11.1 THE WORLD WIDE WEB

The World Wide Web (**WWW**) can be viewed as a huge distributed system consisting of millions of clients and servers for accessing linked documents. Servers maintain collections of documents, while clients provide users an easy-to-use interface for presenting and accessing those documents.

The Web started as a project at CERN, the European Particle Physics Laboratory in Geneva, to let its large and geographically dispersed group of researchers provide access to shared documents using a simple hypertext system. A document could be anything that could be displayed on a user's computer terminal, such as personal notes, reports, figures, blueprints, drawings, and so on. By linking documents to each other, it became easy to integrate documents from different projects into a new document without the necessity for centralized changes. The only thing needed was to construct a document providing links to other relevant documents (see also Berners-Lee et al., 1994).

The Web gradually grew worldwide encompassing sites other than high-energy physics, but popularity really increased when graphical user interfaces became available, notably Mosaic (Vetter et al., 1994). Mosaic provided an easy-to-use interface to present and access documents by merely clicking the mouse. A document was fetched from a server, transferred to a client, and presented on the screen. To a user, there was conceptually no difference between a document stored locally or in another part of the world. In this sense, distribution was transparent.

Since 1994, Web developments are primarily initiated and controlled by the World Wide Web Consortium, a collaboration between CERN and M.I.T. This consortium is responsible for standardizing protocols, improving interoperability, and further enhancing the capabilities of the Web. Its home page can be found at *http://www.w3.org/*.

11.1.1 Overview of WWW

The WWW is essentially a huge client-server system with millions of servers distributed worldwide. Each server maintains a collection of documents; each document is stored as a file (although documents can also be generated on request). A server accepts requests for fetching a document and transfers it to the client. In addition, it can also accept requests for storing new documents.

The simplest way to refer to a document is by means of a reference called a **Uniform Resource Locator** (**URL**). A URL is comparable to an IOR in CORBA and a contact address in Globe. It specifies where a document is located, often by embedding the DNS name of its associated server along with a file name by which the server can look up the document in its local file system. Furthermore, a URL specifies the application-level protocol for transferring the document across the network. There are different protocols available, as we explain below.

A client interacts with Web servers through a special application known as a **browser**. A browser is responsible for properly displaying a document. Also, a browser accepts input from a user mostly by letting the user select a reference to another document, which it then subsequently fetches and displays. This leads to the overall organization shown in Fig. 11-1.

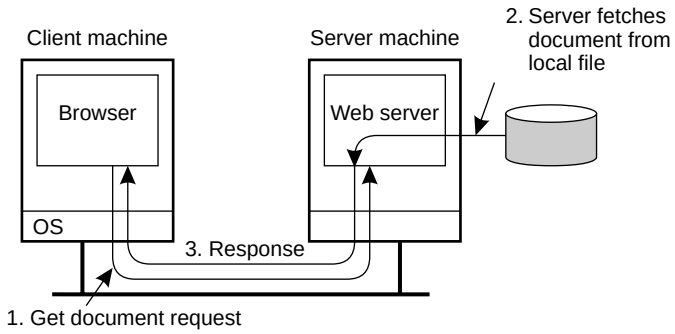


Figure 11-1. The overall organization of the Web.

The Web has evolved considerably since its introduction some 10 years ago. By now, there is a wealth of methods and tools to produce information that can be processed by Web clients and Web servers. In the following text, we will go into detail on how the Web acts as a distributed system. However, we skip most of the methods and tools to actually construct Web documents, as they often have no direct relationship to the distributed nature of the Web. A good and thorough introduction on how to build Web-based applications can be found in (Deitel and Deitel, 2000).

Document Model

Fundamental to the Web is that all information is represented by means of documents. There are many ways in which a document can be expressed. Some documents are as simple as an ASCII text file, while others are expressed by a collection of scripts that are automatically executed when the document is downloaded into a browser.

However, most important is that a document can contain references to other documents. Such references are known as **hyperlinks**. When a document is displayed in a browser, hyperlinks to other documents can be shown explicitly to the user. The user can then select a link by clicking on it. Selecting a hyperlink results in a request to fetch the document that is sent to the server where the referenced document is stored. From there, it is subsequently transferred to the user's machine and displayed by the browser. The new document may either replace the current one or be displayed in a new pop-up window.

Most Web documents are expressed by means of a special language called **HyperText Markup Language** or simply **HTML**. Being a markup language means that HTML provides keywords to structure a document into different sections. For example, each HTML document is divided into a heading section and a main body. HTML also distinguishes headers, lists, tables, and forms. It is also possible to insert images or animations at specific positions in a document. Besides these structural elements, HTML provides various keywords to instruct the browser how to present the document. For example, there are keywords to select a specific font or font size, to present text in italics or boldface, to align parts of text, and so on.

HTML is no longer what it used to be: a simple markup language. By now, it includes many features for producing glossy Web pages. One of its most powerful features is the ability to express parts of a document in the form of a script. To give a simple example, consider the HTML document shown in Fig. 11-2.

```
<HTML>                                          <!-- Start of HTML document      -->
<BODY>                                          <!-- Start of the main body      -->
<H1>Hello World</H1>                            <!-- Basic text to be displayed  -->
<P>                                             <!-- Start of new paragraph      -->
<SCRIPT type = "text/javascript">               <!-- Identify scripting language -->
    document.writeln("<H1>Hello World</H1>");   // Write a line of text
</SCRIPT>                                       <!-- End of scripting section    -->
</P>                                            <!-- End of paragraph section    -->
</BODY>                                         <!-- End of main body            -->
</HTML>                                         <!-- End of HTML section         -->
```

Figure 11-2. A simple Web page embedding a script written in JavaScript.

When this Web page is interpreted and displayed in a browser, the user will see the text ‘‘Hello World’’ twice, on separate lines. The first version of this text is the result of interpreting the HTML line

```
<H1>Hello World</H1>
```

The second version, however, is the result of running a small script written in **JavaScript**, a Java-like scripting language. The script consists of a single line of code

```
document.writeln("<H1>Hello World</H1>");
```

Although the effect of the two versions is exactly the same, there is clearly an important difference. The first version is the result of directly interpreting the HTML commands to properly display the marked up text. The second version is the result of executing a script that was downloaded into the browser as part of the document. In other words, we are faced with a form of mobile code.

When a document is parsed, it is internally stored as a rooted tree, called a parse tree, in which each node represents an element of that document. To achieve portability, the representation of the parse tree has been standardized. For example, each node can represent only one type of element from a predefined collection of element types. Similarly, each node is required to implement a standard interface containing methods for accessing its content, returning references to parent and child nodes, and so on. This standard representation is also known as the **Document Object Model** or **DOM** (le Hors et al., 2000). It is also often referred to as **dynamic HTML**.

The DOM provides a standard programming interface to parsed Web documents. The interface is specified in CORBA IDL, and mappings to various scripting languages such as JavaScript have been standardized. The interface is used by the scripts embedded in a document to traverse the parse tree, inspect and modify nodes, add and delete nodes, and so on. In other words, scripts can be used to inspect and modify the document that they are part of. Clearly, this opens a wealth of possibilities to dynamically adapt a document.

Although most Web documents are still expressed in HTML, an alternative language that also matches the DOM is **XML**, which stands for the **Extensible Markup Language** (Bray et al., 2000). Unlike HTML, XML is used only to structure a document; it contains no keywords to format a document such as centering a paragraph or presenting text in italics. Another important difference with HTML is that XML can be used to define arbitrary structures. In other words, it provides the means to define different *document types*.

```
(1)   <!ELEMENT article (title, author+, journal)>
(2)   <!ELEMENT title (#PCDATA)>
(3)   <!ELEMENT author (name, affiliation?)>
(4)   <!ELEMENT name (#PCDATA)>
(5)   <!ELEMENT affiliation (#PCDATA)>
(6)   <!ELEMENT journal(jname, volume, number?, month?, pages, year)>
(7)   <!ELEMENT jname (#PCDATA)>
(8)   <!ELEMENT volume (#PCDATA)>
(9)   <!ELEMENT number (#PCDATA)>
(10)  <!ELEMENT month (#PCDATA)>
(11)  <!ELEMENT pages (#PCDATA)>
(12)  <!ELEMENT year (#PCDATA)>
```

Figure 11-3. An XML definition for referring to a journal article.

