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

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

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

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

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

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

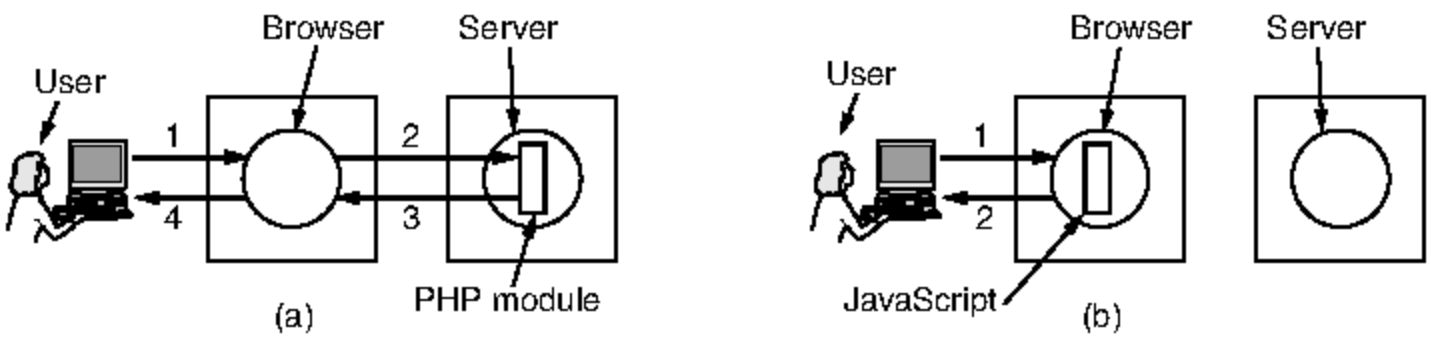

Figure 7-32. (a) Server-side scripting with PHP. (b) Client-side scripting with JavaScript.

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

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

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

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

AJAX—Asynchronous JavaScript and XML

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

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

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

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

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

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

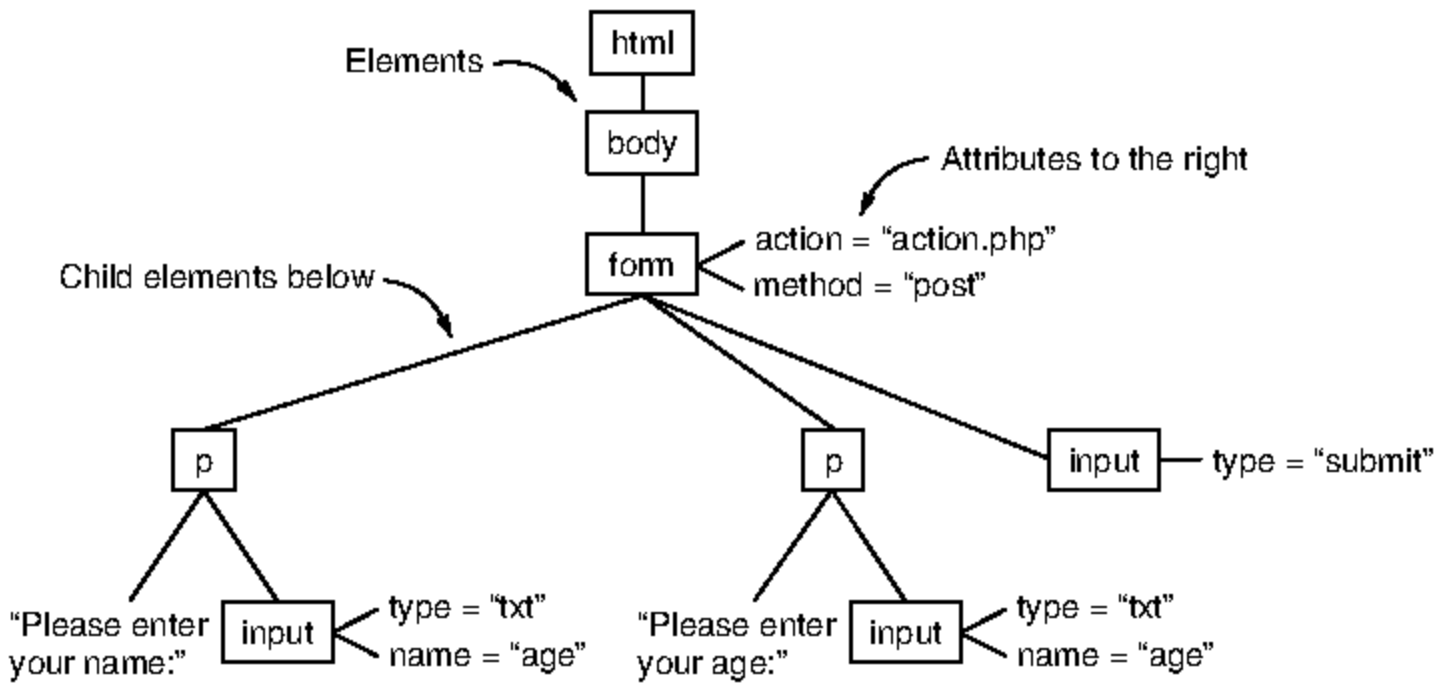

Figure 7-33. The DOM tree for the HTML in Fig. 7-30(a).

