Chapter 12. Distributed Web-Based Systems

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 access 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 slowly to sites other than high-energy physics, but popularity sharply 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 a mouse button. 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.
[Page 546]

Since 1994, Web developments have been initiated 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. In addition, we see many new developments take place outside this consortium, not always leading to the compability one would hope for. By now, the Web is more than just a simple document-based system. Notably with the introduction of Web services we are seeing a huge distributed system emerging in which services rather than just documents are being used, composed, and offered to any user or machine that can find use of them.

In this chapter we will take a closer look at this rapidly growing and pervasive system. Considering that the Web itself is so young and that so much as changed in such a short time, our description can only be a snapshot of its current state. However, as we shall see, many concepts underlying Web technology are based on the principles discussed in the first part of this book. Also, we will see that for many concepts, there is still much room for improvement.

12.1. Architecture

The architecture of Web-based distributed systems is not fundamentally different from other distributed systems. However, it is interesting to see how the initial idea of supporting distributed documents has evolved since its inception in 1990s. Documents turned from being purely static and passive to dynamically generated containing all kinds of active elements. Furthermore, in recent years, many organizations have begun supporting services instead of just documents. In the following, we discuss the architectural impacts of these shifts.

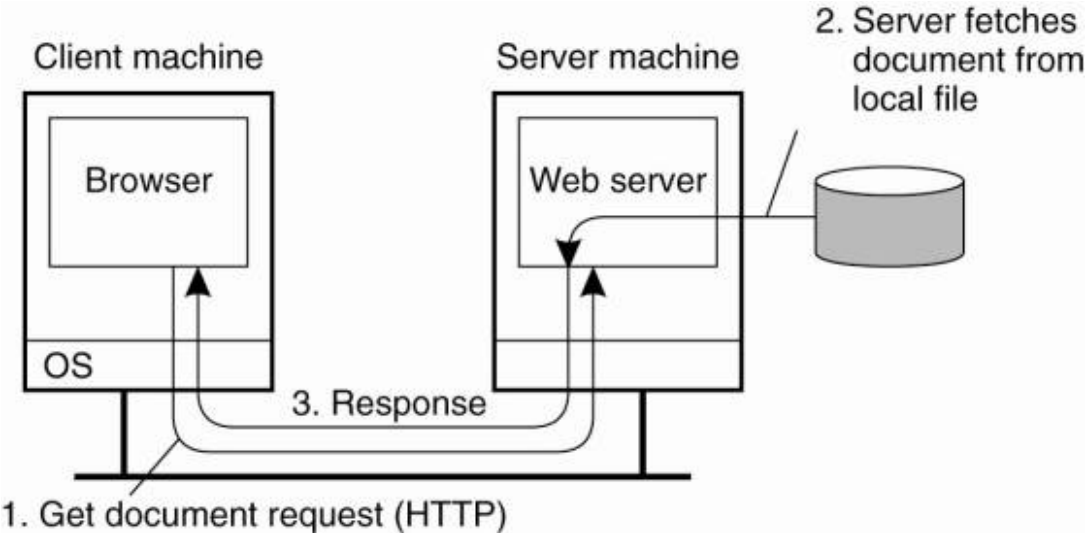
12.1.1. Traditional Web-Based Systems

Unlike many of the distributed systems we have been discussing so far, Webbased distributed systems are relatively new. In this sense, it is somewhat difficult to talk about traditional Web-based systems, although there is a clear distinction between the systems that were available at the beginning and those that are used today.

Many Web-based systems are still organized as relatively simple client-server architectures. The core of a Web site is formed by a process that has access to a local file system storing documents. The simplest way to refer to a document is by means of a reference called a Uniform Resource Locator (URL). 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 several different protocols available, as we explain below.
[Page 547]

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. The communication between a browser and Web server is standardized: they both adhere to the HyperText Transfer Protocol (HTTP) which we will discuss below. This leads to the overall organization shown in Fig. 12-1.

Figure 12-1. The overall organization of a traditional Web site.



The Web has evolved considerably since its introduction. 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, we will go into detail on how the Web acts as a distributed system. However, we skip most of the methods and tools used to construct Web documents, as they often have no direct relationship to the distributed nature of the Web. A good introduction on how to build Web-based applications can be found in Sebesta (2006).

Web Documents

Fundamental to the Web is that virtually all information comes in the form of a document. The concept of a document is to be taken in its broadest sense: not only can it contain plain text, but a document may also include all kinds of dynamic features such as audio, video, animations and so on. In many cases, special helper applications are needed to make a document "come to life." Such interpreters will typically be integrated with a user's browser.

Most documents can be roughly divided into two parts: a main part that at the very least acts as a template for the second part, which consists of many different bits and pieces that jointly constitute the document that is displayed in a browser. The main part is generally written in a markup language, very similar to the type of languages that are used in word-processing systems. The most widely-used markup language in the Web is HTML, which is an acronym for HyperText Markup Language. As its name suggests, HTML allows the embedding of links to other documents. When activating such links in a browser, the referenced document will be fetched from its associated server.
[Page 548]

Another, increasingly important markup language is the Extensible Markup Language (XML) which, as its name suggests, provides much more flexibility in defining what a document should look like. The major difference between HTML and XML is that the latter includes the definitions of the elements that mark up a document. In other words, it is a meta-markup language. This approach provides a lot of flexibility when it comes to specifying exactly what a document looks like: there is no need to stick to a single model as dictated by a fixed markup language such as HTML.

HTML and XML can also include all kinds of tags that refer to embedded documents, that is, references to files that should be included to make a document complete. It can be argued that the embedded documents turn a Web document into something active. Especially when considering that an embedded document can be a complete program that is executed on-the-fly as part of displaying information, it is not hard to imagine the kind of things that can be done.

Embedded documents come in all sorts and flavors. This immediately raises an issue how browsers can be equipped to handle the different file formats and ways to interpret embedded documents. Essentially, we need only two things: a way of specifying the type of an embedded document, and a way of allowing a browser to handle data of a specific type.

Each (embedded) document has an associated MIME type. MIME stands for Multipurpose Internet Mail Exchange and, as its name suggests, 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. These types are also used in the WWW, but it is noted that standardization is difficult with new data formats showing up almost daily.

MIME makes a distinction between top-level types and subtypes. Some common top-level types are shown in Fig. 12-2 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.

Figure 12-2. Six top-level MIME types and some common subtypes.
(This item is displayed on page 549 in the print version)

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

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, of which some are also shown in Fig. 12-2. 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. Many subtypes are experimental, meaning that a special format is used requiring its own application at the user's side. In practice, it is the Web server who will provide this application, either as a separate program that will run aside a browser, or as a so-called plugin that can be installed as part of the browser.
[Page 549]

This (changing) variety of document types forces browsers to be extensible. To this end, some standardization has taken place to allow plug-ins adhering to certain interfaces to be easily integrated in a browser. When certain types become popular enough, they are often shipped with browsers or their updates. We return to this issue below when discussing client-side software.

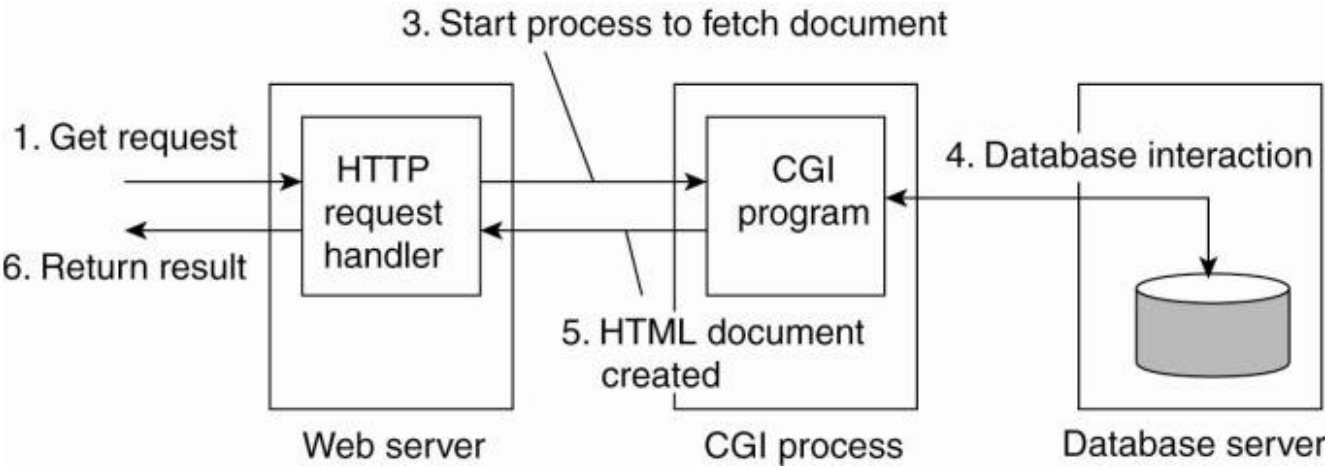
Multitiered Architectures

The combination of HTML (or any other markup language such as 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 two-tiered client-server system shown previously in Fig. 12-1. 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. 12-3. [Page 550]

Figure 12-3. 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. 12-3, 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 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. Note that we have just described a two-tiered organization of server-side software.

However, servers nowadays do much more than just fetching documents. One of the most important enhancements is that servers can also process a 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. 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.

As server-side processing of Web documents increasingly requires more flexibility, it should come as no surprise that many Web sites are now organized as a three-tiered architecture consisting of a Web server, an application server, and a database. The Web server is the traditional Web server that we had before; the application server runs all kinds of programs that may or may not access the third tier, consisting of a database. For example, a server may accept a customer's query, search its database of matching products, and then construct a clickable Web page listing the products found. In many cases the server is responsible for running Java programs, called servlets, that maintain things like shopping carts, implement recommendations, keep lists of favorite items, and so on.
[Page 551]

This three-tiered organization introduces a problem, however: a decrease in performance. Although from an architectural point of view it makes sense to distinguish three tiers, practice shows that the application server and database are potential bottlenecks. Notably improving database performance can turn out to be a nasty problem. We will return to this issue below when discussing caching and replication as solutions to performance problems.

