Type	Meaning	Value
SOA	Start of authority	Parameters for this zone
A	IPv4 address of a host	32-Bit integer
AAAA	IPv6 address of a host	128-Bit integer
MX	Mail exchange	Priority, domain willing to accept email
NS	Name server	Name of a server for this domain
CNAME	Canonical name	Domain name
PTR	Pointer	Alias for an IP address
SPF	Sender policy framework	Text encoding of mail sending policy
SRV	Service	Host that provides it
TXT	Text	Descriptive ASCII text

Figure 7-3. The principal DNS resource record types.

machine is prepared to accept email. If someone wants to send email to, for example, *bill@microsoft.com*, the sending host needs to find some mail server located at *microsoft.com* that is willing to accept email. The *MX* record can provide this information.

Another important record type is the *NS* record. It specifies a name server for the domain or subdomain. This is a host that has a copy of the database for a domain. It is used as part of the process to look up names, which we will describe shortly.

CNAME records allow aliases to be created. For example, a person familiar with Internet naming in general and wanting to send a message to user *paul* in the computer science department at M.I.T. might guess that *paul@cs.mit.edu* will work. Actually, this address will not work, because the domain for M.I.T.'s computer science department is *csail.mit.edu*. However, as a service to people who do not know this, M.I.T. could create a *CNAME* entry to point people and programs in the right direction. An entry like this one might do the job:

```
cs.mit.edu    86400    IN    CNAME    csail.mit.edu
```

Like *CNAME*, *PTR* points to another name. However, unlike *CNAME*, which is really just a macro definition (i.e., a mechanism to replace one string by another), *PTR* is a regular DNS data type whose interpretation depends on the context in which it is found. In practice, it is nearly always used to associate a name with an IP address to allow lookups of the IP address and return the name of the corresponding machine. These are called **reverse lookups**.

SRV is a newer type of record that allows a host to be identified for a given service in a domain. For example, the Web server for *cs.washington.edu* could be identified as *cockatoo.cs.washington.edu*. This record generalizes the *MX* record that performs the same task but it is just for mail servers.

SPF is also a newer type of record. It lets a domain encode information about what machines in the domain will send mail to the rest of the Internet. This helps receiving machines check that mail is valid. If mail is being received from a machine that calls itself *dodgy* but the domain records say that mail will only be sent out of the domain by a machine called *smtp*, chances are that the mail is forged junk mail.

Last on the list, *TXT* records were originally provided to allow domains to identify themselves in arbitrary ways. Nowadays, they usually encode machine-readable information, typically the *SPF* information.

Finally, we have the *Value* field. This field can be a number, a domain name, or an ASCII string. The semantics depend on the record type. A short description of the *Value* fields for each of the principal record types is given in Fig. 7-3.

For an example of the kind of information one might find in the DNS database of a domain, see Fig. 7-4. This figure depicts part of a (hypothetical) database for the *cs.vu.nl* domain shown in Fig. 7-1. The database contains seven types of resource records.

```
; Authoritative data for cs.vu.nl
cs.vu.nl.       86400   IN  SOA     star boss (9527,7200,7200,241920,86400)
cs.vu.nl.       86400   IN  MX      1 zephyr
cs.vu.nl.       86400   IN  MX      2 top
cs.vu.nl.       86400   IN  NS      star

star            86400   IN  A       130.37.56.205
zephyr          86400   IN  A       130.37.20.10
top             86400   IN  A       130.37.20.11
www             86400   IN  CNAME   star.cs.vu.nl
ftp             86400   IN  CNAME   zephyr.cs.vu.nl

flits           86400   IN  A       130.37.16.112
flits           86400   IN  A       192.31.231.165
flits           86400   IN  MX      1 flits
flits           86400   IN  MX      2 zephyr
flits           86400   IN  MX      3 top

rowboat                 IN  A       130.37.56.201
                        IN  MX      1 rowboat
                        IN  MX      2 zephyr

little-sister           IN  A       130.37.62.23

laserjet                IN  A       192.31.231.216
```

Figure 7-4. A portion of a possible DNS database for *cs.vu.nl*.

The first noncomment line of Fig. 7-4 gives some basic information about the domain, which will not concern us further. Then come two entries giving the first

and second places to try to deliver email sent to *person@cs.vu.nl*. The *zephyr* (a specific machine) should be tried first. If that fails, the *top* should be tried as the next choice. The next line identifies the name server for the domain as *star*.

After the blank line (added for readability) come lines giving the IP addresses for the *star*, *zephyr*, and *top*. These are followed by an alias, *www.cs.vu.nl*, so that this address can be used without designating a specific machine. Creating this alias allows *cs.vu.nl* to change its World Wide Web server without invalidating the address people use to get to it. A similar argument holds for *ftp.cs.vu.nl*.

The section for the machine *flits* lists two IP addresses and three choices are given for handling email sent to *flits.cs.vu.nl*. First choice is naturally the *flits* itself, but if it is down, the *zephyr* and *top* are the second and third choices.

The next three lines contain a typical entry for a computer, in this case, *rowboat.cs.vu.nl*. The information provided contains the IP address and the primary and secondary mail drops. Then comes an entry for a computer that is not capable of receiving mail itself, followed by an entry that is likely for a printer that is connected to the Internet.

7.1.3 Name Servers

In theory at least, a single name server could contain the entire DNS database and respond to all queries about it. In practice, this server would be so overloaded as to be useless. Furthermore, if it ever went down, the entire Internet would be crippled.

To avoid the problems associated with having only a single source of information, the DNS name space is divided into nonoverlapping **zones**. One possible way to divide the name space of Fig. 7-1 is shown in Fig. 7-5. Each circled zone contains some part of the tree.

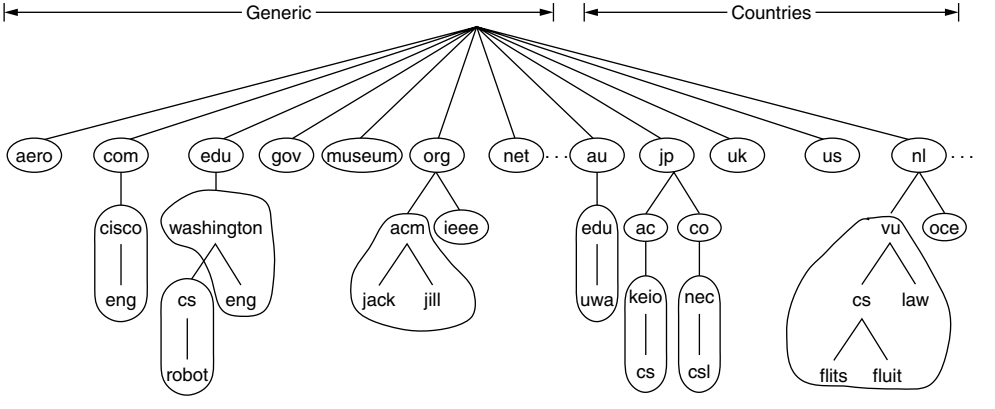


Figure 7-5. Part of the DNS name space divided into zones (which are circled).

Where the zone boundaries are placed within a zone is up to that zone's administrator. This decision is made in large part based on how many name servers are desired, and where. For example, in Fig. 7-5, the University of Washington has a zone for *washington.edu* that handles *eng.washington.edu* but does not handle *cs.washington.edu*. That is a separate zone with its own name servers. Such a decision might be made when a department such as English does not wish to run its own name server, but a department such as Computer Science does.

Each zone is also associated with one or more name servers. These are hosts that hold the database for the zone. Normally, a zone will have one primary name server, which gets its information from a file on its disk, and one or more secondary name servers, which get their information from the primary name server. To improve reliability, some of the name servers can be located outside the zone.

The process of looking up a name and finding an address is called **name resolution**. When a resolver has a query about a domain name, it passes the query to a local name server. If the domain being sought falls under the jurisdiction of the name server, such as *top.cs.vu.nl* falling under *cs.vu.nl*, it returns the authoritative resource records. An **authoritative record** is one that comes from the authority that manages the record and is thus always correct. Authoritative records are in contrast to **cached records**, which may be out of date.

What happens when the domain is remote, such as when *flits.cs.vu.nl* wants to find the IP address of *robot.cs.washington.edu* at UW (University of Washington)? In this case, and if there is no cached information about the domain available locally, the name server begins a remote query. This query follows the process shown in Fig. 7-6. Step 1 shows the query that is sent to the local name server. The query contains the domain name sought, the type (*A*), and the class(*IN*).

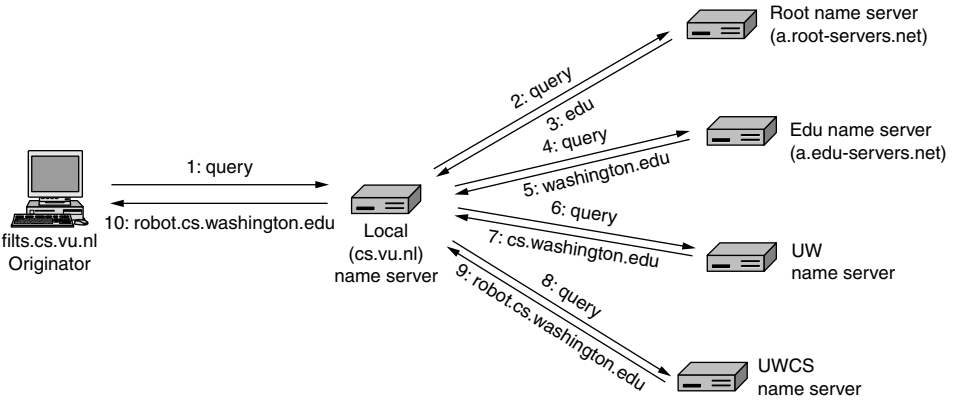


Figure 7-6. Example of a resolver looking up a remote name in 10 steps.

The next step is to start at the top of the name hierarchy by asking one of the **root name servers**. These name servers have information about each top-level

domain. This is shown as step 2 in Fig. 7-6. To contact a root server, each name server must have information about one or more root name servers. This information is normally present in a system configuration file that is loaded into the DNS cache when the DNS server is started. It is simply a list of *NS* records for the root and the corresponding *A* records.

There are 13 root DNS servers, unimaginatively called *a-root-servers.net* through *m.root-servers.net*. Each root server could logically be a single computer. However, since the entire Internet depends on the root servers, they are powerful and heavily replicated computers. Most of the servers are present in multiple geographical locations and reached using anycast routing, in which a packet is delivered to the nearest instance of a destination address; we described anycast in Chap. 5 The replication improves reliability and performance.

The root name server is unlikely to know the address of a machine at UW, and probably does not know the name server for UW either. But it must know the name server for the *edu* domain, in which *cs.washington.edu* is located. It returns the name and IP address for that part of the answer in step 3.

The local name server then continues its quest. It sends the entire query to the *edu* name server (*a.edu-servers.net*). That name server returns the name server for UW. This is shown in steps 4 and 5. Closer now, the local name server sends the query to the UW name server (step 6). If the domain name being sought was in the English department, the answer would be found, as the UW zone includes the English department. But the Computer Science department has chosen to run its own name server. The query returns the name and IP address of the UW Computer Science name server (step 7).

Finally, the local name server queries the UW Computer Science name server (step 8). This server is authoritative for the domain *cs.washington.edu*, so it must have the answer. It returns the final answer (step 9), which the local name server forwards as a response to *flits.cs.vu.nl* (step 10). The name has been resolved.

You can explore this process using standard tools such as the *dig* program that is installed on most UNIX systems. For example, typing

```
dig@a.edu-servers.net robot.cs.washington.edu
```

will send a query for *robot.cs.washington.edu* to the *a.edu-servers.net* name server and print out the result. This will show you the information obtained in step 4 in the example above, and you will learn the name and IP address of the UW name servers.

There are three technical points to discuss about this long scenario. First, two different query mechanisms are at work in Fig. 7-6. When the host *flits.cs.vu.nl* sends its query to the local name server, that name server handles the resolution on behalf of *flits* until it has the desired answer to return. It does *not* return partial answers. They might be helpful, but they are not what the query was seeking. This mechanism is called a **recursive query**.

On the other hand, the root name server (and each subsequent name server) does not recursively continue the query for the local name server. It just returns a partial answer and moves on to the next query. The local name server is responsible for continuing the resolution by issuing further queries. This mechanism is called an **iterative query**.

One name resolution can involve both mechanisms, as this example showed. A recursive query may always seem preferable, but many name servers (especially the root) will not handle them. They are too busy. Iterative queries put the burden on the originator. The rationale for the local name server supporting a recursive query is that it is providing a service to hosts in its domain. Those hosts do not have to be configured to run a full name server, just to reach the local one.

The second point is caching. All of the answers, including all the partial answers returned, are cached. In this way, if another *cs.vu.nl* host queries for *robot.cs.washington.edu* the answer will already be known. Even better, if a host queries for a different host in the same domain, say *galah.cs.washington.edu*, the query can be sent directly to the authoritative name server. Similarly, queries for other domains in *washington.edu* can start directly from the *washington.edu* name server. Using cached answers greatly reduces the steps in a query and improves performance. The original scenario we sketched is in fact the worst case that occurs when no useful information is cached.

However, cached answers are not authoritative, since changes made at *cs.washington.edu* will not be propagated to all the caches in the world that may know about it. For this reason, cache entries should not live too long. This is the reason that the *Time_to_live* field is included in each resource record. It tells remote name servers how long to cache records. If a certain machine has had the same IP address for years, it may be safe to cache that information for 1 day. For more volatile information, it might be safer to purge the records after a few seconds or a minute.

The third issue is the transport protocol that is used for the queries and responses. It is UDP. DNS messages are sent in UDP packets with a simple format for queries, answers, and name servers that can be used to continue the resolution. We will not go into the details of this format. If no response arrives within a short time, the DNS client repeats the query, trying another server for the domain after a small number of retries. This process is designed to handle the case of the server being down as well as the query or response packet getting lost. A 16-bit identifier is included in each query and copied to the response so that a name server can match answers to the corresponding query, even if multiple queries are outstanding at the same time.

Even though its purpose is simple, it should be clear that DNS is a large and complex distributed system that is comprised of millions of name servers that work together. It forms a key link between human-readable domain names and the IP addresses of machines. It includes replication and caching for performance and reliability and is designed to be highly robust.

We have not covered security, but as you might imagine, the ability to change the name-to-address mapping can have devastating consequences if done maliciously. For that reason, security extensions called DNSSEC have been developed for DNS. We will describe them in Chap. 8.

There is also application demand to use names in more flexible ways, for example, by naming content and resolving to the IP address of a nearby host that has the content. This fits the model of searching for and downloading a movie. It is the movie that matters, not the computer that has a copy of it, so all that is wanted is the IP address of any nearby computer that has a copy of the movie. Content distribution networks are one way to accomplish this mapping. We will describe how they build on the DNS later in this chapter, in Sec. 7.5.

7.2 ELECTRONIC MAIL

Electronic mail, or more commonly **email**, has been around for over three decades. Faster and cheaper than paper mail, email has been a popular application since the early days of the Internet. Before 1990, it was mostly used in academia. During the 1990s, it became known to the public at large and grew exponentially, to the point where the number of emails sent per day now is vastly more than the number of **snail mail** (i.e., paper) letters. Other forms of network communication, such as instant messaging and voice-over-IP calls have expanded greatly in use over the past decade, but email remains the workhorse of Internet communication. It is widely used within industry for intracompany communication, for example, to allow far-flung employees all over the world to cooperate on complex projects. Unfortunately, like paper mail, the majority of email—some 9 out of 10 messages—is junk mail or **spam** (McAfee, 2010).

Email, like most other forms of communication, has developed its own conventions and styles. It is very informal and has a low threshold of use. People who would never dream of calling up or even writing a letter to a Very Important Person do not hesitate for a second to send a sloppily written email to him or her. By eliminating most cues associated with rank, age, and gender, email debates often focus on content, not status. With email, a brilliant idea from a summer student can have more impact than a dumb one from an executive vice president.

Email is full of jargon such as BTW (By The Way), ROTFL (Rolling On The Floor Laughing), and IMHO (In My Humble Opinion). Many people also use little ASCII symbols called **smileys**, starting with the ubiquitous ":-)". Rotate the book 90 degrees clockwise if this symbol is unfamiliar. This symbol and other **emoticons** help to convey the tone of the message. They have spread to other terse forms of communication, such as instant messaging.

The email protocols have evolved during the period of their use, too. The first email systems simply consisted of file transfer protocols, with the convention that the first line of each message (i.e., file) contained the recipient's address. As time

went on, email diverged from file transfer and many features were added, such as the ability to send one message to a list of recipients. Multimedia capabilities became important in the 1990s to send messages with images and other non-text material. Programs for reading email became much more sophisticated too, shifting from text-based to graphical user interfaces and adding the ability for users to access their mail from their laptops wherever they happen to be. Finally, with the prevalence of spam, mail readers and the mail transfer protocols must now pay attention to finding and removing unwanted email.

In our description of email, we will focus on the way that mail messages are moved between users, rather than the look and feel of mail reader programs. Nevertheless, after describing the overall architecture, we will begin with the user-facing part of the email system, as it is familiar to most readers.

7.2.1 Architecture and Services

In this section, we will provide an overview of how email systems are organized and what they can do. The architecture of the email system is shown in Fig. 7-7. It consists of two kinds of subsystems: the **user agents**, which allow people to read and send email, and the **message transfer agents**, which move the messages from the source to the destination. We will also refer to message transfer agents informally as **mail servers**.

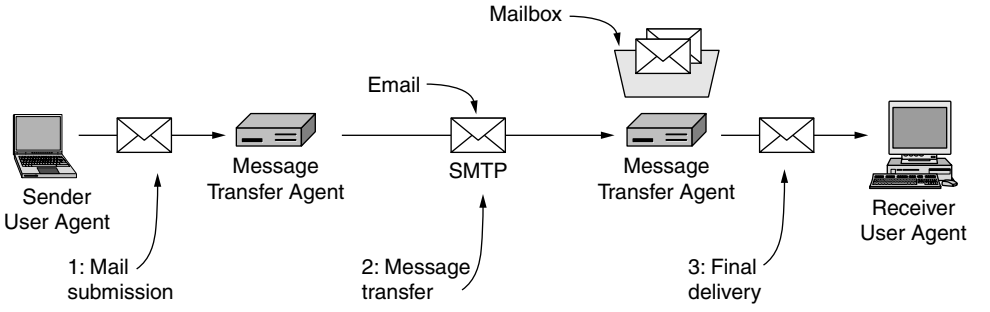


Figure 7-7. Architecture of the email system.

The user agent is a program that provides a graphical interface, or sometimes a text- and command-based interface that lets users interact with the email system. It includes a means to compose messages and replies to messages, display incoming messages, and organize messages by filing, searching, and discarding them. The act of sending new messages into the mail system for delivery is called **mail submission**.

Some of the user agent processing may be done automatically, anticipating what the user wants. For example, incoming mail may be filtered to extract or

deprioritize messages that are likely spam. Some user agents include advanced features, such as arranging for automatic email responses (“I’m having a wonderful vacation and it will be a while before I get back to you”). A user agent runs on the same computer on which a user reads her mail. It is just another program and may be run only some of the time.

The message transfer agents are typically system processes. They run in the background on mail server machines and are intended to be always available. Their job is to automatically move email through the system from the originator to the recipient with **SMTP** (**Simple Mail Transfer Protocol**). This is the message transfer step.

SMTP was originally specified as RFC 821 and revised to become the current RFC 5321. It sends mail over connections and reports back the delivery status and any errors. Numerous applications exist in which confirmation of delivery is important and may even have legal significance (“Well, Your Honor, my email system is just not very reliable, so I guess the electronic subpoena just got lost somewhere”).

Message transfer agents also implement **mailing lists**, in which an identical copy of a message is delivered to everyone on a list of email addresses. Other advanced features are carbon copies, blind carbon copies, high-priority email, secret (i.e., encrypted) email, alternative recipients if the primary one is not currently available, and the ability for assistants to read and answer their bosses’ email.

Linking user agents and message transfer agents are the concepts of mailboxes and a standard format for email messages. **Mailboxes** store the email that is received for a user. They are maintained by mail servers. User agents simply present users with a view of the contents of their mailboxes. To do this, the user agents send the mail servers commands to manipulate the mailboxes, inspecting their contents, deleting messages, and so on. The retrieval of mail is the final delivery (step 3) in Fig. 7-7. With this architecture, one user may use different user agents on multiple computers to access one mailbox.

Mail is sent between message transfer agents in a standard format. The original format, RFC 822, has been revised to the current RFC 5322 and extended with support for multimedia content and international text. This scheme is called MIME and will be discussed later. People still refer to Internet email as RFC 822, though.

A key idea in the message format is the distinction between the **envelope** and its contents. The envelope encapsulates the message. It contains all the information needed for transporting the message, such as the destination address, priority, and security level, all of which are distinct from the message itself. The message transport agents use the envelope for routing, just as the post office does.

The message inside the envelope consists of two separate parts: the **header** and the **body**. The header contains control information for the user agents. The body is entirely for the human recipient. None of the agents care much about it. Envelopes and messages are illustrated in Fig. 7-8.

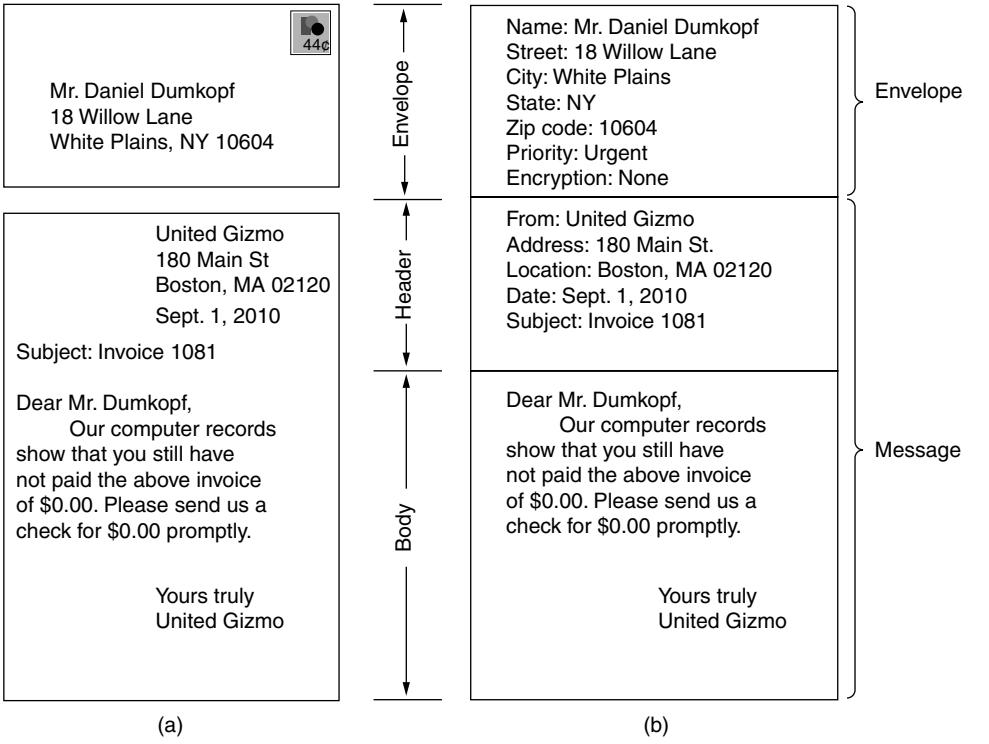


Figure 7-8. Envelopes and messages. (a) Paper mail. (b) Electronic mail.

We will examine the pieces of this architecture in more detail by looking at the steps that are involved in sending email from one user to another. This journey starts with the user agent.

7.2.2 The User Agent

A user agent is a program (sometimes called an **email reader**) that accepts a variety of commands for composing, receiving, and replying to messages, as well as for manipulating mailboxes. There are many popular user agents, including Google gmail, Microsoft Outlook, Mozilla Thunderbird, and Apple Mail. They can vary greatly in their appearance. Most user agents have a menu- or icon-driven graphical interface that requires a mouse, or a touch interface on smaller mobile devices. Older user agents, such as Elm, mh, and Pine, provide text-based interfaces and expect one-character commands from the keyboard. Functionally, these are the same, at least for text messages.

The typical elements of a user agent interface are shown in Fig. 7-9. Your mail reader is likely to be much flashier, but probably has equivalent functions.

When a user agent is started, it will usually present a summary of the messages in the user's mailbox. Often, the summary will have one line for each message in some sorted order. It highlights key fields of the message that are extracted from the message envelope or header.

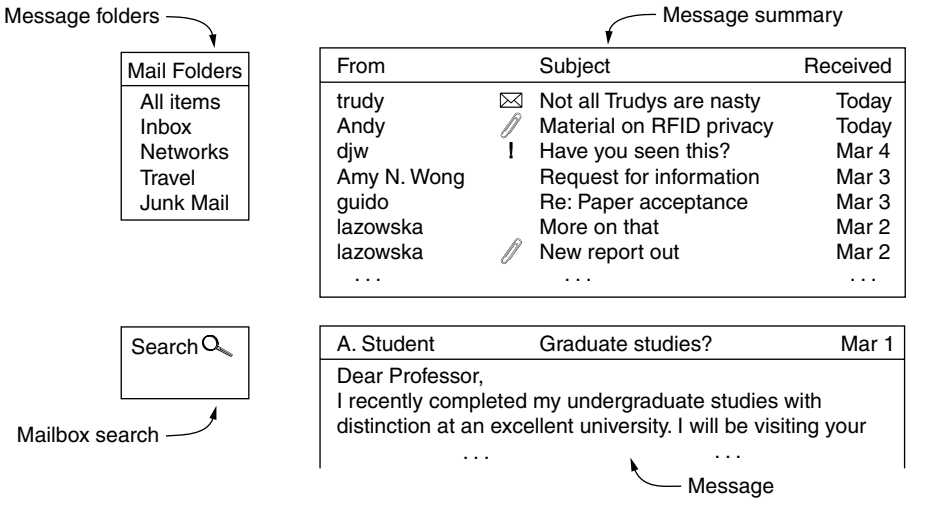


Figure 7-9. Typical elements of the user agent interface.

Seven summary lines are shown in the example of Fig. 7-9. The lines use the *From*, *Subject*, and *Received* fields, in that order, to display who sent the message, what it is about, and when it was received. All the information is formatted in a user-friendly way rather than displaying the literal contents of the message fields, but it is based on the message fields. Thus, people who fail to include a *Subject* field often discover that responses to their emails tend not to get the highest priority.

Many other fields or indications are possible. The icons next to the message subjects in Fig. 7-9 might indicate, for example, unread mail (the envelope), attached material (the paperclip), and important mail, at least as judged by the sender (the exclamation point).

Many sorting orders are also possible. The most common is to order messages based on the time that they were received, most recent first, with some indication as to whether the message is new or has already been read by the user. The fields in the summary and the sort order can be customized by the user according to her preferences.

User agents must also be able to display incoming messages as needed so that people can read their email. Often a short preview of a message is provided, as in Fig. 7-9, to help users decide when to read further. Previews may use small icons or images to describe the contents of the message. Other presentation processing

includes reformatting messages to fit the display, and translating or converting contents to more convenient formats (e.g., digitized speech to recognized text).

After a message has been read, the user can decide what to do with it. This is called **message disposition**. Options include deleting the message, sending a reply, forwarding the message to another user, and keeping the message for later reference. Most user agents can manage one mailbox for incoming mail with multiple folders for saved mail. The folders allow the user to save message according to sender, topic, or some other category.

Filing can be done automatically by the user agent as well, before the user reads the messages. A common example is that the fields and contents of messages are inspected and used, along with feedback from the user about previous messages, to determine if a message is likely to be spam. Many ISPs and companies run software that labels mail as important or spam so that the user agent can file it in the corresponding mailbox. The ISP and company have the advantage of seeing mail for many users and may have lists of known spammers. If hundreds of users have just received a similar message, it is probably spam. By presorting incoming mail as "probably legitimate" and "probably spam," the user agent can save users a fair amount of work separating the good stuff from the junk.

And the most popular spam? It is generated by collections of compromised computers called **botnets** and its content depends on where you live. Fake diplomas are topical in Asia, and cheap drugs and other dubious product offers are topical in the U.S. Unclaimed Nigerian bank accounts still abound. Pills for enlarging various body parts are common everywhere.

Other filing rules can be constructed by users. Each rule specifies a condition and an action. For example, a rule could say that any message received from the boss goes to one folder for immediate reading and any message from a particular mailing list goes to another folder for later reading. Several folders are shown in Fig. 7-9. The most important folders are the Inbox, for incoming mail not filed elsewhere, and Junk Mail, for messages that are thought to be spam.

As well as explicit constructs like folders, user agents now provide rich capabilities to search the mailbox. This feature is also shown in Fig. 7-9. Search capabilities let users find messages quickly, such as the message about "where to buy Vegemite" that someone sent in the last month.

Email has come a long way from the days when it was just file transfer. Providers now routinely support mailboxes with up to 1 GB of stored mail that details a user's interactions over a long period of time. The sophisticated mail handling of user agents with search and automatic forms of processing is what makes it possible to manage these large volumes of email. For people who send and receive thousands of messages a year, these tools are invaluable.

Another useful feature is the ability to automatically respond to messages in some way. One response is to forward incoming email to a different address, for example, a computer operated by a commercial paging service that pages the user

by using radio or satellite and displays the *Subject:* line on his pager. These **auto-responders** must run in the mail server because the user agent may not run all the time and may only occasionally retrieve email. Because of these factors, the user agent cannot provide a true automatic response. However, the interface for automatic responses is usually presented by the user agent.

A different example of an automatic response is a **vacation agent**. This is a program that examines each incoming message and sends the sender an insipid reply such as: "Hi. I'm on vacation. I'll be back on the 24th of August. Talk to you then." Such replies can also specify how to handle urgent matters in the interim, other people to contact for specific problems, etc. Most vacation agents keep track of whom they have sent canned replies to and refrain from sending the same person a second reply. There are pitfalls with these agents, however. For example, it is not advisable to send a canned reply to a large mailing list.

Let us now turn to the scenario of one user sending a message to another user. One of the basic features user agents support that we have not yet discussed is mail composition. It involves creating messages and answers to messages and sending these messages into the rest of the mail system for delivery. Although any text editor can be used to create the body of the message, editors are usually integrated with the user agent so that it can provide assistance with addressing and the numerous header fields attached to each message. For example, when answering a message, the email system can extract the originator's address from the incoming email and automatically insert it into the proper place in the reply. Other common features are appending a **signature block** to the bottom of a message, correcting spelling, and computing digital signatures that show the message is valid.

Messages that are sent into the mail system have a standard format that must be created from the information supplied to the user agent. The most important part of the message for transfer is the envelope, and the most important part of the envelope is the destination address. This address must be in a format that the message transfer agents can deal with.

The expected form of an address is *user@dns-address*. Since we studied DNS earlier in this chapter, we will not repeat that material here. However, it is worth noting that other forms of addressing exist. In particular, **X.400** addresses look radically different from DNS addresses.

X.400 is an ISO standard for message-handling systems that was at one time a competitor to SMTP. SMTP won out handily, though X.400 systems are still used, mostly outside of the U.S. X.400 addresses are composed of *attribute=value* pairs separated by slashes, for example,

```
/C=US/ST=MASSACHUSETTS/L=CAMBRIDGE/PA=360 MEMORIAL DR./CN=KEN SMITH/
```

This address specifies a country, state, locality, personal address, and common name (Ken Smith). Many other attributes are possible, so you can send email to

someone whose exact email address you do not know, provided you know enough other attributes (e.g., company and job title).

Although X.400 names are considerably less convenient than DNS names, the issue is moot for user agents because they have user-friendly aliases (sometimes called nicknames) that allow users to enter or select a person's name and get the correct email address. Consequently, it is usually not necessary to actually type in these strange strings.

A final point we will touch on for sending mail is mailing lists, which let users send the same message to a list of people with a single command. There are two choices for how the mailing list is maintained. It might be maintained locally, by the user agent. In this case, the user agent can just send a separate message to each intended recipient.

Alternatively, the list may be maintained remotely at a message transfer agent. Messages will then be expanded in the message transfer system, which has the effect of allowing multiple users to send to the list. For example, if a group of bird watchers has a mailing list called *birders* installed on the transfer agent *meadowlark.arizona.edu*, any message sent to *birders@meadowlark.arizona.edu* will be routed to the University of Arizona and expanded into individual messages to all the mailing list members, wherever in the world they may be. Users of this mailing list cannot tell that it is a mailing list. It could just as well be the personal mailbox of Prof. Gabriel O. Birders.

7.2.3 Message Formats

Now we turn from the user interface to the format of the email messages themselves. Messages sent by the user agent must be placed in a standard format to be handled by the message transfer agents. First we will look at basic ASCII email using RFC 5322, which is the latest revision of the original Internet message format as described in RFC 822. After that, we will look at multimedia extensions to the basic format.

RFC 5322—The Internet Message Format

Messages consist of a primitive envelope (described as part of SMTP in RFC 5321), some number of header fields, a blank line, and then the message body. Each header field (logically) consists of a single line of ASCII text containing the field name, a colon, and, for most fields, a value. The original RFC 822 was designed decades ago and did not clearly distinguish the envelope fields from the header fields. Although it has been revised to RFC 5322, completely redoing it was not possible due to its widespread usage. In normal usage, the user agent builds a message and passes it to the message transfer agent, which then uses some of the header fields to construct the actual envelope, a somewhat old-fashioned mixing of message and envelope.

