When a packet takes too long to arrive, the player will skip over the missing samples, perhaps playing ambient noise or repeating a frame to mask the loss to the user. There is a trade-off between the size of the buffer used to handle jitter and the amount of media that is lost. A smaller buffer reduces latency but results in more loss due to jitter. Eventually, as the size of the buffer shrinks, the loss will become noticeable to the user.

Observant readers may have noticed that we have said nothing about the network layer protocols so far in this section. The network can reduce latency, or at least jitter, by using quality of service mechanisms. The reason that this issue has not come up before is that streaming is able to operate with substantial latency, even in the live streaming case. If latency is not a major concern, a buffer at the end host is sufficient to handle the problem of jitter. However, for real-time conferencing, it is usually important to have the network reduce delay and jitter to help meet the delay budget. The only time that it is not important is when there is so much network bandwidth that everyone gets good service.

In Chap. 5, we described two quality of service mechanisms that help with this goal. One mechanism is DS (Differentiated Services), in which packets are marked as belonging to different classes that receive different handling within the network. The appropriate marking for voice-over-IP packets is low delay. In practice, systems set the DS codepoint to the well-known value for the *Expedited Forwarding* class with *Low Delay* type of service. This is especially useful over broadband access links, as these links tend to be congested when Web traffic or other traffic competes for use of the link. Given a stable network path, delay and jitter are increased by congestion. Every 1-KB packet takes 8 msec to send over a 1-Mbps link, and a voice-over-IP packet will incur these delays if it is sitting in a queue behind Web traffic. However, with a low delay marking the voice-over-IP packets will jump to the head of the queue, bypassing the Web packets and lowering their delay.

The second mechanism that can reduce delay is to make sure that there is sufficient bandwidth. If the available bandwidth varies or the transmission rate fluctuates (as with compressed video) and there is sometimes not sufficient bandwidth, queues will build up and add to the delay. This will occur even with DS. To ensure sufficient bandwidth, a reservation can be made with the network. This capability is provided by integrated services. Unfortunately, it is not widely deployed. Instead, networks are engineered for an expected traffic level or network customers are provided with service-level agreements for a given traffic level. Applications must operate below this level to avoid causing congestion and introducing unnecessary delays. For casual videoconferencing at home, the user may choose a video quality as a proxy for bandwidth needs, or the software may test the network path and select an appropriate quality automatically.

Any of the above factors can cause the latency to become unacceptable, so real-time conferencing requires that attention be paid to all of them. For an overview of voice over IP and analysis of these factors, see Goode (2002).

Now that we have discussed the problem of latency in the media streaming path, we will move on to the other main problem that conferencing systems must address. This problem is how to set up and tear down calls. We will look at two protocols that are widely used for this purpose, H.323 and SIP. Skype is another important system, but its inner workings are proprietary.

H.323

One thing that was clear to everyone before voice and video calls were made over the Internet was that if each vendor designed its own protocol stack, the system would never work. To avoid this problem, a number of interested parties got together under ITU auspices to work out standards. In 1996, ITU issued recommendation **H.323**, entitled "Visual Telephone Systems and Equipment for Local Area Networks Which Provide a Non-Guaranteed Quality of Service." Only the telephone industry would think of such a name. It was quickly changed to "Packet-based Multimedia Communications Systems" in the 1998 revision. H.323 was the basis for the first widespread Internet conferencing systems. It remains the most widely deployed solution, in its seventh version as of 2009.

H.323 is more of an architectural overview of Internet telephony than a specific protocol. It references a large number of specific protocols for speech coding, call setup, signaling, data transport, and other areas rather than specifying these things itself. The general model is depicted in Fig. 7-58. At the center is a **gateway** that connects the Internet to the telephone network. It speaks the H.323 protocols on the Internet side and the PSTN protocols on the telephone side. The communicating devices are called **terminals**. A LAN may have a **gatekeeper**, which controls the end points under its jurisdiction, called a **zone**.

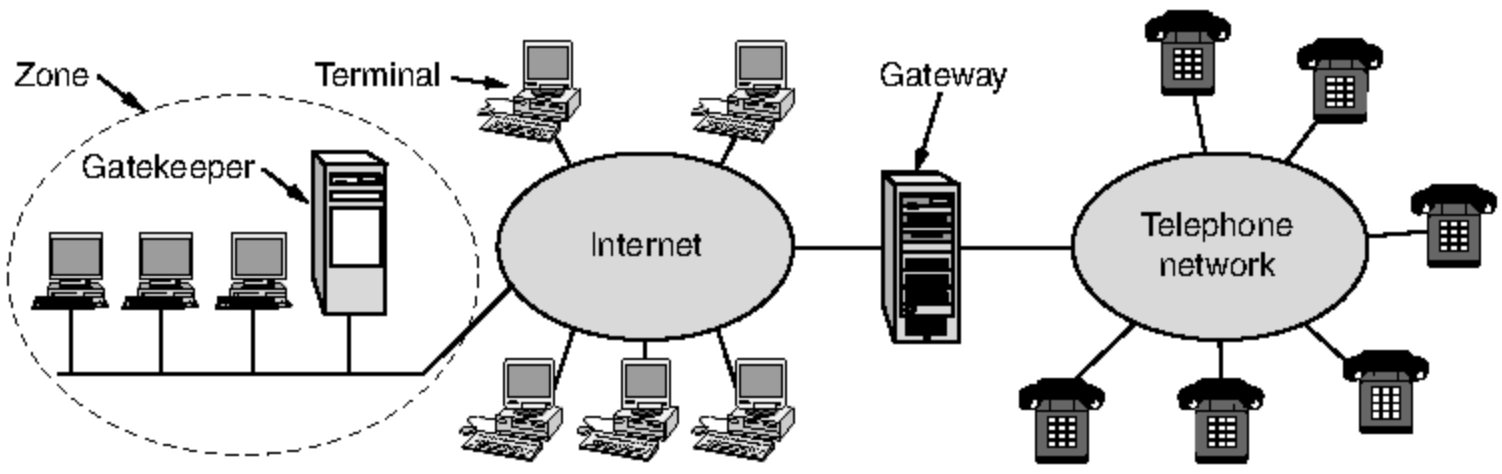

Figure 7-58. The H.323 architectural model for Internet telephony.

A telephone network needs a number of protocols. To start with, there is a protocol for encoding and decoding audio and video. Standard telephony representations of a single voice channel as 64 kbps of digital audio (8000 samples of 8 bits per second) are defined in ITU recommendation **G.711**. All H.323 systems

must support G.711. Other encodings that compress speech are permitted, but not required. They use different compression algorithms and make different trade-offs between quality and bandwidth. For video, the MPEG forms of video compression that we described above are supported, including H.264.

Since multiple compression algorithms are permitted, a protocol is needed to allow the terminals to negotiate which one they are going to use. This protocol is called **H.245**. It also negotiates other aspects of the connection such as the bit rate. RTCP is need for the control of the RTP channels. Also required is a protocol for establishing and releasing connections, providing dial tones, making ringing sounds, and the rest of the standard telephony. ITU **Q.931** is used here. The terminals need a protocol for talking to the gatekeeper (if present) as well. For this purpose, **H.225** is used. The PC-to-gatekeeper channel it manages is called the **RAS** (**Registration/Admission/Status**) channel. This channel allows terminals to join and leave the zone, request and return bandwidth, and provide status updates, among other things. Finally, a protocol is needed for the actual data transmission. RTP over UDP is used for this purpose. It is managed by RTCP, as usual. The positioning of all these protocols is shown in Fig. 7-59.

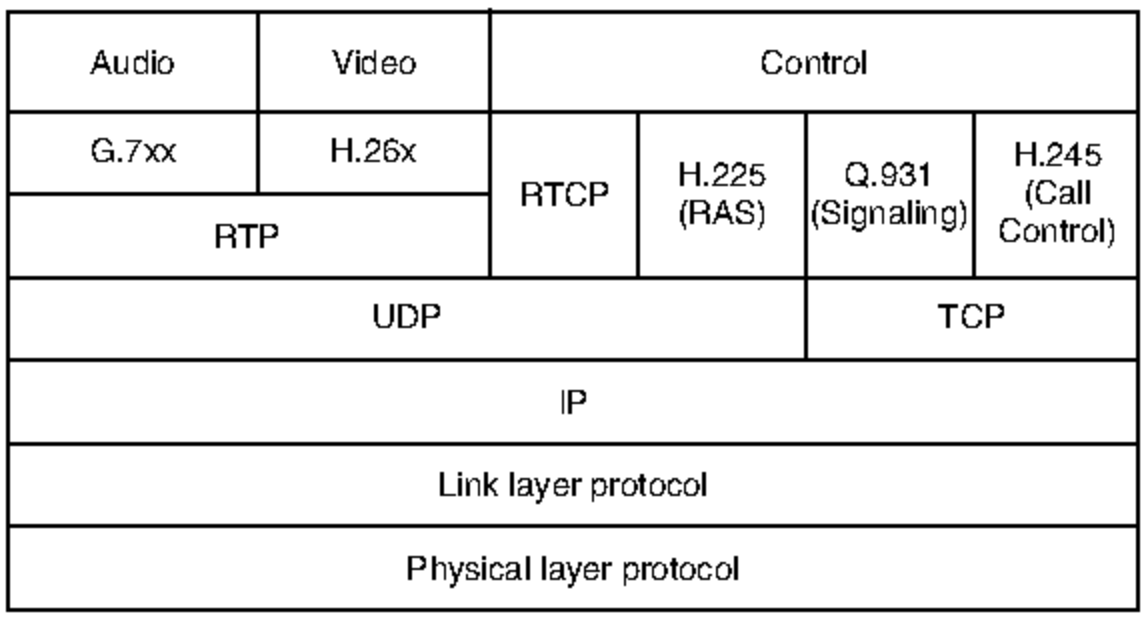

Figure 7-59. The H.323 protocol stack.

To see how these protocols fit together, consider the case of a PC terminal on a LAN (with a gatekeeper) calling a remote telephone. The PC first has to discover the gatekeeper, so it broadcasts a UDP gatekeeper discovery packet to port 1718. When the gatekeeper responds, the PC learns the gatekeeper's IP address. Now the PC registers with the gatekeeper by sending it a RAS message in a UDP packet. After it has been accepted, the PC sends the gatekeeper a RAS admission message requesting bandwidth. Only after bandwidth has been granted may call setup begin. The idea of requesting bandwidth in advance is to allow the gatekeeper to limit the number of calls. It can then avoid oversubscribing the outgoing line in order to help provide the necessary quality of service.

As an aside, the telephone system does the same thing. When you pick up the receiver, a signal is sent to the local end office. If the office has enough spare capacity for another call, it generates a dial tone. If not, you hear nothing. Nowadays, the system is so overdimensioned that the dial tone is nearly always instantaneous, but in the early days of telephony, it often took a few seconds. So if your grandchildren ever ask you "Why are there dial tones?" now you know. Except by then, probably telephones will no longer exist.

The PC now establishes a TCP connection to the gatekeeper to begin call setup. Call setup uses existing telephone network protocols, which are connection oriented, so TCP is needed. In contrast, the telephone system has nothing like RAS to allow telephones to announce their presence, so the H.323 designers were free to use either UDP or TCP for RAS, and they chose the lower-overhead UDP.

Now that it has bandwidth allocated, the PC can send a Q.931 *SETUP* message over the TCP connection. This message specifies the number of the telephone being called (or the IP address and port, if a computer is being called). The gatekeeper responds with a Q.931 *CALL PROCEEDING* message to acknowledge correct receipt of the request. The gatekeeper then forwards the *SETUP* message to the gateway.

The gateway, which is half computer, half telephone switch, then makes an ordinary telephone call to the desired (ordinary) telephone. The end office to which the telephone is attached rings the called telephone and also sends back a Q.931 *ALERT* message to tell the calling PC that ringing has begun. When the person at the other end picks up the telephone, the end office sends back a Q.931 *CONNECT* message to signal the PC that it has a connection.

Once the connection has been established, the gatekeeper is no longer in the loop, although the gateway is, of course. Subsequent packets bypass the gatekeeper and go directly to the gateway's IP address. At this point, we just have a bare tube running between the two parties. This is just a physical layer connection for moving bits, no more. Neither side knows anything about the other one.

The H.245 protocol is now used to negotiate the parameters of the call. It uses the H.245 control channel, which is always open. Each side starts out by announcing its capabilities, for example, whether it can handle video (H.323 can handle video) or conference calls, which codecs it supports, etc. Once each side knows what the other one can handle, two unidirectional data channels are set up and a codec and other parameters are assigned to each one. Since each side may have different equipment, it is entirely possible that the codecs on the forward and reverse channels are different. After all negotiations are complete, data flow can begin using RTP. It is managed using RTCP, which plays a role in congestion control. If video is present, RTCP handles the audio/video synchronization. The various channels are shown in Fig. 7-60. When either party hangs up, the Q.931 call signaling channel is used to tear down the connection after the call has been completed in order to free up resources no longer needed.

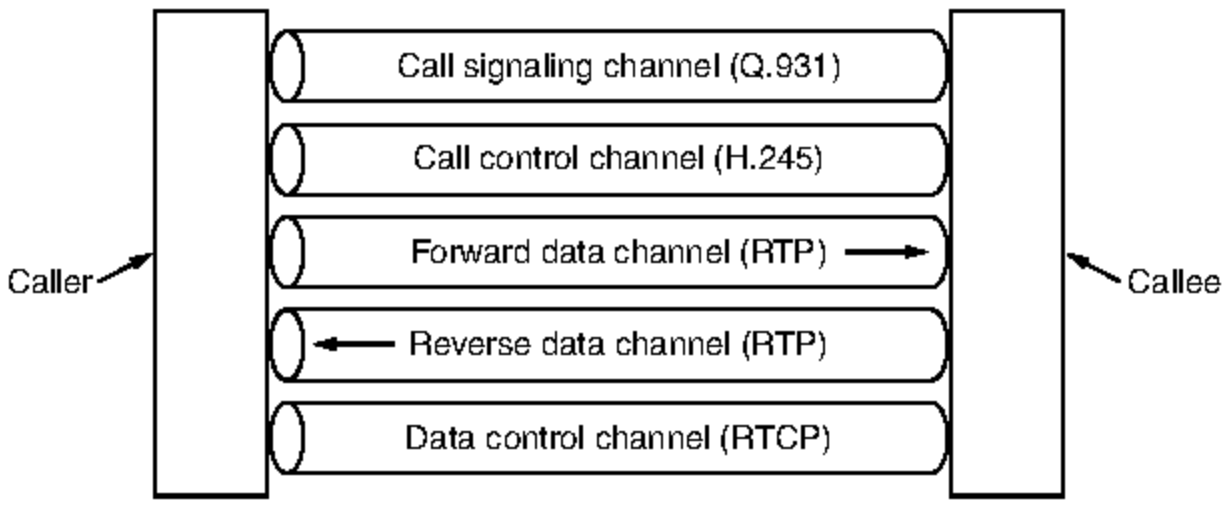

Figure 7-60. Logical channels between the caller and callee during a call.

When the call is terminated, the calling PC contacts the gatekeeper again with a RAS message to release the bandwidth it has been assigned. Alternatively, it can make another call.

We have not said anything about quality of service as part of H.323, even though we have said it is an important part of making real-time conferencing a success. The reason is that QoS falls outside the scope of H.323. If the underlying network is capable of producing a stable, jitter-free connection from the calling PC to the gateway, the QoS on the call will be good; otherwise, it will not be. However, any portion of the call on the telephone side will be jitter-free, because that is how the telephone network is designed.

SIP—The Session Initiation Protocol

H.323 was designed by ITU. Many people in the Internet community saw it as a typical telco product: large, complex, and inflexible. Consequently, IETF set up a committee to design a simpler and more modular way to do voice over IP. The major result to date is **SIP** (**Session Initiation Protocol**). The latest version is described in RFC 3261, which was written in 2002. This protocol describes how to set up Internet telephone calls, video conferences, and other multimedia connections. Unlike H.323, which is a complete protocol suite, SIP is a single module, but it has been designed to interwork well with existing Internet applications. For example, it defines telephone numbers as URLs, so that Web pages can contain them, allowing a click on a link to initiate a telephone call (the same way the *mailto* scheme allows a click on a link to bring up a program to send an email message).

SIP can establish two-party sessions (ordinary telephone calls), multiparty sessions (where everyone can hear and speak), and multicast sessions (one sender, many receivers). The sessions may contain audio, video, or data, the latter being useful for multiplayer real-time games, for example. SIP just handles setup, management, and termination of sessions. Other protocols, such as RTP/RTCP, are

also used for data transport. SIP is an application-layer protocol and can run over UDP or TCP, as required.

SIP supports a variety of services, including locating the callee (who may not be at his home machine) and determining the callee's capabilities, as well as handling the mechanics of call setup and termination. In the simplest case, SIP sets up a session from the caller's computer to the callee's computer, so we will examine that case first.

Telephone numbers in SIP are represented as URLs using the *sip* scheme, for example, *sip:ilse@cs.university.edu* for a user named Ilse at the host specified by the DNS name *cs.university.edu*. SIP URLs may also contain IPv4 addresses, IPv6 addresses, or actual telephone numbers.

The SIP protocol is a text-based protocol modeled on HTTP. One party sends a message in ASCII text consisting of a method name on the first line, followed by additional lines containing headers for passing parameters. Many of the headers are taken from MIME to allow SIP to interwork with existing Internet applications. The six methods defined by the core specification are listed in Fig. 7-61.

