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.

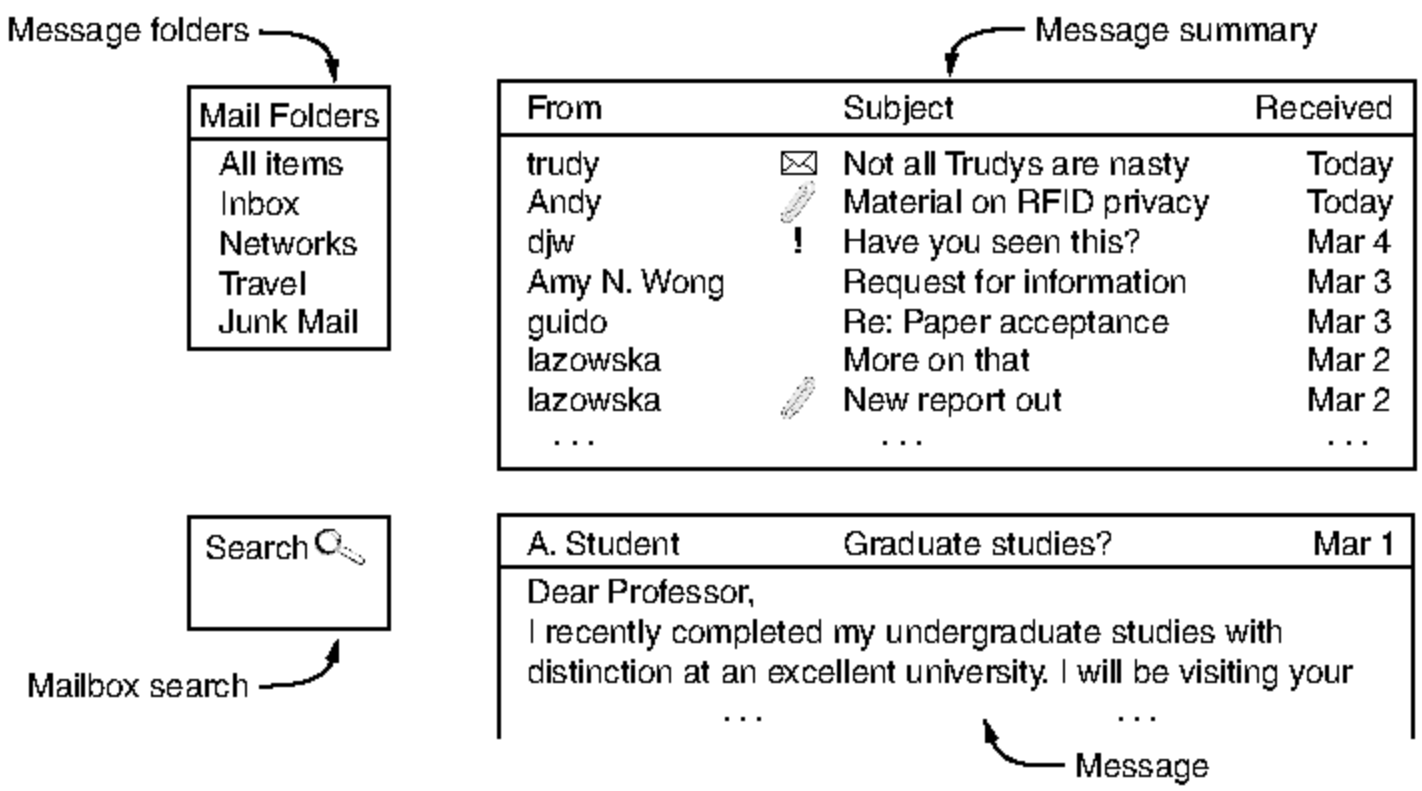


Figure 7-9. Typical elements of the user agent interface.

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.

