DISTRIBUTED SYSTEMS
Principles and Paradigms
Second Edition
ANDREW S. TANENBAUM
MAARTEN VAN STEEN

Chapter 12
Distributed Web-Based Systems

Traditional Web-Based Systems

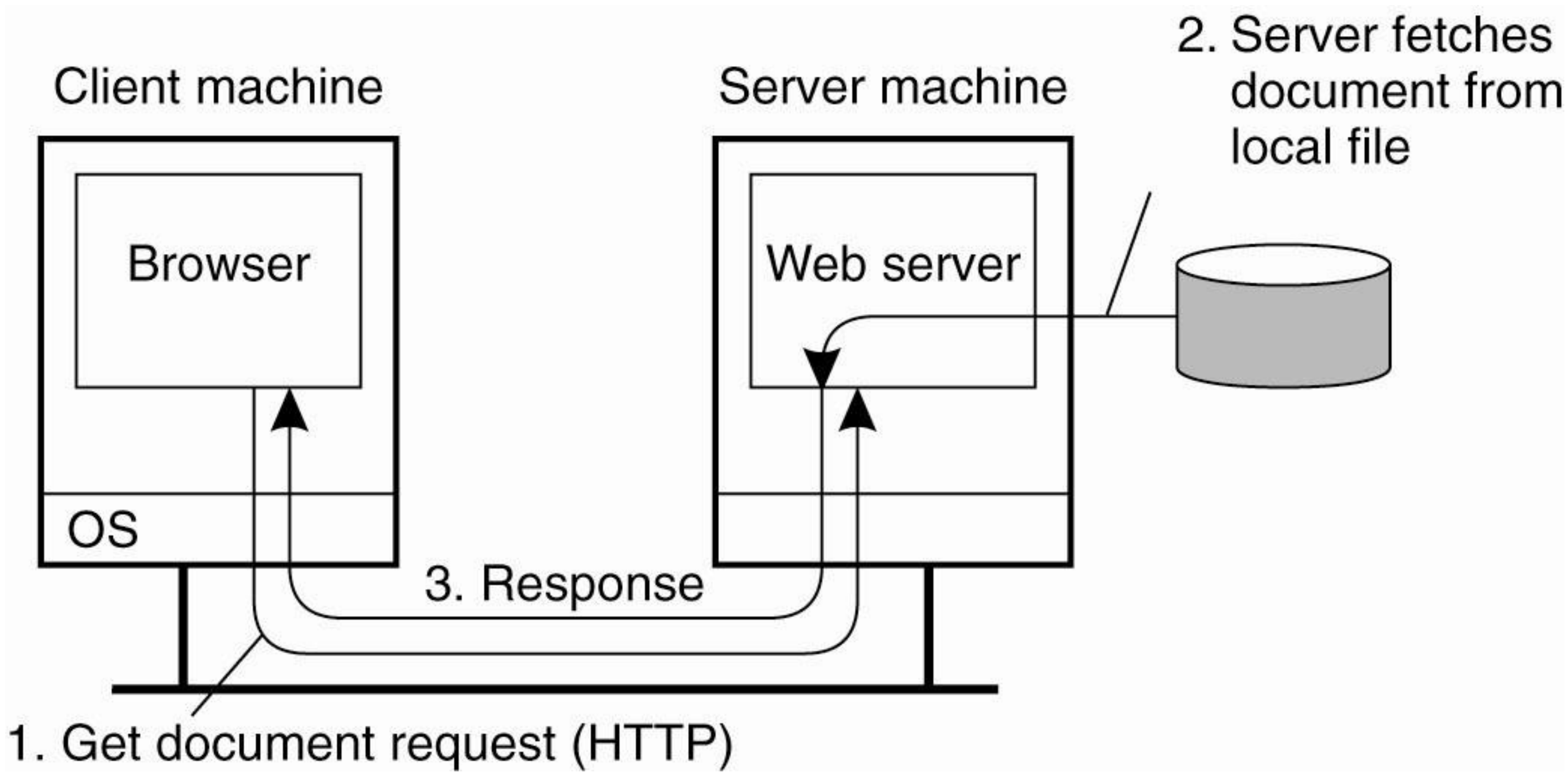
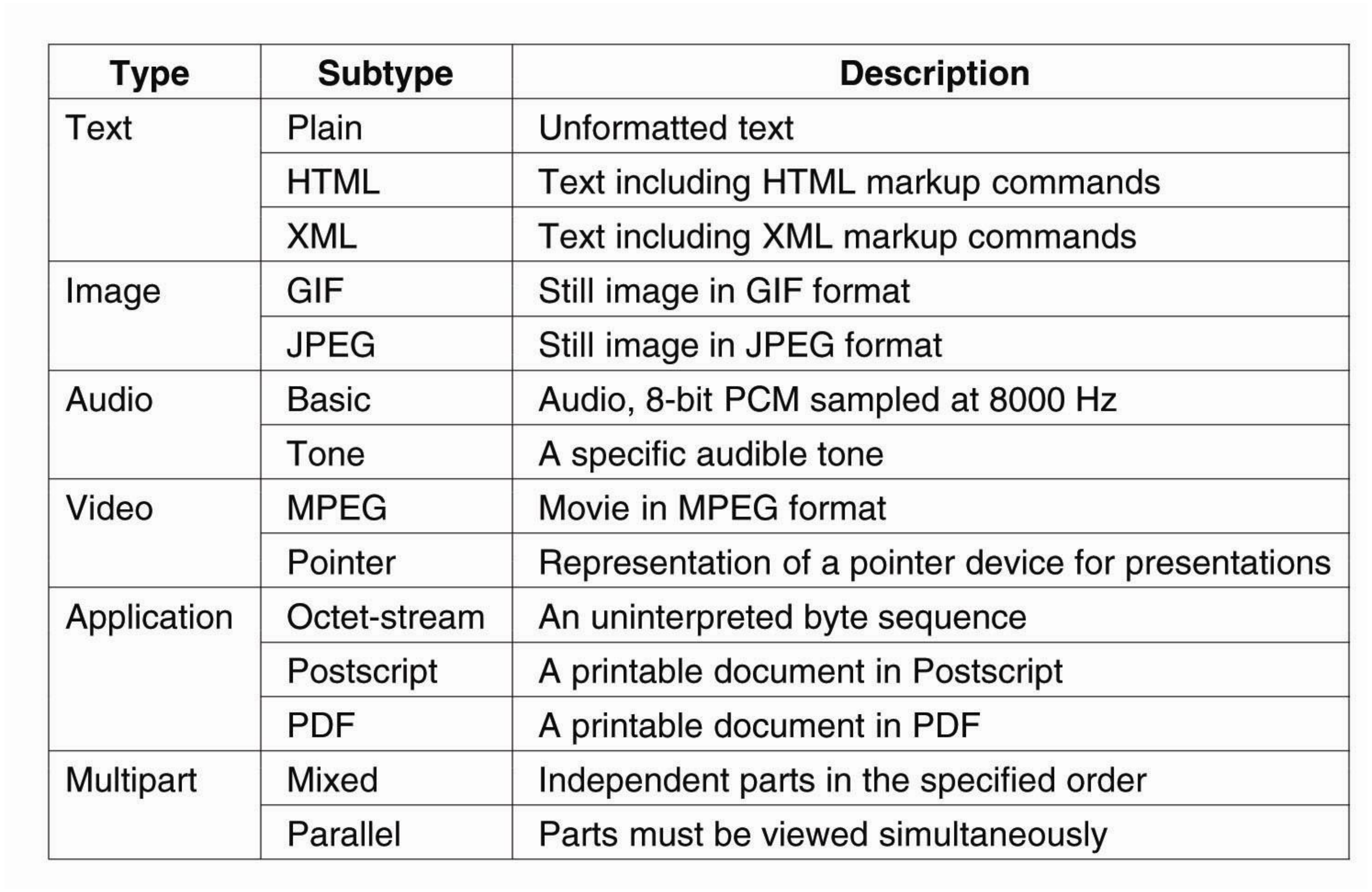

2. Server fetches document from local file
Client machine
Server machine
Browser
Web server
OS
3. Response
1. Get document request (HTTP)

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

Web Documents



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

Figure 12-2. Six top-level MIME types and some common subtypes.

Multitiered Architectures

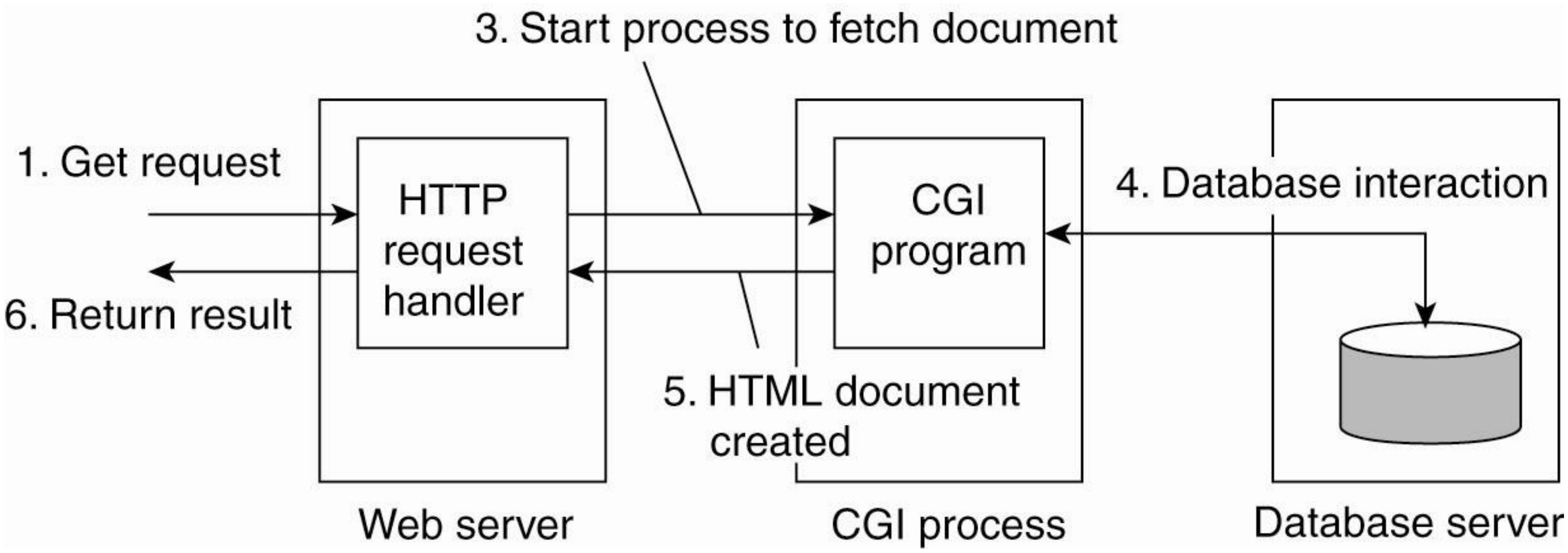


Figure 12-3. The principle of using server-side CGI programs.

The Web Services Fundamental

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).

- ❑ Simply stated, a Web service is nothing but a traditional service (e.g., naming, weather-reporting service, 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. The core of Web services architecture [see also Booth et al. (2004)]

The Web Services Fundamentals

- ❑ This service adheres to the Universal Description, Discovery and Integration standard (UDDI). The UDDI prescribes the layout of a database containing service descriptions that will allow Web service clients to browse for relevant services.