Method	Description
INVITE	Request initiation of a session
ACK	Confirm that a session has been initiated
BYE	Request termination of a session
OPTIONS	Query a host about its capabilities
CANCEL	Cancel a pending request
REGISTER	Inform a redirection server about the user's current location

Figure 7-61. SIP methods.

To establish a session, the caller either creates a TCP connection with the callee and sends an *INVITE* message over it or sends the *INVITE* message in a UDP packet. In both cases, the headers on the second and subsequent lines describe the structure of the message body, which contains the caller's capabilities, media types, and formats. If the callee accepts the call, it responds with an HTTP-type reply code (a three-digit number using the groups of Fig. 7-38, 200 for acceptance). Following the reply-code line, the callee also may supply information about its capabilities, media types, and formats.

Connection is done using a three-way handshake, so the caller responds with an *ACK* message to finish the protocol and confirm receipt of the 200 message.

Either party may request termination of a session by sending a message with the *BYE* method. When the other side acknowledges it, the session is terminated.

The *OPTIONS* method is used to query a machine about its own capabilities. It is typically used before a session is initiated to find out if that machine is even capable of voice over IP or whatever type of session is being contemplated.

The *REGISTER* method relates to SIP's ability to track down and connect to a user who is away from home. This message is sent to a SIP location server that keeps track of who is where. That server can later be queried to find the user's current location. The operation of redirection is illustrated in Fig. 7-62. Here, the caller sends the *INVITE* message to a proxy server to hide the possible redirection. The proxy then looks up where the user is and sends the *INVITE* message there. It then acts as a relay for the subsequent messages in the three-way handshake. The *LOOKUP* and *REPLY* messages are not part of SIP; any convenient protocol can be used, depending on what kind of location server is used.

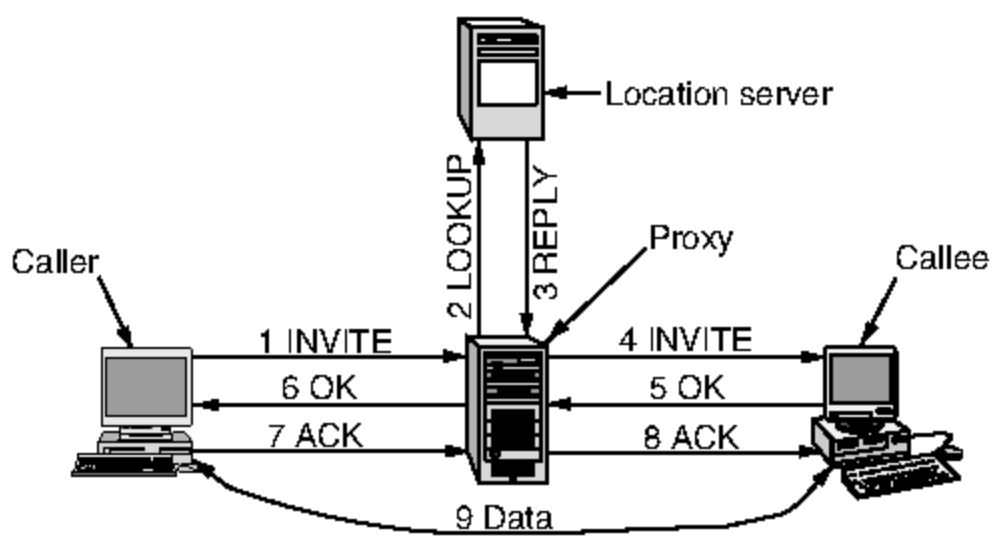

Figure 7-62. Use of a proxy server and redirection with SIP.

SIP has a variety of other features that we will not describe here, including call waiting, call screening, encryption, and authentication. It also has the ability to place calls from a computer to an ordinary telephone, if a suitable gateway between the Internet and telephone system is available.

Comparison of H.323 and SIP

Both H.323 and SIP allow two-party and multiparty calls using both computers and telephones as end points. Both support parameter negotiation, encryption, and the RTP/RTCP protocols. A summary of their similarities and differences is given in Fig. 7-63.

Although the feature sets are similar, the two protocols differ widely in philosophy. H.323 is a typical, heavyweight, telephone-industry standard, specifying the complete protocol stack and defining precisely what is allowed and what is forbidden. This approach leads to very well defined protocols in each layer, easing the task of interoperability. The price paid is a large, complex, and rigid standard that is difficult to adapt to future applications.

In contrast, SIP is a typical Internet protocol that works by exchanging short lines of ASCII text. It is a lightweight module that interworks well with other Internet protocols but less well with existing telephone system signaling protocols.

Item	H.323	SIP
Designed by	ITU	IETF
Compatibility with PSTN	Yes	Largely
Compatibility with Internet	Yes, over time	Yes
Architecture	Monolithic	Modular
Completeness	Full protocol stack	SIP just handles setup
Parameter negotiation	Yes	Yes
Call signaling	Q.931 over TCP	SIP over TCP or UDP
Message format	Binary	ASCII
Media transport	RTP/RTCP	RTP/RTCP
Multiparty calls	Yes	Yes
Multimedia conferences	Yes	No
Addressing	URL or phone number	URL
Call termination	Explicit or TCP release	Explicit or timeout
Instant messaging	No	Yes
Encryption	Yes	Yes
Size of standards	1400 pages	250 pages
Implementation	Large and complex	Moderate, but issues
Status	Widespread, esp. video	Alternative, esp. voice

Figure 7-63. Comparison of H.323 and SIP.

Because the IETF model of voice over IP is highly modular, it is flexible and can be adapted to new applications easily. The downside is that is has suffered from ongoing interoperability problems as people try to interpret what the standard means.

7.5 CONTENT DELIVERY

The Internet used to be all about communication, like the telephone network. Early on, academics would communicate with remote machines, logging in over the network to perform tasks. People have used email to communicate with each other for a long time, and now use video and voice over IP as well. Since the Web grew up, however, the Internet has become more about content than communication. Many people use the Web to find information, and there is a tremendous amount of peer-to-peer file sharing that is driven by access to movies, music, and programs. The switch to content has been so pronounced that the majority of Internet bandwidth is now used to deliver stored videos.

Because the task of distributing content is different from that of communication, it places different requirements on the network. For example, if Sally wants to talk to Jitu, she may make a voice-over-IP call to his mobile. The communication must be with a particular computer; it will do no good to call Paul's computer. But if Jitu wants to watch his team's latest cricket match, he is happy to stream video from whichever computer can provide the service. He does not mind whether the computer is Sally's or Paul's, or, more likely, an unknown server in the Internet. That is, location does not matter for content, except as it affects performance (and legality).

The other difference is that some Web sites that provide content have become tremendously popular. YouTube is a prime example. It allows users to share videos of their own creation on every conceivable topic. Many people want to do this. The rest of us want to watch. With all of these bandwidth-hungry videos, it is estimated that YouTube accounts for up to 10% of Internet traffic today.

No single server is powerful or reliable enough to handle such a startling level of demand. Instead, YouTube and other large content providers build their own content distribution networks. These networks use data centers spread around the world to serve content to an extremely large number of clients with good performance and availability.

The techniques that are used for content distribution have been developed over time. Early in the growth of the Web, its popularity was almost its undoing. More demands for content led to servers and networks that were frequently overloaded. Many people began to call the WWW the World Wide Wait.

In response to consumer demand, very large amounts of bandwidth were provisioned in the core of the Internet, and faster broadband connectivity was rolled out at the edge of the network. This bandwidth was key to improving performance, but it is only part of the solution. To reduce the endless delays, researchers also developed different architectures to use the bandwidth for distributing content.

One architecture is a **CDN** (**Content Distribution Network**). In it, a provider sets up a distributed collection of machines at locations inside the Internet and uses them to serve content to clients. This is the choice of the big players. An alternative architecture is a **P2P** (**Peer-to-Peer**) network. In it, a collection of computers pool their resources to serve content to each other, without separately provisioned servers or any central point of control. This idea has captured people's imagination because, by acting together, many little players can pack an enormous punch.

In this section, we will look at the problem of distributing content on the Internet and some of the solutions that are used in practice. After briefly discussing content popularity and Internet traffic, we will describe how to build powerful Web servers and use caching to improve performance for Web clients. Then we will come to the two main architectures for distributing content: CDNs and P2P networks. There design and properties are quite different, as we will see.

7.5.1 Content and Internet Traffic

To design and engineer networks that work well, we need an understanding of the traffic that they must carry. With the shift to content, for example, servers have migrated from company offices to Internet data centers that provide large numbers of machines with excellent network connectivity. To run even a small server nowadays, it is easier and cheaper to rent a virtual server hosted in an Internet data center than to operate a real machine in a home or office with broadband connectivity to the Internet.

Fortunately, there are only two facts about Internet traffic that is it essential to know. The first fact is that it changes quickly, not only in the details but in the overall makeup. Before 1994, most traffic was traditional FTP file transfer (for moving programs and data sets between computers) and email. Then the Web arrived and grew exponentially. Web traffic left FTP and email traffic in the dust long before the dot com bubble of 2000. Starting around 2000, P2P file sharing for music and then movies took off. By 2003, most Internet traffic was P2P traffic, leaving the Web in the dust. Sometime in the late 2000s, video streamed using content distribution methods by sites like YouTube began to exceed P2P traffic. By 2014, Cisco predicts that 90% of all Internet traffic will be video in one form or another (Cisco, 2010).

It is not always traffic volume that matters. For instance, while voice-over-IP traffic boomed even before Skype started in 2003, it will always be a minor blip on the chart because the bandwidth requirements of audio are two orders of magnitude lower than for video. However, voice-over-IP traffic stresses the network in other ways because it is sensitive to latency. As another example, online social networks have grown furiously since Facebook started in 2004. In 2010, for the first time, Facebook reached more users on the Web per day than Google. Even putting the traffic aside (and there is an awful lot of traffic), online social networks are important because they are changing the way that people interact via the Internet.

The point we are making is that seismic shifts in Internet traffic happen quickly, and with some regularity. What will come next? Please check back in the 6th edition of this book and we will let you know.

The second essential fact about Internet traffic is that it is highly skewed. Many properties with which we are familiar are clustered around an average. For instance, most adults are close to the average height. There are some tall people and some short people, but few very tall or very short people. For these kinds of properties, it is possible to design for a range that is not very large but nonetheless captures the majority of the population.

Internet traffic is not like this. For a long time, it has been known that there are a small number of Web sites with massive traffic and a vast number of Web site with much smaller traffic. This feature has become part of the language of networking. Early papers talked about traffic in terms of **packet trains**, the idea

being that express trains with a large number of packets would suddenly travel down a link (Jain and Routhier, 1986). This was formalized as the notion of **self-similarity**, which for our purposes can be thought of as network traffic that exhibits many short and many long gaps even when viewed at different time scales (Leland et al., 1994). Later work spoke of long traffic flows as **elephants** and short traffic flows as **mice**. The idea is that there are only a few elephants and many mice, but the elephants matter because they are so big.

Returning to Web content, the same sort of skew is evident. Experience with video rental stores, public libraries, and other such organizations shows that not all items are equally popular. Experimentally, when N movies are available, the fraction of all requests for the kth most popular one is approximately C/k. Here, C is computed to normalize the sum to 1, namely,

$$C = 1/(1 + 1/2 + 1/3 + 1/4 + 1/5 + \cdots + 1/N)$$

Thus, the most popular movie is seven times as popular as the number seven movie. This result is known as **Zipf's law** (Zipf, 1949). It is named after George Zipf, a professor of linguistics at Harvard University who noted that the frequency of a word's usage in a large body of text is inversely proportional to its rank. For example, the 40th most common word is used twice as much as the 80th most common word and three times as much as the 120th most common word.

A Zipf distribution is shown in Fig. 7-64(a). It captures the notion that there are a small number of popular items and a great many unpopular items. To recognize distributions of this form, it is convenient to plot the data on a log scale on both axes, as shown in Fig. 7-64(b). The result should be a straight line.

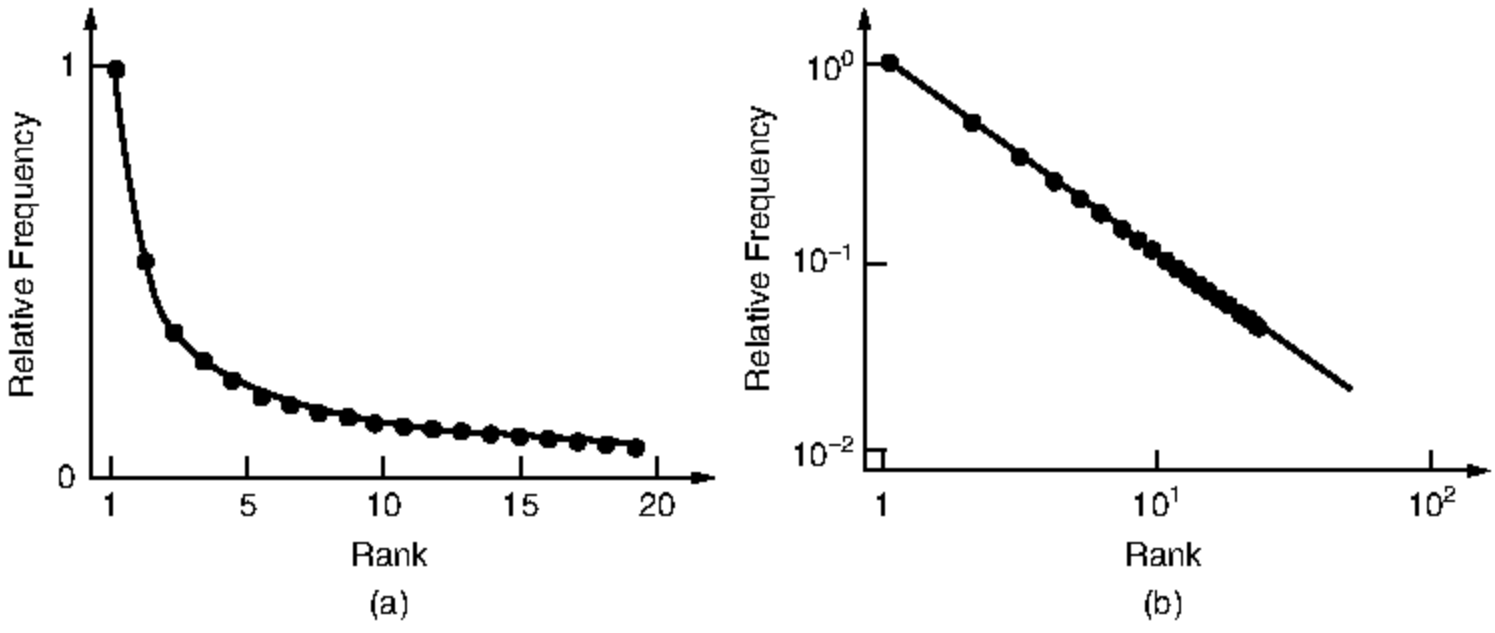

Figure 7-64. Zipf distribution (a) On a linear scale. (b) On a log-log scale.

When people looked at the popularity of Web pages, it also turned out to roughly follow Zipf's law (Breslau et al., 1999). A Zipf distribution is one example in a family of distributions known as **power laws**. Power laws are evident

in many human phenomena, such as the distribution of city populations and of wealth. They have the same propensity to describe a few large players and a great many smaller players, and they too appear as a straight line on a log-log plot. It was soon discovered that the topology of the Internet could be roughly described with power laws (Faloutsos et al., 1999). Next, researchers began plotting every imaginable property of the Internet on a log scale, observing a straight line, and shouting: "Power law!"

However, what matters more than a straight line on a log-log plot is what these distributions mean for the design and use of networks. Given the many forms of content that have Zipf or power law distributions, it seems fundamental that Web sites on the Internet are Zipf-like in popularity. This in turn means that an average site is not a useful representation. Sites are better described as either popular or unpopular. Both kinds of sites matter. The popular sites obviously matter, since a few popular sites may be responsible for most of the traffic on the Internet. Perhaps surprisingly, the unpopular sites can matter too. This is because the total amount of traffic directed to the unpopular sites can add up to a large fraction of the overall traffic. The reason is that there are so many unpopular sites. The notion that, collectively, many unpopular choices can matter has been popularized by books such as *The Long Tail* (Anderson, 2008a).

Curves showing decay like that of Fig. 7-64(a) are common, but they are not all the same. In particular, situations in which the rate of decay is proportional to how much material is left (such as with unstable radioactive atoms) exhibit **exponential decay**, which drops off much faster than Zipf's Law. The number of items, say atoms, left after time t is usually expressed as $e^{-t/\alpha}$, where the constant α determines how fast the decay is. The difference between exponential decay and Zipf's Law is that with exponential decay, it is safe to ignore the end of tail but with Zipf's Law the total weight of the tail is significant and cannot be ignored.

To work effectively in this skewed world, we must be able to build both kinds of Web sites. Unpopular sites are easy to handle. By using DNS, many different sites may actually point to the same computer in the Internet that runs all of the sites. On the other hand, popular sites are difficult to handle. There is no single computer even remotely powerful enough, and using a single computer would make the site inaccessible for millions of users if it fails. To handle these sites, we must build content distribution systems. We will start on that quest next.

7.5.2 Server Farms and Web Proxies

The Web designs that we have seen so far have a single server machine talking to multiple client machines. To build large Web sites that perform well, we can speed up processing on either the server side or the client side. On the server side, more powerful Web servers can be built with a server farm, in which a cluster of computers acts as a single server. On the client side, better performance can

be achieved with better caching techniques. In particular, proxy caches provide a large shared cache for a group of clients.