12.1.2. Web Services

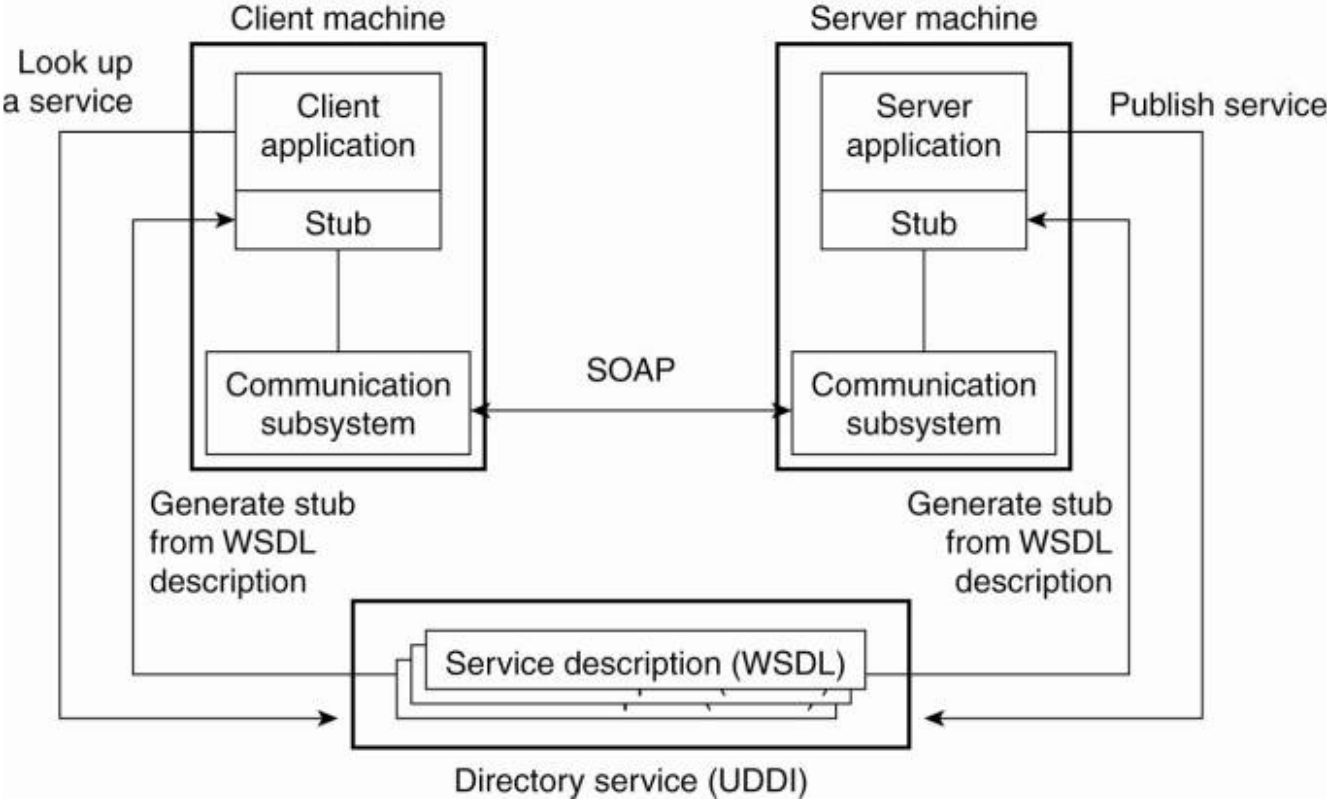
So far, we have implicitly assumed that the client-side software of a Webbased system consists of a browser that acts as the interface to a user. This assumption is no longer universally true anymore. There is a rapidly growing group of Web-based systems that are offering general services to remote applications without immediate interactions from end users. This organization leads to the concept of Web services (Alonso et al., 2004).

Web Services Fundamentals

Simply stated, a Web service is nothing but a traditional service (e.g., a naming service, a weather-reporting service, an electronic supplier, etc.) that is made available over the Internet. What makes a Web service special is that it adheres to a collection of standards that will allow it to be discovered and accessed over the Internet by client applications that follow those standards as well. It should come as no surprise then, that those standards form the core of Web services architecture [see also Booth et al. (2004)].

The principle of providing and using a Web service is quite simple, and is shown in Fig. 12-4. The basic idea is that some client application can call upon the services as provided by a server application. Standardization takes place with respect to how those services are described such that they can be looked up by a client application. In addition, we need to ensure that service call proceeds along the rules set by the server application. Note that this principle is no different from what is needed to realize a remote procedure call.

Figure 12-4. The principle of a Web service.
(This item is displayed on page 552 in the print version)



An important component in the Web services architecture is formed by a directory service storing service descriptions. This service adheres to the Universal Description, Discovery and Integration standard (UDDI). As its name suggests, UDDI prescribes the layout of a database containing service descriptions that will allow Web service clients to browse for relevant services.
[Page 552]

Services are described by means of the Web Services Definition Language (WSDL) which is a formal language very much the same as the interface definition languages used to support RPC-based communication. A WSDL description contains the precise definitions of the interfaces provided by a service, that is, procedure specification, data types, the (logical) location of services, etc. An important issue of a WSDL description is that can be automatically translated to clientside and server-side stubs, again, analogous to the generation of stubs in ordinary RPC-based systems.

Finally, a core element of a Web service is the specification of how communication takes place. To this end, the Simple Object Access Protocol (SOAP) is used, which is essentially a framework in which much of the communication between two processes can be standardized. We will discuss SOAP in detail below, where it will also become clear that calling the framework simple is not really justified.

Web Services Composition and Coordination

The architecture described so far is relatively straightforward: a service is implemented by means of an application and its invocation takes place according to a specific standard. Of course, the application itself may be complex and, in fact, its components may be completely distributed across a local-area network. In such cases, the Web service is most likely implemented by means of an internal proxy or daemon that interacts with the various components constituting the distributed application. In that case, all the principles we have discussed so far can be readily applied as we have discussed.
[Page 553]

In the model so far, a Web service is offered in the form of a single invocation. In practice, much more complex invocation structures need to take place before a service can be considered as completed. For example, take an electronic bookstore. Ordering a book requires selecting a book, paying, and ensuring its delivery. From a service perspective, the actual service should be modeled as a transaction consisting of multiple steps that need to be carried out in a specific order. In other words, we are dealing with a complex service that is built from a number of basic services.

Complexity increases when considering Web services that are offered by combining Web services from different providers. A typical example is devising a Web-based shop. Most shops consist roughly of three parts: a first part by which the goods that a client requires are selected, a second one that handles the payment of those goods, and a third one that takes care of shipping and subsequent tracking of goods. In order to set up such a shop, a provider may want to make use of a electronic bank service that can handle payment, but also a special delivery service that handles the shipping of goods. In this way, a provider can concentrate on its core business, namely the offering of goods.

In these scenarios it is important that a customer sees a coherent service: namely a shop where he can select, pay, and rely on proper delivery. However, internally we need to deal with a situation in which possibly three different organizations need to act in a coordinated way. Providing proper support for such composite services forms an essential element of Web services. There are at least two classes of problems that need to be solved. First, how can the coordination between Web services, possibly from different organizations, take place? Second, how can services be easily composed?

Coordination among Web services is tackled through coordination protocols. Such a protocol prescribes the various steps that need to take place for (composite) service to succeed. The issue, of course, is to enforce the parties taking part in such protocol take the correct steps at the right moment. There are various ways to achieve this; the simplest is to have a single coordinator that controls the messages exchanged between the participating parties.

However, although various solutions exist, from the Web services perspective it is important to standardize the commonalities in coordination protocols. For one, it is important that when a party wants to participate in a specific protocol, that it knows with which other process(es) it should communicate. In addition, it may very well be that a process is involved in multiple coordination protocols at the same time. Therefore, identifying the instance of a protocol is important as well. Finally, a process should know which role it is to fulfill.

These issues are standardized in what is known as Web Services Coordination (Frend et al., 2005). From an architectural point of view, it defines a separate service for handling coordination protocols. The coordination of a protocol is part of this service. Processes can register themselves as participating in the coordination so that their peers know about them.

[Page 554]

To make matters concrete, consider a coordination service for variants of the two-phase protocol (2PC) we discussed in Chap. 8. The whole idea is that such a service would implement the coordinator for various protocol instances. One obvious implementation is that a single process plays the role of coordinator for multiple protocol instances. An alternative is that have each coordinator be implemented by a separate thread.

A process can request the activation of a specific protocol. At that point, it will essentially be returned an identifier that it can pass to other processes for registering as participants in the newly-created protocol instance. Of course, all participating processes will be required to implement the specific interfaces of the protocol that the coordination service is supporting. Once all participants have registered, the coordinator can send the VOTE_REQUEST, COMMIT, and other messages that are part of the 2PC protocol to the participants when needed.

It is not difficult to see that due to the commonality in, for example, 2PC protocols, standardization of interfaces and messages to exchange will make it much easier to compose and coordinate Web services. The actual work that needs to be done is not very difficult. In this respect, the added value of a coordination service is to be sought entirely in the standardization.

Clearly, a coordination service already offers facilities for composing a Web service out of other services. There is only one potential problem: how the service is composed is public. In many cases, this is not a desirable property, as it would allow any competitor to set up exactly the same composite service. What is needed, therefore, are facilities for setting up private coordinators. We will not go into any details here, as they do not touch upon the principles of service composition in Web-based systems. Also, this type of composition is still very much in flux (and may continue to be so for a long time). The interested reader is referred to (Alonso et al., 2004).

12.2. Processes

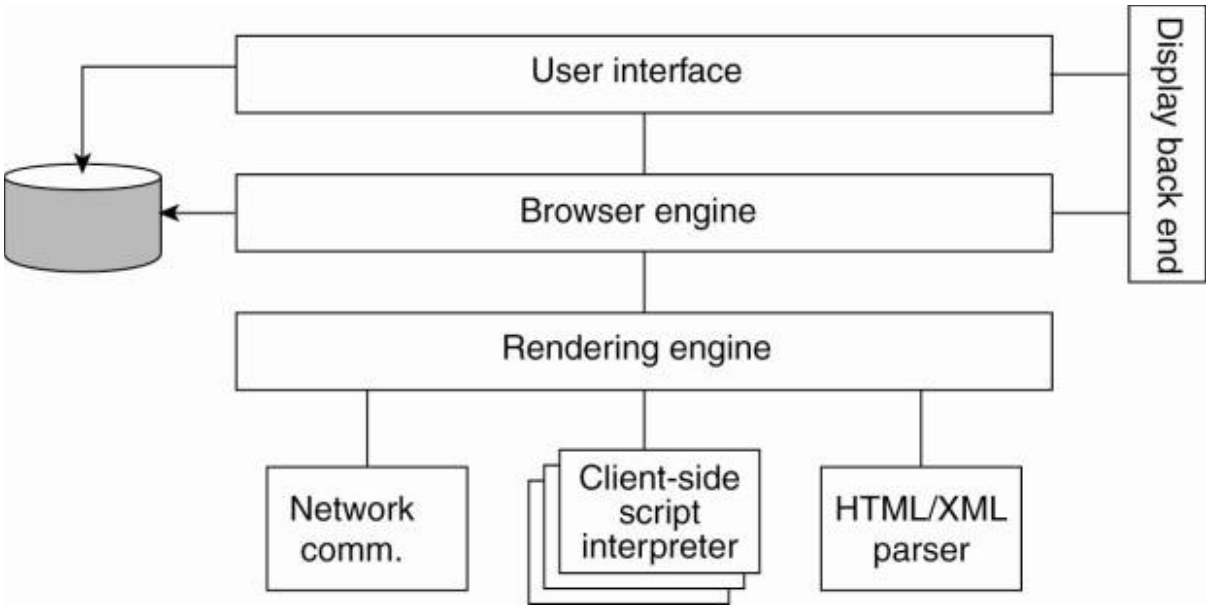
We now turn to the most important processes used in Web-based systems and their internal organization.

12.2.1. 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.
[Page 555]

Web browsers used to be simple programs, but that was long ago. Logically, they consist of several components, shown in Fig. 12-5 [see also Grosskurth and Godfrey (2005)].