- ❑ 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 (IDL) 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 client side 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 Fundamentals

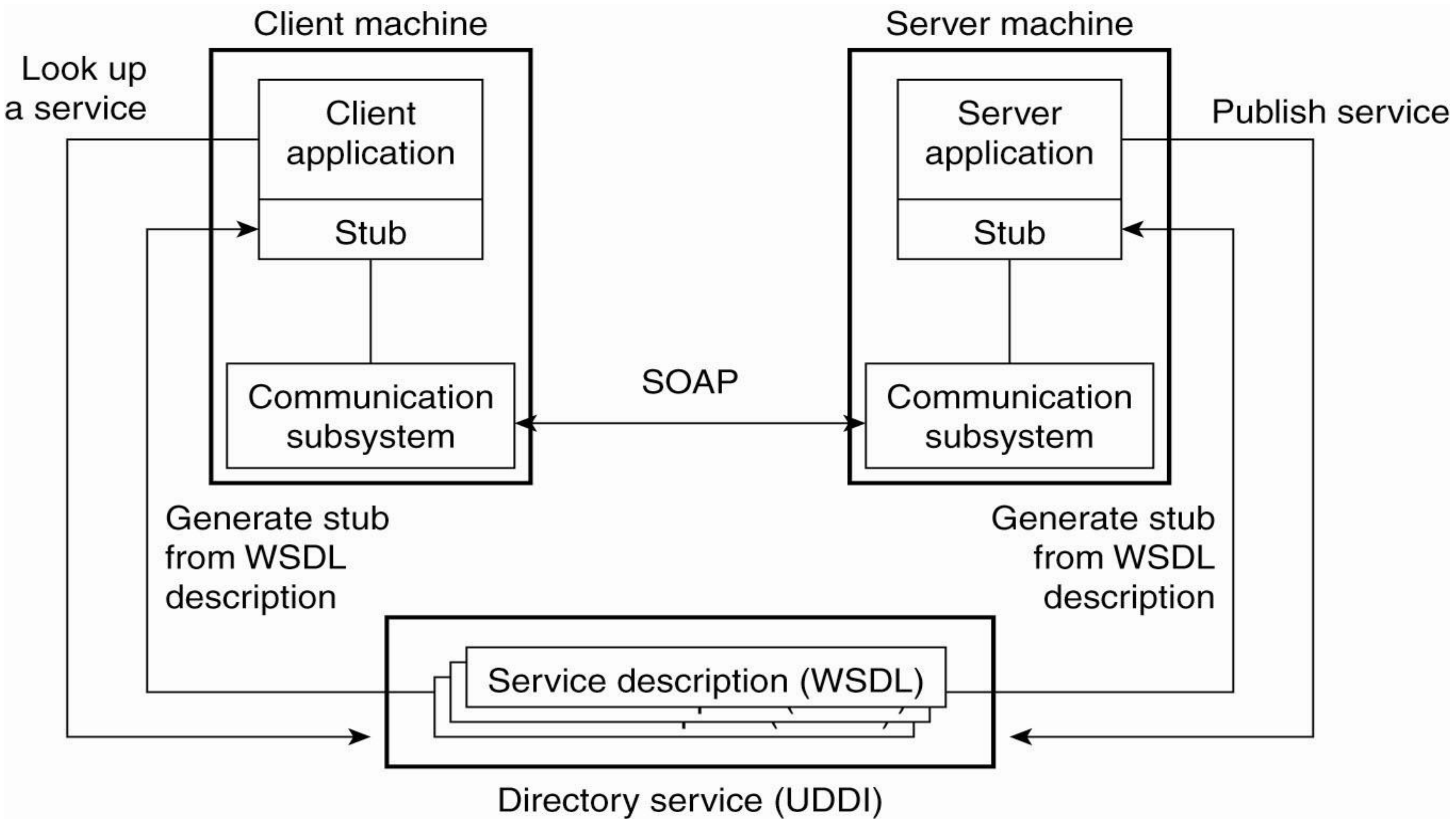


Figure 12-4. The principle of a Web service.

Web Service 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.

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.

1) Ordering a book requires selecting a book, 2) paying, 3) 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.

Web Service Composition and Coordination

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**?

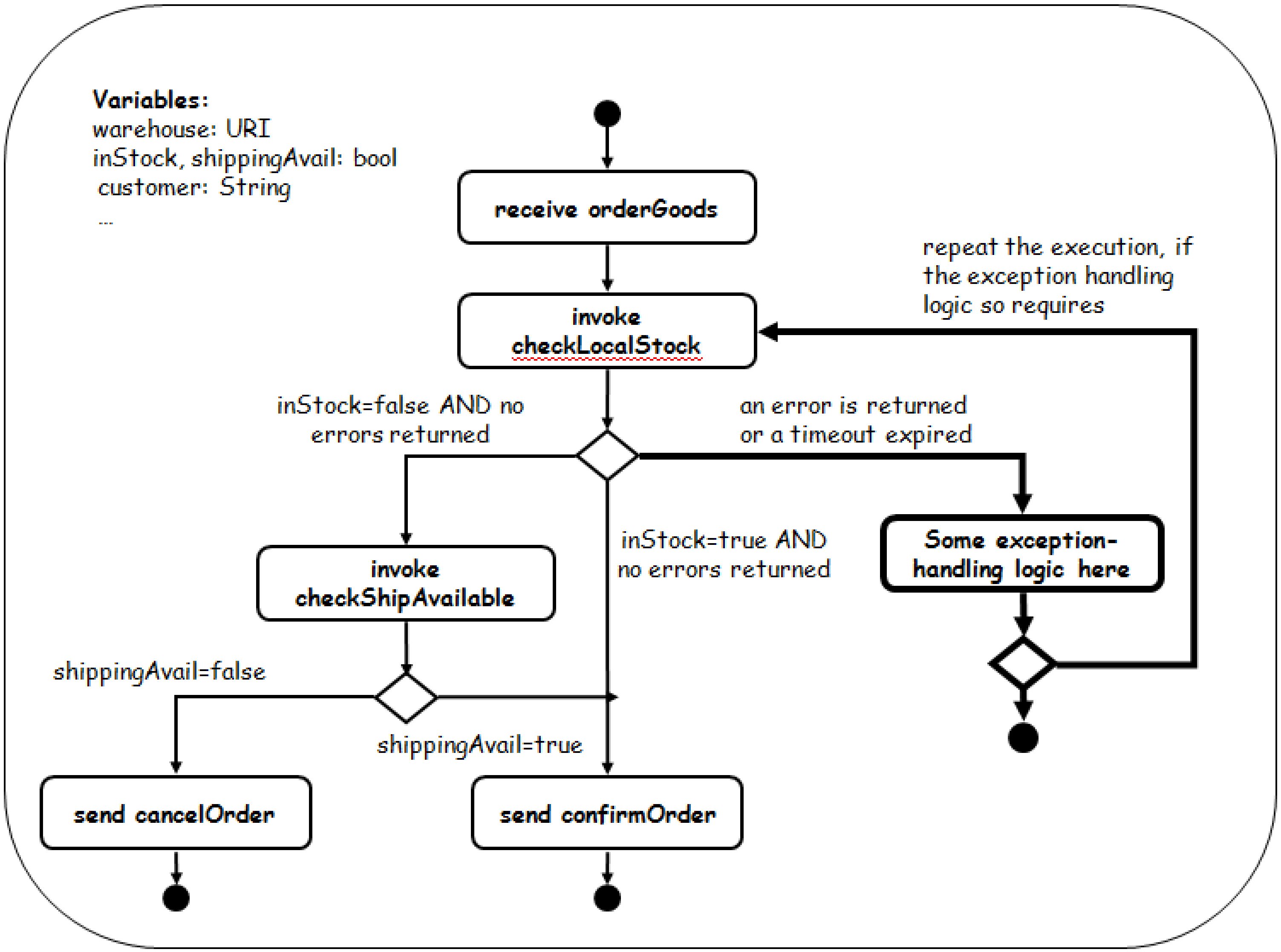
Web Service Composition and Coordination

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, is to *enforce the parties taking part in* such protocol take the correct steps at the right moment.

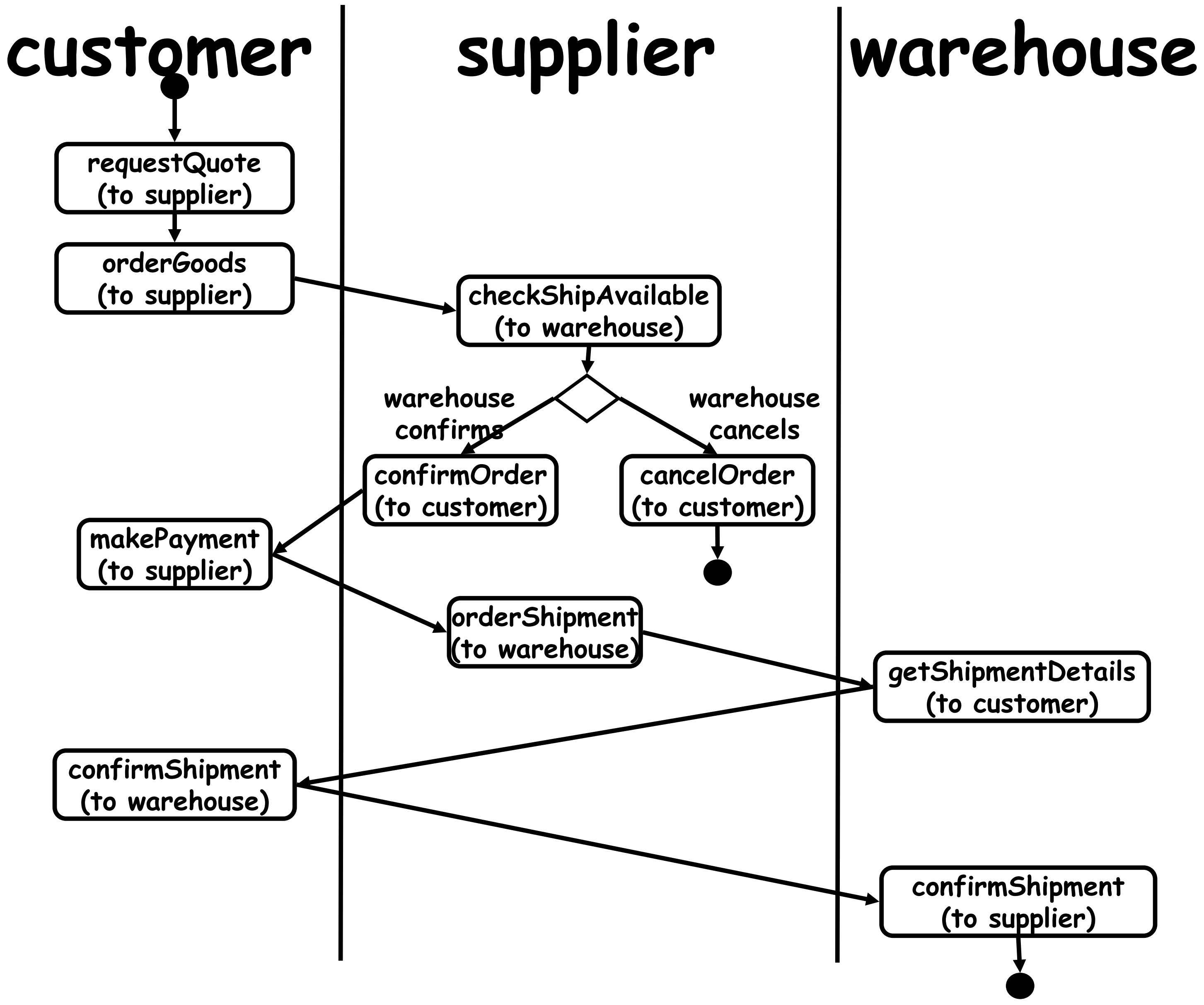
There are various ways to achieve this;

- ❑ 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.