Defining a document type requires that the elements of that document are declared first. As an example, Fig. 11-3 shows the XML definition of a simple general reference to a journal article. (The line numbers are not part of the definitions.) An article reference is declared in line 1 to be a document consisting of three elements: a *title*, an *author* element, and a *journal*. The ''+'' sign following the *author* element indicates that one or more authors are given.

In line 2, the *title* element is declared to consist of a series of characters (referred to in XML by the primitive *#PCDATA* data type). The *author* element is subdivided into two other elements: a *name* and an *affiliation*. The "?" indicates that the *affiliation* element is optional, but cannot be provided more than once per author in an article reference. Likewise, in line 6, the *journal* element is also subdivided into smaller elements. Each element that is not further subdivided (i.e., one that forms a leaf node in the parse tree), is specified to be a series of characters.

An actual article reference in XML can now be given as the XML document shown in Fig. 11-4 (again, the line numbers are not part of the document). Assuming that the definitions from Fig. 11-3 are stored in a file *article.dtd*, line 2 tells the XML parser where to find the definitions associated with the current document. Line 4 gives the title of the article, while lines 5 and 6 contain the name of each of the authors. Note that affiliations have been left out, which is permitted according to the document's XML definitions.

```
(1)  <?xml = version "1.0">
(2)  <!DOCTYPE article SYSTEM "article.dtd">
(3)  <article>
(4)      <title>Prudent Engineering Practice for Cryptographic Protocols</title>
(5)      <author><name>M. Abadi</name></author>
(6)      <author><name>R. Needham</name></author>
(7)      <journal>
(8)          <jname>IEEE Transactions on Software Engineering</jname>
(9)          <volume>22</volume>
(10)         <number>1</number>
(11)         <month>January</month>
(12)         <pages>6-15</pages>
(13)         <year>1996</year>
(14)     </journal>
(15) </article>
```

Figure 11-4. An XML document using the XML definitions from Fig. 11-3.

Presenting an XML document to a user requires that formatting rules are given. One simple approach is to embed an XML document into an HTML document and subsequently use HTML keywords for formatting. Alternatively, a separate formatting language known as the **Extensible Style Language** (**XSL**), can be used to describe the layout of an XML document. Details on how this approach works in practice can be found in (Deitel and Deitel, 2000).

Besides elements that traditionally belong in documents such as headers and paragraphs, HTML and XML can also support special elements that are tailored to multimedia support. For example, it is possible not only to include an image in a document, but also to attach video clips, audio files, and even interactive animations. Clearly, scripting also plays an important role in multimedia documents.

Document Types

There are many other types of documents besides HTML and XML. For example, a script can also be considered as a document. Other examples include documents formatted in Postscript or PDF, images in JPEG or GIF, or audio documents in MP3 format. The type of document is often expressed in the form of a **MIME type**. MIME stands for **Multipurpose Internet Mail Extensions** and was originally developed to provide information on the content of a message body that was sent as part of electronic mail. MIME distinguishes various types of message contents, which are described in (Freed and Borenstein, 1996). These types are also used in the WWW.

MIME makes a distinction between top-level types and subtypes. Different top-level types are shown in Fig. 11-5 and include types for text, image, audio, and video. There is a special *Application* type that indicates that the document contains data that are related to a specific application. In practice, only that application will be able to transform the document into something that can be understood by a human.

Type	Subtype	Description
Text	Plain	Unformatted text
	HTML	Text including HTML markup commands
	XML	Text including XML markup commands
Image	GIF	Still image in GIF format
	JPEG	Still image in JPEG format
Audio	Basic	Audio, 8-bit PCM sampled at 8000 Hz
	Tone	A specific audible tone
Video	MPEG	Movie in MPEG format
	Pointer	Representation of a pointer device for presentations
Application	Octet-stream	An uninterpreted byte sequence
	Postscript	A printable document in Postscript
	PDF	A printable document in PDF
Multipart	Mixed	Independent parts in the specified order
	Parallel	Parts must be viewed simultaneously

Figure 11-5. Six top-level MIME types and some common subtypes.

The *Multipart* type is used for composite documents, that is, documents that consists of several parts where each part will again have its own associated top-level type.

For each top-level type, there may be several subtypes available, also shown in Fig. 11-5. The type of a document is then represented as a combination of top-level type and subtype, such as, for example, *application/PDF*. In this case, it is

expected that a separate application is needed for processing the document, which is represented in PDF. Typically, Web pages are of type *text/HTML*, reflecting that they are represented as ASCII text containing HTML keywords.

Architectural Overview

The combination of HTML or XML with scripting provides a powerful means for expressing documents. However, we have hardly discussed where documents are actually processed, and what kind of processing takes place. The WWW started out as the relatively simple client-server system shown previously in Fig. 11-1. By now, this simple architecture has been extended with numerous components to support the advanced type of documents we just described.

One of the first enhancements to the basic architecture was support for simple user interaction by means of the **Common Gateway Interface** or simply **CGI**. CGI defines a standard way by which a Web server can execute a program taking user data as input. Usually, user data come from an HTML form; it specifies the program that is to be executed at the server side, along with parameter values that are filled in by the user. Once the form has been completed, the program's name and collected parameter values are sent to the server, as shown in Fig. 11-6.

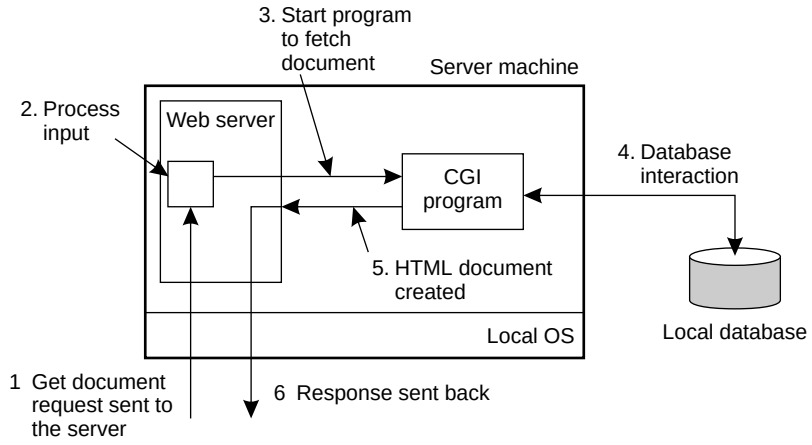


Figure 11-6. The principle of using server-side CGI programs.

When the server sees the request, it starts the program named in the request and passes it the parameter values. At that point, the program simply does its work and generally returns the results in the form of a document that is sent back to the user's browser to be displayed.

CGI programs can be as sophisticated as a developer wants. For example, as shown in Fig. 11-6, many programs operate on a database local to the Web server. After processing the data, the program generates an HTML document and returns

that document to the server. The server will then pass the document to the client. An interesting observation is that to the server, it appears as if it is asking the CGI program to fetch a document. In other words, the server does nothing else but delegate the fetching of a document to an external program.

The main task of a server used to be handling client requests by simply fetching documents. With CGI programs, fetching a document could be delegated in such a way that the server would remain unaware of whether a document had been generated on the fly, or actually read from the local file system. However, servers nowadays do much more than only fetching documents.

One of the most important enhancements is that servers can also process a fetched document before passing it to the client. In particular, a document may contain a **server-side script**, which is executed by the server when the document has been fetched locally. The result of executing a script is sent along with the rest of the document to the client. However, the script itself is not sent. In other words, using a server-side script changes a document by essentially replacing the script with the results of its execution.

To give a simple example of such a script, consider the HTML document shown in Fig. 11-7 (line numbers are not part of the document). When a client requests this document, the server will execute the script specified in lines 5–12. The server creates a local file object called *clientFile*, and uses it to open a file named */data/file.txt* from which it wants to only read data (lines 6–7). As long as there is data in the file, the server will send that data to the client as part of the HTML document (lines 8–9).

```
(1)  <HTML>
(2)  <BODY>
(3)  <P>The current content of <PRE>/data/file.txt</PRE> is:</P>
(4)  <P>
(5)  <SERVER type = "text/javascript">
(6)      clientFile = new File("/data/file.txt");
(7)      if(clientFile.open("r")){
(8)          while(!clientFile.eof())
(9)              document.writeln(clientFile.readln());
(10)         clientFile.close();
(11)     }
(12) </SERVER>
(13) </P>
(14) <P>Thank you for visiting this site.</P>
(15) </BODY>
(16) </HTML>
```

Figure 11-7. An HTML document containing a JavaScript to be executed by the server.