Figure 12-5. The logical components of a Web browser.



An important aspect of Web browsers is that they should (ideally) be platform independent. This goal is often achieved by making use of standard graphical libraries, shown as the display back end, along with standard networking libraries.

The core of a browser is formed by the browser engine and the rendering engine. The latter contains all the code for properly displaying documents as we explained before. This rendering at the very least requires parsing HTML or XML, but may also require script interpretation. In most case, there is only an interpreter for Javascript included, but in theory other interpreters may be included as well. The browser engine provides the mechanisms for an end user to go over a document, select parts of it, activate hyperlinks, etc.

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. As mentioned before, 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 normally offer a standard interface to the browser and, likewise, expect a standard interface from the browser. Logically, they form an extension of the rendering engine shown in Fig. 12-5.

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. 12-6. 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 as HTTP.

Figure 12-6. Using a Web proxy when the browser does not speak FTP.
(This item is displayed on page 556 in the print version)

[Page 556]

By now, most Web browsers are capable of supporting a variety of protocols, or can otherwise be dynamically extended to do so, and for that reason do not need proxies. However, proxies are still used for other reasons. For example, a proxy can be configured for filtering requests and responses (bringing it close to an application-level firewall), logging, compression, but most of all caching. We return to proxy caching below. A widely-used Web proxy is Squid, which has been developed as an open-source project. Detailed information on Squid can be found in Wessels (2004).

12.2.2. The Apache Web Server

By far the most popular Web server is Apache, which is estimated to be used to host approximately 70% of all Web sites. Apache is a complex piece of software, and with the numerous enhancements to the types of documents that are now offered in the Web, it is important that the server is highly configurable and extensible, and at the same time largely independent of specific platforms.

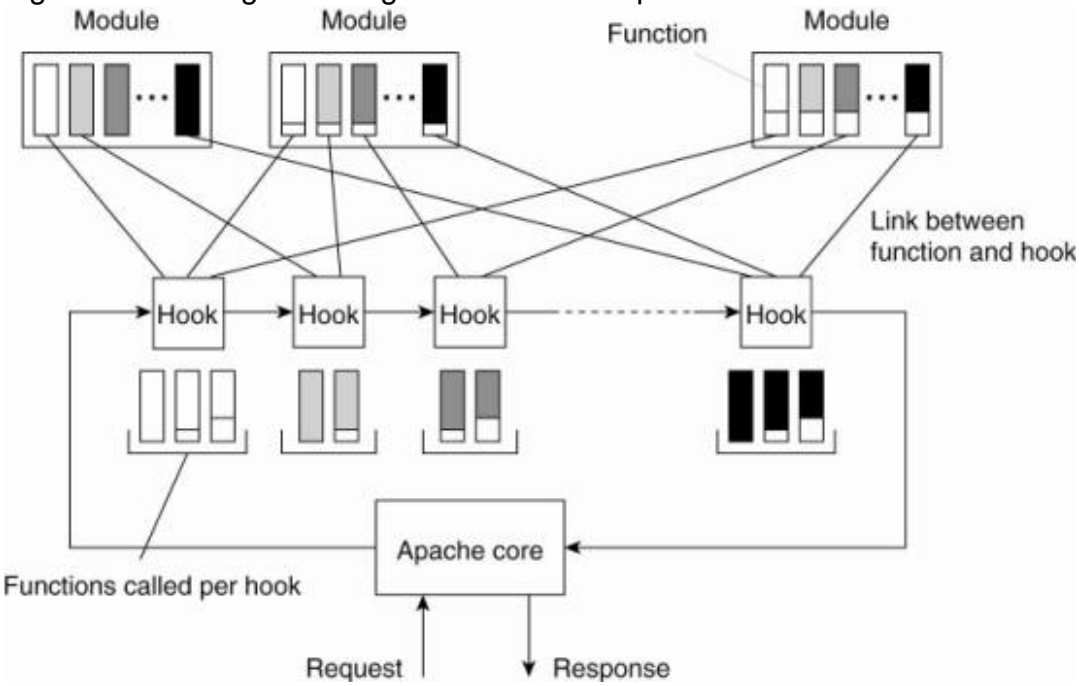
Making the server platform independent is realized by essentially providing its own basic runtime environment, which is then subsequently implemented for different operating systems. This runtime environment, known as the Apache Portable Runtime (APR), is a library that provides a platform-independent interface for file handling, networking, locking, threads, and so on. When extending Apache (as we will discuss shortly), portability is largely guaranteed provided that only calls to the APR are made and that calls to platform-specific libraries are avoided.

As we said, Apache is tailored not only to provide flexibility (in the sense that it can be configured to considerable detail), but also that it is relatively easy to extend its functionality. For example, later in this chapter we will discuss adaptive replication in Globule, a home-brew content delivery network developed in the authors' group at the Vrije Universiteit Amsterdam. Globule is implemented as an extension to Apache, based on the APR, but also largely independent of other extensions developed for Apache.

From a certain perspective, Apache can be considered as a completely general server tailored to produce a response to an incoming request. Of course, there are all kinds of hidden dependencies and assumptions by which Apache turns out to be primarily suited for handling requests for Web documents. For example, as we mentioned, Web browsers and servers use HTTP as their communication protocol. HTTP is virtually always implemented on top of TCP, for which reason the core of Apache assumes that all incoming requests adhere to a TCP-based connectionoriented way of communication. Requests based on, for example, UDP cannot be properly handled without modifying the Apache core.
[Page 557]

However, the Apache core makes few assumptions on how incoming requests should be handled. Its overall organization is shown in Fig. 12-7. Fundamental to this organization is the concept of a hook, which is nothing but a placeholder for a specific group of functions. The Apache core assumes that requests are processed in a number of phases, each phase consisting of a few hooks. Each hook thus represents a group of similar actions that need to be executed as part of processing a request.

Figure 12-7. The general organization of the Apache Web server.



For example, there is a hook to translate a URL to a local file name. Such a translation will almost certainly need to be done when processing a request. Likewise, there is a hook for writing information to a log, a hook for checking a client's identification, a hook for checking access rights, and a hook for checking which MIME type the request is related to (e.g., to make sure that the request can be properly handled). As shown in Fig. 12-7, the hooks are processed in a predetermined order. It is here that we explicitly see that Apache enforces a specific flow of control concerning the processing of requests.

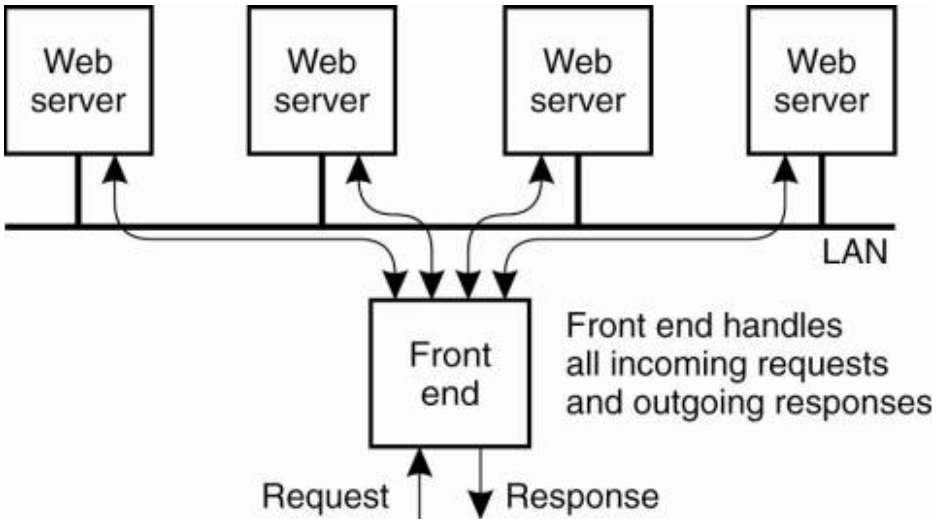
The functions associated with a hook are all provided by separate modules. Although in principle a developer could change the set of hooks that will be processed by Apache, it is far more common to write modules containing the functions that need to be called as part of processing the standard hooks provided by unmodified Apache. The underlying principle is fairly straightforward. Every hook can contain a set of functions that each should match a specific function prototype (i.e., list of parameters and return type). A module developer will write functions for specific hooks. When compiling Apache, the developer specifies which function should be added to which hook. The latter is shown in Fig. 12-7 as the various links between functions and hooks.
[Page 558]

Because there may be tens of modules, each hook will generally contain several functions. Normally, modules are considered to be mutual independent, so that functions in the same hook will be executed in some arbitrary order. However, Apache can also handle module dependencies by letting a developer specify an ordering in which functions from different modules should be processed. By and large, the result is a Web server that is extremely versatile. Detailed information on configuring Apache, as well as a good introduction to how it can be extended can be found in Laurie and Laurie (2002).

12.2.3. Web 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 servers and use a separate mechanism, such as a front end, to redirect client requests to one of the replicas. This principle is shown in Fig. 12-8, and is an example of horizontal distribution as we discussed in Chap. 2.

Figure 12-8. The principle of using a server cluster in combination with a front end to implement a Web service.



A crucial aspect of this organization is the design of the front end. as it can become a serious performance bottleneck, what will all the traffic passing through it. In general, a distinction is made between front ends operating as transportlayer switches, and those that operate at the level of the application layer.
[Page 559]

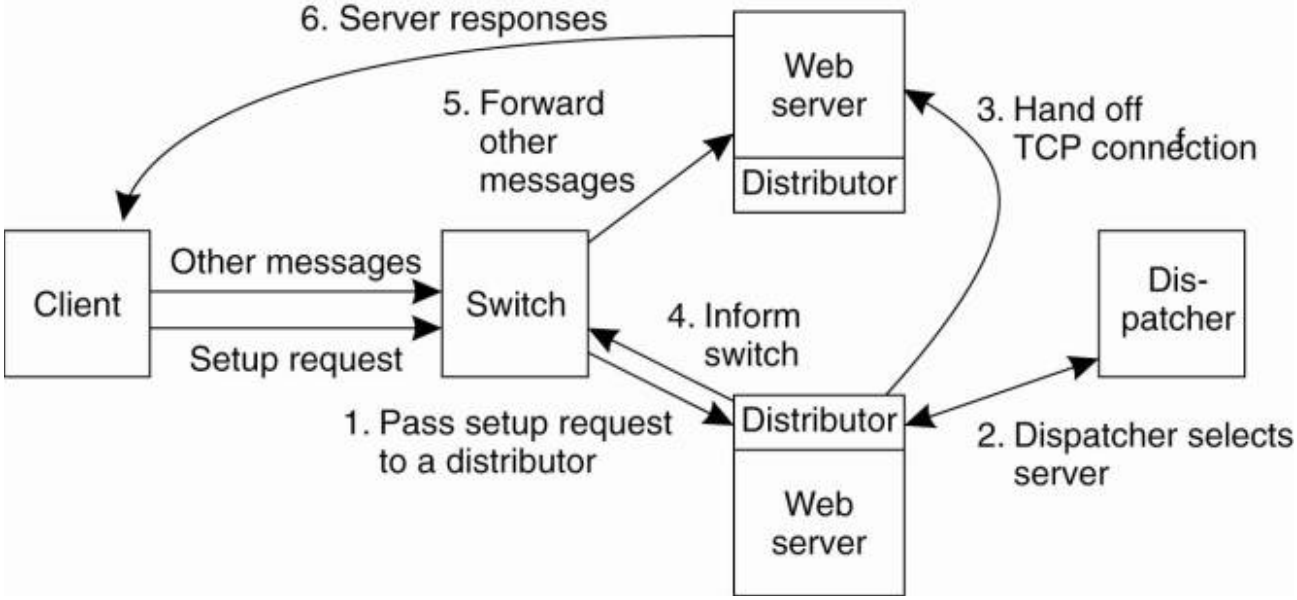
Whenever a client issues an HTTP request, it sets up a TCP connection to the server. A transport-layer switch simply passes the data sent along the TCP connection to one of the servers, depending on some measurement of the server's load. The response from that server is returned to the switch, which will then forward it to the requesting client. As an optimization, the switch and servers can collaborate in implementing a TCP handoff, as we discussed in Chap. 3. The main drawback of a transport-layer switch is that the switch cannot take into account the content of the HTTP request that is sent along the TCP connection. At best, it can only base its redirection decisions on server loads.

