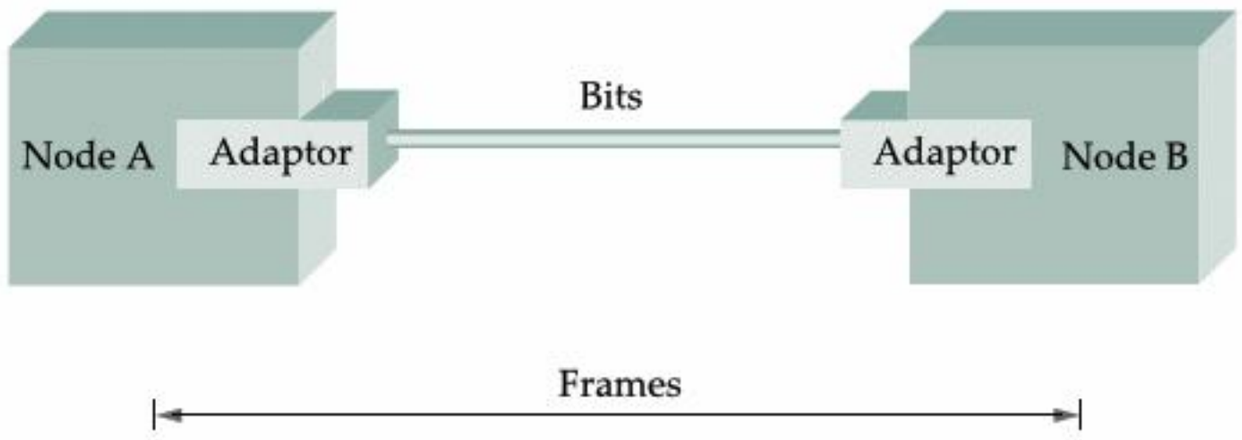

Figure 2.11 Bits flow between adaptors, frames between hosts.

2.3 Framing

Now that we have seen how to transmit a sequence of bits over a point-to-point link—from adaptor to adaptor—let's consider the scenario illustrated in Figure 2.11. Recall from Chapter 1 that we are focusing on packet-switched networks, which means that blocks of data (called frames at this level), not bitstreams, are exchanged between nodes. It is the network adaptor that enables the nodes to exchange frames. When node A wishes to transmit a frame to node B, it tells its adaptor to transmit a frame from the node's memory. This results in a sequence of bits being sent over the link. The adaptor on node B then collects together the sequence of bits arriving on the link and deposits the corresponding frame in B's memory. Recognizing exactly what set of bits constitute a frame—that is, determining where the frame begins and ends—is the central challenge faced by the adaptor.

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

2.3.1 Byte-Oriented Protocols (PPP)

One of the oldest approaches to framing—it has its roots in connecting terminals to mainframes—is to view each frame as a collection of bytes (characters) rather than a collection of bits. Such a *byte-oriented* approach is exemplified by older protocols such as the Binary Synchronous Communication (BISYNC) protocol developed by IBM in the late 1960s, and the Digital Data Communication Message Protocol (DDCMP) used in Digital Equipment Corporation's DECNET. The more recent and widely used Point-to-Point Protocol (PPP) provides another example of this approach.

Sentinel-based Approaches

Figure 2.12 illustrates the BISYNC protocol's frame format. This figure is the first of many that you will see in this book that are used to illustrate frame or packet formats,

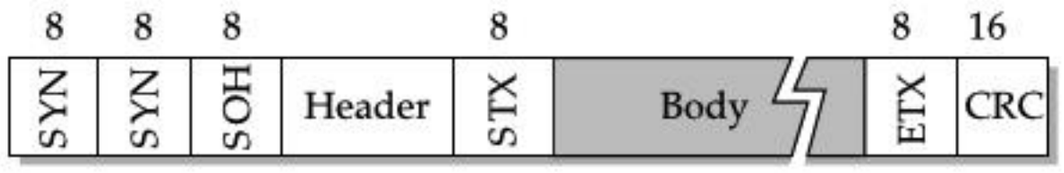

Figure 2.12 BISYNC frame format.

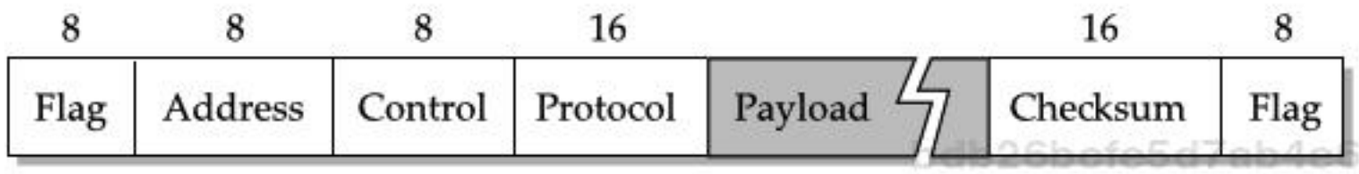

Figure 2.13 PPP frame format.

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

BISYNC uses special characters known as *sentinel characters* to indicate where frames start and end. The beginning of a frame is denoted by sending a special SYN (synchronization) character. The data portion of the frame is then contained between two more special characters: STX (start of text) and ETX (end of text). The SOH (start of header) field serves much the same purpose as the STX field. The problem with the sentinel approach, of course, is that the ETX character might appear in the data portion of the frame. BISYNC overcomes this problem by "escaping" the ETX character by preceding it with a data-link-escape (DLE) character whenever it appears in the body of a frame; the DLE character is also escaped (by preceding it with an extra DLE) in the frame body. (C programmers may notice that this is analogous to the way a quotation mark is escaped by the backslash when it occurs inside a string.) This approach is often called *character stuffing* because extra characters are inserted in the data portion of the frame.

The frame format also includes a field labeled cyclic redundancy check (CRC) that is used to detect transmission errors; various algorithms for error detection are presented in Section 2.4. Finally, the frame contains additional header fields that are used for, among other things, the link-level reliable delivery algorithm. Examples of these algorithms are given in Section 2.5.

The more recent PPP, which is commonly used to carry Internet Protocol packets over various sorts of point-to-point links, is similar to BISYNC in that it also uses sentinels and character stuffing. The format for a PPP frame is given in Figure 2.13.

The special start-of-text character, denoted as the Flag field in Figure 2.13, is 01111110. The Address and Control fields usually contain default values, and so are uninteresting. The Protocol field is used for demultiplexing: it identifies the high-level protocol such as IP or IPX (an IP-like protocol developed by Novell). The frame payload size can be negotiated, but it is 1,500 bytes by default. The Checksum field is either 2 (by default) or 4 bytes long.

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

Byte-Counting Approach

As every Computer Sciences 101 student knows, the alternative to detecting the end of a file with a sentinel value is to include the number of items in the file at the beginning of the file. The same is true in framing—the number of bytes contained in a frame can be included

What's in a Layer?