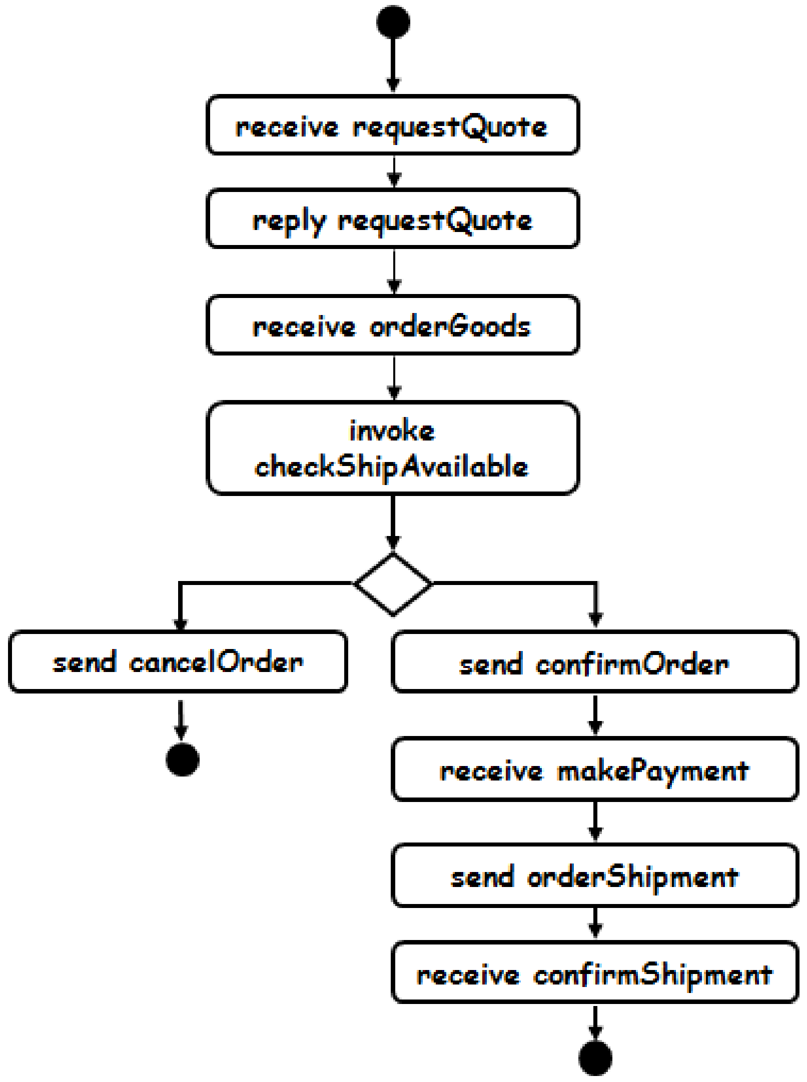
Web Service Composition and Coordination
A Sample


Variables:
warehouse: URI
inStock, shippingAvail: bool
customer: String
...
receive orderGoods
invoke checkLocalStock
repeat the execution, if the exception handling logic so requires
inStock=false AND no errors returned
an error is returned or a timeout expired
inStock=true AND no errors returned
Some exception-handling logic here
invoke checkShipAvailable
shippingAvail=false
shippingAvail=true
send cancelOrder
send confirmOrder

Web Service Composition and Coordination


customer
supplier
warehouse
requestQuote (to supplier)
orderGoods (to supplier)
checkShipAvailable (to warehouse)
warehouse confirms
warehouse cancels
confirmOrder (to customer)
cancelOrder (to customer)
makePayment (to supplier)
orderShipment (to warehouse)
getShipmentDetails (to customer)
confirmShipment (to warehouse)
confirmShipment (to supplier)

Web Service Composition and Coordination



Processes – Clients (1)

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 users' 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.

Important aspect of Web browsers:

❑ Platform independent.

❑ Should be easily extensible so that it, can support any type of document that is returned by a server.

❑ Another client-side process that is often used is a Web proxy as shown in Figure 12.6

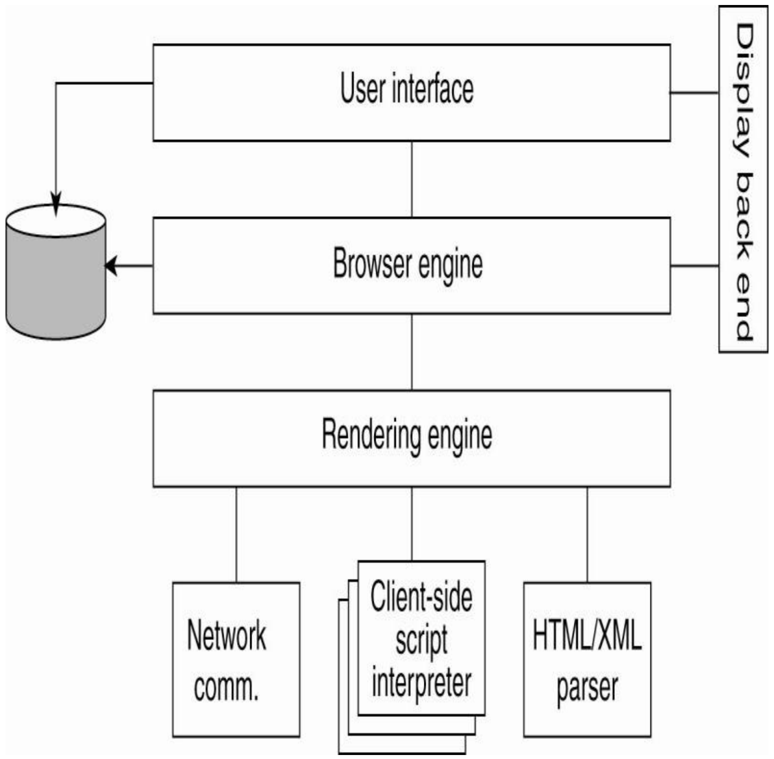


Figure 12-5. The logical components of a Web browser.

Processes – Clients

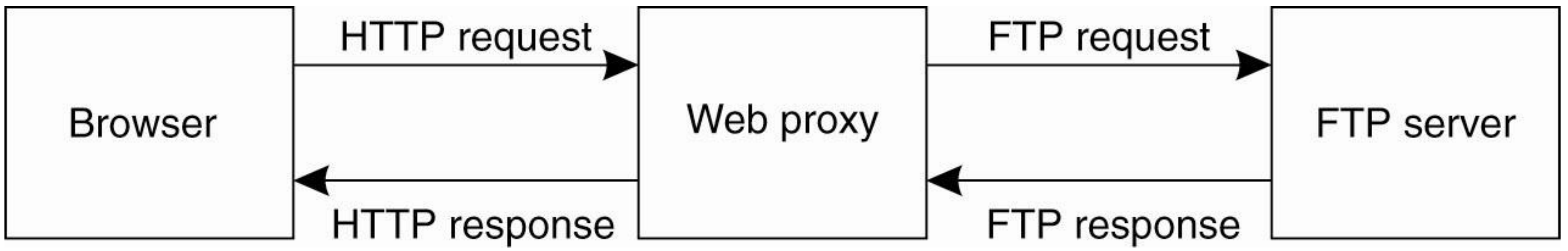


Figure 12-6. Using a Web proxy when the browser does not speak FTP.

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's 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.

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.

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
- 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**.

The Apache Web Server

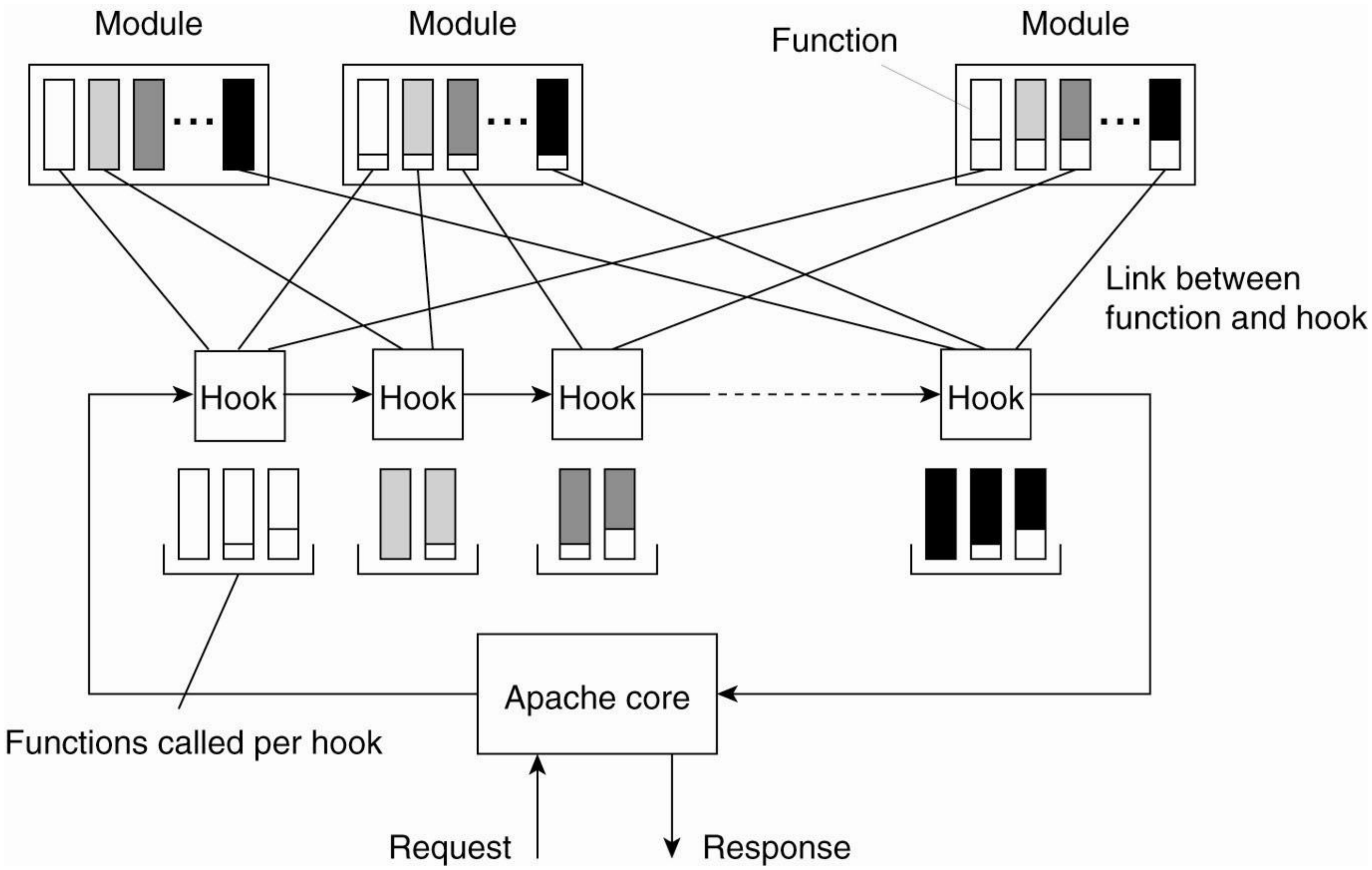


Figure 12-7. The general organization of the Apache Web server.

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.

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 **transport layer switches**, and those that operate at the level of the application layer.

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.

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.