The principal header fields related to message transport are listed in Fig. 7-10. The *To:* field gives the DNS address of the primary recipient. Having multiple recipients is also allowed. The *Cc:* field gives the addresses of any secondary recipients. In terms of delivery, there is no distinction between the primary and secondary recipients. It is entirely a psychological difference that may be important to the people involved but is not important to the mail system. The term *Cc:* (Carbon copy) is a bit dated, since computers do not use carbon paper, but it is well established. The *Bcc:* (Blind carbon copy) field is like the *Cc:* field, except that this line is deleted from all the copies sent to the primary and secondary recipients. This feature allows people to send copies to third parties without the primary and secondary recipients knowing this.

Header	Meaning
To:	Email address(es) of primary recipient(s)
Cc:	Email address(es) of secondary recipient(s)
Bcc:	Email address(es) for blind carbon copies
From:	Person or people who created the message
Sender:	Email address of the actual sender
Received:	Line added by each transfer agent along the route
Return-Path:	Can be used to identify a path back to the sender

Figure 7-10. RFC 5322 header fields related to message transport.

The next two fields, *From:* and *Sender:*, tell who wrote and sent the message, respectively. These need not be the same. For example, a business executive may write a message, but her assistant may be the one who actually transmits it. In this case, the executive would be listed in the *From:* field and the assistant in the *Sender:* field. The *From:* field is required, but the *Sender:* field may be omitted if it is the same as the *From:* field. These fields are needed in case the message is undeliverable and must be returned to the sender.

A line containing *Received:* is added by each message transfer agent along the way. The line contains the agent's identity, the date and time the message was received, and other information that can be used for debugging the routing system.

The *Return-Path:* field is added by the final message transfer agent and was intended to tell how to get back to the sender. In theory, this information can be gathered from all the *Received:* headers (except for the name of the sender's mailbox), but it is rarely filled in as such and typically just contains the sender's address.

In addition to the fields of Fig. 7-10, RFC 5322 messages may also contain a variety of header fields used by the user agents or human recipients. The most common ones are listed in Fig. 7-11. Most of these are self-explanatory, so we will not go into all of them in much detail.

Header	Meaning
Date:	The date and time the message was sent
Reply-To:	Email address to which replies should be sent
Message-Id:	Unique number for referencing this message later
In-Reply-To:	Message-Id of the message to which this is a reply
References:	Other relevant Message-Ids
Keywords:	User-chosen keywords
Subject:	Short summary of the message for the one-line display

Figure 7-11. Some fields used in the RFC 5322 message header.

The *Reply-To:* field is sometimes used when neither the person composing the message nor the person sending the message wants to see the reply. For example, a marketing manager may write an email message telling customers about a new product. The message is sent by an assistant, but the *Reply-To:* field lists the head of the sales department, who can answer questions and take orders. This field is also useful when the sender has two email accounts and wants the reply to go to the other one.

The *Message-Id:* is an automatically generated number that is used to link messages together (e.g., when used in the *In-Reply-To:* field) and to prevent duplicate delivery.

The RFC 5322 document explicitly says that users are allowed to invent optional headers for their own private use. By convention since RFC 822, these headers start with the string *X-*. It is guaranteed that no future headers will use names starting with *X-*, to avoid conflicts between official and private headers. Sometimes wiseguy undergraduates make up fields like *X-Fruit-of-the-Day:* or *X-Disease-of-the-Week:*, which are legal, although not always illuminating.

After the headers comes the message body. Users can put whatever they want here. Some people terminate their messages with elaborate signatures, including quotations from greater and lesser authorities, political statements, and disclaimers of all kinds (e.g., The XYZ Corporation is not responsible for my opinions; in fact, it cannot even comprehend them).

MIME—The Multipurpose Internet Mail Extensions

In the early days of the ARPANET, email consisted exclusively of text messages written in English and expressed in ASCII. For this environment, the early RFC 822 format did the job completely: it specified the headers but left the content entirely up to the users. In the 1990s, the worldwide use of the Internet and demand to send richer content through the mail system meant that this approach was no longer adequate. The problems included sending and receiving messages

in languages with accents (e.g., French and German), non-Latin alphabets (e.g., Hebrew and Russian), or no alphabets (e.g., Chinese and Japanese), as well as sending messages not containing text at all (e.g., audio, images, or binary documents and programs).

The solution was the development of **MIME** (**Multipurpose Internet Mail Extensions**). It is widely used for mail messages that are sent across the Internet, as well as to describe content for other applications such as Web browsing. MIME is described in RFCs 2045–2047, 4288, 4289, and 2049.

The basic idea of MIME is to continue to use the RFC 822 format (the precursor to RFC 5322 the time MIME was proposed) but to add structure to the message body and define encoding rules for the transfer of non-ASCII messages. Not deviating from RFC 822 allowed MIME messages to be sent using the existing mail transfer agents and protocols (based on RFC 821 then, and RFC 5321 now). All that had to be changed were the sending and receiving programs, which users could do for themselves.

MIME defines five new message headers, as shown in Fig. 7-12. The first of these simply tells the user agent receiving the message that it is dealing with a MIME message, and which version of MIME it uses. Any message not containing a *MIME-Version:* header is assumed to be an English plaintext message (or at least one using only ASCII characters) and is processed as such.

Header	Meaning
MIME-Version:	Identifies the MIME version
Content-Description:	Human-readable string telling what is in the message
Content-Id:	Unique identifier
Content-Transfer-Encoding:	How the body is wrapped for transmission
Content-Type:	Type and format of the content

Figure 7-12. Message headers added by MIME.

The *Content-Description:* header is an ASCII string telling what is in the message. This header is needed so the recipient will know whether it is worth decoding and reading the message. If the string says ''Photo of Barbara's hamster'' and the person getting the message is not a big hamster fan, the message will probably be discarded rather than decoded into a high-resolution color photograph.

The *Content-Id:* header identifies the content. It uses the same format as the standard *Message-Id:* header.

The *Content-Transfer-Encoding:* tells how the body is wrapped for transmission through the network. A key problem at the time MIME was developed was that the mail transfer (SMTP) protocols expected ASCII messages in which no line exceeded 1000 characters. ASCII characters use 7 bits out of each 8-bit byte. Binary data such as executable programs and images use all 8 bits of each byte, as

do extended character sets. There was no guarantee this data would be transferred safely. Hence, some method of carrying binary data that made it look like a regular ASCII mail message was needed. Extensions to SMTP since the development of MIME do allow 8-bit binary data to be transferred, though even today binary data may not always go through the mail system correctly if unencoded.

MIME provides five transfer encoding schemes, plus an escape to new schemes—just in case. The simplest scheme is just ASCII text messages. ASCII characters use 7 bits and can be carried directly by the email protocol, provided that no line exceeds 1000 characters.

The next simplest scheme is the same thing, but using 8-bit characters, that is, all values from 0 up to and including 255 are allowed. Messages using the 8-bit encoding must still adhere to the standard maximum line length.

Then there are messages that use a true binary encoding. These are arbitrary binary files that not only use all 8 bits but also do not adhere to the 1000-character line limit. Executable programs fall into this category. Nowadays, mail servers can negotiate to send data in binary (or 8-bit) encoding, falling back to ASCII if both ends do not support the extension.

The ASCII encoding of binary data is called **base64 encoding**. In this scheme, groups of 24 bits are broken up into four 6-bit units, with each unit being sent as a legal ASCII character. The coding is "A" for 0, "B" for 1, and so on, followed by the 26 lowercase letters, the 10 digits, and finally + and / for 62 and 63, respectively. The == and = sequences indicate that the last group contained only 8 or 16 bits, respectively. Carriage returns and line feeds are ignored, so they can be inserted at will in the encoded character stream to keep the lines short enough. Arbitrary binary text can be sent safely using this scheme, albeit inefficiently. This encoding was very popular before binary-capable mail servers were widely deployed. It is still commonly seen.

For messages that are almost entirely ASCII but with a few non-ASCII characters, base64 encoding is somewhat inefficient. Instead, an encoding known as **quoted-printable encoding** is used. This is just 7-bit ASCII, with all the characters above 127 encoded as an equals sign followed by the character's value as two hexadecimal digits. Control characters, some punctuation marks and math symbols, as well as trailing spaces are also so encoded.

Finally, when there are valid reasons not to use one of these schemes, it is possible to specify a user-defined encoding in the *Content-Transfer-Encoding:* header.

The last header shown in Fig. 7-12 is really the most interesting one. It specifies the nature of the message body and has had an impact well beyond email. For instance, content downloaded from the Web is labeled with MIME types so that the browser knows how to present it. So is content sent over streaming media and real-time transports such as voice over IP.

Initially, seven MIME types were defined in RFC 1521. Each type has one or more available subtypes. The type and subtype are separated by a slash, as in

"Content-Type: video/mpeg". Since then, hundreds of subtypes have been added, along with another type. Additional entries are being added all the time as new types of content are developed. The list of assigned types and subtypes is maintained online by IANA at *www.iana.org/assignments/media-types*.

The types, along with examples of commonly used subtypes, are given in Fig. 7-13. Let us briefly go through them, starting with *text*. The *text/plain* combination is for ordinary messages that can be displayed as received, with no encoding and no further processing. This option allows ordinary messages to be transported in MIME with only a few extra headers. The *text/html* subtype was added when the Web became popular (in RFC 2854) to allow Web pages to be sent in RFC 822 email. A subtype for the eXtensible Markup Language, *text/xml*, is defined in RFC 3023. XML documents have proliferated with the development of the Web. We will study HTML and XML in Sec. 7.3.

Type	Example subtypes	Description
text	plain, html, xml, css	Text in various formats
image	gif, jpeg, tiff	Pictures
audio	basic, mpeg, mp4	Sounds
video	mpeg, mp4, quicktime	Movies
model	vrml	3D model
application	octet-stream, pdf, javascript, zip	Data produced by applications
message	http, rfc822	Encapsulated message
multipart	mixed, alternative, parallel, digest	Combination of multiple types

Figure 7-13. MIME content types and example subtypes.

The next MIME type is *image*, which is used to transmit still pictures. Many formats are widely used for storing and transmitting images nowadays, both with and without compression. Several of these, including GIF, JPEG, and TIFF, are built into nearly all browsers. Many other formats and corresponding subtypes exist as well.

The *audio* and *video* types are for sound and moving pictures, respectively. Please note that *video* may include only the visual information, not the sound. If a movie with sound is to be transmitted, the video and audio portions may have to be transmitted separately, depending on the encoding system used. The first video format defined was the one devised by the modestly named Moving Picture Experts Group (MPEG), but others have been added since. In addition to *audio/basic*, a new audio type, *audio/mpeg*, was added in RFC 3003 to allow people to email MP3 audio files. The *video/mp4* and *audio/mp4* types signal video and audio data that are stored in the newer MPEG 4 format.

The *model* type was added after the other content types. It is intended for describing 3D model data. However, it has not been widely used to date.

The *application* type is a catchall for formats that are not covered by one of the other types and that require an application to interpret the data. We have listed the subtypes *pdf*, *javascript*, and *zip* as examples for PDF documents, JavaScript programs, and Zip archives, respectively. User agents that receive this content use a third-party library or external program to display the content; the display may or may not appear to be integrated with the user agent.

By using MIME types, user agents gain the extensibility to handle new types of application content as it is developed. This is a significant benefit. On the other hand, many of the new forms of content are executed or interpreted by applications, which presents some dangers. Obviously, running an arbitrary executable program that has arrived via the mail system from "friends" poses a security hazard. The program may do all sorts of nasty damage to the parts of the computer to which it has access, especially if it can read and write files and use the network. Less obviously, document formats can pose the same hazards. This is because formats such as PDF are full-blown programming languages in disguise. While they are interpreted and restricted in scope, bugs in the interpreter often allow devious documents to escape the restrictions.

Besides these examples, there are many more application subtypes because there are many more applications. As a fallback to be used when no other subtype is known to be more fitting, the *octet-stream* subtype denotes a sequence of uninterpreted bytes. Upon receiving such a stream, it is likely that a user agent will display it by suggesting to the user that it be copied to a file. Subsequent processing is then up to the user, who presumably knows what kind of content it is.

The last two types are useful for composing and manipulating messages themselves. The *message* type allows one message to be fully encapsulated inside another. This scheme is useful for forwarding email, for example. When a complete RFC 822 message is encapsulated inside an outer message, the *rfc822* subtype should be used. Similarly, it is common for HTML documents to be encapsulated. And the *partial* subtype makes it possible to break an encapsulated message into pieces and send them separately (for example, if the encapsulated message is too long). Parameters make it possible to reassemble all the parts at the destination in the correct order.

Finally, the *multipart* type allows a message to contain more than one part, with the beginning and end of each part being clearly delimited. The *mixed* subtype allows each part to be a different type, with no additional structure imposed. Many email programs allow the user to provide one or more attachments to a text message. These attachments are sent using the *multipart* type.

In contrast to *mixed*, the *alternative* subtype allows the same message to be included multiple times but expressed in two or more different media. For example, a message could be sent in plain ASCII, in HMTL, and in PDF. A properly designed user agent getting such a message would display it according to user preferences. Likely PDF would be the first choice, if that is possible. The second choice would be HTML. If neither of these were possible, then the flat ASCII

text would be displayed. The parts should be ordered from simplest to most complex to help recipients with pre-MIME user agents make some sense of the message (e.g., even a pre-MIME user can read flat ASCII text).

The *alternative* subtype can also be used for multiple languages. In this context, the Rosetta Stone can be thought of as an early *multipart/alternative* message.

Of the other two example subtypes, the *parallel* subtype is used when all parts must be "viewed" simultaneously. For example, movies often have an audio channel and a video channel. Movies are more effective if these two channels are played back in parallel, instead of consecutively. The *digest* subtype is used when multiple messages are packed together into a composite message. For example, some discussion groups on the Internet collect messages from subscribers and then send them out to the group periodically as a single *multipart/digest* message.

As an example of how MIME types may be used for email messages, a multimedia message is shown in Fig. 7-14. Here, a birthday greeting is transmitted in alternative forms as HTML and as an audio file. Assuming the receiver has audio capability, the user agent there will play the sound file. In this example, the sound is carried by reference as a *message/external-body* subtype, so first the user agent must fetch the sound file *birthday.snd* using FTP. If the user agent has no audio capability, the lyrics are displayed on the screen in stony silence. The two parts are delimited by two hyphens followed by a (software-generated) string specified in the *boundary* parameter.

Note that the *Content-Type* header occurs in three positions within this example. At the top level, it indicates that the message has multiple parts. Within each part, it gives the type and subtype of that part. Finally, within the body of the second part, it is required to tell the user agent what kind of external file it is to fetch. To indicate this slight difference in usage, we have used lowercase letters here, although all headers are case insensitive. The *Content-Transfer-Encoding* is similarly required for any external body that is not encoded as 7-bit ASCII.

7.2.4 Message Transfer

Now that we have described user agents and mail messages, we are ready to look at how the message transfer agents relay messages from the originator to the recipient. The mail transfer is done with the SMTP protocol.

The simplest way to move messages is to establish a transport connection from the source machine to the destination machine and then just transfer the message. This is how SMTP originally worked. Over the years, however, two different uses of SMTP have been differentiated. The first use is **mail submission**, step 1 in the email architecture of Fig. 7-7. This is the means by which user agents send messages into the mail system for delivery. The second use is to transfer messages between message transfer agents (step 2 in Fig. 7-7). This

```
From: alice@cs.washington.edu
To: bob@ee.uwa.edu.au
MIME-Version: 1.0
Message-Id: <0704760941.AA00747@cs.washington.edu>
Content-Type: multipart/alternative; boundary=qwertyuiopasdfghjklzxcvbnm
Subject: Earth orbits sun integral number of times

This is the preamble. The user agent ignores it. Have a nice day.

--qwertyuiopasdfghjklzxcvbnm
Content-Type: text/html

<p>Happy birthday to you<br>
Happy birthday to you<br>
Happy birthday dear <b> Bob </b><br>
Happy birthday to you</p>

--qwertyuiopasdfghjklzxcvbnm
Content-Type: message/external-body;
        access-type="anon-ftp";
        site="bicycle.cs.washington.edu";
        directory="pub";
        name="birthday.snd"

content-type: audio/basic
content-transfer-encoding: base64
--qwertyuiopasdfghjklzxcvbnm--
```

Figure 7-14. A multipart message containing HTML and audio alternatives.

sequence delivers mail all the way from the sending to the receiving message transfer agent in one hop. Final delivery is accomplished with different protocols that we will describe in the next section.

In this section, we will describe the basics of the SMTP protocol and its extension mechanism. Then we will discuss how it is used differently for mail submission and message transfer.

SMTP (Simple Mail Transfer Protocol) and Extensions

Within the Internet, email is delivered by having the sending computer establish a TCP connection to port 25 of the receiving computer. Listening to this port is a mail server that speaks **SMTP** (**Simple Mail Transfer Protocol**). This server accepts incoming connections, subject to some security checks, and accepts messages for delivery. If a message cannot be delivered, an error report containing the first part of the undeliverable message is returned to the sender.

SMTP is a simple ASCII protocol. This is not a weakness but a feature. Using ASCII text makes protocols easy to develop, test, and debug. They can be

tested by sending commands manually, and records of the messages are easy to read. Most application-level Internet protocols now work this way (e.g., HTTP).

We will walk through a simple message transfer between mail servers that delivers a message. After establishing the TCP connection to port 25, the sending machine, operating as the client, waits for the receiving machine, operating as the server, to talk first. The server starts by sending a line of text giving its identity and telling whether it is prepared to receive mail. If it is not, the client releases the connection and tries again later.

If the server is willing to accept email, the client announces whom the email is coming from and whom it is going to. If such a recipient exists at the destination, the server gives the client the go-ahead to send the message. Then the client sends the message and the server acknowledges it. No checksums are needed because TCP provides a reliable byte stream. If there is more email, that is now sent. When all the email has been exchanged in both directions, the connection is released. A sample dialog for sending the message of Fig. 7-14, including the numerical codes used by SMTP, is shown in Fig. 7-15. The lines sent by the client (i.e., the sender) are marked *C:*. Those sent by the server (i.e., the receiver) are marked *S:*.

The first command from the client is indeed meant to be *HELO*. Of the various four-character abbreviations for *HELLO*, this one has numerous advantages over its biggest competitor. Why all the commands had to be four characters has been lost in the mists of time.

In Fig. 7-15, the message is sent to only one recipient, so only one *RCPT* command is used. Such commands are allowed to send a single message to multiple receivers. Each one is individually acknowledged or rejected. Even if some recipients are rejected (because they do not exist at the destination), the message can be sent to the other ones.

Finally, although the syntax of the four-character commands from the client is rigidly specified, the syntax of the replies is less rigid. Only the numerical code really counts. Each implementation can put whatever string it wants after the code.

The basic SMTP works well, but it is limited in several respects. It does not include authentication. This means that the *FROM* command in the example could give any sender address that it pleases. This is quite useful for sending spam. Another limitation is that SMTP transfers ASCII messages, not binary data. This is why the base64 MIME content transfer encoding was needed. However, with that encoding the mail transmission uses bandwidth inefficiently, which is an issue for large messages. A third limitation is that SMTP sends messages in the clear. It has no encryption to provide a measure of privacy against prying eyes.

To allow these and many other problems related to message processing to be addressed, SMTP was revised to have an extension mechanism. This mechanism is a mandatory part of the RFC 5321 standard. The use of SMTP with extensions is called **ESMTP** (**Extended SMTP**).

```
                    S: 220 ee.uwa.edu.au SMTP service ready
C: HELO abcd.com
                    S: 250 cs.washington.edu says hello to ee.uwa.edu.au
C: MAIL FROM: <alice@cs.washington.edu>
                    S: 250 sender ok
C: RCPT TO: <bob@ee.uwa.edu.au>
                    S: 250 recipient ok
C: DATA
                    S: 354 Send mail; end with "." on a line by itself
C: From: alice@cs.washington.edu
C: To: bob@ee.uwa.edu.au
C: MIME-Version: 1.0
C: Message-Id: <0704760941.AA00747@ee.uwa.edu.au>
C: Content-Type: multipart/alternative; boundary=qwertyuiopasdfghjklzxcvbnm
C: Subject: Earth orbits sun integral number of times
C:
C: This is the preamble. The user agent ignores it. Have a nice day.
C:
C: --qwertyuiopasdfghjklzxcvbnm
C: Content-Type: text/html
C:
C: <p>Happy birthday to you
C: Happy birthday to you
C: Happy birthday dear <bold> Bob </bold>
C: Happy birthday to you
C:
C: --qwertyuiopasdfghjklzxcvbnm
C: Content-Type: message/external-body;
C:          access-type="anon-ftp";
C:          site="bicycle.cs.washington.edu";
C:          directory="pub";
C:          name="birthday.snd"
C:
C: content-type: audio/basic
C: content-transfer-encoding: base64
C: --qwertyuiopasdfghjklzxcvbnm
C: .
                    S: 250 message accepted
C: QUIT
                    S: 221 ee.uwa.edu.au closing connection
```

Figure 7-15. Sending a message from *alice@cs.washington.edu* to *bob@ee.uwa.edu.au*.

Clients wanting to use an extension send an *EHLO* message instead of *HELO* initially. If this is rejected, the server is a regular SMTP server, and the client should proceed in the usual way. If the *EHLO* is accepted, the server replies with the extensions that it supports. The client may then use any of these extensions. Several common extensions are shown in Fig. 7-16. The figure gives the keyword

as used in the extension mechanism, along with a description of the new functionality. We will not go into extensions in further detail.

Keyword	Description
AUTH	Client authentication
BINARYMIME	Server accepts binary messages
CHUNKING	Server accepts large messages in chunks
SIZE	Check message size before trying to send
STARTTLS	Switch to secure transport (TLS; see Chap. 8)
UTF8SMTP	Internationalized addresses

Figure 7-16. Some SMTP extensions.

To get a better feel for how SMTP and some of the other protocols described in this chapter work, try them out. In all cases, first go to a machine connected to the Internet. On a UNIX (or Linux) system, in a shell, type

```
telnet mail.isp.com 25
```

substituting the DNS name of your ISP's mail server for *mail.isp.com*. On a Windows XP system, click on Start, then Run, and type the command in the dialog box. On a Vista or Windows 7 machine, you may have to first install the telnet program (or equivalent) and then start it yourself. This command will establish a telnet (i.e., TCP) connection to port 25 on that machine. Port 25 is the SMTP port; see Fig. 6-34 for the ports for other common protocols. You will probably get a response something like this:

```
Trying 192.30.200.66...
Connected to mail.isp.com
Escape character is '^]'.
220 mail.isp.com Smail #74 ready at Thu, 25 Sept 2002 13:26 +0200
```

The first three lines are from telnet, telling you what it is doing. The last line is from the SMTP server on the remote machine, announcing its willingness to talk to you and accept email. To find out what commands it accepts, type

```
HELP
```

From this point on, a command sequence such as the one in Fig. 7-16 is possible if the server is willing to accept mail from you.

Mail Submission

Originally, user agents ran on the same computer as the sending message transfer agent. In this setting, all that is required to send a message is for the user agent to talk to the local mail server, using the dialog that we have just described. However, this setting is no longer the usual case.

User agents often run on laptops, home PCs, and mobile phones. They are not always connected to the Internet. Mail transfer agents run on ISP and company servers. They are always connected to the Internet. This difference means that a user agent in Boston may need to contact its regular mail server in Seattle to send a mail message because the user is traveling.

By itself, this remote communication poses no problem. It is exactly what the TCP/IP protocols are designed to support. However, an ISP or company usually does not want any remote user to be able to submit messages to its mail server to be delivered elsewhere. The ISP or company is not running the server as a public service. In addition, this kind of **open mail relay** attracts spammers. This is because it provides a way to launder the original sender and thus make the message more difficult to identify as spam.

Given these considerations, SMTP is normally used for mail submission with the *AUTH* extension. This extension lets the server check the credentials (username and password) of the client to confirm that the server should be providing mail service.

There are several other differences in the way SMTP is used for mail submission. For example, port 587 is used in preference to port 25 and the SMTP server can check and correct the format of the messages sent by the user agent. For more information about the restricted use of SMTP for mail submission, please see RFC 4409.

Message Transfer

Once the sending mail transfer agent receives a message from the user agent, it will deliver it to the receiving mail transfer agent using SMTP. To do this, the sender uses the destination address. Consider the message in Fig. 7-15, addressed to *bob@ee.uwa.edu.au*. To what mail server should the message be delivered?

To determine the correct mail server to contact, DNS is consulted. In the previous section, we described how DNS contains multiple types of records, including the *MX*, or mail exchanger, record. In this case, a DNS query is made for the *MX* records of the domain *ee.uwa.edu.au*. This query returns an ordered list of the names and IP addresses of one or more mail servers.

The sending mail transfer agent then makes a TCP connection on port 25 to the IP address of the mail server to reach the receiving mail transfer agent, and uses SMTP to relay the message. The receiving mail transfer agent will then place mail for the user *bob* in the correct mailbox for Bob to read it at a later time. This local delivery step may involve moving the message among computers if there is a large mail infrastructure.

With this delivery process, mail travels from the initial to the final mail transfer agent in a single hop. There are no intermediate servers in the message transfer stage. It is possible, however, for this delivery process to occur multiple times. One example that we have described already is when a message transfer agent

implements a mailing list. In this case, a message is received for the list. It is then expanded as a message to each member of the list that is sent to the individual member addresses.

As another example of relaying, Bob may have graduated from M.I.T. and also be reachable via the address *bob@alum.mit.edu*. Rather than reading mail on multiple accounts, Bob can arrange for mail sent to this address to be forwarded to *bob@ee.uwa.edu*. In this case, mail sent to *bob@alum.mit.edu* will undergo two deliveries. First, it will be sent to the mail server for *alum.mit.edu*. Then, it will be sent to the mail server for *ee.uwa.edu.au*. Each of these legs is a complete and separate delivery as far as the mail transfer agents are concerned.

Another consideration nowadays is spam. Nine out of ten messages sent today are spam (McAfee, 2010). Few people want more spam, but it is hard to avoid because it masquerades as regular mail. Before accepting a message, additional checks may be made to reduce the opportunities for spam. The message for Bob was sent from *alice@cs.washington.edu*. The receiving mail transfer agent can look up the sending mail transfer agent in DNS. This lets it check that the IP address of the other end of the TCP connection matches the DNS name. More generally, the receiving agent may look up the sending domain in DNS to see if it has a mail sending policy. This information is often given in the *TXT* and *SPF* records. It may indicate that other checks can be made. For example, mail sent from *cs.washington.edu* may always be sent from the host *june.cs.washington.edu*. If the sending mail transfer agent is not *june*, there is a problem.

If any of these checks fail, the mail is probably being forged with a fake sending address. In this case, it is discarded. However, passing these checks does not imply that mail is not spam. The checks merely ensure that the mail seems to be coming from the region of the network that it purports to come from. The idea is that spammers should be forced to use the correct sending address when they send mail. This makes spam easier to recognize and delete when it is unwanted.

7.2.5 Final Delivery

Our mail message is almost delivered. It has arrived at Bob's mailbox. All that remains is to transfer a copy of the message to Bob's user agent for display. This is step 3 in the architecture of Fig. 7-7. This task was straightforward in the early Internet, when the user agent and mail transfer agent ran on the same machine as different processes. The mail transfer agent simply wrote new messages to the end of the mailbox file, and the user agent simply checked the mailbox file for new mail.

Nowadays, the user agent on a PC, laptop, or mobile, is likely to be on a different machine than the ISP or company mail server. Users want to be able to access their mail remotely, from wherever they are. They want to access email from work, from their home PCs, from their laptops when on business trips, and from cybercafes when on so-called vacation. They also want to be able to work offline,

then reconnect to receive incoming mail and send outgoing mail. Moreover, each user may run several user agents depending on what computer it is convenient to use at the moment. Several user agents may even be running at the same time.

In this setting, the job of the user agent is to present a view of the contents of the mailbox, and to allow the mailbox to be remotely manipulated. Several different protocols can be used for this purpose, but SMTP is not one of them. SMTP is a push-based protocol. It takes a message and connects to a remote server to transfer the message. Final delivery cannot be achieved in this manner both because the mailbox must continue to be stored on the mail transfer agent and because the user agent may not be connected to the Internet at the moment that SMTP attempts to relay messages.

IMAP—The Internet Message Access Protocol

One of the main protocols that is used for final delivery is **IMAP** (**Internet Message Access Protocol**). Version 4 of the protocol is defined in RFC 3501. To use IMAP, the mail server runs an IMAP server that listens to port 143. The user agent runs an IMAP client. The client connects to the server and begins to issue commands from those listed in Fig. 7-17.

First, the client will start a secure transport if one is to be used (in order to keep the messages and commands confidential), and then log in or otherwise authenticate itself to the server. Once logged in, there are many commands to list folders and messages, fetch messages or even parts of messages, mark messages with flags for later deletion, and organize messages into folders. To avoid confusion, please note that we use the term "folder" here to be consistent with the rest of the material in this section, in which a user has a single mailbox made up of multiple folders. However, in the IMAP specification, the term *mailbox* is used instead. One user thus has many IMAP mailboxes, each of which is typically presented to the user as a folder.

IMAP has many other features, too. It has the ability to address mail not by message number, but by using attributes (e.g., give me the first message from Alice). Searches can be performed on the server to find the messages that satisfy certain criteria so that only those messages are fetched by the client.

IMAP is an improvement over an earlier final delivery protocol, **POP3** (**Post Office Protocol, version 3**), which is specified in RFC 1939. POP3 is a simpler protocol but supports fewer features and is less secure in typical usage. Mail is usually downloaded to the user agent computer, instead of remaining on the mail server. This makes life easier on the server, but harder on the user. It is not easy to read mail on multiple computers, plus if the user agent computer breaks, all email may be lost permanently. Nonetheless, you will still find POP3 in use.

Proprietary protocols can also be used because the protocol runs between a mail server and user agent that can be supplied by the same company. Microsoft Exchange is a mail system with a proprietary protocol.

Command	Description
CAPABILITY	List server capabilities
STARTTLS	Start secure transport (TLS; see Chap. 8)
LOGIN	Log on to server
AUTHENTICATE	Log on with other method
SELECT	Select a folder
EXAMINE	Select a read-only folder
CREATE	Create a folder
DELETE	Delete a folder
RENAME	Rename a folder
SUBSCRIBE	Add folder to active set
UNSUBSCRIBE	Remove folder from active set
LIST	List the available folders
LSUB	List the active folders
STATUS	Get the status of a folder
APPEND	Add a message to a folder
CHECK	Get a checkpoint of a folder
FETCH	Get messages from a folder
SEARCH	Find messages in a folder
STORE	Alter message flags
COPY	Make a copy of a message in a folder
EXPUNGE	Remove messages flagged for deletion
UID	Issue commands using unique identifiers
NOOP	Do nothing
CLOSE	Remove flagged messages and close folder
LOGOUT	Log out and close connection

Figure 7-17. IMAP (version 4) commands.

Webmail

An increasingly popular alternative to IMAP and SMTP for providing email service is to use the Web as an interface for sending and receiving mail. Widely used **Webmail** systems include Google Gmail, Microsoft Hotmail and Yahoo! Mail. Webmail is one example of software (in this case, a mail user agent) that is provided as a service using the Web.

In this architecture, the provider runs mail servers as usual to accept messages for users with SMTP on port 25. However, the user agent is different. Instead of

being a standalone program, it is a user interface that is provided via Web pages. This means that users can use any browser they like to access their mail and send new messages.

We have not yet studied the Web, but a brief description that you might come back to is as follows. When the user goes to the email Web page of the provider, a form is presented in which the user is asked for a login name and password. The login name and password are sent to the server, which then validates them. If the login is successful, the server finds the user's mailbox and builds a Web page listing the contents of the mailbox on the fly. The Web page is then sent to the browser for display.

Many of the items on the page showing the mailbox are clickable, so messages can be read, deleted, and so on. To make the interface responsive, the Web pages will often include JavaScript programs. These programs are run locally on the client in response to local events (e.g., mouse clicks) and can also download and upload messages in the background, to prepare the next message for display or a new message for submission. In this model, mail submission happens using the normal Web protocols by posting data to a URL. The Web server takes care of injecting messages into the traditional mail delivery system that we have described. For security, the standard Web protocols can be used as well. These protocols concern themselves with encrypting Web pages, not whether the content of the Web page is a mail message.

7.3 THE WORLD WIDE WEB

The Web, as the World Wide Web is popularly known, is an architectural framework for accessing linked content spread out over millions of machines all over the Internet. In 10 years it went from being a way to coordinate the design of high-energy physics experiments in Switzerland to the application that millions of people think of as being "The Internet." Its enormous popularity stems from the fact that it is easy for beginners to use and provides access with a rich graphical interface to an enormous wealth of information on almost every conceivable subject, from aardvarks to Zulus.

The Web began in 1989 at CERN, the European Center for Nuclear Research. The initial idea was to help large teams, often with members in half a dozen or more countries and time zones, collaborate using a constantly changing collection of reports, blueprints, drawings, photos, and other documents produced by experiments in particle physics. The proposal for a web of linked documents came from CERN physicist Tim Berners-Lee. The first (text-based) prototype was operational 18 months later. A public demonstration given at the Hypertext '91 conference caught the attention of other researchers, which led Marc Andreessen at the University of Illinois to develop the first graphical browser. It was called Mosaic and released in February 1993.

The rest, as they say, is now history. Mosaic was so popular that a year later Andreessen left to form a company, Netscape Communications Corp., whose goal was to develop Web software. For the next three years, Netscape Navigator and Microsoft's Internet Explorer engaged in a "browser war," each one trying to capture a larger share of the new market by frantically adding more features (and thus more bugs) than the other one.