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

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

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

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

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

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

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

<book_list>

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

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

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

</book_list>
```

Figure 7-34. A simple XML document.

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

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

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

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

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

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

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

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

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

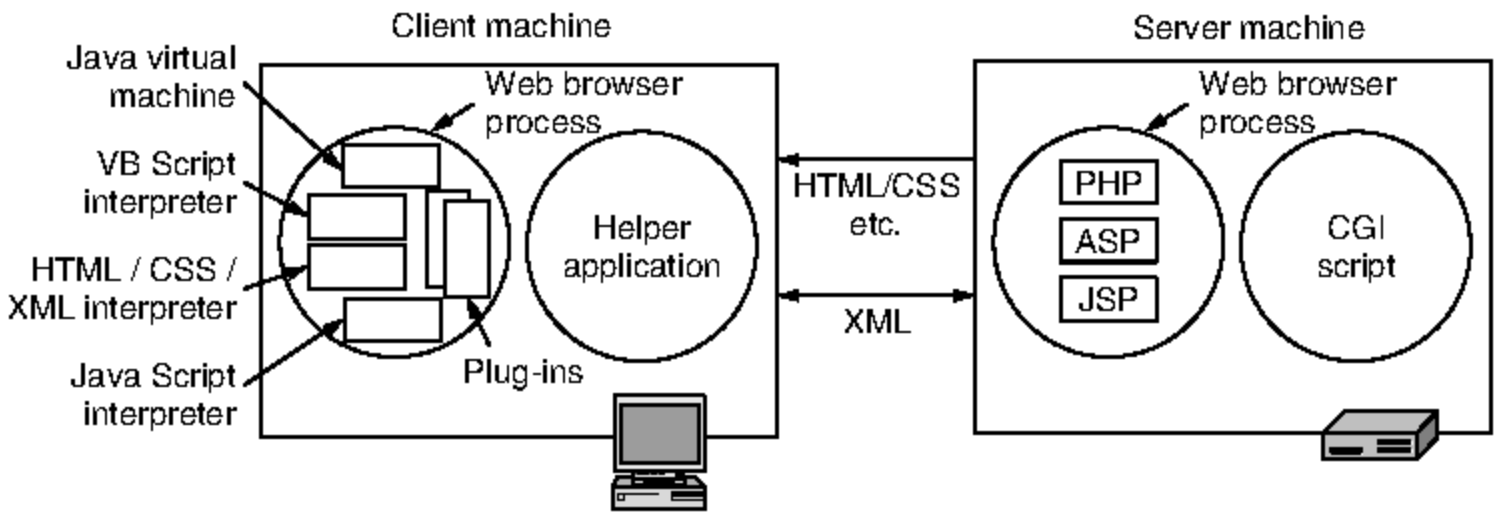

Figure 7-35. Various technologies used to generate dynamic pages.

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

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

7.3.4 HTTP—The HyperText Transfer Protocol

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

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

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

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

Connections

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

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

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

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

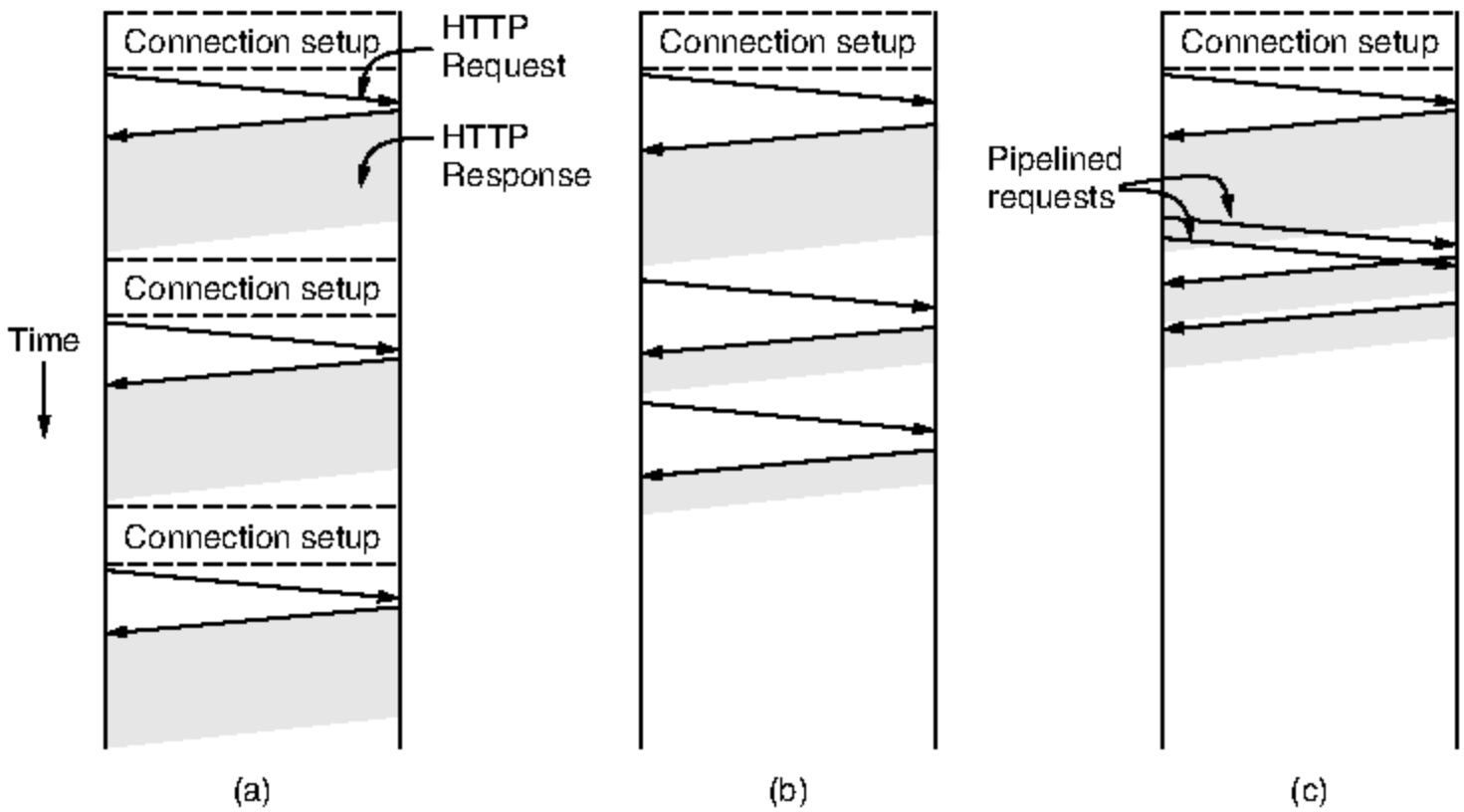

Figure 7-36. HTTP with (a) multiple connections and sequential requests. (b) A persistent connection and sequential requests. (c) A persistent connection and pipelined requests.

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

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

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

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

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

Methods

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

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

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

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

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

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