As a general rule, 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. 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 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. Ideally, one would like to have the efficiency of TCP handoff and the functionality of content-aware distribution. What we need to do is distribute the work of the front end, and combine that with a transport-layer switch, as proposed 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.
[Page 560]

These two tasks can be distributed as shown in Fig. 12-9. 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 12-9. A scalable content-aware cluster of Web servers.
(This item is displayed on page 559 in the print version)



There are various other alternatives and further refinements for setting up Web server clusters. For example, instead of using any kind of front end, it is also possible to use round-robin DNS by which a single domain name is associated with multiple IP addresses. In this case, when resolving the host name of a Web site, a client browser would receive a list of multiple addresses, each address corresponding to one of the Web servers. Normally, browsers choose the first address on the list. However, what a popular DNS server such as BIND does is circulate the entries of the list it returns (Albitz and Liu, 2001). As a consequence, we obtain a simple distribution of requests over the servers in the cluster.

Finally, it is also possible not to use any sort of intermediate but simply to give each Web server with the same IP address. In that case, we do need to assume that the servers are all connected through a single broadcast LAN. What will happen is that when an HTTP request arrives, the IP router connected to that LAN will simply forward it to all servers, who then run the same distributed algorithm to deterministically decide which of them will handle the request.

The different ways of organizing Web clusters and alternatives like the ones we discussed above, are described in an excellent survey by Cardellini et al., (2002). The interested reader is referred to their paper for further details and references.

12.3. Communication

When it comes to Web-based distributed systems, there are only a few communication protocols that are used. First, for traditional Web systems, HTTP is the standard protocol for exchanging messages. Second, when considering Web services, SOAP is the default way for message exchange. Both protocols will be discussed in a fair amount of detail in this section.

12.3.1. Hypertext Transfer Protocol

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. HTTP is described in Fielding et al. (1999).
[Page 561]

HTTP Connections

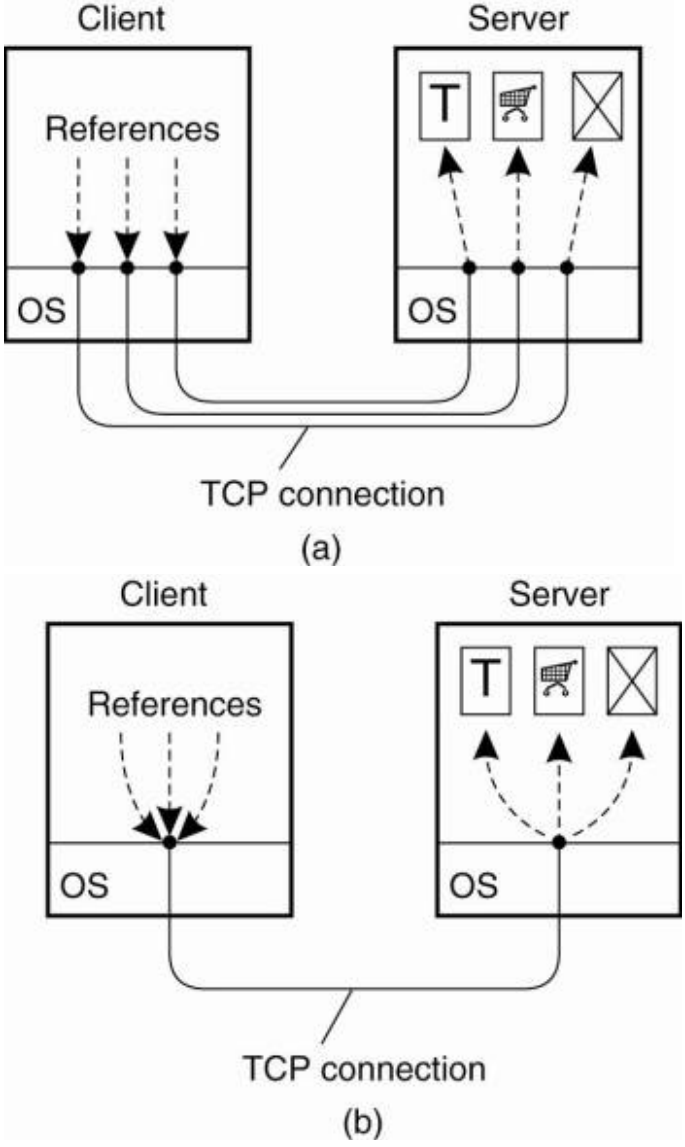
HTTP is based on TCP. Whenever a client issues a request to a server, it first sets up a TCP connection to the server and then sends its request message on that connection. The same connection is used for receiving the response. By using TCP as its underlying protocol, HTTP need not be concerned about 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. 12-10(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 12-10. (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. Many browsers use this approach to improve performance.
[Page 562]

Another approach that is followed in HTTP version 1.1 is to make use of a persistent connection, which can be used to issue several requests (and their respective responses), without the need for a separate connection for each (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. 12-10(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. 12-11.

Figure 12-11. Operations supported by HTTP.

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

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 also 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, (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.
[Page 563]

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

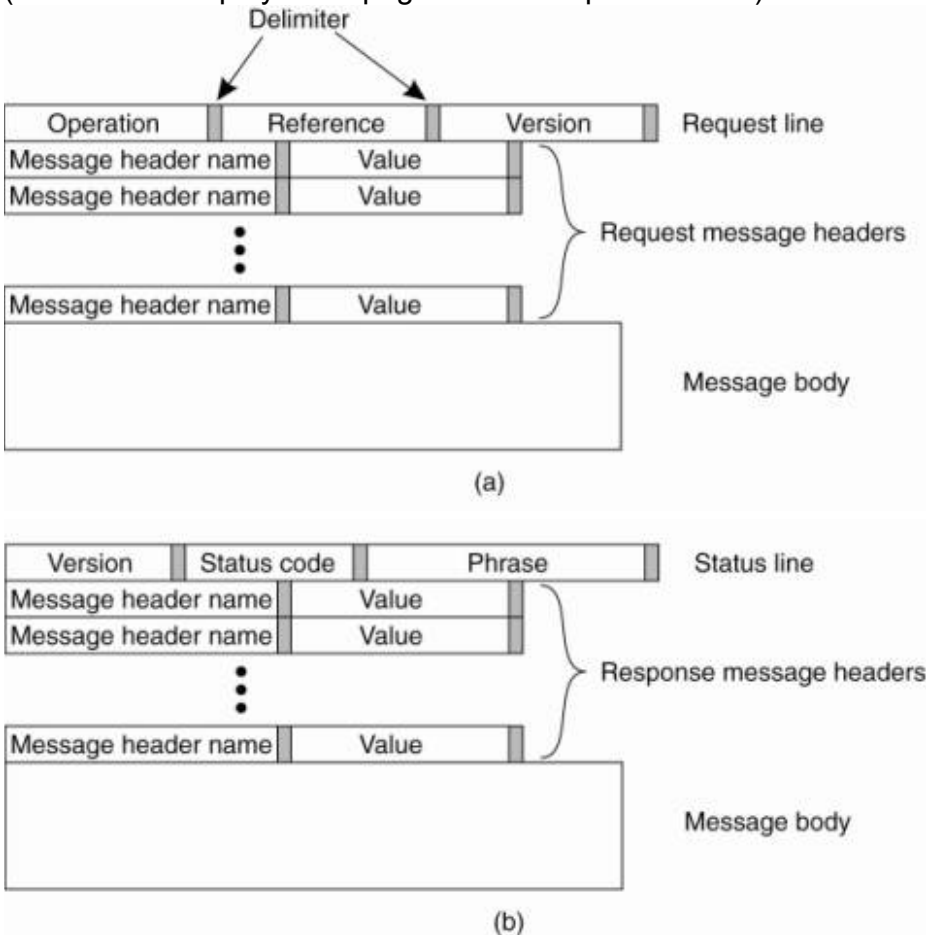
article to a news group. The distinguishing feature, compared to a put operation is that a post operation tells to which group of documents an article should be "added." The article is sent along with the request. In contrast, a put operation carries a document and the name under which the server is 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. 12-12(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 12-12. (a) HTTP request message. (b) HTTP response message.
(This item is displayed on page 564 in the print version)



A response message starts with a status line containing a version number and also a three-digit status code, as shown in Fig. 12-12(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.
[Page 564]

Fig. 12-13 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.

Figure 12-13. Some HTTP message headers.
(This item is displayed on page 565 in the print version)

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 DNS name 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

There are various message 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 program available on most Windows and UNIX machines. In that case, the client will send an Accept-Encoding message header along with its request, with its content containing "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. 5. 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.
[Page 565]

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."
[Page 566]

12.3.2. Simple Object Access Protocol

Where HTTP is the standard communication protocol for traditional Webbased distributed systems, the Simple Object Access Protocol (SOAP) forms the standard for communication with Web services (Gudgin et al., 2003). SOAP has made HTTP even more important than it already was: most SOAP communications are implemented through HTTP. SOAP by itself is not a difficult protocol. Its main purpose is to provide a relatively simple means to let different parties who may know very little of each other be able to communicate. In other words, the protocol is designed with the assumption that two communicating parties have very little common knowledge.

Based on this assumption, it should come as no surprise that SOAP messages are largely based on XML. Recall that XML is a meta-markup language, meaning that an XML description includes the definition of the elements that are used to describe a document. In practice, this means that the definition of the syntax as used for a message is part of that message. Providing this syntax allows a receiver to parse very different types of messages. Of course, the meaning of a message is still left undefined, and thus also what actions to take when a message comes in. If the receiver cannot make any sense out of the contents of a message, no progress can be made.

A SOAP message generally consists of two parts, which are jointly put inside what is called a SOAP envelope. The body contains the actual message, whereas the header is optional, containing information relevant for nodes along the path from sender to receiver. Typically, such nodes consist of the various processes in a multitiered implementation of a Web service. Everything in the envelope is expressed in XML, that is, the header and the body.