Through the 1990s and 2000s, Web sites and Web pages, as Web content is called, grew exponentially until there were millions of sites and billions of pages. A small number of these sites became tremendously popular. Those sites and the companies behind them largely define the Web as people experience it today. Examples include: a bookstore (Amazon, started in 1994, market capitalization $50 billion), a flea market (eBay, 1995, $30B), search (Google, 1998, $150B), and social networking (Facebook, 2004, private company valued at more than $15B). The period through 2000, when many Web companies became worth hundreds of millions of dollars overnight, only to go bust practically the next day when they turned out to be hype, even has a name. It is called the **dot com era**. New ideas are still striking it rich on the Web. Many of them come from students. For example, Mark Zuckerberg was a Harvard student when he started Facebook, and Sergey Brin and Larry Page were students at Stanford when they started Google. Perhaps you will come up with the next big thing.

In 1994, CERN and M.I.T. signed an agreement setting up the **W3C** (**World Wide Web Consortium**), an organization devoted to further developing the Web, standardizing protocols, and encouraging interoperability between sites. Berners-Lee became the director. Since then, several hundred universities and companies have joined the consortium. Although there are now more books about the Web than you can shake a stick at, the best place to get up-to-date information about the Web is (naturally) on the Web itself. The consortium's home page is at *www.w3.org*. Interested readers are referred there for links to pages covering all of the consortium's numerous documents and activities.

7.3.1 Architectural Overview

From the users' point of view, the Web consists of a vast, worldwide collection of content in the form of **Web pages**, often just called **pages** for short. Each page may contain links to other pages anywhere in the world. Users can follow a link by clicking on it, which then takes them to the page pointed to. This process can be repeated indefinitely. The idea of having one page point to another, now called **hypertext**, was invented by a visionary M.I.T. professor of electrical engineering, Vannevar Bush, in 1945 (Bush, 1945). This was long before the Internet was invented. In fact, it was before commercial computers existed although several universities had produced crude prototypes that filled large rooms and had less power than a modern pocket calculator.

Pages are generally viewed with a program called a **browser**. Firefox, Internet Explorer, and Chrome are examples of popular browsers. The browser fetches the page requested, interprets the content, and displays the page, properly formatted, on the screen. The content itself may be a mix of text, images, and formatting commands, in the manner of a traditional document, or other forms of content such as video or programs that produce a graphical interface with which users can interact.

A picture of a page is shown on the top-left side of Fig. 7-18. It is the page for the Computer Science & Engineering department at the University of Washington. This page shows text and graphical elements (that are mostly too small to read). Some parts of the page are associated with links to other pages. A piece of text, icon, image, and so on associated with another page is called a **hyperlink**. To follow a link, the user places the mouse cursor on the linked portion of the page area (which causes the cursor to change shape) and clicks. Following a link is simply a way of telling the browser to fetch another page. In the early days of the Web, links were highlighted with underlining and colored text so that they would stand out. Nowadays, the creators of Web pages have ways to control the look of linked regions, so a link might appear as an icon or change its appearance when the mouse passes over it. It is up to the creators of the page to make the links visually distinct, to provide a usable interface.

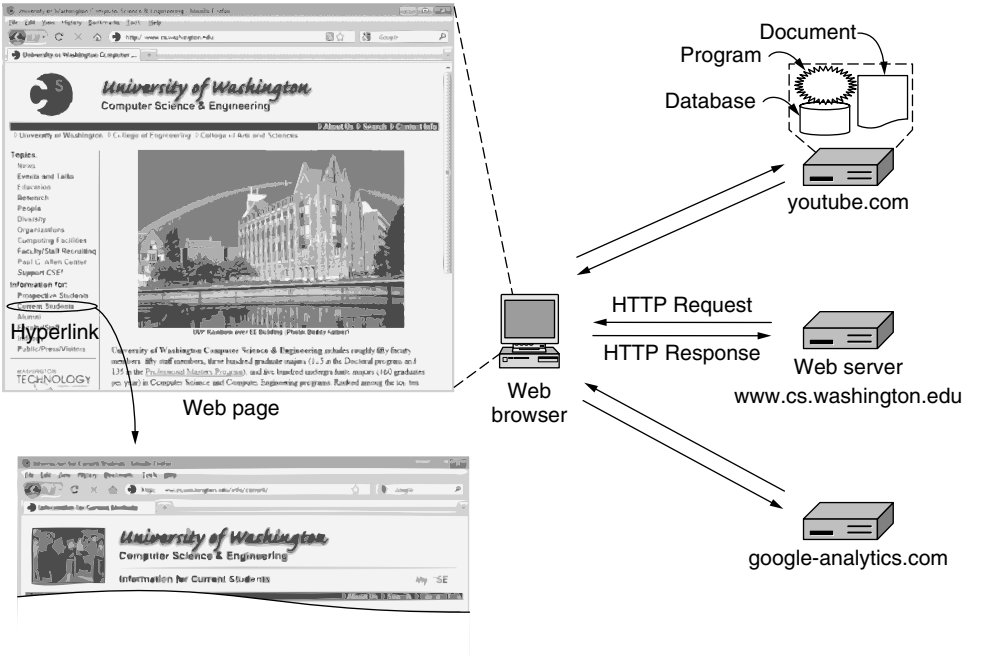


Figure 7-18. Architecture of the Web.

Students in the department can learn more by following a link to a page with information especially for them. This link is accessed by clicking in the circled area. The browser then fetches the new page and displays it, as partially shown in the bottom left of Fig. 7-18. Dozens of other pages are linked off the first page besides this example. Every other page can be comprised of content on the same machine(s) as the first page, or on machines halfway around the globe. The user cannot tell. Page fetching is done by the browser, without any help from the user. Thus, moving between machines while viewing content is seamless.

The basic model behind the display of pages is also shown in Fig. 7-18. The browser is displaying a Web page on the client machine. Each page is fetched by sending a request to one or more servers, which respond with the contents of the page. The request-response protocol for fetching pages is a simple text-based protocol that runs over TCP, just as was the case for SMTP. It is called **HTTP** (**HyperText Transfer Protocol**). The content may simply be a document that is read off a disk, or the result of a database query and program execution. The page is a **static page** if it is a document that is the same every time it is displayed. In contrast, if it was generated on demand by a program or contains a program it is a **dynamic page**.

A dynamic page may present itself differently each time it is displayed. For example, the front page for an electronic store may be different for each visitor. If a bookstore customer has bought mystery novels in the past, upon visiting the store's main page, the customer is likely to see new thrillers prominently displayed, whereas a more culinary-minded customer might be greeted with new cookbooks. How the Web site keeps track of who likes what is a story to be told shortly. But briefly, the answer involves cookies (even for culinarily challenged visitors).

In the figure, the browser contacts three servers to fetch the two pages, *cs.washington.edu*, *youtube.com*, and *google-analytics.com*. The content from these different servers is integrated for display by the browser. Display entails a range of processing that depends on the kind of content. Besides rendering text and graphics, it may involve playing a video or running a script that presents its own user interface as part of the page. In this case, the *cs.washington.edu* server supplies the main page, the *youtube.com* server supplies an embedded video, and the *google-analytics.com* server supplies nothing that the user can see but tracks visitors to the site. We will have more to say about trackers later.

The Client Side

Let us now examine the Web browser side in Fig. 7-18 in more detail. In essence, a browser is a program that can display a Web page and catch mouse clicks to items on the displayed page. When an item is selected, the browser follows the hyperlink and fetches the page selected.

When the Web was first created, it was immediately apparent that having one page point to another Web page required mechanisms for naming and locating pages. In particular, three questions had to be answered before a selected page could be displayed:

1. What is the page called?
2. Where is the page located?
3. How can the page be accessed?

If every page were somehow assigned a unique name, there would not be any ambiguity in identifying pages. Nevertheless, the problem would not be solved. Consider a parallel between people and pages. In the United States, almost everyone has a social security number, which is a unique identifier, as no two people are supposed to have the same one. Nevertheless, if you are armed only with a social security number, there is no way to find the owner's address, and certainly no way to tell whether you should write to the person in English, Spanish, or Chinese. The Web has basically the same problems.

The solution chosen identifies pages in a way that solves all three problems at once. Each page is assigned a **URL** (**Uniform Resource Locator**) that effectively serves as the page's worldwide name. URLs have three parts: the protocol (also known as the **scheme**), the DNS name of the machine on which the page is located, and the path uniquely indicating the specific page (a file to read or program to run on the machine). In the general case, the path has a hierarchical name that models a file directory structure. However, the interpretation of the path is up to the server; it may or may not reflect the actual directory structure.

As an example, the URL of the page shown in Fig. 7-18 is

```
http://www.cs.washington.edu/index.html
```

This URL consists of three parts: the protocol (*http*), the DNS name of the host (*www.cs.washington.edu*), and the path name (*index.html*).

When a user clicks on a hyperlink, the browser carries out a series of steps in order to fetch the page pointed to. Let us trace the steps that occur when our example link is selected:

1. The browser determines the URL (by seeing what was selected).
2. The browser asks DNS for the IP address of the server *www.cs.washington.edu*.
3. DNS replies with 128.208.3.88.
4. The browser makes a TCP connection to 128.208.3.88 on port 80, the well-known port for the HTTP protocol.
5. It sends over an HTTP request asking for the page */index.html*.

6. The *www.cs.washington.edu* server sends the page as an HTTP response, for example, by sending the file */index.html*.
7. If the page includes URLs that are needed for display, the browser fetches the other URLs using the same process. In this case, the URLs include multiple embedded images also fetched from *www.cs.washington.edu*, an embedded video from *youtube.com*, and a script from *google-analytics.com*.
8. The browser displays the page */index.html* as it appears in Fig. 7-18.
9. The TCP connections are released if there are no other requests to the same servers for a short period.

Many browsers display which step they are currently executing in a status line at the bottom of the screen. In this way, when the performance is poor, the user can see if it is due to DNS not responding, a server not responding, or simply page transmission over a slow or congested network.

The URL design is open-ended in the sense that it is straightforward to have browsers use multiple protocols to get at different kinds of resources. In fact, URLs for various other protocols have been defined. Slightly simplified forms of the common ones are listed in Fig. 7-19.

Name	Used for	Example
http	Hypertext (HTML)	http://www.ee.uwa.edu/~rob/
https	Hypertext with security	https://www.bank.com/accounts/
ftp	FTP	ftp://ftp.cs.vu.nl/pub/minix/README
file	Local file	file:///usr/suzanne/prog.c
mailto	Sending email	mailto:JohnUser@acm.org
rtsp	Streaming media	rtsp://youtube.com/montypython.mpg
sip	Multimedia calls	sip:eve@adversary.com
about	Browser information	about:plugins

Figure 7-19. Some common URL schemes.

Let us briefly go over the list. The *http* protocol is the Web's native language, the one spoken by Web servers. **HTTP** stands for **HyperText Transfer Protocol**. We will examine it in more detail later in this section.

The *ftp* protocol is used to access files by FTP, the Internet's file transfer protocol. FTP predates the Web and has been in use for more than three decades. The Web makes it easy to obtain files placed on numerous FTP servers throughout the world by providing a simple, clickable interface instead of a command-line interface. This improved access to information is one reason for the spectacular growth of the Web.

It is possible to access a local file as a Web page by using the *file* protocol, or more simply, by just naming it. This approach does not require having a server. Of course, it works only for local files, not remote ones.

The *mailto* protocol does not really have the flavor of fetching Web pages, but is useful anyway. It allows users to send email from a Web browser. Most browsers will respond when a *mailto* link is followed by starting the user's mail agent to compose a message with the address field already filled in.

The *rtsp* and *sip* protocols are for establishing streaming media sessions and audio and video calls.

Finally, the *about* protocol is a convention that provides information about the browser. For example, following the *about:plugins* link will cause most browsers to show a page that lists the MIME types that they handle with browser extensions called plug-ins.

In short, the URLs have been designed not only to allow users to navigate the Web, but to run older protocols such as FTP and email as well as newer protocols for audio and video, and to provide convenient access to local files and browser information. This approach makes all the specialized user interface programs for those other services unnecessary and integrates nearly all Internet access into a single program: the Web browser. If it were not for the fact that this idea was thought of by a British physicist working a research lab in Switzerland, it could easily pass for a plan dreamed up by some software company's advertising department.

Despite all these nice properties, the growing use of the Web has turned up an inherent weakness in the URL scheme. A URL points to one specific host, but sometimes it is useful to reference a page without simultaneously telling where it is. For example, for pages that are heavily referenced, it is desirable to have multiple copies far apart, to reduce the network traffic. There is no way to say: "I want page *xyz*, but I do not care where you get it."

To solve this kind of problem, URLs have been generalized into **URIs** (**Uniform Resource Identifiers**). Some URIs tell how to locate a resource. These are the URLs. Other URIs tell the name of a resource but not where to find it. These URIs are called **URN**s (**Uniform Resource Names**). The rules for writing URIs are given in RFC 3986, while the different URI schemes in use are tracked by IANA. There are many different kinds of URIs besides the schemes listed in Fig. 7-19, but those schemes dominate the Web as it is used today.

MIME Types

To be able to display the new page (or any page), the browser has to understand its format. To allow all browsers to understand all Web pages, Web pages are written in a standardized language called HTML. It is the lingua franca of the Web (for now). We will discuss it in detail later in this chapter.

Although a browser is basically an HTML interpreter, most browsers have numerous buttons and features to make it easier to navigate the Web. Most have a button for going back to the previous page, a button for going forward to the next page (only operative after the user has gone back from it), and a button for going straight to the user's preferred start page. Most browsers have a button or menu item to set a bookmark on a given page and another one to display the list of bookmarks, making it possible to revisit any of them with only a few mouse clicks.

As our example shows, HTML pages can contain rich content elements and not simply text and hypertext. For added generality, not all pages need contain HTML. A page may consist of a video in MPEG format, a document in PDF format, a photograph in JPEG format, a song in MP3 format, or any one of hundreds of other file types. Since standard HTML pages may link to any of these, the browser has a problem when it hits a page it does not know how to interpret.

Rather than making the browsers larger and larger by building in interpreters for a rapidly growing collection of file types, most browsers have chosen a more general solution. When a server returns a page, it also returns some additional information about the page. This information includes the MIME type of the page (see Fig. 7-13). Pages of type *text/html* are just displayed directly, as are pages in a few other built-in types. If the MIME type is not one of the built-in ones, the browser consults its table of MIME types to determine how to display the page. This table associates MIME types with viewers.

There are two possibilities: plug-ins and helper applications. A **plug-in** is a third-party code module that is installed as an extension to the browser, as illustrated in Fig. 7-20(a). Common examples are plug-ins for PDF, Flash, and Quicktime to render documents and play audio and video. Because plug-ins run inside the browser, they have access to the current page and can modify its appearance.

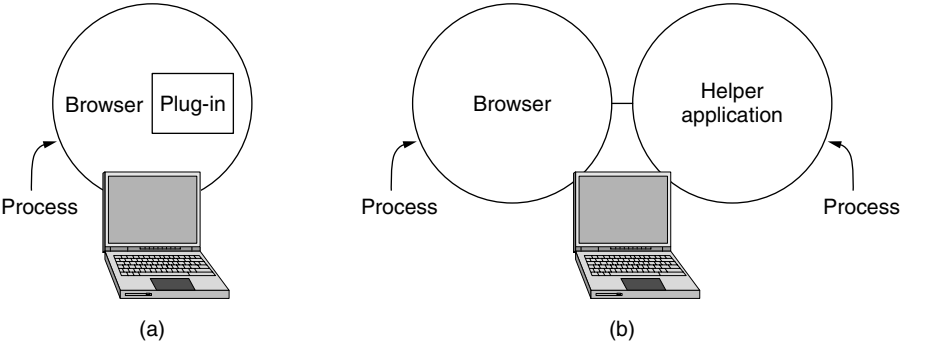


Figure 7-20. (a) A browser plug-in. (b) A helper application.

Each browser has a set of procedures that all plug-ins must implement so the browser can call the plug-ins. For example, there is typically a procedure the

browser's base code calls to supply the plug-in with data to display. This set of procedures is the plug-in's interface and is browser specific.

In addition, the browser makes a set of its own procedures available to the plug-in, to provide services to plug-ins. Typical procedures in the browser interface are for allocating and freeing memory, displaying a message on the browser's status line, and querying the browser about parameters.

Before a plug-in can be used, it must be installed. The usual installation procedure is for the user to go to the plug-in's Web site and download an installation file. Executing the installation file unpacks the plug-in and makes the appropriate calls to register the plug-in's MIME type with the browser and associate the plug-in with it. Browsers usually come preloaded with popular plug-ins.

The other way to extend a browser is make use of a **helper application**. This is a complete program, running as a separate process. It is illustrated in Fig. 7-20(b). Since the helper is a separate program, the interface is at arm's length from the browser. It usually just accepts the name of a scratch file where the content file has been stored, opens the file, and displays the contents. Typically, helpers are large programs that exist independently of the browser, for example, Microsoft Word or PowerPoint.

Many helper applications use the MIME type *application*. As a consequence, a considerable number of subtypes have been defined for them to use, for example, *application/vnd.ms-powerpoint* for PowerPoint files. *vnd* denotes vendor-specific formats. In this way, a URL can point directly to a PowerPoint file, and when the user clicks on it, PowerPoint is automatically started and handed the content to be displayed. Helper applications are not restricted to using the *application* MIME type.. Adobe Photoshop uses *image/x-photoshop*, for example.

Consequently, browsers can be configured to handle a virtually unlimited number of document types with no changes to themselves. Modern Web servers are often configured with hundreds of type/subtype combinations and new ones are often added every time a new program is installed.

A source of conflicts is that multiple plug-ins and helper applications are available for some subtypes, such as *video/mpeg*. What happens is that the last one to register overwrites the existing association with the MIME type, capturing the type for itself. As a consequence, installing a new program may change the way a browser handles existing types.

Browsers can also open local files, with no network in sight, rather than fetching them from remote Web servers. However, the browser needs some way to determine the MIME type of the file. The standard method is for the operating system to associate a file extension with a MIME type. In a typical configuration, opening *foo.pdf* will open it in the browser using an *application/pdf* plug-in and opening *bar.doc* will open it in Word as the *application/msword* helper.

Here, too, conflicts can arise, since many programs are willing—no, make that eager—to handle, say, mpg. During installation, programs intended for sophisticated users often display checkboxes for the MIME types and extensions

they are prepared to handle to allow the user to select the appropriate ones and thus not overwrite existing associations by accident. Programs aimed at the consumer market assume that the user does not have a clue what a MIME type is and simply grab everything they can without regard to what previously installed programs have done.

The ability to extend the browser with a large number of new types is convenient but can also lead to trouble. When a browser on a Windows PC fetches a file with the extension *exe*, it realizes that this file is an executable program and therefore has no helper. The obvious action is to run the program. However, this could be an enormous security hole. All a malicious Web site has to do is produce a Web page with pictures of, say, movie stars or sports heroes, all of which are linked to a virus. A single click on a picture then causes an unknown and potentially hostile executable program to be fetched and run on the user's machine. To prevent unwanted guests like this, Firefox and other browsers come configured to be cautious about running unknown programs automatically, but not all users understand what choices are safe rather than convenient.

The Server Side

So much for the client side. Now let us take a look at the server side. As we saw above, when the user types in a URL or clicks on a line of hypertext, the browser parses the URL and interprets the part between *http://* and the next slash as a DNS name to look up. Armed with the IP address of the server, the browser establishes a TCP connection to port 80 on that server. Then it sends over a command containing the rest of the URL, which is the path to the page on that server. The server then returns the page for the browser to display.

To a first approximation, a simple Web server is similar to the server of Fig. 6-6. That server is given the name of a file to look up and return via the network. In both cases, the steps that the server performs in its main loop are:

1. Accept a TCP connection from a client (a browser).
2. Get the path to the page, which is the name of the file requested.
3. Get the file (from disk).
4. Send the contents of the file to the client.
5. Release the TCP connection.

Modern Web servers have more features, but in essence, this is what a Web server does for the simple case of content that is contained in a file. For dynamic content, the third step may be replaced by the execution of a program (determined from the path) that returns the contents.

However, Web servers are implemented with a different design to serve many requests per second. One problem with the simple design is that accessing files is

often the bottleneck. Disk reads are very slow compared to program execution, and the same files may be read repeatedly from disk using operating system calls. Another problem is that only one request is processed at a time. The file may be large, and other requests will be blocked while it is transferred.

One obvious improvement (used by all Web servers) is to maintain a cache in memory of the n most recently read files or a certain number of gigabytes of content. Before going to disk to get a file, the server checks the cache. If the file is there, it can be served directly from memory, thus eliminating the disk access. Although effective caching requires a large amount of main memory and some extra processing time to check the cache and manage its contents, the savings in time are nearly always worth the overhead and expense.

To tackle the problem of serving a single request at a time, one strategy is to make the server **multithreaded**. In one design, the server consists of a front-end module that accepts all incoming requests and k processing modules, as shown in Fig. 7-21. The $k + 1$ threads all belong to the same process, so the processing modules all have access to the cache within the process' address space. When a request comes in, the front end accepts it and builds a short record describing it. It then hands the record to one of the processing modules.

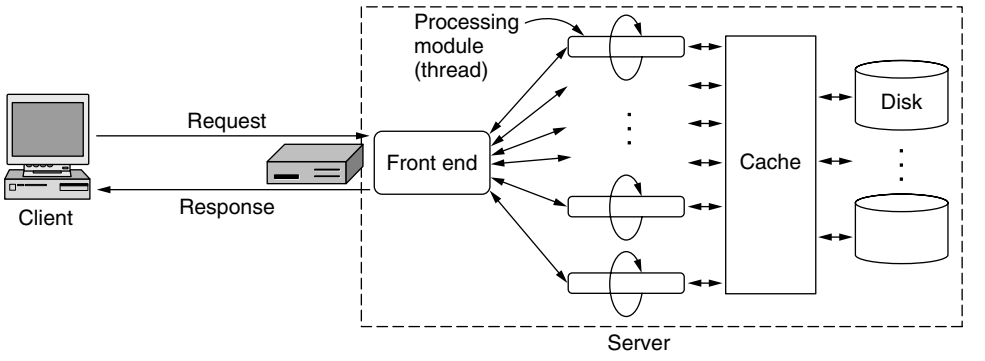


Figure 7-21. A multithreaded Web server with a front end and processing modules.

The processing module first checks the cache to see if the file needed is there. If so, it updates the record to include a pointer to the file in the record. If it is not there, the processing module starts a disk operation to read it into the cache (possibly discarding some other cached file(s) to make room for it). When the file comes in from the disk, it is put in the cache and also sent back to the client.

The advantage of this scheme is that while one or more processing modules are blocked waiting for a disk or network operation to complete (and thus consuming no CPU time), other modules can be actively working on other requests. With k processing modules, the throughput can be as much as k times higher than with a single-threaded server. Of course, when the disk or network is the limiting

factor, it is necessary to have multiple disks or a faster network to get any real improvement over the single-threaded model.

Modern Web servers do more than just accept path names and return files. In fact, the actual processing of each request can get quite complicated. For this reason, in many servers each processing module performs a series of steps. The front end passes each incoming request to the first available module, which then carries it out using some subset of the following steps, depending on which ones are needed for that particular request. These steps occur after the TCP connection and any secure transport mechanism (such as SSL/TLS, which will be described in Chap. 8) have been established.

1. Resolve the name of the Web page requested.
2. Perform access control on the Web page.
3. Check the cache.
4. Fetch the requested page from disk or run a program to build it.
5. Determine the rest of the response (e.g., the MIME type to send).
6. Return the response to the client.
7. Make an entry in the server log.

Step 1 is needed because the incoming request may not contain the actual name of a file or program as a literal string. It may contain built-in shortcuts that need to be translated. As a simple example, the URL *http://www.cs.vu.nl/* has an empty file name. It has to be expanded to some default file name that is usually *index.html*. Another common rule is to map *~user/* onto *user*'s Web directory. These rules can be used together. Thus, the home page of one of the authors (AST) can be reached at

http://www.cs.vu.nl/~ast/

even though the actual file name is *index.html* in a certain default directory.

Also, modern browsers can specify configuration information such as the browser software and the user's default language (e.g., Italian or English). This makes it possible for the server to select a Web page with small pictures for a mobile device and in the preferred language, if available. In general, name expansion is not quite so trivial as it might at first appear, due to a variety of conventions about how to map paths to the file directory and programs.

Step 2 checks to see if any access restrictions associated with the page are met. Not all pages are available to the general public. Determining whether a client can fetch a page may depend on the identity of the client (e.g., as given by usernames and passwords) or the location of the client in the DNS or IP space. For example, a page may be restricted to users inside a company. How this is

accomplished depends on the design of the server. For the popular Apache server, for instance, the convention is to place a file called *.htaccess* that lists the access restrictions in the directory where the restricted page is located.

Steps 3 and 4 involve getting the page. Whether it can be taken from the cache depends on processing rules. For example, pages that are created by running programs cannot always be cached because they might produce a different result each time they are run. Even files should occasionally be checked to see if their contents have changed so that the old contents can be removed from the cache. If the page requires a program to be run, there is also the issue of setting the program parameters or input. These data come from the path or other parts of the request.

Step 5 is about determining other parts of the response that accompany the contents of the page. The MIME type is one example. It may come from the file extension, the first few words of the file or program output, a configuration file, and possibly other sources.

Step 6 is returning the page across the network. To increase performance, a single TCP connection may be used by a client and server for multiple page fetches. This reuse means that some logic is needed to map a request to a shared connection and to return each response so that it is associated with the correct request.

Step 7 makes an entry in the system log for administrative purposes, along with keeping any other important statistics. Such logs can later be mined for valuable information about user behavior, for example, the order in which people access the pages.

Cookies

Navigating the Web as we have described it so far involves a series of independent page fetches. There is no concept of a login session. The browser sends a request to a server and gets back a file. Then the server forgets that it has ever seen that particular client.

This model is perfectly adequate for retrieving publicly available documents, and it worked well when the Web was first created. However, it is not suited for returning different pages to different users depending on what they have already done with the server. This behavior is needed for many ongoing interactions with Web sites. For example, some Web sites (e.g., newspapers) require clients to register (and possibly pay money) to use them. This raises the question of how servers can distinguish between requests from users who have previously registered and everyone else. A second example is from e-commerce. If a user wanders around an electronic store, tossing items into her virtual shopping cart from time to time, how does the server keep track of the contents of the cart? A third example is customized Web portals such as Yahoo!. Users can set up a personalized

detailed initial page with only the information they want (e.g., their stocks and their favorite sports teams), but how can the server display the correct page if it does not know who the user is?

At first glance, one might think that servers could track users by observing their IP addresses. However, this idea does not work. Many users share computers, especially at home, and the IP address merely identifies the computer, not the user. Even worse, many companies use NAT, so that outgoing packets bear the same IP address for all users. That is, all of the computers behind the NAT box look the same to the server. And many ISPs assign IP addresses to customers with DHCP. The IP addresses change over time, so to a server you might suddenly look like your neighbor. For all of these reasons, the server cannot use IP addresses to track users.

This problem is solved with an oft-critized mechanism called **cookies**. The name derives from ancient programmer slang in which a program calls a procedure and gets something back that it may need to present later to get some work done. In this sense, a UNIX file descriptor or a Windows object handle can be considered to be a cookie. Cookies were first implemented in the Netscape browser in 1994 and are now specified in RFC 2109.

When a client requests a Web page, the server can supply additional information in the form of a cookie along with the requested page. The cookie is a rather small, named string (of at most 4 KB) that the server can associate with a browser. This association is not the same thing as a user, but it is much closer and more useful than an IP address. Browsers store the offered cookies for an interval, usually in a cookie directory on the client's disk so that the cookies persist across browser invocations, unless the user has disabled cookies. Cookies are just strings, not executable programs. In principle, a cookie could contain a virus, but since cookies are treated as data, there is no official way for the virus to actually run and do damage. However, it is always possible for some hacker to exploit a browser bug to cause activation.

A cookie may contain up to five fields, as shown in Fig. 7-22. The *Domain* tells where the cookie came from. Browsers are supposed to check that servers are not lying about their domain. Each domain should store no more than 20 cookies per client. The *Path* is a path in the server's directory structure that identifies which parts of the server's file tree may use the cookie. It is often /, which means the whole tree.

The *Content* field takes the form *name* = *value*. Both *name* and *value* can be anything the server wants. This field is where the cookie's content is stored.

The *Expires* field specifies when the cookie expires. If this field is absent, the browser discards the cookie when it exits. Such a cookie is called a **nonpersistent cookie**. If a time and date are supplied, the cookie is said to be a **persistent cookie** and is kept until it expires. Expiration times are given in Greenwich Mean Time. To remove a cookie from a client's hard disk, a server just sends it again, but with an expiration time in the past.

Domain	Path	Content	Expires	Secure
toms-casino.com	/	CustomerID=297793521	15-10-10 17:00	Yes
jills-store.com	/	Cart=1-00501;1-07031;2-13721	11-1-11 14:22	No
aportal.com	/	Prefs=Stk:CSCO+ORCL;Spt:Jets	31-12-20 23:59	No
sneaky.com	/	UserID=4627239101	31-12-19 23:59	No

Figure 7-22. Some examples of cookies.

Finally, the *Secure* field can be set to indicate that the browser may only return the cookie to a server using a secure transport, namely SSL/TLS (which we will describe in Chap. 8). This feature is used for e-commerce, banking, and other secure applications.

We have now seen how cookies are acquired, but how are they used? Just before a browser sends a request for a page to some Web site, it checks its cookie directory to see if any cookies there were placed by the domain the request is going to. If so, all the cookies placed by that domain, and only that domain, are included in the request message. When the server gets them, it can interpret them any way it wants to.

Let us examine some possible uses for cookies. In Fig. 7-22, the first cookie was set by *toms-casino.com* and is used to identify the customer. When the client returns next week to throw away some more money, the browser sends over the cookie so the server knows who it is. Armed with the customer ID, the server can look up the customer's record in a database and use this information to build an appropriate Web page to display. Depending on the customer's known gambling habits, this page might consist of a poker hand, a listing of today's horse races, or a slot machine.

The second cookie came from *jills-store.com*. The scenario here is that the client is wandering around the store, looking for good things to buy. When she finds a bargain and clicks on it, the server adds it to her shopping cart (maintained on the server) and also builds a cookie containing the product code of the item and sends the cookie back to the client. As the client continues to wander around the store by clicking on new pages, the cookie is returned to the server on every new page request. As more purchases accumulate, the server adds them to the cookie. Finally, when the client clicks on PROCEED TO CHECKOUT, the cookie, now containing the full list of purchases, is sent along with the request. In this way, the server knows exactly what the customer wants to buy.

The third cookie is for a Web portal. When the customer clicks on a link to the portal, the browser sends over the cookie. This tells the portal to build a page containing the stock prices for Cisco and Oracle, and the New York Jets' football results. Since a cookie can be up to 4 KB, there is plenty of room for more detailed preferences concerning newspaper headlines, local weather, special offers, etc.

A more controversial use of cookies is to track the online behavior of users. This lets Web site operators understand how users navigate their sites, and advertisers build up profiles of the ads or sites a particular user has viewed. The controversy is that users are typically unaware that their activity is being tracked, even with detailed profiles and across seemingly unrelated Web sites. Nonetheless, **Web tracking** is big business. DoubleClick, which provides and tracks ads, is ranked among the 100 busiest Web sites in the world by the Web monitoring company Alexa. Google Analytics, which tracks site usage for operators, is used by more than half of the busiest 100,000 sites on the Web.

It is easy for a server to track user activity with cookies. Suppose a server wants to keep track of how many unique visitors it has had and how many pages each visitor looked at before leaving the site. When the first request comes in, there will be no accompanying cookie, so the server sends back a cookie containing *Counter = 1*. Subsequent page views on that site will send the cookie back to the server. Each time the counter is incremented and sent back to the client. By keeping track of the counters, the server can see how many people give up after seeing the first page, how many look at two pages, and so on.

Tracking the browsing behavior of users across sites is only slightly more complicated. It works like this. An advertising agency, say, Sneaky Ads, contacts major Web sites and places ads for its clients' products on their pages, for which it pays the site owners a fee. Instead, of giving the sites the ad as a GIF file to place on each page, it gives them a URL to add to each page. Each URL it hands out contains a unique number in the path, such as

http://www.sneaky.com/382674902342.gif

When a user first visits a page, *P*, containing such an ad, the browser fetches the HTML file. Then the browser inspects the HTML file and sees the link to the image file at *www.sneaky.com*, so it sends a request there for the image. A GIF file containing an ad is returned, along with a cookie containing a unique user ID, 4627239101 in Fig. 7-22. Sneaky records the fact that the user with this ID visited page *P*. This is easy to do since the path requested (*382674902342.gif*) is referenced only on page *P*. Of course, the actual ad may appear on thousands of pages, but each time with a different name. Sneaky probably collects a fraction of a penny from the product manufacturer each time it ships out the ad.

Later, when the user visits another Web page containing any of Sneaky's ads, the browser first fetches the HTML file from the server. Then it sees the link to, say, *http://www.sneaky.com/193654919923.gif* on the page and requests that file. Since it already has a cookie from the domain *sneaky.com*, the browser includes Sneaky's cookie containing the user's ID. Sneaky now knows a second page the user has visited.

In due course, Sneaky can build up a detailed profile of the user's browsing habits, even though the user has never clicked on any of the ads. Of course, it does not yet have the user's name (although it does have his IP address, which

may be enough to deduce the name from other databases). However, if the user ever supplies his name to any site cooperating with Sneaky, a complete profile along with a name will be available for sale to anyone who wants to buy it. The sale of this information may be profitable enough for Sneaky to place more ads on more Web sites and thus collect more information.