We will describe each of these techniques in turn. However, note that neither technique is sufficient to build the largest Web sites. Those popular sites require the content distribution methods that we describe in the following sections, which combine computers at many different locations.

Server Farms

No matter how much bandwidth one machine has, it can only serve so many Web requests before the load is too great. The solution in this case is to use more than one computer to make a Web server. This leads to the **server farm** model of Fig. 7-65.

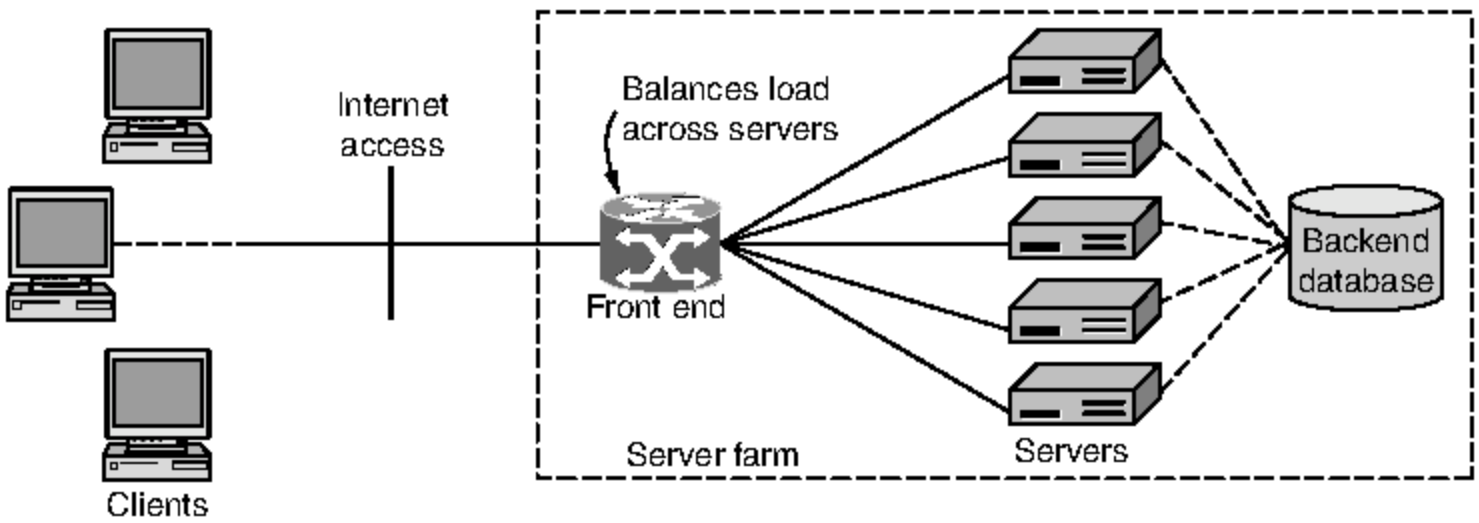

Figure 7-65. A server farm.

The difficulty with this seemingly simple model is that the set of computers that make up the server farm must look like a single logical Web site to clients. If they do not, we have just set up different Web sites that run in parallel.

There are several possible solutions to make the set of servers appear to be one Web site. All of the solutions assume that any of the servers can handle a request from any client. To do this, each server must have a copy of the Web site. The servers are shown as connected to a common back-end database by a dashed line for this purpose.

One solution is to use DNS to spread the requests across the servers in the server farm. When a DNS request is made for the Web URL, the DNS server returns a rotating list of the IP addresses of the servers. Each client tries one IP address, typically the first on the list. The effect is that different clients contact different servers to access the same Web site, just as intended. The DNS method is at the heart of CDNs, and we will revisit it later in this section.

The other solutions are based on a **front end** that sprays incoming requests over the pool of servers in the server farm. This happens even when the client

contacts the server farm using a single destination IP address. The front end is usually a link-layer switch or an IP router, that is, a device that handles frames or packets. All of the solutions are based on it (or the servers) peeking at the network, transport, or application layer headers and using them in nonstandard ways. A Web request and response are carried as a TCP connection. To work correctly, the front end must distribute all of the packets for a request to the same server.

A simple design is for the front end to broadcast all of the incoming requests to all of the servers. Each server answers only a fraction of the requests by prior agreement. For example, 16 servers might look at the source IP address and reply to the request only if the last 4 bits of the source IP address match their configured selectors. Other packets are discarded. While this is wasteful of incoming bandwidth, often the responses are much longer than the request, so it is not nearly as inefficient as it sounds.

In a more general design, the front end may inspect the IP, TCP, and HTTP headers of packets and arbitrarily map them to a server. The mapping is called a **load balancing** policy as the goal is to balance the workload across the servers. The policy may be simple or complex. A simple policy might be to use the servers one after the other in turn, or round-robin. With this approach, the front end must remember the mapping for each request so that subsequent packets that are part of the same request will be sent to the same server. Also, to make the site more reliable than a single server, the front end should notice when servers have failed and stop sending them requests.

Much like NAT, this general design is perilous, or at least fragile, in that we have just created a device that violates the most basic principle of layered protocols: each layer must use its own header for control purposes and may not inspect and use information from the payload for any purpose. But people design such systems anyway and when they break in the future due to changes in higher layers, they tend to be surprised. The front end in this case is a switch or router, but it may take action based on transport layer information or higher. Such a box is called a **middlebox** because it interposes itself in the middle of a network path in which it has no business, according to the protocol stack. In this case, the front end is best considered an internal part of a server farm that terminates all layers up to the application layer (and hence can use all of the header information for those layers).

Nonetheless, as with NAT, this design is useful in practice. The reason for looking at TCP headers is that it is possible to do a better job of load balancing than with IP information alone. For example, one IP address may represent an entire company and make many requests. It is only by looking at TCP or higher-layer information that these requests can be mapped to different servers.

The reason for looking at the HTTP headers is somewhat different. Many Web interactions access and update databases, such as when a customer looks up her most recent purchase. The server that fields this request will have to query the back-end database. It is useful to direct subsequent requests from the same user to

the same server, because that server has already cached information about the user. The simplest way to cause this to happen is to use Web cookies (or other information to distinguish the user) and to inspect the HTTP headers to find the cookies.

As a final note, although we have described this design for Web sites, a server farm can be built for other kinds of servers as well. An example is servers streaming media over UDP. The only change that is required is for the front end to be able to load balance these requests (which will have different protocol header fields than Web requests).

Web Proxies

Web requests and responses are sent using HTTP. In Sec. 7.3, we described how browsers can cache responses and reuse them to answer future requests. Various header fields and rules are used by the browser to determine if a cached copy of a Web page is still fresh. We will not repeat that material here.

Caching improves performance by shortening the response time and reducing the network load. If the browser can determine that a cached page is fresh by itself, the page can be fetched from the cache immediately, with no network traffic at all. However, even if the browser must ask the server for confirmation that the page is still fresh, the response time is shortened and the network load is reduced, especially for large pages, since only a small message needs to be sent.

However, the best the browser can do is to cache all of the Web pages that the user has previously visited. From our discussion of popularity, you may recall that as well as a few popular pages that many people visit repeatedly, there are many, many unpopular pages. In practice, this limits the effectiveness of browser caching because there are a large number of pages that are visited just once by a given user. These pages always have to be fetched from the server.

One strategy to make caches more effective is to share the cache among multiple users. That way, a page already fetched for one user can be returned to another user when that user makes the same request. Without browser caching, both users would need to fetch the page from the server. Of course, this sharing cannot be done for encrypted traffic, pages that require authentication, and uncacheable pages (e.g., current stock prices) that are returned by programs. Dynamic pages created by programs, especially, are a growing case for which caching is not effective. Nonetheless, there are plenty of Web pages that are visible to many users and look the same no matter which user makes the request (e.g., images).

A **Web proxy** is used to share a cache among users. A proxy is an agent that acts on behalf of someone else, such as the user. There are many kinds of proxies. For instance, an ARP proxy replies to ARP requests on behalf of a user who is elsewhere (and cannot reply for himself). A Web proxy fetches Web requests on behalf of its users. It normally provides caching of the Web responses, and since it is shared across users it has a substantially larger cache than a browser.

When a proxy is used, the typical setup is for an organization to operate one Web proxy for all of its users. The organization might be a company or an ISP. Both stand to benefit by speeding up Web requests for its users and reducing its bandwidth needs. While flat pricing, independent of usage, is common for end users, most companies and ISPs are charged according to the bandwidth that they use.

This setup is shown in Fig. 7-66. To use the proxy, each browser is configured to make page requests to the proxy instead of to the page's real server. If the proxy has the page, it returns the page immediately. If not, it fetches the page from the server, adds it to the cache for future use, and returns it to the client that requested it.

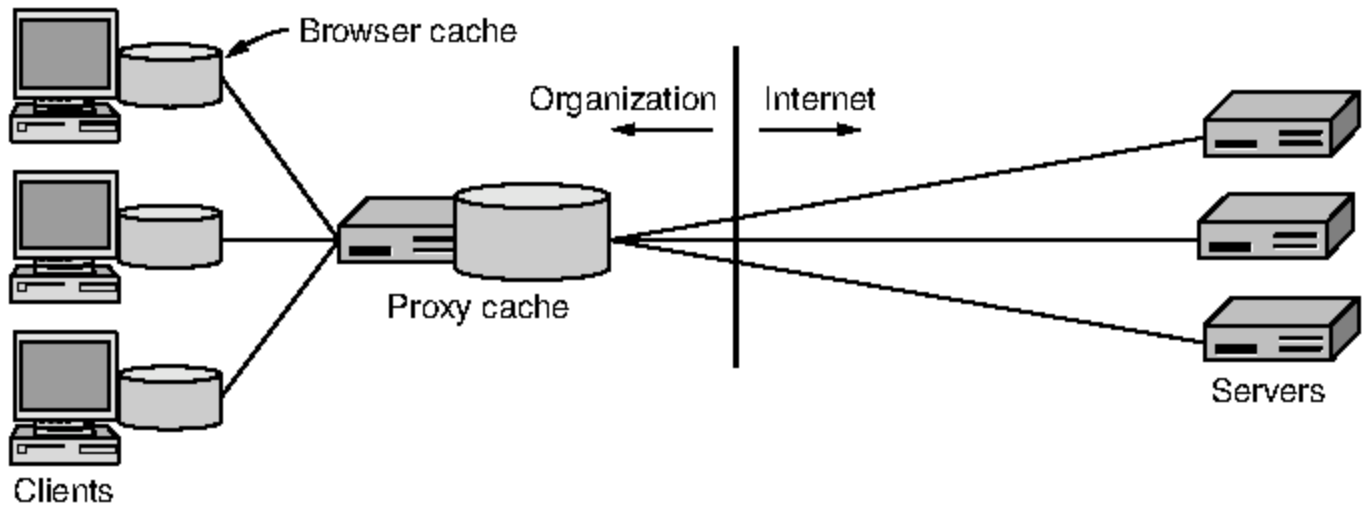

Figure 7-66. A proxy cache between Web browsers and Web servers.

As well as sending Web requests to the proxy instead of the real server, clients perform their own caching using its browser cache. The proxy is only consulted after the browser has tried to satisfy the request from its own cache. That is, the proxy provides a second level of caching.

Further proxies may be added to provide additional levels of caching. Each proxy (or browser) makes requests via its **upstream proxy**. Each upstream proxy caches for the **downstream proxies** (or browsers). Thus, it is possible for browsers in a company to use a company proxy, which uses an ISP proxy, which contacts Web servers directly. However, the single level of proxy caching we have shown in Fig. 7-66 is often sufficient to gain most of the potential benefits, in practice. The problem again is the long tail of popularity. Studies of Web traffic have shown that shared caching is especially beneficial until the number of users reaches about the size of a small company (say, 100 people). As the number of people grows larger, the benefits of sharing a cache become marginal because of the unpopular requests that cannot be cached due to lack of storage space (Wolman et al., 1999).

Web proxies provide additional benefits that are often a factor in the decision to deploy them. One benefit is to filter content. The administrator may configure

the proxy to blacklist sites or otherwise filter the requests that it makes. For example, many administrators frown on employees watching YouTube videos (or worse yet, pornography) on company time and set their filters accordingly. Another benefit of having proxies is privacy or anonymity, when the proxy shields the identity of the user from the server.

7.5.3 Content Delivery Networks

Server farms and Web proxies help to build large sites and to improve Web performance, but they are not sufficient for truly popular Web sites that must serve content on a global scale. For these sites, a different approach is needed.

CDNs (**Content Delivery Networks**) turn the idea of traditional Web caching on its head. Instead, of having clients look for a copy of the requested page in a nearby cache, it is the provider who places a copy of the page in a set of nodes at different locations and directs the client to use a nearby node as the server.

An example of the path that data follows when it is distributed by a CDN is shown in Fig. 7-67. It is a tree. The origin server in the CDN distributes a copy of the content to other nodes in the CDN, in Sydney, Boston, and Amsterdam, in this example. This is shown with dashed lines. Clients then fetch pages from the nearest node in the CDN. This is shown with solid lines. In this way, the clients in Sydney both fetch the page copy that is stored in Sydney; they do not both fetch the page from the origin server, which may be in Europe.

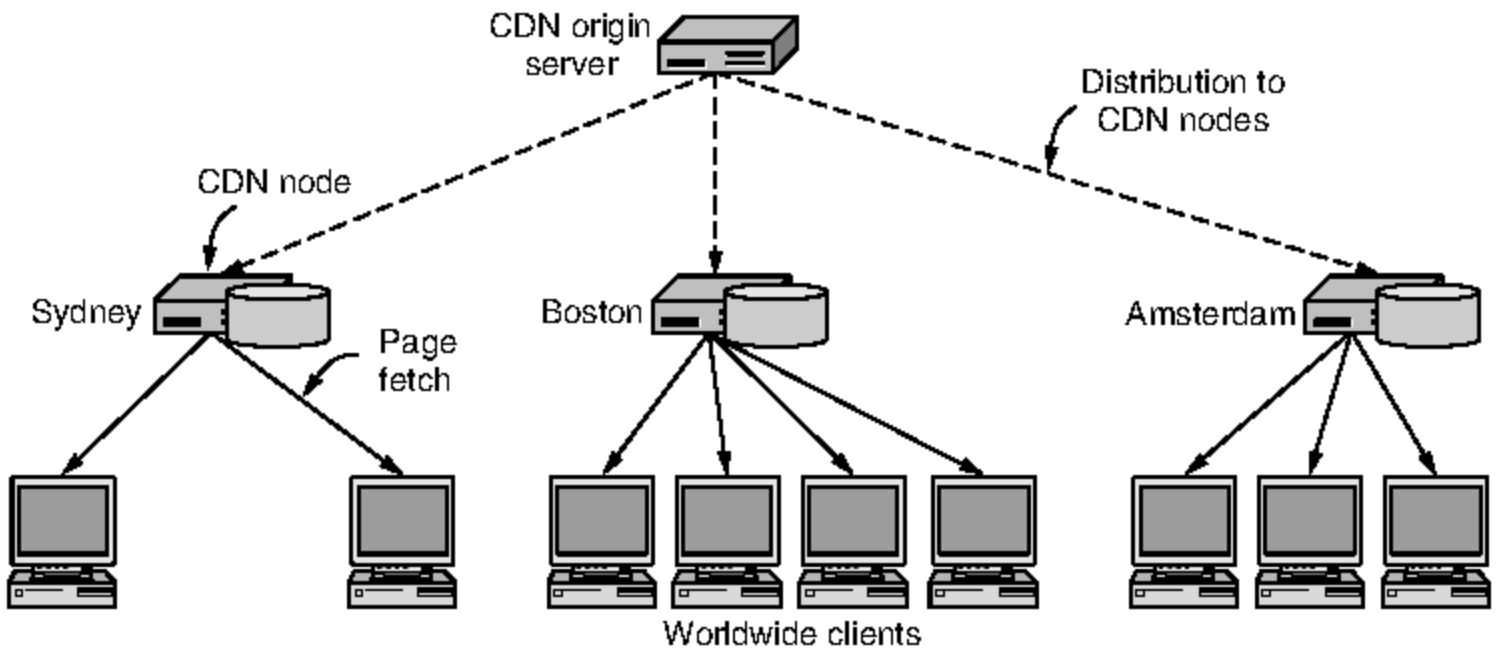

Figure 7-67. CDN distribution tree.

Using a tree structure has three virtues. First, the content distribution can be scaled up to as many clients as needed by using more nodes in the CDN, and more levels in the tree when the distribution among CDN nodes becomes the bottleneck. No matter how many clients there are, the tree structure is efficient. The origin server is not overloaded because it talks to the many clients via the tree

of CDN nodes; it does not have to answer each request for a page by itself. Second, each client gets good performance by fetching pages from a nearby server instead of a distant server. This is because the round-trip time for setting up a connection is shorter, TCP slow-start ramps up more quickly because of the shorter round-trip time, and the shorter network path is less likely to pass through regions of congestion in the Internet. Finally, the total load that is placed on the network is also kept at a minimum. If the CDN nodes are well placed, the traffic for a given page should pass over each part of the network only once. This is important because someone pays for network bandwidth, eventually.