One of the important contributions of the OSI reference model presented in Chapter 1 was to provide some vocabulary for talking about protocols and, in particular, protocol layers. This vocabulary has provided fuel for plenty of arguments along the lines of "Your protocol does function X at layer Y, and the OSI reference model says it should be done at layer Z—that's a layer violation." In fact, figuring out the right layer at which to perform a given function can be very difficult, and the reasoning is usually a lot more subtle than "What does the OSI model say?" It is partly for this reason that this book avoids a rigidly layerist approach. Instead, it shows you a lot of functions that need to be performed by protocols and looks at some ways that they have been successfully implemented.

In spite of our nonlayerist approach, sometimes we need convenient ways to talk about classes of protocols, and the name of the layer at which they operate is often the best choice. Thus, for example, this chapter focuses primarily on link-layer protocols. (Bit encoding, described in Section 2.2, is the exception, being considered a physical-layer function.) Link-layer protocols can be identified by the fact that they run over single links—the type of network discussed in this chapter. Network-layer

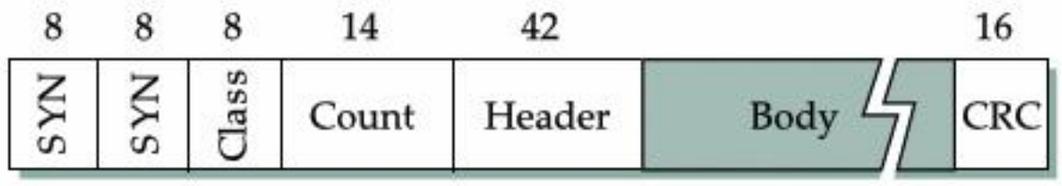

Figure 2.14 DDCMP frame format.

protocols, by contrast, run over switched networks that contain lots of links interconnected by switches or routers. Topics related to network-layer protocols are discussed in Chapters 3 and 4.

Note that protocol layers are supposed to be helpful—they provide helpful ways to talk about classes of protocols, and they help us divide the problem of building networks into manageable subtasks. However, they are not meant to be overly restrictive—the mere fact that something is a layer violation does not end the argument about whether it is a worthwhile thing to do. In other words, layering makes a good slave, but a poor master. A particularly interesting argument about the best layer in which to place a certain function comes up when we look at congestion control in Chapter 6.

as a field in the frame header. The DECNET's DDCMP protocol uses this approach, as illustrated in Figure 2.14. In this example, the COUNT field specifies how many bytes are contained in the frame's body.

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

2.3.2 Bit-Oriented Protocols (HDLC)

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

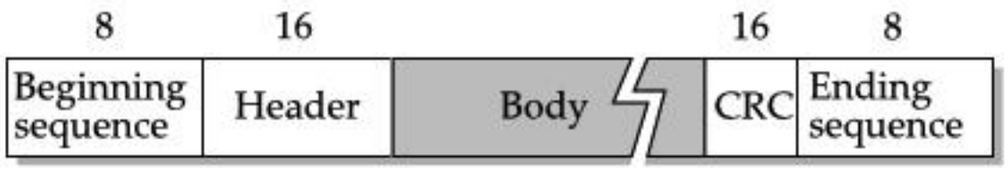

Figure 2.15 HDLC frame format.

protocol. In the following discussion, we use HDLC as an example; its frame format is given in Figure 2.15.

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

Bit stuffing in the HDLC protocol works as follows. On the sending side, any time five consecutive 1s have been transmitted from the body of the message (i.e., excluding when the sender is trying to transmit the distinguished 01111110 sequence), the sender inserts a 0 before transmitting the next bit. On the receiving side, should five consecutive 1s arrive, the receiver makes its decision based on the next bit it sees (i.e., the bit following the five 1s). If the next bit is a 0, it must have been stuffed, and so the receiver removes it. If the next bit is a 1, then one of two things is true: Either this is the end-of-frame marker or an error has been introduced into the bitstream. By looking at the *next* bit, the receiver can distinguish between these two cases: if it sees a 0 (i.e., the last eight bits it has looked at are 01111110), then it is the end-of-frame marker; if it sees a 1 (i.e., the last eight bits it has looked at are 01111111), then there must have been an error and the whole frame is discarded. In the latter case, the receiver has to wait for the next 01111110 before it can start receiving again, and as a consequence, there is the potential that the receiver will fail to receive two consecutive frames. Obviously, there are still ways that framing errors can go undetected, such as when an entire spurious end-of-frame pattern is generated by errors, but these failures are relatively unlikely. Robust ways of detecting errors are discussed in Section 2.4.

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

2.3.3 Clock-Based Framing (SONET)

A third approach to framing is exemplified by the Synchronous Optical Network (SONET) standard. For lack of a widely accepted generic term, we refer to this approach simply as *clock-based framing*. SONET was first proposed by Bell Communications Research (Bellcore), and then developed under the American National Standards Institute (ANSI) for digital transmission over optical fiber; it has since been adopted by the ITU-T. Who standardized what and when is not the interesting issue, though. The thing to remember about SONET is that it is the dominant standard for long-distance transmission of data over optical networks.

An important point to make about SONET before we go any further is that the full specification is substantially larger than this book. Thus, the following discussion will necessarily cover only the high points of the standard. Also, SONET addresses both the framing problem and the encoding problem. It also addresses a problem that is very important for phone companies—the multiplexing of several low-speed links onto one high-speed link. We begin with framing and discuss the other issues following.

As with the previously discussed framing schemes, a SONET frame has some special information that tells the receiver where the frame starts and ends. However, that is about as far as the similarities go. Notably, no bit stuffing is used, so that a frame's length does not depend on the data being sent. So the question to ask is, "How does the receiver know where each frame starts and ends?" We consider this question for the lowest-speed SONET link, which is known as STS-1 and runs at 51.84 Mbps. An STS-1 frame is shown in Figure 2.16. It is arranged as nine rows of 90 bytes each, and the first 3 bytes of each row are overhead, with the rest being available for data that is being transmitted over the link. The first 2 bytes of the frame contain a special bit pattern, and it is these bytes that enable the receiver to determine where the frame starts. However, since bit stuffing is not used, there is no reason why this pattern will not occasionally turn up in the payload portion of the frame. To guard against this, the receiver looks for the special bit pattern consistently, hoping to see it appearing once every 810 bytes, since each frame is $9 \times 90 = 810$ bytes long. When the special pattern turns up in the right place enough times, the receiver concludes that it is in sync and can then interpret the frame correctly.