The server-side script is recognized as such by using the special, nonstandard HTML <*SERVER*> tag. Servers from different software manufacturers each have their own way to recognize server-side scripts. In this example, by processing the

script, the original document is modified so that the actual content of the file */data/file.txt* replaces the server-side script. In other words, the client will never get to see the script. If a document contains any client-side scripts, then these are just passed to the client in the usual manner.

Besides executing client-side and server-side scripts, it is also possible to pass precompiled programs to a client in the form of applets. In general, an **applet** is a small stand-alone application that can be sent to the client and executed in the browser's address space. The most common use of applets are in the form of Java programs that have been compiled to interpretable Java bytecodes. For example, the following includes a Java applet in an HTML document:

```
<OBJECT codetype = "application/java" classid = "java:welcome.class">
```

Applets are executed at the client. There is also a server-side counterpart of applets, known as servlets. Like an applet, a **servlet** is a precompiled program that is executed in the address space of the server. In current practice, servlets are mostly implemented in Java, but there is no fundamental restriction to use other languages. A servlet implements methods that are also found in HTTP: the standard communication protocol between client and server. HTTP is discussed in detail below.

Whenever a server receives an HTTP request addressed to a servlet, it calls the method associated with that request at the servlet. The latter, in turn, processes the request and, in general, returns a response in the form of an HTML document. The important difference with CGI scripts is that the latter are executed as separate processes, whereas servlets are executed by the server.

We can now give a more complete architectural view of the organization of clients and servers in the Web. This organization is shown in Fig. 11-8. Whenever a user issues a request for a document, the Web server can generally do one of three things, depending on what the document specifies it should do. First, it can fetch the document directly from its local file system. Second, it can start a CGI program that will generate a document, possibly using data from a local database. Third, it can pass the request to a servlet.

Once the document has been fetched, it may require some postprocessing by executing the server-side scripts it contains. In practice, this happens only to documents that have been directly fetched from the local file system, that is, without intervention of a servlet or CGI program. The document is then subsequently passed to the user's browser.

Once the document has been returned to the client, the browser will execute any client-side scripts, and possibly fetch and execute applets as referenced in the document. Document processing results in its content being displayed at the user's terminal.

In the architecture described so far, we have mainly assumed that a browser can display HTML documents, and possibly XML documents as well. However, there are many documents in other formats, such as Postscript or PDF. In such

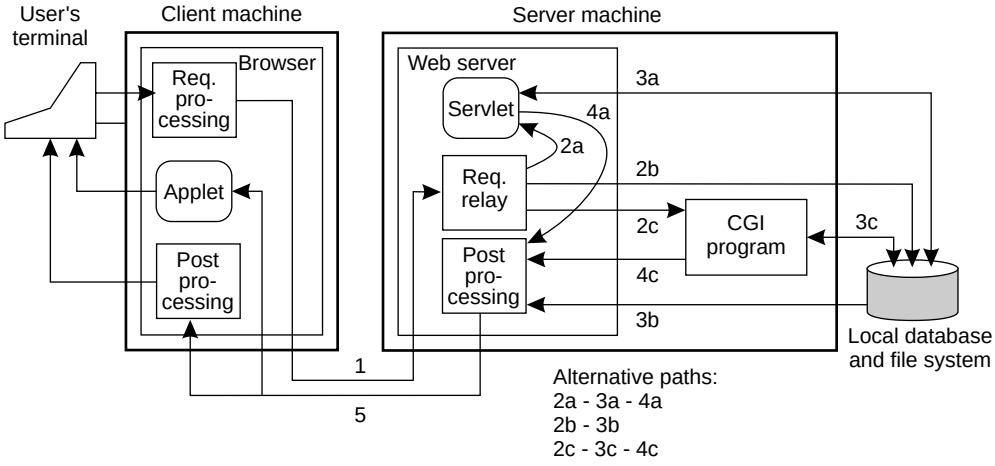


Figure 11-8. Architectural details of a client and server in the Web.

cases, the browser simply requests the server to fetch the document from its local file system and return it without further intervention. In effect, this approach results in a file being returned to the client. Using the file's extension, the browser subsequently launches an appropriate application to display or process its content. As they assist the browser in its task of displaying documents, these applications are also referred to as **helper applications**. Below, we discuss an alternative to using such applications.

11.1.2 Communication

All communication in the Web between clients and servers is based on the **Hypertext Transfer Protocol** (**HTTP**). HTTP is a relatively simple client-server protocol; a client sends a request message to a server and waits for a response message. An important property of HTTP is that it is stateless. In other words, it does not have any concept of open connection and does not require a server to maintain information on its clients. The most recent version of HTTP is described in (Fielding et al., 1999).

HTTP Connections

HTTP is based on TCP. Whenever a client issues a request to a server, it sets up a TCP connection to the server and sends its request message along that connection. The same connection is used for receiving the response. By using TCP as its underlying protocol, HTTP need not be concerned about lost requests and responses. A client and server may simply assume that their messages make it to the other side. If things do go wrong, for example, the connection is broken or a

time-out occurs an error is reported. However, in general, no attempt is made to recover from the failure.

One of the problems with the first versions of HTTP was its inefficient use of TCP connections. Each Web document is constructed from a collection of different files from the same server. To properly display a document, it is necessary that these files are also transferred to the client. Each of these files is, in principle, just another document for which the client can issue a separate request to the server where they are stored.

In HTTP version 1.0 and older, each request to a server required setting up a separate connection, as shown in Fig. 11-9(a). When the server had responded, the connection was broken down again. Such connections are referred to as being **nonpersistent**. A major drawback of nonpersistent connections is that it is relatively costly to set up a TCP connection. As a consequence, the time it can take to transfer an entire document with all its elements to a client may be considerable.

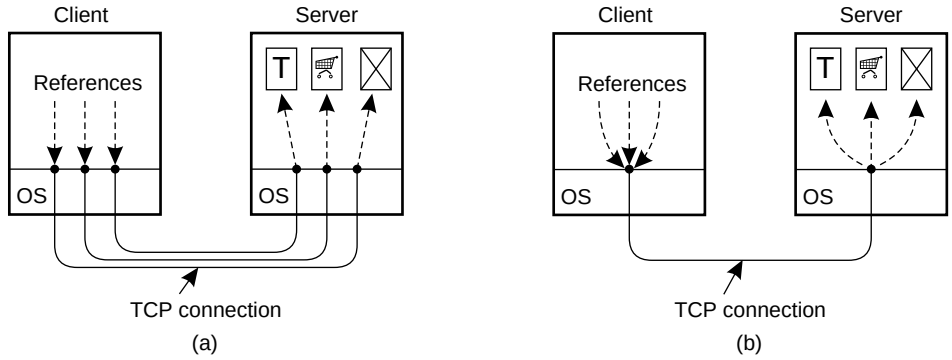


Figure 11-9. (a) Using nonpersistent connections. (b) Using persistent connections.

Note that HTTP does not preclude that a client sets up several connections simultaneously to the same server. This approach is often used to hide latency caused by the connection setup time, and to transfer data in parallel from the server to the client.

A better approach that is followed in HTTP version 1.1 is to use a **persistent connection**, which can be used to issue several requests (and their respective responses), without the need for a separate connection per *(request, response)*-pair. To further improve performance, a client can issue several requests in a row without waiting for the response to the first request (also referred to as **pipelining**). Using persistent connections is illustrated in Fig. 11-9(b).

HTTP Methods

HTTP has been designed as a general-purpose client-server protocol oriented toward the transfer of documents in both directions. A client can request each of these operations to be carried out at the server by sending a request message

containing the operation desired to the server. A list of the most commonly used request messages is given in Fig. 11-10.

Operation	Description
Head	Request to return the header of a document
Get	Request to return a document to the client
Put	Request to store a document
Post	Provide data that are to be added to a document (collection)
Delete	Request to delete a document

Figure 11-10. Operations supported by HTTP.