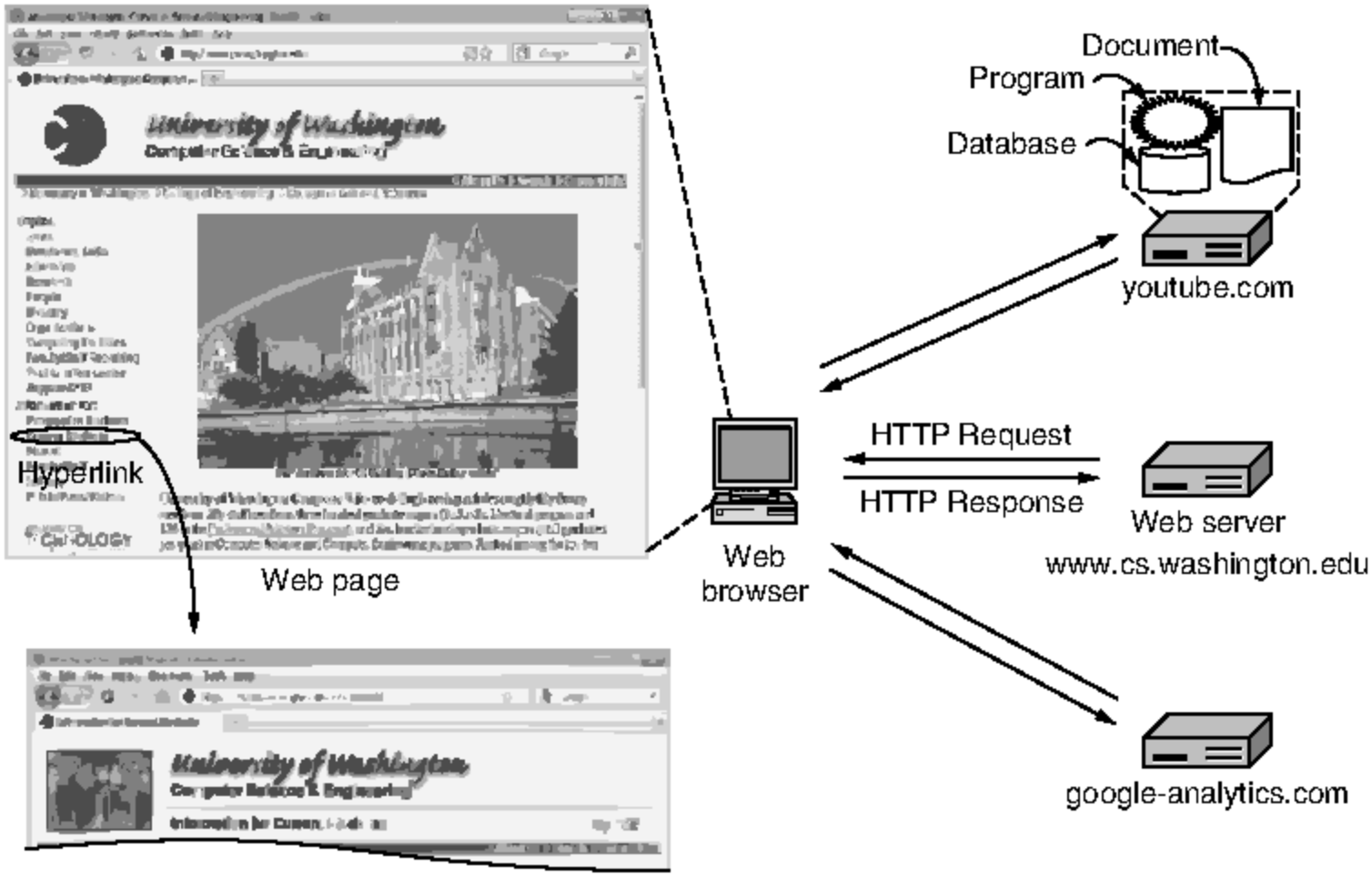


Figure 7-18. Architecture of the Web.

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 **URNs** (**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.