The idea of using a distribution tree is straightforward. What is less simple is how to organize the clients to use this tree. For example, proxy servers would seem to provide a solution. Looking at Fig. 7-67, if each client was configured to use the Sydney, Boston or Amsterdam CDN node as a caching Web proxy, the distribution would follow the tree. However, this strategy falls short in practice, for three reasons. The first reason is that the clients in a given part of the network probably belong to different organizations, so they are probably using different Web proxies. Recall that caches are not usually shared across organizations because of the limited benefit of caching over a large number of clients, and for security reasons too. Second, there can be multiple CDNs, but each client uses only a single proxy cache. Which CDN should a client use as its proxy? Finally, perhaps the most practical issue of all is that Web proxies are configured by clients. They may or may not be configured to benefit content distribution by a CDN, and there is little that the CDN can do about it.

Another simple way to support a distribution tree with one level is to use **mirroring**. In this approach, the origin server replicates content over the CDN nodes as before. The CDN nodes in different network regions are called **mirrors**. The Web pages on the origin server contain explicit links to the different mirrors, usually telling the user their location. This design lets the user manually select a nearby mirror to use for downloading content. A typical use of mirroring is to place a large software package on mirrors located in, for example, the East and West coasts of the U.S., Asia, and Europe. Mirrored sites are generally completely static, and the choice of sites remains stable for months or years. They are a tried and tested technique. However, they depend on the user to do the distribution as the mirrors are really different Web sites, even if they are linked together.

The third approach, which overcomes the difficulties of the previous two approaches, uses DNS and is called **DNS redirection**. Suppose that a client wants to fetch a page with the URL *http://www.cdn.com/page.html*. To fetch the page, the browser will use DNS to resolve *www.cdn.com* to an IP address. This DNS lookup proceeds in the usual manner. By using the DNS protocol, the browser learns the IP address of the name server for *cdn.com*, then contacts the name server to ask it to resolve *www.cdn.com*. Now comes the really clever bit. The name server is run by the CDN. Instead, of returning the same IP address for each request, it will look at the IP address of the client making the request and return

different answers. The answer will be the IP address of the CDN node that is nearest the client. That is, if a client in Sydney asks the CDN name server to resolve *www.cdn.com*, the name server will return the IP address of the Sydney CDN node, but if a client in Amsterdam makes the same request, the name server will return the IP address of the Amsterdam CDN node instead.

This strategy is perfectly legal according to the semantics of DNS. We have previously seen that name servers may return changing lists of IP addresses. After the name resolution, the Sydney client will fetch the page directly from the Sydney CDN node. Further pages on the same "server" will be fetched directly from the Sydney CDN node as well because of DNS caching. The overall sequence of steps is shown in Fig. 7-68.

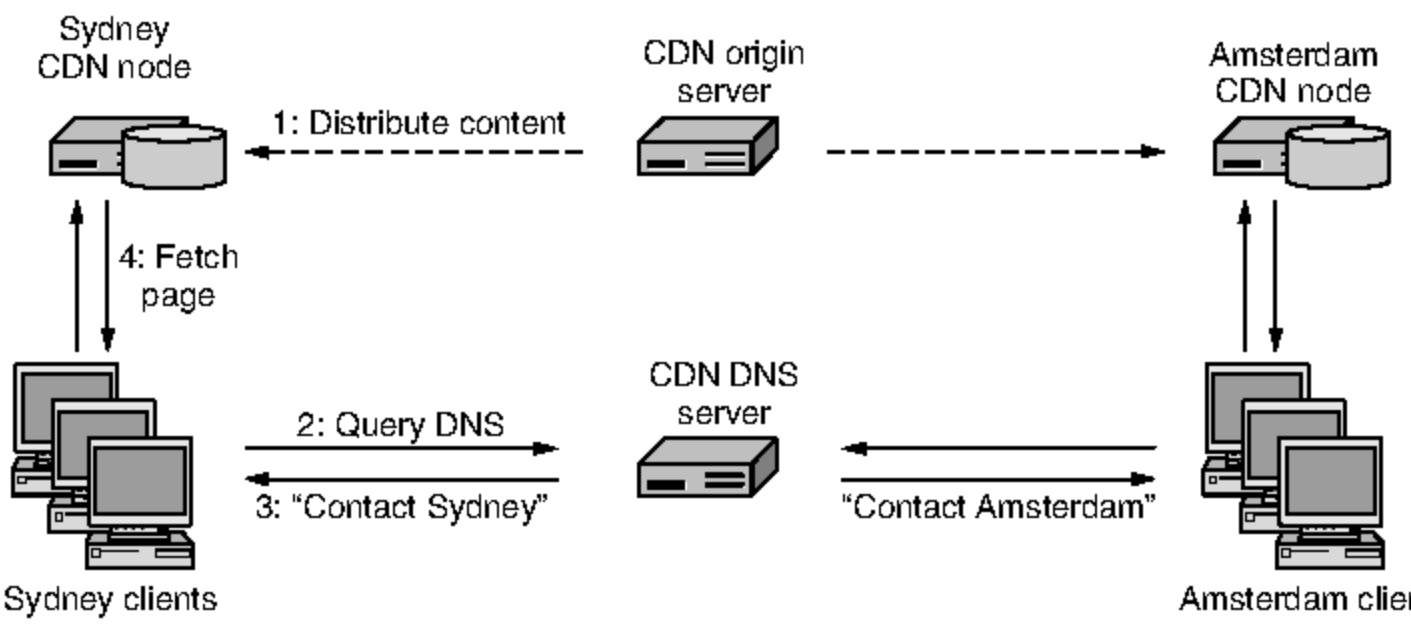

Figure 7-68. Directing clients to nearby CDN nodes using DNS.

A complex question in the above process is what it means to find the nearest CDN node, and how to go about it. To define nearest, it is not really geography that matters. There are at least two factors to consider in mapping a client to a CDN node. One factor is the network distance. The client should have a short and high-capacity network path to the CDN node. This situation will produce quick downloads. CDNs use a map they have previously computed to translate between the IP address of a client and its network location. The CDN node that is selected might be the one at the shortest distance as the crow flies, or it might not. What matters is some combination of the length of the network path and any capacity limits along it. The second factor is the load that is already being carried by the CDN node. If the CDN nodes are overloaded, they will deliver slow responses, just like the overloaded Web server that we sought to avoid in the first place. Thus, it may be necessary to balance the load across the CDN nodes, mapping some clients to nodes that are slightly further away but more lightly loaded.

The techniques for using DNS for content distribution were pioneered by Akamai starting in 1998, when the Web was groaning under the load of its early

growth. Akamai was the first major CDN and became the industry leader. Probably even more clever than the idea of using DNS to connect clients to nearby nodes was the incentive structure of their business. Companies pay Akamai to deliver their content to clients, so that they have responsive Web sites that customers like to use. The CDN nodes must be placed at network locations with good connectivity, which initially meant inside ISP networks. For the ISPs, there is a benefit to having a CDN node in their networks, namely that the CDN node cuts down the amount of upstream network bandwidth that they need (and must pay for), just as with proxy caches. In addition, the CDN node improves responsiveness for the ISP's customers, which makes the ISP look good in their eyes, giving them a competitive advantage over ISPs that do not have a CDN node. These benefits (at no cost to the ISP) makes installing a CDN node a no brainer for the ISP. Thus, the content provider, the ISP, and the customers all benefit and the CDN makes money. Since 1998, other companies have gotten into the business, so it is now a competitive industry with multiple providers.

As this description implies, most companies do not build their own CDN. Instead, they use the services of a CDN provider such as Akamai to actually deliver their content. To let other companies use the service of a CDN, we need to add one last step to our picture.

After the contract is signed for a CDN to distribute content on behalf of a Web site owner, the owner gives the CDN the content. This content is pushed to the CDN nodes. In addition, the owner rewrites any of its Web pages that link to the content. Instead of linking to the content on their Web site, the pages link to the content via the CDN. As an example of how this scheme works, consider the source code for Fluffy Video's Web page, given in Fig. 7-69(a). After preprocessing, it is transformed to Fig. 7-69(b) and placed on Fluffy Video's server as *www.fluffyvideo.com/index.html*.

When a user types in the URL *www.fluffyvideo.com* to his browser, DNS returns the IP address of Fluffy Video's own Web site, allowing the main (HTML) page to be fetched in the normal way. When the user clicks on any of the hyperlinks, the browser asks DNS to look up *www.cdn.com*. This lookup contacts the CDN's DNS server, which returns the IP address of the nearby CDN node. The browser then sends a regular HTTP request to the CDN node, for example, for */fluffyvideo/koalas.mpg*. The URL identifies the page to return, starting the path with *fluffyvideo* so that the CDN node can separate requests for the different companies that it serves. Finally, the video is returned and the user sees cute fluffy animals.

The strategy behind this split of content hosted by the CDN and entry pages hosted by the content owner is that it gives the content owner control while letting the CDN move the bulk of the data. Most entry pages are tiny, being just HTML text. These pages often link to large files, such as videos and images. It is precisely these large files that are served by the CDN, even though the use of a CDN is completely transparent to users. The site looks the same, but performs faster.

```
<html>
<head> <title> Fluffy Video </title> </head>
<body>
<h1> Fluffy Video's Product List </h1>
<p> Click below for free samples. </p>

<a href="koalas.mpg"> Koalas Today </a> <br>
<a href="kangaroos.mpg"> Funny Kangaroos </a> <br>
<a href="wombats.mpg"> Nice Wombats </a> <br>
</body>
</html>
```

(a)

```
<html>
<head> <title> Fluffy Video </title> </head>
<body>
<h1> Fluffy Video's Product List </h1>
<p> Click below for free samples. </p>

<a href="http://www.cdn.com/fluffyvideo/koalas.mpg"> Koalas Today </a> <br>
<a href="http://www.cdn.com/fluffyvideo/kangaroos.mpg"> Funny Kangaroos </a> <br>
<a href="http://www.cdn.com/fluffyvideo/wombats.mpg"> Nice Wombats </a> <br>
</body>
</html>
```

(b)

Figure 7-69. (a) Original Web page. (b) Same page after linking to the CDN.

There is another advantage for sites using a shared CDN. The future demand for a Web site can be difficult to predict. Frequently, there are surges in demand known as **flash crowds**. Such a surge may happen when the latest product is released, there is a fashion show or other event, or the company is otherwise in the news. Even a Web site that was a previously unknown, unvisited backwater can suddenly become the focus of the Internet if it is newsworthy and linked from popular sites. Since most sites are not prepared to handle massive increases in traffic, the result is that many of them crash when traffic surges.

Case in point. Normally the Florida Secretary of State's Web site is not a busy place, although you can look up information about Florida corporations, notaries, and cultural affairs, as well as information about voting and elections there. For some odd reason, on Nov. 7, 2000 (the date of the U.S. presidential election with Bush vs. Gore), a whole lot of people were suddenly interested in the election results page of this site. The site suddenly became one of the busiest Web sites in the world and naturally crashed as a result. If it had been using a CDN, it would probably have survived.

By using a CDN, a site has access to a very large content-serving capacity. The largest CDNs have tens of thousands of servers deployed in countries all over the world. Since only a small number of sites will be experiencing a flash crowd

at any one time (by definition), those sites may use the CDN's capacity to handle the load until the storm passes. That is, the CDN can quickly scale up a site's serving capacity.

The preceding discussion above is a simplified description of how Akamai works. There are many more details that matter in practice. The CDN nodes pictured in our example are normally clusters of machines. DNS redirection is done with two levels: one to map clients to the approximate network location, and another to spread the load over nodes in that location. Both reliability and performance are concerns. To be able to shift a client from one machine in a cluster to another, DNS replies at the second level are given with short TTLs so that the client will repeat the resolution after a short while. Finally, while we have concentrated on distributing static objects like images and videos, CDNs can also support dynamic page creation, streaming media, and more. For more information about CDNs, see Dilley et al. (2002).

7.5.4 Peer-to-Peer Networks

Not everyone can set up a 1000-node CDN at locations around the world to distribute their content. (Actually, it is not hard to rent 1000 virtual machines around the globe because of the well-developed and competitive hosting industry. However, setting up a CDN only starts with getting the nodes.) Luckily, there is an alternative for the rest of us that is simple to use and can distribute a tremendous amount of content. It is a P2P (Peer-to-Peer) network.

P2P networks burst onto the scene starting in 1999. The first widespread application was for mass crime: 50 million Napster users were exchanging copyrighted songs without the copyright owners' permission until Napster was shut down by the courts amid great controversy. Nevertheless, peer-to-peer technology has many interesting and legal uses. Other systems continued development, with such great interest from users that P2P traffic quickly eclipsed Web traffic. Today, BitTorrent is the most popular P2P protocol. It is used so widely to share (licensed and public domain) videos, as well as other content, that it accounts for a large fraction of all Internet traffic. We will look at it in this section.

The basic idea of a **P2P** (**Peer-to-Peer**) file-sharing network is that many computers come together and pool their resources to form a content distribution system. The computers are often simply home computers. They do not need to be machines in Internet data centers. The computers are called peers because each one can alternately act as a client to another peer, fetching its content, and as a server, providing content to other peers. What makes peer-to-peer systems interesting is that there is no dedicated infrastructure, unlike in a CDN. Everyone participates in the task of distributing content, and there is often no central point of control.

Many people are excited about P2P technology because it is seen as empowering the little guy. The reason is not only that it takes a large company to run a

CDN, while anyone with a computer can join a P2P network. It is that P2P networks have a formidable capacity to distribute content that can match the largest of Web sites.

Consider a P2P network made up of N average users, each with broadband connectivity at 1 Mbps. The aggregate upload capacity of the P2P network, or rate at which the users can send traffic into the Internet, is N Mbps. The download capacity, or rate at which the users can receive traffic, is also N Mbps. Each user can upload and download at the same time, too, because they have a 1-Mbps link in each direction.

It is not obvious that this should be true, but it turns out that all of the capacity can be used productively to distribute content, even for the case of sharing a single copy of a file with all the other users. To see how this can be so, imagine that the users are organized into a binary tree, with each non-leaf user sending to two other users. The tree will carry the single copy of the file to all the other users. To use the upload bandwidth of as many users as possible at all times (and hence distribute the large file with low latency), we need to pipeline the network activity of the users. Imagine that the file is divided into 1000 pieces. Each user can receive a new piece from somewhere up the tree and send the previously received piece down the tree at the same time. This way, once the pipeline is started, after a small number of pieces (equal to the depth of the tree) are sent, all non-leaf users will be busy uploading the file to other users. Since there are approximately $N/2$ non-leaf users, the upload bandwidth of this tree is $N/2$ Mbps. We can repeat this trick and create another tree that uses the other $N/2$ Mbps of upload bandwidth by swapping the roles of leaf and non-leaf nodes. Together, this construction uses all of the capacity.

This argument means that P2P networks are self-scaling. Their usable upload capacity grows in tandem with the download demands that can be made by their users. They are always "large enough" in some sense, without the need for any dedicated infrastructure. In contrast, the capacity of even a large Web site is fixed and will either be too large or too small. Consider a site with only 100 clusters, each capable of 10 Gbps. This enormous capacity does not help when there are a small number of users. The site cannot get information to N users at a rate faster than N Mbps because the limit is at the users and not the Web site. And when there are more than one million 1-Mbps users, the Web site cannot pump out data fast enough to keep all the users busy downloading. That may seem like a large number of users, but large BitTorrent networks (e.g., Pirate Bay) claim to have more than 10,000,000 users. That is more like 10 terabits/sec in terms of our example!

You should take these back-of-the-envelope numbers with a grain (or better yet, a metric ton) of salt because they oversimplify the situation. A significant challenge for P2P networks is to use bandwidth well when users can come in all shapes and sizes, and have different download and upload capacities. Nevertheless, these numbers do indicate the enormous potential of P2P.

There is another reason that P2P networks are important. CDNs and other centrally run services put the providers in a position of having a trove of personal information about many users, from browsing preferences and where people shop online, to people's locations and email addresses. This information can be used to provide better, more personalized service, or it can be used to intrude on people's privacy. The latter may happen either intentionally—say as part of a new product—or through an accidental disclosure or compromise. With P2P systems, there can be no single provider that is capable of monitoring the entire system. This does not mean that P2P systems will necessarily provide privacy, as users are trusting each other to some extent. It only means that they can provide a different form of privacy than centrally managed systems. P2P systems are now being explored for services beyond file sharing (e.g., storage, streaming), and time will tell whether this advantage is significant.

P2P technology has followed two related paths as it has been developed. On the more practical side, there are the systems that are used every day. The most well known of these systems are based on the BitTorrent protocol. On the more academic side, there has been intense interest in DHT (Distributed Hash Table) algorithms that let P2P systems perform well as a whole, yet rely on no centralized components at all. We will look at both of these technologies.

BitTorrent

The BitTorrent protocol was developed by Brahm Cohen in 2001 to let a set of peers share files quickly and easily. There are dozens of freely available clients that speak this protocol, just as there are many browsers that speak the HTTP protocol to Web servers. The protocol is available as an open standard at *www.bittorrent.org*.

In a typical peer-to-peer system, like that formed with BitTorrent, the users each have some information that may be of interest to other users. This information may be free software, music, videos, photographs, and so on. There are three problems that need to be solved to share content in this setting:

1. How does a peer find other peers that have the content it wants to download?
2. How is content replicated by peers to provide high-speed downloads for everyone?
3. How do peers encourage each other to upload content to others as well as download content for themselves?

The first problem exists because not all peers will have all of the content, at least initially. The approach taken in BitTorrent is for every content provider to create a content description called a **torrent**. The torrent is much smaller than the

content, and is used by a peer to verify the integrity of the data that it downloads from other peers. Other users who want to download the content must first obtain the torrent, say, by finding it on a Web page advertising the content.