HTTP assumes that each document may have associated metadata, which are stored in a separate header that is sent along with a request or response. The head operation is submitted to the server when a client does not want the actual document, but rather only its associated metadata. For example, using the head operation will return the time the referred document was modified. This operation can be used to verify the validity of the document as cached by the client. It can also be used to check whether a document exists, without having to actually transfer the document.

The most important operation is get. This operation is used to actually fetch a document from the server and return it to the requesting client. It is possible to specify that a document should be returned only if it has been modified after a specific time. Also, HTTP allows documents to have associated **tags**, which are represented as character strings, and to fetch a document only if it matches certain tags.

The put operation is the opposite of the get operation. A client can request a server to store a document under a given name (which is sent along with the request). Of course, a server will in general not blindly execute put operations, but will only accept such requests from authorized clients. How these security issues are dealt with is discussed later.

The operation post is somewhat similar to storing a document, except that a client will request data to be added to a document or collection of documents. A typical example is posting an article to a news group. The distinguishing feature compared to a put operation, is that a post operation tells to which group of documents an article should be ''added.'' The article is sent along with the request. In contrast, a put operation carries a document and the name under which the server is requested to store that document.

Finally, the delete operation is used to request a server to remove the document that is named in the message sent to the server. Again, whether or not deletion actually takes place depends on various security measures. It may even be the case that the server itself does not have the proper permissions to delete the referred document. After all, the server is just a user process.

HTTP Messages

All communication between a client and server takes place through messages. HTTP recognizes only request and response messages. A request message consists of three parts, as shown in Fig. 11-11(a). The **request line** is mandatory and identifies the operation that the client wants the server to carry out along with a reference to the document associated with that request. A separate field is used to identify the version of HTTP the client is expecting. We explain the additional message headers below.

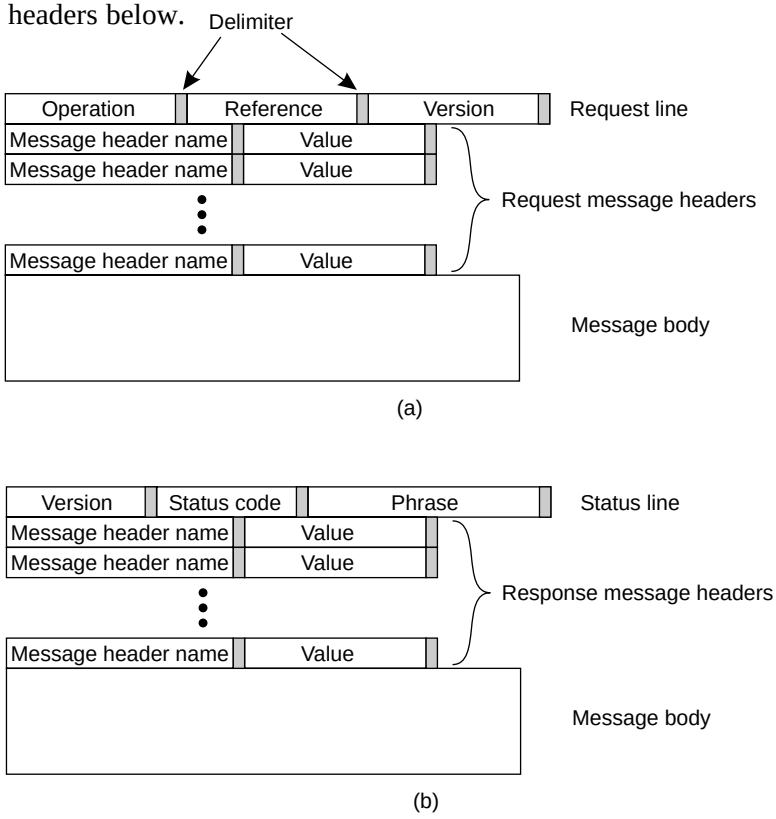


Figure 11-11. (a) HTTP request message. (b) HTTP response message.

A response message starts with a **status line** containing a version number and also a three-digit status code, as shown in Fig. 11-11(b). The code is briefly explained with a textual phrase that is sent along as part of the status line. For example, status code 200 indicates that a request could be honored, and has the associated phrase ‘‘OK.’’ Other frequently used codes are:

400 (Bad Request)
403 (Forbidden)
404 (Not Found).

A request or response message may contain additional headers. For example, if a client has requested a post operation for a read-only document, the server will respond with a message having status code 405 (''Method Not Allowed'') along with an *Allow* message header specifying the permitted operations (e.g., head and get). As another example, a client may be interested only in a document if it has not been modified since some time *T*. In that case, the client's get request is augmented with an *If-Modified-Since* message header specifying value *T*.

Header	Source	Contents
Accept	Client	The type of documents the client can handle
Accept-Charset	Client	The character sets are acceptable for the client
Accept-Encoding	Client	The document encodings the client can handle
Accept-Language	Client	The natural language the client can handle
Authorization	Client	A list of the client's credentials
WWW-Authenticate	Server	Security challenge the client should respond to
Date	Both	Date and time the message was sent
ETag	Server	The tags associated with the returned document
Expires	Server	The time for how long the response remains valid
From	Client	The client's e-mail address
Host	Client	The TCP address of the document's server
If-Match	Client	The tags the document should have
If-None-Match	Client	The tags the document should not have
If-Modified-Since	Client	Tells the server to return a document only if it has been modified since the specified time
If-Unmodified-Since	Client	Tells the server to return a document only if it has not been modified since the specified time
Last-Modified	Server	The time the returned document was last modified
Location	Server	A document reference to which the client should redirect its request
Referer	Client	Refers to client's most recently requested document
Upgrade	Both	The application protocol the sender wants to switch to
Warning	Both	Information about the status of the data in the message

Figure 11-12. Some HTTP message headers.

Fig. 11-12 shows a number of valid message headers that can be sent along with a request or response. Most of the headers are self-explanatory, so we will not discuss every one of them.

There are various headers that the client can send to the server explaining what it is able to accept as response. For example, a client may be able to accept

responses that have been compressed using the *gzip* compression tool available on most Windows and UNIX machines. In that case, the client will send an *Accept-Encoding* message header along with its request, with content ''Accept-Encoding:gzip.'' Likewise, an *Accept* message header can be used to specify, for example, that only HTML Web pages may be returned.

There are two message headers for security, but as we discuss later in this section, Web security is usually handled with a separate transport-layer protocol.

The *Location* and *Referer* message header are used to **redirect** a client to another document (note that ''Referer'' is misspelled in the specification). Redirecting corresponds to the use of forwarding pointers for locating a document, as explained in Chap. 4. When a client issues a request for document *D*, the server may possibly respond with a *Location* message header, specifying that the client should reissue the request, but now for document *D′*. When using the reference to *D′*, the client can add a *Referer* message header containing the reference to *D* to indicate what caused the redirection. In general, this message header is used to indicate the client's most recently requested document.

The *Upgrade* message header is used to switch to another protocol. For example, client and server may use HTTP/1.1 initially only to have a generic way of setting up a connection. The server may immediately respond with telling the client that it wants to continue communication with a secure version of HTTP, such as SHTTP (Rescorla and Schiffman, 1999). In that case, the server will send an *Upgrade* message header with content ''Upgrade:SHTTP.''

11.1.3 Processes

In essence, the Web makes use of only two kinds of processes: browsers by which users can access Web documents and have them displayed on their local screen, and Web servers, which respond to browser requests. Browsers may be assisted by helper applications, as we discussed above. Likewise, servers may be surrounded by additional programs, such as CGI scripts. In the following, we take a closer look at the typical client-side and server-side software that are used in the Web.

Clients

The most important Web client is a piece of software called a **Web browser**, which enables a user to navigate through Web pages by fetching those pages from servers and subsequently displaying them on the user's screen. A browser typically provides an interface by which hyperlinks are displayed in such a way that the user can easily select them through a single mouse click.

Web browsers are, in principle, simple programs. However, because they need to be able to handle a wide variety of document types and also provide an easy-to-use interface to users, they are generally complex pieces of software.

One of the problems that Web browser designers have to face is that a browser should be easily extensible so that it, in principle, can support any type of document that is returned by a server. The approach followed in most cases is to offer facilities for what are known as plug-ins. A **plug-in** is a small program that can be dynamically loaded into a browser for handling a specific document type. The latter is generally given as a MIME type. A plug-in should be locally available, possibly after being specifically transferred by a user from a remote server.