```
GET filename HTTP/1.1
```

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

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

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

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

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

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

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

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

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

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

Figure 7-38. The status code response groups.

Message Headers

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

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

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

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

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

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

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

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

Figure 7-39. Some HTTP message headers.

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

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

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

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

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

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

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

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

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

Caching

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

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

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

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

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

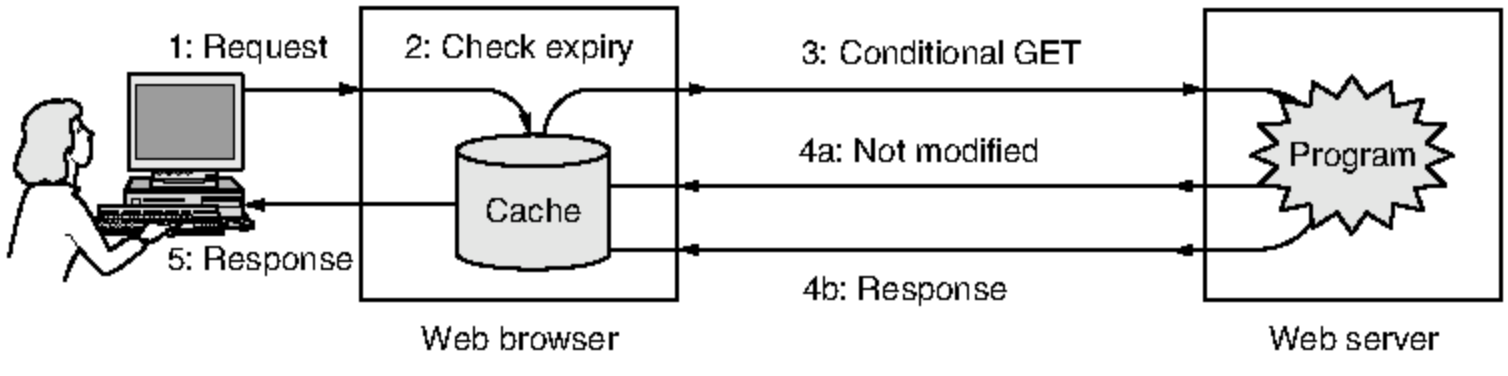

Figure 7-40. HTTP caching.

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

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

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

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

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

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

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

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

Experimenting with HTTP