Strange as it may seem, a SOAP envelope does not contain the address of the recipient. Instead, SOAP explicitly assumes that the recipient is specified by the protocol that is used to transfer messages. To this end, SOAP specifies bindings to underlying transfer protocols. At present, two such bindings exist: one to HTTP and one to SMTP, the Internet mail-transfer protocol. So, for example, when a SOAP message is bound to HTTP, the recipient will be specified in the form of a URL, whereas a binding to SMTP will specify the recipient in the form of an email address.

These two different types of bindings also indicate two different styles of interactions. The first, most common one, is the conversational exchange style. In this style, two parties essentially exchange structured documents. For example, such a document may contain a complete purchase order as one would fill in when electronically booking a flight. The response to such an order could be a confirmation document, now containing an order number, flight information, a seat reservation, and perhaps also a bar code that needs to be scanned when boarding.

In contrast, an RPC-style exchange adheres closer to the traditional requestresponse behavior when invoking a Web service. In this case, the SOAP message will identify explicitly the procedure to be called, and also provide a list of parameter values as input to that call. Likewise, the response will be a formal message containing the response to the call.
[Page 567]

Typically, an RPC-style exchange is supported by a binding to HTTP, whereas a conversational style message will be bound to either SMTP or HTTP. However, in practice, most SOAP messages are sent over HTTP.

An important observation is that, although XML makes it much easier to use a general parser because syntax definitions are now part of a message, the XML syntax itself is extremely verbose. As a result, parsing XML messages in practice often introduces a serious performance bottleneck (Allman, 2003). In this respect, it is somewhat surprising that improving XML performance receives relatively little attention, although solutions are underway (see, e.g., Kostoulas et al., 2006).

What is equally surprising is that many people believe that XML specifications can be conveniently read by human beings. The example shown in Fig. 12-14 is taken from the official SOAP specification (Gudgin et al., 2003). Discovering what this SOAP message conveys requires some searching, and it is not hard to imagine that obscurity in general may come as a natural by-product of using XML. The question then comes to mind, whether the text-based approach as followed for XML has been the right one: no one can conveniently read XML documents, and parsers are severely slowed down.

Figure 12-14. An example of an XML-based SOAP message.

```
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
   <env:Header>
```

```
      <n:alertcontrol xmlns:n="http://example.org/alertcontrol">
        <n:priority>1</n:priority>
        <n:expires>2001-06-22T14:00:00-05:00</n:expires>
      </n:alertcontrol>
    </env:Header>
    <env:Body>
      <m:alert xmlns:m="http://example.org/alert">
        <m:msg>Pick up Mary at school at 2pm</m:msg>
      </m:alert>
    </env:Body>
</env:Envelope>
```

12.4. Naming

The Web uses a single naming system to refer to documents. The names used are called Uniform Resource Identifiers or simply URIs (Berners-Lee et al., 2005). 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. 5. A URN is used as a globally unique, location-independent, and persistent reference to a document.
[Page 568]

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. We should also note that the difference between URL and URN is gradually diminishing. Instead, it is now common to simply define URI name spaces [see also Daigle et al. (2002)].

In the case of URLs, we see that they often contain information on how and where to access a document. How to access a document is generally 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 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. 12-15.

Figure 12-15. 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.

Scheme		Host name	Pathname
http	://	www.cs.vu.nl	/home/steen/mbox

(a)

Scheme		Host name		Port	Pathname
http	://	www.cs.vu.nl	:	80	/home/steen/mbox

(b)

Scheme		Host name		Port	Pathname
http	://	130.37.24.11	:	80	/home/steen/mbox

(c)

Resolving a URL such as those shown in Fig. 12-15 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. [Page 569]

Although URLs are still commonplace in the Web, various separate URI name spaces have been proposed for other kinds of Web resources. Fig. 12-16 shows a number of examples of URIs. The http URI is used to transfer documents using HTTP as we explained above. Likewise, there is an ftp URI for file transfer using FTP.

Figure 12-16. Examples of URIs.

Name	Used for	Example
http	HTTP	http://www.cs.vu.nl:80/globe
mailto	E-mail	mailto:steen@cs.vu.nl
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

An immediate form of documents is supported by data URIs (Masinter, 1998). In such a URI, the document itself is embedded in the URI, similar to embedding the data of a file in an inode (Mullender and Tanenbaum, 1984). The example shows a URI containing plain text for the Greek character string αβγ.

URIs are often used as well for purposes other than referring to a document. For example, a telnet URI is used for setting up a telnet session to a server. There are also URIs for telephone-based communication as described in Schulzrinne (2005). The tel URI as shown in Fig. 12-16 essentially embeds only a telephone number and simply lets the client to establish a call across the telephone network. In this case, the client will typically be a telephone. The modem URI can be used to set up a modem-based connection with another computer. In the example, the URI states that the remote modem should adhere to the ITU-T V32 standard.

12.5. Synchronization

Synchronization has not been much of an issue for most traditional Webbased systems 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 or entity, and hardly ever introduce write-write conflicts.

However, things are changing. For example, 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. Likewise, with the introduction of Web services, we are now seeing a need for servers to synchronize with each other and that their actions are coordinated. We already discussed coordination in Web services above. We therefore briefly pay some attention to synchronization for collaborative maintenance of Web documents.
[Page 570]

Distributed authoring of Web documents is handled through a separate protocol, namely WebDAV (Goland et al., 1999). 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. An overview of how WebDAV can be used in a practical setting is provided in Kim et al. (2004).

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.

12.6. Consistency and Replication

Perhaps one of the most important systems-oriented developments in Webbased distributed systems is ensuring that access to Web documents meets stringent performance and availability requirements. These requirements have led to numerous proposals for caching and replicating Web content, of which various ones will be discussed in this section. Where the original schemes (which are still largely deployed) have been targeted toward supporting static content, much effort is also being put into support dynamic content, that is, supporting documents that are generated as the result of a request, as well as those containing scripts and such. An excellent and complete picture of Web caching and replication is provided by Rabinovich and Spatscheck (2002).
[Page 571]

12.6.1. 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.

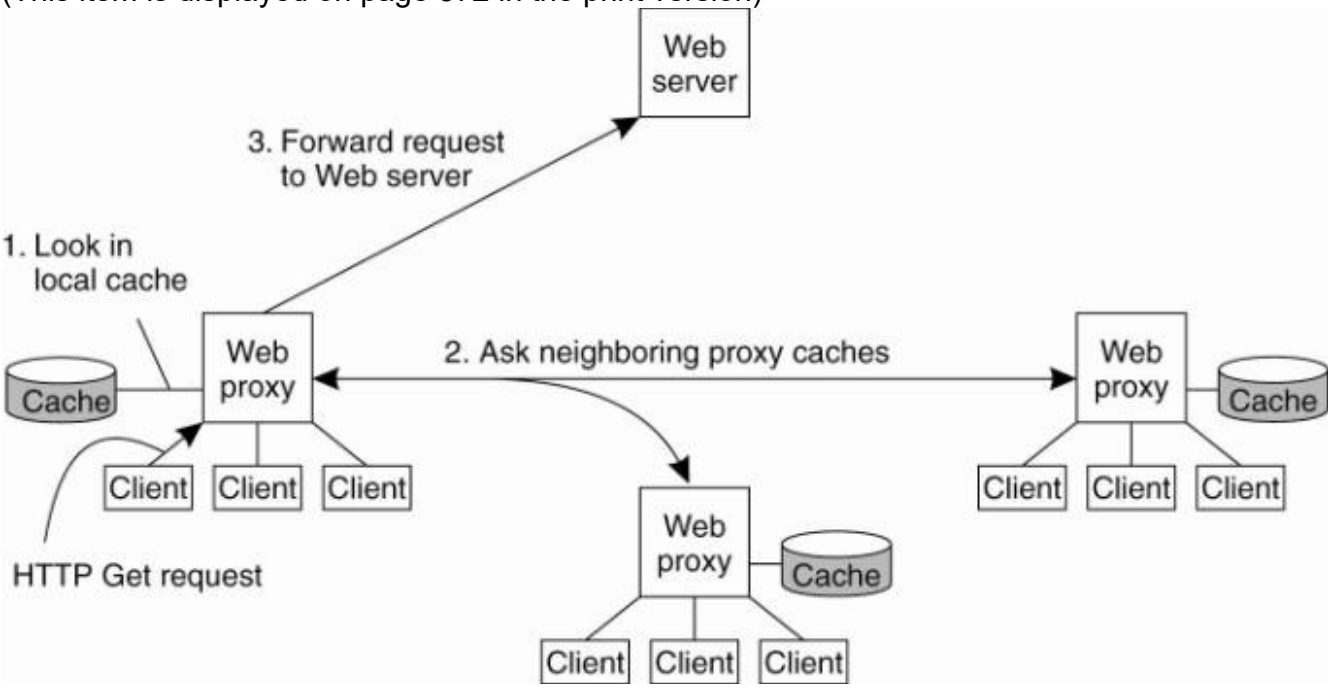
In the second place, a client's site often runs a Web proxy. As we explained, a Web proxy accepts requests from local clients and passes these to Web servers. When a response comes in, the result is passed to the client. The advantage of this approach is that the proxy can cache the result and return 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 hierarchical caches. Such schemes are mainly used to reduce network traffic, but have the disadvantage of potentially incurring a higher latency compared to using nonhierarchical schemes. This higher latency is caused by the need for the client to check multiple caches rather than just one in the nonhierarchical scheme. However, this higher latency is strongly related to the popularity of a document: for popular documents, the chance of finding a copy in a cache closer to the client is higher than for a unpopular document.

As an alternative to building hierarchical caches, one can also organize caches for cooperative deployment as shown in Fig. 12-17. In cooperative caching or distributed 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. This scheme is primarily deployed with Web caches belonging to the same organization or institution that are colocated in the

same LAN. It is interesting to note that a study by Wolman et al. (1999) shows that cooperative caching may be effective for only 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.

Figure 12-17. The principle of cooperative caching.
(This item is displayed on page 572 in the print version)



A comparison between hierarchical and cooperative caching by Rodriguez et al. (2001) makes clear that there are various trade-offs to make. For example, because cooperative caches are generally connected through high-speed links, the transmission time needed to fetch a document is much lower than for a hierarchical cache. Also, as is to be expected, storage requirements are less strict for cooperative caches than hierarchical ones. Also, they find that expected latencies for hierarchical caches are lower than for distributed caches.
[Page 572]

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. 7, 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 widelyused Squid Web proxy (Wessels, 2004) assigns an expiration time Texpire that depends on how long ago the document was last modified when it is cached. In particular, if Tlast_modified is the last

modification time of a document (as recorded by its owner), and Tcached 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 Texpire, 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.
[Page 573]

Note that documents that have not been modified for a long time will not be checked for modifications as soon as recently modified documents. 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. Note that this state is largely dictated by the expiration times set for leases: the lower, the less caches a server needs to keep track of. Nevertheless, invalidation protocols for Web proxy caches are hardly ever applied.