Plug-ins offer a standard interface to the browser and, likewise, expect a standard interface from the browser, as shown in Fig. 11-13. When a browser encounters a document type for which it needs a plug-in, it loads the plug-in locally and creates an instance. After initialization, the interaction with the rest of the browser is specific to the plug-in, although only the methods in the standardized interfaces will be used. The plug-in is removed from the browser when it is no longer needed.

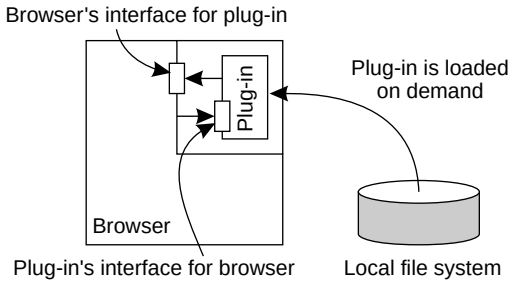


Figure 11-13. Using a plug-in in a Web browser.

Another client-side process that is often used is a **Web proxy** (Luotonen and Altis, 1994). Originally, such a process was used to allow a browser to handle application-level protocols other than HTTP, as shown in Fig. 11-14. For example, to transfer a file from an FTP server, the browser can issue an HTTP request to a local FTP proxy, which will then fetch the file and return it embedded in an HTTP response message.



Figure 11-14. Using a Web proxy when the browser does not speak FTP.

By now, most Web browsers are capable of supporting a variety of protocols and for that reason do not need proxies. However, Web proxies are still popular, but for a completely different reason, namely for providing a cache shared by a number of browsers. As we discuss below, when requesting a Web document, a

browser can pass its request to the local Web proxy. The proxy will then check whether the requested document is in its local cache before contacting the remote server where the document lies.

Servers

As we explained, a Web server is a program that handles incoming HTTP requests by fetching the requested document and returning it to the client. To give a concrete example, let us briefly take a look at the general organization of the **Apache server**, which is the dominant Web server on UNIX platforms.

The general organization of the Apache Web server is shown in Fig. 11-15. The server consists of a number of modules that are controlled by a single core module. The core module accepts incoming HTTP requests, which it subsequently passes to the other modules in a pipelined fashion. In other words, the core module determines the flow of control for handling a request.

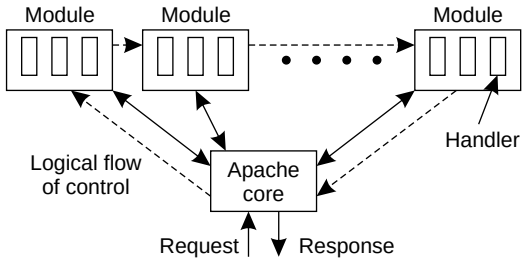


Figure 11-15. General organization of the Apache Web server.

For each incoming request, the core module allocates a request record with fields for the document reference contained in the HTTP request, the associated HTTP request headers, HTTP response headers, and so on. Each module operates on the record by reading and modifying fields as appropriate. Finally, when all modules have done their share in processing the request, the last one returns the requested document to the client. Note that, in principle, each request can follow its own pipeline.

Apache servers are highly configurable; numerous modules can be included to assist in processing an incoming HTTP request. To support this flexibility, the following approach is taken. Each module is required to provide one or more **handlers** that can be invoked by the core module. Handlers are all alike in the sense that they take a pointer to a request record as their sole input parameter. They are also the same in the sense that they can read and modify the fields in a request record.

In order to invoke the appropriate handler at the right time, processing HTTP requests is broken down into several phases. A module can register a handler for a specific phase. Whenever a phase is reached, the core module inspects which

handlers have been registered for that phase and invokes one of them as we discuss shortly. The phases are presented below:

1. Resolving the document reference to a local file name.
2. Client authentication.
3. Client access control.
4. Request access control.
5. MIME type determination of the response.
6. General phase for handling leftovers.
7. Transmission of the response.
8. Logging data on the processing of the request.

References are provided in the form of a **Uniform Resource Identifier** (**URI**), which we discuss below. There are several ways in which URIs can be resolved. In many cases, the local file name under which the referenced document is known will be part of the URI. However, when the document is to be generated, as in the case of a CGI program, the URI will contain information on which program the server should start, along with any input parameter values.

The second phase consists of authenticating the client. In principle, it consists only of verifying the client's identity without doing any checks concerning the client's access permissions.

Client access control deals with checking whether the client is actually known to the server and whether it has any access rights. For example, during this phase, it may turn out that client belongs to a certain group of users.

During request access control, a check is made as to whether the client possesses the permissions to have the issued request carried out at the referenced document.

The next phase deals with determining the MIME type of the document. There may be different ways of determining a document's type, such as bu using file extensions or using a separate file containing a document's MIME type.

Any additional request processing that does not fit into any of the other phases is handled in a general phase for ''leftovers.'' As an example, checking the syntax of a returned reference, or building up a user profile is typically something handled in this phase.

Finally, the response is transmitted to the user. Again, there are different ways of sending back a result depending on what was requested and how the response was generated.

The last phase consists of logging the various actions taken during the processing of the request. These logs are used for various purposes.

To process these phases, the core module maintains a list of handlers that have been registered for that phase. It chooses a handler and simply invokes it. A handler can either decline a request, handle it, or report an error. When a handler declines a requests, it effectively states that it cannot handle it so that the core module will need to select another handler that has been registered for the current phase. If a request could be handled, the next phase is generally started. When an error is reported, request processing is broken off, and the client is returned an error message.

Which modules are actually part of an Apache server is determined at configuration time. The simplest scenario is that the core module has to do all the request processing, in which case the server effectively reduces to a simple Web server that can handle only HTML documents. To allow several requests to be handled at the same time, the core module will generally start a new process for each incoming request. The maximum number of processes is also a configuration parameter. Details on configuring and programming the Apache server can be found in (Laurie and Laurie, 1999).

Server Clusters

An important problem related to the client-server nature of the Web is that a Web server can easily become overloaded. A practical solution employed in many designs is to simply replicate a server on a cluster of workstations, and use a front end to redirect client requests to one of the replicas (Fox et al., 1997). This principle is shown in Fig. 11-16, and is an example of horizontal distribution as we discussed in Chap. 1.

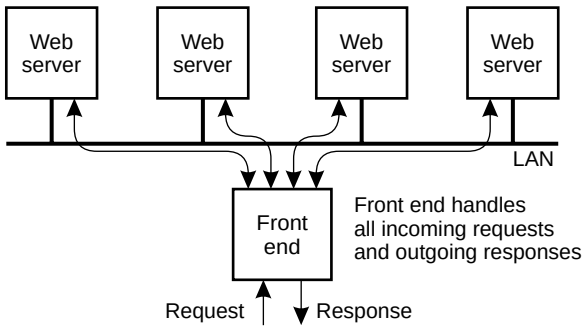


Figure 11-16. The principle of using a cluster of workstations to implement a Web service.

A crucial aspect of this organization is the design of the front end as it can easily become a serious performance bottleneck. In general, a distinction is made

between front ends operating as transport-layer switches, and those that operate at the level of the application layer.

As we mentioned, whenever a client issues an HTTP request, it sets up a TCP connection to the server. A transport-layer switch simply passes the data sent along the TCP connection to one of the servers, depending on some measurement of the server's load. The main drawback of this approach is that the switch cannot take into account the content of the HTTP request that is sent along the TCP connection. It can only base its redirection decisions on server loads.

In general, a better approach is to deploy **content-aware request distribution**, by which the front end first inspects an incoming HTTP request, and then decides which server it should forward that request to. This scheme can be combined with distributing content across a cluster of servers as described in (Yang and Luo, 2000).

Content-aware distribution has several advantages. For example, if the front end always forwards requests for the same document to the same server, that server may be able to effectively cache the document resulting in higher response times. In addition, it is possible to actually distribute the collection of documents among the servers instead of having to replicate each document for each server. This approach makes more efficient use of the available storage capacity and allows to using dedicated servers to handle special documents such as audio or video.