And if Sneaky wants to be supersneaky, the ad need not be a classical banner ad. An "ad" consisting of a single pixel in the background color (and thus invisible) has exactly the same effect as a banner ad: it requires the browser to go fetch the 1×1-pixel GIF image and send it all cookies originating at the pixel's domain.

Cookies have become a focal point for the debate over online privacy because of tracking behavior like the above. The most insidious part of the whole business is that many users are completely unaware of this information collection and may even think they are safe because they do not click on any of the ads. For this reason, cookies that track users across sites are considered by many to be **spyware**. Have a look at the cookies that are already stored by your browser. Most browsers will display this information along with the current privacy preferences. You might be surprised to find names, email addresses, or passwords as well as opaque identifiers. Hopefully, you will not find credit card numbers, but the potential for abuse is clear.

To maintain a semblance of privacy, some users configure their browsers to reject all cookies. However, this can cause problems because many Web sites will not work properly without cookies. Alternatively, most browsers let users block **third-party cookies**. A third-party cookie is one from a different site than the main page that is being fetched, for example, the *sneaky.com* cookie that is used when interacting with page *P* on a completely different Web site. Blocking these cookies helps to prevent tracking across Web sites. Browser extensions can also be installed to provide fine-grained control over how cookies are used (or, rather, not used). As the debate continues, many companies are developing privacy policies that limit how they will share information to prevent abuse. Of course, the policies are simply how the companies say they will handle information. For example: "We may use the information collected from you in the conduct of our business"—which might be selling the information.

7.3.2 Static Web Pages

The basis of the Web is transferring Web pages from server to client. In the simplest form, Web pages are static. That is, they are just files sitting on some server that present themselves in the same way each time they are fetched and viewed. Just because they are static does not mean that the pages are inert at the browser, however. A page containing a video can be a static Web page.

As mentioned earlier, the lingua franca of the Web, in which most pages are written, is HTML. The home pages of teachers are usually static HTML pages.

The home pages of companies are usually dynamic pages put together by a Web design company. In this section, we will take a brief look at static HTML pages as a foundation for later material. Readers already familiar with HTML can skip ahead to the next section, where we describe dynamic content and Web services.

HTML—The HyperText Markup Language

HTML (**HyperText Markup Language**) was introduced with the Web. It allows users to produce Web pages that include text, graphics, video, pointers to other Web pages, and more. HTML is a markup language, or language for describing how documents are to be formatted. The term "markup" comes from the old days when copyeditors actually marked up documents to tell the printer—in those days, a human being—which fonts to use, and so on. Markup languages thus contain explicit commands for formatting. For example, in HTML, **means start boldface mode, and** means leave boldface mode. LaTeX and TeX are other examples of markup languages that are well known to most academic authors.

The key advantage of a markup language over one with no explicit markup is that it separates content from how it should be presented. Writing a browser is then straightforward: the browser simply has to understand the markup commands and apply them to the content. Embedding all the markup commands within each HTML file and standardizing them makes it possible for any Web browser to read and reformat any Web page. That is crucial because a page may have been produced in a 1600×1200 window with 24-bit color on a high-end computer but may have to be displayed in a 640×320 window on a mobile phone.

While it is certainly possible to write documents like this with any plain text editor, and many people do, it is also possible to use word processors or special HTML editors that do most of the work (but correspondingly give the user less direct control over the details of the final result).

A simple Web page written in HTML and its presentation in a browser are given in Fig. 7-23. A Web page consists of a head and a body, each enclosed by <html> and </html> **tags** (formatting commands), although most browsers do not complain if these tags are missing. As can be seen in Fig. 7-23(a), the head is bracketed by the <head> and </head> tags and the body is bracketed by the <body> and </body> tags. The strings inside the tags are called **directives**. Most, but not all, HTML tags have this format. That is, they use <something> to mark the beginning of something and </something> to mark its end.

Tags can be in either lowercase or uppercase. Thus, <head> and <HEAD> mean the same thing, but lower case is best for compatibility. Actual layout of the HTML document is irrelevant. HTML parsers ignore extra spaces and carriage returns since they have to reformat the text to make it fit the current display area. Consequently, white space can be added at will to make HTML documents more

readable, something most of them are badly in need of. As another consequence, blank lines cannot be used to separate paragraphs, as they are simply ignored. An explicit tag is required.

Some tags have (named) parameters, called **attributes**. For example, the <img> tag in Fig. 7-23 is used for including an image inline with the text. It has two attributes, src and alt. The first attribute gives the URL for the image. The HTML standard does not specify which image formats are permitted. In practice, all browsers support GIF and JPEG files. Browsers are free to support other formats, but this extension is a two-edged sword. If a user is accustomed to a browser that supports, say, TIFF files, he may include these in his Web pages and later be surprised when other browsers just ignore all of his wonderful art.

The second attribute gives alternate text to use if the image cannot be displayed. For each tag, the HTML standard gives a list of what the permitted parameters, if any, are, and what they mean. Because each parameter is named, the order in which the parameters are given is not significant.

Technically, HTML documents are written in the ISO 8859-1 Latin-1 character set, but for users whose keyboards support only ASCII, escape sequences are present for the special characters, such as è. The list of special characters is given in the standard. All of them begin with an ampersand and end with a semicolon. For example,   produces a space, è produces è and é produces é. Since <, >, and & have special meanings, they can be expressed only with their escape sequences, <, >, and &, respectively.

The main item in the head is the title, delimited by <title> and </title>. Certain kinds of metainformation may also be present, though none are present in our example. The title itself is not displayed on the page. Some browsers use it to label the page's window.

Several headings are used in Fig. 7-23. Each heading is generated by an <h*n*> tag, where *n* is a digit in the range 1 to 6. Thus, <h1> is the most important heading; <h6> is the least important one. It is up to the browser to render these appropriately on the screen. Typically, the lower-numbered headings will be displayed in a larger and heavier font. The browser may also choose to use different colors for each level of heading. Usually, <h1> headings are large and boldface with at least one blank line above and below. In contrast, <h2> headings are in a smaller font with less space above and below.

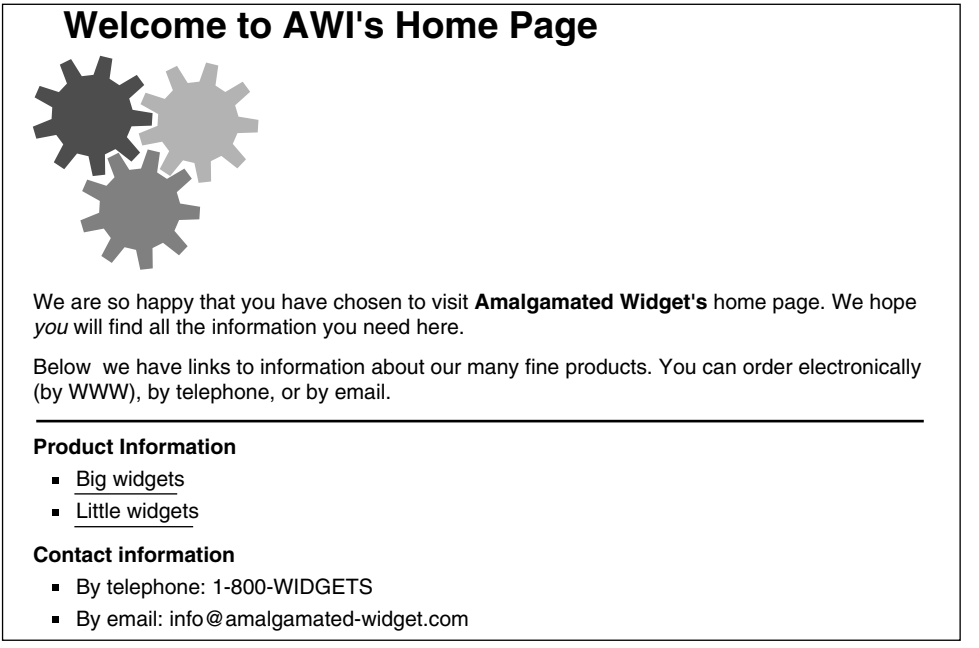
The tags <b> and <i> are used to enter boldface and italics mode, respectively. The <hr> tag forces a break and draws a horizontal line across the display.

The tag starts a paragraph. The browser might display this by inserting a blank line and some indentation, for example. Interestingly, the tag that exists to mark the end of a paragraph is often omitted by lazy HTML programmers.

HTML provides various mechanisms for making lists, including nested lists. Unordered lists, like the ones in Fig. 7-23 are started with <ul>, with <li> used to mark the start of items. There is also an <ol> tag to starts an ordered list. The

```
<html>
<head> <title> AMALGAMATED WIDGET, INC. </title> </head>
<body> <h1> Welcome to AWI's Home Page </h1>
<img src="http://www.widget.com/images/logo.gif" ALT="AWI Logo"> <br>
We are so happy that you have chosen to visit <b> Amalgamated Widget's</b>
home page. We hope <i> you </i> will find all the information you need here.
<p>Below we have links to information about our many fine products.
You can order electronically (by WWW), by telephone, or by email. </p>
<hr>
<h2> Product information </h2>
<ul>
   <li> <a href="http://widget.com/products/big"> Big widgets </a> </li>
   <li> <a href="http://widget.com/products/little"> Little widgets </a> </li>
</ul>
<h2> Contact information </h2>
<ul>
   <li> By telephone: 1-800-WIDGETS </li>
   <li> By email: info@amalgamated-widget.com </li>
</ul>
</body>
</html>
```

(a)



(b)

Figure 7-23. (a) The HTML for a sample Web page. (b) The formatted page.

individual items in unordered lists often appear with bullets (•) in front of them. Items in ordered lists are numbered by the browser.

Finally, we come to hyperlinks. Examples of these are seen in Fig. 7-23 using the <a> (anchor) and tags. The <a> tag has various parameters, the most important of which is *href* the linked URL. The text between the <a> and is displayed. If it is selected, the hyperlink is followed to a new page. It is also permitted to link other elements. For example, an image can be given between the <a> and tags using <img>. In this case, the image is displayed and clicking on it activates the hyperlink.

There are many other HTML tags and attributes that we have not seen in this simple example. For instance, the <a> tag can take a parameter *name* to plant a hyperlink, allowing a hyperlink to point to the middle of a page. This is useful, for example, for Web pages that start out with a clickable table of contents. By clicking on an item in the table of contents, the user jumps to the corresponding section of the same page. An example of a different tag is
. It forces the browser to break and start a new line.

Probably the best way to understand tags is to look at them in action. To do this, you can pick a Web page and look at the HTML in your browser to see how the page was put together. Most browsers have a VIEW SOURCE menu item (or something similar). Selecting this item displays the current page's HTML source, instead of its formatted output.

We have sketched the tags that have existed from the early Web. HTML keeps evolving. Fig. 7-24 shows some of the features that have been added with successive versions of HTML. HTML 1.0 refers to the version of HTML used with the introduction of the Web. HTML versions 2.0, 3.0, and 4.0 appeared in rapid succession in the space of only a few years as the Web exploded. After HTML 4.0, a period of almost ten years passed before the path to standarization of the next major version, HTML 5.0, became clear. Because it is a major upgrade that consolidates the ways that browsers handle rich content, the HTML 5.0 effort is ongoing and not expected to produce a standard before 2012 at the earliest. Standards notwithstanding, the major browsers already support HTML 5.0 functionality.

The progression through HTML versions is all about adding new features that people wanted but had to handle in nonstandard ways (e.g., plug-ins) until they became standard. For example, HTML 1.0 and HTML 2.0 did not have tables. They were added in HTML 3.0. An HTML table consists of one or more rows, each consisting of one or more table cells that can contain a wide range of material (e.g., text, images, other tables). Before HTML 3.0, authors needing a table had to resort to ad hoc methods, such as including an image showing the table.

In HTML 4.0, more new features were added. These included accessibility features for handicapped users, object embedding (a generalization of the <img> tag so other objects can also be embedded in pages), support for scripting languages (to allow dynamic content), and more.

Item	HTML 1.0	HTML 2.0	HTML 3.0	HTML 4.0	HTML 5.0
Hyperlinks	x	x	x	x	x
Images	x	x	x	x	x
Lists	x	x	x	x	x
Active maps & images		x	x	x	x
Forms		x	x	x	x
Equations			x	x	x
Toolbars			x	x	x
Tables			x	x	x
Accessibility features				x	x
Object embedding				x	x
Style sheets				x	x
Scripting				x	x
Video and audio					x
Inline vector graphics					x
XML representation					x
Background threads					x
Browser storage					x
Drawing canvas					x

Figure 7-24. Some differences between HTML versions.

HTML 5.0 includes many features to handle the rich media that are now routinely used on the Web. Video and audio can be included in pages and played by the browser without requiring the user to install plug-ins. Drawings can be built up in the browser as vector graphics, rather than using bitmap image formats (like JPEG and GIF) There is also more support for running scripts in browsers, such as background threads of computation and access to storage. All of these features help to support Web pages that are more like traditional applications with a user interface than documents. This is the direction the Web is heading.

Input and Forms

There is one important capability that we have not discussed yet: input. HTML 1.0 was basically one-way. Users could fetch pages from information providers, but it was difficult to send information back the other way. It quickly became apparent that there was a need for two-way traffic to allow orders for products to be placed via Web pages, registration cards to be filled out online, search terms to be entered, and much, much more.

Sending input from the user to the server (via the browser) requires two kinds of support. First, it requires that HTTP be able to carry data in that direction. We describe how this is done in a later section; it uses the *POST* method. The second requirement is to be able to present user interface elements that gather and package up the input. **Forms** were included with this functionality in HTML 2.0.

Forms contain boxes or buttons that allow users to fill in information or make choices and then send the information back to the page's owner. Forms are written just like other parts of HTML, as seen in the example of Fig. 7-25. Note that forms are still static content. They exhibit the same behavior regardless of who is using them. Dynamic content, which we will cover later, provides more sophisticated ways to gather input by sending a program whose behavior may depend on the browser environment.

Like all forms, this one is enclosed between the <form> and </form> tags. The attributes of this tag tell what to do with the data that are input, in this case using the *POST* method to send the data to the specified URL. Text not enclosed in a tag is just displayed. All the usual tags (e.g., <b>) are allowed in a form to let the author of the page control the look of the form on the screen.

Three kinds of input boxes are used in this form, each of which uses the <input> tag. It has a variety of parameters for determining the size, nature, and usage of the box displayed. The most common forms are blank fields for accepting user text, boxes that can be checked, and *submit* buttons that cause the data to be returned to the server.

The first kind of input box is a *text* box that follows the text ''Name''. The box is 46 characters wide and expects the user to type in a string, which is then stored in the variable *customer*.

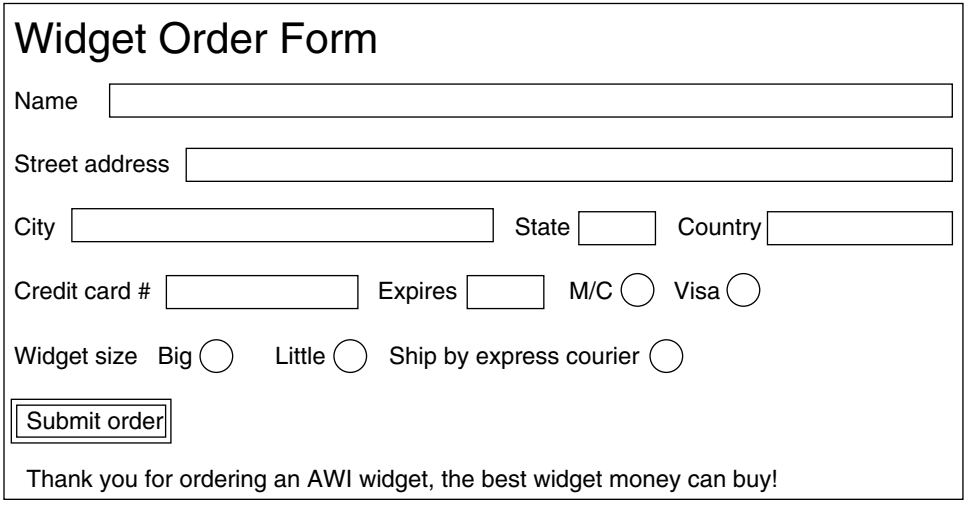
The next line of the form asks for the user's street address, 40 characters wide. Then comes a line asking for the city, state, and country. Since no <p> tags are used between these fields, the browser displays them all on one line (instead of as separate paragraphs) if they will fit. As far as the browser is concerned, the one paragraph contains just six items: three strings alternating with three boxes. The next line asks for the credit card number and expiration date. Transmitting credit card numbers over the Internet should only be done when adequate security measures have been taken. We will discuss some of these in Chap. 8.

Following the expiration date, we encounter a new feature: radio buttons. These are used when a choice must be made among two or more alternatives. The intellectual model here is a car radio with half a dozen buttons for choosing stations. Clicking on one button turns off all the other ones in the same group. The visual presentation is up to the browser. Widget size also uses two radio buttons. The two groups are distinguished by their *name* parameter, not by static scoping using something like <radiobutton> ... </radiobutton>.

The *value* parameters are used to indicate which radio button was pushed. For example, depending on which credit card options the user has chosen, the variable *cc* will be set to either the string ''mastercard'' or the string ''visacard''.

```
<html>
<head> <title> AWI CUSTOMER ORDERING FORM </title> </head>
<body>
<h1> Widget Order Form </h1>
<form ACTION="http://widget.com/cgi-bin/order.cgi" method=POST>
<p> Name <input name="customer" size=46> </p>
<p> Street address <input name="address" size=40> </p>
<p> City <input name="city" size=20> State <input name="state" size =4>
Country <input name="country" size=10> </p>
<p> Credit card # <input name="cardno" size=10>
Expires <input name="expires" size=4>
M/C <input name="cc" type=radio value="mastercard">
VISA <input name="cc" type=radio value="visacard"> </p>
<p> Widget size Big <input name="product" type=radio value="expensive">
Little <input name="product" type=radio value="cheap">
Ship by express courier <input name="express" type=checkbox> </p>
<p><input type=submit value="Submit order"> </p>
Thank you for ordering an AWI widget, the best widget money can buy!
</form>
</body>
</html>
```

(a)



(b)

Figure 7-25. (a) The HTML for an order form. (b) The formatted page.

After the two sets of radio buttons, we come to the shipping option, represented by a box of type *checkbox*. It can be either on or off. Unlike radio buttons, where exactly one out of the set must be chosen, each box of type *checkbox* can be on or off, independently of all the others.

Finally, we come to the *submit* button. The *value* string is the label on the button and is displayed. When the user clicks the *submit* button, the browser packages the collected information into a single long line and sends it back to the server to the URL provided as part of the <form> tag. A simple encoding is used. The & is used to separate fields and + is used to represent space. For our example form, the line might look like the contents of Fig. 7-26.

```
customer=John+Doe&address=100+Main+St.&city=White+Plains&
state=NY&country=USA&cardno=1234567890&expires=6/14&cc=mastercard&
product=cheap&express=on
```

Figure 7-26. A possible response from the browser to the server with information filled in by the user.

The string is sent back to the server as one line. (It is broken into three lines here because the page is not wide enough.) It is up to the server to make sense of this string, most likely by passing the information to a program that will process it. We will discuss how this can be done in the next section.

There are also other types of input that are not shown in this simple example. Two other types are *password* and *textarea*. A *password* box is the same as a *text* box (the default type that need not be named), except that the characters are not displayed as they are typed. A *textarea* box is also the same as a *text* box, except that it can contain multiple lines.

For long lists from which a choice must be made, the <select> and </select> tags are provided to bracket a list of alternatives. This list is often rendered as a drop-down menu. The semantics are those of radio buttons unless the *multiple* parameter is given, in which case the semantics are those of checkboxes.

Finally, there are ways to indicate default or initial values that the user can change. For example, if a *text* box is given a *value* field, the contents are displayed in the form for the user to edit or erase.

CSS—Cascading Style Sheets

The original goal of HTML was to specify the *structure* of the document, not its *appearance*. For example,

```
<h1> Deborah's Photos </h1>
```

instructs the browser to emphasize the heading, but does not say anything about the typeface, point size, or color. That is left up to the browser, which knows the properties of the display (e.g., how many pixels it has). However, many Web page designers wanted absolute control over how their pages appeared, so new tags were added to HTML to control appearance, such as

```
<font face="helvetica" size="24" color="red"> Deborah's Photos </font>
```

Also, ways were added to control positioning on the screen accurately. The trouble with this approach is that it is tedious and produces bloated HTML that is not portable. Although a page may render perfectly in the browser it is developed on, it may be a complete mess in another browser or another release of the same browser or at a different screen resolution.

A better alternative is the use of style sheets. Style sheets in text editors allow authors to associate text with a logical style instead of a physical style, for example, "initial paragraph" instead of "italic text." The appearance of each style is defined separately. In this way, if the author decides to change the initial paragraphs from 14-point italics in blue to 18-point boldface in shocking pink, all it requires is changing one definition to convert the entire document.

CSS (**Cascading Style Sheets**) introduced style sheets to the Web with HTML 4.0, though widespread use and browser support did not take off until 2000. CSS defines a simple language for describing rules that control the appearance of tagged content. Let us look at an example. Suppose that AWI wants snazzy Web pages with navy text in the Arial font on an off-white background, and level headings that are an extra 100% and 50% larger than the text for each level, respectively. The CSS definition in Fig. 7-27 gives these rules.

```
body {background-color:linen; color:navy; font-family:Arial;}
h1 {font-size:200%;}
h2 {font-size:150%;}
```

Figure 7-27. CSS example.

As can be seen, the style definitions can be compact. Each line selects an element to which it applies and gives the values of properties. The properties of an element apply as defaults to all other HTML elements that it contains. Thus, the style for body sets the style for paragraphs of text in the body. There are also convenient shorthands for color names (e.g., red). Any style parameters that are not defined are filled with defaults by the browser. This behavior makes style sheet definitions optional; some reasonable presentation will occur without them.

Style sheets can be placed in an HTML file (e.g., using the <style> tag), but it is more common to place them in a separate file and reference them. For example, the <head> tag of the AWI page can be modified to refer to a style sheet in the file *awistyle.css* as shown in Fig. 7-28. The example also shows the MIME type of CSS files to be *text/css*.

```
<head>
<title> AMALGAMATED WIDGET, INC. </title>
<link rel="stylesheet" type="text/css" href="awistyle.css" />
</head>
```

Figure 7-28. Including a CSS style sheet.

This strategy has two advantages. First, it lets one set of styles be applied to many pages on a Web site. This organization lends a consistent appearance to pages even if they were developed by different authors at different times, and allows the look of the entire site to be changed by editing one CSS file and not the HTML. This method can be compared to an #include file in a C program: changing one macro definition there changes it in all the program files that include the header. The second advantage is that the HTML files that are downloaded are kept small. This is because the browser can download one copy of the CSS file for all pages that reference it. It does not need to download a new copy of the definitions along with each Web page.

7.3.3 Dynamic Web Pages and Web Applications

The static page model we have used so far treats pages as multimedia documents that are conveniently linked together. It was a fitting model in the early days of the Web, as vast amounts of information were put online. Nowadays, much of the excitement around the Web is using it for applications and services. Examples include buying products on e-commerce sites, searching library catalogs, exploring maps, reading and sending email, and collaborating on documents.

These new uses are like traditional application software (e.g., mail readers and word processors). The twist is that these applications run inside the browser, with user data stored on servers in Internet data centers. They use Web protocols to access information via the Internet, and the browser to display a user interface. The advantage of this approach is that users do not need to install separate application programs, and user data can be accessed from different computers and backed up by the service operator. It is proving so successful that it is rivaling traditional application software. Of course, the fact that these applications are offered for free by large providers helps. This model is the prevalent form of **cloud computing**, in which computing moves off individual desktop computers and into shared clusters of servers in the Internet.

To act as applications, Web pages can no longer be static. Dynamic content is needed. For example, a page of the library catalog should reflect which books are currently available and which books are checked out and are thus not available. Similarly, a useful stock market page would allow the user to interact with the page to see stock prices over different periods of time and compute profits and losses. As these examples suggest, dynamic content can be generated by programs running on the server or in the browser (or in both places).

In this section, we will examine each of these two cases in turn. The general situation is as shown in Fig. 7-29. For example, consider a map service that lets the user enter a street address and presents a corresponding map of the location. Given a request for a location, the Web server must use a program to create a page that shows the map for the location from a database of streets and other geographic information. This action is shown as steps 1 through 3. The request (step

1) causes a program to run on the server. The program consults a database to generate the appropriate page (step 2) and returns it to the browser (step 3).

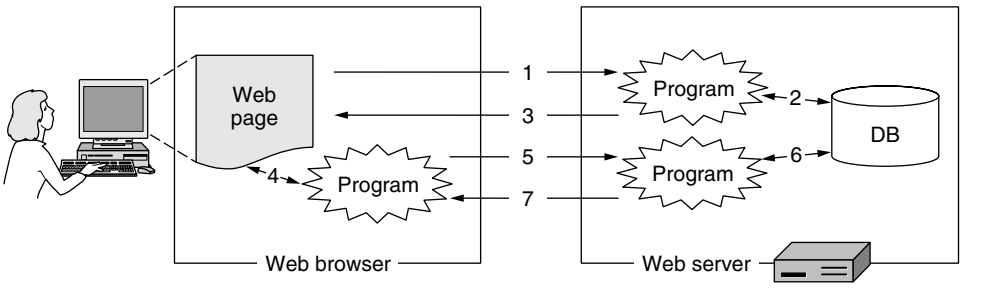


Figure 7-29. Dynamic pages.

There is more to dynamic content, however. The page that is returned may itself contain programs that run in the browser. In our map example, the program would let the user find routes and explore nearby areas at different levels of detail. It would update the page, zooming in or out as directed by the user (step 4). To handle some interactions, the program may need more data from the server. In this case, the program will send a request to the server (step 5) that will retrieve more information from the database (step 6) and return a response (step 7). The program will then continue updating the page (step 4). The requests and responses happen in the background; the user may not even be aware of them because the page URL and title typically do not change. By including client-side programs, the page can present a more responsive interface than with server-side programs alone.

Server-Side Dynamic Web Page Generation

Let us look at the case of server-side content generation in more detail. A simple situation in which server-side processing is necessary is the use of forms. Consider the user filling out the AWI order form of Fig. 7-25(b) and clicking the *Submit order* button. When the user clicks, a request is sent to the server at the URL specified with the form (a *POST* to *http://widget.com/cgi-bin/order.cgi* in this case) along with the contents of the form as filled in by the user. These data must be given to a program or script to process. Thus, the URL identifies the program to run; the data are provided to the program as input. In this case, processing would involve entering the order in AWI's internal system, updating customer records, and charging the credit card. The page returned by this request will depend on what happens during the processing. It is not fixed like a static page. If the order succeeds, the page returned might give the expected shipping date. If it is unsuccessful, the returned page might say that widgets requested are out of stock or the credit card was not valid for some reason.

Exactly how the server runs a program instead of retrieving a file depends on the design of the Web server. It is not specified by the Web protocols themselves. This is because the interface can be proprietary and the browser does not need to know the details. As far as the browser is concerned, it is simply making a request and fetching a page.

Nonetheless, standard APIs have been developed for Web servers to invoke programs. The existence of these interfaces makes it easier for developers to extend different servers with Web applications. We will briefly look at two APIs to give you a sense of what they entail.

The first API is a method for handling dynamic page requests that has been available since the beginning of the Web. It is called the **CGI** (**Common Gateway Interface**) and is defined in RFC 3875. CGI provides an interface to allow Web servers to talk to back-end programs and scripts that can accept input (e.g., from forms) and generate HTML pages in response. These programs may be written in whatever language is convenient for the developer, usually a scripting language for ease of development. Pick Python, Ruby, Perl or your favorite language.

By convention, programs invoked via CGI live in a directory called *cgi-bin*, which is visible in the URL. The server maps a request to this directory to a program name and executes that program as a separate process. It provides any data sent with the request as input to the program. The output of the program gives a Web page that is returned to the browser.

In our example, the program *order.cgi* is invoked with input from the form encoded as shown in Fig. 7-26. It will parse the parameters and process the order. A useful convention is that the program will return the HTML for the order form if no form input is provided. In this way, the program will be sure to know the representation of the form.

The second API we will look at is quite different. The approach here is to embed little scripts inside HTML pages and have them be executed by the server itself to generate the page. A popular language for writing these scripts is **PHP** (**PHP: Hypertext Preprocessor**). To use it, the server has to understand PHP, just as a browser has to understand CSS to interpret Web pages with style sheets. Usually, servers identify Web pages containing PHP from the file extension *php* rather than *html* or *htm*.

PHP is simpler to use than CGI. As an example of how it works with forms, see the example in Fig. 7-30(a). The top part of this figure contains a normal HTML page with a simple form in it. This time, the <form> tag specifies that *action.php* is to be invoked to handle the parameters when the user submits the form. The page displays two text boxes, one with a request for a name and one with a request for an age. After the two boxes have been filled in and the form submitted, the server parses the Fig. 7-26-type string sent back, putting the name in the *name* variable and the age in the *age* variable. It then starts to process the *action.php* file, shown in Fig. 7-30(b), as a reply. During the processing of this file,

the PHP commands are executed. If the user filled in "Barbara" and "24" in the boxes, the HTML file sent back will be the one given in Fig. 7-30(c). Thus, handling forms becomes extremely simple using PHP.

```
<html>
<body>
<form action="action.php" method="post">
<p> Please enter your name: <input type="text" name="name"> </p>
<p> Please enter your age: <input type="text" name="age"> </p>
<input type="submit">
</form>
</body>
</html>
```

(a)

```
<html>
<body>
<h1> Reply: </h1>
Hello <?php echo $name; ?>.
Prediction: next year you will be <?php echo $age + 1; ?>
</body>
</html>
```

(b)

```
<html>
<body>
<h1> Reply: </h1>
Hello Barbara.
Prediction: next year you will be 33
</body>
</html>
```

(c)

Figure 7-30. (a) A Web page containing a form. (b) A PHP script for handling the output of the form. (c) Output from the PHP script when the inputs are "Barbara" and "32", respectively.

Although PHP is easy to use, it is actually a powerful programming language for interfacing the Web and a server database. It has variables, strings, arrays, and most of the control structures found in C, but much more powerful I/O than just *printf*. PHP is open source code, freely available, and widely used. It was designed specifically to work well with Apache, which is also open source and is the world's most widely used Web server. For more information about PHP, see Valade (2009).

We have now seen two different ways to generate dynamic HTML pages: CGI scripts and embedded PHP. There are several others to choose from. **JSP** (**JavaServer Pages**) is similar to PHP, except that the dynamic part is written in

the Java programming language instead of in PHP. Pages using this technique have the file extension *.jsp*. **ASP.NET** (**Active Server Pages .NET**) is Microsoft's version of PHP and JavaServer Pages. It uses programs written in Microsoft's proprietary .NET networked application framework for generating the dynamic content. Pages using this technique have the extension *.aspx*. The choice among these three techniques usually has more to do with politics (open source vs. Microsoft) than with technology, since the three languages are roughly comparable.

Client-Side Dynamic Web Page Generation

PHP and CGI scripts solve the problem of handling input and interactions with databases on the server. They can all accept incoming information from forms, look up information in one or more databases, and generate HTML pages with the results. What none of them can do is respond to mouse movements or interact with users directly. For this purpose, it is necessary to have scripts embedded in HTML pages that are executed on the client machine rather than the server machine. Starting with HTML 4.0, such scripts are permitted using the tag <script>. The technologies used to produce these interactive Web pages are broadly referred to as **dynamic HTML**

The most popular scripting language for the client side is **JavaScript**, so we will now take a quick look at it. Despite the similarity in names, JavaScript has almost nothing to do with the Java programming language. Like other scripting languages, it is a very high-level language. For example, in a single line of JavaScript it is possible to pop up a dialog box, wait for text input, and store the resulting string in a variable. High-level features like this make JavaScript ideal for designing interactive Web pages. On the other hand, the fact that it is mutating faster than a fruit fly trapped in an X-ray machine makes it extremely difficult to write JavaScript programs that work on all platforms, but maybe some day it will stabilize.

As an example of a program in JavaScript, consider that of Fig. 7-31. Like that of Fig. 7-30, it displays a form asking for a name and age, and then predicts how old the person will be next year. The body is almost the same as the PHP example, the main difference being the declaration of the *Submit* button and the assignment statement in it. This assignment statement tells the browser to invoke the *response* script on a button click and pass it the form as a parameter.

What is completely new here is the declaration of the JavaScript function *response* in the head of the HTML file, an area normally reserved for titles, background colors, and so on. This function extracts the value of the *name* field from the form and stores it in the variable *person* as a string. It also extracts the value of the *age* field, converts it to an integer by using the *eval* function, adds 1 to it, and stores the result in *years*. Then it opens a document for output, does four

```
<html>
<head>
<script language="javascript" type="text/javascript">
function response(test_form) {
   var person = test_form.name.value;
   var years = eval(test_form.age.value) + 1;
   document.open();
   document.writeln("<html> <body>");
   document.writeln("Hello " + person + ".<br>");
   document.writeln("Prediction: next year you will be " + years + ".");
   document.writeln("</body> </html>");
   document.close();
}
</script>
</head>

<body>
<form>
Please enter your name: <input type="text" name="name">
<p>
Please enter your age: <input type="text" name="age">
<p>
<input type="button" value="submit" onclick="response(this.form)">
</form>
</body>
</html>
```

Figure 7-31. Use of JavaScript for processing a form.

writes to it using the *writeln* method, and closes the document. The document is an HTML file, as can be seen from the various HTML tags in it. The browser then displays the document on the screen.