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

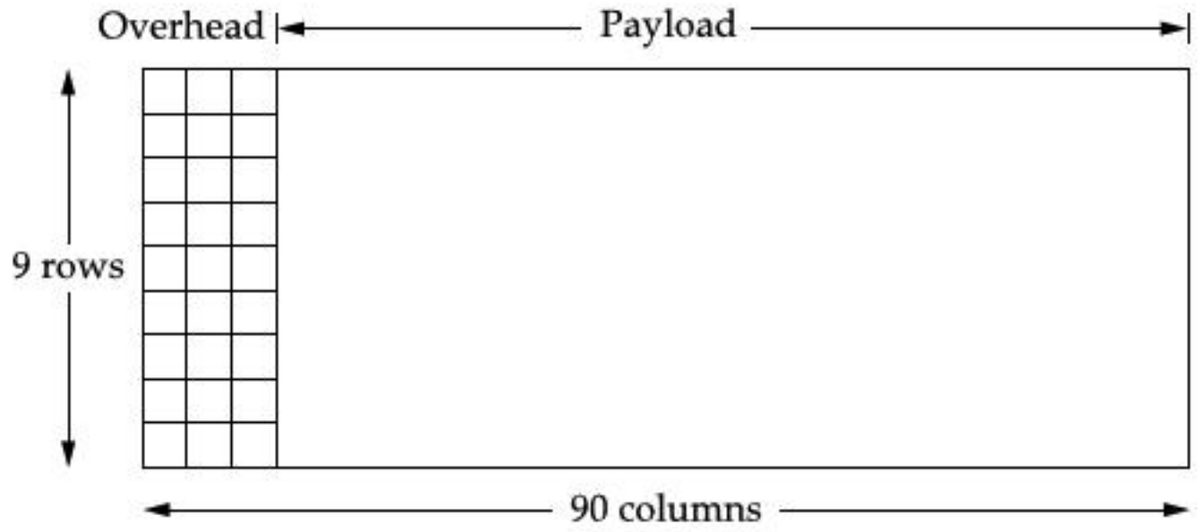

Figure 2.16 A SONET STS-1 frame.

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

SONET supports the multiplexing of multiple low-speed links in the following way. A given SONET link runs at one of a finite set of possible rates, ranging from 51.84 Mbps (STS-1) to 2,488.32 Mbps (STS-48), and beyond. (See Table 2.2 in Section 2.1 for the full set of SONET data rates.) Note that all of these rates are integer multiples of STS-1. The significance for framing is that a single SONET frame can contain subframes for multiple lower-rate channels. A second related feature is that each frame is 125 μs long. This means that at STS-1 rates, a SONET frame is 810 bytes long, while at STS-3 rates, each SONET frame is 2,430 bytes long. Notice the synergy between these two features: $3 \times 810 = 2{,}430$, meaning that three STS-1 frames fit exactly in a single STS-3 frame.

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

Although it is accurate to view an STS-N signal as being used to multiplex N STS-1 frames, the payload from these STS-1 frames can be linked together to form a larger STS-N payload; such a link is denoted STS-Nc (for *concatenated*). One of the

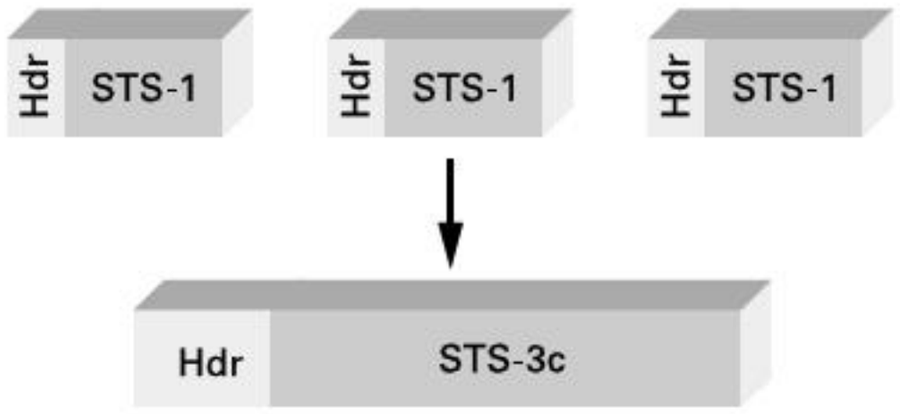

Figure 2.17 Three STS-1 frames multiplexed onto one STS-3c frame.

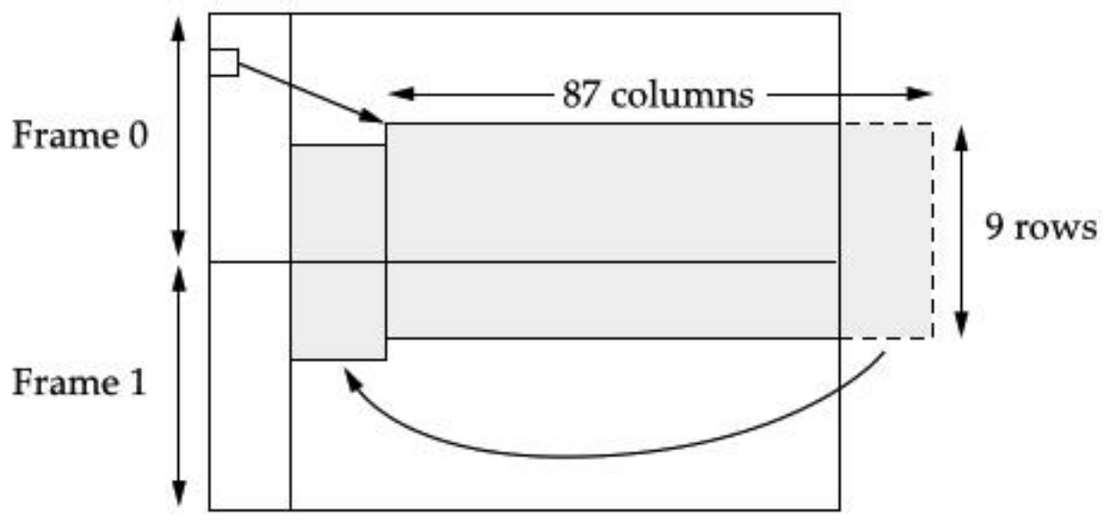

Figure 2.18 SONET frames out of phase.

fields in the overhead is used for this purpose. Figure 2.17 schematically depicts concatenation in the case of three STS-1 frames being concatenated into a single STS-3c frame. The significance of a SONET link being designated as STS-3c rather than STS-3 is that, in the former case, the user of the link can view it as a single 155.25-Mbps pipe, whereas an STS-3 should really be viewed as three 51.84-Mbps links that happen to share a fiber.

Finally, the preceding description of SONET is overly simplistic in that it assumes that the payload for each frame is completely contained within the frame. (Why wouldn't it be?) In fact, we should view the STS-1 frame just described as simply a placeholder for the frame, where the actual payload may *float* across frame boundaries. This situation is illustrated in Figure 2.18. Here we see both the STS-1 payload floating across two STS-1 frames, and the payload shifted some number of bytes to the right and, therefore, wrapped around. One of the fields in the frame overhead points to the beginning of the payload. The value of this capability is that it simplifies the task of synchronizing the clocks used throughout the carriers' networks, which is something that carriers spend a lot of their time worrying about.

2.4 Error Detection

As discussed in Chapter 1, bit errors are sometimes introduced into frames. This happens, for example, because of electrical interference or thermal noise. Although errors are rare, especially on optical links, some mechanism is needed to detect these errors so that corrective action can be taken. Otherwise, the end user is left wondering why the C program that successfully compiled just a moment ago now suddenly has a syntax error in it, when all that happened in the interim is that it was copied across a network file system.