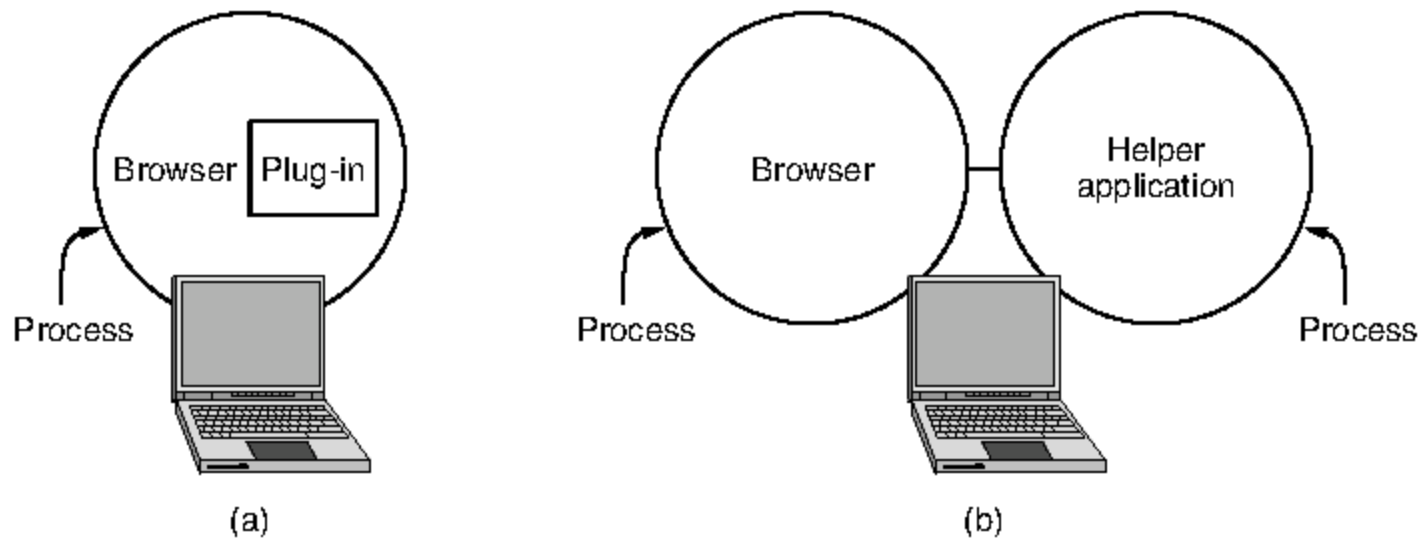


Figure 7-20. (a) A browser plug-in. (b) A helper application.

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.