It is very important to understand that while PHP and JavaScript look similar in that they both embed code in HTML files, they are processed totally differently. In the PHP example of Fig. 7-30, after the user has clicked on the *submit* button, the browser collects the information into a long string and sends it off to the server as a request for a PHP page. The server loads the PHP file and executes the PHP script that is embedded in to produce a new HTML page. That page is sent back to the browser for display. The browser cannot even be sure that it was produced by a program. This processing is shown as steps 1 to 4 in Fig. 7-32(a).

In the JavaScript example of Fig. 7-31, when the *submit* button is clicked the browser interprets a JavaScript function contained on the page. All the work is done locally, inside the browser. There is no contact with the server. This processing is shown as steps 1 and 2 in Fig. 7-32(b). As a consequence, the result is displayed virtually instantaneously, whereas with PHP there can be a delay of several seconds before the resulting HTML arrives at the client.

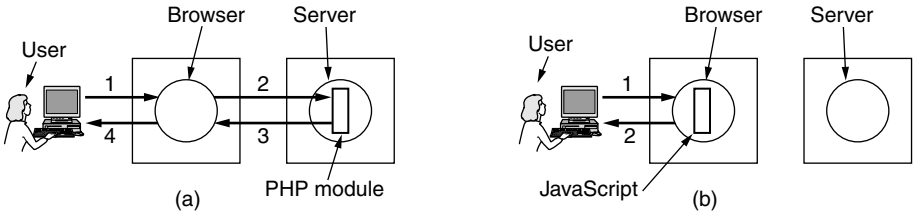


Figure 7-32. (a) Server-side scripting with PHP. (b) Client-side scripting with JavaScript.

This difference does not mean that JavaScript is better than PHP. Their uses are completely different. PHP (and, by implication, JSP and ASP) is used when interaction with a database on the server is needed. JavaScript (and other client-side languages we will mention, such as VBScript) is used when the interaction is with the user at the client computer. It is certainly possible to combine them, as we will see shortly.

JavaScript is not the only way to make Web pages highly interactive. An alternative on Windows platforms is **VBScript**, which is based on Visual Basic. Another popular method across platforms is the use of **applets**. These are small Java programs that have been compiled into machine instructions for a virtual computer called the **JVM** (**Java Virtual Machine**). Applets can be embedded in HTML pages (between <applet> and </applet>) and interpreted by JVM-capable browsers. Because Java applets are interpreted rather than directly executed, the Java interpreter can prevent them from doing Bad Things. At least in theory. In practice, applet writers have found a nearly endless stream of bugs in the Java I/O libraries to exploit.

Microsoft's answer to Sun's Java applets was allowing Web pages to hold **ActiveX controls**, which are programs compiled to x86 machine language and executed on the bare hardware. This feature makes them vastly faster and more flexible than interpreted Java applets because they can do anything a program can do. When Internet Explorer sees an ActiveX control in a Web page, it downloads it, verifies its identity, and executes it. However, downloading and running foreign programs raises enormous security issues, which we will discuss in Chap. 8.

Since nearly all browsers can interpret both Java programs and JavaScript, a designer who wants to make a highly interactive Web page has a choice of at least two techniques, and if portability to multiple platforms is not an issue, ActiveX in addition. As a general rule, JavaScript programs are easier to write, Java applets execute faster, and ActiveX controls run fastest of all. Also, since all browsers implement exactly the same JVM but no two browsers implement the same version of JavaScript, Java applets are more portable than JavaScript programs. For more information about JavaScript, there are many books, each with many (often with more than 1000) pages. See, for example, Flanagan (2010).

AJAX—Asynchronous JavaScript and XML

Compelling Web applications need responsive user interfaces and seamless access to data stored on remote Web servers. Scripting on the client (e.g., with JavaScript) and the server (e.g., with PHP) are basic technologies that provide pieces of the solution. These technologies are commonly used with several other key technologies in a combination called **AJAX** (**Asynchronous JAvascript and Xml**). Many full-featured Web applications, such as Google's Gmail, Maps, and Docs, are written with AJAX.

AJAX is somewhat confusing because it is not a language. It is a set of technologies that work together to enable Web applications that are every bit as responsive and powerful as traditional desktop applications. The technologies are:

1. HTML and CSS to present information as pages.
2. DOM (Document Object Model) to change parts of pages while they are viewed.
3. XML (eXtensible Markup Language) to let programs exchange application data with the server.
4. An asynchronous way for programs to send and retrieve XML data.
5. JavaScript as a language to bind all this functionality together.

As this is quite a collection, we will go through each piece to see what it contributes. We have already seen HTML and CSS. They are standards for describing content and how it should be displayed. Any program that can produce HTML and CSS can use a Web browser as a display engine.

DOM (**Document Object Model**) is a representation of an HTML page that is accessible to programs. This representation is structured as a tree that reflects the structure of the HTML elements. For instance, the DOM tree of the HTML in Fig. 7-30(a) is given in Fig. 7-33. At the root is an *html* element that represents the entire HTML block. This element is the parent of the *body* element, which is in turn parent to a *form* element. The form has two attributes that are drawn to the right-hand side, one for the form method (a *POST*) and one for the form action (the URL to request). This element has three children, reflecting the two paragraph tags and one input tag that are contained within the form. At the bottom of the tree are leaves that contain either elements or literals, such as text strings.

The significance of the DOM model is that it provides programs with a straightforward way to change parts of the page. There is no need to rewrite the entire page. Only the node that contains the change needs to be replaced. When this change is made, the browser will correspondingly update the display. For example, if an image on part of the page is changed in DOM, the browser will update that image without changing the other parts of the page. We have already seen DOM in action when the JavaScript example of Fig. 7-31 added lines to the

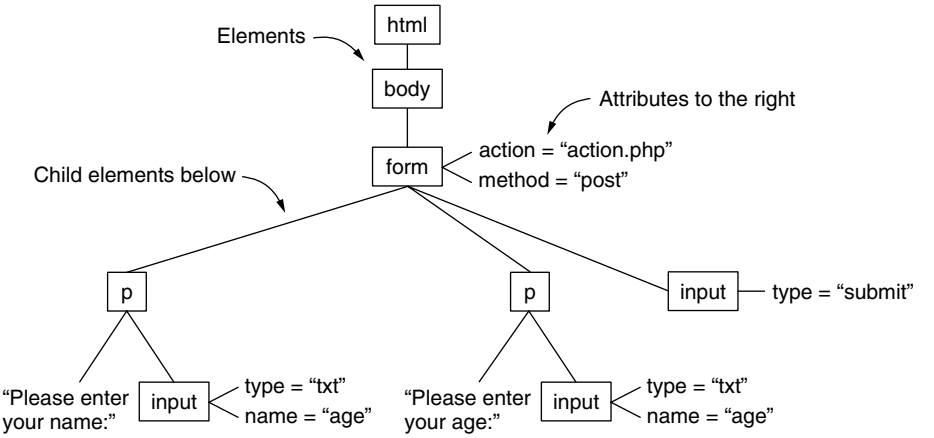


Figure 7-33. The DOM tree for the HTML in Fig. 7-30(a).

document element to cause new lines of text to appear at the bottom of the browser window. The DOM is a powerful method for producing pages that can evolve.

The third technology, **XML** (**eXtensible Markup Language**), is a language for specifying structured content. HTML mixes content with formatting because it is concerned with the presentation of information. However, as Web applications become more common, there is an increasing need to separate structured content from its presentation. For example, consider a program that searches the Web for the best price for some book. It needs to analyze many Web pages looking for the item's title and price. With Web pages in HTML, it is very difficult for a program to figure out where the title is and where the price is.

For this reason, the W3C developed XML (Bray et al., 2006) to allow Web content to be structured for automated processing. Unlike HTML, there are no defined tags for XML. Each user can define her own tags. A simple example of an XML document is given in Fig. 7-34. It defines a structure called book_list, which is a list of books. Each book has three fields, the title, author, and year of publication. These structures are extremely simple. It is permitted to have structures with repeated fields (e.g., multiple authors), optional fields (e.g., URL of the audio book), and alternative fields (e.g., URL of a bookstore if it is in print or URL of an auction site if it is out of print).

In this example, each of the three fields is an indivisible entity, but it is also permitted to further subdivide the fields. For example, the author field could have been done as follows to give finer-grained control over searching and formatting:

```
<author>
  <first_name> George </first_name>
  <last_name> Zipf </last_name>
</author>
```

Each field can be subdivided into subfields and subsubfields, arbitrarily deeply.

```
<?xml version="1.0" ?>

<book_list>

<book>
   <title> Human Behavior and the Principle of Least Effort </title>
   <author> George Zipf </author>
   <year> 1949 </year>
</book>

<book>
   <title> The Mathematical Theory of Communication </title>
   <author> Claude E. Shannon </author>
   <author> Warren Weaver </author>
   <year> 1949 </year>
</book>

<book>
   <title> Nineteen Eighty-Four </title>
   <author> George Orwell </author>
   <year> 1949 </year>
</book>

</book_list>
```

Figure 7-34. A simple XML document.

All the file of Fig. 7-34 does is define a book list containing three books. It is well suited for transporting information between programs running in browsers and servers, but it says nothing about how to display the document as a Web page. To do that, a program that consumes the information and judges 1949 to be a fine year for books might output HTML in which the titles are marked up as italic text. Alternatively, a language called **XSLT** (**eXtensible Stylesheet Language Transformations**), can be used to define how XML should be transformed into HTML. XSLT is like CSS, but much more powerful. We will spare you the details.

The other advantage of expressing data in XML, instead of HTML, is that it is easier for programs to analyze. HTML was originally written manually (and often is still) so a lot of it is a bit sloppy. Sometimes the closing tags, like </p>, are left out. Other tags do not have a matching closing tag, like
. Still other tags may be nested improperly, and the case of tag and attribute names can vary. Most browsers do their best to work out what was probably intended. XML is stricter and cleaner in its definition. Tag names and attributes are always lowercase, tags must always be closed in the reverse of the order that they were opened (or indicate clearly if they are an empty tag with no corresponding close), and attribute values must be enclosed in quotation marks. This precision makes parsing easier and unambiguous.

HTML is even being defined in terms of XML. This approach is called **XHTML** (**eXtended HyperText Markup Language**). Basically, it is a Very

Picky version of HTML. XHTML pages must strictly conform to the XML rules, otherwise they are not accepted by the browser. No more shoddy Web pages and inconsistencies across browsers. As with XML, the intent is to produce pages that are better for programs (in this case Web applications) to process. While XHTML has been around since 1998, it has been slow to catch on. People who produce HTML do not see why they need XHTML, and browser support has lagged. Now HTML 5.0 is being defined so that a page can be represented as either HTML or XHTML to aid the transition. Eventually, XHTML should replace HTML, but it will be a long time before this transition is complete.

XML has also proved popular as a language for communication between programs. When this communication is carried by the HTTP protocol (described in the next section) it is called a Web service. In particular, **SOAP** (**Simple Object Access Protocol**) is a way of implementing Web services that performs RPC between programs in a language- and system-independent way. The client just constructs the request as an XML message and sends it to the server, using the HTTP protocol. The server sends back a reply as an XML-formatted message. In this way, applications on heterogeneous platforms can communicate.

Getting back to AJAX, our point is simply that XML is a useful format to exchange data between programs running in the browser and the server. However, to provide a responsive interface in the browser while sending or receiving data, it must be possible for scripts to perform **asynchronous I/O** that does not block the display while awaiting the response to a request. For example, consider a map that can be scrolled in the browser. When it is notified of the scroll action, the script on the map page may request more map data from the server if the view of the map is near the edge of the data. The interface should not freeze while those data are fetched. Such an interface would win no user awards. Instead, the scrolling should continue smoothly. When the data arrive, the script is notified so that it can use the data. If all goes well, new map data will be fetched before it is needed. Modern browsers have support for this model of communication.

The final piece of the puzzle is a scripting language that holds AJAX together by providing access to the above list of technologies. In most cases, this language is JavaScript, but there are alternatives such as VBScript. We presented a simple example of JavaScript earlier. Do not be fooled by this simplicity. JavaScript has many quirks, but it is a full-blown programming language, with all the power of C or Java. It has variables, strings, arrays, objects, functions, and all the usual control structures. It also has interfaces specific to the browser and Web pages. JavaScript can track mouse motion over objects on the screen, which makes it easy to make a menu suddenly appear and leads to lively Web pages. It can use DOM to access pages, manipulate HTML and XML, and perform asynchronous HTTP communication.

Before leaving the subject of dynamic pages, let us briefly summarize the technologies we have covered so far by relating them on a single figure. Complete Web pages can be generated on the fly by various scripts on the server

machine. The scripts can be written in server extension languages like PHP, JSP, or ASP.NET, or run as separate CGI processes and thus be written in any language. These options are shown in Fig. 7-35.

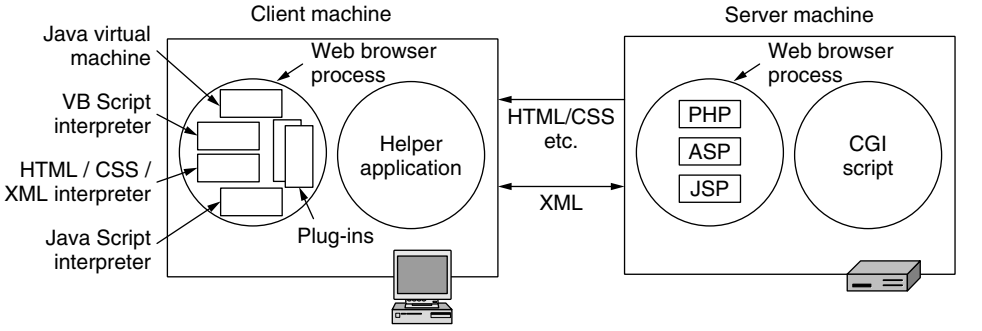


Figure 7-35. Various technologies used to generate dynamic pages.

Once these Web pages are received by the browser, they are treated as normal pages in HTML, CSS and other MIME types and just displayed. Plug-ins that run in the browser and helper applications that run outside of the browser can be installed to extend the MIME types that are supported by the browser.

Dynamic content generation is also possible on the client side. The programs that are embedded in Web pages can be written in JavaScript, VBScript, Java, and other languages. These programs can perform arbitrary computations and update the display. With AJAX, programs in Web pages can asynchronously exchange XML and other kinds of data with the server. This model supports rich Web applications that look just like traditional applications, except that they run inside the browser and access information that is stored at servers on the Internet.

7.3.4 HTTP—The HyperText Transfer Protocol

Now that we have an understanding of Web content and applications, it is time to look at the protocol that is used to transport all this information between Web servers and clients. It is **HTTP** (**HyperText Transfer Protocol**), as specified in RFC 2616.

HTTP is a simple request-response protocol that normally runs over TCP. It specifies what messages clients may send to servers and what responses they get back in return. The request and response headers are given in ASCII, just like in SMTP. The contents are given in a MIME-like format, also like in SMTP. This simple model was partly responsible for the early success of the Web because it made development and deployment straightforward.

In this section, we will look at the more important properties of HTTP as it is used nowadays. However, before getting into the details we will note that the way

it is used in the Internet is evolving. HTTP is an application layer protocol because it runs on top of TCP and is closely associated with the Web. That is why we are covering it in this chapter. However, in another sense HTTP is becoming more like a transport protocol that provides a way for processes to communicate content across the boundaries of different networks. These processes do not have to be a Web browser and Web server. A media player could use HTTP to talk to a server and request album information. Antivirus software could use HTTP to download the latest updates. Developers could use HTTP to fetch project files. Consumer electronics products like digital photo frames often use an embedded HTTP server as an interface to the outside world. Machine-to-machine communication increasingly runs over HTTP. For example, an airline server might use SOAP (an XML RPC over HTTP) to contact a car rental server and make a car reservation, all as part of a vacation package. These trends are likely to continue, along with the expanding use of HTTP.

Connections

The usual way for a browser to contact a server is to establish a TCP connection to port 80 on the server's machine, although this procedure is not formally required. The value of using TCP is that neither browsers nor servers have to worry about how to handle long messages, reliability, or congestion control. All of these matters are handled by the TCP implementation.

Early in the Web, with HTTP 1.0, after the connection was established a single request was sent over and a single response was sent back. Then the TCP connection was released. In a world in which the typical Web page consisted entirely of HTML text, this method was adequate. Quickly, the average Web page grew to contain large numbers of embedded links for content such as icons and other eye candy. Establishing a separate TCP connection to transport each single icon became a very expensive way to operate.

This observation led to HTTP 1.1, which supports **persistent connections**. With them, it is possible to establish a TCP connection, send a request and get a response, and then send additional requests and get additional responses. This strategy is also called **connection reuse**. By amortizing the TCP setup, startup, and release costs over multiple requests, the relative overhead due to TCP is reduced per request. It is also possible to pipeline requests, that is, send request 2 before the response to request 1 has arrived.

The performance difference between these three cases is shown in Fig. 7-36. Part (a) shows three requests, one after the other and each in a separate connection. Let us suppose that this represents a Web page with two embedded images on the same server. The URLs of the images are determined as the main page is fetched, so they are fetched after the main page. Nowadays, a typical page has around 40 other objects that must be fetched to present it, but that would make our figure far too big so we will use only two embedded objects.

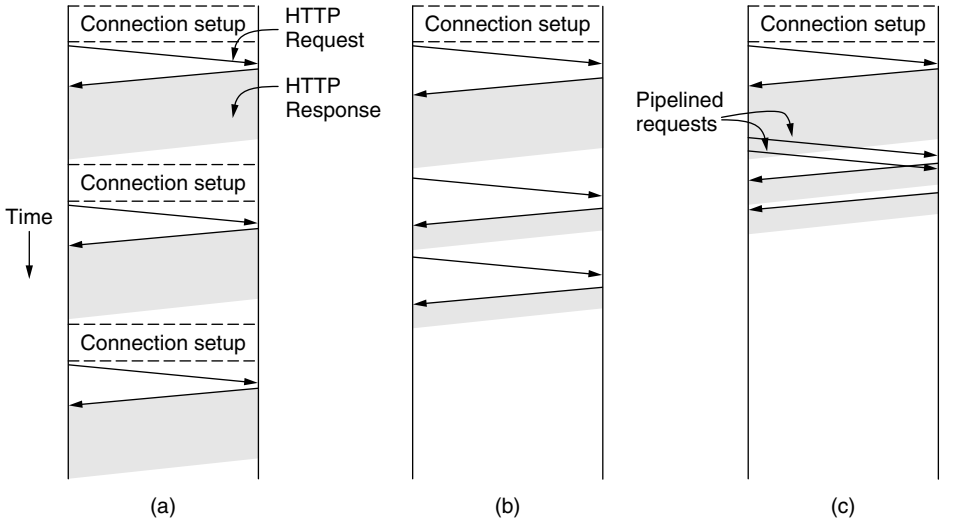


Figure 7-36. HTTP with (a) multiple connections and sequential requests. (b) A persistent connection and sequential requests. (c) A persistent connection and pipelined requests.

In Fig. 7-36(b), the page is fetched with a persistent connection. That is, the TCP connection is opened at the beginning, then the same three requests are sent, one after the other as before, and only then is the connection closed. Observe that the fetch completes more quickly. There are two reasons for the speedup. First, time is not wasted setting up additional connections. Each TCP connection requires at least one round-trip time to establish. Second, the transfer of the same images proceeds more quickly. Why is this? It is because of TCP congestion control. At the start of a connection, TCP uses the slow-start procedure to increase the throughput until it learns the behavior of the network path. The consequence of this warmup period is that multiple short TCP connections take disproportionately longer to transfer information than one longer TCP connection.

Finally, in Fig. 7-36(c), there is one persistent connection and the requests are pipelined. Specifically, the second and third requests are sent in rapid succession as soon as enough of the main page has been retrieved to identify that the images must be fetched. The responses for these requests follow eventually. This method cuts down the time that the server is idle, so it further improves performance.

Persistent connections do not come for free, however. A new issue that they raise is when to close the connection. A connection to a server should stay open while the page loads. What then? There is a good chance that the user will click on a link that requests another page from the server. If the connection remains open, the next request can be sent immediately. However, there is no guarantee that the client will make another request of the server any time soon. In practice,

clients and servers usually keep persistent connections open until they have been idle for a short time (e.g., 60 seconds) or they have a large number of open connections and need to close some.

The observant reader may have noticed that there is one combination that we have left out so far. It is also possible to send one request per TCP connection, but run multiple TCP connections in parallel. This **parallel connection** method was widely used by browsers before persistent connections. It has the same disadvantage as sequential connections—extra overhead—but much better performance. This is because setting up and ramping up the connections in parallel hides some of the latency. In our example, connections for both of the embedded images could be set up at the same time. However, running many TCP connections to the same server is discouraged. The reason is that TCP performs congestion control for each connection independently. As a consequence, the connections compete against each other, causing added packet loss, and in aggregate are more aggressive users of the network than an individual connection. Persistent connections are superior and used in preference to parallel connections because they avoid overhead and do not suffer from congestion problems.

Methods

Although HTTP was designed for use in the Web, it was intentionally made more general than necessary with an eye to future object-oriented uses. For this reason, operations, called **methods**, other than just requesting a Web page are supported. This generality is what permitted SOAP to come into existence.

Each request consists of one or more lines of ASCII text, with the first word on the first line being the name of the method requested. The built-in methods are listed in Fig. 7-37. The names are case sensitive, so *GET* is allowed but not *get*.

Method	Description
GET	Read a Web page
HEAD	Read a Web page's header
POST	Append to a Web page
PUT	Store a Web page
DELETE	Remove the Web page
TRACE	Echo the incoming request
CONNECT	Connect through a proxy
OPTIONS	Query options for a page

Figure 7-37. The built-in HTTP request methods.

The *GET* method requests the server to send the page. (When we say "page" we mean "object" in the most general case, but thinking of a page as the contents

of a file is sufficient to understand the concepts.) The page is suitably encoded in MIME. The vast majority of requests to Web servers are *GET*s. The usual form of *GET* is

```
GET filename HTTP/1.1
```

where *filename* names the page to be fetched and 1.1 is the protocol version.

The *HEAD* method just asks for the message header, without the actual page. This method can be used to collect information for indexing purposes, or just to test a URL for validity.

The *POST* method is used when forms are submitted. Both it and *GET* are also used for SOAP Web services. Like *GET*, it bears a URL, but instead of simply retrieving a page it uploads data to the server (i.e., the contents of the form or RPC parameters). The server then does something with the data that depends on the URL, conceptually appending the data to the object. The effect might be to purchase an item, for example, or to call a procedure. Finally, the method returns a page indicating the result.

The remaining methods are not used much for browsing the Web. The *PUT* method is the reverse of *GET*: instead of reading the page, it writes the page. This method makes it possible to build a collection of Web pages on a remote server. The body of the request contains the page. It may be encoded using MIME, in which case the lines following the *PUT* might include authentication headers, to prove that the caller indeed has permission to perform the requested operation.

DELETE does what you might expect: it removes the page, or at least it indicates that the Web server has agreed to remove the page. As with *PUT*, authentication and permission play a major role here.

The *TRACE* method is for debugging. It instructs the server to send back the request. This method is useful when requests are not being processed correctly and the client wants to know what request the server actually got.

The *CONNECT* method lets a user make a connection to a Web server through an intermediate device, such as a Web cache.

The *OPTIONS* method provides a way for the client to query the server for a page and obtain the methods and headers that can be used with that page.

Every request gets a response consisting of a status line, and possibly additional information (e.g., all or part of a Web page). The status line contains a three-digit status code telling whether the request was satisfied and, if not, why not. The first digit is used to divide the responses into five major groups, as shown in Fig. 7-38. The 1xx codes are rarely used in practice. The 2xx codes mean that the request was handled successfully and the content (if any) is being returned. The 3xx codes tell the client to look elsewhere, either using a different URL or in its own cache (discussed later). The 4xx codes mean the request failed due to a client error such an invalid request or a nonexistent page. Finally, the 5xx errors mean the server itself has an internal problem, either due to an error in its code or to a temporary overload.

Code	Meaning	Examples
1xx	Information	100 = server agrees to handle client's request
2xx	Success	200 = request succeeded; 204 = no content present
3xx	Redirection	301 = page moved; 304 = cached page still valid
4xx	Client error	403 = forbidden page; 404 = page not found
5xx	Server error	500 = internal server error; 503 = try again later

Figure 7-38. The status code response groups.

Message Headers

The request line (e.g., the line with the *GET* method) may be followed by additional lines with more information. They are called **request headers**. This information can be compared to the parameters of a procedure call. Responses may also have **response headers**. Some headers can be used in either direction. A selection of the more important ones is given in Fig. 7-39. This list is not short, so as you might imagine there is often a variety of headers on each request and response.

The *User-Agent* header allows the client to inform the server about its browser implementation (e.g., *Mozilla/5.0* and *Chrome/5.0.375.125*). This information is useful to let servers tailor their responses to the browser, since different browsers can have widely varying capabilities and behaviors.

The four *Accept* headers tell the server what the client is willing to accept in the event that it has a limited repertoire of what is acceptable. The first header specifies the MIME types that are welcome (e.g., *text/html*). The second gives the character set (e.g., *ISO-8859-5* or *Unicode-1-1*). The third deals with compression methods (e.g., *gzip*). The fourth indicates a natural language (e.g., Spanish). If the server has a choice of pages, it can use this information to supply the one the client is looking for. If it is unable to satisfy the request, an error code is returned and the request fails.

The *If-Modified-Since* and *If-None-Match* headers are used with caching. They let the client ask for a page to be sent only if the cached copy is no longer valid. We will describe caching shortly.

The *Host* header names the server. It is taken from the URL. This header is mandatory. It is used because some IP addresses may serve multiple DNS names and the server needs some way to tell which host to hand the request to.

The *Authorization* header is needed for pages that are protected. In this case, the client may have to prove it has a right to see the page requested. This header is used for that case.

The client uses the misspelled *Referer* header to give the URL that referred to the URL that is now requested. Most often this is the URL of the previous page.

Header	Type	Contents
User-Agent	Request	Information about the browser and its platform
Accept	Request	The type of pages the client can handle
Accept-Charset	Request	The character sets that are acceptable to the client
Accept-Encoding	Request	The page encodings the client can handle
Accept-Language	Request	The natural languages the client can handle
If-Modified-Since	Request	Time and date to check freshness
If-None-Match	Request	Previously sent tags to check freshness
Host	Request	The server's DNS name
Authorization	Request	A list of the client's credentials
Referer	Request	The previous URL from which the request came
Cookie	Request	Previously set cookie sent back to the server
Set-Cookie	Response	Cookie for the client to store
Server	Response	Information about the server
Content-Encoding	Response	How the content is encoded (e.g., *gzip*)
Content-Language	Response	The natural language used in the page
Content-Length	Response	The page's length in bytes
Content-Type	Response	The page's MIME type
Content-Range	Response	Identifies a portion of the page's content
Last-Modified	Response	Time and date the page was last changed
Expires	Response	Time and date when the page stops being valid
Location	Response	Tells the client where to send its request
Accept-Ranges	Response	Indicates the server will accept byte range requests
Date	Both	Date and time the message was sent
Range	Both	Identifies a portion of a page
Cache-Control	Both	Directives for how to treat caches
ETag	Both	Tag for the contents of the page
Upgrade	Both	The protocol the sender wants to switch to

Figure 7-39. Some HTTP message headers.

This header is particularly useful for tracking Web browsing, as it tells servers how a client arrived at the page.

Although cookies are dealt with in RFC 2109 rather than RFC 2616, they also have headers. The *Set-Cookie* header is how servers send cookies to clients. The client is expected to save the cookie and return it on subsequent requests to the server by using the *Cookie* header. (Note that there is a more recent specification for cookies with newer headers, RFC 2965, but this has largely been rejected by industry and is not widely implemented.)

Many other headers are used in responses. The *Server* header allows the server to identify its software build if it wishes. The next five headers, all starting with *Content-*, allow the server to describe properties of the page it is sending.

The *Last-Modified* header tells when the page was last modified, and the *Expires* header tells for how long the page will remain valid. Both of these headers play an important role in page caching.

The *Location* header is used by the server to inform the client that it should try a different URL. This can be used if the page has moved or to allow multiple URLs to refer to the same page (possibly on different servers). It is also used for companies that have a main Web page in the *com* domain but redirect clients to a national or regional page based on their IP addresses or preferred language.

If a page is very large, a small client may not want it all at once. Some servers will accept requests for byte ranges, so the page can be fetched in multiple small units. The *Accept-Ranges* header announces the server's willingness to handle this type of partial page request.

Now we come to headers that can be used in both directions. The *Date* header can be used in both directions and contains the time and date the message was sent, while the *Range* header tells the byte range of the page that is provided by the response.

The *ETag* header gives a short tag that serves as a name for the content of the page. It is used for caching. The *Cache-Control* header gives other explicit instructions about how to cache (or, more usually, how not to cache) pages.

Finally, the *Upgrade* header is used for switching to a new communication protocol, such as a future HTTP protocol or a secure transport. It allows the client to announce what it can support and the server to assert what it is using.

Caching

People often return to Web pages that they have viewed before, and related Web pages often have the same embedded resources. Some examples are the images that are used for navigation across the site, as well as common style sheets and scripts. It would be very wasteful to fetch all of these resources for these pages each time they are displayed because the browser already has a copy.

Squirreling away pages that are fetched for subsequent use is called **caching**. The advantage is that when a cached page can be reused, it is not necessary to repeat the transfer. HTTP has built-in support to help clients identify when they can safely reuse pages. This support improves performance by reducing both network traffic and latency. The trade-off is that the browser must now store pages, but this is nearly always a worthwhile trade-off because local storage is inexpensive. The pages are usually kept on disk so that they can be used when the browser is run at a later date.

The difficult issue with HTTP caching is how to determine that a previously cached copy of a page is the same as the page would be if it was fetched again.

This determination cannot be made solely from the URL. For example, the URL may give a page that displays the latest news item. The contents of this page will be updated frequently even though the URL stays the same. Alternatively, the contents of the page may be a list of the gods from Greek and Roman mythology. This page should change somewhat less rapidly.

HTTP uses two strategies to tackle this problem. They are shown in Fig. 7-40 as forms of processing between the request (step 1) and the response (step 5). The first strategy is page validation (step 2). The cache is consulted, and if it has a copy of a page for the requested URL that is known to be fresh (i.e., still valid), there is no need to fetch it anew from the server. Instead, the cached page can be returned directly. The *Expires* header returned when the cached page was originally fetched and the current date and time can be used to make this determination.

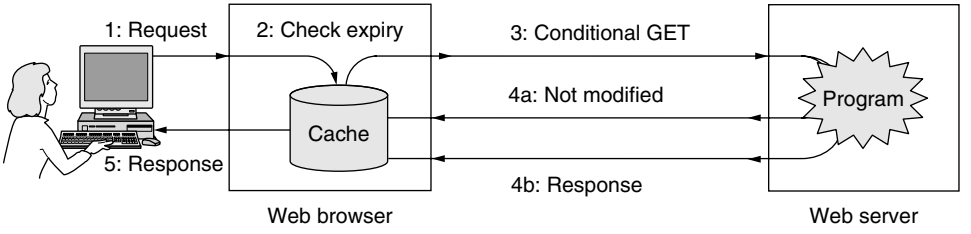


Figure 7-40. HTTP caching.

However, not all pages come with a convenient *Expires* header that tells when the page must be fetched again. After all, making predictions is hard—especially about the future. In this case, the browser may use heuristics. For example, if the page has not been modified in the past year (as told by the *Last-Modified* header) it is a fairly safe bet that it will not change in the next hour. There is no guarantee, however, and this may be a bad bet. For example, the stock market might have closed for the day so that the page will not change for hours, but it will change rapidly once the next trading session starts. Thus, the cacheability of a page may vary wildly over time. For this reason, heuristics should be used with care, though they often work well in practice.

Finding pages that have not expired is the most beneficial use of caching because it means that the server does not need to be contacted at all. Unfortunately, it does not always work. Servers must use the *Expires* header conservatively, since they may be unsure when a page will be updated. Thus, the cached copies may still be fresh, but the client does not know.

The second strategy is used in this case. It is to ask the server if the cached copy is still valid. This request is a **conditional GET**, and it is shown in Fig. 7-40 as step 3. If the server knows that the cached copy is still valid, it can send a short reply to say so (step 4a). Otherwise, it must send the full response (step 4b).

More header fields are used to let the server check whether a cached copy is still valid. The client has the time a cached page was last updated from the *Last-Modified* header. It can send this time to the server using the *If-Modified-Since* header to ask for the page only if it has been changed in the meantime.

Alternatively, the server may return an *ETag* header with a page. This header gives a tag that is a short name for the content of the page, like a checksum but better. (It can be a cryptographic hash, which we will describe in Chap. 8.) The client can validate cached copies by sending the server an *If-None-Match* header listing the tags of the cached copies. If any of the tags match the content that the server would respond with, the corresponding cached copy may be used. This method can be used when it is not convenient or useful to determine freshness. For example, a server may return different content for the same URL depending on what languages and MIME types are preferred. In this case, the modification date alone will not help the server to determine if the cached page is fresh.

Finally, note that both of these caching strategies are overridden by the directives carried in the *Cache-Control* header. These directives can be used to restrict caching (e.g., *no-cache*) when it is not appropriate. An example is a dynamic page that will be different the next time it is fetched. Pages that require authorization are also not cached.

There is much more to caching, but we only have the space to make two important points. First, caching can be performed at other places besides in the browser. In the general case, HTTP requests can be routed through a series of caches. The use of a cache external to the browser is called **proxy caching**. Each added level of caching can help to reduce requests further up the chain. It is common for organizations such as ISPs and companies to run proxy caches to gain the benefits of caching pages across different users. We will discuss proxy caching with the broader topic of content distribution in Sec. 7.5 at the end of this chapter.