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

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

One of the most common techniques for detecting transmission errors is a technique known as the *cyclic redundancy check (CRC)*. It is used in nearly all the link-level protocols discussed in the previous section—for example, HDLC, DDCMP—as well as in the CSMA and token ring protocols described later in this chapter. Section 2.4.3 outlines the basic CRC algorithm. Before discussing that approach, we consider two simpler schemes that are also widely used: *two-dimensional parity* and *checksums*. The former is used by the BISYNC protocol when it is transmitting ASCII characters (CRC is used as the error code when BISYNC is used to transmit EBCDIC), and the latter is used by several Internet protocols.

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

Fortunately, we can do a lot better than this simple scheme. In general, we can provide quite strong error detection capability while sending only k redundant bits for

an n-bit message, where $k \ll n$. On an Ethernet, for example, a frame carrying up to 12,000 bits (1,500 bytes) of data requires only a 32-bit CRC code, or as it is commonly expressed, uses CRC-32. Such a code will catch the overwhelming majority of errors, as we will see below.

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

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

2.4.1 Two-Dimensional Parity

Two-dimensional parity is exactly what the name suggests. It is based on "simple" (one-dimensional) parity, which usually involves adding one extra bit to a 7-bit code to balance the number of 1s in the byte. For example, odd parity sets the eighth bit to 1 if needed to give an odd number of 1s in the byte, and even parity sets the eighth bit to 1 if needed to give an even number of 1s in the byte. Two-dimensional parity does a similar calculation for each bit position across each of the bytes contained in the frame. This results in an extra parity byte for the entire frame, in addition to a parity bit for each byte. Figure 2.19 illustrates how two-dimensional even parity works for an example frame containing 6 bytes of data. Notice that the third bit of the parity byte is 1 since there is an odd number of 1s in the third bit across the 6 bytes in the frame. It can be shown that two-dimensional parity catches all 1-, 2-, and 3-bit errors, and most 4-bit errors. In this case, we have added 14 bits of redundant information to a 42-bit message, and yet we have stronger protection against common errors than the "repetition code" described above.

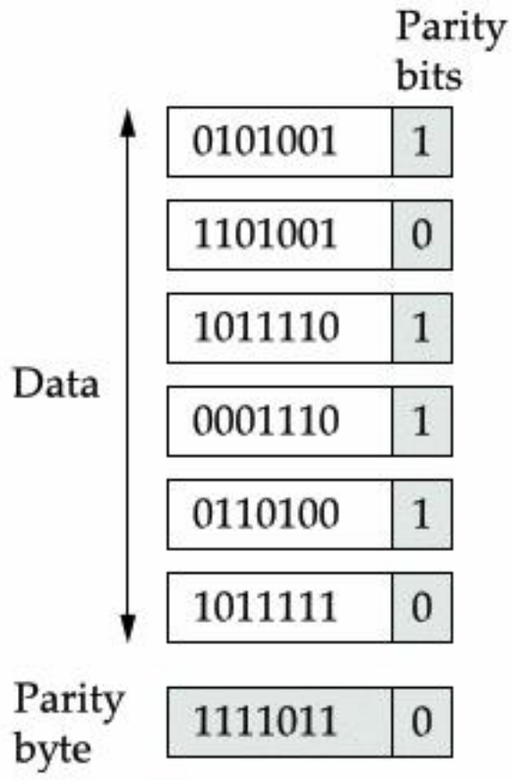

Figure 2.19 Two-dimensional parity.

2.4.2 Internet Checksum Algorithm

A second approach to error detection is exemplified by the Internet checksum. Although it is not used at the link level, it nevertheless provides the same sort of functionality as CRCs and parity, so we discuss it here. We will see examples of its use in Sections 4.1, 5.1, and 5.2.

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

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

In ones complement arithmetic, a negative integer $-x$ is represented as the complement of x, that is, each bit of x is inverted. When adding numbers in ones complement arithmetic, a carryout from the most significant bit needs to be added to the result. Consider, for example, the addition of -5 and -3 in ones complement arithmetic on 4-bit integers: $+5$ is 0101, so -5 is 1010; $+3$ is 0011, so -3 is 1100. If we add 1010 and 1100 ignoring the carry, we get 0110. In ones complement arithmetic, the fact that this

operation caused a carry from the most significant bit causes us to increment the result, giving 0111, which is the ones complement representation of −8 (obtained by inverting the bits in 1000), as we would expect.

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

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

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


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

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

2.4.3 Cyclic Redundancy Check

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

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

$$\begin{aligned} M(x) = {} & 1 \times x^7 + 0 \times x^6 + 0 \times x^5 \\ & + 1 \times x^4 + 1 \times x^3 + 0 \times x^2 \\ & + 1 \times x^1 + 0 \times x^0 \\ = {} & x^7 + x^4 + x^3 + x^1 \end{aligned}$$

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

For the purposes of calculating a CRC, a sender and receiver have to agree on a *divisor* polynomial, $C(x)$. $C(x)$ is a polynomial of degree k. For example, suppose $C(x) = x^3 + x^2 + 1$. In this case, $k = 3$. The answer to the question "Where did $C(x)$ come from?" is, in most practical cases, "You look it up in a book." In fact, the choice of $C(x)$ has a significant impact on what types of errors can be reliably detected, as we

Simple Probability Calculations

When dealing with network errors and other unlikely (we hope) events, we often have use for simple back-of-the-envelope probability estimates. A useful approximation here is that if two independent events have *small* probabilities p and q, then the probability of either event is $p + q$; the exact answer is $1 - (1 - p)(1 - q) = p + q - pq$. For $p = q = .01$, this estimate is .02, while the exact value is .0199.

For a simple application of this, suppose that the per-bit error rate on a link is 1 in 10^7. Now suppose we are interested in estimating the probability of at least one bit in a 10,000-bit packet being errored. Using the above approximation repeatedly over all the bits, we can say that we are interested in the probability of either the first bit being errored, or the second bit, or the third, and so on. Assuming bit errors are all independent (which they aren't), we can therefore estimate that

discuss below. There are a handful of divisor polynomials that are very good choices for various environments, and the exact choice is normally made as part of the protocol design. For example, the Ethernet standard uses a well-known polynomial of degree 32.

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

It will help to understand the following if you know a little about polynomial arithmetic; it is just slightly different from normal integer arithmetic. We are dealing with a special class of polynomial arithmetic here, where coefficients may be only one or zero, and operations on the coefficients are performed using modulo 2 arithmetic. This is referred to as polynomial arithmetic modulo 2. Since this is a networking book, not a mathematics text, let's focus on the key properties of this type of arithmetic for our purposes (which we ask you to accept on faith):

- Any polynomial $B(x)$ can be divided by a divisor polynomial $C(x)$ if $B(x)$ is of higher degree than $C(x)$;
- Any polynomial $B(x)$ can be divided once by a divisor polynomial $C(x)$ if $B(x)$ is of the same degree as $C(x)$;

the probability of at least one error in a 10,000-bit (10^4 bit) packet is $10^4 \times 10^{-7} = 10^{-3}$. The exact answer, computed as $1 - P(\text{no errors})$, would be $1 - (1 - 10^{-7})^{10,000} = .00099950$.