The torrent is just a file in a specified format that contains two key kinds of information. One kind is the name of a **tracker**, which is a server that leads peers to the content of the torrent. The other kind of information is a list of equal-sized pieces, or **chunks**, that make up the content. Different chunk sizes can be used for different torrents, typically 64 KB to 512 KB. The torrent file contains the name of each chunk, given as a 160-bit SHA-1 hash of the chunk. We will cover cryptographic hashes such as SHA-1 in Chap. 8. For now, you can think of a hash as a longer and more secure checksum. Given the size of chunks and hashes, the torrent file is at least three orders of magnitude smaller than the content, so it can be transferred quickly.

To download the content described in a torrent, a peer first contacts the tracker for the torrent. The **tracker** is a server that maintains a list of all the other peers that are actively downloading and uploading the content. This set of peers is called a **swarm**. The members of the swarm contact the tracker regularly to report that they are still active, as well as when they leave the swarm. When a new peer contacts the tracker to join the swarm, the tracker tells it about other peers in the swarm. Getting the torrent and contacting the tracker are the first two steps for downloading content, as shown in Fig. 7-70.

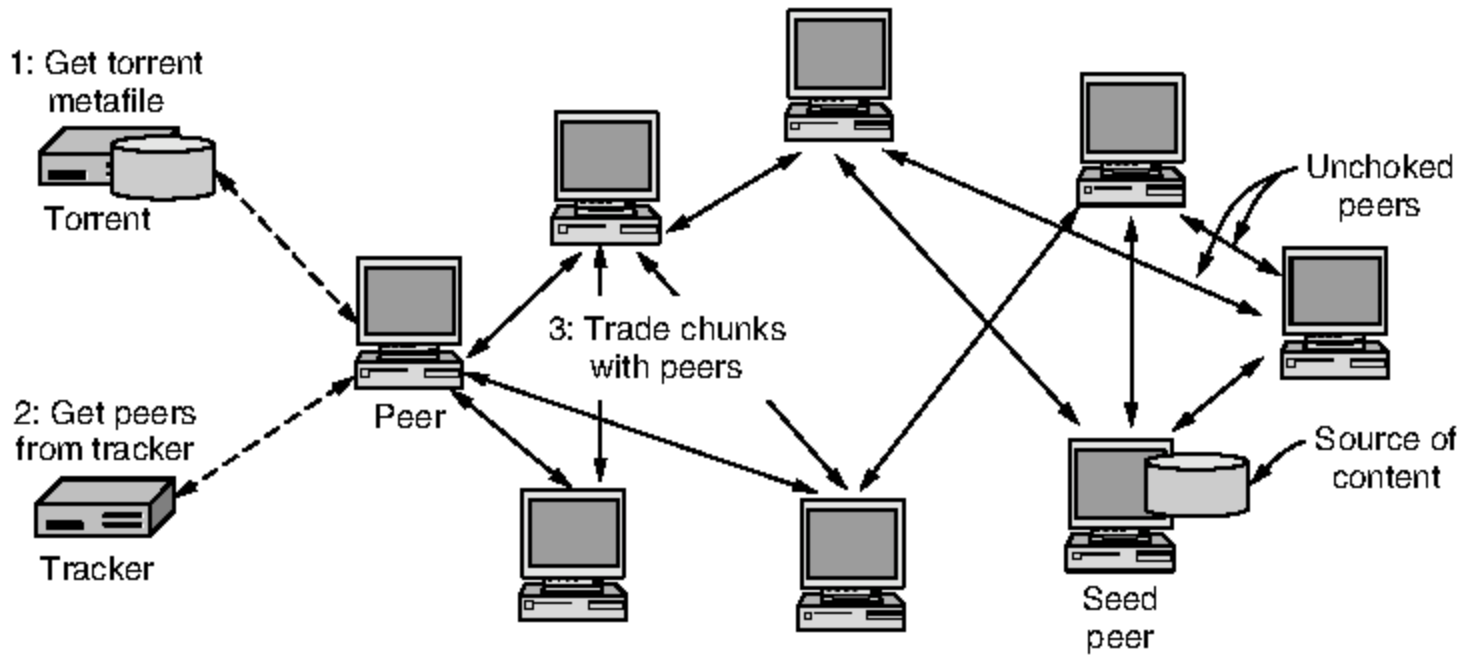

Figure 7-70. BitTorrent.

The second problem is how to share content in a way that gives rapid downloads. When a swarm is first formed, some peers must have all of the chunks that make up the content. These peers are called **seeders**. Other peers that join the swarm will have no chunks; they are the peers that are downloading the content.

While a peer participates in a swarm, it simultaneously downloads chunks that it is missing from other peers, and uploads chunks that it has to other peers who

need them. This trading is shown as the last step of content distribution in Fig. 7-70. Over time, the peer gathers more chunks until it has downloaded all of the content. The peer can leave the swarm (and return) at any time. Normally a peer will stay for a short period after finishes its own download. With peers coming and going, the rate of churn in a swarm can be quite high.

For the above method to work well, each chunk should be available at many peers. If everyone were to get the chunks in the same order, it is likely that many peers would depend on the seeders for the next chunk. This would create a bottleneck. Instead, peers exchange lists of the chunks they have with each other. Then they select rare chunks that are hard to find to download. The idea is that downloading a rare chunk will make a copy of it, which will make the chunk easier for other peers to find and download. If all peers do this, after a short while all chunks will be widely available.

The third problem is perhaps the most interesting. CDN nodes are set up exclusively to provide content to users. P2P nodes are not. They are users' computers, and the users may be more interested in getting a movie than helping other users with their downloads. Nodes that take resources from a system without contributing in kind are called **free-riders** or **leechers**. If there are too many of them, the system will not function well. Earlier P2P systems were known to host them (Saroiu et al., 2003) so BitTorrent sought to minimize them.

The approach taken in BitTorrent clients is to reward peers who show good upload behavior. Each peer randomly samples the other peers, retrieving chunks from them while it uploads chunks to them. The peer continues to trade chunks with only a small number of peers that provide the highest download performance, while also randomly trying other peers to find good partners. Randomly trying peers also allows newcomers to obtain initial chunks that they can trade with other peers. The peers with which a node is currently exchanging chunks are said to be **unchoked**.

Over time, this algorithm is intended to match peers with comparable upload and download rates with each other. The more a peer is contributing to the other peers, the more it can expect in return. Using a set of peers also helps to saturate a peer's download bandwidth for high performance. Conversely, if a peer is not uploading chunks to other peers, or is doing so very slowly, it will be cut off, or **choked**, sooner or later. This strategy discourages antisocial behavior in which peers free-ride on the swarm.

The choking algorithm is sometimes described as implementing the **tit-for-tat** strategy that encourages cooperation in repeated interactions. However, it does not prevent clients from gaming the system in any strong sense (Piatek et al., 2007). Nonetheless, attention to the issue and mechanisms that make it more difficult for casual users to free-ride have likely contributed to the success of BitTorrent.

As you can see from our discussion, BitTorrent comes with a rich vocabulary. There are torrents, swarms, leechers, seeders, and trackers, as well as snubbing,

choking, lurking, and more. For more information see the short paper on BitTorrent (Cohen, 2003) and look on the Web starting with *www.bittorrent.org*.

DHTs—Distributed Hash Tables

The emergence of P2P file sharing networks around 2000 sparked much interest in the research community. The essence of P2P systems is that they avoid the centrally managed structures of CDNs and other systems. This can be a significant advantage. Centrally managed components become a bottleneck as the system grows very large and are a single point of failure. Central components can also be used as a point of control (e.g., to shut off the P2P network). However, the early P2P systems were only partly decentralized, or, if they were fully decentralized, they were inefficient.

The traditional form of BitTorrent that we just described uses peer-to-peer transfers and a centralized tracker for each swarm. It is the tracker that turns out to be the hard part to decentralize in a peer-to-peer system. The key problem is how to find out which peers have specific content that is being sought. For example, each user might have one or more data items such as songs, photographs, programs, files, and so on that other users might want to read. How do the other users find them? Making one index of who has what is simple, but it is centralized. Having every peer keep its own index does not help. True, it is distributed. However, it requires so much work to keep the indexes of all peers up to date (as content is moved about the system) that it is not worth the effort.

The question tackled by the research community was whether it was possible to build P2P indexes that were entirely distributed but performed well. By perform well, we mean three things. First, each node keeps only a small amount of information about other nodes. This property means that it will not be expensive to keep the index up to date. Second, each node can look up entries in the index quickly. Otherwise, it is not a very useful index. Third, each node can use the index at the same time, even as other nodes come and go. This property means the performance of the index grows with the number of nodes.

The answer is to the question was: "Yes." Four different solutions were invented in 2001. They are Chord (Stoica et al., 2001), CAN (Ratnasamy et al., 2001), Pastry (Rowstron and Druschel, 2001), and Tapestry (Zhao et al., 2004). Other solutions were invented soon afterwards, including Kademlia, which is used in practice (Maymounkov and Mazieres, 2002). The solutions are known as **DHTs** (**Distributed Hash Tables**) because the basic functionality of an index is to map a key to a value. This is like a hash table, and the solutions are distributed versions, of course.

DHTs do their work by imposing a regular structure on the communication between the nodes, as we will see. This behavior is quite different than that of traditional P2P networks that use whatever connections peers happen to make.

For this reason, DHTs are called **structured P2P networks**. Traditional P2P protocols build **unstructured P2P networks**.

The DHT solution that we will describe is Chord. As a scenario, consider how to replace the centralized tracker traditionally used in BitTorrent with a fully-distributed tracker. Chord can be used to solve this problem. In this scenario, the overall index is a listing of all of the swarms that a computer may join to download content. The key used to look up the index is the torrent description of the content. It uniquely identifies a swarm from which content can be downloaded as the hashes of all the content chunks. The value stored in the index for each key is the list of peers that comprise the swarm. These peers are the computers to contact to download the content. A person wanting to download content such as a movie has only the torrent description. The question the DHT must answer is how, lacking a central database, does a person find out which peers (out of the millions of BitTorrent nodes) to download the movie from?

A Chord DHT consists of n participating nodes. They are nodes running BitTorrent in our scenario. Each node has an IP address by which it may be contacted. The overall index is spread across the nodes. This implies that each node stores bits and pieces of the index for use by other nodes. The key part of Chord is that it navigates the index using identifiers in a virtual space, not the IP addresses of nodes or the names of content like movies. Conceptually, the identifiers are simply m-bit numbers that can be arranged in ascending order into a ring.

To turn a node address into an identifier, it is mapped to an m-bit number using a hash function, *hash*. Chord uses SHA-1 for *hash*. This is the same hash that we mentioned when describing BitTorrent. We will look at it when we discuss cryptography in Chap. 8. For now, suffice it to say that it is just a function that takes a variable-length byte string as an argument and produces a highly random 160-bit number. Thus, we can use it to convert any IP address to a 160-bit number called the **node identifier**.

In Fig. 7-71(a), we show the node identifier circle for $m = 5$. (Just ignore the arcs in the middle for the moment.) Some of the identifiers correspond to nodes, but most do not. In this example, the nodes with identifiers 1, 4, 7, 12, 15, 20, and 27 correspond to actual nodes and are shaded in the figure; the rest do not exist.

Let us now define the function *successor*(k) as the node identifier of the first actual node following k around the circle, clockwise. For example, *successor* (6) = 7, *successor* (8) = 12, and *successor* (22) = 27.

A **key** is also produced by hashing a content name with *hash* (i.e., SHA-1) to generate a 160-bit number. In our scenario, the content name is the torrent. Thus, in order to convert *torrent* (the torrent description file) to its key, we compute *key* = *hash*(*torrent*). This computation is just a local procedure call to *hash*.

To start a new a swarm, a node needs to insert a new key-value pair consisting of (*torrent*, *my-IP-address*) into the index. To accomplish this, the node asks *successor*(*hash*(*torrent*)) to store *my-IP-address*. In this way, the index is distributed over the nodes at random. For fault tolerance, p different hash functions

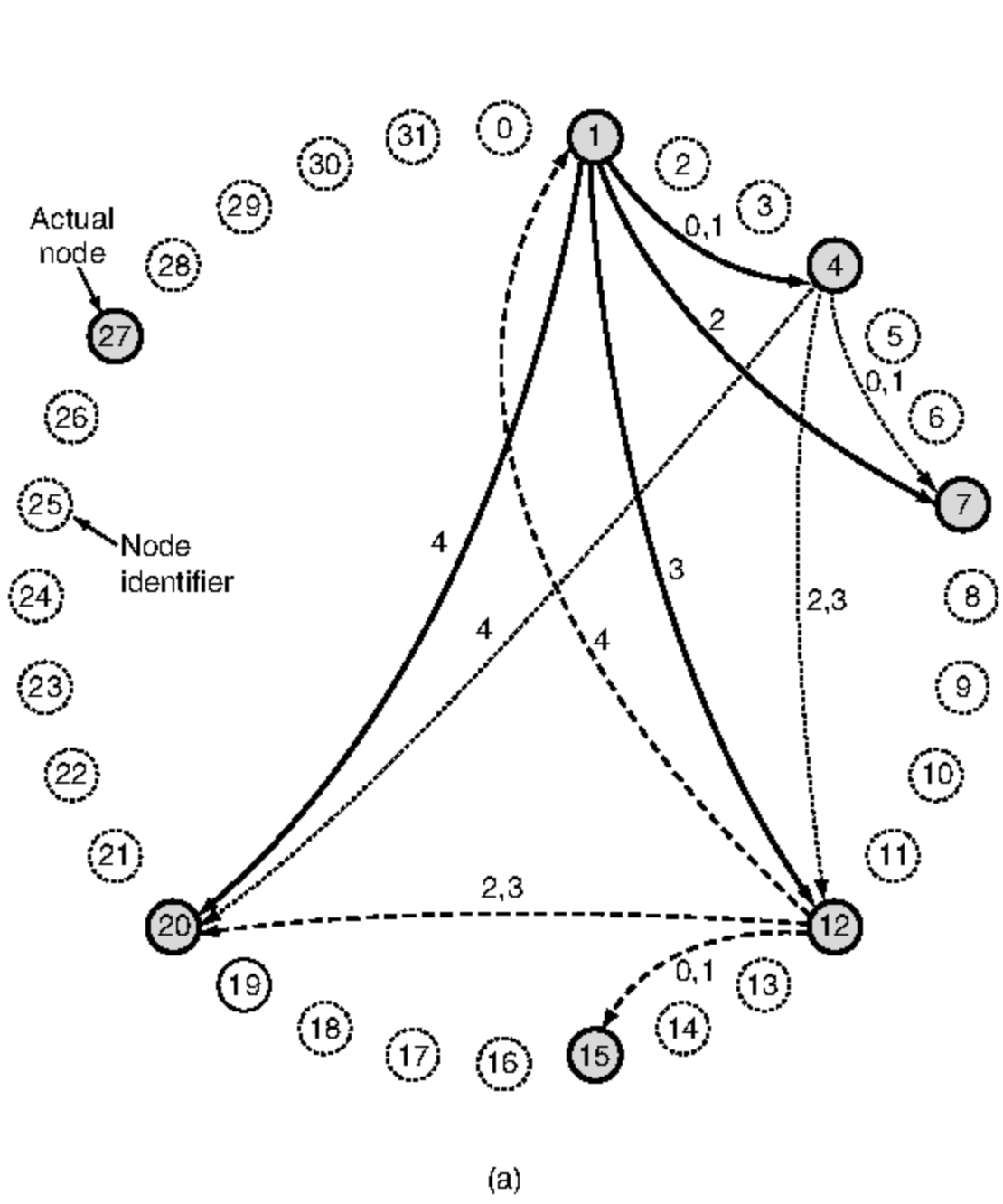

Figure 7-71. (a) A set of 32 node identifiers arranged in a circle. The shaded ones correspond to actual machines. The arcs show the fingers from nodes 1, 4, and 12. The labels on the arcs are the table indices. (b) Examples of the finger tables.

could be used to store the data at p nodes, but we will not consider the subject of fault tolerance further here.

Some time after the DHT is constructed, another node wants to find a torrent so that it can join the swarm and download content. A node looks up *torrent* by first hashing it to get *key*, and second using *successor* (*key*) to find the IP address of the node storing the corresponding value. The value is the list of peers in the swarm; the node can add its IP address to the list and contact the other peers to download content with the BitTorrent protocol.

The first step is easy; the second one is not easy. To make it possible to find the IP address of the node corresponding to a certain key, each node is required to

maintain certain administrative data structures. One of these is the IP address of its successor node along the node identifier circle. For example, in Fig. 7-71, node 4's successor is 7 and node 7's successor is 12.

Lookup can now proceed as follows. The requesting node sends a packet to its successor containing its IP address and the key it is looking for. The packet is propagated around the ring until it locates the successor to the node identifier being sought. That node checks to see if it has any information matching the key, and if so, returns it directly to the requesting node, whose IP address it has.

However, linearly searching all the nodes is very inefficient in a large peer-to-peer system since the mean number of nodes required per search is $n/2$. To greatly speed up the search, each node also maintains what Chord calls a **finger table**. The finger table has m entries, indexed by 0 through $m-1$, each one pointing to a different actual node. Each of the entries has two fields: *start* and the IP address of *successor*(*start*), as shown for three example nodes in Fig. 7-71(b). The values of the fields for entry i at a node with identifier k are:

$start = k + 2^i \ \ (modulo \ \ 2^m)$
IP address of *successor*(*start* [i])

Note that each node stores the IP addresses of a relatively small number of nodes and that most of these are fairly close by in terms of node identifier.