Second, caches provide an important boost to performance, but not as much as one might hope. The reason is that, while there are certainly popular documents on the Web, there are also a great many unpopular documents that people fetch, many of which are also very long (e.g., videos). The ''long tail'' of unpopular documents take up space in caches, and the number of requests that can be handled from the cache grows only slowly with the size of the cache. Web caches are always likely to be able to handle less than half of the requests. See Breslau et al. (1999) for more information.

Experimenting with HTTP

Because HTTP is an ASCII protocol, it is quite easy for a person at a terminal (as opposed to a browser) to directly talk to Web servers. All that is needed is a TCP connection to port 80 on the server. Readers are encouraged to experiment with the following command sequence. It will work in most UNIX shells and the command window on Windows (once the telnet program is enabled).

```
telnet www.ietf.org 80
GET /rfc.html HTTP/1.1
Host: www.ietf.org
```

This sequence of commands starts up a telnet (i.e., TCP) connection to port 80 on IETF's Web server, *www.ietf.org*. Then comes the *GET* command naming the path of the URL and the protocol. Try servers and URLs of your choosing. The next line is the mandatory *Host* header. A blank line following the last header is mandatory. It tells the server that there are no more request headers. The server will then send the response. Depending on the server and the URL, many different kinds of headers and pages can be observed.

7.3.5 The Mobile Web

The Web is used from most every type of computer, and that includes mobile phones. Browsing the Web over a wireless network while mobile can be very useful. It also presents technical problems because much Web content was designed for flashy presentations on desktop computers with broadband connectivity. In this section we will describe how Web access from mobile devices, or the **mobile Web**, is being developed.

Compared to desktop computers at work or at home, mobile phones present several difficulties for Web browsing:

1. Relatively small screens preclude large pages and large images.
2. Limited input capabilities make it tedious to enter URLs or other lengthy input.
3. Network bandwidth is limited over wireless links, particularly on cellular (3G) networks, where it is often expensive too.
4. Connectivity may be intermittent.
5. Computing power is limited, for reasons of battery life, size, heat dissipation, and cost.

These difficulties mean that simply using desktop content for the mobile Web is likely to deliver a frustrating user experience.

Early approaches to the mobile Web devised a new protocol stack tailored to wireless devices with limited capabilities. **WAP** (**Wireless Application Protocol**) is the most well-known example of this strategy. The WAP effort was started in 1997 by major mobile phone vendors that included Nokia, Ericsson, and Motorola. However, something unexpected happened along the way. Over the next decade, network bandwidth and device capabilities grew tremendously with the deployment of 3G data services and mobile phones with larger color displays,

faster processors, and 802.11 wireless capabilities. All of a sudden, it was possible for mobiles to run simple Web browsers. There is still a gap between these mobiles and desktops that will never close, but many of the technology problems that gave impetus to a separate protocol stack have faded.

The approach that is increasingly used is to run the same Web protocols for mobiles and desktops, and to have Web sites deliver mobile-friendly content when the user happens to be on a mobile device. Web servers are able to detect whether to return desktop or mobile versions of Web pages by looking at the request headers. The *User-Agent* header is especially useful in this regard because it identifies the browser software. Thus, when a Web server receives a request, it may look at the headers and return a page with small images, less text, and simpler navigation to an iPhone and a full-featured page to a user on a laptop.

W3C is encouraging this approach in several ways. One way is to standardize best practices for mobile Web content. A list of 60 such best practices is provided in the first specification (Rabin and McCathieNevile, 2008). Most of these practices take sensible steps to reduce the size of pages, including by the use of compression, since the costs of communication are higher than those of computation, and by maximizing the effectiveness of caching. This approach encourages sites, especially large sites, to create mobile Web versions of their content because that is all that is required to capture mobile Web users. To help those users along, there is also a logo to indicate pages that can be viewed (well) on the mobile Web.

Another useful tool is a stripped-down version of HTML called **XHTML Basic**. This language is a subset of XHTML that is intended for use by mobile phones, televisions, PDAs, vending machines, pagers, cars, game machines, and even watches. For this reason, it does not support style sheets, scripts, or frames, but most of the standard tags are there. They are grouped into 11 modules. Some are required; some are optional. All are defined in XML. The modules and some example tags are listed in Fig. 7-41.

However, not all pages will be designed to work well on the mobile Web. Thus, a complementary approach is the use of **content transformation** or **transcoding**. In this approach, a computer that sits between the mobile and the server takes requests from the mobile, fetches content from the server, and transforms it to mobile Web content. A simple transformation is to reduce the size of large images by reformatting them at a lower resolution. Many other small but useful transformations are possible. Transcoding has been used with some success since the early days of the mobile Web. See, for example, Fox et al. (1996). However, when both approaches are used there is a tension between the mobile content decisions that are made by the server and by the transcoder. For instance, a Web site may select a particular combination of image and text for a mobile Web user, only to have a transcoder change the format of the image.

Our discussion so far has been about content, not protocols, as it is the content that is the biggest problem in realizing the mobile Web. However, we will briefly mention the issue of protocols. The HTTP, TCP, and IP protocols used by the

Module	Req.?	Function	Example tags
Structure	Yes	Doc. structure	body, head, html, title
Text	Yes	Information	br, code, dfn, em, h*n*, kbd, p, strong
Hypertext	Yes	Hyperlinks	a
List	Yes	Itemized lists	dl, dt, dd, ol, ul, li
Forms	No	Fill-in forms	form, input, label, option, textarea
Tables	No	Rectangular tables	caption, table, td, th, tr
Image	No	Pictures	img
Object	No	Applets, maps, etc.	object, param
Meta-information	No	Extra info	meta
Link	No	Similar to <a>	link
Base	No	URL starting point	base

Figure 7-41. The XHTML Basic modules and tags.

Web may consume a significant amount of bandwidth on protocol overheads such as headers. To tackle this problem, WAP and other solutions defined special-purpose protocols. This turns out to be largely unecessary. Header compression technologies, such as ROHC (RObust Header Compression) described in Chap. 6, can reduce the overheads of these protocols. In this way, it is possible to have one set of protocols (HTTP, TCP, IP) and use them over either high- or low- bandwidth links. Use over the low-bandwidth links simply requires that header compression be turned on.

7.3.6 Web Search

To finish our description of the Web, we will discuss what is arguably the most successful Web application: search. In 1998, Sergey Brin and Larry Page, then graduate students at Stanford, formed a startup called Google to build a better Web search engine. They were armed with the then-radical idea that a search algorithm that counted how many times each page was pointed to by other pages was a better measure of its importance than how many times it contained the key words being sought. For instance, many pages link to the main Cisco page, which makes this page more important to a user searching for ''Cisco'' than a page outside of the company that happens to use the word ''Cisco'' many times.

They were right. It did prove possible to build a better search engine, and people flocked to it. Backed by venture capital, Google grew tremendously. It became a public company in 2004, with a market capitalization of $23 billion. By 2010, it was estimated to run more than one million servers in data centers throughout the world.

In one sense, search is simply another Web application, albeit one of the most mature Web applications because it has been under development since the early days of the Web. However, Web search has proved indispensible in everyday usage. Over one billion Web searches are estimated to be done each day. People looking for all manner of information use search as a starting point. For example, to find out where to buy Vegemite in Seattle, there is no obvious Web site to use as a starting point. But chances are that a search engine knows of a page with the desired information and can quickly direct you to the answer.

To perform a Web search in the traditional manner, the user directs her browser to the URL of a Web search site. The major search sites include Google, Yahoo!, and Bing. Next, the user submits search terms using a form. This act causes the search engine to perform a query on its database for relevant pages or images, or whatever kind of resource is being searched for, and return the result as a dynamic page. The user can then follow links to the pages that have been found.

Web search is an interesting topic for discussion because it has implications for the design and use of networks. First, there is the question of how Web search finds pages. The Web search engine must have a database of pages to run a query. Each HTML page may contain links to other pages, and everything interesting (or at least searchable) is linked somewhere. This means that it is theoretically possible to start with a handful of pages and find all other pages on the Web by doing a traversal of all pages and links. This process is called **Web crawling**. All Web search engines use Web crawlers.

One issue with crawling is the kind of pages that it can find. Fetching static documents and following links is easy. However, many Web pages contain programs that display different pages depending on user interaction. An example is an online catalog for a store. The catalog may contain dynamic pages created from a product database and queries for different products. This kind of content is different from static pages that are easy to traverse. How do Web crawlers find these dynamic pages? The answer is that, for the most part, they do not. This kind of hidden content is called the **deep Web**. How to search the deep Web is an open problem that researchers are now tackling. See, for example, madhavan et al. (2008). There are also conventions by which sites make a page (known as *robots.txt*) to tell crawlers what parts of the sites should or should not be visited.

A second consideration is how to process all of the crawled data. To let indexing algorithms be run over the mass of data, the pages must be stored. Estimates vary, but the main search engines are thought to have an index of tens of billions of pages taken from the visible part of the Web. The average page size is estimated at 320 KB. These figures mean that a crawled copy of the Web takes on the order of 20 petabytes or 2×10^{16} bytes to store. While this is a truly huge number, it is also an amount of data that can comfortably be stored and processed in Internet data centers (Chang et al., 2006). For example, if disk storage costs \$20/TB, then 2×10^4 TB costs \$400,000, which is not exactly a huge amount for companies the size of Google, Microsoft, and Yahoo!. And while the Web is

expanding, disk costs are dropping dramatically, so storing the entire Web may continue to be feasible for large companies for the foreseeable future.

Making sense of this data is another matter. You can appreciate how XML can help programs extract the structure of the data easily, while ad hoc formats will lead to much guesswork. There is also the issue of conversion between formats, and even translation between languages. But even knowing the structure of data is only part of the problem. The hard bit is to understand what it means. This is where much value can be unlocked, starting with more relevant result pages for search queries. The ultimate goal is to be able to answer questions, for example, where to buy a cheap but decent toaster oven in your city.

A third aspect of Web search is that it has come to provide a higher level of naming. There is no need to remember a long URL if it is just as reliable (or perhaps more) to search for a Web page by a person's name, assuming that you are better at remembering names than URLs. This strategy is increasingly successful. In the same way that DNS names relegated IP addresses to computers, Web search is relegating URLs to computers. Also in favor of search is that it corrects spelling and typing errors, whereas if you type in a URL wrong, you get the wrong page.

Finally, Web search shows us something that has little to do with network design but much to do with the growth of some Internet services: there is much money in advertising. Advertising is the economic engine that has driven the growth of Web search. The main change from print advertising is the ability to target advertisements depending on what people are searching for, to increase the relevance of the advertisements. Variations on an auction mechanism are used to match the search query to the most valuable advertisement (Edelman et al., 2007). This new model has given rise to new problems, of course, such as **click fraud**, in which programs imitate users and click on advertisements to cause payments that have not been fairly earned.

7.4 STREAMING AUDIO AND VIDEO

Web applications and the mobile Web are not the only exciting developments in the use of networks. For many people, audio and video are the holy grail of networking. When the word "multimedia" is mentioned, both the propellerheads and the suits begin salivating as if on cue. The former see immense technical challenges in providing voice over IP and video-on-demand to every computer. The latter see equally immense profits in it.

While the idea of sending audio and video over the Internet has been around since the 1970s at least, it is only since roughly 2000 that **real-time audio** and **real-time video** traffic has grown with a vengeance. Real-time traffic is different from Web traffic in that it must be played out at some predetermined rate to be useful. After all, watching a video in slow motion with fits and starts is not most

people's idea of fun. In contrast, the Web can have short interruptions, and page loads can take more or less time, within limits, without it being a major problem.

Two things happened to enable this growth. First, computers have became much more powerful and are equipped with microphones and cameras so that they can input, process, and output audio and video data with ease. Second, a flood of Internet bandwidth has come to be available. Long-haul links in the core of the Internet run at many gigabits/sec, and broadband and 802.11 wireless reaches users at the edge of the Internet. These developments allow ISPs to carry tremendous levels of traffic across their backbones and mean that ordinary users can connect to the Internet 100–1000 times faster than with a 56-kbps telephone modem.

The flood of bandwidth caused audio and video traffic to grow, but for different reasons. Telephone calls take up relatively little bandwidth (in principle 64 kbps but less when compressed) yet telephone service has traditionally been expensive. Companies saw an opportunity to carry voice traffic over the Internet using existing bandwidth to cut down on their telephone bills. Startups such as Skype saw a way to let customers make free telephone calls using their Internet connections. Upstart telephone companies saw a cheap way to carry traditional voice calls using IP networking equipment. The result was an explosion of voice data carried over Internet networks that is called **voice over IP** or **Internet telephony**.

Unlike audio, video takes up a large amount of bandwidth. Reasonable quality Internet video is encoded with compression at rates of around 1 Mbps, and a typical DVD movie is 2 GB of data. Before broadband Internet access, sending movies over the network was prohibitive. Not so any more. With the spread of broadband, it became possible for the first time for users to watch decent, streamed video at home. People love to do it. Around a quarter of the Internet users on any given day are estimated to visit YouTube, the popular video sharing site. The movie rental business has shifted to online downloads. And the sheer size of videos has changed the overall makeup of Internet traffic. The majority of Internet traffic is already video, and it is estimated that 90% of Internet traffic will be video within a few years (Cisco, 2010).

Given that there is enough bandwidth to carry audio and video, the key issue for designing streaming and conferencing applications is network delay. Audio and video need real-time presentation, meaning that they must be played out at a predetermined rate to be useful. Long delays mean that calls that should be interactive no longer are. This problem is clear if you have ever talked on a satellite phone, where the delay of up to half a second is quite distracting. For playing music and movies over the network, the absolute delay does not matter, because it only affects when the media starts to play. But the variation in delay, called **jitter**, still matters. It must be masked by the player or the audio will sound unintelligible and the video will look jerky.

In this section, we will discuss some strategies to handle the delay problem, as well as protocols for setting up audio and video sessions. After an introduction to

digital audio and video, our presentation is broken into three cases for which different designs are used. The first and easiest case to handle is streaming stored media, like watching a video on YouTube. The next case in terms of difficulty is streaming live media. Two examples are Internet radio and IPTV, in which radio and television stations broadcast to many users live on the Internet. The last and most difficult case is a call as might be made with Skype, or more generally an interactive audio and video conference.

As an aside, the term **multimedia** is often used in the context of the Internet to mean video and audio. Literally, multimedia is just two or more media. That definition makes this book a multimedia presentation, as it contains text and graphics (the figures). However, that is probably not what you had in mind, so we use the term "multimedia" to imply two or more **continuous media**, that is, media that have to be played during some well-defined time interval. The two media are normally video with audio, that is, moving pictures with sound. Many people also refer to pure audio, such as Internet telephony or Internet radio, as multimedia as well, which it is clearly not. Actually, a better term for all these cases is **streaming media**. Nonetheless, we will follow the herd and consider real-time audio to be multimedia as well.

7.4.1 Digital Audio

An audio (sound) wave is a one-dimensional acoustic (pressure) wave. When an acoustic wave enters the ear, the eardrum vibrates, causing the tiny bones of the inner ear to vibrate along with it, sending nerve pulses to the brain. These pulses are perceived as sound by the listener. In a similar way, when an acoustic wave strikes a microphone, the microphone generates an electrical signal, representing the sound amplitude as a function of time.

The frequency range of the human ear runs from 20 Hz to 20,000 Hz. Some animals, notably dogs, can hear higher frequencies. The ear hears loudness logarithmically, so the ratio of two sounds with power A and B is conventionally expressed in **dB** (**decibels**) as the quantity $10 \log_{10}(A/B)$. If we define the lower limit of audibility (a sound pressure of about 20 μPascals) for a 1-kHz sine wave as 0 dB, an ordinary conversation is about 50 dB and the pain threshold is about 120 dB. The dynamic range is a factor of more than 1 million.

The ear is surprisingly sensitive to sound variations lasting only a few milliseconds. The eye, in contrast, does not notice changes in light level that last only a few milliseconds. The result of this observation is that jitter of only a few milliseconds during the playout of multimedia affects the perceived sound quality much more than it affects the perceived image quality.

Digital audio is a digital representation of an audio wave that can be used to recreate it. Audio waves can be converted to digital form by an **ADC** (**Analog-to-Digital Converter**). An ADC takes an electrical voltage as input and generates a binary number as output. In Fig. 7-42(a) we see an example of a sine wave.

To represent this signal digitally, we can sample it every ΔT seconds, as shown by the bar heights in Fig. 7-42(b). If a sound wave is not a pure sine wave but a linear superposition of sine waves where the highest frequency component present is f, the Nyquist theorem (see Chap. 2) states that it is sufficient to make samples at a frequency $2f$. Sampling more often is of no value since the higher frequencies that such sampling could detect are not present.

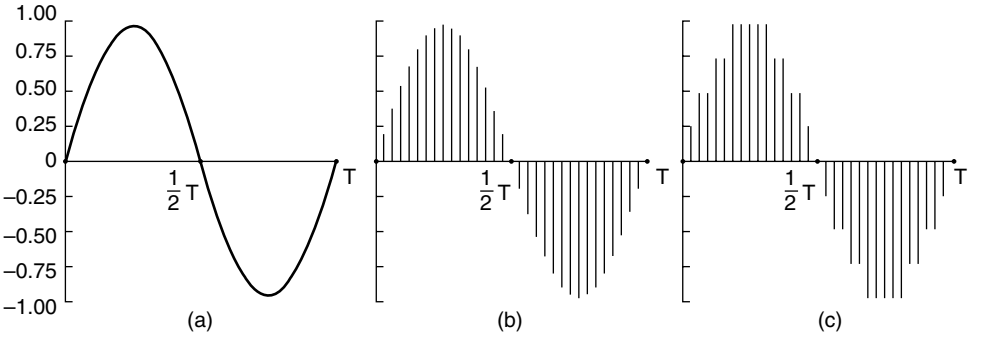


Figure 7-42. (a) A sine wave. (b) Sampling the sine wave. (c) Quantizing the samples to 4 bits.

The reverse process takes digital values and produces an analog electrical voltage. It is done by a **DAC** (**Digital-to-Analog Converter**). A loudspeaker can then convert the analog voltage to acoustic waves so that people can hear sounds.

Digital samples are never exact. The samples of Fig. 7-42(c) allow only nine values, from −1.00 to +1.00 in steps of 0.25. An 8-bit sample would allow 256 distinct values. A 16-bit sample would allow 65,536 distinct values. The error introduced by the finite number of bits per sample is called the **quantization noise**. If it is too large, the ear detects it.

Two well-known examples where sampled sound is used are the telephone and audio compact discs. Pulse code modulation, as used within the telephone system, uses 8-bit samples made 8000 times per second. The scale is nonlinear to minimize perceived distortion, and with only 8000 samples/sec, frequencies above 4 kHz are lost. In North America and Japan, the **μ-law** encoding is used. In Europe and internationally, the **A-law** encoding is used. Each encoding gives a data rate of 64,000 bps.

Audio CDs are digital with a sampling rate of 44,100 samples/sec, enough to capture frequencies up to 22,050 Hz, which is good enough for people but bad for canine music lovers. The samples are 16 bits each and are linear over the range of amplitudes. Note that 16-bit samples allow only 65,536 distinct values, even though the dynamic range of the ear is more than 1 million. Thus, even though CD-quality audio is much better than telephone-quality audio, using only 16 bits per sample introduces noticeable quantization noise (although the full dynamic range is not covered—CDs are not supposed to hurt). Some fanatic audiophiles

still prefer 33-RPM LP records to CDs because records do not have a Nyquist frequency cutoff at 22 kHz and have no quantization noise. (But they do have scratches unless handled very carefully) With 44,100 samples/sec of 16 bits each, uncompressed CD-quality audio needs a bandwidth of 705.6 kbps for monaural and 1.411 Mbps for stereo.

Audio Compression

Audio is often compressed to reduce bandwidth needs and transfer times, even though audio data rates are much lower than video data rates. All compression systems require two algorithms: one for compressing the data at the source, and another for decompressing it at the destination. In the literature, these algorithms are referred to as the **encoding** and **decoding** algorithms, respectively. We will use this terminology too.

Compression algorithms exhibit certain asymmetries that are important to understand. Even though we are considering audio first, these asymmetries hold for video as well. For many applications, a multimedia document will only be encoded once (when it is stored on the multimedia server) but will be decoded thousands of times (when it is played back by customers). This asymmetry means that it is acceptable for the encoding algorithm to be slow and require expensive hardware provided that the decoding algorithm is fast and does not require expensive hardware. The operator of a popular audio (or video) server might be quite willing to buy a cluster of computers to encode its entire library, but requiring customers to do the same to listen to music or watch movies is not likely to be a big success. Many practical compression systems go to great lengths to make decoding fast and simple, even at the price of making encoding slow and complicated.

On the other hand, for live audio and video, such as a voice-over-IP calls, slow encoding is unacceptable. Encoding must happen on the fly, in real time. Consequently, real-time multimedia uses different algorithms or parameters than stored audio or videos on disk, often with appreciably less compression.

A second asymmetry is that the encode/decode process need not be invertible. That is, when compressing a data file, transmitting it, and then decompressing it, the user expects to get the original back, accurate down to the last bit. With multimedia, this requirement does not exist. It is usually acceptable to have the audio (or video) signal after encoding and then decoding be slightly different from the original as long as it sounds (or looks) the same. When the decoded output is not exactly equal to the original input, the system is said to be **lossy**. If the input and output are identical, the system is **lossless**. Lossy systems are important because accepting a small amount of information loss normally means a huge payoff in terms of the compression ratio possible.

Historically, long-haul bandwidth in the telephone network was very expensive, so there is a substantial body of work on **vocoders** (short for "voice coders") that compress audio for the special case of speech. Human speech tends to be in

the 600-Hz to 6000-Hz range and is produced by a mechanical process that depends on the speaker's vocal tract, tongue, and jaw. Some vocoders make use of models of the vocal system to reduce speech to a few parameters (e.g., the sizes and shapes of various cavities) and a data rate of as little as 2.4 kbps. How these vocoders work is beyond the scope of this book, however.

We will concentrate on audio as sent over the Internet, which is typically closer to CD-quality. It is also desirable to reduce the data rates for this kind of audio. At 1.411 Mbps, stereo audio would tie up many broadband links, leaving less room for video and other Web traffic. Its data rate with compression can be reduced by an order of magnitude with little to no perceived loss of quality.

Compression and decompression require signal processing. Fortunately, digitized sound and movies can be easily processed by computers in software. In fact, dozens of programs exist to let users record, display, edit, mix, and store media from multiple sources. This has led to large amounts of music and movies being available on the Internet—not all of it legal—which has resulted in numerous lawsuits from the artists and copyright owners.

Many audio compression algorithms have been developed. Probably the most popular formats are **MP3** (**MPEG audio layer 3**) and **AAC** (**Advanced Audio Coding**) as carried in **MP4** (**MPEG-4**) files. To avoid confusion, note that MPEG provides audio and video compression. MP3 refers to the audio compression portion (part 3) of the MPEG-1 standard, not the third version of MPEG. In fact, no third version of MPEG was released, only MPEG-1, MPEG-2, and MPEG-4. AAC is the successor to MP3 and the default audio encoding used in MPEG-4. MPEG-2 allows both MP3 and AAC audio. Is that clear now? The nice thing about standards is that there are so many to choose from. And if you do not like any of them, just wait a year or two.

Audio compression can be done in two ways. In **waveform coding**, the signal is transformed mathematically by a Fourier transform into its frequency components. In Chap. 2, we showed an example function of time and its Fourier amplitudes in Fig. 2-1(a). The amplitude of each component is then encoded in a minimal way. The goal is to reproduce the waveform fairly accurately at the other end in as few bits as possible.

The other way, **perceptual coding**, exploits certain flaws in the human auditory system to encode a signal in such a way that it sounds the same to a human listener, even if it looks quite different on an oscilloscope. Perceptual coding is based on the science of **psychoacoustics**—how people perceive sound. Both MP3 and AAC are based on perceptual coding.

The key property of perceptual coding is that some sounds can **mask** other sounds. Imagine you are broadcasting a live flute concert on a warm summer day. Then all of a sudden, out of the blue, a crew of workmen nearby turn on their jackhammers and start tearing up the street. No one can hear the flute any more. Its sounds have been masked by the jackhammers. For transmission purposes, it is now sufficient to encode just the frequency band used by the jackhammers

because the listeners cannot hear the flute anyway. This is called **frequency masking**—the ability of a loud sound in one frequency band to hide a softer sound in another frequency band that would have been audible in the absence of the loud sound. In fact, even after the jackhammers stop, the flute will be inaudible for a short period of time because the ear turns down its gain when they start and it takes a finite time to turn it up again. This effect is called **temporal masking**.

To make these effects more quantitative, imagine experiment 1. A person in a quiet room puts on headphones connected to a computer's sound card. The computer generates a pure sine wave at 100 Hz at low, but gradually increasing, power. The subject is instructed to strike a key when she hears the tone. The computer records the current power level and then repeats the experiment at 200 Hz, 300 Hz, and all the other frequencies up to the limit of human hearing. When averaged over many people, a log-log graph of how much power it takes for a tone to be audible looks like that of Fig. 7-43(a). A direct consequence of this curve is that it is never necessary to encode any frequencies whose power falls below the threshold of audibility. For example, if the power at 100 Hz were 20 dB in Fig. 7-43(a), it could be omitted from the output with no perceptible loss of quality because 20 dB at 100 Hz falls below the level of audibility.

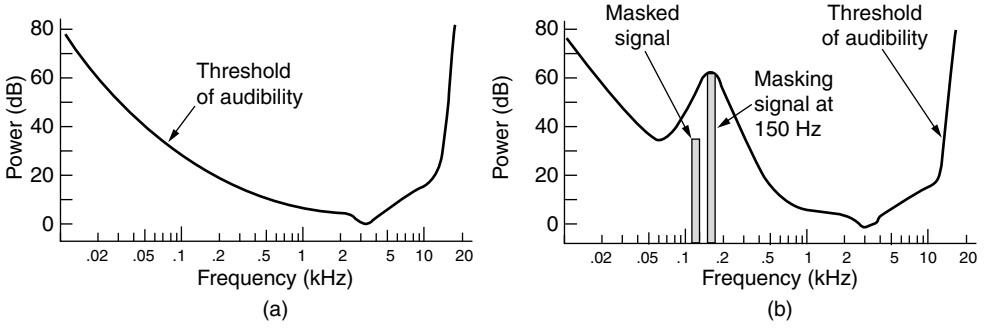


Figure 7-43. (a) The threshold of audibility as a function of frequency. (b) The masking effect.

Now consider experiment 2. The computer runs experiment 1 again, but this time with a constant-amplitude sine wave at, say, 150 Hz superimposed on the test frequency. What we discover is that the threshold of audibility for frequencies near 150 Hz is raised, as shown in Fig. 7-43(b).

The consequence of this new observation is that by keeping track of which signals are being masked by more powerful signals in nearby frequency bands, we can omit more and more frequencies in the encoded signal, saving bits. In Fig. 7-43, the 125-Hz signal can be completely omitted from the output and no one will be able to hear the difference. Even after a powerful signal stops in some frequency band, knowledge of its temporal masking properties allows us to continue to omit the masked frequencies for some time interval as the ear recovers. The

essence of MP3 and AAC is to Fourier-transform the sound to get the power at each frequency and then transmit only the unmasked frequencies, encoding these in as few bits as possible.

With this information as background, we can now see how the encoding is done. The audio compression is done by sampling the waveform at a rate from 8 to 96 kHz for AAC, often at 44.1 kHz, to mimic CD sound. Sampling can be done on one (mono) or two (stereo) channels. Next, the output bit rate is chosen. MP3 can compress a stereo rock 'n roll CD down to 96 kbps with little perceptible loss in quality, even for rock 'n roll fans with no hearing loss. For a piano concert, AAC with at least 128 kbps is needed. The difference is because the signal-to-noise ratio for rock 'n roll is much higher than for a piano concert (in an engineering sense, anyway). It is also possible to choose lower output rates and accept some loss in quality.

The samples are processed in small batches. Each batch is passed through a bank of digital filters to get frequency bands. The frequency information is fed into a psychoacoustic model to determine the masked frequencies. Then the available bit budget is divided among the bands, with more bits allocated to the bands with the most unmasked spectral power, fewer bits allocated to unmasked bands with less spectral power, and no bits allocated to masked bands. Finally, the bits are encoded using Huffman encoding, which assigns short codes to numbers that appear frequently and long codes to those that occur infrequently. There are many more details for the curious reader. For more information, see Brandenburg (1999).

7.4.2 Digital Video

Now that we know all about the ear, it is time to move on to the eye. (No, this section is not followed by one on the nose.) The human eye has the property that when an image appears on the retina, the image is retained for some number of milliseconds before decaying. If a sequence of images is drawn at 50 images/sec, the eye does not notice that it is looking at discrete images. All video systems exploit this principle to produce moving pictures.

The simplest digital representation of video is a sequence of frames, each consisting of a rectangular grid of picture elements, or **pixels**. Each pixel can be a single bit, to represent either black or white. However, the quality of such a system is awful. Try using your favorite image editor to convert the pixels of a color image to black and white (and *not* shades of gray).

The next step up is to use 8 bits per pixel to represent 256 gray levels. This scheme gives high-quality "black-and-white'' video. For color video, many systems use 8 bits for each of the red, green and blue (RGB) primary color components. This representation is possible because any color can be constructed from a linear superposition of red, green, and blue with the appropriate intensities. With

24 bits per pixel, there are about 16 million colors, which is more than the human eye can distinguish.

On color LCD computer monitors and televisions, each discrete pixel is made up of closely spaced red, green and blue subpixels. Frames are displayed by setting the intensity of the subpixels, and the eye blends the color components.

Common frame rates are 24 frames/sec (inherited from 35mm motion-picture film), 30 frames/sec (inherited from NTSC U.S. televisions), and 30 frames/sec (inherited from the PAL television system used in nearly all the rest of the world). (For the truly picky, NTSC color television runs at 29.97 frames/sec. The original black-and-white system ran at 30 frames/sec, but when color was introduced, the engineers needed a bit of extra bandwidth for signaling so they reduced the frame rate to 29.97. NTSC videos intended for computers really use 30.) PAL was invented after NTSC and really uses 25.000 frames/sec. To make this story complete, a third system, SECAM, is used in France, Francophone Africa, and Eastern Europe. It was first introduced into Eastern Europe by then Communist East Germany so the East German people could not watch West German (PAL) television lest they get Bad Ideas. But many of these countries are switching to PAL. Technology and politics at their best.

Actually, for broadcast television, 25 frames/sec is not quite good enough for smooth motion so the images are split into two **fields**, one with the odd-numbered scan lines and one with the even-numbered scan lines. The two (half-resolution) fields are broadcast sequentially, giving almost 60 (NTSC) or exactly 50 (PAL) fields/sec, a system known as **interlacing**. Videos intended for viewing on a computer are **progressive**, that is, do not use interlacing because computer monitors have buffers on their graphics cards, making it possible for the CPU to put a new image in the buffer 30 times/sec but have the graphics card redraw the screen 50 or even 100 times/sec to eliminate flicker. Analog television sets do not have a frame buffer the way computers do. When an interlaced video with rapid movement is displayed on a computer, short horizontal lines will be visible near sharp edges, an effect known as **combing**.

The frame sizes used for video sent over the Internet vary widely for the simple reason that larger frames require more bandwidth, which may not always be available. Low-resolution video might be 320 by 240 pixels, and ''full-screen'' video is 640 by 480 pixels. These dimensions approximate those of early computer monitors and NTSC television, respectively. The **aspect ratio**, or width to height ratio, of 4:3, is the same as a standard television. **HDTV** (**High-Definition TeleVision**) videos can be downloaded with 1280 by 720 pixels. These ''widescreen'' images have an aspect ratio of 16:9 to more closely match the 3:2 aspect ratio of film. For comparison, standard DVD video is usually 720 by 480 pixels, and video on Blu-ray discs is usually HDTV at 1080 by 720 pixels.

On the Internet, the number of pixels is only part of the story, as media players can present the same image at different sizes. Video is just another window on a computer screen that can be blown up or shrunk down. The role of more

pixels is to increase the quality of the image, so that it does not look blurry when it is expanded. However, many monitors can show images (and hence videos) with even more pixels than even HDTV.

Video Compression

It should be obvious from our discussion of digital video that compression is critical for sending video over the Internet. Even a standard-quality video with 640 by 480 pixel frames, 24 bits of color information per pixel, and 30 frames/sec takes over 200 Mbps. This far exceeds the bandwidth by which most company offices are connected to the Internet, let alone home users, and this is for a single video stream. Since transmitting uncompressed video is completely out of the question, at least over wide area networks, the only hope is that massive compression is possible. Fortunately, a large body of research over the past few decades has led to many compression techniques and algorithms that make video transmission feasible.

Many formats are used for video that is sent over the Internet, some proprietary and some standard. The most popular encoding is MPEG in its various forms. It is an open standard found in files with mpg and mp4 extensions, as well as in other container formats. In this section, we will look at MPEG to study how video compression is accomplished. To begin, we will look at the compression of still images with JPEG. A video is just a sequence of images (plus sound). One way to compress video is to encode each image in succession. To a first approximation, MPEG is just the JPEG encoding of each frame, plus some extra features for removing the redundancy across frames.

The JPEG Standard

The **JPEG** (**Joint Photographic Experts Group**) standard for compressing continuous-tone still pictures (e.g., photographs) was developed by photographic experts working under the joint auspices of ITU, ISO, and IEC, another standards body. It is widely used (look for files with the extension jpg) and often provides compression ratios of 10:1 or better for natural images.