For a slightly more complex application, we compute the probability of exactly two errors in such a packet; this is the probability of an error that would sneak past a 1-parity-bit checksum. If we consider two particular bits in the packet, say bit i and bit j, the probability of those exact bits being errored is $10^{-7} \times 10^{-7}$. Now the total number of possible bit pairs in the packet is

$$\binom{10^4}{2} = 10^4 \times (10^4 - 1)/2 \approx 5 \times 10^7$$

So again using the approximation of repeatedly adding the probabilities of many rare events (in this case, of any possible bit pair being errored), our total probability of at least two errored bits is $5 \times 10^7 \times 10^{-14} = 5 \times 10^{-7}$.

- The remainder obtained when $B(x)$ is divided by $C(x)$ is obtained by subtracting $C(x)$ from $B(x)$;
- To subtract $C(x)$ from $B(x)$, we simply perform the exclusive-OR (XOR) operation on each pair of matching coefficients.

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

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

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

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

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

Consider the message $x^7 + x^4 + x^3 + x^1$, or 10011010. We begin by multiplying by x^3, since our divisor polynomial is of degree 3. This gives 10011010000. We divide this by $C(x)$, which corresponds to 1101 in this case. Figure 2.20 shows the polynomial long-division operation. Given the rules of polynomial arithmetic described above, the long division operation proceeds much as it would if we were dividing integers. Thus, in the first step of our example, we see that the divisor 1101 divides once into the first four bits of the message (1001), since they are of the same degree, and leaves a remainder of 100 (1101 XOR 1001). The next step is to bring down a digit from the message polynomial until we get another polynomial with the same degree as $C(x)$, in this case 1001. We calculate the remainder again (100) and continue until the calculation is complete. Note that the "result" of the long division, which appears at the top of the calculation, is not really of much interest—it is the remainder at the end that matters.

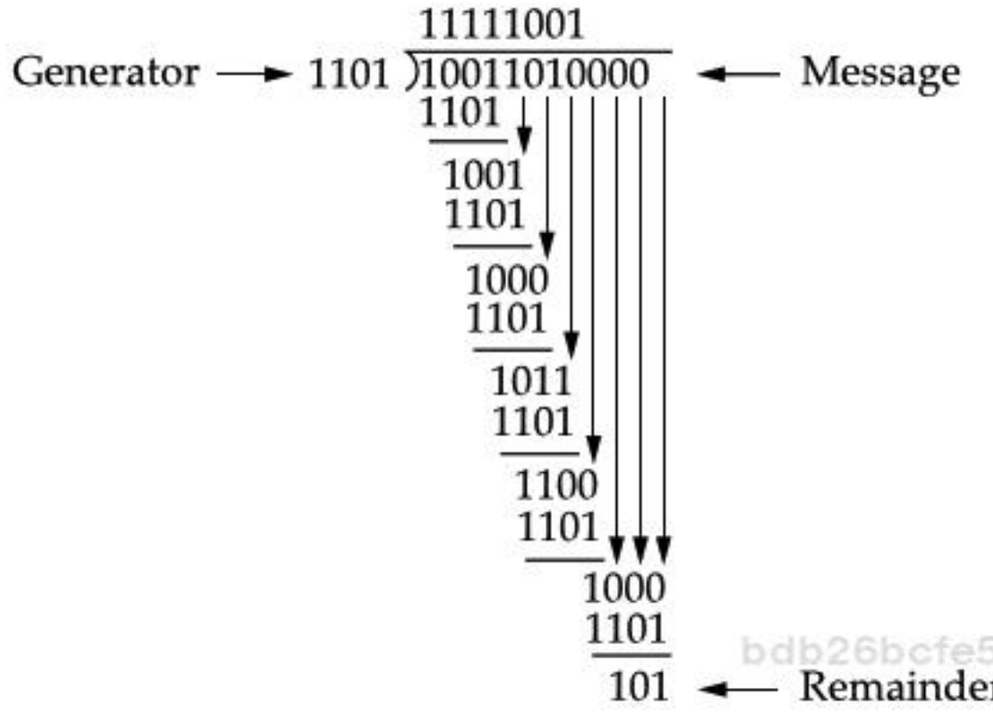

Figure 2.20 CRC calculation using polynomial long division.

You can see from the very bottom of Figure 2.20 that the remainder of the example calculation is 101. So we know that 10011010000 minus 101 would be exactly divisible by $C(x)$, and this is what we send. The minus operation in polynomial arithmetic is the logical XOR operation, so we actually send 10011010101. As noted above, this turns out to be just the original message with the remainder from the long-division calculation appended to it. The recipient divides the received polynomial by $C(x)$ and, if the result is 0, concludes that there were no errors. If the result is nonzero, it may be necessary to discard the corrupted message; with some codes, it may be possible to *correct* a small error (e.g., if the error affected only one bit). A code that enables error correction is called an *error correcting code (ECC)*.

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

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

- All single-bit errors, as long as the x^k and x^0 terms have nonzero coefficients;
- All double-bit errors, as long as $C(x)$ has a factor with at least three terms;
- Any odd number of errors, as long as $C(x)$ contains the factor $(x + 1)$;
- Any "burst" error (i.e., sequence of consecutive errored bits) for which the length of the burst is less than k bits. (Most burst errors of larger than k bits can also be detected.)

Six versions of $C(x)$ are widely used in link-level protocols (shown in Table 2.5). For example, the Ethernet and 802.5 networks described later in this chapter use CRC-32, while HDLC uses CRC-CCITT. ATM, as described in Chapter 3, uses CRC-8, CRC-10, and CRC-32.

Finally, we note that the CRC algorithm, while seemingly complex, is easily implemented in hardware using a k-bit shift register and XOR gates. The number of bits in the shift register equals the degree of the generator polynomial (k). Figure 2.21 shows the hardware that would be used for the generator $x^3 + x^2 + 1$ from our previous example. The message is shifted in from the left, beginning with the most significant bit and ending with the string of k zeros that is attached to the message, just as in the long-division example. When all the bits have been shifted in and appropriately XORed, the register contains the remainder, that is, the CRC (most significant bit on the right). The position of the XOR gates is determined as follows: If the bits in the shift register are labeled 0 through $k - 1$, left to right,

Error Detection or Error Correction?

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

At first glance, it would seem that correction is always better, since with detection we are forced to throw away the message and, in general, ask for another copy to be transmitted. This uses up bandwidth and may introduce latency while waiting for the retransmission. However, there is a downside to correction: It generally requires a greater number of redundant bits to send an error correcting code that is as strong (that is, able to cope with the same range of errors) as a code that only detects errors. Thus, while error detection requires more bits to be sent when errors occur, error correction requires more bits to be sent *all the time*. As a result, error correction tends to be most useful when (1) errors are quite probable, as they

CRC	$C(x)$
CRC-8	$x^8 + x^2 + x^1 + 1$
CRC-10	$x^{10} + x^9 + x^5 + x^4 + x^1 + 1$
CRC-12	$x^{12} + x^{11} + x^3 + x^2 + 1$
CRC-16	$x^{16} + x^{15} + x^2 + 1$
CRC-CCITT	$x^{16} + x^{12} + x^5 + 1$
CRC-32	$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$

Table 2.5 Common CRC polynomials.