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

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

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

7.3.5 The Mobile Web

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

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

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

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

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

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

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

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

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

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

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

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

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

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

7.3.6 Web Search

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

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

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

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

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

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

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

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

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

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

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

7.4 STREAMING AUDIO AND VIDEO

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

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

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

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

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

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

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

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

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

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

7.4.1 Digital Audio

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

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

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

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

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

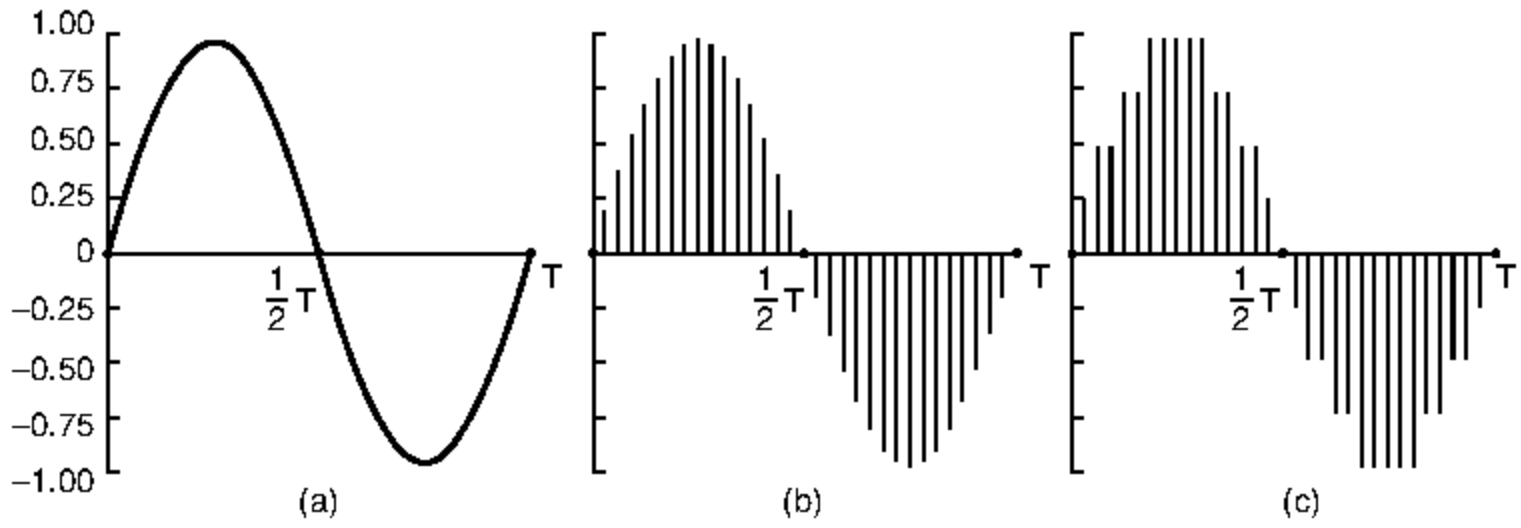

Figure 7-42. (a) A sine wave. (b) Sampling the sine wave. (c) Quantizing the samples to 4 bits.

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

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

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

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

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

Audio Compression

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

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

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

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

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

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

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

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

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

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

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

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

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

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

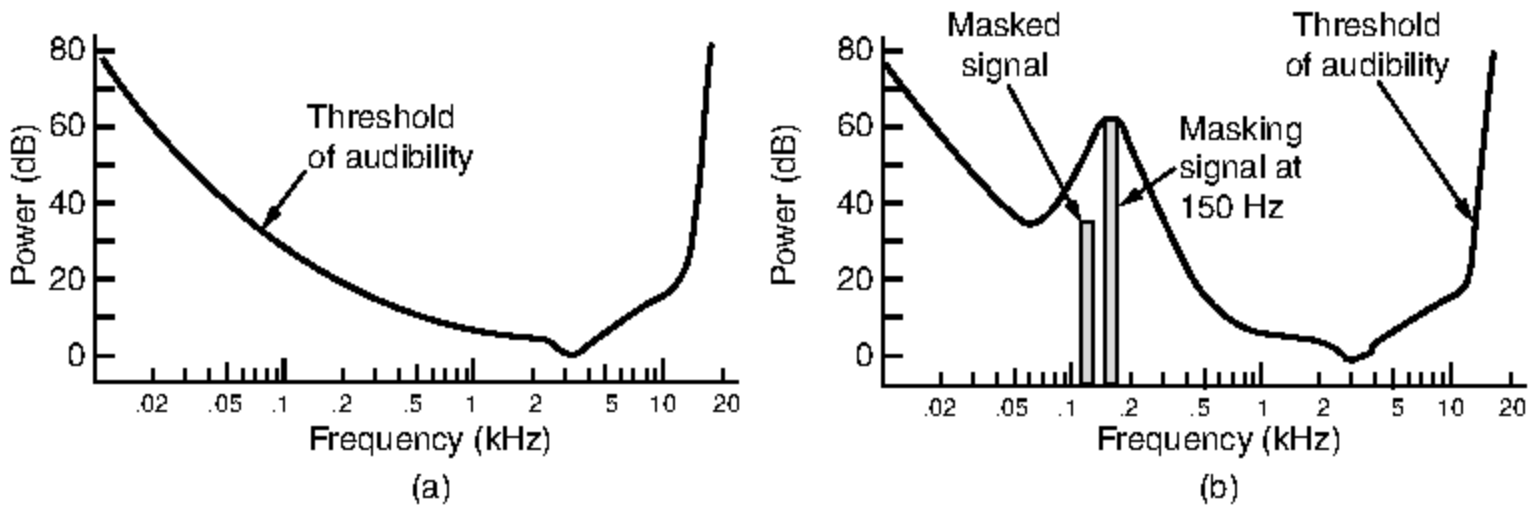

Figure 7-43. (a) The threshold of audibility as a function of frequency. (b) The masking effect.

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

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

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

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

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

7.4.2 Digital Video

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

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

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

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

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

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

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

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

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

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

Video Compression

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

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

The JPEG Standard

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

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