Web Server Clusters

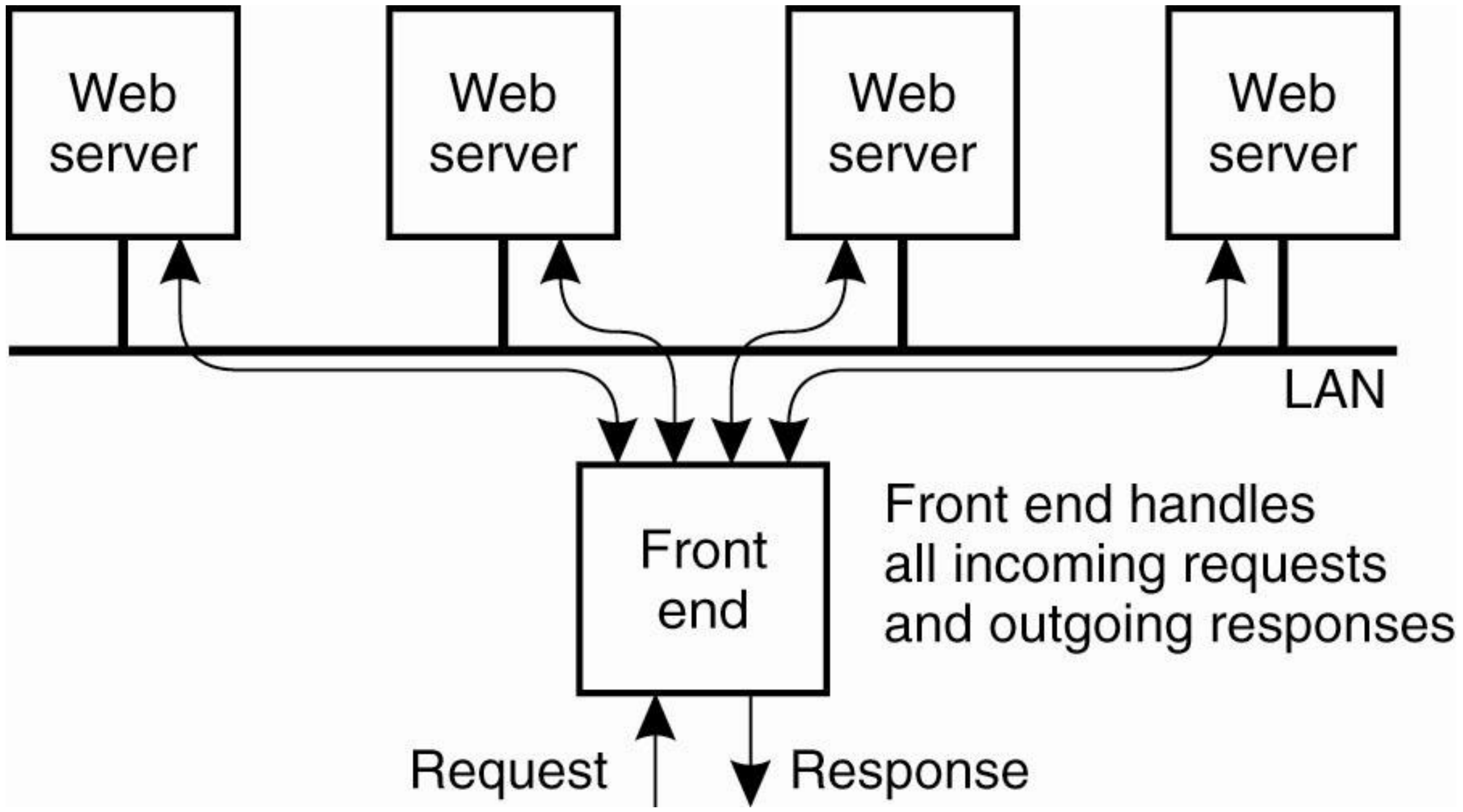


Figure 12-8. The principle of using a server cluster in combination with a front end to implement a Web service.

Web Server Clusters

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**.

Web Server Clusters

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.

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.

Web Server Clusters

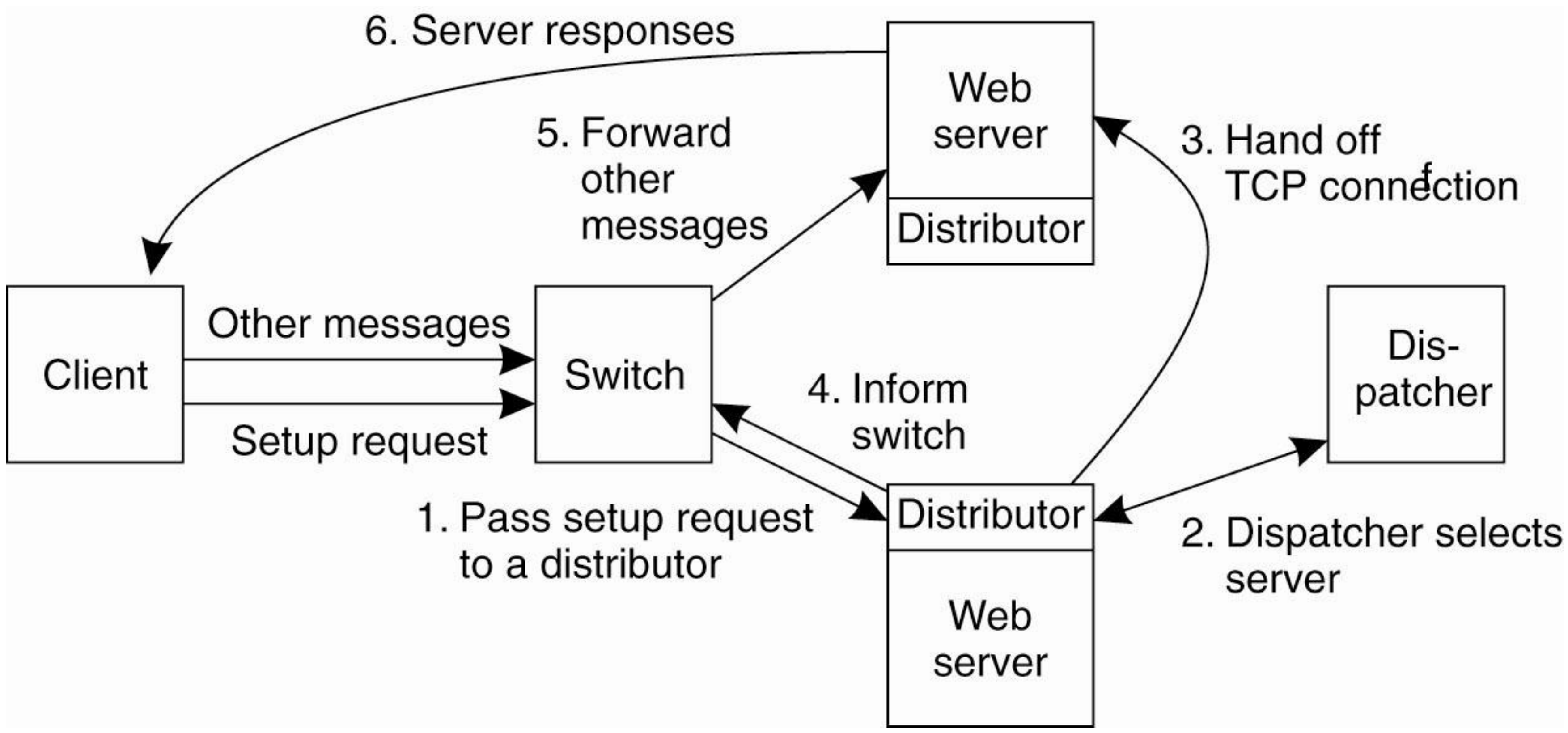


Figure 12-9. A scalable content-aware cluster of Web servers.

Web Server Clusters

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.

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.

Communication
HTTP Connections

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 **non-persistent.**

A **major drawback** of **non-persistent** 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.

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.

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).

Communication HTTP Connections

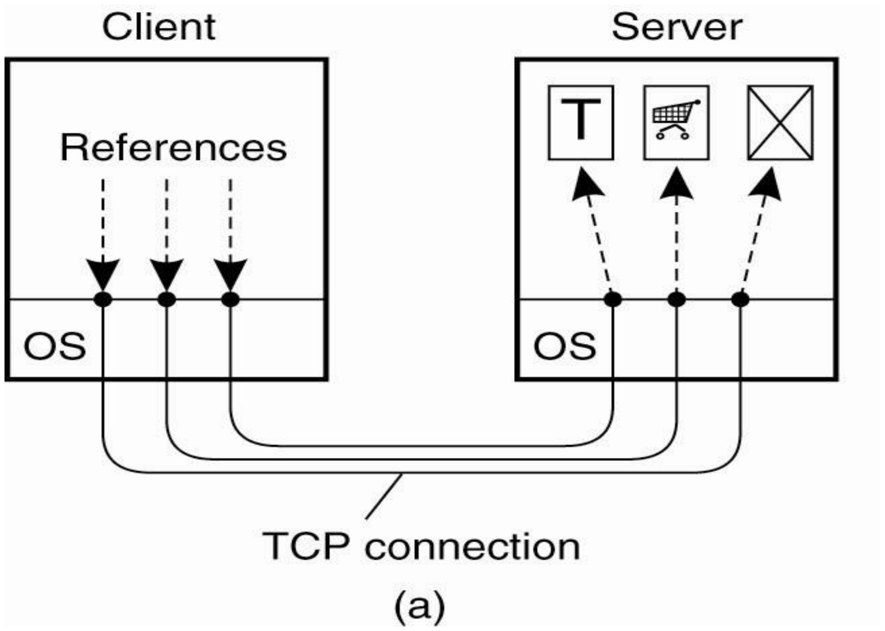


Figure 12-10. (a)

Using non-persistent connections.

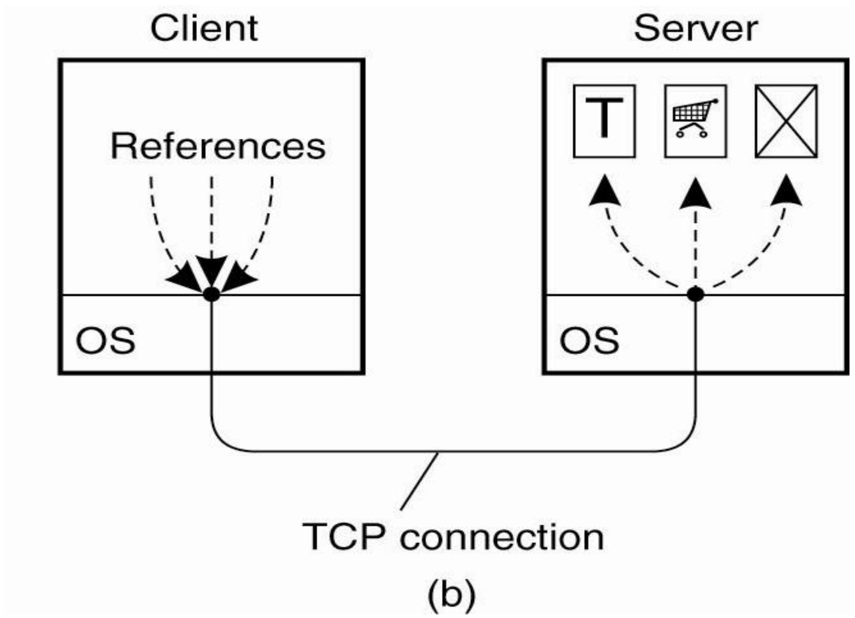
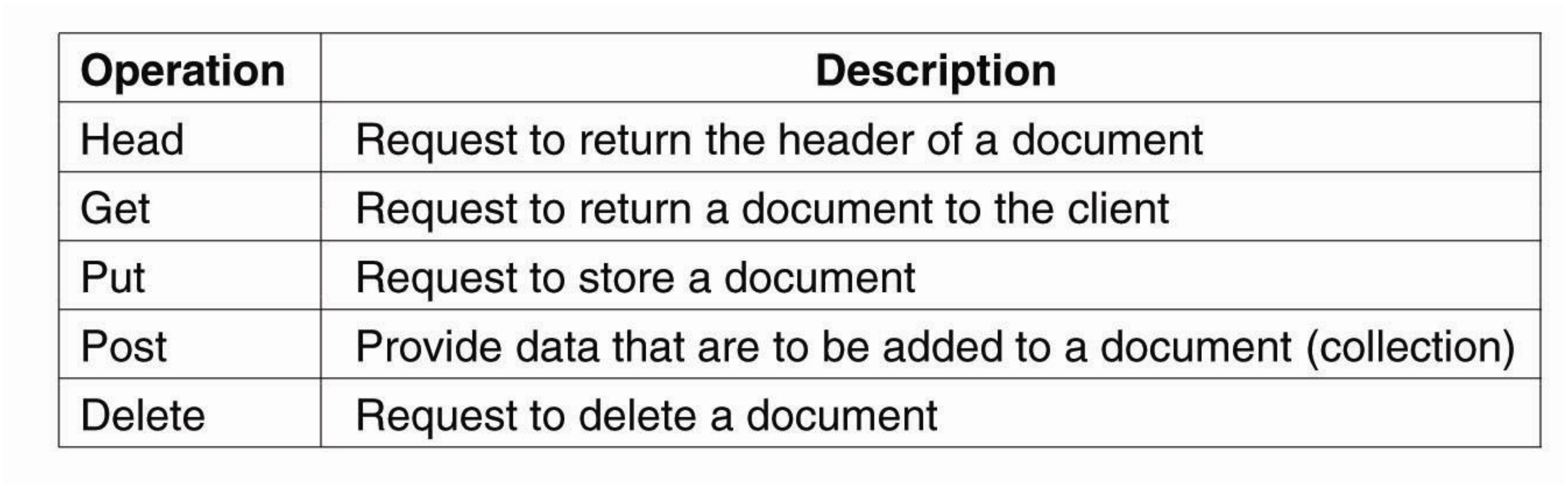


Figure 12-10. (b)

Using persistent connections.

HTTP Methods



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

Figure 12-11. Operations supported by HTTP.