A comparison of Web caching consistency policies can be found in Cao and Oszu (2002). Their conclusion is that letting the server send invalidations can outperform any other method in terms of bandwidth and perceived client latency, while maintaining cached documents consistent with those at the origin server. These findings hold for access patterns as often observed for electronic commerce applications.

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. These dynamically generated documents are often unique in the sense that the same request from a client will presumably lead to a different response the next time. For example, many documents contain advertisements (called banners) which change for every request made. We return to this situation below when we discuss caching and replication for Web applications.

Finally, we should also mention that much research has been conducted to find out what the best cache replacement strategies are. Numerous proposals exist, but by-and-large, simple replacement strategies such as evicting the least recently used object work well enough. An in-depth survey of replacement strategies is presented in Podling and Boszormenyi (2003).

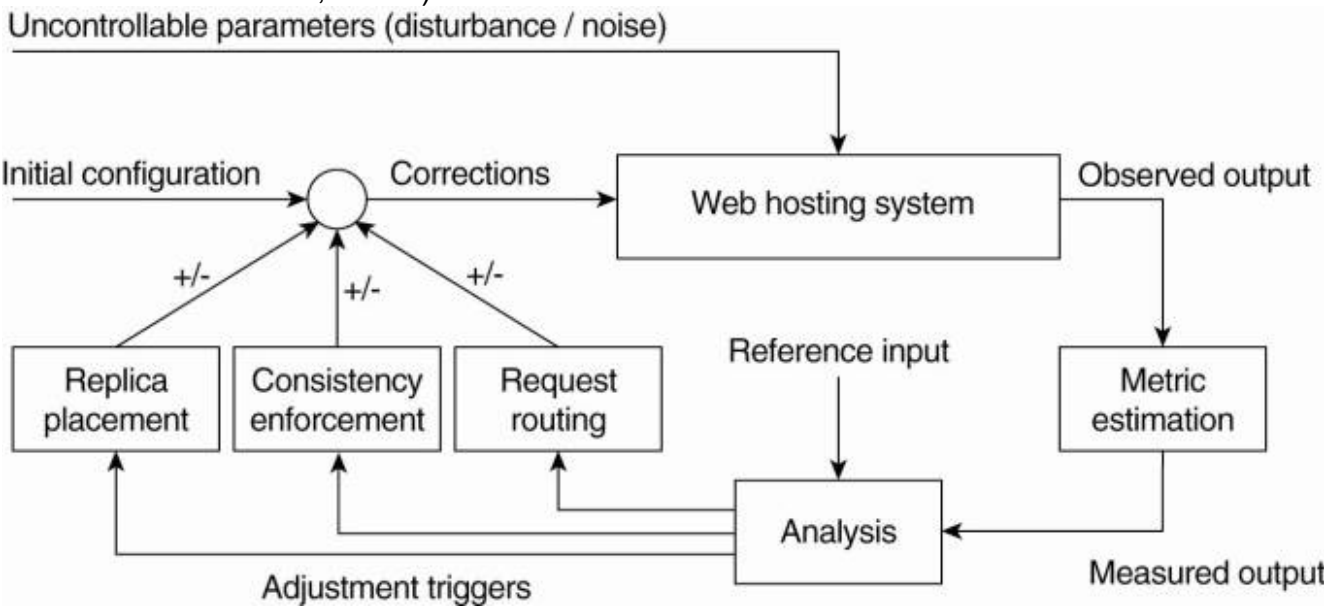
12.6.2. Replication for Web Hosting Systems

As the importance of the Web continues to increase as a vehicle for organizations to present themselves and to directly interact with end users, we see a shift between maintaining the content of a Web site and making sure that the site is easily and continuously accessible. This distinction has paved the way for content delivery networks (CDNs). The main idea underlying

these CDNs is that they act as a Web hosting service, providing an infrastructure for distributing and replicating the Web documents of multiple sites across the Internet. The size of the infrastructure can be impressive. For example, as of 2006, Akamai is reported to have over 18,000 servers spread across 70 countries.
[Page 574]

The sheer size of a CDN requires that hosted documents are automatically distributed and replicated, leading to the architecture of a self-managing system as we discussed in Chap. 2. In most cases, a large-scale CDN is organized along the lines of a feedback-control loop, as shown in Fig. 12-18 and which is described extensively in Sivasubramanian et al. (2004b).

Figure 12-18. The general organization of a CDN as a feedback-control system (adapted from Sivasubramanian et al., 2004b).



There are essentially three different kinds of aspects related to replication in Web hosting systems: metric estimation, adaptation triggering, and taking appropriate measures. The latter can be subdivided into replica placement decisions, consistency enforcement, and client-request routing. In the following, we briefly pay attention to each these.
Metric Estimation

An interesting aspect of CDNs is that they need to make a trade-off between many aspects when it comes to hosting replicated content. For example, access times for a document may be optimal if a document is massively replicated, but at the same time this incurs a financial cost, as well as a cost in terms of bandwidth usage for disseminating updates. By and large, there are many proposals for estimating how well a CDN is performing. These proposals can be grouped into several classes.

First, there are latency metrics, by which the time is measured for an action, for example, fetching a document, to take place. Trivial as this may seem, estimating latencies becomes difficult when, for example, a process deciding on the placement of replicas needs to know the

delay between a client and some remote server. Typically, an algorithm globally positioning nodes as discussed in Chap. 6 will need to be deployed.
[Page 575]

Instead of estimating latency, it may be more important to measure the available bandwidth between two nodes. This information is particularly important when large documents need to be transferred, as in that case the responsiveness of the system is largely dictated by the time that a document can be transferred. There are various tools for measuring available bandwidth, but in all cases it turns out that accurate measurements can be difficult to attain. Further information can be found in Strauss et al. (2003).

Another class consists of spatial metrics which mainly consist of measuring the distance between nodes in terms of the number of network-level routing hops, or hops between autonomous systems. Again, determining the number of hops between two arbitrary nodes can be very difficult, and may also not even correlate with latency (Huffaker et al., 2002). Moreover, simply looking at routing tables is not going to work when low-level techniques such as multi-protocol label switching (MPLS) are deployed. MPLS circumvents network-level routing by using virtual-circuit techniques to immediately and efficiently forward packets to their destination [see also Guichard et al. (2005)]. Packets may thus follow completely different routes than advertised in the tables of network-level routers.

A third class is formed by network usage metrics which most often entails consumed bandwidth. Computing consumed bandwidth in terms of the number of bytes to transfer is generally easy. However, to do this correctly, we need to take into account how often the document is read, how often it is updated, and how often it is replicated. We leave this as an exercise to the reader.

Consistency metrics tell us to what extent a replica is deviating from its master copy. We already discussed extensively how consistency can be measured in the context of continuous consistency in Chap. 7 (Yu and Vahdat, 2002).

Finally, financial metrics form another class for measuring how well a CDN is doing. Although not technical at all, considering that most CDN operate on a commercial basis, it is clear that in many cases financial metrics will be decisive. Moreover, the financial metrics are closely related to the actual infrastructure of the Internet. For example, most commercial CDNs place servers at the edge of the Internet, meaning that they hire capacity from ISPs directly servicing end users. At this point, business models become intertwined with technological issues, an area that is not at all well understood. There is only few material available on the relation between financial performance and technological issues (Janiga et al., 2001).

From these examples it should become clear that simply measuring the performance of a CDN, or even estimating its performance may by itself be an extremely complex task. In practice, for commercial CDNs the issue that really counts is whether they can meet the service-level agreements that have been made with customers. These agreements are often formulated simply in terms of how quickly customers are to be serviced. It is then up to the CDN to make sure that these agreements are met.
[Page 576]

Adaptation Triggering

Another question that needs to be addressed is when and how adaptations are to be triggered. A simple model is to periodically estimate metrics and subsequently take measures as needed. This approach is often seen in practice. Special processes located at the servers collect information and periodically check for changes.

A major drawback of periodic evaluation is that sudden changes may be missed. One type of sudden change that is receiving considerable attention is that of flash crowds. A flash crowd is a sudden burst in requests for a specific Web document. In many cases, these type of bursts can bring down an entire service, in turn causing a cascade of service outages as witnessed during several events in the recent history of the Internet.

Handling flash crowds is difficult. A very expensive solution is to massively replicate a Web site and as soon as request rates start to rapidly increase, requests should be redirected to the replicas to offload the master copy. This type of over-provisioning is obviously not the way to go. Instead, what is needed is a flash-crowd predictor that will provide a server enough time to dynamically install replicas of Web documents, after which it can redirect requests when the going gets tough. One of the problems with attempting to predict flash crowds is that they can be so very different. Fig. 12-19 shows access traces for four different Web sites that suffered from a flash crowd. As a point of reference, Fig. 12-19(a) shows regular access traces spanning two days. There are also some very strong peaks, but otherwise there is nothing shocking going on. In contrast, Fig. 12-19(b) shows a two-day trace with four sudden flash crowds. There is still some regularity, which may be discovered after a while so that measures can be taken. However, the damage may be been done before reaching that point.

Figure 12-19. One normal and three different access patterns reflecting flashcrowd behavior (adapted from Baryshnikov et al., 2005).
(This item is displayed on page 577 in the print version)