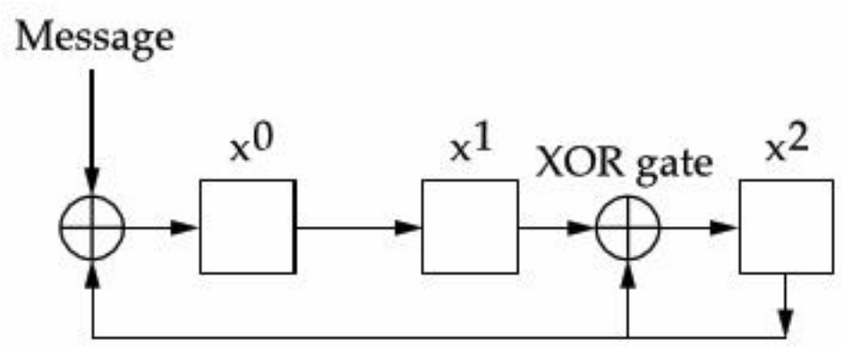

Figure 2.21 CRC calculation using shift register.

may be, for example, in a wireless environment, or (2) the cost of retransmission is too high, for example, because of the latency involved retransmitting a packet over a satellite link.

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

then put an XOR gate in front of bit n if there is a term x^n in the generator polynomial. Thus, we see an XOR gate in front of positions 0 and 2 for the generator $x^3 + x^2 + x^0$.

2.5 Reliable Transmission

As we saw in the previous section, frames are sometimes corrupted while in transit, with an error code like CRC used to detect such errors. While some error codes are strong enough also to correct errors, in practice the overhead is typically

too large to handle the range of bit and burst errors that can be introduced on a network link. Even when error correcting codes are used (e.g., on wireless links) some errors will be too severe to be corrected. As a result, some corrupt frames must be discarded. A link-level protocol that wants to deliver frames reliably must somehow recover from these discarded (lost) frames.

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

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

2.5.1 Stop-and-Wait

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

Figure 2.22 illustrates four different scenarios that result from this basic algorithm. This figure is a timeline, a common way to depict a protocol's behavior (see also the sidebar on this sort of diagram). The sending side is represented on the left, the receiving side is depicted on the right, and time flows from top to bottom. Figure 2.22(a) shows the situation in which the ACK is received before the timer expires, (b) and (c) show the situation in which the original frame and the ACK, respectively, are lost, and (d) shows the situation in which the timeout fires too soon. Recall that by "lost" we mean that the frame was corrupted while in transit, that this corruption was detected by an error code on the receiver, and that the frame was subsequently discarded.

There is one important subtlety in the stop-and-wait algorithm. Suppose the sender sends a frame and the receiver acknowledges it, but the acknowledgment is either lost or delayed in arriving. This situation is illustrated in timelines (c) and (d) of Figure 2.22. In both cases, the sender times out and retransmits the original frame, but the receiver will think that it is the next frame, since it correctly received and acknowledged the first

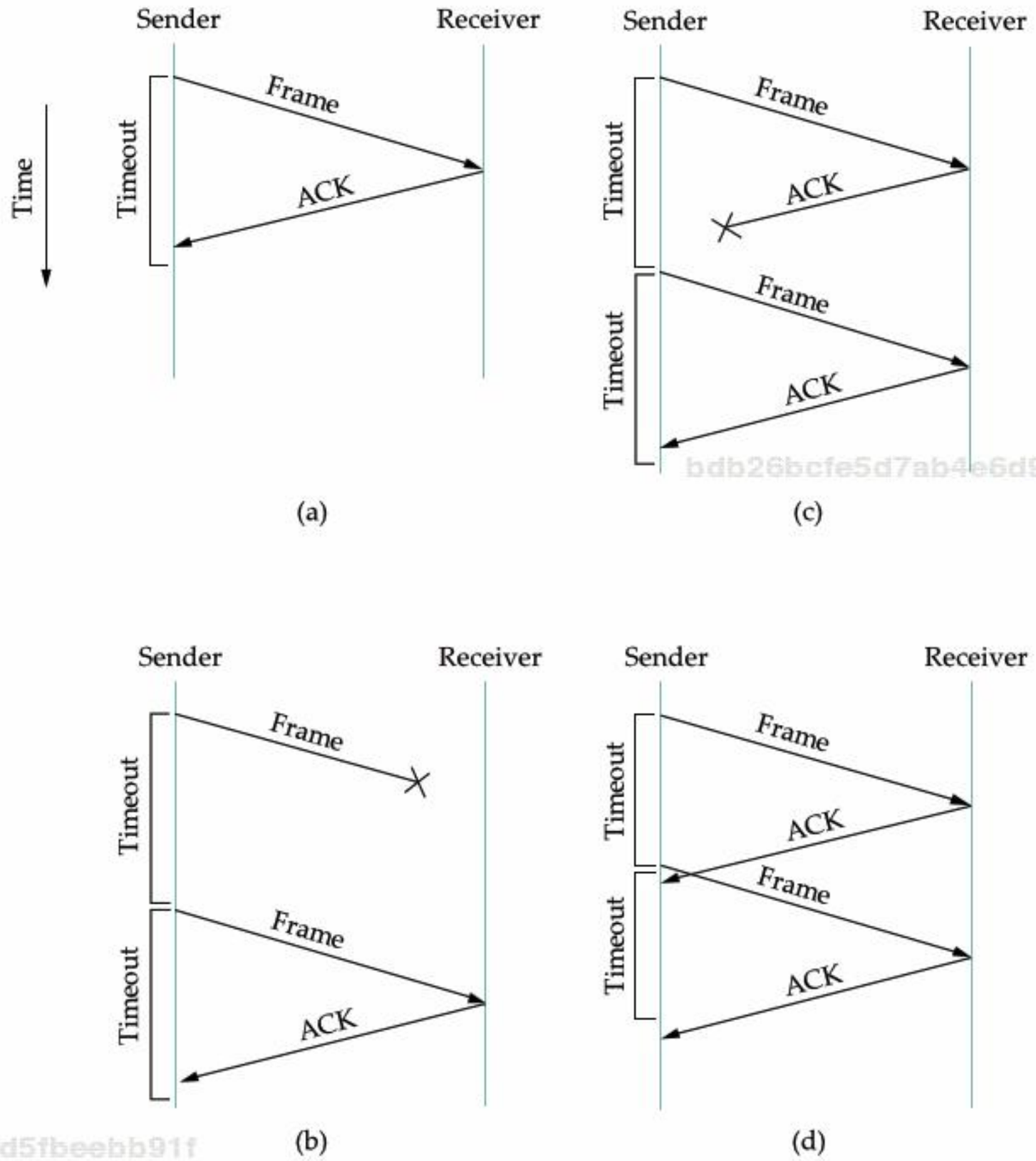

Figure 2.22 Timeline showing four different scenarios for the stop-and-wait algorithm. (a) The ACK is received before the timer expires; (b) the original frame is lost; (c) the ACK is lost; (d) the timeout fires too soon.