Using the finger table, the lookup of *key* at node k proceeds as follows. If *key* falls between k and *successor*(k), the node holding information about *key* is *successor*(k) and the search terminates. Otherwise, the finger table is searched to find the entry whose *start* field is the closest predecessor of *key*. A request is then sent directly to the IP address in that finger table entry to ask it to continue the search. Since it is closer to *key* but still below it, chances are good that it will be able to return the answer with only a small number of additional queries. In fact, since every lookup halves the remaining distance to the target, it can be shown that the average number of lookups is $\log_2 n$.

As a first example, consider looking up $key = 3$ at node 1. Since node 1 knows that 3 lies between it and its successor, 4, the desired node is 4 and the search terminates, returning node 4's IP address.

As a second example, consider looking up $key = 16$ at node 1. Since 16 does not lie between 1 and 4, the finger table is consulted. The closest predecessor to 16 is 9, so the request is forwarded to the IP address of 9's entry, namely, that of node 12. Node 12 also does not know the answer itself, so it looks for the node most closely preceding 16 and finds 14, which yields the IP address of node 15. A query is then sent there. Node 15 observes that 16 lies between it and its successor (20), so it returns the IP address of 20 to the caller, which works its way back to node 1.

Since nodes join and leave all the time, Chord needs a way to handle these operations. We assume that when the system began operation it was small enough that the nodes could just exchange information directly to build the first circle and

finger tables. After that, an automated procedure is needed. When a new node, r, wants to join, it must contact some existing node and ask it to look up the IP address of *successor*(r) for it. Next, the new node then asks *successor*(r) for its predecessor. The new node then asks both of these to insert r in between them in the circle. For example, if 24 in Fig. 7-71 wants to join, it asks any node to look up *successor*(24), which is 27. Then it asks 27 for its predecessor (20). After it tells both of those about its existence, 20 uses 24 as its successor and 27 uses 24 as its predecessor. In addition, node 27 hands over those keys in the range 21–24, which now belong to 24. At this point, 24 is fully inserted.

However, many finger tables are now wrong. To correct them, every node runs a background process that periodically recomputes each finger by calling *successor*. When one of these queries hits a new node, the corresponding finger entry is updated.

When a node leaves gracefully, it hands its keys over to its successor and informs its predecessor of its departure so the predecessor can link to the departing node's successor. When a node crashes, a problem arises because its predecessor no longer has a valid successor. To alleviate this problem, each node keeps track not only of its direct successor but also its s direct successors, to allow it to skip over up to $s - 1$ consecutive failed nodes and reconnect the circle if disaster strikes.

There has been a tremendous amount of research on DHTs since they were invented. To give you an idea of just how much research, let us pose a question: what is the most-cited networking paper of all time? You will find it difficult to come up with a paper that is cited more than the seminal Chord paper (Stoica et al., 2001). Despite this veritable mountain of research, applications of DHTs are only slowly beginning to emerge. Some BitTorrent clients use DHTs to provide a fully distributed tracker of the kind that we described. Large commercial cloud services such as Amazon's Dynamo also incorporate DHT techniques (DeCandia et al., 2007).

7.6 SUMMARY

Naming in the ARPANET started out in a very simple way: an ASCII text file listed the names of all the hosts and their corresponding IP addresses. Every night all the machines downloaded this file. But when the ARPANET morphed into the Internet and exploded in size, a far more sophisticated and dynamic naming scheme was required. The one used now is a hierarchical scheme called the Domain Name System. It organizes all the machines on the Internet into a set of trees. At the top level are the well-known generic domains, including *com* and *edu*, as well as about 200 country domains. DNS is implemented as a distributed database with servers all over the world. By querying a DNS server, a process

can map an Internet domain name onto the IP address used to communicate with a computer for that domain.

Email is the original killer app of the Internet. It is still widely used by everyone from small children to grandparents. Most email systems in the world use the mail system now defined in RFCs 5321 and 5322. Messages have simple ASCII headers, and many kinds of content can be sent using MIME. Mail is submitted to message transfer agents for delivery and retrieved from them for presentation by a variety of user agents, including Web applications. Submitted mail is delivered using SMTP, which works by making a TCP connection from the sending message transfer agent to the receiving one.

The Web is the application that most people think of as being the Internet. Originally, it was a system for seamlessly linking hypertext pages (written in HTML) across machines. The pages are downloaded by making a TCP connection from the browser to a server and using HTTP. Nowadays, much of the content on the Web is produced dynamically, either at the server (e.g., with PHP) or in the browser (e.g., with JavaScript). When combined with back-end databases, dynamic server pages allow Web applications such as e-commerce and search. Dynamic browser pages are evolving into full-featured applications, such as email, that run inside the browser and use the Web protocols to communicate with remote servers.

Caching and persistent connections are widely used to enhance Web performance. Using the Web on mobile devices can be challenging, despite the growth in the bandwidth and processing power of mobiles. Web sites often send tailored versions of pages with smaller images and less complex navigation to devices with small displays.

The Web protocols are increasingly being used for machine-to-machine communication. XML is preferred to HTML as a description of content that is easy for machines to process. SOAP is an RPC mechanism that sends XML messages using HTTP.

Digital audio and video have been key drivers for the Internet since 2000. The majority of Internet traffic today is video. Much of it is streamed from Web sites over a mix of protocols (including RTP/UDP and RTP/HTTP/TCP). Live media is streamed to many consumers. It includes Internet radio and TV stations that broadcast all manner of events. Audio and video are also used for real-time conferencing. Many calls use voice over IP, rather than the traditional telephone network, and include videoconferencing.

There are a small number of tremendously popular Web sites, as well as a very large number of less popular ones. To serve the popular sites, content distribution networks have been deployed. CDNs use DNS to direct clients to a nearby server; the servers are placed in data centers all around the world. Alternatively, P2P networks let a collection of machines share content such as movies among themselves. They provide a content distribution capacity that scales with the number of machines in the P2P network and which can rival the largest of sites.

PROBLEMS

1. Many business computers have three distinct and worldwide unique identifiers. What are they?
2. In Fig. 7-4, there is no period after *laserjet*. Why not?
3. Consider a situation in which a cyberterrorist makes all the DNS servers in the world crash simultaneously. How does this change one's ability to use the Internet?
4. DNS uses UDP instead of TCP. If a DNS packet is lost, there is no automatic recovery. Does this cause a problem, and if so, how is it solved?
5. John wants to have an original domain name and uses a randomized program to generate a secondary domain name for him. He wants to register this domain name in the *com* generic domain. The domain name that was generated is 253 characters long. Will the *com* registrar allow this domain name to be registered?
6. Can a machine with a single DNS name have multiple IP addresses? How could this occur?
7. The number of companies with a Web site has grown explosively in recent years. As a result, thousands of companies are registered in the *com* domain, causing a heavy load on the top-level server for this domain. Suggest a way to alleviate this problem without changing the naming scheme (i.e., without introducing new top-level domain names). It is permitted that your solution requires changes to the client code.
8. Some email systems support a *Content Return:* header field. It specifies whether the body of a message is to be returned in the event of nondelivery. Does this field belong to the envelope or to the header?
9. Electronic mail systems need directories so people's email addresses can be looked up. To build such directories, names should be broken up into standard components (e.g., first name, last name) to make searching possible. Discuss some problems that must be solved for a worldwide standard to be acceptable.
10. A large law firm, which has many employees, provides a single email address for each employee. Each employee's email address is *<login>@lawfirm.com*. However, the firm did not explicitly define the format of the login. Thus, some employees use their first names as their login names, some use their last names, some use their initials, etc. The firm now wishes to make a fixed format, for example:

    firstname.lastname@lawfirm.com,

    that can be used for the email addresses of all its employees. How can this be done without rocking the boat too much?
11. A binary file is 4560 bytes long. How long will it be if encoded using base64 encoding, with a CR+LF pair inserted after every 110 bytes sent and at the end?
12. Name five MIME types not listed in this book. You can check your browser or the Internet for information.

13. Suppose that you want to send an MP3 file to a friend, but your friend's ISP limits the size of each incoming message to 1 MB and the MP3 file is 4 MB. Is there a way to handle this situation by using RFC 5322 and MIME?

14. Suppose that John just set up an auto-forwarding mechanism on his work email address, which receives all of his business-related emails, to forward them to his personal email address, which he shares with his wife. John's wife was unaware of this, and activated a vacation agent on their personal account. Because John forwarded his email, he did not set up a vacation daemon on his work machine. What happens when an email is received at John's work email address?

15. In any standard, such as RFC 5322, a precise grammar of what is allowed is needed so that different implementations can interwork. Even simple items have to be defined carefully. The SMTP headers allow white space between the tokens. Give *two* plausible alternative definitions of white space between tokens.

16. Is the vacation agent part of the user agent or the message transfer agent? Of course, it is set up using the user agent, but does the user agent actually send the replies? Explain your answer.

17. In a simple version of the Chord algorithm for peer-to-peer lookup, searches do not use the finger table. Instead, they are linear around the circle, in either direction. Can a node accurately predict which direction it should search in? Discuss your answer.

18. IMAP allows users to fetch and download email from a remote mailbox. Does this mean that the internal format of mailboxes has to be standardized so any IMAP program on the client side can read the mailbox on any mail server? Discuss your answer.

19. Consider the Chord circle of Fig. 7-71. Suppose that node 18 suddenly goes online. Which of the finger tables shown in the figure are affected? how?

20. Does Webmail use POP3, IMAP, or neither? If one of these, why was that one chosen? If neither, which one is it closer to in spirit?

21. When Web pages are sent out, they are prefixed by MIME headers. Why?

22. Is it possible that when a user clicks on a link with Firefox, a particular helper is started, but clicking on the same link in Internet Explorer causes a completely different helper to be started, even though the MIME type returned in both cases is identical? Explain your answer.

23. Although it was not mentioned in the text, an alternative form for a URL is to use the IP address instead of its DNS name. Use this information to explain why a DNS name cannot end with a digit.

24. Imagine that someone in the math department at Stanford has just written a new document including a proof that he wants to distribute by FTP for his colleagues to review. He puts the program in the FTP directory *ftp/pub/forReview/newProof.pdf*. What is the URL for this program likely to be?

25. In Fig. 7-22, *www.aportal.com* keeps track of user preferences in a cookie. A disadvantage of this scheme is that cookies are limited to 4 KB, so if the preferences are

extensive, for example, many stocks, sports teams, types of news stories, weather for multiple cities, specials in numerous product categories, and more, the 4-KB limit may be reached. Design an alternative way to keep track of preferences that does not have this problem.

26. Sloth Bank wants to make online banking easy for its lazy customers, so after a customer signs up and is authenticated by a password, the bank returns a cookie containing a customer ID number. In this way, the customer does not have to identify himself or type a password on future visits to the online bank. What do you think of this idea? Will it work? Is it a good idea?

27. (a) Consider the following HTML tag:

```
<h1 title="this is the header"> HEADER 1 </h1>
```

Under what conditions does the browser use the *TITLE* attribute, and how?
(b) How does the *TITLE* attribute differ from the *ALT* attribute?

28. How do you make an image clickable in HTML? Give an example.

29. Write an HTML page that includes a link to the email address *username@DomainName.com*. What happens when a user clicks this link?

30. Write an XML page for a university registrar listing multiple students, each having a name, an address, and a GPA.

31. For each of the following applications, tell whether it would be (1) possible and (2) better to use a PHP script or JavaScript, and why:
(a) Displaying a calendar for any requested month since September 1752.
(b) Displaying the schedule of flights from Amsterdam to New York.
(c) Graphing a polynomial from user-supplied coefficients.

32. Write a program in JavaScript that accepts an integer greater than 2 and tells whether it is a prime number. Note that JavaScript has if and while statements with the same syntax as C and Java. The modulo operator is %. If you need the square root of x, use *Math.sqrt*(x).

33. An HTML page is as follows:

```
<html> <body>
<a href="www.info-source.com/welcome.html"> Click here for info </a>
</body> </html>
```

If the user clicks on the hyperlink, a TCP connection is opened and a series of lines is sent to the server. List all the lines sent.

34. The *If-Modified-Since* header can be used to check whether a cached page is still valid. Requests can be made for pages containing images, sound, video, and so on, as well as HTML. Do you think the effectiveness of this technique is better or worse for JPEG images as compared to HTML? Think carefully about what "effectiveness'' means and explain your answer.

35. On the day of a major sporting event, such as the championship game in some popular sport, many people go to the official Web site. Is this a flash crowd in the same sense as the 2000 Florida presidential election? Why or why not?

36. Does it make sense for a single ISP to function as a CDN? If so, how would that work? If not, what is wrong with the idea?

37. Assume that compression is not used for audio CDs. How many MB of data must the compact disc contain in order to be able to play two hours of music?

38. In Fig. 7-42(c), quantization noise occurs due to the use of 4-bit samples to represent nine signal values. The first sample, at 0, is exact, but the next few are not. What is the percent error for the samples at 1/32, 2/32, and 3/32 of the period?

39. Could a psychoacoustic model be used to reduce the bandwidth needed for Internet telephony? If so, what conditions, if any, would have to be met to make it work? If not, why not?

40. An audio streaming server has a one-way "distance" of 100 msec to a media player. It outputs at 1 Mbps. If the media player has a 2-MB buffer, what can you say about the position of the low-water mark and the high-water mark?

41. Does voice over IP have the same problems with firewalls that streaming audio does? Discuss your answer.

42. What is the bit rate for transmitting uncompressed 1200×800 pixel color frames with 16 bits/pixel at 50 frames/sec?

43. Can a 1-bit error in an MPEG frame affect more than the frame in which the error occurs? Explain your answer.

44. Consider a 50,000-customer video server, where each customer watches three movies per month. Two-thirds of the movies are served at 9 P.M. How many movies does the server have to transmit at once during this time period? If each movie requires 6 Mbps, how many OC-12 connections does the server need to the network?

45. Suppose that Zipf's law holds for accesses to a 10,000-movie video server. If the server holds the most popular 1000 movies in memory and the remaining 9000 on disk, give an expression for the fraction of all references that will be to memory. Write a little program to evaluate this expression numerically.

46. Some cybersquatters have registered domain names that are misspellings of common corporate sites, for example, *www.microsfot.com*. Make a list of at least five such domains.

47. Numerous people have registered DNS names that consist of *www.word.com*, where *word* is a common word. For each of the following categories, list five such Web sites and briefly summarize what it is (e.g., *www.stomach.com* belongs to a gastroenterologist on Long Island). Here is the list of categories: animals, foods, household objects, and body parts. For the last category, please stick to body parts above the waist.

48. Rewrite the server of Fig. 6-6 as a true Web server using the *GET* command for HTTP 1.1. It should also accept the *Host* message. The server should maintain a cache of files recently fetched from the disk and serve requests from the cache when possible.

8

NETWORK SECURITY

For the first few decades of their existence, computer networks were primarily used by university researchers for sending email and by corporate employees for sharing printers. Under these conditions, security did not get a lot of attention. But now, as millions of ordinary citizens are using networks for banking, shopping, and filing their tax returns, and weakness after weakness has been found, network security has become a problem of massive proportions. In this chapter, we will study network security from several angles, point out numerous pitfalls, and discuss many algorithms and protocols for making networks more secure.

Security is a broad topic and covers a multitude of sins. In its simplest form, it is concerned with making sure that nosy people cannot read, or worse yet, secretly modify messages intended for other recipients. It is concerned with people trying to access remote services that they are not authorized to use. It also deals with ways to tell whether that message purportedly from the IRS "Pay by Friday, or else" is really from the IRS and not from the Mafia. Security also deals with the problems of legitimate messages being captured and replayed, and with people later trying to deny that they sent certain messages.

Most security problems are intentionally caused by malicious people trying to gain some benefit, get attention, or harm someone. A few of the most common perpetrators are listed in Fig. 8-1. It should be clear from this list that making a network secure involves a lot more than just keeping it free of programming errors. It involves outsmarting often intelligent, dedicated, and sometimes well-funded adversaries. It should also be clear that measures that will thwart casual

attackers will have little impact on the serious ones. Police records show that the most damaging attacks are not perpetrated by outsiders tapping a phone line but by insiders bearing a grudge. Security systems should be designed accordingly.

Adversary	Goal
Student	To have fun snooping on people's email
Cracker	To test out someone's security system; steal data
Sales rep	To claim to represent all of Europe, not just Andorra
Corporation	To discover a competitor's strategic marketing plan
Ex-employee	To get revenge for being fired
Accountant	To embezzle money from a company
Stockbroker	To deny a promise made to a customer by email
Identity thief	To steal credit card numbers for sale
Government	To learn an enemy's military or industrial secrets
Terrorist	To steal biological warfare secrets

Figure 8-1. Some people who may cause security problems, and why.

Network security problems can be divided roughly into four closely intertwined areas: secrecy, authentication, nonrepudiation, and integrity control. Secrecy, also called confidentiality, has to do with keeping information out of the grubby little hands of unauthorized users. This is what usually comes to mind when people think about network security. Authentication deals with determining whom you are talking to before revealing sensitive information or entering into a business deal. Nonrepudiation deals with signatures: how do you prove that your customer really placed an electronic order for ten million left-handed doohickeys at 89 cents each when he later claims the price was 69 cents? Or maybe he claims he never placed any order. Finally, integrity control has to do with how you can be sure that a message you received was really the one sent and not something that a malicious adversary modified in transit or concocted.

All these issues (secrecy, authentication, nonrepudiation, and integrity control) occur in traditional systems, too, but with some significant differences. Integrity and secrecy are achieved by using registered mail and locking documents up. Robbing the mail train is harder now than it was in Jesse James' day.