A problem with content-aware distribution is that the front end needs to do a lot of work. To improve performance, Pai et al. (1998) introduced a mechanism by which a TCP connection to the front end is handed off to a server. In effect, the server will directly respond to the client with no further interference of the front end, as shown in Fig. 11-17(a). TCP handoff is completely transparent to the client; the client will send its TCP messages to the front end (including acknowledgements and such), but will always receive messages from the server to which the connection had been handed off.

Further improvement can be accomplished by distributing the work of the front end in combination with a transport-layer switch, as discussed in (Aron et al., 2000). In combination with TCP handoff, the front end has two tasks. First, when a request initially comes in, it must decide which server will handle the rest of the communication with the client. Second, the front end should forward the client's TCP messages associated with the handed-off TCP connection.

These two tasks can be distributed as shown in Fig. 11-17(b). The dispatcher is responsible for deciding to which server a TCP connection should be handed off; a distributor monitors incoming TCP traffic for a handed-off connection. The switch is used to forward TCP messages to a distributor. When a client first contacts the Web service, its TCP connection setup message is forwarded to a distributor, which in turn contacts the dispatcher to let it decide to which server the connection should be handed off. At that point, the switch is notified that it should send all further TCP messages for that connection to the selected server.

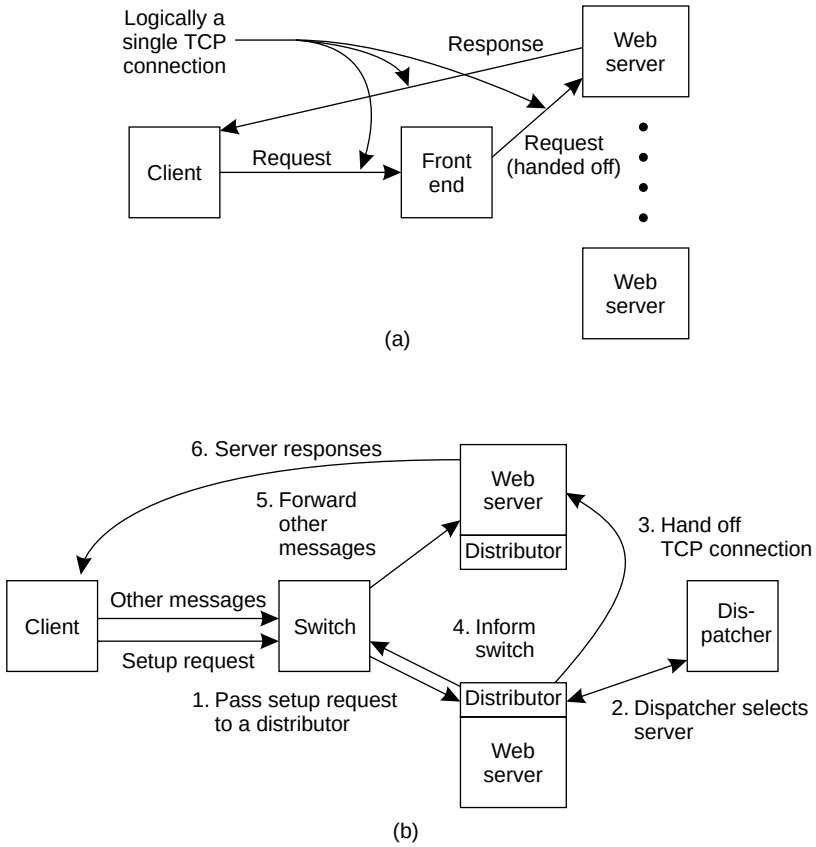


Figure 11-17. (a) The principle of TCP handoff. (b) A scalable content-aware cluster of Web servers.

11.1.4 Naming

The Web uses a single naming scheme to refer to documents, called **Uniform Resource Identifiers** or simply **URI**s (Berners-Lee et al., 1998). URIs come in two forms. A **Uniform Resource Locator** (**URL**) is a URI that identifies a document by including information on how and where to access the document. In other words, a URL is a location-dependent reference to a document. In contrast, a **Uniform Resource Name** (**URN**) acts as true identifier as discussed in Chap. 4. A URN is used as a globally unique, location-independent, and persistent reference to a document.

The actual syntax of a URI is determined by its associated **scheme**. The name of a scheme is part of the URI. Many different schemes have been defined, and in the following we will mention a few of them, along with examples of their associated URIs. The *http* scheme is the best known, but it is not the only one.

Uniform Resource Locators

A URL contains information on how and where to access a document. How to access a document is often reflected by the name of the scheme that is part of the URL, such as *http*, *ftp*, or *telnet*. Where a document is located is often embedded in a URL by means of the DNS name of the server to which an access request can be sent, although an IP address can also be used. The number of the port on which the server will be listening for such requests is also part of the URL; when left out, a default port is used. Finally, a URL also contains the name of the document to be looked up by that server, leading to the general structures shown in Fig. 11-18.

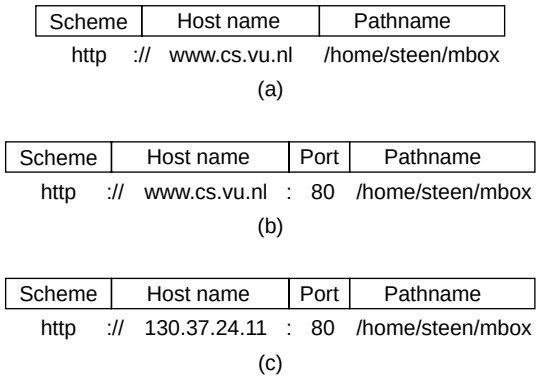


Figure 11-18. Often-used structures for URLs. (a) Using only a DNS name. (b) Combining a DNS name with a port number. (c) Combining an IP address with a port number.

Resolving a URL such as those shown in Fig. 11-18 is straightforward. If the server is referred to by its DNS name, that name will need to be resolved to the server's IP address. Using the port number contained in the URL, the client can then contact the server using the protocol named by the scheme, and pass it the document's name that forms the last part of the URL.

Fig. 11-19 shows a number of examples of URLs. The *http* URL is used to transfer documents using HTTP as we explained above. Likewise, there is an *ftp* URL for file transfer using FTP.

An immediate form of documents is supported by *data* URLs (Masinter, 1998). In such a URL, the document itself is embedded in the URL, similar to embedding the data of a file in an inode (Mullender and Tanenbaum, 1984). The example shows a URL containing plain text representing the Greek character string $\alpha\beta\gamma$.

URLs are often used as well for other purposes than referring to a document. For example, a *telnet* URL is used for setting up a telnet session to a server. There are also URLs for telephone-based communication as described in (Vaha-Sipila,

Name	Used for	Example
http	HTTP	http://www.cs.vu.nl:80/globe
ftp	FTP	ftp://ftp.cs.vu.nl/pub/minix/README
file	Local file	file:/edu/book/work/chp/11/11
data	Inline data	data:text/plain;charset=iso-8859-7,%e1%e2%e3
telnet	Remote login	telnet://flits.cs.vu.nl
tel	Telephone	tel:+31201234567
modem	Modem	modem:+31201234567;type=v32

Figure 11-19. Examples of URLs.

2000). The *tel* URL as shown in Fig. 11-19 essentially embeds only a telephone number and simply lets the client establish a call across the telephone network. In this case, the client will typically be a device such as a handheld telephone. The *modem* URL can be used to set up a modem-based connection with another computer. In the example, the URL states that the remote modem should adhere to the ITU-T V32 standard.

Uniform Resource Names

In contrast to URLs, URNs are location-independent references to documents. A URN follows the structure shown in Fig. 11-20, which consists of three parts (Moats, 1997). Note that URNs form a subset of all URIs by means of the *urn* scheme. The name-space identifier, however, determines the syntactic rules for the third part of a URN.

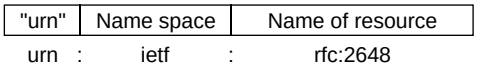


Figure 11-20. The general structure of a URN.

A typical example of a URN is the one used for identifying books by means of their ISBN such as *urn:isbn:0-13-349945-6* (Lynch et al., 1998), or URNs for identifying IETF documents such as *urn:ietf:rfc:2648* (Moats, 1999). **IETF** stands for the **Internet Engineering Task Force**, which is the organization responsible for developing Internet standards.