HTTP Messages (1)

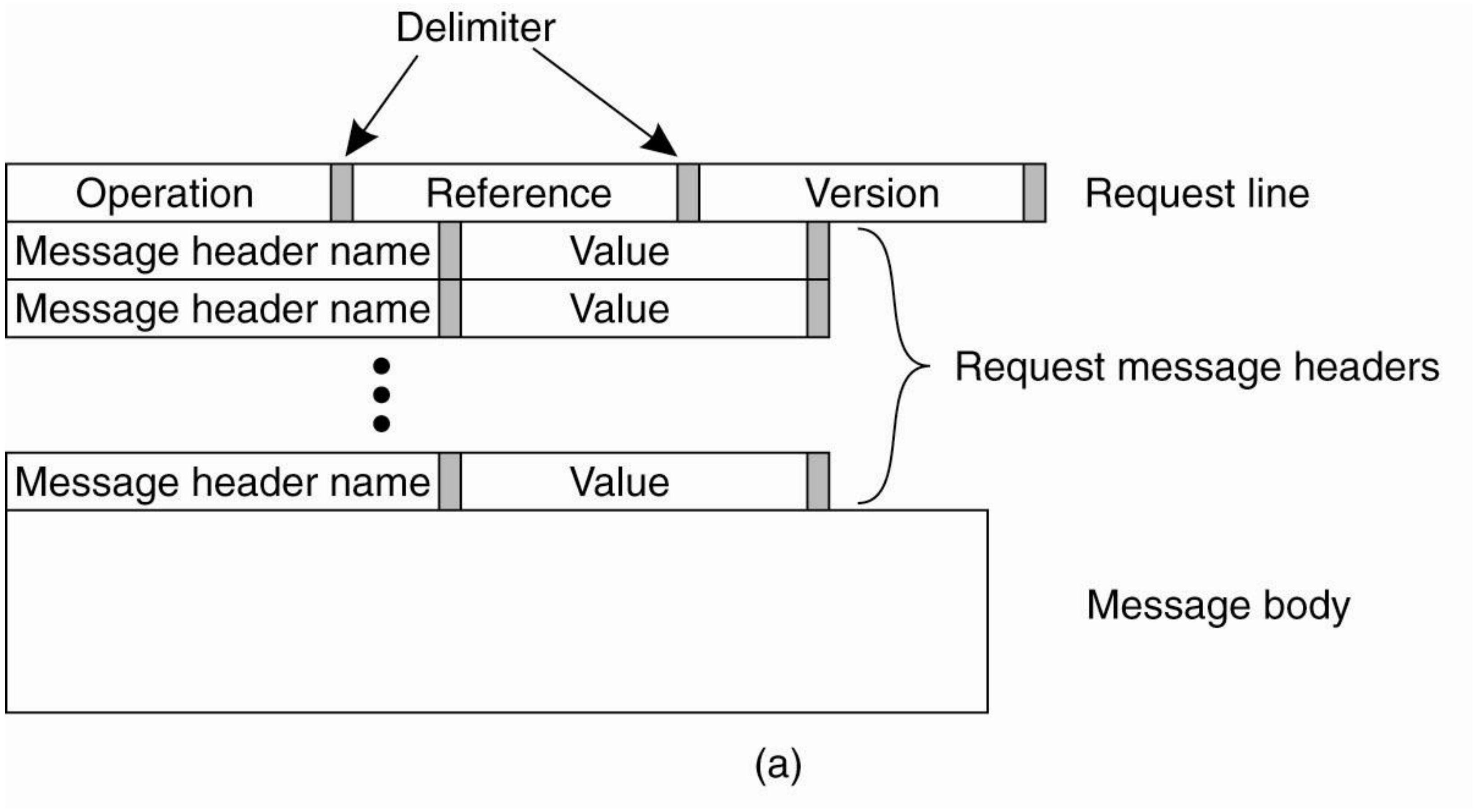


Figure 12-12. (a) HTTP request message.

HTTP Messages (2)

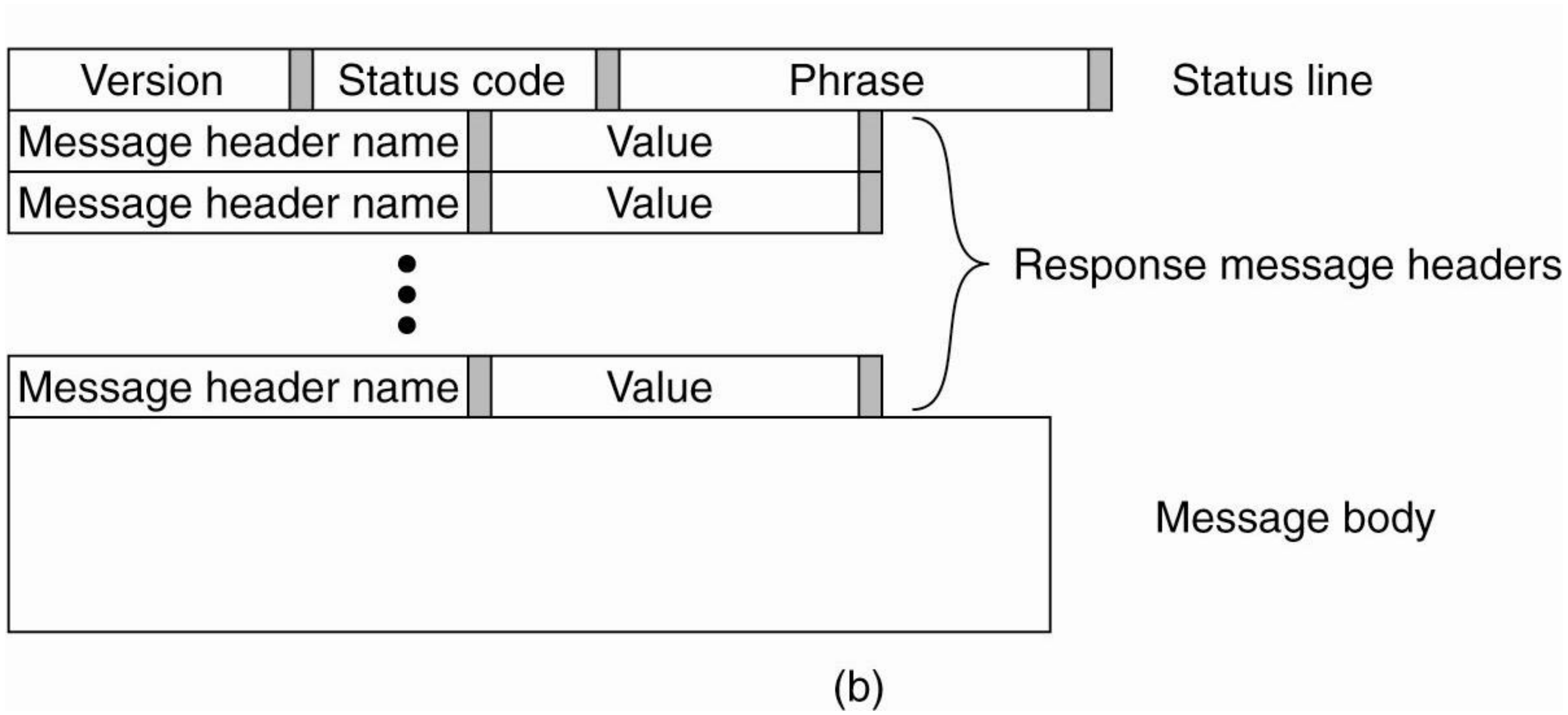
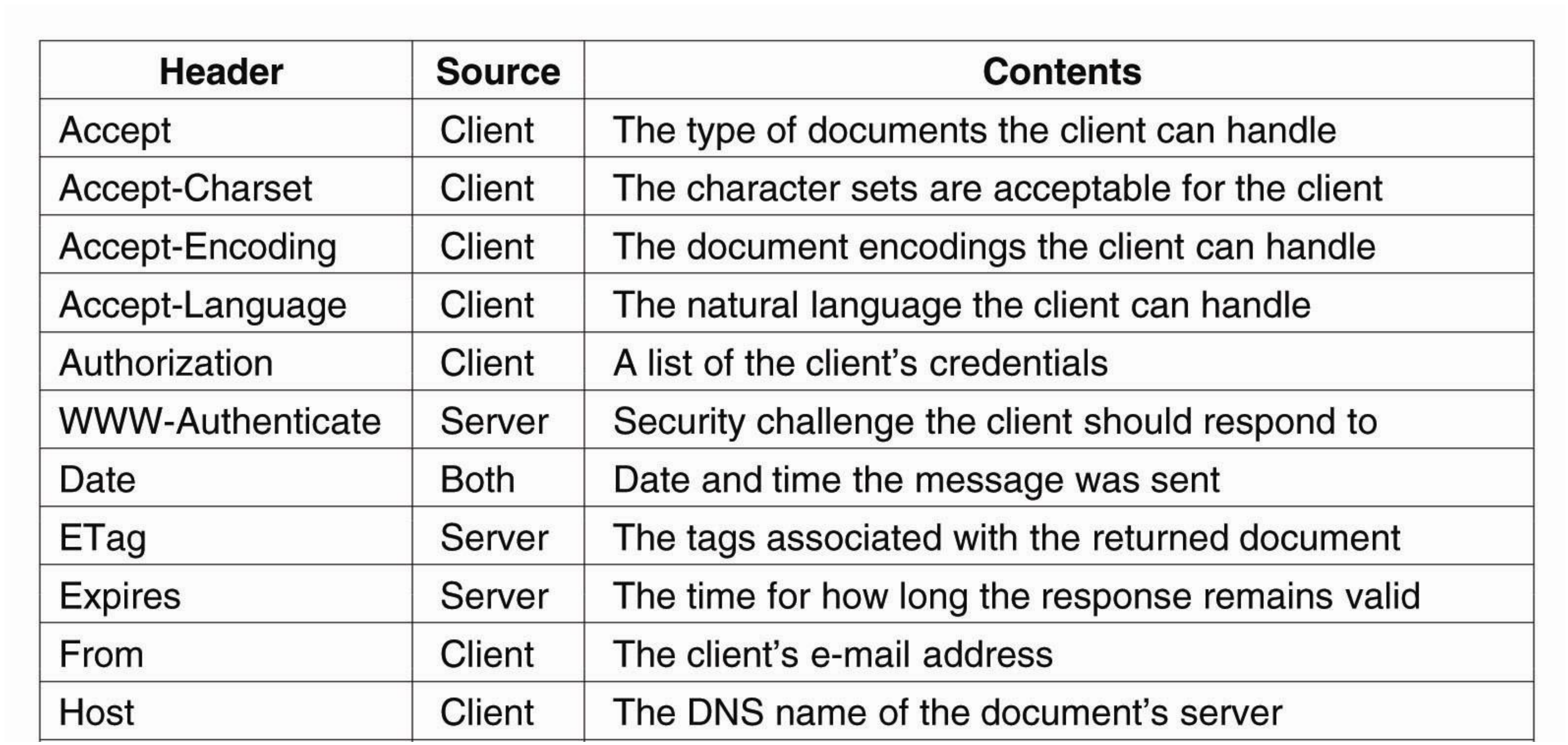


Figure 12-12. (b) HTTP response message.