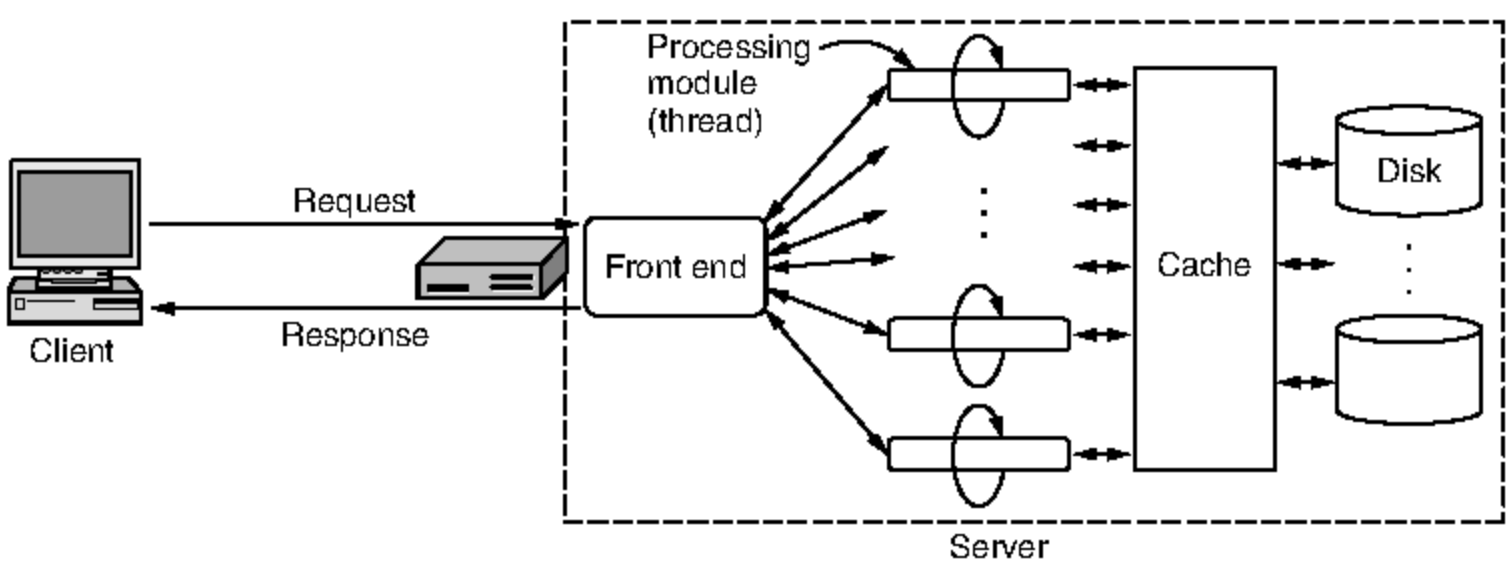


Figure 7-21. A multithreaded Web server with a front end and processing modules.

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.

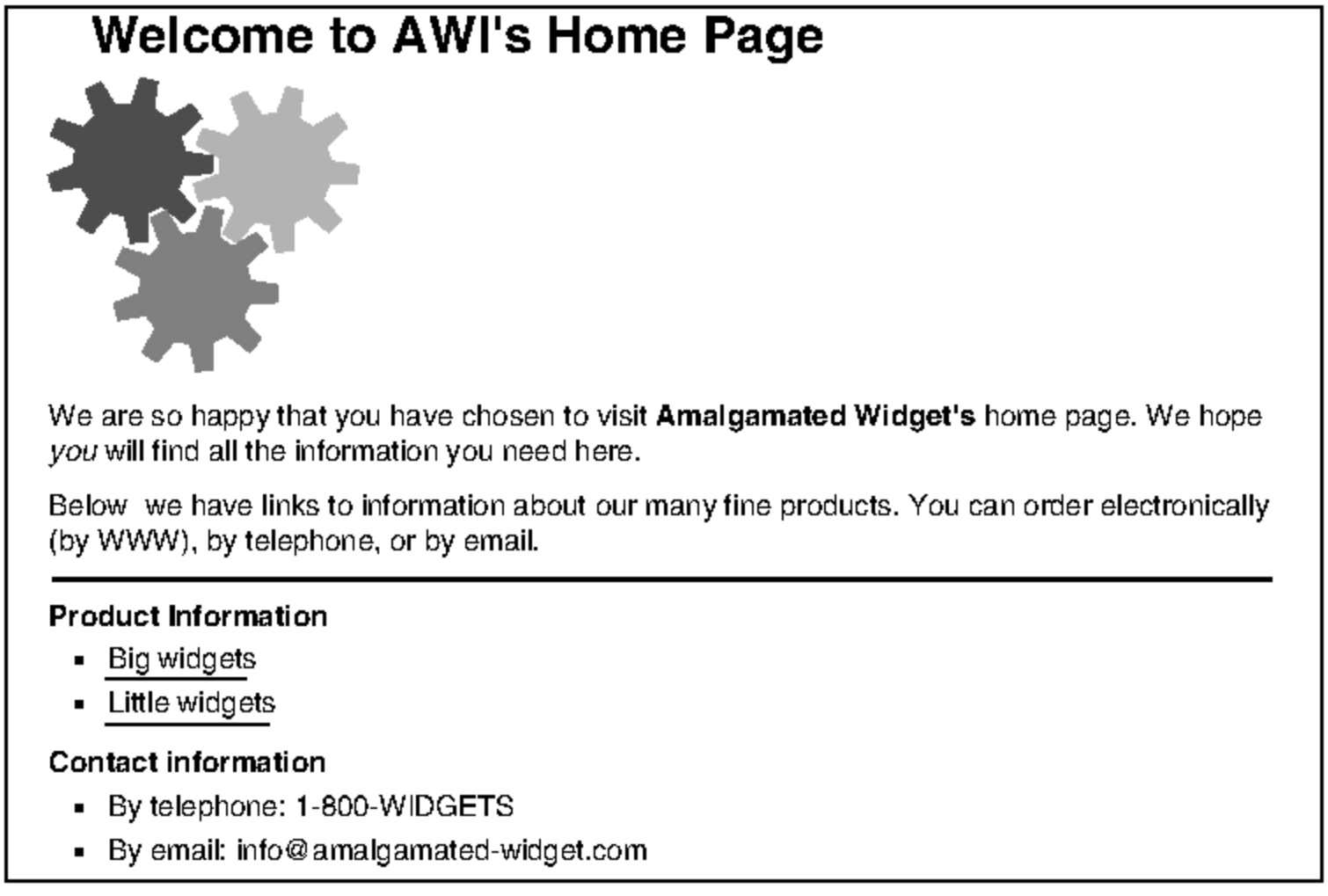
The tags <b> and <i> are used to enter boldface and italics mode, respectively. The <hr> tag forces a break and draws a horizontal line across the display.