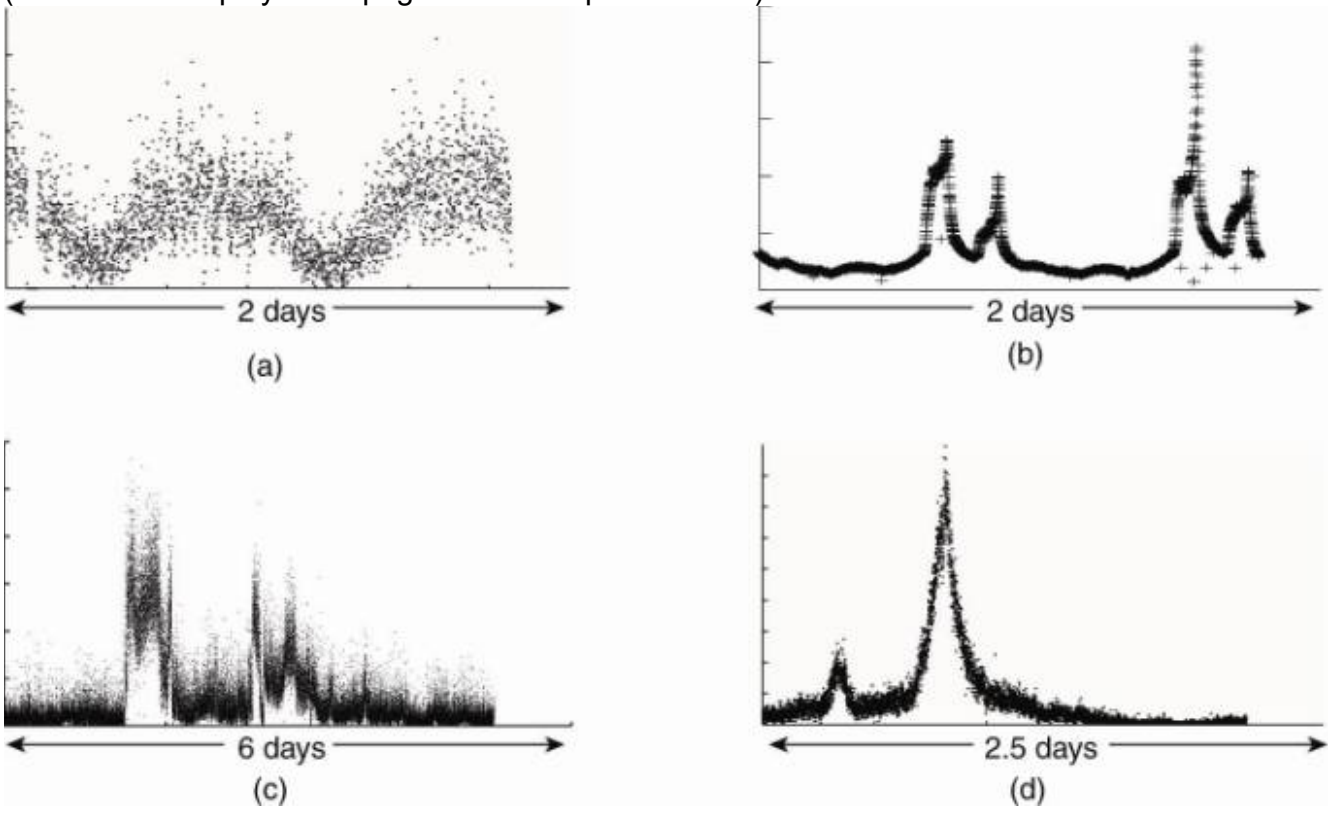


Fig. 12-19(c) shows a trace spanning six days with at least two flash crowds. In this case, any predictor is going to have a serious problem, as it turns out that both increases in request rate are almost instantaneously. Finally, Fig. 12-19(d) shows a situation in which the first peak should probably cause no adaptations, but the second obviously should. This situation turns out to be the type of behavior that can be dealt with quite well through runtime analysis.

One promising method to predict flash crowds is using a simple linear extrapolation technique. Baryshikov et al. (2005) propose to continuously measure the number of requests to a document during a specific time interval [t - W,t), where W is the window size. The interval itself is divided into small slots, where for each slot the number of requests are counted. Then, by applying simple linear regression, we can fit a curve ft expressing the number of accesses as a function of time. By extrapolating the curve to time instances beyond t, we obtain a prediction for the number of requests. If the number of requests are predicted to exceed a given threshold, an alarm is raised.
[Page 577]

This method works remarkably well for multple access patterns. Unfortunately, the window size as well as determining what the alarm threshold are supposed to be depends highly on the Web server traffic. In practice, this means that much manual fine tuning is needed to configure an ideal predictor for a specific site. It is yet unknown how flash-crowd predictors can be automatically configured.

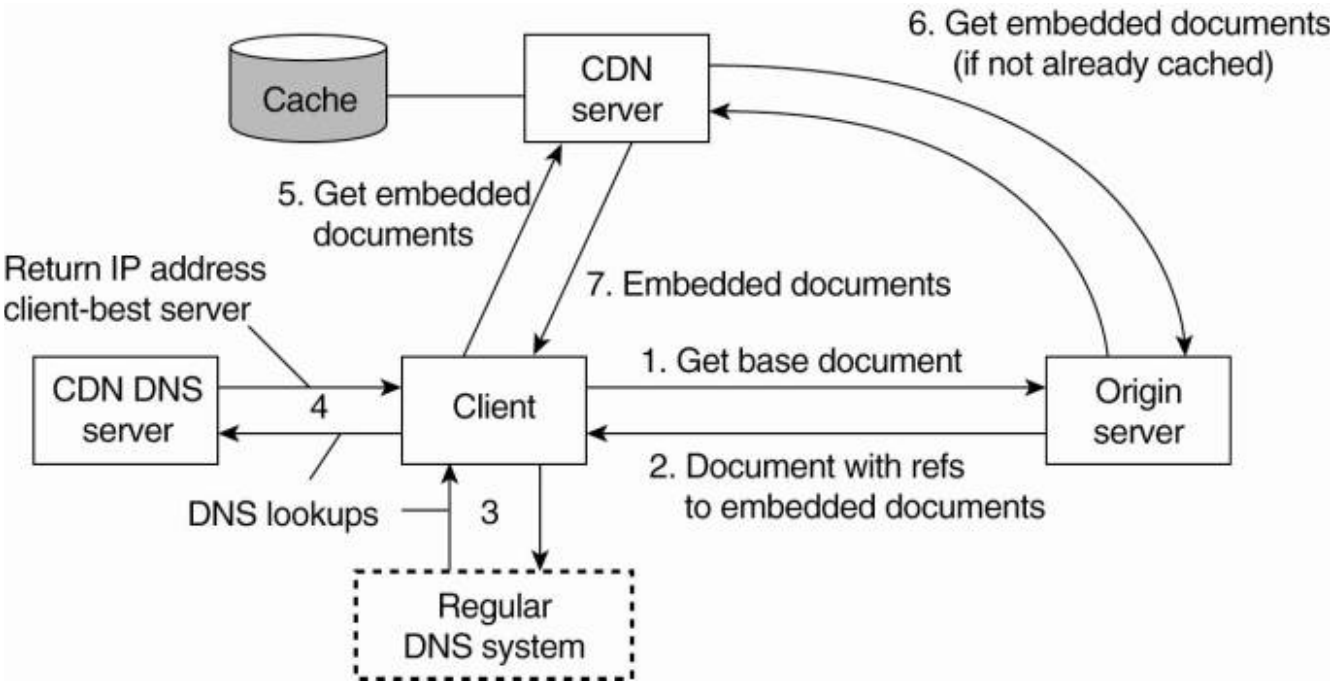
Adjustment Measures

As mentioned, there are essentially only three (related) measures that can be taken to change the behavior of a Web hosting service: changing the placement of replicas, changing consistency enforcement, and deciding on how and when to redirect client requests. We already discussed the first two measures extensively in Chap. 7. Client-request redirection deserves some more attention. Before we discuss some of the trade-offs, let us first consider how consistency and replication are dealt with in a practical setting by considering the Akamai situation (Leighton and Lewin, 2000; and Dilley et al., 2002).

The basic idea is that each Web document consists of a main HTML (or XML) 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.
[Page 578]

Each embedded document is normally referenced through a URL. However, 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 URL also contains the host name of the origin server for reasons we explain next. The modified URL is resolved as follows, as is also shown in Fig. 12-20.

Figure 12-20. The principal working of the Akamai CDN.



The name of the virtual ghost includes a DNS name such as ghosting.com, which is resolved by the regular DNS naming system to a CDN DNS server (the result of step 3). Each such DNS server keeps track of servers close to the client. To this end, any of the proximity metrics we have discussed previously could be used. In effect, the CDN DNS servers redirects the client to a replica server best for that client (step 4), which could mean the closest one, the least-loaded one, or a combination of several such metrics (the actual redirection policy is proprietary).

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 6), 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 forthwith. Note that in order to fetch the embedded document, the replica server must be able to send a request to the origin server, for which reason its host name is also contained in the embedded document's URL.

An interesting aspect of this scheme is the simplicity by which consistency of documents can be enforced. Clearly, whenever a main document is changed, a client will always be able to fetch it from the origin server. In the case of embedded documents, a different approach needs to be followed as these documents are, in principle, fetched from a nearby replica server. To this end, a URL for an embedded document not only refers to a special host name that eventually leads to a CDN DNS server, but also contains a unique identifier that is changed every time the embedded document changes. In effect, this identifier changes the name of the embedded document. As a consequence, when the client is redirected to a specific CDN server, that server will not find the named document in its cache and will thus fetch it from the origin server. The old document will eventually be evicted from the server's cache as it is no longer referenced.
[Page 579]

This example already shows the importance of client-request redirection. In principle, by properly redirecting clients, a CDN can stay in control when it comes to client-perceived performance, but also taking into account global system performance by, for example, avoiding that requests are sent to heavily loaded servers. These so-called adaptive redirection policies can be applied when information on the system's current behavior is provided to the processes that take redirection decisions. This brings us partly back to the metric estimation techniques discussed previously.

Besides the different policies, an important issue is whether request redirection is transparent to the client or not. In essence, there are only three redirection techniques: TCP handoff, DNS redirection, and HTTP redirection. We already discussed TCP handoff. This technique is applicable only for server clusters and does not scale to wide-area networks.

DNS redirection is a transparent mechanism by which the client can be kept completely unaware of where documents are located. Akamai's two-level redirection is one example of this technique. We can also directly deploy DNS to return one of several addresses as we discussed before. Note, however, that DNS redirection can be applied only to an entire site: the name of individual documents does not fit into the DNS name space.

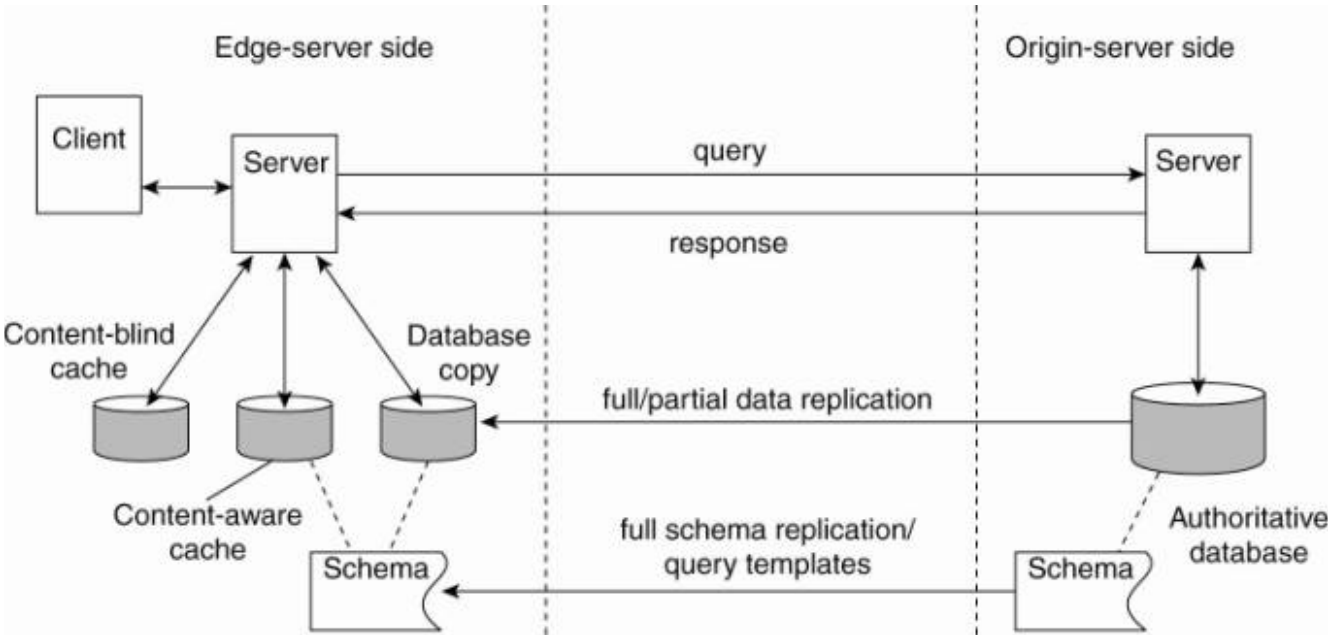
HTTP redirection, finally, is a nontransparent mechanism. When a client requests a specific document, it may be given an alternative URL as part of an HTTP response message to which it is then redirected. An important observation is that this URL is visible to the client's browser. In fact, the user may decide to bookmark the referral URL, potentially rendering the redirection policy useless.