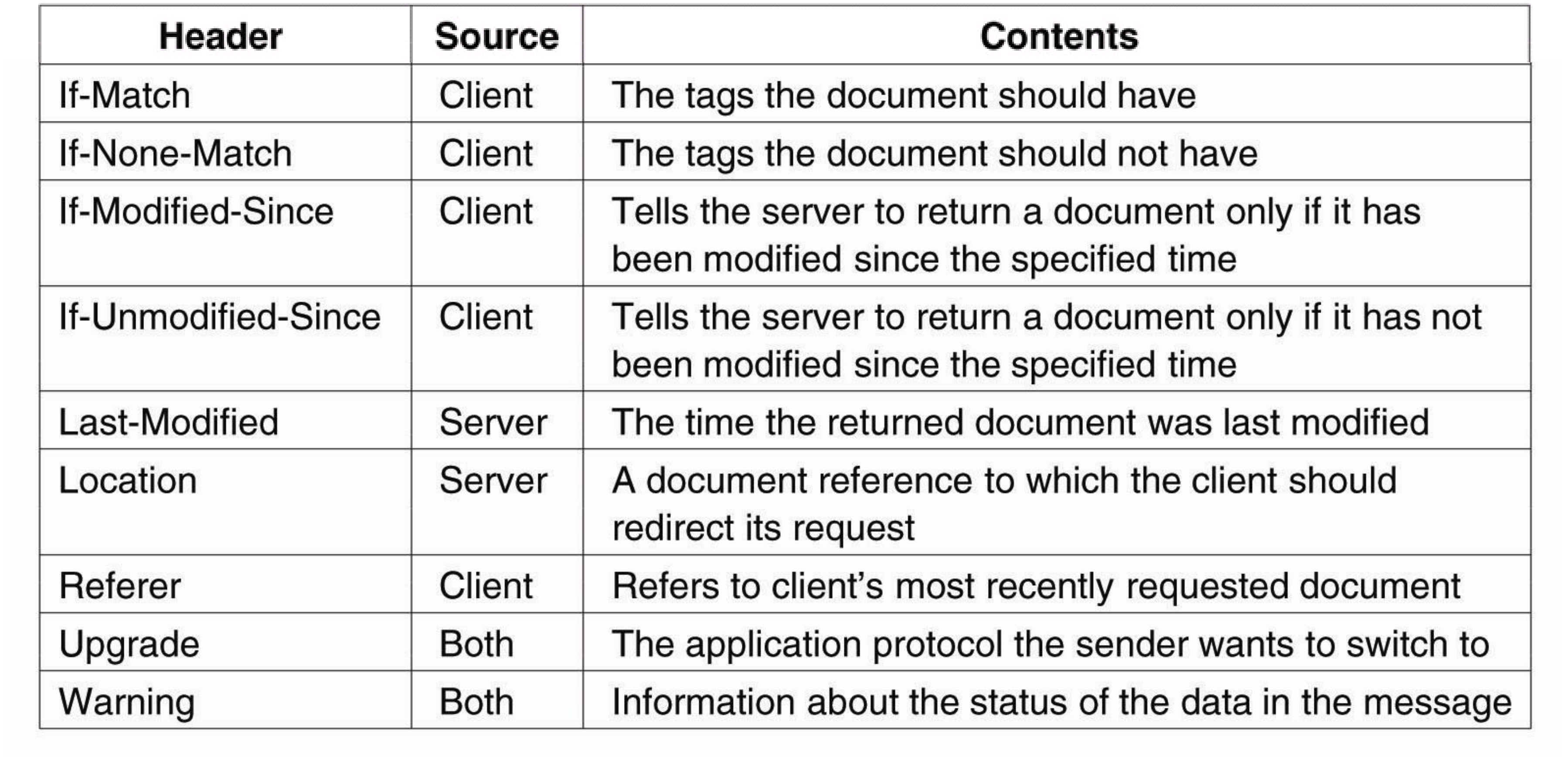
HTTP Messages (3)



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

Figure 12-13. Some HTTP message headers.

HTTP Messages (4)

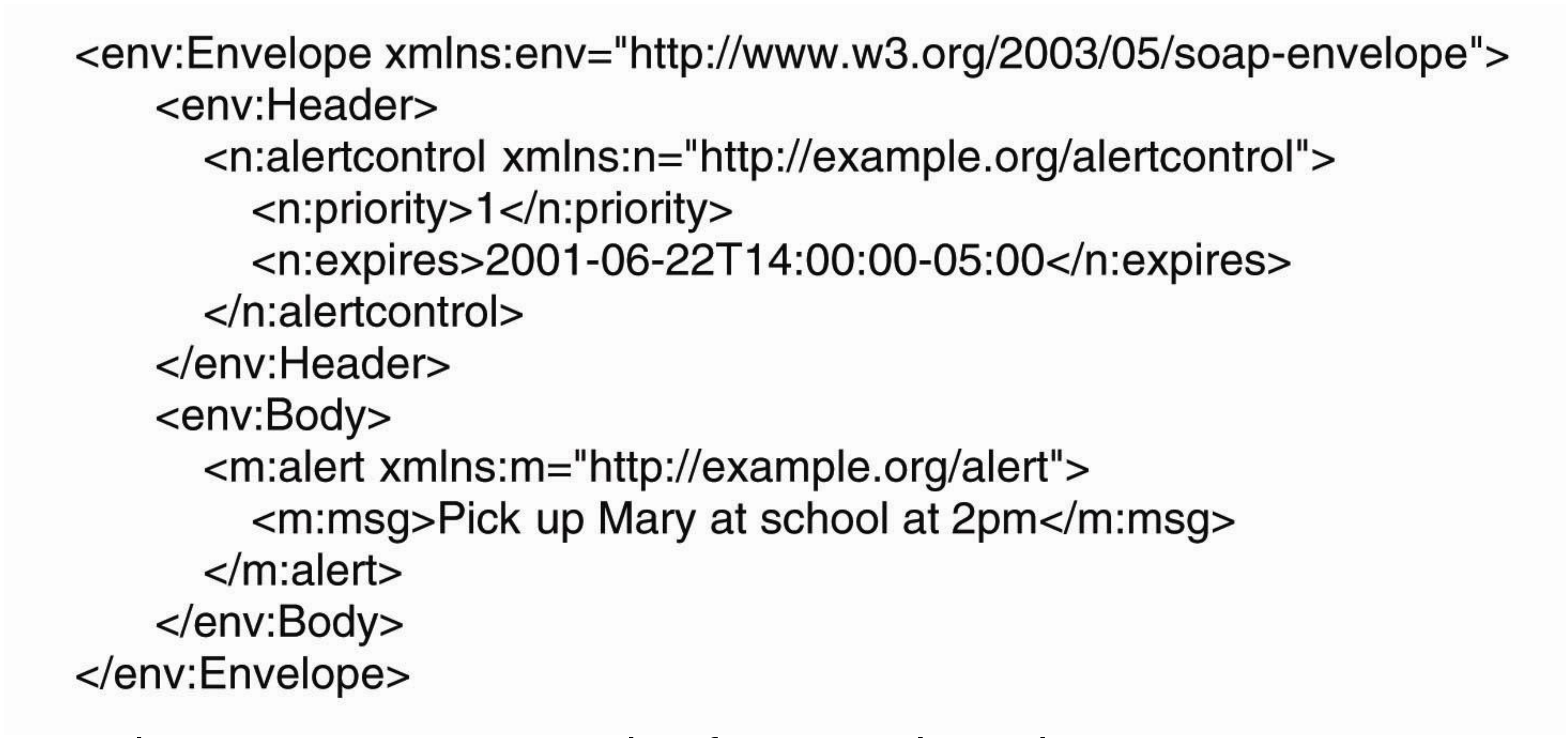


Header	Source	Contents
If-Match	Client	The tags the document should have
If-None-Match	Client	The tags the document should not have
If-Modified-Since	Client	Tells the server to return a document only if it has been modified since the specified time
If-Unmodified-Since	Client	Tells the server to return a document only if it has not been modified since the specified time
Last-Modified	Server	The time the returned document was last modified
Location	Server	A document reference to which the client should redirect its request
Referer	Client	Refers to client's most recently requested document
Upgrade	Both	The application protocol the sender wants to switch to
Warning	Both	Information about the status of the data in the message

Figure 12-13. Some HTTP message headers.

Simple Object Access Protocol SOAP

Where HTTP is the standard communication protocol for traditional Web-based distributed systems, the Simple Object Access Protocol (SOAP) forms the standard for communication with Web services (Gudgin et aI., 2003)



```
<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>
```

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

Simple Object Access Protocol
SOAP

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 multi-tiered 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 Simple 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**.

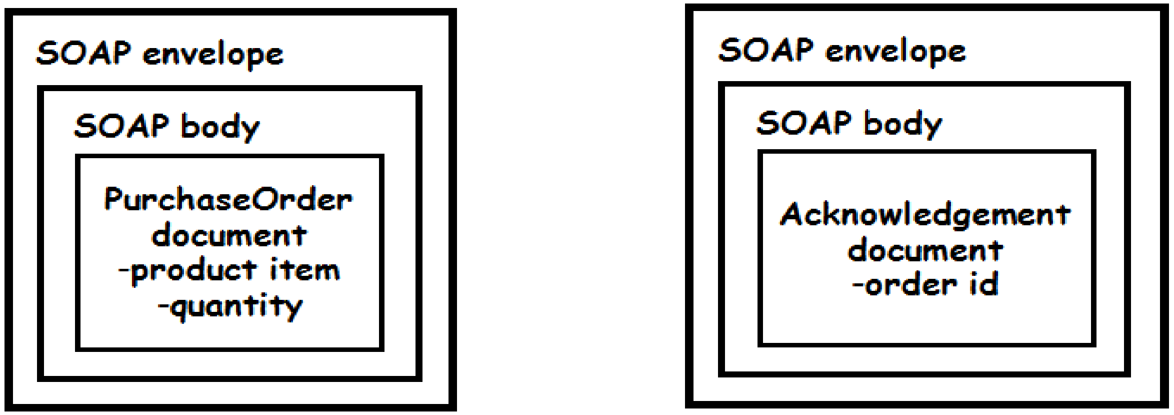
Simple Object Access Protocol SOAP

These two different types of bindings also indicate two different styles of interactions.

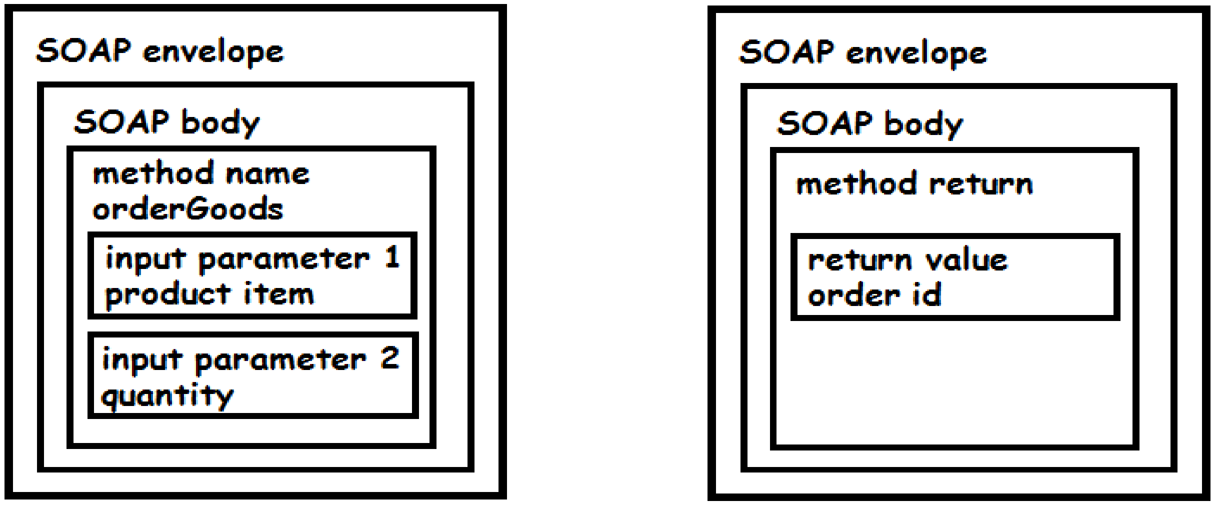
The first, most common one. is the conversational exchange style. In this style, two parties essentially exchange structured documents. For example, such a document may contain a complete purchase order as one would fill in when electronically booking a flight. The response to such an order could be 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 request-response 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. 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.

Simple Object Access Protocol SOAP


SOAP envelope
SOAP body
PurchaseOrder document -product item -quantity
SOAP envelope
SOAP body
Acknowledgement document -order id

(a) Document-style interaction



(b) RPC-style interaction

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) **Uniform Resource Identifier (URI)** 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. A URN is used as a globally unique, **location-independent**, and persistent reference to a document.

The actual syntax of a URI is determined by its associated **scheme**. The name

of a scheme is part of the URI. Many different schemes have been defined, and in the following we will mention a few of them along with examples of their associated URIs.

The **http scheme** is the best known, but it is not the only one. 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)].