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

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

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

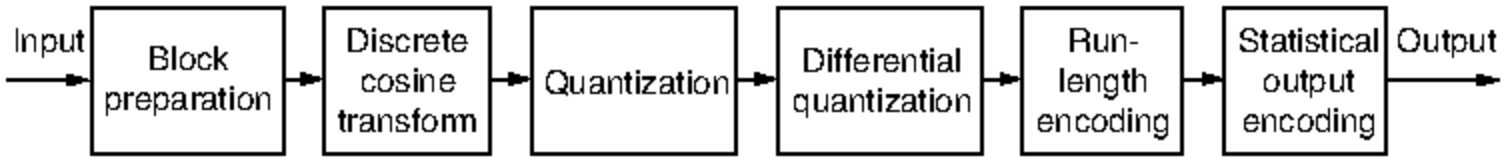

Figure 7-44. Steps in JPEG lossy sequential encoding.

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

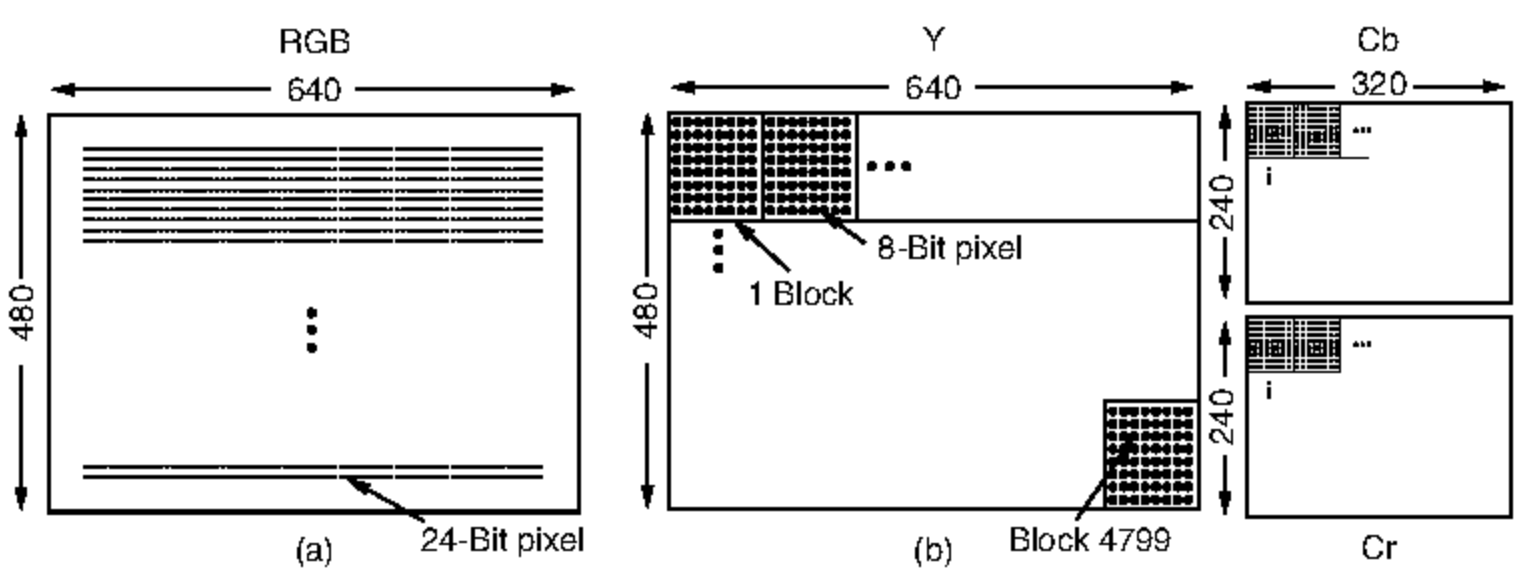

Figure 7-45. (a) RGB input data. (b) After block preparation.

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

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

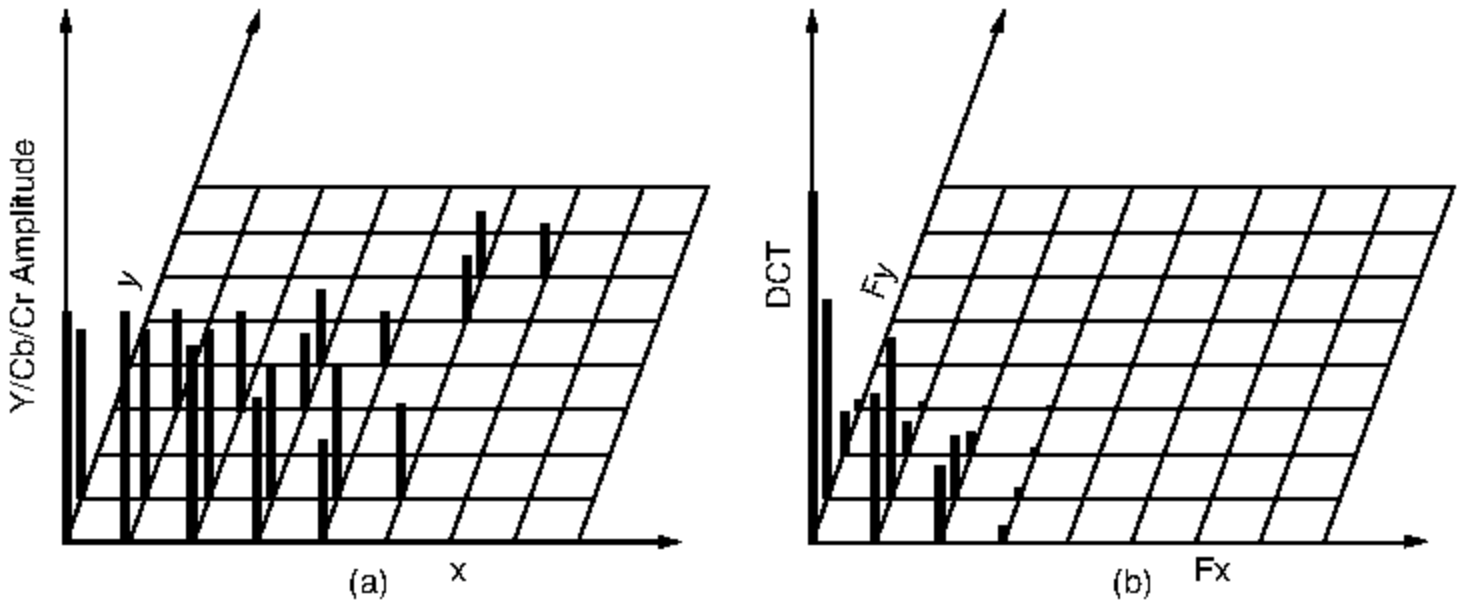

Figure 7-46. (a) One block of the Y matrix. (b) The DCT coefficients.

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

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

DCT coefficients

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

Quantization table

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

Quantized coefficients

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

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

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

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

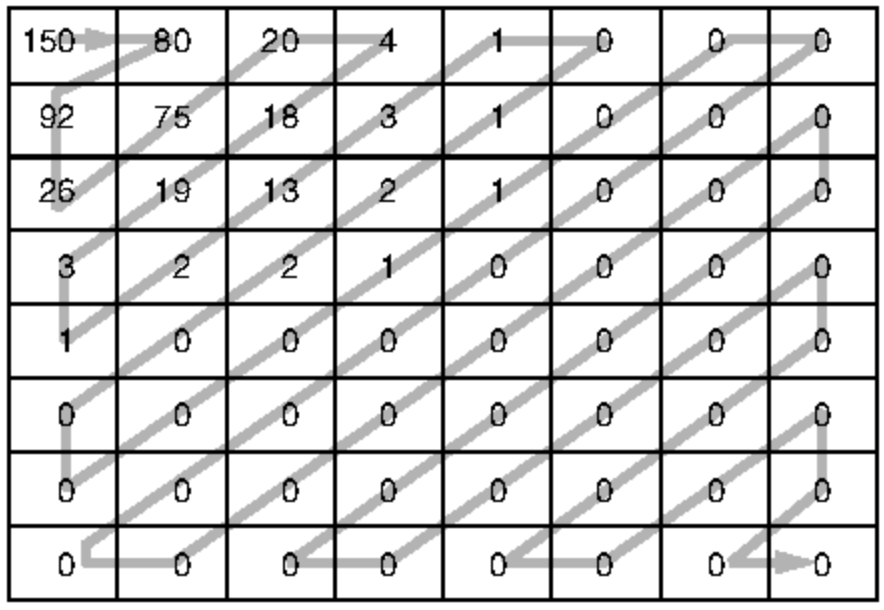

Figure 7-48. The order in which the quantized values are transmitted.

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

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

The MPEG Standard

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

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

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

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

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

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

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

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

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

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

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

MPEG output consists of three kinds of frames:

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

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

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

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

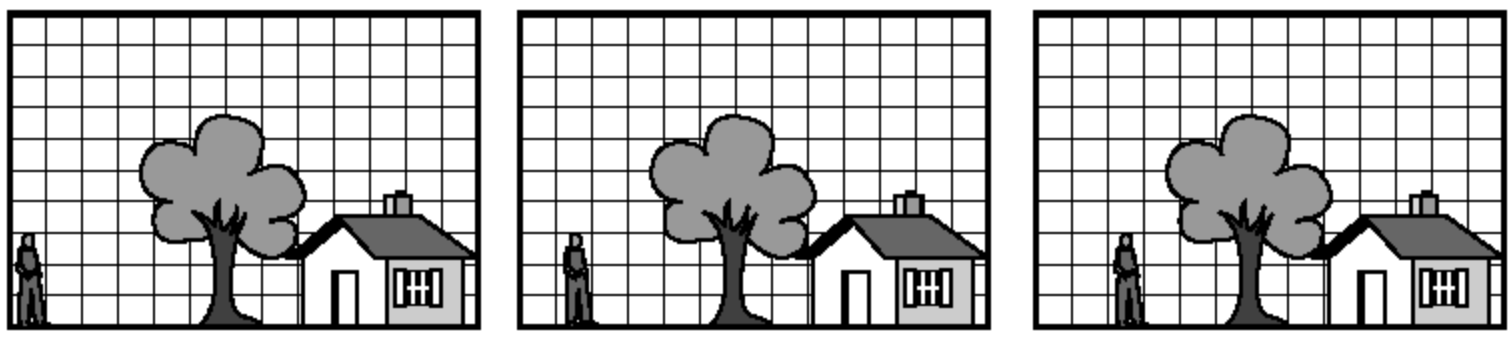

Figure 7-49. Three consecutive frames.

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

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