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*.

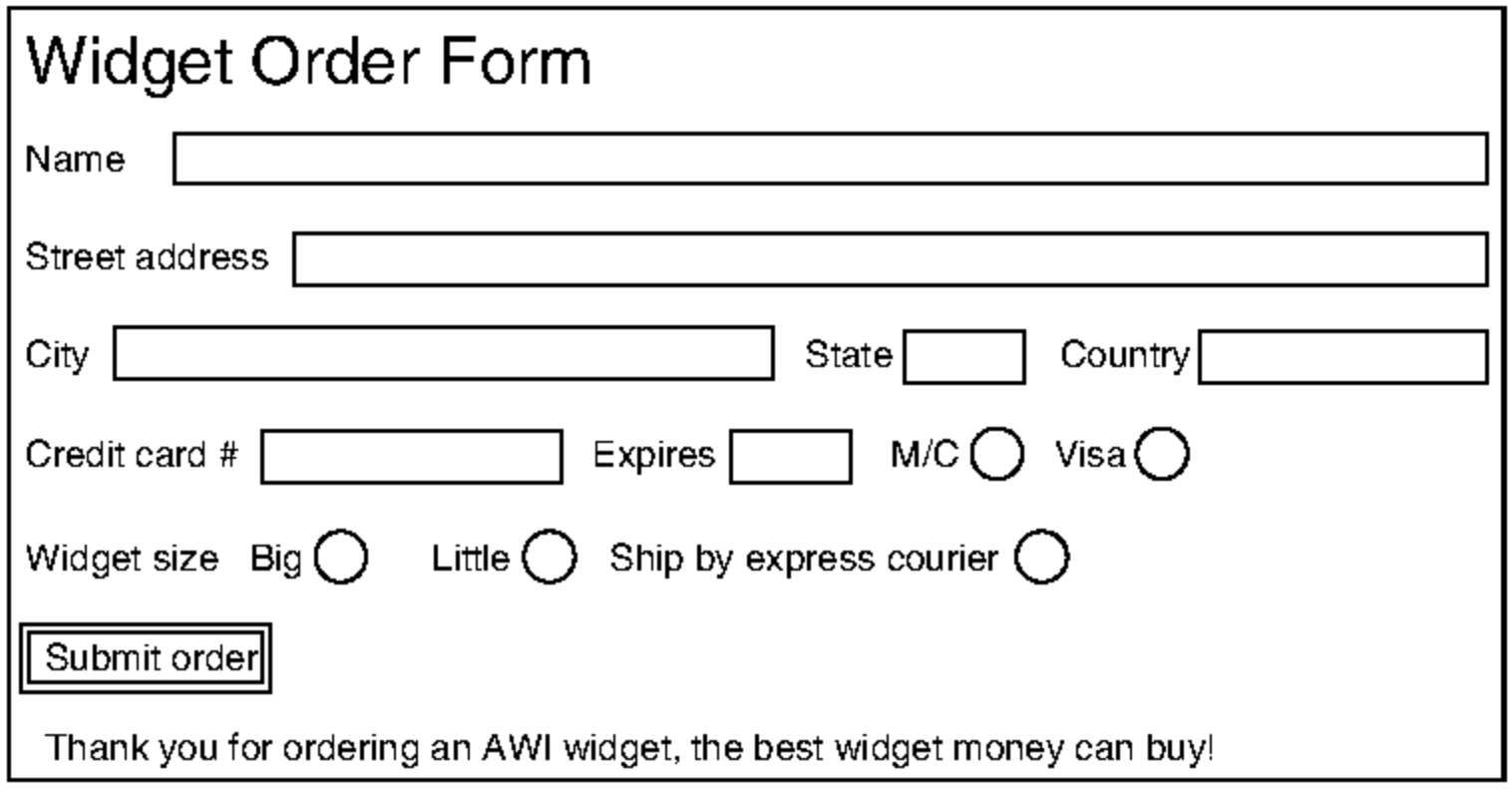
The next line of the form asks for the user's street address, 40 characters wide. Then comes a line asking for the city, state, and country. Since no <p> tags are used between these fields, the browser displays them all on one line (instead of as separate paragraphs) if they will fit. As far as the browser 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).