Naming

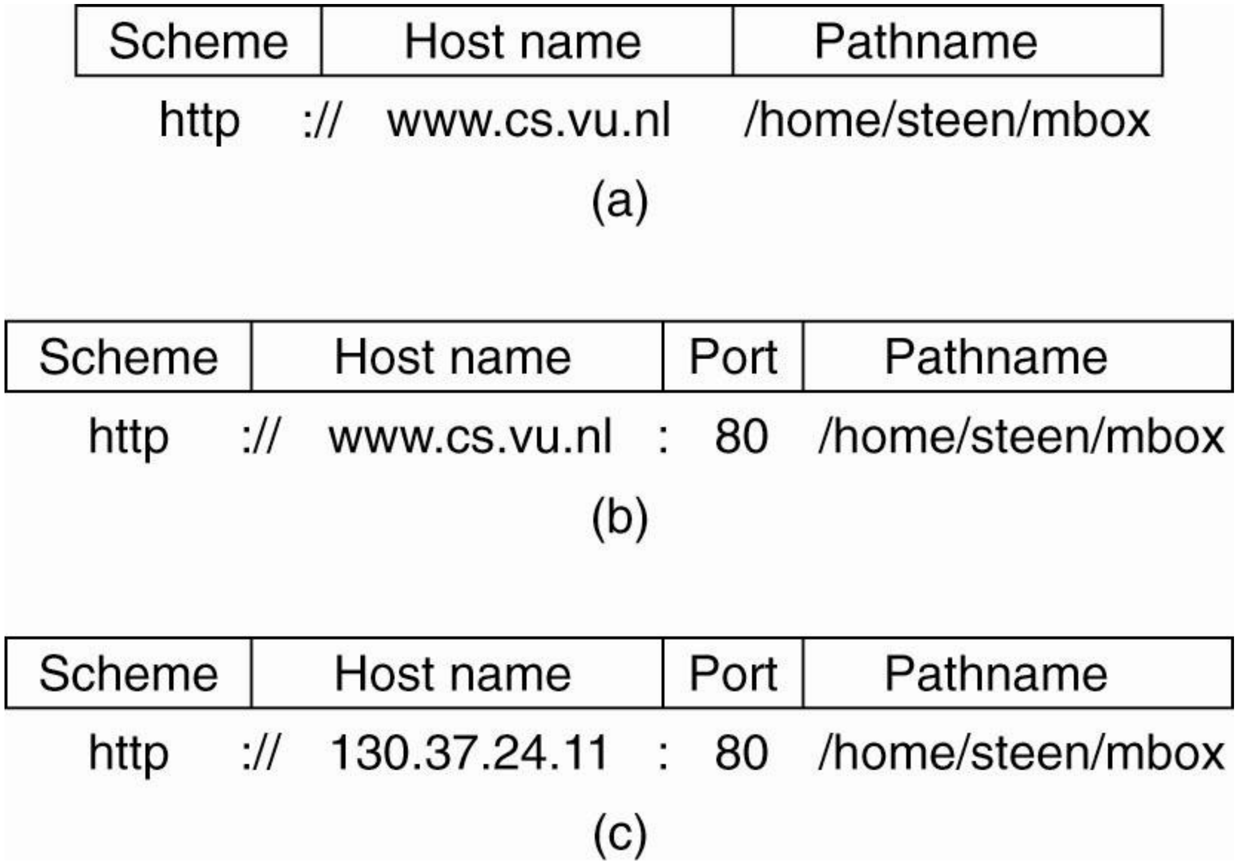
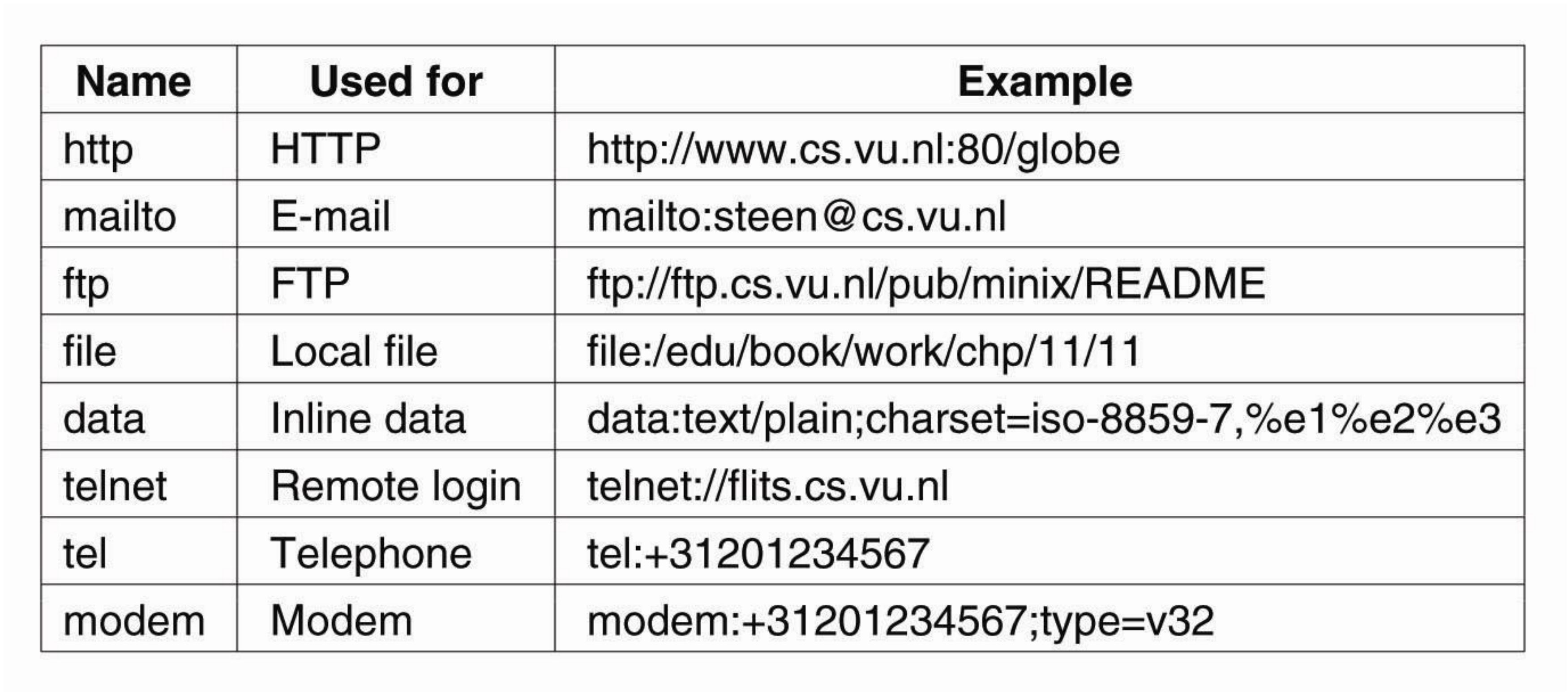

Scheme
Host name
Pathname
http :// www.cs.vu.nl /home/steen/mbox
(a)
Scheme
Host name
Port
Pathname
http :// www.cs.vu.nl : 80 /home/steen/mbox
(b)
Scheme
Host name
Port
Pathname
http :// 130.37.24.11 : 80 /home/steen/mbox
(c)

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.

Naming



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

Figure 12-16. Examples of URIs.

Consistency and Replication

Perhaps one of the most important systems-oriented developments in Web-based 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.

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.

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**.

Web Proxy Caching

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 non-hierarchical 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 co-located in the same LAN.

Web Proxy Caching

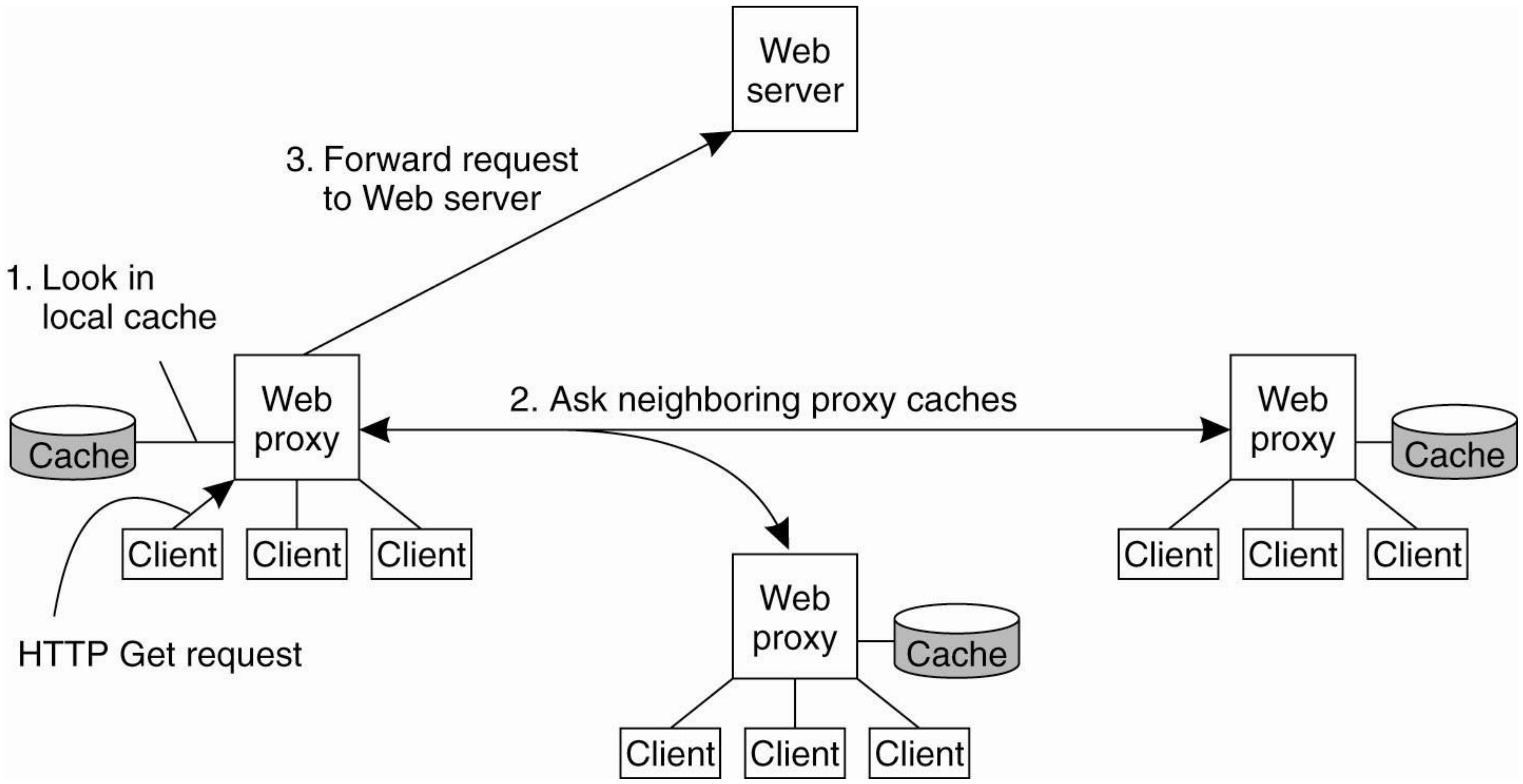


Figure 12-17. The principle of cooperative caching.

Web Proxy Caching

A comparison between hierarchical and cooperative caching by Rodriguez et al. (2001) makes clear that there are various trade-offs to make.

- 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 (Only for the cases that cached are on the web).

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.

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.

The sheer size of a CDN requires that hosted documents are **automatically** distributed and replicated, leading to the architecture of a self-managing system -> Chap. 2.

In most cases, a large-scale CDN is organized along the lines of a feedback-control loop, as shown in Fig. 12-8 and which is described extensively in Sivasubramanian et al. (2004b).