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

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

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

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

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

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

7.4.3 Streaming Stored Media

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

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

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

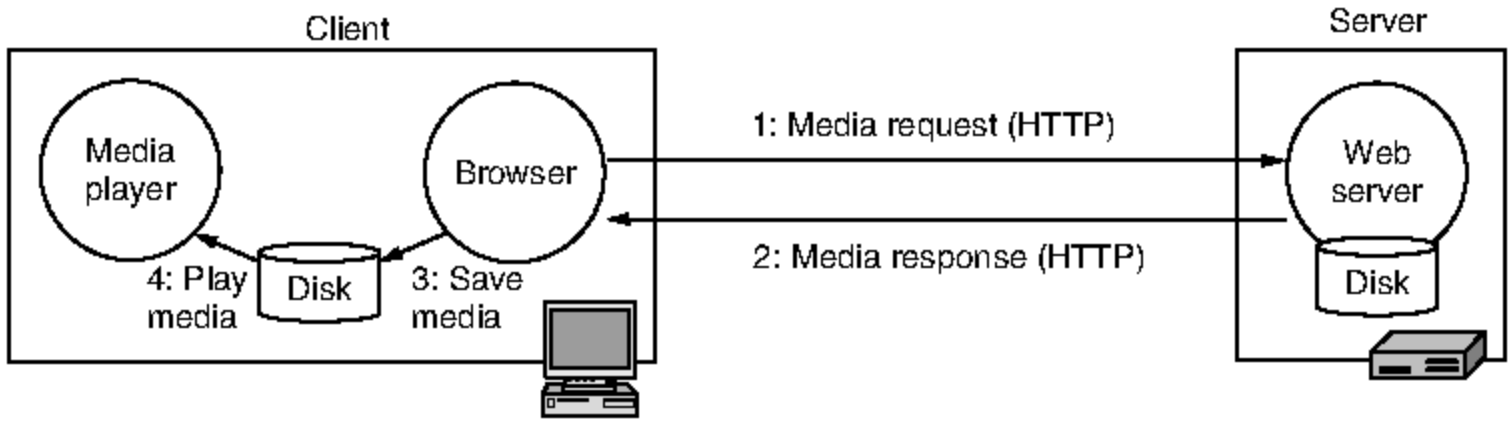

Figure 7-50. Playing media over the Web via simple downloads.

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

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

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

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

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

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

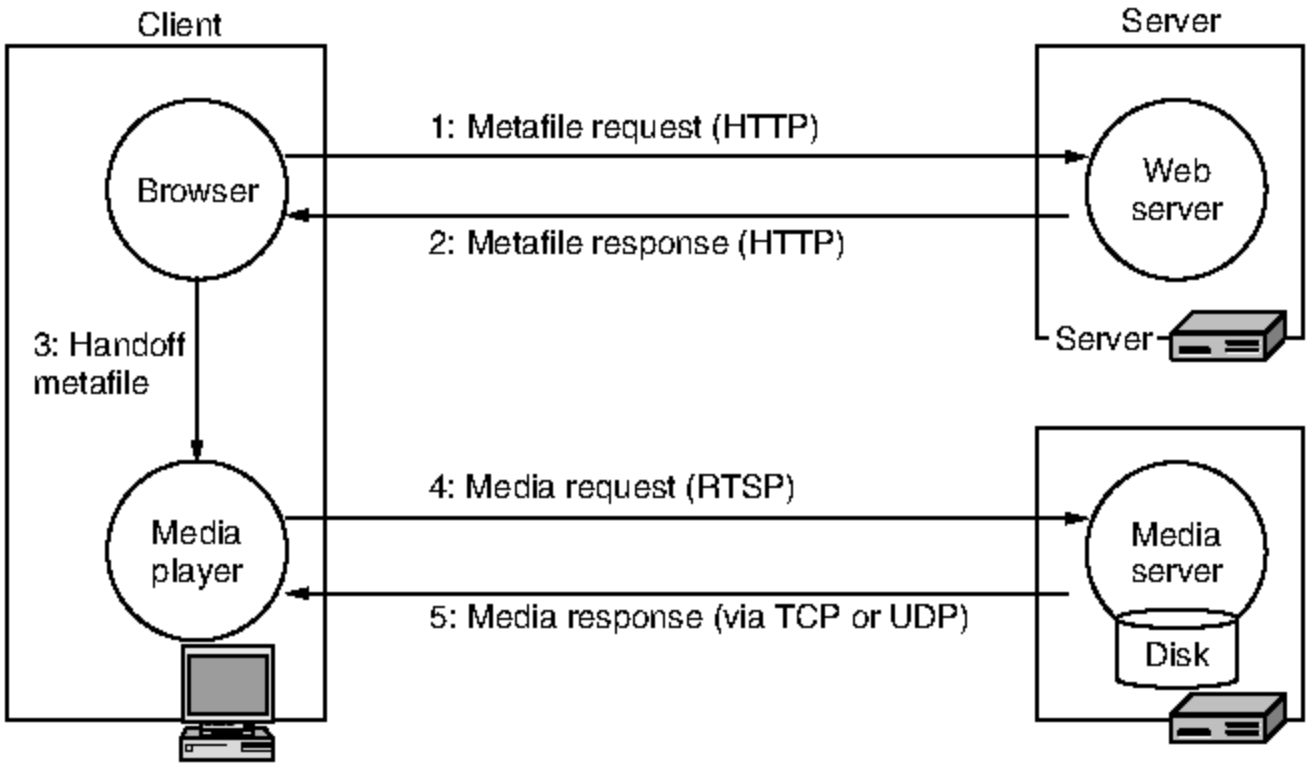

Figure 7-51. Streaming media using the Web and a media server.

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

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

The media player has four major jobs to do:

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

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

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

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

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

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

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

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

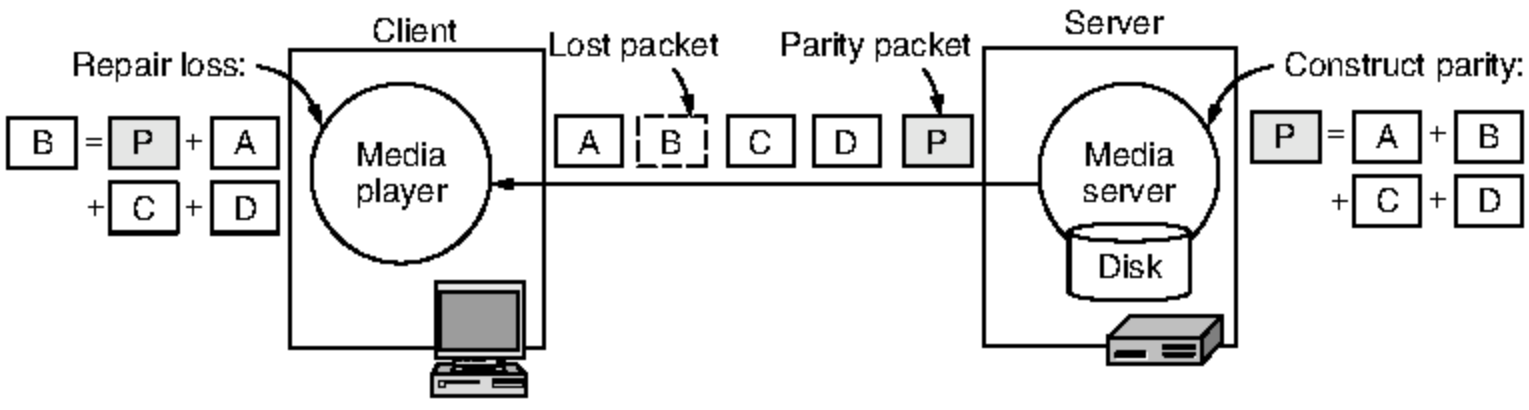

Figure 7-52. Using a parity packet to repair loss.

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

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

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

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

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

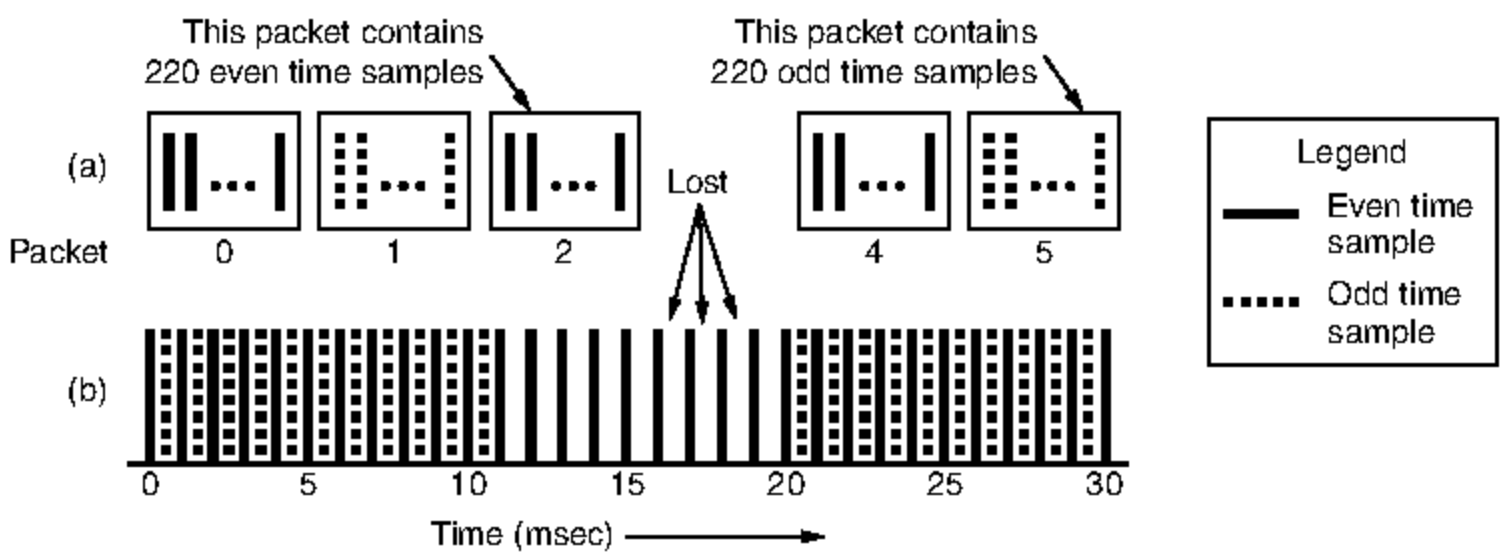

Figure 7-53. When packets carry alternate samples, the loss of a packet reduces the temporal resolution rather than creating a gap in time.

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

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

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

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