Also, people can usually tell the difference between an original paper document and a photocopy, and it often matters to them. As a test, make a photocopy of a valid check. Try cashing the original check at your bank on Monday. Now try cashing the photocopy of the check on Tuesday. Observe the difference in the bank's behavior. With electronic checks, the original and the copy are indistinguishable. It may take a while for banks to learn how to handle this.

People authenticate other people by various means, including recognizing their faces, voices, and handwriting. Proof of signing is handled by signatures on letterhead paper, raised seals, and so on. Tampering can usually be detected by

handwriting, ink, and paper experts. None of these options are available electronically. Clearly, other solutions are needed.

Before getting into the solutions themselves, it is worth spending a few moments considering where in the protocol stack network security belongs. There is probably no one single place. Every layer has something to contribute. In the physical layer, wiretapping can be foiled by enclosing transmission lines (or better yet, optical fibers) in sealed tubes containing an inert gas at high pressure. Any attempt to drill into a tube will release some gas, reducing the pressure and triggering an alarm. Some military systems use this technique.

In the data link layer, packets on a point-to-point line can be encrypted as they leave one machine and decrypted as they enter another. All the details can be handled in the data link layer, with higher layers oblivious to what is going on. This solution breaks down when packets have to traverse multiple routers, however, because packets have to be decrypted at each router, leaving them vulnerable to attacks from within the router. Also, it does not allow some sessions to be protected (e.g., those involving online purchases by credit card) and others not. Nevertheless, **link encryption**, as this method is called, can be added to any network easily and is often useful.

In the network layer, firewalls can be installed to keep good packets and bad packets out. IP security also functions in this layer.

In the transport layer, entire connections can be encrypted end to end, that is, process to process. For maximum security, end-to-end security is required.

Finally, issues such as user authentication and nonrepudiation can only be handled in the application layer.

Since security does not fit neatly into any layer, it does not fit into any chapter of this book. For this reason, it rates its own chapter.

While this chapter is long, technical, and essential, it is also quasi-irrelevant for the moment. It is well documented that most security failures at banks, for example, are due to lax security procedures and incompetent employees, numerous implementation bugs that enable remote break-ins by unauthorized users, and so-called social engineering attacks, where customers are tricked into revealing their account details. All of these security problems are more prevalent than clever criminals tapping phone lines and then decoding encrypted messages. If a person can walk into a random branch of a bank with an ATM slip he found on the street claiming to have forgotten his PIN and get a new one on the spot (in the name of good customer relations), all the cryptography in the world will not prevent abuse. In this respect, Ross Anderson's (2008a) book is a real eye-opener, as it documents hundreds of examples of security failures in numerous industries, nearly all of them due to what might politely be called sloppy business practices or inattention to security. Nevertheless, the technical foundation on which e-commerce is built when all of these other factors are done well is cryptography.

Except for physical layer security, nearly all network security is based on cryptographic principles. For this reason, we will begin our study of security by

examining cryptography in some detail. In Sec. 8.1, we will look at some of the basic principles. In Sec. 8-2 through Sec. 8-5, we will examine some of the fundamental algorithms and data structures used in cryptography. Then we will examine in detail how these concepts can be used to achieve security in networks. We will conclude with some brief thoughts about technology and society.

Before starting, one last thought is in order: what is not covered. We have tried to focus on networking issues, rather than operating system and application issues, although the line is often hard to draw. For example, there is nothing here about user authentication using biometrics, password security, buffer overflow attacks, Trojan horses, login spoofing, code injection such as cross-site scripting, viruses, worms, and the like. All of these topics are covered at length in Chap. 9 of *Modern Operating Systems* (Tanenbaum, 2007). The interested reader is referred to that book for the systems aspects of security. Now let us begin our journey.

8.1 CRYPTOGRAPHY

Cryptography comes from the Greek words for "secret writing." It has a long and colorful history going back thousands of years. In this section, we will just sketch some of the highlights, as background information for what follows. For a complete history of cryptography, Kahn's (1995) book is recommended reading. For a comprehensive treatment of modern security and cryptographic algorithms, protocols, and applications, and related material, see Kaufman et al. (2002). For a more mathematical approach, see Stinson (2002). For a less mathematical approach, see Burnett and Paine (2001).

Professionals make a distinction between ciphers and codes. A **cipher** is a character-for-character or bit-for-bit transformation, without regard to the linguistic structure of the message. In contrast, a **code** replaces one word with another word or symbol. Codes are not used any more, although they have a glorious history. The most successful code ever devised was used by the U.S. armed forces during World War II in the Pacific. They simply had Navajo Indians talking to each other using specific Navajo words for military terms, for example *chay-da-gahi-nail-tsaidi* (literally: tortoise killer) for antitank weapon. The Navajo language is highly tonal, exceedingly complex, and has no written form. And not a single person in Japan knew anything about it.

In September 1945, the *San Diego Union* described the code by saying "For three years, wherever the Marines landed, the Japanese got an earful of strange gurgling noises interspersed with other sounds resembling the call of a Tibetan monk and the sound of a hot water bottle being emptied." The Japanese never broke the code and many Navajo code talkers were awarded high military honors for extraordinary service and bravery. The fact that the U.S. broke the Japanese code but the Japanese never broke the Navajo code played a crucial role in the American victories in the Pacific.

8.1.1 Introduction to Cryptography

Historically, four groups of people have used and contributed to the art of cryptography: the military, the diplomatic corps, diarists, and lovers. Of these, the military has had the most important role and has shaped the field over the centuries. Within military organizations, the messages to be encrypted have traditionally been given to poorly paid, low-level code clerks for encryption and transmission. The sheer volume of messages prevented this work from being done by a few elite specialists.

Until the advent of computers, one of the main constraints on cryptography had been the ability of the code clerk to perform the necessary transformations, often on a battlefield with little equipment. An additional constraint has been the difficulty in switching over quickly from one cryptographic method to another one, since this entails retraining a large number of people. However, the danger of a code clerk being captured by the enemy has made it essential to be able to change the cryptographic method instantly if need be. These conflicting requirements have given rise to the model of Fig. 8-2.

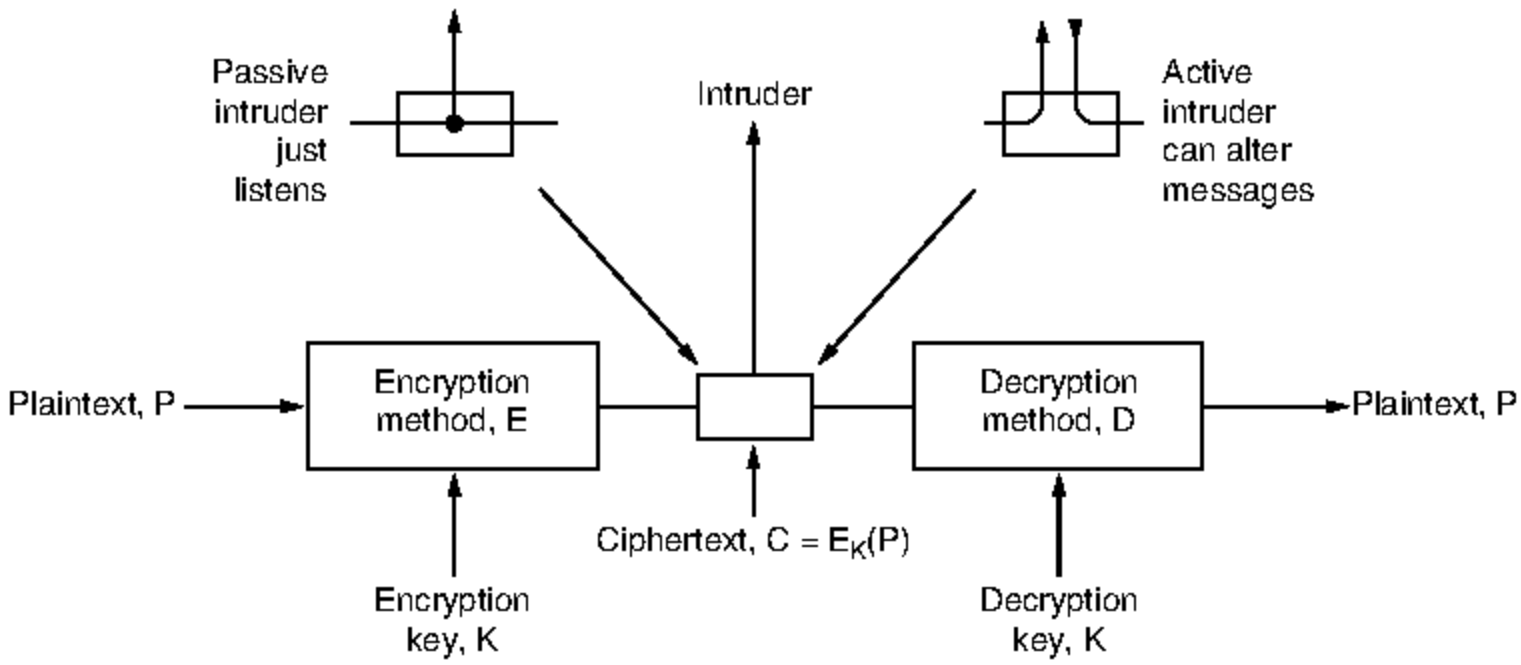

Figure 8-2. The encryption model (for a symmetric-key cipher).

The messages to be encrypted, known as the **plaintext**, are transformed by a function that is parameterized by a **key**. The output of the encryption process, known as the **ciphertext**, is then transmitted, often by messenger or radio. We assume that the enemy, or **intruder**, hears and accurately copies down the complete ciphertext. However, unlike the intended recipient, he does not know what the decryption key is and so cannot decrypt the ciphertext easily. Sometimes the intruder can not only listen to the communication channel (passive intruder) but can also record messages and play them back later, inject his own messages, or modify legitimate messages before they get to the receiver (active intruder). The art of

breaking ciphers, known as **cryptanalysis**, and the art of devising them (cryptography) are collectively known as **cryptology**.

It will often be useful to have a notation for relating plaintext, ciphertext, and keys. We will use $C = E_K(P)$ to mean that the encryption of the plaintext P using key K gives the ciphertext C. Similarly, $P = D_K(C)$ represents the decryption of C to get the plaintext again. It then follows that

$$D_K(E_K(P)) = P$$

This notation suggests that E and D are just mathematical functions, which they are. The only tricky part is that both are functions of two parameters, and we have written one of the parameters (the key) as a subscript, rather than as an argument, to distinguish it from the message.

A fundamental rule of cryptography is that one must assume that the cryptanalyst knows the methods used for encryption and decryption. In other words, the cryptanalyst knows how the encryption method, E, and decryption, D, of Fig. 8-2 work in detail. The amount of effort necessary to invent, test, and install a new algorithm every time the old method is compromised (or thought to be compromised) has always made it impractical to keep the encryption algorithm secret. Thinking it is secret when it is not does more harm than good.

This is where the key enters. The key consists of a (relatively) short string that selects one of many potential encryptions. In contrast to the general method, which may only be changed every few years, the key can be changed as often as required. Thus, our basic model is a stable and publicly known general method parameterized by a secret and easily changed key. The idea that the cryptanalyst knows the algorithms and that the secrecy lies exclusively in the keys is called **Kerckhoff's principle**, named after the Flemish military cryptographer Auguste Kerckhoff who first stated it in 1883 (Kerckhoff, 1883). Thus, we have

Kerckhoff's principle: All algorithms must be public; only the keys are secret

The nonsecrecy of the algorithm cannot be emphasized enough. Trying to keep the algorithm secret, known in the trade as **security by obscurity**, never works. Also, by publicizing the algorithm, the cryptographer gets free consulting from a large number of academic cryptologists eager to break the system so they can publish papers demonstrating how smart they are. If many experts have tried to break the algorithm for a long time after its publication and no one has succeeded, it is probably pretty solid.

Since the real secrecy is in the key, its length is a major design issue. Consider a simple combination lock. The general principle is that you enter digits in sequence. Everyone knows this, but the key is secret. A key length of two digits means that there are 100 possibilities. A key length of three digits means 1000 possibilities, and a key length of six digits means a million. The longer the key, the higher the **work factor** the cryptanalyst has to deal with. The work factor for breaking the system by exhaustive search of the key space is exponential in the

key length. Secrecy comes from having a strong (but public) algorithm and a long key. To prevent your kid brother from reading your email, 64-bit keys will do. For routine commercial use, at least 128 bits should be used. To keep major governments at bay, keys of at least 256 bits, preferably more, are needed.

From the cryptanalyst's point of view, the cryptanalysis problem has three principal variations. When he has a quantity of ciphertext and no plaintext, he is confronted with the **ciphertext-only** problem. The cryptograms that appear in the puzzle section of newspapers pose this kind of problem. When the cryptanalyst has some matched ciphertext and plaintext, the problem is called the **known plaintext** problem. Finally, when the cryptanalyst has the ability to encrypt pieces of plaintext of his own choosing, we have the **chosen plaintext** problem. Newspaper cryptograms could be broken trivially if the cryptanalyst were allowed to ask such questions as "What is the encryption of ABCDEFGHIJKL?"

Novices in the cryptography business often assume that if a cipher can withstand a ciphertext-only attack, it is secure. This assumption is very naive. In many cases, the cryptanalyst can make a good guess at parts of the plaintext. For example, the first thing many computers say when you call them up is "login:". Equipped with some matched plaintext-ciphertext pairs, the cryptanalyst's job becomes much easier. To achieve security, the cryptographer should be conservative and make sure that the system is unbreakable even if his opponent can encrypt arbitrary amounts of chosen plaintext.

Encryption methods have historically been divided into two categories: substitution ciphers and transposition ciphers. We will now deal with each of these briefly as background information for modern cryptography.

8.1.2 Substitution Ciphers

In a **substitution cipher**, each letter or group of letters is replaced by another letter or group of letters to disguise it. One of the oldest known ciphers is the **Caesar cipher**, attributed to Julius Caesar. With this method, *a* becomes *D*, *b* becomes *E*, *c* becomes *F*, . . . , and *z* becomes *C*. For example, *attack* becomes *DWWDFN*. In our examples, plaintext will be given in lowercase letters, and ciphertext in uppercase letters.

A slight generalization of the Caesar cipher allows the ciphertext alphabet to be shifted by k letters, instead of always three. In this case, k becomes a key to the general method of circularly shifted alphabets. The Caesar cipher may have fooled Pompey, but it has not fooled anyone since.

The next improvement is to have each of the symbols in the plaintext, say, the 26 letters for simplicity, map onto some other letter. For example,

plaintext:	a b c d e f g h i j k l m n o p q r s t u v w x y z
ciphertext:	Q W E R T Y U I O P A S D F G H J K L Z X C V B N M

The general system of symbol-for-symbol substitution is called a **monoalphabetic substitution cipher**, with the key being the 26-letter string corresponding to the full alphabet. For the key just given, the plaintext *attack* would be transformed into the ciphertext *QZZQEA*.

At first glance this might appear to be a safe system because although the cryptanalyst knows the general system (letter-for-letter substitution), he does not know which of the $26! \approx 4 \times 10^{26}$ possible keys is in use. In contrast with the Caesar cipher, trying all of them is not a promising approach. Even at 1 nsec per solution, a million computer chips working in parallel would take 10,000 years to try all the keys.

Nevertheless, given a surprisingly small amount of ciphertext, the cipher can be broken easily. The basic attack takes advantage of the statistical properties of natural languages. In English, for example, *e* is the most common letter, followed by *t*, *o*, *a*, *n*, *i*, etc. The most common two-letter combinations, or **digrams**, are *th*, *in*, *er*, *re*, and *an*. The most common three-letter combinations, or **trigrams**, are *the*, *ing*, *and*, and *ion*.

A cryptanalyst trying to break a monoalphabetic cipher would start out by counting the relative frequencies of all letters in the ciphertext. Then he might tentatively assign the most common one to *e* and the next most common one to *t*. He would then look at trigrams to find a common one of the form *tXe*, which strongly suggests that *X* is *h*. Similarly, if the pattern *thYt* occurs frequently, the *Y* probably stands for *a*. With this information, he can look for a frequently occurring trigram of the form *aZW*, which is most likely *and*. By making guesses at common letters, digrams, and trigrams and knowing about likely patterns of vowels and consonants, the cryptanalyst builds up a tentative plaintext, letter by letter.

Another approach is to guess a probable word or phrase. For example, consider the following ciphertext from an accounting firm (blocked into groups of five characters):

```
CTBMN BYCTC BTJDS QXBNS GSTJC BTSWX CTQTZ CQVUJ
QJSGS TJQZZ MNQJS VLNSX VSZJU JDSTS JQUUS JUBXJ
DSKSU JSNTK BGAQJ ZBGYQ TLCTZ BNYBN QJSW
```

A likely word in a message from an accounting firm is *financial*. Using our knowledge that *financial* has a repeated letter (*i*), with four other letters between their occurrences, we look for repeated letters in the ciphertext at this spacing. We find 12 hits, at positions 6, 15, 27, 31, 42, 48, 56, 66, 70, 71, 76, and 82. However, only two of these, 31 and 42, have the next letter (corresponding to *n* in the plaintext) repeated in the proper place. Of these two, only 31 also has the *a* correctly positioned, so we know that *financial* begins at position 30. From this point on, deducing the key is easy by using the frequency statistics for English text and looking for nearly complete words to finish off.