Although defining a URN name space is relatively easy, resolving a URN is much more difficult, as URNs deriving from different name spaces may have very different structures. As a consequence, resolution mechanisms will need to be introduced per name space. Unfortunately, none of the currently devised URN name spaces provide hints for such a resolution mechanism.

11.1.5 Synchronization

Synchronization has not been much of an issue for the Web, mainly for two reasons. First, the strict client-server organization of the Web, in which servers never exchange information with other servers (or clients with other clients) means that there is nothing much to synchronize. Second, the Web can be considered as being a read-mostly system. Updates are generally done by a single person, and hardly ever introduce write-write conflicts.

However, things are changing, and there is an increasing demand to provide support for collaborative authoring of Web documents. In other words, the Web should provide support for concurrent updates of documents by a group of collaborating users or processes.

For this reason, an extension to HTTP has been proposed, called **WebDAV** (Whitehead and Wiggins, 1998). WebDAV stands for **Web Distributed Authoring and Versioning** and provides a simple means to lock a shared document, and to create, delete, copy, and move documents from remote Web servers. We briefly describe synchronization as supported in WebDAV. More information on additional features are described in (Whitehead and Golan, 1999). A detailed specification of WebDAV can be found in (Golan et al., 1999).

To synchronize concurrent access to a shared document, WebDAV supports a simple locking mechanism. There are two types of write locks. An exclusive write lock can be assigned to a single client, and will prevent any other client from modifying the shared document while it is locked. There is also a shared write lock, which allows multiple clients to simultaneously update the document. Because locking takes place at the granularity of an entire document, shared write locks are convenient when clients modify different parts of the same document. However, the clients, themselves, will need to take care that no write-write conflicts occur.

Assigning a lock is done by passing a lock token to the requesting client. The server registers which client currently has the lock token. Whenever the client wants to modify the document, it sends an HTTP post request to the server, along with the lock token. The token shows that the client has write-access to the document, for which reason the server will carry out the request.

An important design issue is that there is no need to maintain a connection between the client and the server while holding the lock. The client can simply disconnect from the server after acquiring the lock, and reconnect to the server when sending an HTTP request.

Note that when a client holding a lock token crashes, the server will one way or the other have to reclaim the lock. WebDAV does not specify how servers should handle these and similar situations, but leaves that open to specific implementations. The reasoning is that the best solution will depend on the type of documents that WebDAV is being used for. The reason for this approach is that there is no general way to solve the problem of orphan locks in a clean way.

11.1.6 Caching and Replication

To improve the performance of the Web, client-side caching has always played an important role in the design of Web clients, and continues to do so (see, e.g., Barish and Obraczka, 2000). In addition, in an attempt to offload heavily-used Web servers, it has been common practice to replicate Web sites and place copies across the Internet. More recently, sophisticated techniques for Web replication are emerging, while caching is gradually losing popularity. Let us take a closer look at these issues.

Web Proxy Caching

Client-side caching generally occurs at two places. In the first place, most browsers are equipped with a simple caching facility. Whenever a document is fetched it is stored in the browser's cache from where it is loaded the next time. Clients can generally configure caching by indicating when consistency checking should take place, as we explain for the general case below.

In the second place, a client's site often runs a Web proxy. As we explained above, a Web proxy accepts requests from local clients and passes these to Web servers. When a response comes in, the result is passed to the client. The advantage of this approach is that the proxy can cache the result and return that result to another client, if necessary. In other words, a Web proxy can implement a shared cache.

In addition to caching at browsers and proxies, it is also possible to place caches that cover a region, or even a country, thus leading to a hierarchical caching scheme. Such schemes are mainly used to reduce network traffic, but have the disadvantage of incurring a higher latency compared to using nonhierarchical schemes. This higher latency is caused by the fact that several cache servers may need to be contacted instead of at most one compared to the nonhierarchical scheme.

Different cache-consistency protocols have been deployed in the Web. To guarantee that a document returned from the cache is consistent, some Web proxies first send a conditional HTTP get request to the server with an additional *If-Modified-Since* request header, specifying the last modification time associated with the cached document. Only if the document has been changed since that time, will the server return the entire document. Otherwise, the Web proxy can simply return its cached version to the requesting local client. Following the terminology introduced in Chap. 6, this corresponds to a pull-based protocol.

Unfortunately, this strategy requires that the proxy contacts a server for each request. To improve performance at the cost of weaker consistency, the widely-used Squid Web proxy (Chankhunthod et al., 1996) assigns an expiration time T_{expire} that depends on how long ago the document was last modified when it is

cached. In particular, if $T_{last_modified}$ is the last modification time of a document (as recorded by its owner), and T_{cached} is the time it was cached, then

$$T_{expire} = \alpha(T_{cached} - T_{last_modified}) + T_{cached}$$

with $\alpha = 0.2$ (this value has been derived from practical experience). Until T_{expire}, the document is considered valid and the proxy will not contact the server. After the expiration time, the proxy requests the server to send a fresh copy, unless it had not been modified. In other words, when $\alpha = 0$, the strategy is the same as the previous one we discussed.

Note that documents that have not been modified for a long time will not be checked for modifications as soon as recently modified documents. This strategy was first proposed for the Alex file system (Cate, 1992). The obvious drawback is that a proxy may return an invalid document, that is, a document that is older than the current version stored at the server. Worse yet, there is no way for the client to detect the fact that it just received an obsolete document.

As an alternative to the pull-based protocol is that the server notifies proxies that a document has been modified by sending an invalidation. The problem with this approach for Web proxies, is that the server may need to keep track of a large number of proxies, inevitably leading to a scalability problem. However, by combining leases and invalidations, Cao and Liu (1998) show that the state to be maintained at the server can be kept within acceptable bounds. Nevertheless, invalidation protocols for Web proxy caches are hardly ever applied.

One of the problems the designers of Web proxy caches are faced with is that caching at proxies makes sense only if documents are indeed shared by different clients. In practice, it turns out that cache hit ratios are limited to approximately 50 percent, and only if the cache is allowed to be extremely large. One approach to building virtually very large caches that can serve a huge number of clients, is to make use of cooperative caches of which the principle is shown in Fig. 11-21.

In **cooperative caching**, whenever a cache miss occurs at a Web proxy, the proxy first checks a number of neighboring proxies to see if one of them contains the requested document. If such a check fails, the proxy forwards the request to the Web server responsible for the document (see, e.g., Tewari et al., 1999). A study by Wolman et al. (1999) shows that cooperative caching may be effective for relatively small groups of clients (in the order of tens of thousands of users). However, such groups can also be serviced by using a single proxy cache, which is much cheaper in terms of communication and resource usage.

Another problem with Web proxy caches is that they can be used only for static documents, that is, documents that are not generated on-the-fly by Web servers as the response to a client's request. Such documents are unique in the sense that the same request from a client will presumably lead to a different response the next time.

In some cases, it is possible to shift generation of the document from the server to the proxy cache, as proposed in **active caches** (Cao et al., 1998). In this

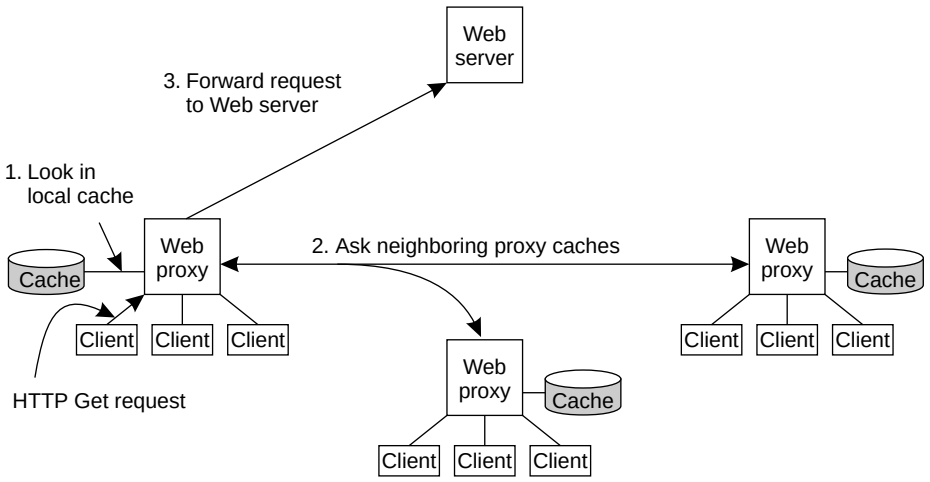


Figure 11-21. The principle of cooperative caching.

approach, when fetching a document that is normally generated, the server returns an applet that is to be cached at the proxy. The proxy is subsequently responsible for generating the response for the requesting client by running the applet. The applet itself is kept in the cache for the next time the same request is issued, at which point it is executed again. Note that the applet itself can also become obsolete if it is modified at the server.