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

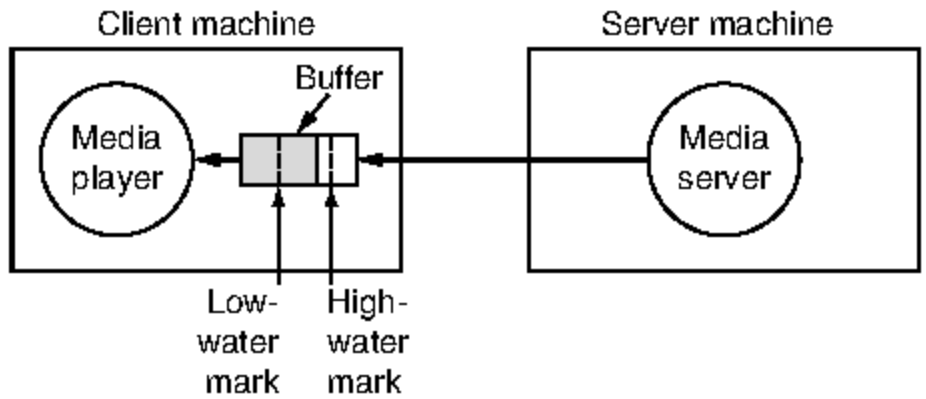

Figure 7-54. The media player buffers input from the media server and plays from the buffer rather than directly from the network.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

7.4.4 Streaming Live Media

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

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

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

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

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

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

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

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

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

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

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

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

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

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

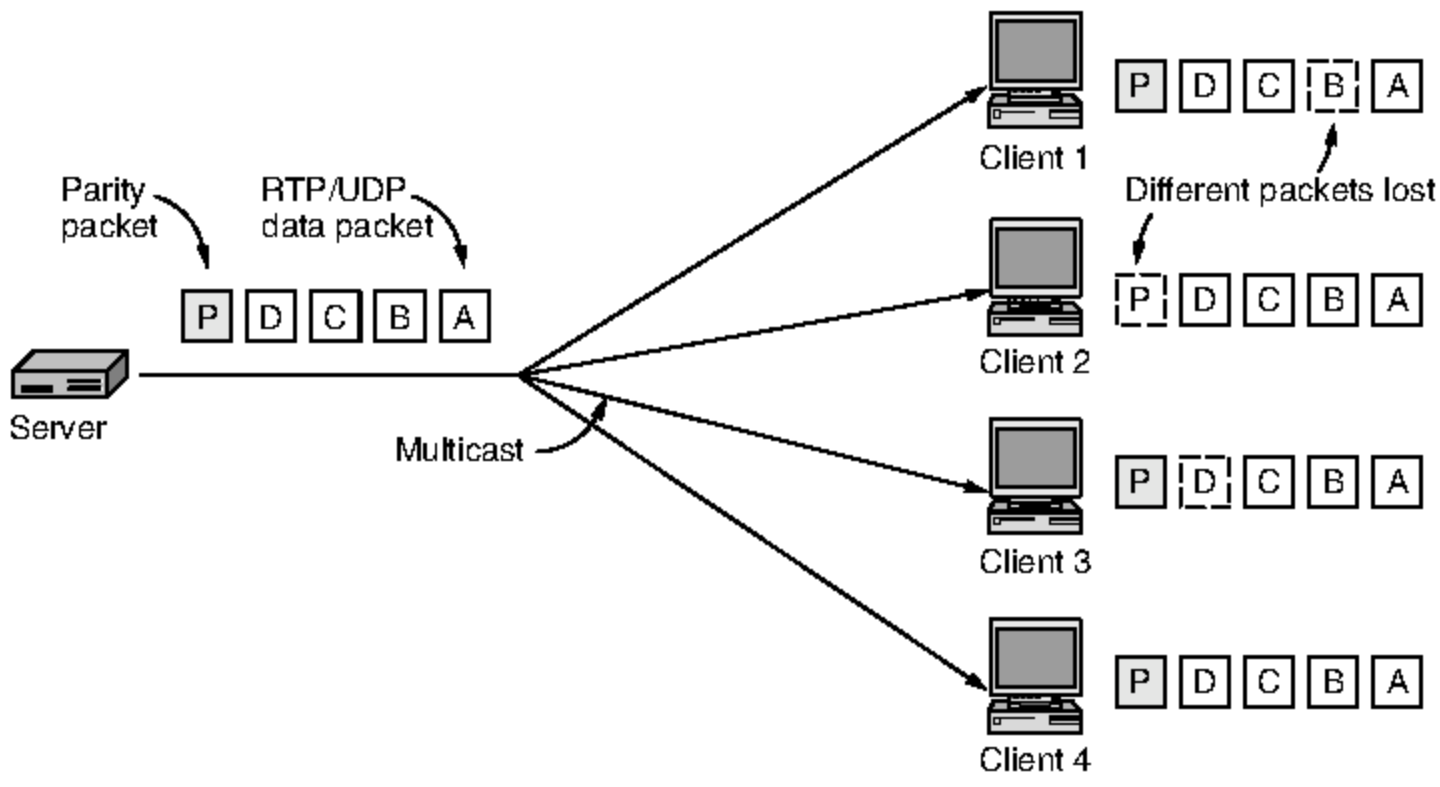

Figure 7-56. Multicast streaming media with a parity packet.

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

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

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

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

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

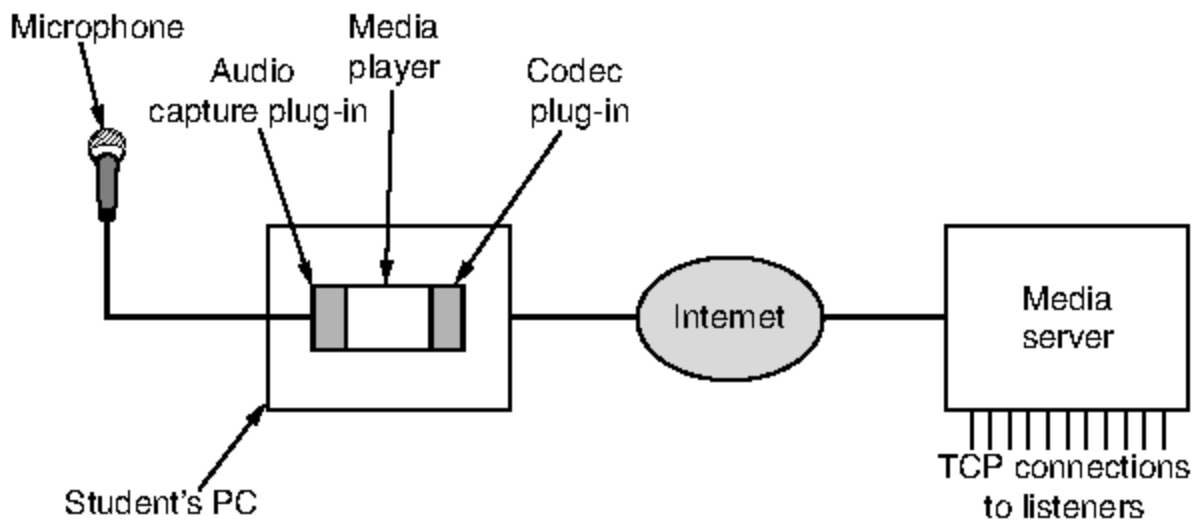

Figure 7-57. A student radio station.

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

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

7.4.5 Real-Time Conferencing

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

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

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

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

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

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

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

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

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

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

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

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

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

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