12.6.3. Replication of Web Applications

Up to this point we have mainly concentrated on caching and replicating static Web content. In practice, we see that the Web is increasingly offering more dynamically generated content, but that it is also expanding toward offering services that can be called by remote applications. Also in these situations we see that caching and replication can help considerably in improving the overall performance, although the methods to reach such improvements are more subtle than what we discussed so far [see also Conti et al. (2005)].

When considering improving performance of Web applications through caching and replication, matters are complicated by the fact that several solutions can be deployed, with no single one standing out as the best. Let us consider the edge-server situation as sketched in Fig. 12-21. In this case, we assume a CDN in which each hosted site has an origin server that acts as the authoritative site for all read and update operations. An edge server is used to handle client requests, and has the ability to store (partial) information as also kept at an origin server.

Figure 12-21. Alternatives for caching and replication with Web applications.



Recall that in an edge-server architecture, Web clients request data through an edge server, which, in turn, gets its information from the origin server associated with the specific Web site referred to by the client. As also shown in Fig. 12-21, we assume that the origin server consists of a database from which responses are dynamically created. Although we have shown only a single Web server, it is common to organize each server according to a multitiered architecture as we discussed before. An edge server can now be roughly organized along the following lines.

First, to improve performance, we can decide to apply full replication of the data stored at the origin server. This scheme works well whenever the update ratio is low and when queries require an extensive database search. As mentioned above, we assume that all updates are carried out at the origin server, which takes responsibility for keeping the replicas and the edge servers in a consistent state. Read operations can thus take place at the edge servers. Here we see that replicating for performance will fail when the update ratio is high, as each update will incur communication over a wide-area network to bring the replicas into a consistent state. As shown in Sivasubramanian et al. (2004a), the read/update ratio is the determining factor to what extent the origin database in a wide-area setting should be replicated.
[Page 581]

Another case for full replication is when queries are generally complex. In the case of a relational database, this means that a query requires that multiple tables need to be searched and processed, as is generally the case with a join operation. Opposed to complex queries are simple ones that generally require access to only a single table in order to produce a response. In the latter case, partial replication by which only a subset of the data is stored at the edge server may suffice.

The problem with partial replication is that it may be very difficult to manually decide which data is needed at the edge server. Sivasubramanian et al. (2005) propose to handle this automatically by replicating records according to the same principle that Globule replicates its

Web pages. As we discussed in Chap. 2, this means that an origin server analyzes access traces for data records on which it subsequently bases its decision on where to place records. Recall that in Globule, decision-making was driven by taking the cost into account for executing read and update operations once data was in place (and possibly replicated). Costs are expressed in a simple linear function:

cost=(w1 x m1) + (w2 x m2)+ . . .+(wn x mn)

with mk being a performance metric (such as consumed bandwidth) and wk > 0 the relative weight indicating how important that metric is.

An alternative to partial replication is to make use of content-aware caches. The basic idea in this case is that an edge server maintains a local database that is now tailored to the type of queries that can be handled at the origin server. To explain, in a full-fledged database system a query will operate on a database in which the data has been organized into tables such that, for example, redundancy is minimized. Such databases are also said to be normalized.

In such databases, any query that adheres to the data schema can, in principle, be processed, although perhaps at considerable costs. With content-aware caches, an edge server maintains a database that is organized according to the structure of queries. What this means is that queries are assumed to adhere to a limited number of templates, effectively meaning that the different kinds of queries that can be processed is restricted. In these cases, whenever a query is received, the edge server matches the query against the available templates, and subsequently looks in its local database to compose a response, if possible. If the requested data is not available, the query is forwarded to the origin server after which the response is cached before returning it to the client.

In effect, what the edge server is doing is checking whether a query can be answered with the data that is stored locally. This is also referred to as a query containment check. Note that such data was stored locally as responses to previously issued queries. This approach works best when queries tend to be repeated.
[Page 582]

Part of the complexity of content-aware caching comes from the fact that the data at the edge server needs to be kept consistent. To this end, the origin server needs to know which records are associated with which templates, so that any update of a record, or any update of a table, can be properly addressed by, for example, sending an invalidation message to the appropriate edge servers. Another source of complexity comes from the fact that queries still need to be processed at edge servers. In other words, there is nonnegligible computational power needed to handle queries. Considering that databases often form a performance bottleneck in Web servers, alternative solutions may be needed. Finally, caching results from queries that span multiple tables (i.e., when queries are complex) such that a query containment check can be carried out effectively is not trivial. The reason is that the organization of the results may be very different from the organization of the tables on which the query operated.

These observations lead us to a third solution, namely content-blind caching, described in detail by Sivasubramanian et al. (2006). The idea of content-blind caching is extremely simple: when a client submits a query to an edge server, the server first computes a unique hash value for that query. Using this hash value, it subsequently looks in its cache whether it has processed this query before. If not, the query is forwarded to the origin and the result is cached before

returning it to the client. If the query had been processed before, the previously cached result is returned to the client.

The main advantage of this scheme is the reduced computational effort that is required from an edge server in comparison to the database approaches described above. However, content-blind caching can be wasteful in terms of storage as the caches may contain much more redundant data in comparison to content-aware caching or database replication. Note that such redundancy also complicates the process of keeping the cache up to date as the origin server may need to keep an accurate account of which updates can potentially affect cached query results. These problems can be alleviated when assuming that queries can match only a limited set of predefined templates as we discussed above.

Obviously, these techniques can be equally well deployed for the upcoming generation of Web services, but there is still much research needed before stable solutions can be identified.

12.7. Fault Tolerance

Fault tolerance in the Web-based distributed systems 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. as an example we mentioned before, DNS allows several addresses to be returned as the result of a name lookup. In traditional Web-based systems, fault tolerance can be relatively easy to achieve considering the stateless design of servers, along with the often static nature of the provided content.
[Page 583]

When it comes to Web services, similar observations hold: hardly any new or special techniques are introduced to deal with faults (Birman, 2005). However, it should be clear that problems of masking failures and recoveries can be more severe. For example, Web services support wide-area distributed transactions and solutions will definitely have to deal with failing participating services or unreliable communication.

Even more important is that in the case of Web services we may easily be dealing with complex calling graphs. Note that in many Web-based systems computing follows a simple two-tiered client-server calling convention. This means that a client calls a server, which then computes a response without the need of additional external services. As said, fault tolerance can often be achieved by simply replicating the server or relying partly on result caching.

This situation no longer holds for Web services. In many cases, we are now dealing with multitiered solutions in which servers also act as clients. Applying replication to servers means that callers and callees need to handle replicated invocations, just as in the case of replicated objects as we discussed back in Chap. 10.

Problems are aggravated for services that have been designed to handle Byzantine failures. Replication of components plays a crucial role here, but so does the protocol that clients execute. In addition, we now have to face the situation that a Byzantine fault-tolerant (BFT) service may need to act as a client of another nonreplicated service. A solution to this problem

is proposed in Merideth et al. (2005) that is based on the BFT system proposed by Castro and Liskov (2002), which we discussed in Chap. 11.

There are three issues that need to be handled. First, clients of a BFT service should see that service as just another Web service. In particular, this means that the internal replication of that service should be hidden from the client, along with a proper processing of responses. For example, a client needs to collect k + 1 identical answers from up to 2k + 1 responses, assuming that the BFT service is designed to handle at most k failing processes. Typically, this type of response processing can be hidden away in client-side stubs, which can be automatically generated from WSDL specifications.

Second, a BFT service should guarantee internal consistency when acting as a client. In particular, it needs to handle the case that the external service it is calling upon returns different answers to different replicas. This could happen, for example, when the external service itself is failing for whatever reason. As a result, the replicas may need to run an additional agreement protocol as an extension to the protocols they are already executing to provide Byzantine fault tolerance. After executing this protocol, they can send their answers back to the client.
[Page 584]

Finally, external services should also treat a BFT service acting as a client, as a single entity. In particular, a service cannot simply accept a request coming from a single replica, but can proceed only when it has received at least k + 1 identical requests from different replicas.

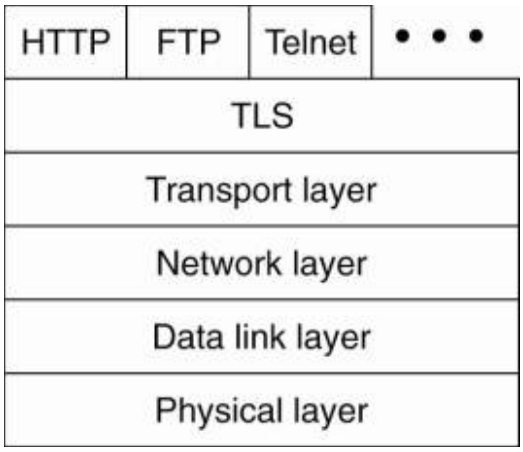
These three situations lead to three different pieces of software that need to be integrated into toolkits for developing Web services. Details and performance evaluations can be found in Merideth et al. (2005).

12.8. 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 implemented by Netscape. Although SSL has never been formally standardized, most Web clients and servers nevertheless support it. An update of SSL has been formally laid down in RFC 2246 and RFC 3546, now referred to as the Transport Layer Security (TLS) protocol (Dierks and Allen, 1996; and Blake-Wilson et al., 2003).

As shown in Fig. 12-22, 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.

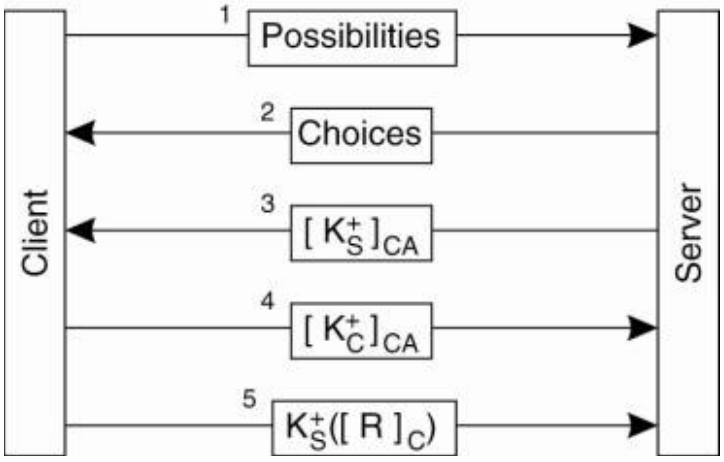
Figure 12-22. The position of TLS in the Internet protocol stack.



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.
[Page 585]

Setting up a secure channel proceeds in two phases, as shown in Fig. 12-23. 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 first two messages shown in Fig. 12-23.

Figure 12-23. 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. 12-23.

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. 12-23. (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.