JPEG is defined in International Standard 10918. Really, it is more like a shopping list than a single algorithm, but of the four modes that are defined only the lossy sequential mode is relevant to our discussion. Furthermore, we will concentrate on the way JPEG is normally used to encode 24-bit RGB video images and will leave out some of the options and details for the sake of simplicity.

The algorithm is illustrated in Fig. 7-44. Step 1 is block preparation. For the sake of specificity, let us assume that the JPEG input is a 640×480 RGB image with 24 bits/pixel, as shown in Fig. 7-44(a). RGB is not the best color model to use for compression. The eye is much more sensitive to the **luminance**, or brightness, of video signals than the **chrominance**, or color, of video signals. Thus, we

first compute the luminance, *Y*, and the two chrominances, *Cb* and *Cr*, from the *R*, *G*, and *B* components. The following formulas are used for 8-bit values that range from 0 to 255:

$$\begin{aligned} Y &= 16 + 0.26R + 0.50G + 0.09B \\ Cb &= 128 + 0.15R - 0.29G - 0.44B \\ Cr &= 128 + 0.44R - 0.37G + 0.07B \end{aligned}$$

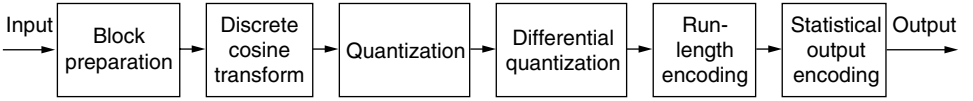


Figure 7-44. Steps in JPEG lossy sequential encoding.

Separate matrices are constructed for *Y*, *Cb*, and *Cr*. Next, square blocks of four pixels are averaged in the *Cb* and *Cr* matrices to reduce them to 320×240. This reduction is lossy, but the eye barely notices it since the eye responds to luminance more than to chrominance. Nevertheless, it compresses the total amount of data by a factor of two. Now 128 is subtracted from each element of all three matrices to put 0 in the middle of the range. Finally, each matrix is divided up into 8×8 blocks. The *Y* matrix has 4800 blocks; the other two have 1200 blocks each, as shown in Fig. 7-45(b).

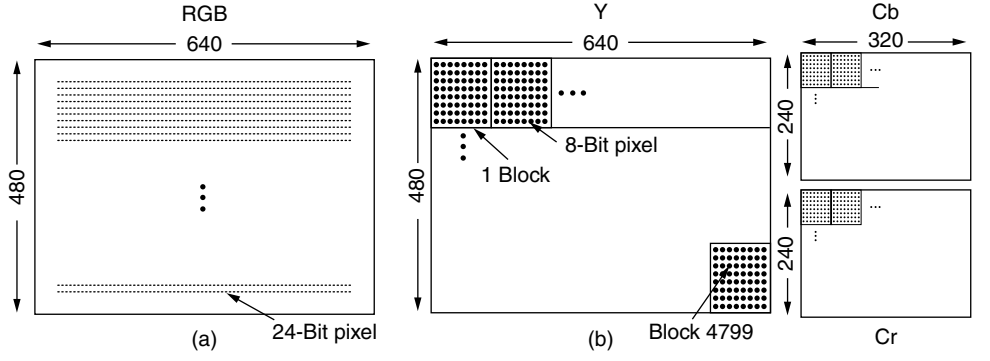


Figure 7-45. (a) RGB input data. (b) After block preparation.

Step 2 of JPEG encoding is to apply a **DCT** (**Discrete Cosine Transformation**) to each of the 7200 blocks separately. The output of each DCT is an 8×8 matrix of DCT coefficients. DCT element (0, 0) is the average value of the block. The other elements tell how much spectral power is present at each spatial frequency. Normally, these elements decay rapidly with distance from the origin, (0, 0), as suggested by Fig. 7-46.

Once the DCT is complete, JPEG encoding moves on to step 3, called **quantization**, in which the less important DCT coefficients are wiped out. This (lossy)

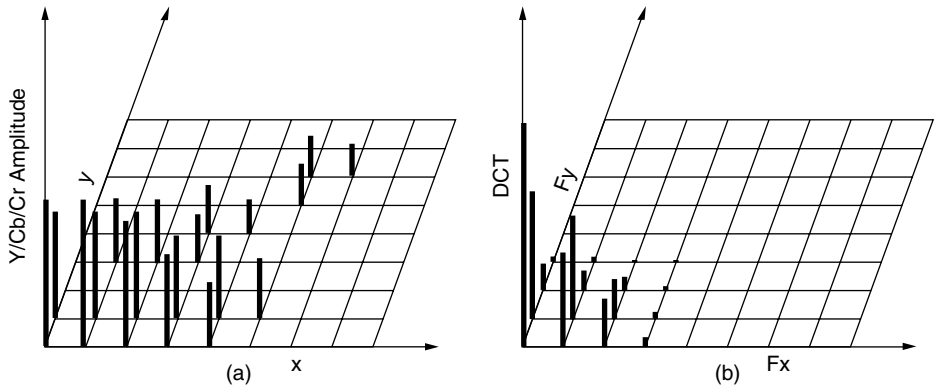


Figure 7-46. (a) One block of the Y matrix. (b) The DCT coefficients.

transformation is done by dividing each of the coefficients in the 8×8 DCT matrix by a weight taken from a table. If all the weights are 1, the transformation does nothing. However, if the weights increase sharply from the origin, higher spatial frequencies are dropped quickly.

An example of this step is given in Fig. 7-47. Here we see the initial DCT matrix, the quantization table, and the result obtained by dividing each DCT element by the corresponding quantization table element. The values in the quantization table are not part of the JPEG standard. Each application must supply its own, allowing it to control the loss-compression trade-off.

DCT coefficients

150	80	40	14	4	2	1	0
92	75	36	10	6	1	0	0
52	38	26	8	7	4	0	0
12	8	6	4	2	1	0	0
4	3	2	0	0	0	0	0
2	2	1	1	0	0	0	0
1	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Quantization table

1	1	2	4	8	16	32	64
1	1	2	4	8	16	32	64
2	2	2	4	8	16	32	64
4	4	4	4	8	16	32	64
8	8	8	8	8	16	32	64
16	16	16	16	16	16	32	64
32	32	32	32	32	32	32	64
64	64	64	64	64	64	64	64

Quantized coefficients

150	80	20	4	1	0	0	0
92	75	18	3	1	0	0	0
26	19	13	2	1	0	0	0
3	2	2	1	0	0	0	0
1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Figure 7-47. Computation of the quantized DCT coefficients.

Step 4 reduces the (0, 0) value of each block (the one in the upper-left corner) by replacing it with the amount it differs from the corresponding element in the previous block. Since these elements are the averages of their respective blocks, they should change slowly, so taking the differential values should reduce most of them to small values. No differentials are computed from the other values.

Step 5 linearizes the 64 elements and applies run-length encoding to the list. Scanning the block from left to right and then top to bottom will not concentrate the zeros together, so a zigzag scanning pattern is used, as shown in Fig. 7-48. In this example, the zigzag pattern produces 38 consecutive 0s at the end of the matrix. This string can be reduced to a single count saying there are 38 zeros, a technique known as **run-length encoding**.

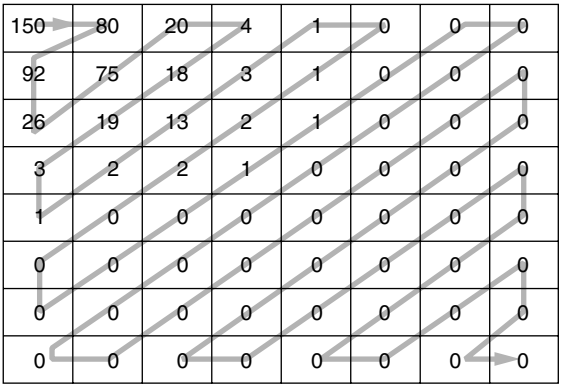


Figure 7-48. The order in which the quantized values are transmitted.

Now we have a list of numbers that represent the image (in transform space). Step 6 Huffman-encodes the numbers for storage or transmission, assigning common numbers shorter codes than uncommon ones.

JPEG may seem complicated, but that is because it *is* complicated. Still, the benefits of up to 20:1 compression are worth it. Decoding a JPEG image requires running the algorithm backward. JPEG is roughly symmetric: decoding takes as long as encoding. This property is not true of all compression algorithms, as we shall now see.

The MPEG Standard

Finally, we come to the heart of the matter: the **MPEG** (**Motion Picture Experts Group**) standards. Though there are many proprietary algorithms, these standards define the main algorithms used to compress videos. They have been international standards since 1993. Because movies contain both images and sound, MPEG can compress both audio and video. We have already examined audio compression and still image compression, so let us now examine video compression.

The MPEG-1 standard (which includes MP3 audio) was first published in 1993 and is still widely used. Its goal was to produce video-recorder-quality output that was compressed 40:1 to rates of around 1 Mbps. This video is suitable for

broad Internet use on Web sites. Do not worry if you do not remember video recorders—MPEG-1 was also used for storing movies on CDs when they existed. If you do not know what CDs are, we will have to move on to MPEG-2.

The MPEG-2 standard, released in 1996, was designed for compressing broadcast-quality video. It is very common now, as it is used as the basis for video encoded on DVDs (which inevitably finds its way onto the Internet) and for digital broadcast television (as DVB). DVD quality video is typically encoded at rates of 4–8 Mbps.

The MPEG-4 standard has two video formats. The first format, released in 1999, encodes video with an object-based representation. This allows for the mixing of natural and synthetic images and other kinds of media, for example, a weatherperson standing in front of a weather map. With this structure, it is easy to let programs interact with movie data. The second format, released in 2003, is known as **H.264** or **AVC** (**Advanced Video Coding**). Its goal is to encode video at half the rate of earlier encoders for the same quality level, all the better to support the transmission of video over networks. This encoder is used for HDTV on most Blu-ray discs.

The details of all these standards are many and varied. The later standards also have many more features and encoding options than the earlier standards. However, we will not go into the details. For the most part, the gains in video compression over time have come from numerous small improvements, rather than fundamental shifts in how video is compressed. Thus, we will sketch the overall concepts.

MPEG compresses both audio and video. Since the audio and video encoders work independently, there is an issue of how the two streams get synchronized at the receiver. The solution is to have a single clock that outputs timestamps of the current time to both encoders. These timestamps are included in the encoded output and propagated all the way to the receiver, which can use them to synchronize the audio and video streams.

MPEG video compression takes advantage of two kinds of redundancies that exist in movies: spatial and temporal. Spatial redundancy can be utilized by simply coding each frame separately with JPEG. This approach is occasionally used, especially when random access to each frame is needed, as in editing video productions. In this mode, JPEG levels of compression are achieved.

Additional compression can be achieved by taking advantage of the fact that consecutive frames are often almost identical. This effect is smaller than it might first appear since many movie directors cut between scenes every 3 or 4 seconds (time a movie fragment and count the scenes). Nevertheless, runs of 75 or more highly similar frames offer the potential of a major reduction over simply encoding each frame separately with JPEG.

For scenes in which the camera and background are stationary and one or two actors are moving around slowly, nearly all the pixels will be identical from frame to frame. Here, just subtracting each frame from the previous one and running

JPEG on the difference would do fine. However, for scenes where the camera is panning or zooming, this technique fails badly. What is needed is some way to compensate for this motion. This is precisely what MPEG does; it is the main difference between MPEG and JPEG.

MPEG output consists of three kinds of frames:

1. I- (Intracoded) frames: self-contained compressed still pictures.
2. P- (Predictive) frames: block-by-block difference with the previous frames.
3. B- (Bidirectional) frames: block-by-block differences between previous and future frames.

I-frames are just still pictures. They can be coded with JPEG or something similar. It is valuable to have I-frames appear in the output stream periodically (e.g., once or twice per second) for three reasons. First, MPEG can be used for a multicast transmission, with viewers tuning in at will. If all frames depended on their predecessors going back to the first frame, anybody who missed the first frame could never decode any subsequent frames. Second, if any frame were received in error, no further decoding would be possible: everything from then on would be unintelligble junk. Third, without I-frames, while doing a fast forward or rewind the decoder would have to calculate every frame passed over so it would know the full value of the one it stopped on.

P-frames, in contrast, code interframe differences. They are based on the idea of **macroblocks**, which cover, for example, 16×16 pixels in luminance space and 8×8 pixels in chrominance space. A macroblock is encoded by searching the previous frame for it or something only slightly different from it.

An example of where P-frames would be useful is given in Fig. 7-49. Here we see three consecutive frames that have the same background, but differ in the position of one person. The macroblocks containing the background scene will match exactly, but the macroblocks containing the person will be offset in position by some unknown amount and will have to be tracked down.

Figure 7-49. Three consecutive frames.

The MPEG standards do not specify how to search, how far to search, or how good a match has to be in order to count. This is up to each implementation. For

example, an implementation might search for a macroblock at the current position in the previous frame, and all other positions offset $\pm\Delta x$ in the x direction and $\pm\Delta y$ in the y direction. For each position, the number of matches in the luminance matrix could be computed. The position with the highest score would be declared the winner, provided it was above some predefined threshold. Otherwise, the macroblock would be said to be missing. Much more sophisticated algorithms are also possible, of course.

If a macroblock is found, it is encoded by taking the difference between its current value and the one in the previous frame (for luminance and both chrominances). These difference matrices are then subjected to the discrete cosine transformation, quantization, run-length encoding, and Huffman encoding, as usual. The value for the macroblock in the output stream is then the motion vector (how far the macroblock moved from its previous position in each direction), followed by the encoding of its difference. If the macroblock is not located in the previous frame, the current value is encoded, just as in an I-frame.

Clearly, this algorithm is highly asymmetric. An implementation is free to try every plausible position in the previous frame if it wants to, in a desperate attempt to locate every last macroblock, no matter where it has moved to. This approach will minimize the encoded MPEG stream at the expense of very slow encoding. This approach might be fine for a one-time encoding of a film library but would be terrible for real-time videoconferencing.

Similarly, each implementation is free to decide what constitutes a "found" macroblock. This freedom allows implementers to compete on the quality and speed of their algorithms, but always produce compliant MPEG output.

So far, decoding MPEG is straightforward. Decoding I-frames is similar to decoding JPEG images. Decoding P-frames requires the decoder to buffer the previous frames so it can build up the new one in a separate buffer based on fully encoded macroblocks and macroblocks containing differences from the previous frames. The new frame is assembled macroblock by macroblock.

B-frames are similar to P-frames, except that they allow the reference macroblock to be in either previous frames or succeeding frames. This additional freedom allows for improved motion compensation. It is useful, for example, when objects pass in front of, or behind, other objects. To do B-frame encoding, the encoder needs to hold a sequence of frames in memory at once: past frames, the current frame being encoded, and future frames. Decoding is similarly more complicated and adds some delay. This is because a given B-frame cannot be decoded until the successive frames on which it depends are decoded. Thus, although B-frames give the best compression, they are not always used due to their greater complexity and buffering requirements.

The MPEG standards contain many enhancements to these techniques to achieve excellent levels of compression. AVC can be used to compress video at ratios in excess of 50:1, which reduces network bandwidth requirements by the same factor. For more information on AVC, see Sullivan and Wiegand (2005).

7.4.3 Streaming Stored Media

Let us now move on to network applications. Our first case is streaming media that is already stored in files. The most common example of this is watching videos over the Internet. This is one form of **VoD** (**Video on Demand**). Other forms of video on demand use a provider network that is separate from the Internet to deliver the movies (e.g., the cable network).

In the next section, we will look at streaming live media, for example, broadcast IPTV and Internet radio. Then we will look at the third case of real-time conferencing. An example is a voice-over-IP call or video conference with Skype. These three cases place increasingly stringent requirements on how we can deliver the audio and video over the network because we must pay increasing attention to delay and jitter.

The Internet is full of music and video sites that stream stored media files. Actually, the easiest way to handle stored media is *not* to stream it. Imagine you want to create an online movie rental site to compete with Apple's iTunes. A regular Web site will let users download and then watch videos (after they pay, of course). The sequence of steps is shown in Fig. 7-50. We will spell them out to contrast them with the next example.

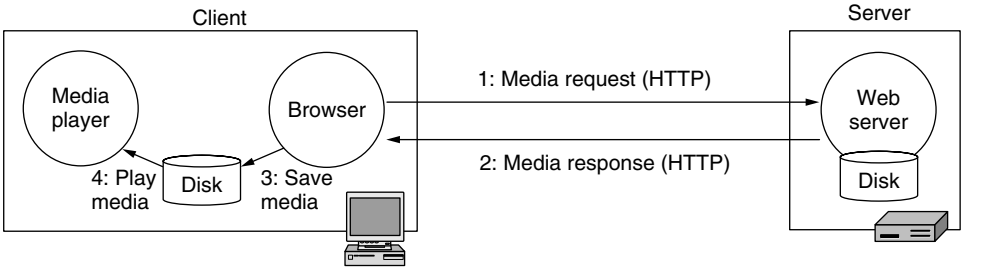


Figure 7-50. Playing media over the Web via simple downloads.

The browser goes into action when the user clicks on a movie. In step 1, it sends an HTTP request for the movie to the Web server to which the movie is linked. In step 2, the server fetches the movie (which is just a file in MP4 or some other format) and sends it back to the browser. Using the MIME type, for example, *video/mp4*, the browser looks up how it is supposed to display the file. In this case, it is with a media player that is shown as a helper application, though it could also be a plug-in. The browser saves the entire movie to a scratch file on disk in step 3. It then starts the media player, passing it the name of the scratch file. Finally, in step 4 the media player starts reading the file and playing the movie.

In principle, this approach is completely correct. It will play the movie. There is no real-time network issue to address either because the download is simply a

file download. The only trouble is that the entire video must be transmitted over the network before the movie starts. Most customers do not want to wait an hour for their ''video on demand.'' This model can be problematic even for audio. Imagine previewing a song before purchasing an album. If the song is 4 MB, which is a typical size for an MP3 song, and the broadband connectivity is 1 Mbps, the user will be greeted by half a minute of silence before the preview starts. This model is unlikely to sell many albums.

To get around this problem without changing how the browser works, sites can use the design shown in Fig. 7-51. The page linked to the movie is not the actual movie file. Instead, it is what is called a **metafile**, a very short file just naming the movie (and possibly having other key descriptors). A simple metafile might be only one line of ASCII text and look like this:

```
rtsp://joes-movie-server/movie-0025.mp4
```

The browser gets the page as usual, now a one-line file, in steps 1 and 2. Then it starts the media player and hands it the one-line file in step 3, all as usual. The media player reads the metafile and sees the URL of where to get the movie. It contacts *joes-video-server* and asks for the movie in step 4. The movie is then streamed back to the media player in step 5. The advantage of this arrangement is that the media player starts quickly, after only a very short metafile is downloaded. Once this happens, the browser is not in the loop any more. The media is sent directly to the media player, which can start showing the movie before the entire file has been downloaded.

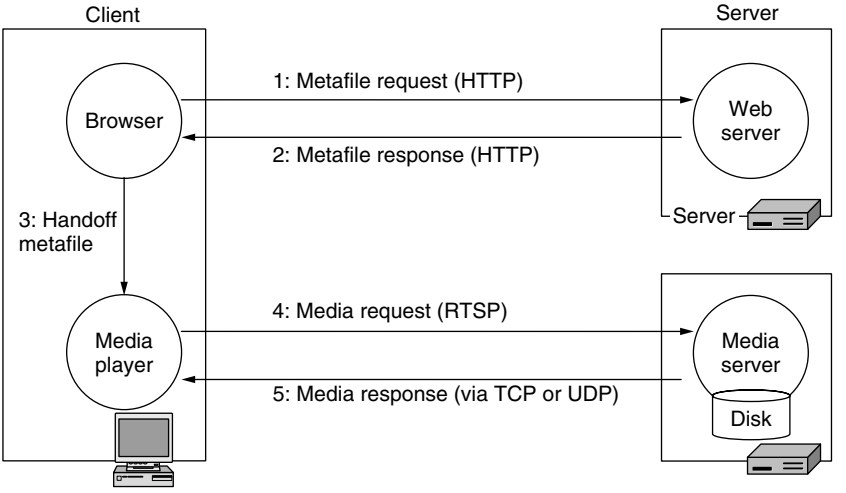


Figure 7-51. Streaming media using the Web and a media server.

We have shown two servers in Fig. 7-51 because the server named in the metafile is often not the same as the Web server. In fact, it is generally not even

an HTTP server, but a specialized media server. In this example, the media server uses **RTSP** (**Real Time Streaming Protocol**), as indicated by the scheme name *rtsp*.

The media player has four major jobs to do:

1. Manage the user interface.
2. Handle transmission errors.
3. Decompress the content.
4. Eliminate jitter.

Most media players nowadays have a glitzy user interface, sometimes simulating a stereo unit, with buttons, knobs, sliders, and visual displays. Often there are interchangeable front panels, called **skins**, that the user can drop onto the player. The media player has to manage all this and interact with the user.

The other jobs are related and depend on the network protocols. We will go through each one in turn, starting with handling transmission errors. Dealing with errors depends on whether a TCP-based transport like HTTP is used to transport the media, or a UDP-based transport like RTP is used. Both are used in practice. If a TCP-based transport is being used then there are no errors for the media player to correct because TCP already provides reliability by using retransmissions. This is an easy way to handle errors, at least for the media player, but it does complicate the removal of jitter in a later step.

Alternatively, a UDP-based transport like RTP can be used to move the data. We studied it in Chap. 6. With these protocols, there are no retransmissions. Thus, packet loss due to congestion or transmission errors will mean that some of the media does not arrive. It is up to the media player to deal with this problem.

Let us understand the difficulty we are up against. The loss is a problem because customers do not like large gaps in their songs or movies. However, it is not as much of a problem as loss in a regular file transfer because the loss of a small amount of media need not degrade the presentation for the user. For video, the user is unlikely to notice if there are occasionally 24 new frames in some second instead of 25 new frames. For audio, short gaps in the playout can be masked with sounds close in time. The user is unlikely to detect this substitution unless they are paying *very* close attention.

The key to the above reasoning, however, is that the gaps are very short. Network congestion or a transmission error will generally cause an entire packet to be lost, and packets are often lost in small bursts. Two strategies can be used to reduce the impact of packet loss on the media that is lost: FEC and interleaving. We will describe each in turn.

FEC (**Forward Error Correction**) is simply the error-correcting coding that we studied in Chap. 3 applied at the application level. Parity across packets provides an example (Shacham and McKenny, 1990). For every four data packets

that are sent, a fifth **parity packet** can be constructed and sent. This is shown in Fig. 7-52 with packets *A*, *B*, *C*, and *D*. The parity packet, *P*, contains redundant bits that are the parity or exclusive-OR sums of the bits in each of the four data packets. Hopefully, all of the packets will arrive for most groups of five packets. When this happens, the parity packet is simply discarded at the receiver. Or, if only the parity packet is lost, no harm is done.

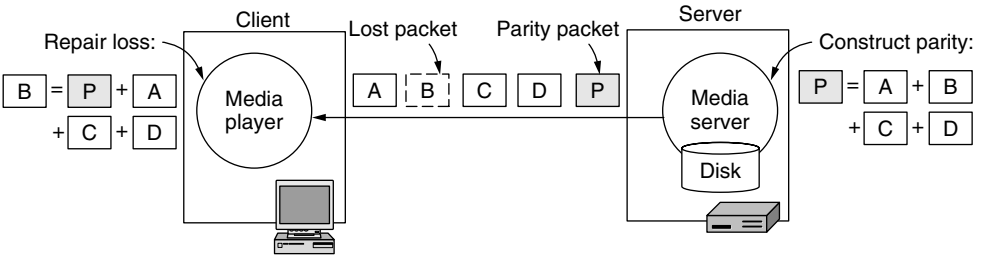


Figure 7-52. Using a parity packet to repair loss.

Occasionally, however, a data packet may be lost during transmission, as *B* is in Fig. 7-52. The media player receives only three data packets, *A*, *C*, and *D*, plus the parity packet, *P*. By design, the bits in the missing data packet can be reconstructed from the parity bits. To be specific, using "+" to represent exclusive-OR or modulo 2 addition, *B* can be reconstructed as $B = P + A + C + D$ by the properties of exclusive-OR (i.e., $X + Y + Y = X$).

FEC can reduce the level of loss seen by the media player by repairing some of the packet losses, but it only works up to a certain level. If two packets in a group of five are lost, there is nothing we can do to recover the data. The other property to note about FEC is the cost that we have paid to gain this protection. Every four packets have become five packets, so the bandwidth requirements for the media are 25% larger. The latency of decoding has increased too, as we may need to wait until the parity packet has arrived before we can reconstruct a data packet that came before it.

There is also one clever trick in the technique above. In Chap. 3, we described parity as providing error detection. Here we are providing error-correction. How can it do both? The answer is that in this case it is known which packet was lost. The lost data is called an **erasure**. In Chap. 3, when we considered a frame that was received with some bits in error, we did not know which bit was errored. This case is harder to deal with than erasures. Thus, with erasures parity can provide error correction, and without erasures parity can only provide error detection. We will see another unexpected benefit of parity soon, when we get to multicast scenarios.

The second strategy is called **interleaving**. This approach is based on mixing up or interleaving the order of the media before transmission and unmixing or

deinterleaving it on reception. That way, if a packet (or burst of packets) is lost, the loss will be spread out over time by the unmixing. It will not result in a single, large gap when the media is played out. For example, a packet might contain 220 stereo samples, each containing a pair of 16-bit numbers, normally good for 5 msec of music. If the samples were sent in order, a lost packet would represent a 5 msec gap in the music. Instead, the samples are transmitted as shown in Fig. 7-53. All the even samples for a 10-msec interval are sent in one packet, followed by all the odd samples in the next one. The loss of packet 3 now does not represent a 5-msec gap in the music, but the loss of every other sample for 10 msec. This loss can be handled easily by having the media player interpolate using the previous and succeeding samples. The result is lower temporal resolution for 10 msec, but not a noticeable time gap in the media.

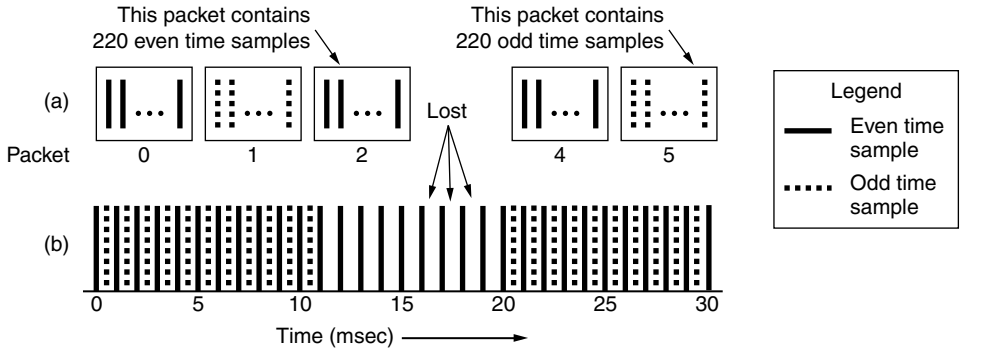


Figure 7-53. When packets carry alternate samples, the loss of a packet reduces the temporal resolution rather than creating a gap in time.

This interleaving scheme above only works with uncompressed sampling. However, interleaving (over short periods of time, not individual samples) can also be applied after compression as long as there is a way to find sample boundaries in the compressed stream. RFC 3119 gives a scheme that works with compressed audio.

Interleaving is an attractive technique when it can be used because it needs no additional bandwidth, unlike FEC. However, interleaving adds to the latency, just like FEC, because of the need to wait for a group of packets to arrive (so they can be de-interleaved).

The media player's third job is decompressing the content. Although this task is computationally intensive, it is fairly straightforward. The thorny issue is how to decode media if the network protocol does not correct transmission errors. In many compression schemes, later data cannot be decompressed until the earlier data has been decompressed, because the later data is encoded relative to the earlier data. For a UDP-based transport, there can be packet loss. Thus, the encoding

process must be designed to permit decoding despite packet loss. This requirement is why MPEG uses I-, P- and B-frames. Each I-frame can be decoded independently of the other frames to recover from the loss of any earlier frames.

The fourth job is to eliminate jitter, the bane of all real-time systems. The general solution that we described in Sec. 6.4.3 is to use a playout buffer. All streaming systems start by buffering 5–10 sec worth of media before starting to play, as shown in Fig. 7-54. Playing drains media regularly from the buffer so that the audio is clear and the video is smooth. The startup delay gives the buffer a chance to fill to the **low-water mark**. The idea is that data should now arrive regularly enough that the buffer is never completely emptied. If that were to happen, the media playout would stall. The value of buffering is that if the data are sometimes slow to arrive due to congestion, the buffered media will allow the playout to continue normally until new media arrive and the buffer is replenished.

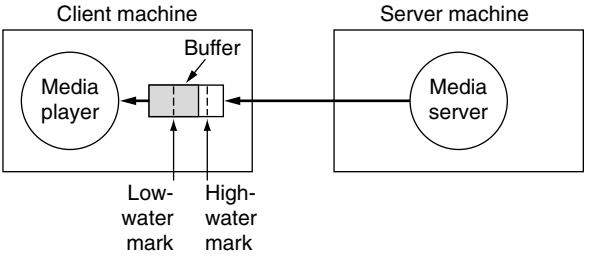


Figure 7-54. The media player buffers input from the media server and plays from the buffer rather than directly from the network.

How much buffering is needed, and how fast the media server sends media to fill up the buffer, depend on the network protocols. There are many possibilities. The largest factor in the design is whether a UDP-based transport or a TCP-based transport is used.

Suppose that a UDP-based transport like RTP is used. Further suppose that there is ample bandwidth to send packets from the media server to the media player with little loss, and little other traffic in the network. In this case, packets can be sent at the exact rate that the media is being played. Each packet will transit the network and, after a propagation delay, arrive at about the right time for the media player to present the media. Very little buffering is needed, as there is no variability in delay. If interleaving or FEC is used, more buffering is needed for at least the group of packets over which the interleaving or FEC is performed. However, this adds only a small amount of buffering.

Unfortunately, this scenario is unrealistic in two respects. First, bandwidth varies over network paths, so it is usually not clear to the media server whether there will be sufficient bandwidth before it tries to stream the media. A simple solution is to encode media at multiple resolutions and let each user choose a

resolution that is supported by his Internet connectivity. Often there are just two levels: high quality, say, encoded at 1.5 Mbps or better, and low quality, say encoded at 512 kbps or less.

Second, there will be some jitter, or variation in how long it takes media samples to cross the network. This jitter comes from two sources. There is often an appreciable amount of competing traffic in the network—some of which can come from multitasking users themselves browsing the Web while ostensibly watching a streamed movie). This traffic will cause fluctuations in when the media arrives. Moreover, we care about the arrival of video frames and audio samples, not packets. With compression, video frames in particular may be larger or smaller depending on their content. An action sequence will typically take more bits to encode than a placid landscape. If the network bandwidth is constant, the rate of media delivery versus time will vary. The more jitter, or variation in delay, from these sources, the larger the low-water mark of the buffer needs to be to avoid underrun.

Now suppose that a TCP-based transport like HTTP is used to send the media. By performing retransmissions and waiting to deliver packets until they are in order, TCP will increase the jitter that is observed by the media player, perhaps significantly. The result is that a larger buffer and higher low-water mark are needed. However, there is an advantage. TCP will send data as fast as the network will carry it. Sometimes media may be delayed if loss must be repaired. But much of the time, the network will be able to deliver media faster than the player consumes it. In these periods, the buffer will fill and prevent future underruns. If the network is significantly faster than the average media rate, as is often the case, the buffer will fill rapidly after startup such that emptying it will soon cease to be a concern.

With TCP, or with UDP and a transmission rate that exceeds the playout rate, a question is how far ahead of the playout point the media player and media server are willing to proceed. Often they are willing to download the entire file.

However, proceeding far ahead of the playout point performs work that is not yet needed, may require significant storage, and is not necessary to avoid buffer underruns. When it is not wanted, the solution is for the media player to define a **high-water mark** in the buffer. Basically, the server just pumps out data until the buffer is filled to the high-water mark. Then the media player tells it to pause. Since data will continue to pour in until the server has gotten the pause request, the distance between the high-water mark and the end of the buffer has to be greater than the bandwidth-delay product of the network. After the server has stopped, the buffer will begin to empty. When it hits the low-water mark, the media player tells the media server to start again. To avoid underrun, the low-water mark must also take the bandwidth-delay product of the network into account when asking the media server to resume sending the media.

To start and stop the flow of media, the media player needs a remote control for it. This is what RTSP provides. It is defined in RFC 2326 and provides the

mechanism for the player to control the server. As well as starting and stopping the stream, it can seek back or forward to a position, play specified intervals, and play at fast or slow speeds. It does not provide for the data stream, though, which is usually RTP over UDP or RTP over HTTP over TCP.

The main commands provided by RTSP are listed in Fig. 7-55. They have a simple text format, like HTTP messages, and are usually carried over TCP. RTSP can run over UDP too, since each command is acknowledged (and so can be resent if it is not acknowledged).

Command	Server action
DESCRIBE	List media parameters
SETUP	Establish a logical channel between the player and the server
PLAY	Start sending data to the client
RECORD	Start accepting data from the client
PAUSE	Temporarily stop sending data
TEARDOWN	Release the logical channel

Figure 7-55. RTSP commands from the player to the server.