8.1.3 Transposition Ciphers

Substitution ciphers preserve the order of the plaintext symbols but disguise them. **Transposition ciphers**, in contrast, reorder the letters but do not disguise them. Figure 8-3 depicts a common transposition cipher, the columnar transposition. The cipher is keyed by a word or phrase not containing any repeated letters. In this example, MEGABUCK is the key. The purpose of the key is to order the columns, with column 1 being under the key letter closest to the start of the alphabet, and so on. The plaintext is written horizontally, in rows, padded to fill the matrix if need be. The ciphertext is read out by columns, starting with the column whose key letter is the lowest.

M	E	G	A	B	U	C	K
7	4	5	1	2	8	3	6
p	l	e	a	s	e	t	r
a	n	s	f	e	r	o	n
e	m	i	l	l	i	o	n
d	o	l	l	a	r	s	t
o	m	y	s	w	i	s	s
b	a	n	k	a	c	c	o
u	n	t	s	i	x	t	w
o	t	w	o	a	b	c	d

Plaintext

pleasetransferonemilliondollarsto
myswissbankaccountsixtwotwo

Ciphertext

AFLLSKSOSELAWAIATOOSSCTCLNMOMANT
ESILYNTWRNNTSOWDPAEDOBUOERIRICXB

Figure 8-3. A transposition cipher.

To break a transposition cipher, the cryptanalyst must first be aware that he is dealing with a transposition cipher. By looking at the frequency of *E*, *T*, *A*, *O*, *I*, *N*, etc., it is easy to see if they fit the normal pattern for plaintext. If so, the cipher is clearly a transposition cipher, because in such a cipher every letter represents itself, keeping the frequency distribution intact.

The next step is to make a guess at the number of columns. In many cases, a probable word or phrase may be guessed at from the context. For example, suppose that our cryptanalyst suspects that the plaintext phrase *milliondollars* occurs somewhere in the message. Observe that digrams *MO*, *IL*, *LL*, *LA*, *IR*, and *OS* occur in the ciphertext as a result of this phrase wrapping around. The ciphertext letter *O* follows the ciphertext letter *M* (i.e., they are vertically adjacent in column 4) because they are separated in the probable phrase by a distance equal to the key length. If a key of length seven had been used, the digrams *MD*, *IO*, *LL*, *LL*, *IA*, *OR*, and *NS* would have occurred instead. In fact, for each key length, a different set of digrams is produced in the ciphertext. By hunting for the various possibilities, the cryptanalyst can often easily determine the key length.

The remaining step is to order the columns. When the number of columns, k, is small, each of the $k(k-1)$ column pairs can be examined in turn to see if its digram frequencies match those for English plaintext. The pair with the best match is assumed to be correctly positioned. Now each of the remaining columns is tentatively tried as the successor to this pair. The column whose digram and trigram frequencies give the best match is tentatively assumed to be correct. The next column is found in the same way. The entire process is continued until a potential ordering is found. Chances are that the plaintext will be recognizable at this point (e.g., if *milloin* occurs, it is clear what the error is).

Some transposition ciphers accept a fixed-length block of input and produce a fixed-length block of output. These ciphers can be completely described by giving a list telling the order in which the characters are to be output. For example, the cipher of Fig. 8-3 can be seen as a 64 character block cipher. Its output is 4, 12, 20, 28, 36, 44, 52, 60, 5, 13, . . . , 62. In other words, the fourth input character, *a*, is the first to be output, followed by the twelfth, *f*, and so on.

8.1.4 One-Time Pads

Constructing an unbreakable cipher is actually quite easy; the technique has been known for decades. First choose a random bit string as the key. Then convert the plaintext into a bit string, for example, by using its ASCII representation. Finally, compute the XOR (eXclusive OR) of these two strings, bit by bit. The resulting ciphertext cannot be broken because in a sufficiently large sample of ciphertext, each letter will occur equally often, as will every digram, every trigram, and so on. This method, known as the **one-time pad**, is immune to all present and future attacks, no matter how much computational power the intruder has. The reason derives from information theory: there is simply no information in the message because all possible plaintexts of the given length are equally likely.

An example of how one-time pads are used is given in Fig. 8-4. First, message 1, "I love you." is converted to 7-bit ASCII. Then a one-time pad, pad 1, is chosen and XORed with the message to get the ciphertext. A cryptanalyst could try all possible one-time pads to see what plaintext came out for each one. For example, the one-time pad listed as pad 2 in the figure could be tried, resulting in plaintext 2, "Elvis lives'', which may or may not be plausible (a subject beyond the scope of this book). In fact, for every 11-character ASCII plaintext, there is a one-time pad that generates it. That is what we mean by saying there is no information in the ciphertext: you can get any message of the correct length out of it.

One-time pads are great in theory but have a number of disadvantages in practice. To start with, the key cannot be memorized, so both sender and receiver must carry a written copy with them. If either one is subject to capture, written keys are clearly undesirable. Additionally, the total amount of data that can be transmitted is limited by the amount of key available. If the spy strikes it rich and discovers a wealth of data, he may find himself unable to transmit them back to

Message 1:	1001001	0100000	1101100	1101111	1110110	1100101	0100000	1111001	1101111	1110101	0101110
Pad 1:	1010010	1001011	1110010	1010101	1010010	1100011	0001011	0101010	1010111	1100110	0101011
Ciphertext:	0011011	1101011	0011110	0111010	0100100	0000110	0101011	1010011	0111000	0010011	0000101
Pad 2:	1011110	0000111	1101000	1010011	1010111	0100110	1000111	0111010	1001110	1110110	1110110
Plaintext 2:	1000101	1101100	1110110	1101001	1110011	0100000	1101100	1101001	1110110	1100101	1110011

Figure 8-4. The use of a one-time pad for encryption and the possibility of getting any possible plaintext from the ciphertext by the use of some other pad.

headquarters because the key has been used up. Another problem is the sensitivity of the method to lost or inserted characters. If the sender and receiver get out of synchronization, all data from then on will appear garbled.

With the advent of computers, the one-time pad might potentially become practical for some applications. The source of the key could be a special DVD that contains several gigabytes of information and, if transported in a DVD movie box and prefixed by a few minutes of video, would not even be suspicious. Of course, at gigabit network speeds, having to insert a new DVD every 30 sec could become tedious. And the DVDs must be personally carried from the sender to the receiver before any messages can be sent, which greatly reduces their practical utility.

Quantum Cryptography

Interestingly, there may be a solution to the problem of how to transmit the one-time pad over the network, and it comes from a very unlikely source: quantum mechanics. This area is still experimental, but initial tests are promising. If it can be perfected and be made efficient, virtually all cryptography will eventually be done using one-time pads since they are provably secure. Below we will briefly explain how this method, **quantum cryptography**, works. In particular, we will describe a protocol called **BB84** after its authors and publication year (Bennet and Brassard, 1984).

Suppose that a user, Alice, wants to establish a one-time pad with a second user, Bob. Alice and Bob are called **principals**, the main characters in our story. For example, Bob is a banker with whom Alice would like to do business. The names "Alice" and "Bob" have been used for the principals in virtually every paper and book on cryptography since Ron Rivest introduced them many years ago (Rivest et al., 1978). Cryptographers love tradition. If we were to use "Andy" and "Barbara" as the principals, no one would believe anything in this chapter. So be it.

If Alice and Bob could establish a one-time pad, they could use it to communicate securely. The question is: how can they establish it without previously exchanging DVDs? We can assume that Alice and Bob are at the opposite ends

of an optical fiber over which they can send and receive light pulses. However, an intrepid intruder, Trudy, can cut the fiber to splice in an active tap. Trudy can read all the bits sent in both directions. She can also send false messages in both directions. The situation might seem hopeless for Alice and Bob, but quantum cryptography can shed some new light on the subject.

Quantum cryptography is based on the fact that light comes in little packets called **photons**, which have some peculiar properties. Furthermore, light can be polarized by being passed through a polarizing filter, a fact well known to both sunglasses wearers and photographers. If a beam of light (i.e., a stream of photons) is passed through a polarizing filter, all the photons emerging from it will be polarized in the direction of the filter's axis (e.g., vertically). If the beam is now passed through a second polarizing filter, the intensity of the light emerging from the second filter is proportional to the square of the cosine of the angle between the axes. If the two axes are perpendicular, no photons get through. The absolute orientation of the two filters does not matter; only the angle between their axes counts.

To generate a one-time pad, Alice needs two sets of polarizing filters. Set one consists of a vertical filter and a horizontal filter. This choice is called a **rectilinear basis**. A basis (plural: bases) is just a coordinate system. The second set of filters is the same, except rotated 45 degrees, so one filter runs from the lower left to the upper right and the other filter runs from the upper left to the lower right. This choice is called a **diagonal basis**. Thus, Alice has two bases, which she can rapidly insert into her beam at will. In reality, Alice does not have four separate filters, but a crystal whose polarization can be switched electrically to any of the four allowed directions at great speed. Bob has the same equipment as Alice. The fact that Alice and Bob each have two bases available is essential to quantum cryptography.

For each basis, Alice now assigns one direction as 0 and the other as 1. In the example presented below, we assume she chooses vertical to be 0 and horizontal to be 1. Independently, she also chooses lower left to upper right as 0 and upper left to lower right as 1. She sends these choices to Bob as plaintext.

Now Alice picks a one-time pad, for example based on a random number generator (a complex subject all by itself). She transfers it bit by bit to Bob, choosing one of her two bases at random for each bit. To send a bit, her photon gun emits one photon polarized appropriately for the basis she is using for that bit. For example, she might choose bases of diagonal, rectilinear, rectilinear, diagonal, rectilinear, etc. To send her one-time pad of 1001110010100110 with these bases, she would send the photons shown in Fig. 8-5(a). Given the one-time pad and the sequence of bases, the polarization to use for each bit is uniquely determined. Bits sent one photon at a time are called **qubits**.

Bob does not know which bases to use, so he picks one at random for each arriving photon and just uses it, as shown in Fig. 8-5(b). If he picks the correct basis, he gets the correct bit. If he picks the incorrect basis, he gets a random bit

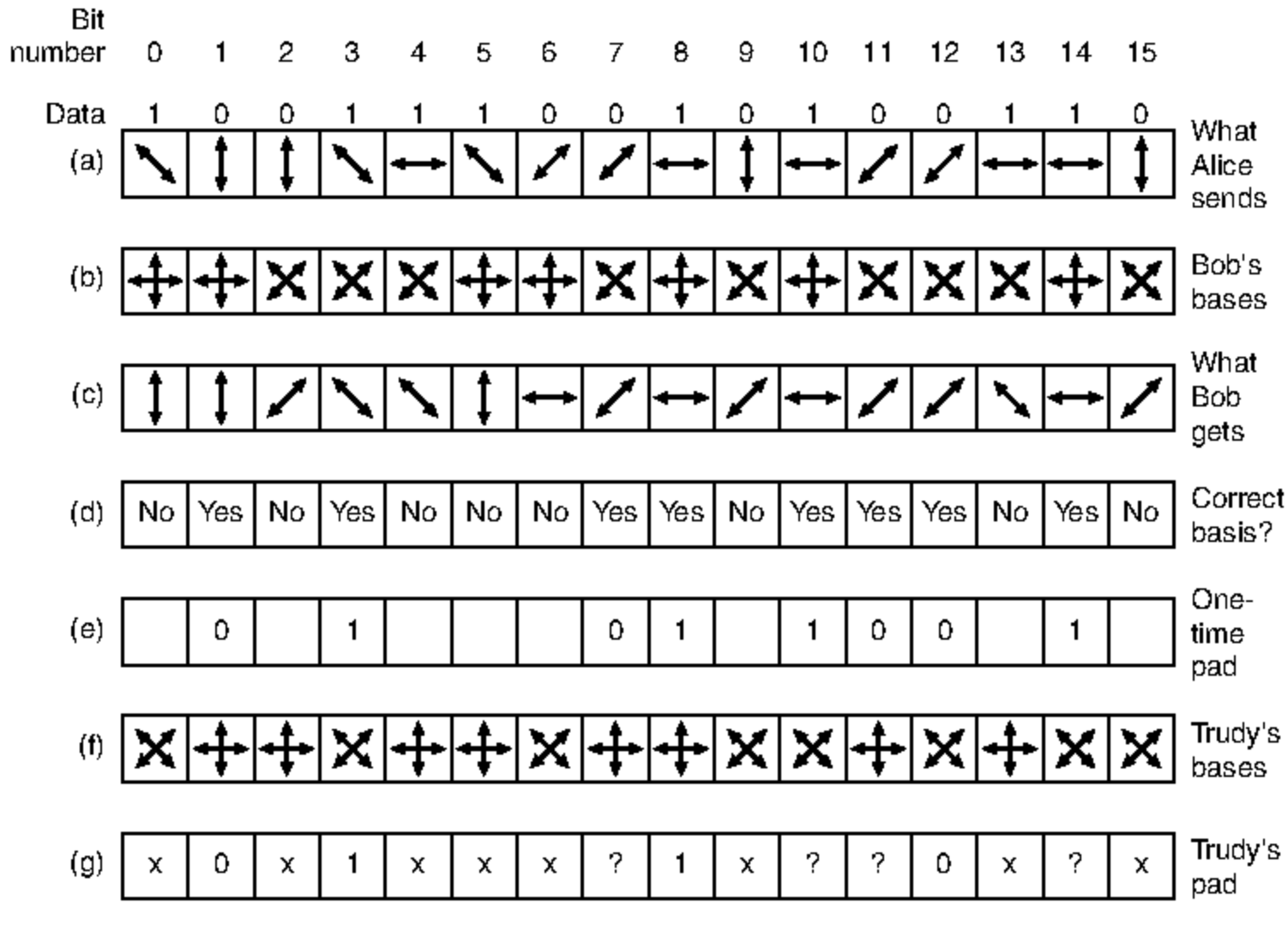

Figure 8-5. An example of quantum cryptography.

because if a photon hits a filter polarized at 45 degrees to its own polarization, it randomly jumps to the polarization of the filter or to a polarization perpendicular to the filter, with equal probability. This property of photons is fundamental to quantum mechanics. Thus, some of the bits are correct and some are random, but Bob does not know which are which. Bob's results are depicted in Fig. 8-5(c).

How does Bob find out which bases he got right and which he got wrong? He simply tells Alice which basis he used for each bit in plaintext and she tells him which are right and which are wrong in plaintext, as shown in Fig. 8-5(d). From this information, both of them can build a bit string from the correct guesses, as shown in Fig. 8-5(e). On the average, this bit string will be half the length of the original bit string, but since both parties know it, they can use it as a one-time pad. All Alice has to do is transmit a bit string slightly more than twice the desired length, and she and Bob will have a one-time pad of the desired length. Done.

But wait a minute. We forgot Trudy. Suppose that she is curious about what Alice has to say and cuts the fiber, inserting her own detector and transmitter. Unfortunately for her, she does not know which basis to use for each photon either. The best she can do is pick one at random for each photon, just as Bob does. An example of her choices is shown in Fig. 8-5(f). When Bob later reports (in plaintext) which bases he used and Alice tells him (in plaintext) which ones are

correct, Trudy now knows when she got it right and when she got it wrong. In Fig. 8-5, she got it right for bits 0, 1, 2, 3, 4, 6, 8, 12, and 13. But she knows from Alice's reply in Fig. 8-5(d) that only bits 1, 3, 7, 8, 10, 11, 12, and 14 are part of the one-time pad. For four of these bits (1, 3, 8, and 12), she guessed right and captured the correct bit. For the other four (7, 10, 11, and 14), she guessed wrong and does not know the bit transmitted. Thus, Bob knows the one-time pad starts with 01011001, from Fig. 8-5(e) but all Trudy has is 01?1??0?, from Fig. 8-5(g).

Of course, Alice and Bob are aware that Trudy may have captured part of their one-time pad, so they would like to reduce the information Trudy has. They can do this by performing a transformation on it. For example, they could divide the one-time pad into blocks of 1024 bits, square each one to form a 2048-bit number, and use the concatenation of these 2048-bit numbers as the one-time pad. With her partial knowledge of the bit string transmitted, Trudy has no way to generate its square and so has nothing. The transformation from the original one-time pad to a different one that reduces Trudy's knowledge is called **privacy amplification**. In practice, complex transformations in which every output bit depends on every input bit are used instead of squaring.

Poor Trudy. Not only does she have no idea what the one-time pad is, but her presence is not a secret either. After all, she must relay each received bit to Bob to trick him into thinking he is talking to Alice. The trouble is, the best she can do is transmit the qubit she received, using the polarization she used to receive it, and about half the time she will be wrong, causing many errors in Bob's one-time pad.

When Alice finally starts sending data, she encodes it using a heavy forward-error-correcting code. From Bob's point of view, a 1-bit error in the one-time pad is the same as a 1-bit transmission error. Either way, he gets the wrong bit. If there is enough forward error correction, he can recover the original message despite all the errors, but he can easily count how many errors were corrected. If this number is far more than the expected error rate of the equipment, he knows that Trudy has tapped the line and can act accordingly (e.g., tell Alice to switch to a radio channel, call the police, etc.). If Trudy had a way to clone a photon so she had one photon to inspect and an identical photon to send to Bob, she could avoid detection, but at present no way to clone a photon perfectly is known. And even if Trudy could clone photons, the value of quantum cryptography to establish one-time pads would not be reduced.

Although quantum cryptography has been shown to operate over distances of 60 km of fiber, the equipment is complex and expensive. Still, the idea has promise. For more information about quantum cryptography, see Mullins (2002).

8.1.5 Two Fundamental Cryptographic Principles

Although we will study many different cryptographic systems in the pages ahead, two principles underlying all of them are important to understand. Pay attention. You violate them at your peril.