Timelines and Packet Exchange Diagrams

Figures 2.22 and 2.23 are two examples of a frequently-used tool in teaching, explaining, and designing protocols: the timeline or packet exchange diagram. You are going to see

frame. This has the potential to cause duplicate copies of a frame to be delivered. To address this problem, the header for a stop-and-wait protocol usually includes a 1-bit sequence number—that is, the sequence number can take on the values 0 and 1—and the sequence numbers used for each frame alternate, as illustrated in

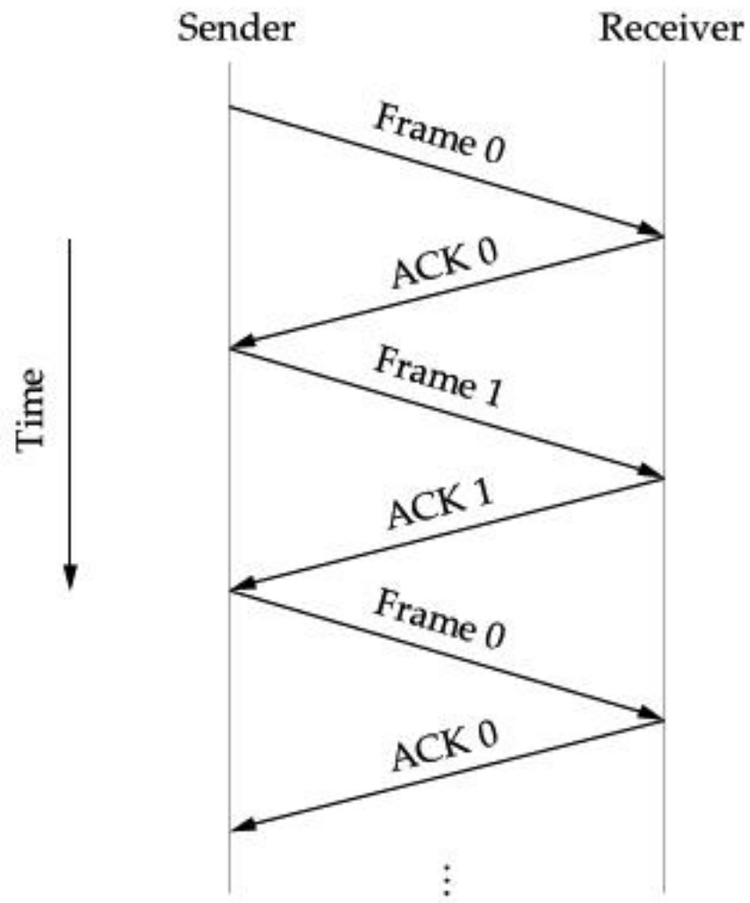

Figure 2.23 Timeline for stop-and-wait with 1-bit sequence number.

Figure 2.23. Thus, when the sender retransmits frame 0, the receiver can determine that it is seeing a second copy of frame 0 rather than the first copy of frame 1 and therefore can ignore it (the receiver still acknowledges it, in case the first ACK was lost).

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

$$\text{Bits Per Frame} \div \text{Time Per Frame}$$

$$= 1024 \times 8 \div 0.045$$

$$= 182 \text{ Kbps}$$

or about one-eighth of the link's capacity.

many more of them in this book—see Figures 9.12 and 9.16 for more complex examples. They are very useful because they capture visually the behavior over time of a distributed system—something that can be quite hard to analyze. When designing a protocol, you often have to be prepared for the unexpected—a system crashes, a message gets lost, or something that you expected to happen quickly turns out to take a long time. These sorts of diagrams can often help understand what might go wrong in such cases and thus help a protocol designer be prepared for every eventuality.

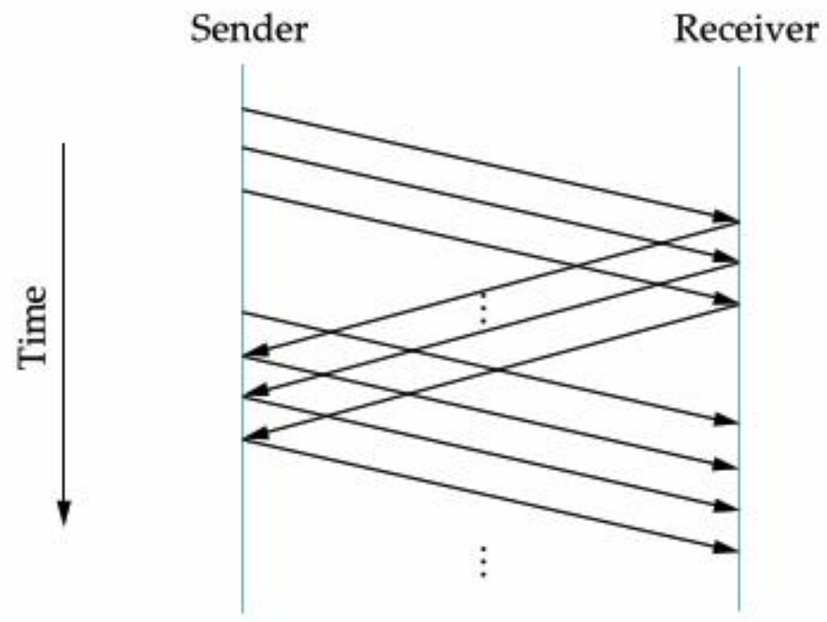

Figure 2.24 Timeline for the sliding window algorithm.

To use the link fully, then, we'd like the sender to be able to transmit up to eight frames before having to wait for an acknowledgment.

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

2.5.2 Sliding Window

Consider again the scenario in which the link has a delay × bandwidth product of 8 KB and frames are of 1-KB size. We would like the sender to be ready to transmit the ninth frame at pretty much the same moment that the ACK for the first frame arrives. The algorithm that allows us to do this is called *sliding window*, and an illustrative timeline is given in Figure 2.24.

The Sliding Window Algorithm

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

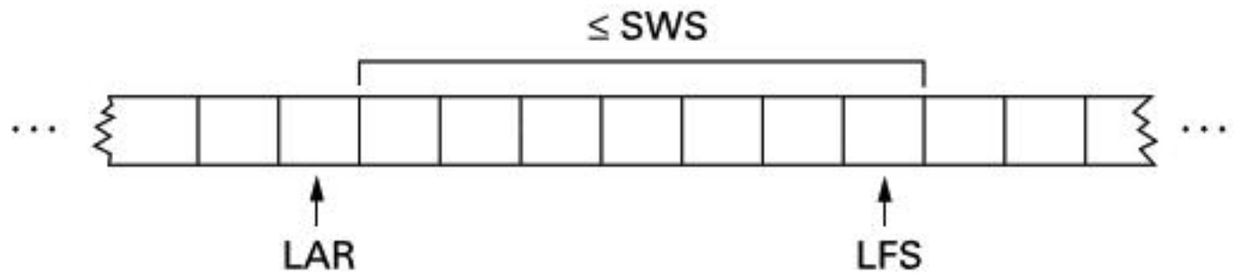

Figure 2.25 Sliding window on sender.