There are several applications for active caches. For example, whenever a generated document is very large, a cached applet can be used to request the server to send only the differences the next time the document needs to be regenerated.

As another example, consider a document that includes a list of advertisement banners from which only one is displayed each time the document is requested. In this case, the first time the document is requested, the server sends the document and the complete list of banners to the proxy along with an applet for choosing a banner. The next time the document is requested, the proxy can handle the request by itself, and generate a response but this time also returning a different banner to display.

Server Replication

There are two ways in which server replication in the Web generally takes place. First, as we explained above, heavily-loaded Web sites use Web server clusters to improve response times. In general, this type of replication is transparent to clients. Second, a nontransparent form of replication that is widely deployed is to make an entire copy of a Web site available at a different server.

This approach is also called **mirroring**. A client is then offered to choose among several servers to access the Web site.

Recently, a third form of replication is gradually gaining popularity and follows the server-initiated replica placement strategy we discussed in Chap. 6. In this approach, a collection of servers is distributed across the Internet offering the facilities for hosting Web documents by replicating documents across the servers taking client access characteristics into account. The collection of servers are known as a **Content Delivery Network** (**CDN**), sometimes also called content distribution network.

There are various techniques for replicating and distributing documents across a CDN. We presented one solution in Chap. 6, which is used in the RaDaR Web hosting service (Rabinovich and Aggarwal, 1999). In RaDaR, a server keeps track of the number of requests that come from clients in a particular region. Requests are measured as if they came from a RaDaR server in that region, after which a decision is made to migrate or copy a document to that region's server. Another popular, but simpler approach is implemented by Akamai and works roughly as follows (see also (Leighton and Lewin, 2000).

The basic idea is that each Web document consists of a main HTML page in which several other documents such as images, video, and audio have been embedded. To display the entire document, it is necessary that the embedded documents are fetched by the user's browser as well. The assumption is that these embedded documents rarely change, for which reason it makes sense to cache or replicate them.

Each embedded document is normally referenced through a URL similar to those shown in Fig. 11-18. In Akamai's CDN, such a URL is modified such that it refers to a **virtual ghost**, which is a reference to an actual server in the CDN. The modified URL is resolved as follows, as is also shown in Fig. 11-22.

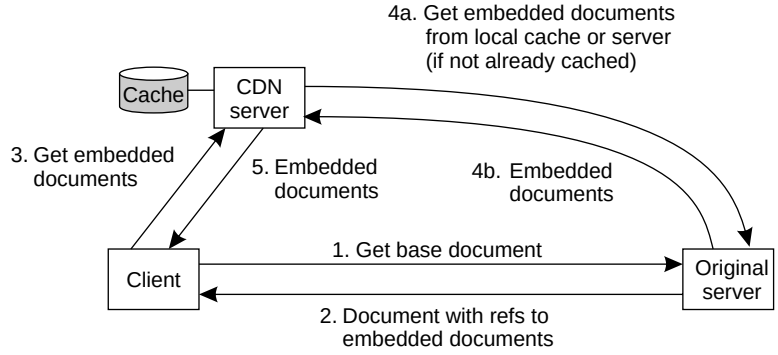


Figure 11-22. The principle working of the Akamai CDN.

The name of the virtual ghost includes the DNS name *ghosting.com*, which is resolved by the regular DNS naming system to a CDN DNS server closest to the

client that requested the DNS name to be resolved. The selection of the closest CDN DNS server is done by looking up the client's IP address in a database containing a ''map'' of the Internet. Each of the CDN DNS servers keeps track of the regular CDN servers in its proximity. Name resolution then proceeds at the CDN DNS server, which returns the address of one of the regular CDN servers in its proximity, according to some selection criteria such as current load.

Finally, the client forwards the request for the embedded document to the selected CDN server. If this server does not yet have the document, it fetches it from the original Web server (shown as step 4 in Fig. 11-22), caches it locally, and subsequently passes it to the client. If the document was already in the CDN server's cache, it can be returned immediately.

An important aspect in Akamai's CDN, and any other CDN for that matter, is locating a nearby CDN server. One possible approach is to use a location service like the one described in Chap. 4. The approach used in Akamai as just described relies on DNS, in combination with maintaining a ''map'' of the Internet. An alternative approach, described in (Amir et al., 1998), is to assign the same IP address to several DNS servers, and to let the underlying network layer automatically direct the name lookup request to the nearest server based on its shortest-path routing protocol.

11.1.7 Fault Tolerance

Fault tolerance in the Web is mainly achieved through client-side caching and server replication. No special methods are incorporated in, for example, HTTP to assist fault tolerance or recovery. Note, however, that high availability in the Web is achieved through redundancy that makes use of generally available techniques in crucial services such as DNS. For example, DNS allows several addresses to be returned as the result of a name lookup.

11.1.8 Security

Considering the open nature of the Internet, devising a security architecture that protects clients and servers against various attacks is crucially important. Most of the security issues in the Web deal with setting up a secure channel between a client and server. The predominant approach for setting up a secure channel in the Web is to use the **Secure Socket Layer** (**SSL**), originally proposed by Netscape. Although SSL has never been formally standardized, most Web clients and servers support it. Recently, an update of SSL has been formerly laid down in RFC 2246, now referred to as the **Transport Layer Security** (**TLS**) protocol (Dierks and Allen, 1996).

As shown in Fig. 11-23, TLS is an application-independent security protocol that is logically layered on top of a transport protocol. For reasons of simplicity, TLS (and SSL) implementations are usually based on TCP. TLS can support a

variety of higher-level protocols, including HTTP, as we discuss below. For example, it is possible to implement secure versions of FTP or Telnet using TLS.

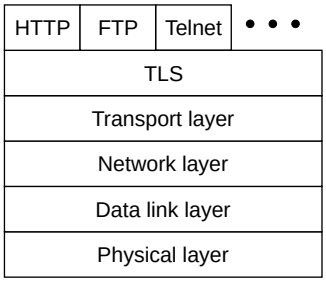


Figure 11-23. The position of TLS in the Internet protocol stack.

TLS itself is organized into two layers. The core of the protocol is formed by the **TLS record protocol layer**, which implements a secure channel between a client and server. The exact characteristics of the channel are determined during its setup, but may include message fragmentation and compression, which are applied in conjunction with message authentication, integrity, and confidentiality.

Setting up a secure channel proceeds in two phases, as shown in Fig. 11-24. First, the client informs the server of the cryptographic algorithms it can handle, as well as any compression methods it supports. The actual choice is always made by the server, which reports its choice back to the client. These are first two messages shown in Fig. 11-24.

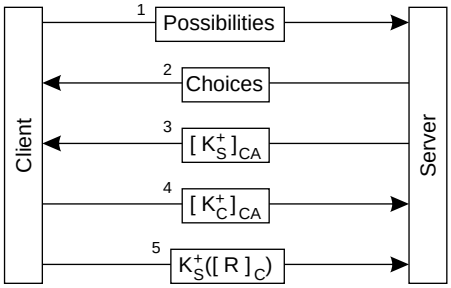


Figure 11-24. TLS with mutual authentication.

In the second phase, authentication takes place. The server is always required to authenticate itself, for which reason it passes the client a certificate containing its public key signed by a certification authority *CA*. If the server requires that the client be authenticated, the client will have to send a certificate to the server as well, shown as message 4 in Fig. 11-24.

The client generates a random number that will be used by both sides for constructing a session key, and sends this number to the server, encrypted with the

server's public key. In addition, if client authentication is required, the client signs the number with its private key, leading to message 5 in Fig. 11-24. (In reality, a separate message is sent with a scrambled and signed version of the random number, establishing the same effect.) At that point, the server can verify the identity of the client, after which the secure channel has been set up.