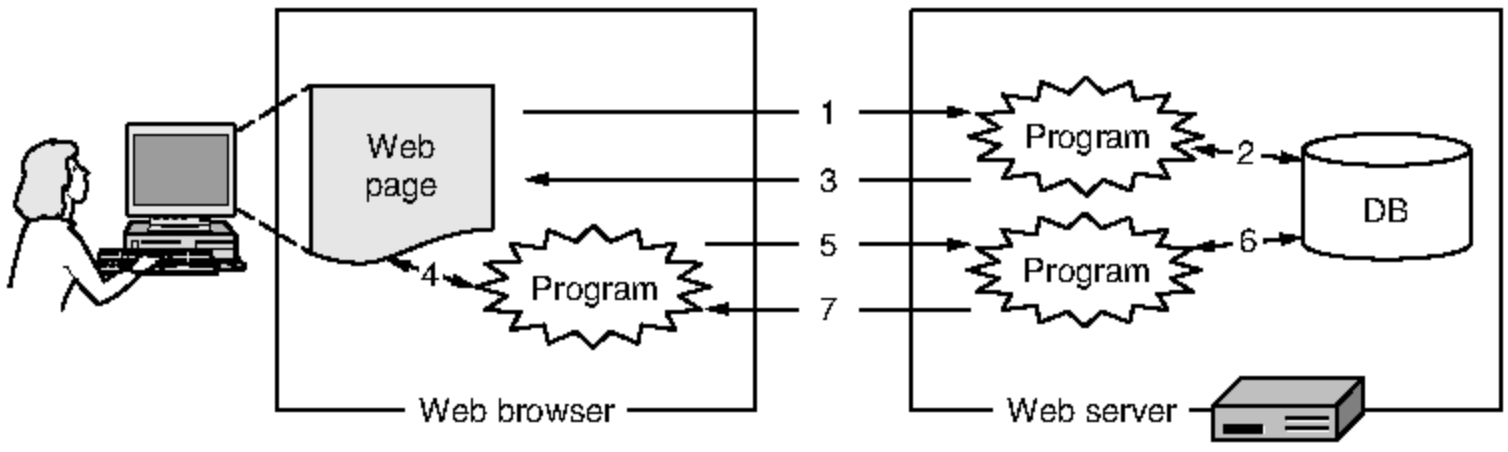


Figure 7-29. Dynamic pages.

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