Even though TCP would seem a poor fit to real-time traffic, it is often used in practice. The main reason is that it is able to pass through firewalls more easily than UDP, especially when run over the HTTP port. Most administrators configure firewalls to protect their networks from unwelcome visitors. They almost always allow TCP connections from remote port 80 to pass through for HTTP and Web traffic. Blocking that port quickly leads to unhappy campers. However, most other ports are blocked, including for RSTP and RTP, which use ports 554 and 5004, amongst others. Thus, the easiest way to get streaming media through the firewall is for the Web site to pretend it is an HTTP server sending a regular HTTP response, at least to the firewall.

There are some other advantages of TCP, too. Because it provides reliability, TCP gives the client a complete copy of the media. This makes it easy for a user to rewind to a previously viewed playout point without concern for lost data. Finally, TCP will buffer as much of the media as possible as quickly as possible. When buffer space is cheap (which it is when the disk is used for storage), the media player can download the media while the user watches. Once the download is complete, the user can watch uninterrupted, even if he loses connectivity. This property is helpful for mobiles because connectivity can change rapidly with motion.

The disadvantage of TCP is the added startup latency (because of TCP startup) and also a higher low-water mark. However, this is rarely much of a penalty as long as the network bandwidth exceeds the media rate by a large factor.

7.4.4 Streaming Live Media

It is not only recorded videos that are tremendously popular on the Web. Live media streaming is very popular too. Once it became possible to stream audio and video over the Internet, commercial radio and TV stations got the idea of broadcasting their content over the Internet as well as over the air. Not so long after that, college stations started putting their signals out over the Internet. Then college *students* started their own Internet broadcasts.

Today, people and companies of all sizes stream live audio and video. The area is a hotbed of innovation as the technologies and standards evolve. Live streaming is used for an online presence by major television stations. This is called **IPTV** (**IP TeleVision**). It is also used to broadcast radio stations like the BBC. This is called **Internet radio**. Both IPTV and Internet radio reach audiences worldwide for events ranging from fashion shows to World Cup soccer and test matches live from the Melbourne Cricket Ground. Live streaming over IP is used as a technology by cable providers to build their own broadcast systems. And it is widely used by low-budget operations from adult sites to zoos. With current technology, virtually anyone can start live streaming quickly and with little expense.

One approach to live streaming is to record programs to disk. Viewers can connect to the server's archives, pull up any program, and download it for listening. A **podcast** is an episode retrieved in this manner. For scheduled events, it is also possible to store content just after it is broadcast live, so the archive is only running, say, half an hour or less behind the live feed.

In fact, this approach is exactly the same as that used for the streaming media we just discussed. It is easy to do, all the techniques we have discussed work for it, and viewers can pick and choose among all the programs in the archive.

A different approach is to broadcast live over the Internet. Viewers tune in to an ongoing media stream, just like turning on the television. However, media players provide the added features of letting the user pause or rewind the playout. The live media will continue to be streamed and will be buffered by the player until the user is ready for it. From the browser's point of view, it looks exactly like the case of streaming stored media. It does not matter to the player whether the content comes from a file or is being sent live, and usually the player will not be able to tell (except that it is not possible to skip forward with a live stream). Given the similarity of mechanism, much of our previous discussion applies, but there are also some key differences.

Importantly, there is still the need for buffering at the client side to smooth out jitter. In fact, a larger amount of buffering is often needed for live streaming (independent of the consideration that the user may pause playback). When streaming from a file, the media can be pushed out at a rate that is greater than the playback rate. This will build up a buffer quickly to compensate for network jitter (and the player will stop the stream if it does not want to buffer more data). In contrast, live media streaming is always transmitted at precisely the rate it is

generated, which is the same as the rate at which it is played back. It cannot be sent faster than this. As a consequence, the buffer must be large enough to handle the full range of network jitter. In practice, a 10–15 second startup delay is usually adequate, so this is not a large problem.

The other important difference is that live streaming events usually have hundreds or thousands of simultaneous viewers of the same content. Under these circumstances, the natural solution for live streaming is to use multicasting. This is not the case for streaming stored media because the users typically stream different content at any given time. Streaming to many users then consists of many individual streaming sessions that happen to occur at the same time.

A multicast streaming scheme works as follows. The server sends each media packet once using IP multicast to a group address. The network delivers a copy of the packet to each member of the group. All of the clients who want to receive the stream have joined the group. The clients do this using IGMP, rather than sending an RTSP message to the media server. This is because the media server is already sending the live stream (except before the first user joins). What is needed is to arrange for the stream to be received locally.

Since multicast is a one-to-many delivery service, the media is carried in RTP packets over a UDP transport. TCP only operates between a single sender and a single receiver. Since UDP does not provide reliability, some packets may be lost. To reduce the level of media loss to an acceptable level, we can use FEC and interleaving, as before.

In the case of FEC, there is a beneficial interaction with multicast that is shown in the parity example of Fig. 7-56. When the packets are multicast, different clients may lose different packets. For example, client 1 has lost packet *B*, client 2 lost the parity packet *P*, client 3 lost *D*, and client 4 did not lose any packets. However, even though three different packets are lost across the clients, each client can recover all of the data packets in this example. All that is required is that each client lose no more than one packet, whichever one it may be, so that the missing packet can be recovered by a parity computation. Nonnenmacher et al. (1997) describe how this idea can be used to boost reliability.

For a server with a large number of clients, multicast of media in RTP and UDP packets is clearly the most efficient way to operate. Otherwise, the server must transmit *N* streams when it has *N* clients, which will require a very large amount of network bandwidth at the server for large streaming events.

It may surprise you to learn that the Internet does not work like this in practice. What usually happens is that each user establishes a separate TCP connection to the server, and the media is streamed over that connection. To the client, this is the same as streaming stored media. And as with streaming stored media, there are several reasons for this seemingly poor choice.

The first reason is that IP multicast is not broadly available on the Internet. Some ISPs and networks support it internally, but it is usually not available across network boundaries as is needed for wide-area streaming. The other reasons are

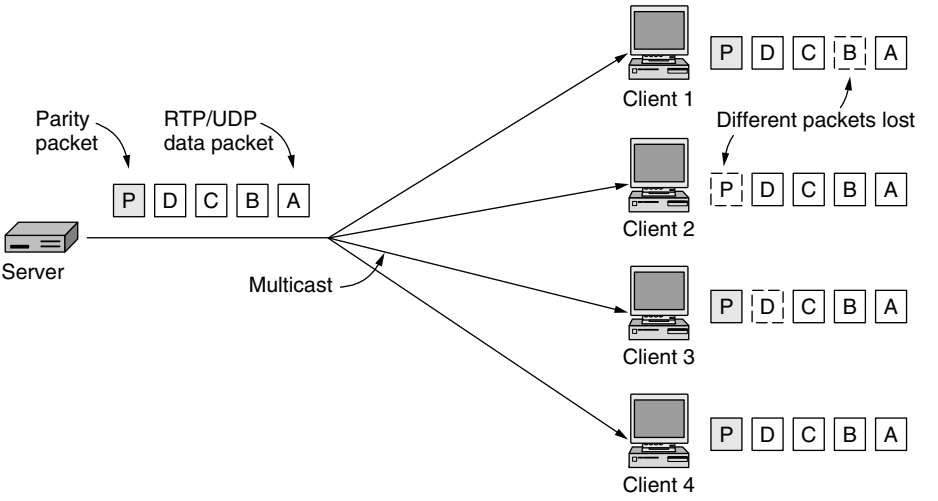


Figure 7-56. Multicast streaming media with a parity packet.

the same advantages of TCP over UDP as discussed earlier. Streaming with TCP will reach nearly all clients on the Internet, particularly when disguised as HTTP to pass through firewalls, and reliable media delivery allows users to rewind easily.

There is one important case in which UDP and multicast can be used for streaming, however: within a provider network. For example, a cable company might decide to broadcast TV channels to customer set-top boxes using IP technology instead of traditional video broadcasts. The use of IP to distribute broadcast video is broadly called IPTV, as discussed above. Since the cable company has complete control of its own network, it can engineer it to support IP multicast and have sufficient bandwidth for UDP-based distribution. All of this is invisible to the customer, as the IP technology exists within the **walled garden** of the provider. It looks just like cable TV in terms of service, but it is IP underneath, with the set-top box being a computer running UDP and the TV set being simply a monitor attached to the computer.

Back to the Internet case, the disadvantage of live streaming over TCP is that the server must send a separate copy of the media for each client. This is feasible for a moderate number of clients, especially for audio. The trick is to place the server at a location with good Internet connectivity so that there is sufficient bandwidth. Usually this means renting a server in a data center from a hosting provider, not using a server at home with only broadband Internet connectivity. There is a very competitive hosting market, so this need not be expensive.

In fact, it is easy for anybody, even a student, to set up and operate a streaming media server such as an Internet radio station. The main components of this

station are illustrated in Fig. 7-57. The basis of the station is an ordinary PC with a decent sound card and microphone. Popular software is used to capture audio and encode it in various formats, for example, MP4, and media players are used to listen to the audio as usual.

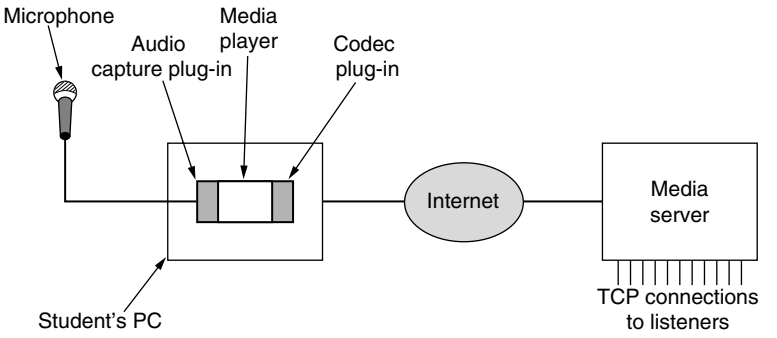


Figure 7-57. A student radio station.

The audio stream captured on the PC is then fed over the Internet to a media server with good network connectivity, either as podcasts for stored file streaming or for live streaming. The server handles the task of distributing the media via large numbers of TCP connections. It also presents a front-end Web site with pages about the station and links to the content that is available for streaming. There are commercial software packages for managing all the pieces, as well as open source packages such as icecast.

However, for a very large number of clients, it becomes infeasible to use TCP to send media to each client from a single server. There is simply not enough bandwidth to the one server. For large streaming sites, the streaming is done using a set of servers that are geographically spread out, so that a client can connect to the nearest server. This is a content distribution network that we will study at the end of the chapter.

7.4.5 Real-Time Conferencing

Once upon a time, voice calls were carried over the public switched telephone network, and network traffic was primarily voice traffic, with a little bit of data traffic here and there. Then came the Internet, and the Web. The data traffic grew and grew, until by 1999 there was as much data traffic as voice traffic (since voice is now digitized, both can be measured in bits). By 2002, the volume of data traffic was an order of magnitude more than the volume of voice traffic and still growing exponentially, with voice traffic staying almost flat.

The consequence of this growth has been to flip the telephone network on its head. Voice traffic is now carried using Internet technologies, and represents only

a tiny fraction of the network bandwidth. This disruptive technology is known as **voice over IP**, and also as **Internet telephony**.

Voice-over-IP is used in several forms that are driven by strong economic factors. (English translation: it saves money so people use it.) One form is to have what look like regular (old-fashioned?) telephones that plug into the Ethernet and send calls over the network. Pehr Anderson was an undergraduate student at M.I.T. when he and his friends prototyped this design for a class project. They got a "B" grade. Not content, he started a company called NBX in 1996, pioneered this kind of voice over IP, and sold it to 3Com for $90 million three years later. Companies love this approach because it lets them do away with separate telephone lines and make do with the networks that they have already.

Another approach is to use IP technology to build a long-distance telephone network. In countries such as the U.S., this network can be accessed for competitive long-distance service by dialing a special prefix. Voice samples are put into packets that are injected into the network and pulled out of the packets when they leave it. Since IP equipment is much cheaper than telecommunications equipment this leads to cheaper services.

As an aside, the difference in price is not entirely technical. For many decades, telephone service was a regulated monopoly that guaranteed the phone companies a fixed percentage profit over their costs. Not surprisingly, this led them to run up costs, for example, by having lots and lots of redundant hardware, justified in the name of better reliability (the telephone system was only allowed to be down for a total of 2 hours every 40 years, or 3 min/year on average). This effect was often referred to as the "gold-plated telephone pole syndrome." Since deregulation, the effect has decreased, of course, but legacy equipment still exists. The IT industry never had any history operating like this, so it has always been lean and mean.

However, we will concentrate on the form of voice over IP that is likely the most visible to users: using one computer to call another computer. This form became commonplace as PCs began shipping with microphones, speakers, cameras, and CPUs fast enough to process media, and people started connecting to the Internet from home at broadband rates. A well-known example is the Skype software that was released starting in 2003. Skype and other companies also provide gateways to make it easy to call regular telephone numbers as well as computers with IP addresses.

As network bandwidth increased, video calls joined voice calls. Initially, video calls were in the domain of companies. Videoconferencing systems were designed to exchange video between two or more locations enabling executives at different locations to see each other while they held their meetings. However, with good broadband Internet connectivity and video compression software, home users can also videoconference. Tools such as Skype that started as audio-only now routinely include video with the calls so that friends and family across the world can see as well as hear each other.

From our point of view, Internet voice or video calls are also a media streaming problem, but one that is much more constrained than streaming a stored file or a live event. The added constraint is the low latency that is needed for a two-way conversation. The telephone network allows a one-way latency of up to 150 msec for acceptable usage, after which delay begins to be perceived as annoying by the participants. (International calls may have a latency of up to 400 msec, by which point they are far from a positive user experience.)

This low latency is difficult to achieve. Certainly, buffering 5–10 seconds of media is not going to work (as it would for broadcasting a live sports event). Instead, video and voice-over-IP systems must be engineered with a variety of techniques to minimize latency. This goal means starting with UDP as the clear choice rather than TCP, because TCP retransmissions introduce at least one round-trip worth of delay. Some forms of latency cannot be reduced, however, even with UDP. For example, the distance between Seattle and Amsterdam is close to 8,000 km. The speed-of-light propagation delay for this distance in optical fiber is 40 msec. Good luck beating that. In practice, the propagation delay through the network will be longer because it will cover a larger distance (the bits do not follow a great circle route) and have transmission delays as each IP router stores and forwards a packet. This fixed delay eats into the acceptable delay budget.

Another source of latency is related to packet size. Normally, large packets are the best way to use network bandwidth because they are more efficient. However, at an audio sampling rate of 64 kbps, a 1-KB packet would take 125 msec to fill (and even longer if the samples are compressed). This delay would consume most of the overall delay budget. In addition, if the 1-KB packet is sent over a broadband access link that runs at just 1 Mbps, it will take 8 msec to transmit. Then add another 8 msec for the packet to go over the broadband link at the other end. Clearly, large packets will not work.

Instead, voice-over-IP systems use short packets to reduce latency at the cost of bandwidth efficiency. They batch audio samples in smaller units, commonly 20 msec. At 64 kbps, this is 160 bytes of data, less with compression. However, by definition the delay from this packetization will be 20 msec. The transmission delay will be smaller as well because the packet is shorter. In our example, it would reduce to around 1 msec. By using short packets, the minimum one-way delay for a Seattle-to-Amsterdam packet has been reduced from an unacceptable 181 msec (40 + 125 + 16) to an acceptable 62 msec (40 + 20 + 2).

We have not even talked about the software overhead, but it, too, will eat up some of the delay budget. This is especially true for video, since compression is usually needed to fit video into the available bandwidth. Unlike streaming from a stored file, there is no time to have a computationally intensive encoder for high levels of compression. The encoder and the decoder must both run quickly.

Buffering is still needed to play out the media samples on time (to avoid unintelligible audio or jerky video), but the amount of buffering must be kept very small since the time remaining in our delay budget is measured in milliseconds.

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**.

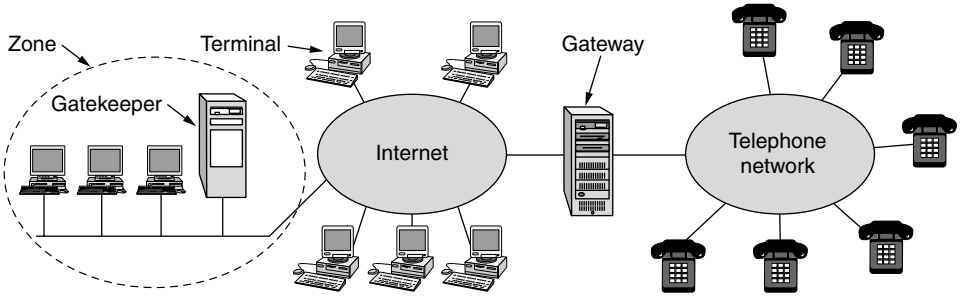


Figure 7-58. The H.323 architectural model for Internet telephony.

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 tradeoffs 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.

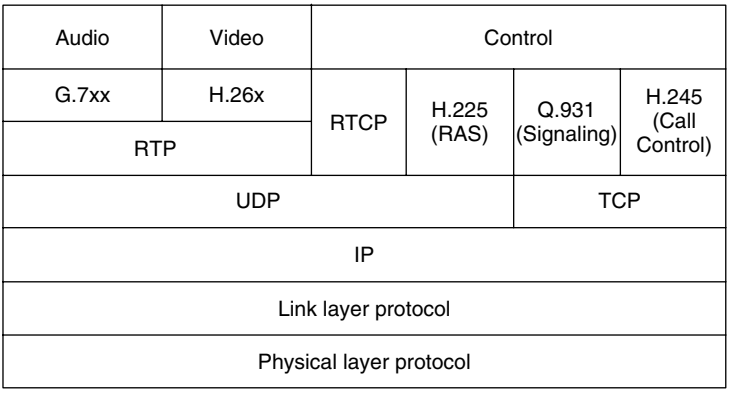


Figure 7-59. The H.323 protocol stack.

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.

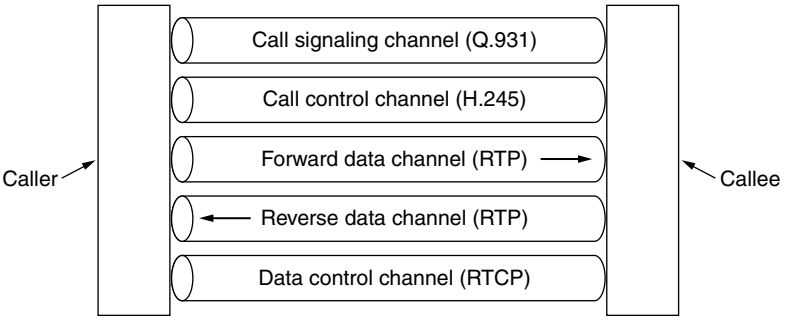


Figure 7-60. Logical channels between the caller and callee during a call.

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.

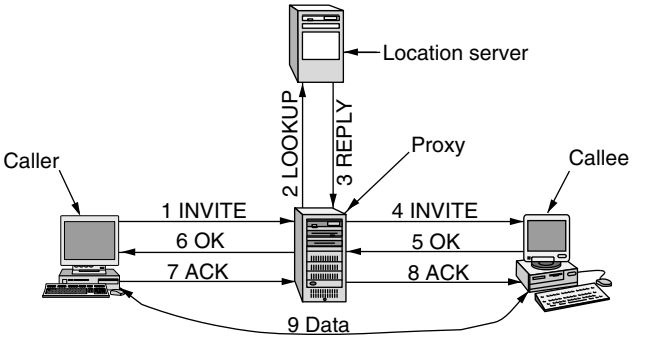


Figure 7-62. Use of a proxy server and redirection with SIP.

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.

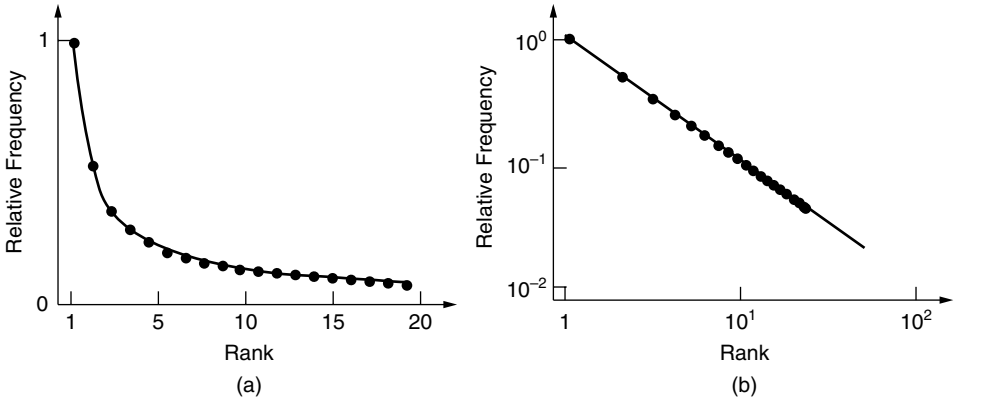


Figure 7-64. Zipf distribution (a) On a linear scale. (b) On a log-log scale.

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.

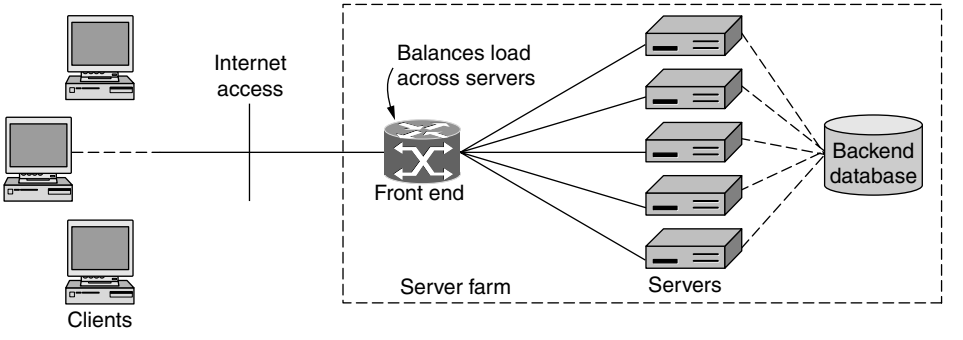


Figure 7-65. A server farm.

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.

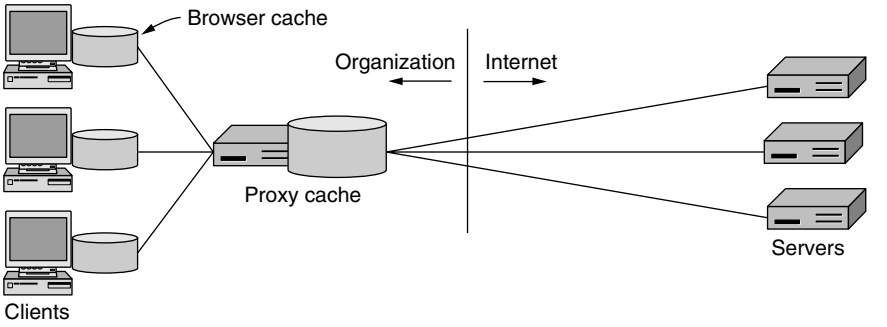


Figure 7-66. A proxy cache between Web browsers and Web servers.

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.

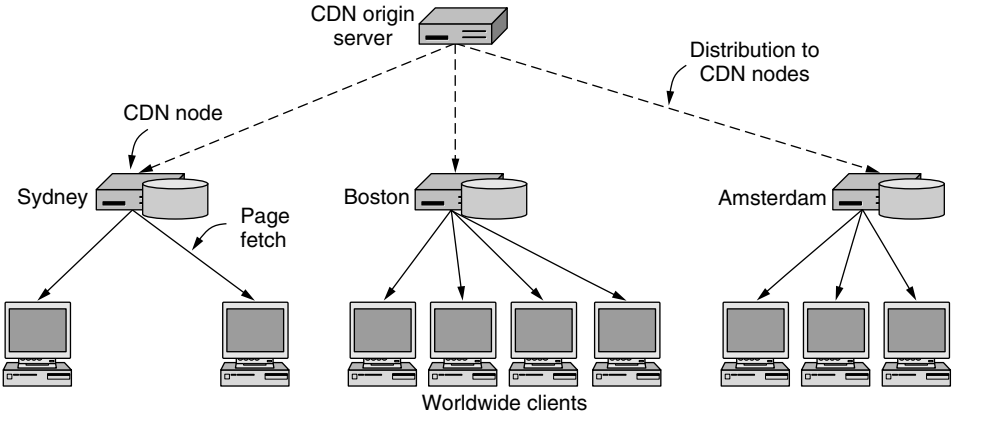


Figure 7-67. CDN distribution tree.

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.

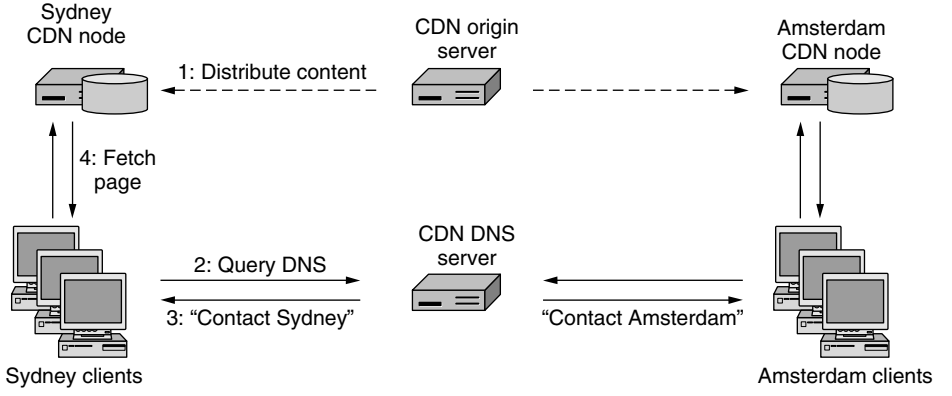


Figure 7-68. Directing clients to nearby CDN nodes using DNS.

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.

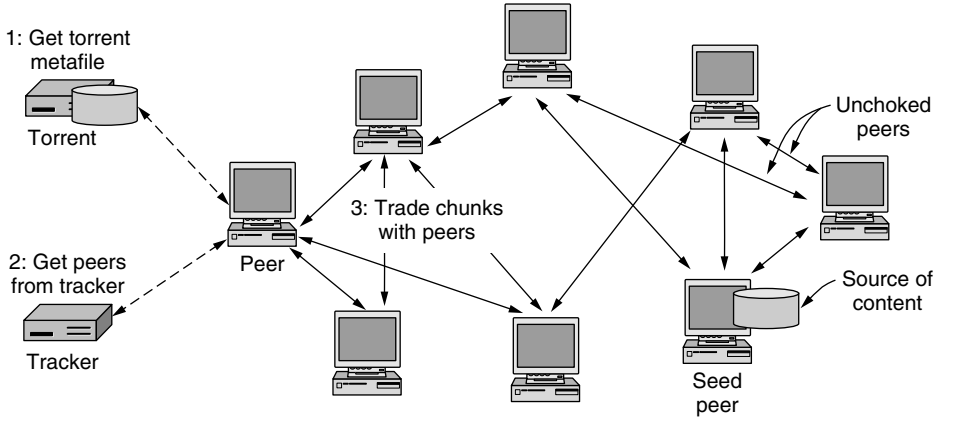


Figure 7-70. BitTorrent.

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

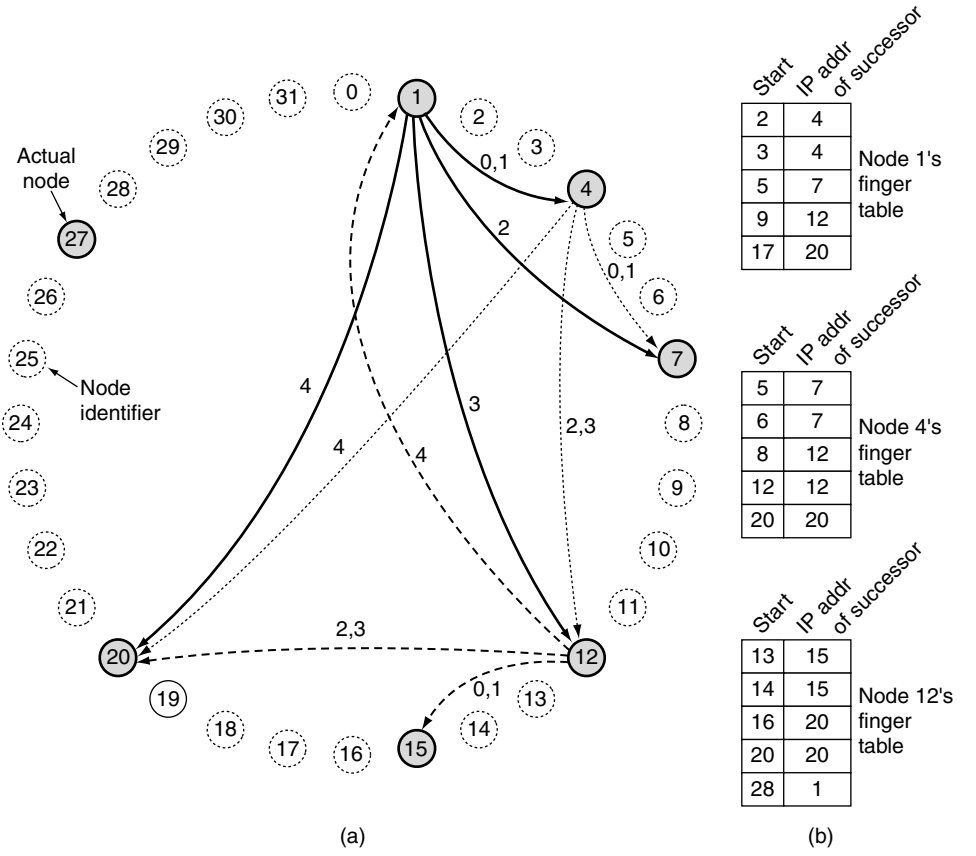


Figure 7-71. (a) A set of 32 node identifiers arranged in a circle. The shaded ones correspond to actual machines. The arcs show the fingers from nodes 1, 4, and 12. The labels on the arcs are the table indices. (b) Examples of the finger tables.

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.

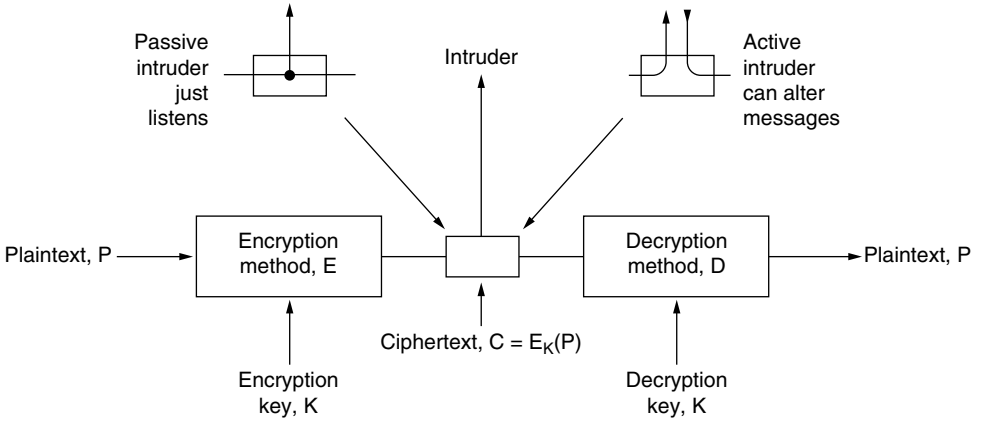


Figure 8-2. The encryption model (for a symmetric-key cipher).

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

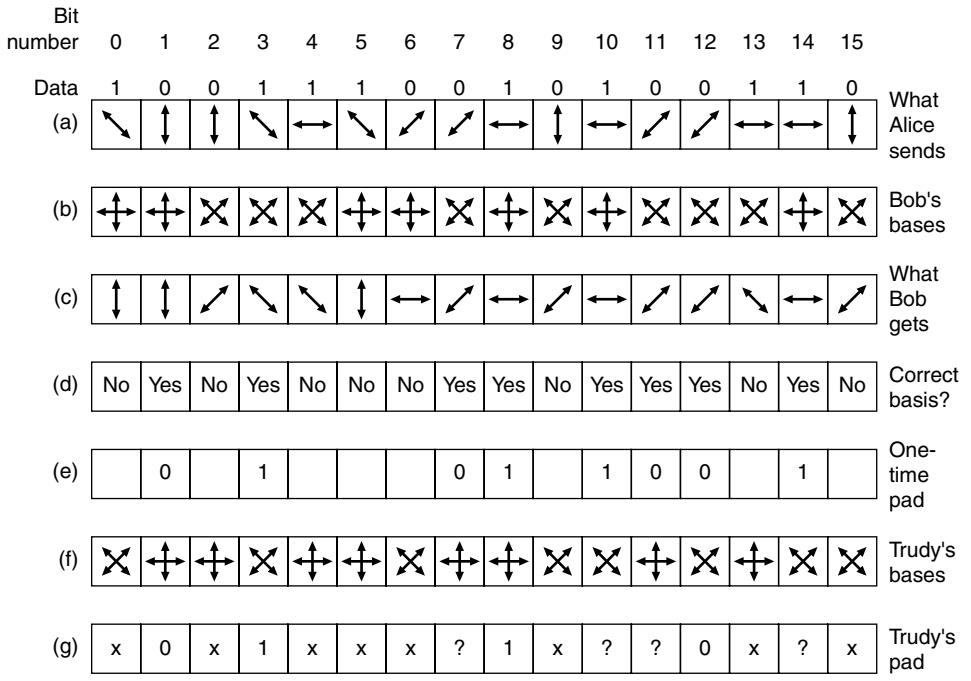


Figure 8-5. An example of quantum cryptography.

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.