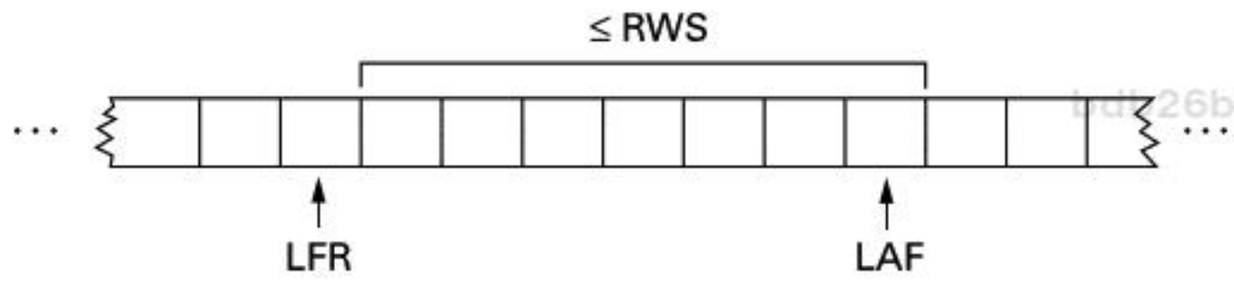

Figure 2.26 Sliding window on receiver.

maintains the following invariant:

$$\mathsf{LFS} - \mathsf{LAR} \leq \mathsf{SWS}$$

This situation is illustrated in Figure 2.25.

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

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

$$\mathsf{LAF} - \mathsf{LFR} \leq \mathsf{RWS}$$

This situation is illustrated in Figure 2.26.

When a frame with sequence number SeqNum arrives, the receiver takes the following action. If $\mathsf{SeqNum} \leq \mathsf{LFR}$ or $\mathsf{SeqNum} > \mathsf{LAF}$, then the frame is outside the receiver's window and it is discarded. If $\mathsf{LFR} < \mathsf{SeqNum} \leq \mathsf{LAF}$, then the

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

For example, suppose LFR = 5 (i.e., the last ACK the receiver sent was for sequence number 5), and RWS = 4. This implies that LAF = 9. Should frames 7 and 8 arrive, they will be buffered because they are within the receiver's window. However, no ACK needs to be sent since frame 6 is yet to arrive. Frames 7 and 8 are said to have arrived out of order. (Technically, the receiver could resend an ACK for frame 5 when frames 7 and 8 arrive.) Should frame 6 then arrive—perhaps it is late because it was lost the first time and had to be retransmitted, or perhaps it was simply delayed[2]—the receiver acknowledges frame 8, bumps LFR to 8, and sets LAF to 12. If frame 6 was in fact lost, then a timeout will have occurred at the sender, causing it to retransmit frame 6.

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

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

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

The sending window size is selected according to how many frames we want to have outstanding on the link at a given time; SWS is easy to compute for a given delay ×

[2] It's unlikely that a packet could be delayed in this way on a point-to-point link, but later on we will see this same algorithm used on more complex networks where such delays are possible.

bandwidth product.[3] On the other hand, the receiver can set RWS to whatever it wants. Two common settings are RWS = 1, which implies that the receiver will not buffer any frames that arrive out of order, and RWS = SWS, which implies that the receiver can buffer any of the frames the sender transmits. It makes no sense to set RWS > SWS since it's impossible for more than SWS frames to arrive out of order.

Finite Sequence Numbers and Sliding Window

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

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

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

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

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

[3] Easy, that is, if we know the delay and the bandwidth. Sometimes we do not, and estimating them well is a challenge to protocol designers. We discuss this further in Chapter 5.

Note that this rule is specific to the situation where RWS = SWS. We leave it as an exercise to determine the more general rule that works for arbitrary values of RWS and SWS. Also note that the relationship between the window size and the sequence number space depends on an assumption that is so obvious that it is easy to overlook, namely, that frames are not reordered in transit. This cannot happen on a direct point-to-point link since there is no way for one frame to overtake another during transmission. However, we will see the sliding window algorithm used in a different environment in Chapter 5, and we will need to devise another rule.

Implementation of Sliding Window

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

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

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


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

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

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

    /* receiver side state: */
    SwpSeqno    NFE;
                /* seqno of next frame expected */
    struct recvQ_slot {
        int     received; /* is msg valid? */
        Msg     msg;
    } recvQ[RWS];
} SwpState;
```


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

One detail worth noting is the call to store_swp_hdr just before the call to msgAddHdr. This routine translates the C structure that holds the SWP header (state->hdr) into a byte string that can be safely attached to the front of the message (hbuf). This routine (not shown) must translate each integer field in the header into network byte order and remove any padding that the compiler has added to the C structure. The issue of byte order is discussed more fully in Section 7.1, but for now it is enough to assume that this routine places the most significant bit of a multiword integer in the byte with the highest address.

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

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

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

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

Now to SWP's protocol-specific implementation of the deliver operation, which is given in procedure deliverSWP. This routine actually handles two different kinds

of incoming messages: ACKs for frames sent earlier from this node and data frames arriving at this node. In a sense, the ACK half of this routine is the counterpart to the sender side of the algorithm given in sendSWP. A decision as to whether the incoming message is an ACK or a data frame is made by checking the Flags field in the header. Note that this particular implementation does not support piggybacking ACKs on data frames.

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

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

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

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

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

    if (hdr.Flags & FLAG_HAS_DATA)
    {
        struct recvQ_slot *slot;

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

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

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

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

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

Frame Order and Flow Control

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

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

The third role that the sliding window algorithm sometimes plays is to support *flow control*—a feedback mechanism by which the receiver is able to throttle the sender. Such a mechanism is used to keep the sender from overrunning the receiver, that is, from transmitting more data than the receiver is able to process. This is usually accomplished by augmenting the sliding window protocol so that the receiver not only acknowledges

frames it has received, but also informs the sender of how many frames it has room to receive. The number of frames that the receiver is capable of receiving corresponds to how much free buffer space it has. As in the case of ordered delivery, we need to make sure that flow control is necessary at the link level before incorporating it into the sliding window protocol.

One important concept to take away from this discussion is the system design principle we call *separation of concerns*. That is, you must be careful to distinguish between different functions that are sometimes rolled together in one mechanism, and you must make sure that each function is necessary and being supported in the most effective way. In this particular case, reliable delivery, ordered delivery, and flow control are sometimes combined in a single sliding window protocol, and we should ask ourselves if this is the right thing to do at the link level. With this question in mind, we revisit the sliding window algorithm in Chapter 3 (we show how X.25 networks use it to implement hop-by-hop flow control) and in Chapter 5 (we describe how TCP uses it to implement a reliable byte-stream channel).

2.5.3 Concurrent Logical Channels

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

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

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

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