There are essentially three different kinds of aspects related to replication in Web hosting systems:

1) **Metric estimation,**

2) **Adaptation triggering,**

3) **Taking appropriate measures**:

   A. **Replica placement decisions**

   B. **Consistency enforcement**

   C. **Client-request routing.**

Replication for Web Hosting Systems

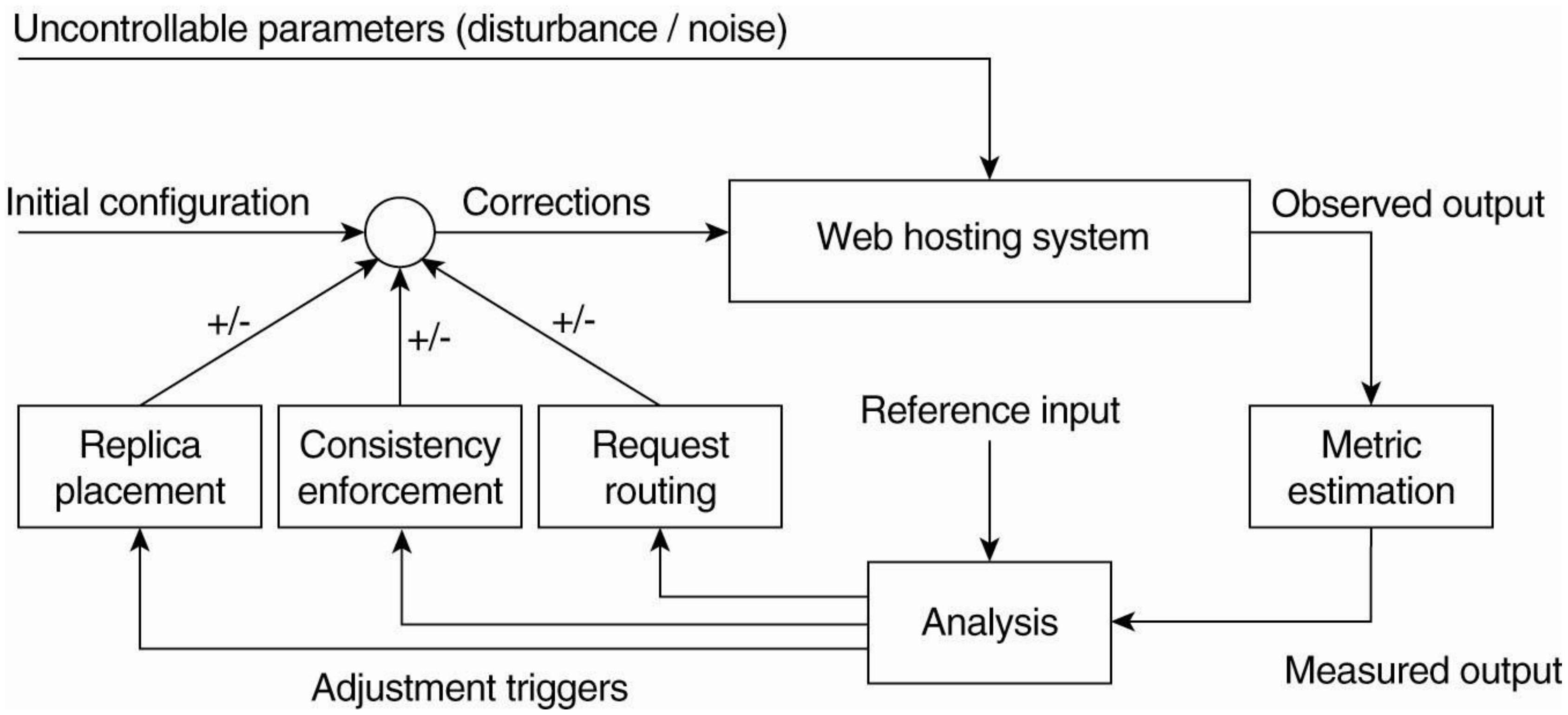


Figure 12-18. The general organization of a CDN as a feedback-control system (adapted from Sivasubramanian et al., 2004b).

CDN-Metric Estimation

A requirement 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.

First, there are **latency metrics**, by which the time is measured for an action. For example, fetching a document, to take place. **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.7 will need to be deployed.

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).

CDN-Metric Estimation

Next 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.

CDN-Metrics

Another 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.

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

CDN-Metrics

Finally, **financial metrics** form **fourth 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 (SLA)** that have been made with customers.

CDN-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 off-load 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.

CDN-Adaptation Triggering

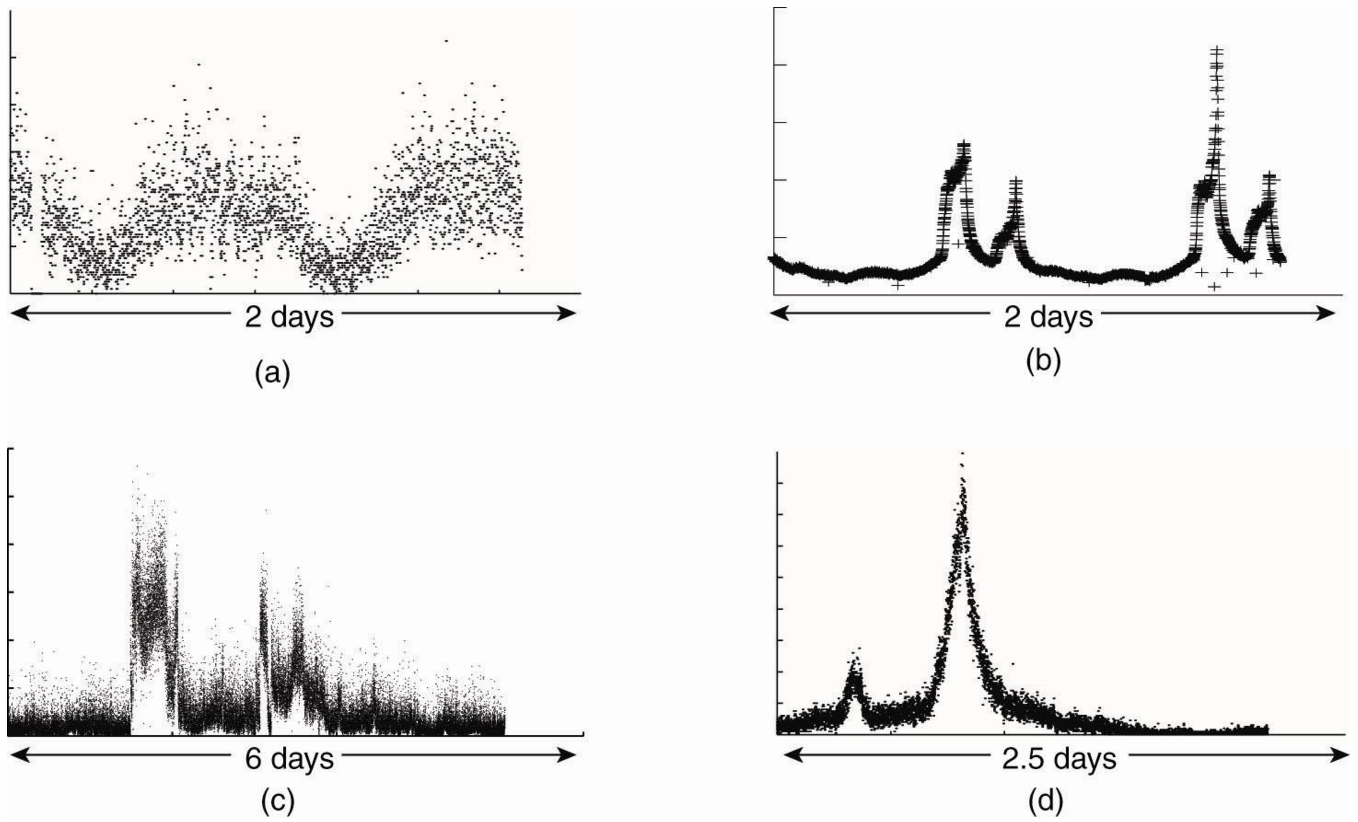


Figure 12-19. One normal and three different access patterns reflecting flash-crowd behavior (adapted from Baryshnikov et al., 2005).

CDN-Adjustment Measures

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 -> Already discussed in Chapter 7
- Changing consistency enforcement -> Already discussed in Chapter 7
- Deciding on how and when to redirect client requests.

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 within 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.**

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 host (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.

Adjustment Measures

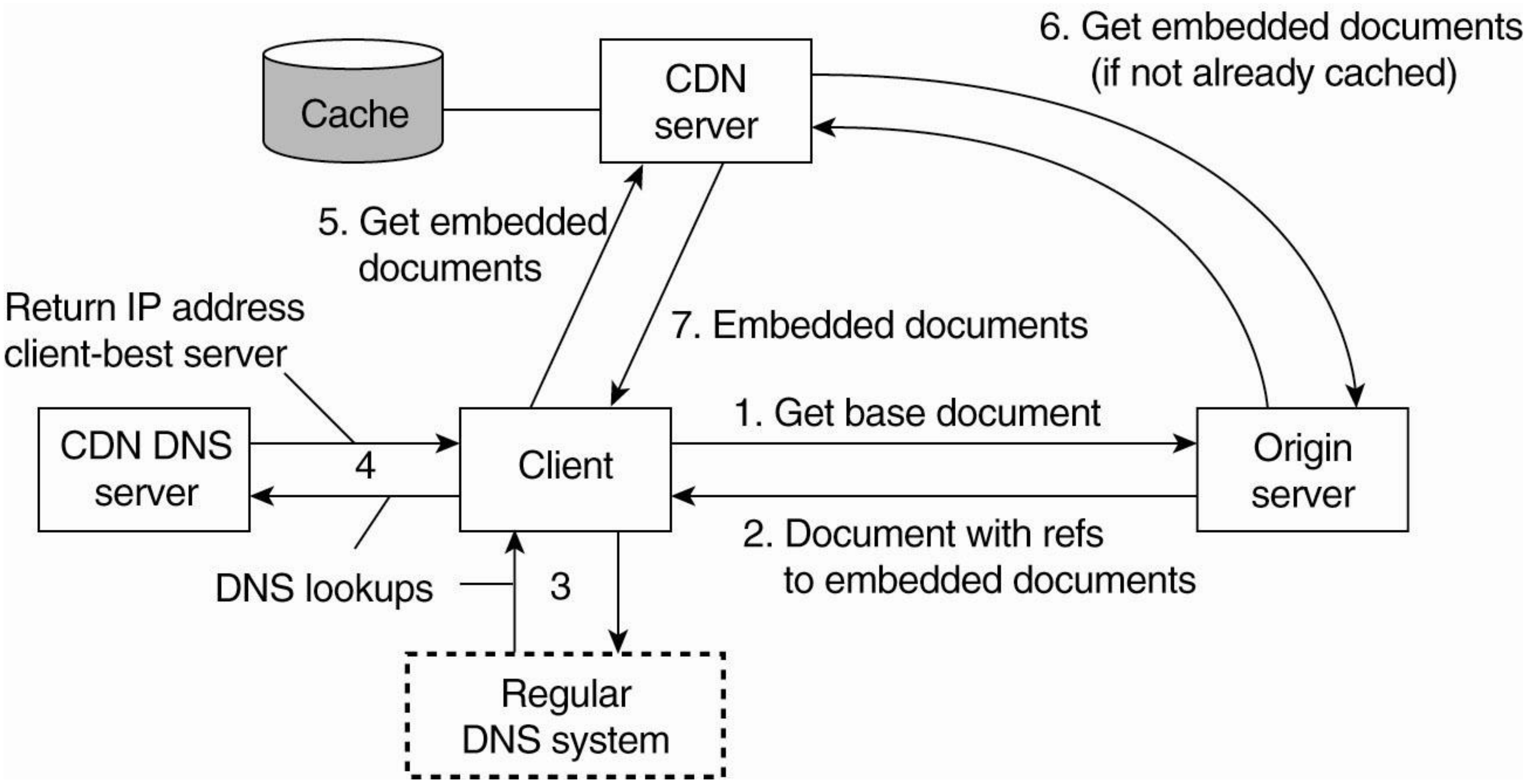

Cache
CDN server
6. Get embedded documents (if not already cached)
5. Get embedded documents
Return IP address client-best server
7. Embedded documents
1. Get base document
CDN DNS server
4
Client
Origin server
2. Document with refs to embedded documents
DNS lookups
3
Regular DNS system

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

CDN-Adjustment Measure

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.

CDN-Adjustment Measure

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.

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.

CDN-Adjustment Measure

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,**
- **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 non-transparent 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.

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)]. The fact is that several solutions can be deployed, with no single one standing out as the best.

In Fig. 12-21, 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.

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 multi-tiered architecture as we discussed before.

Replication of Web Applications

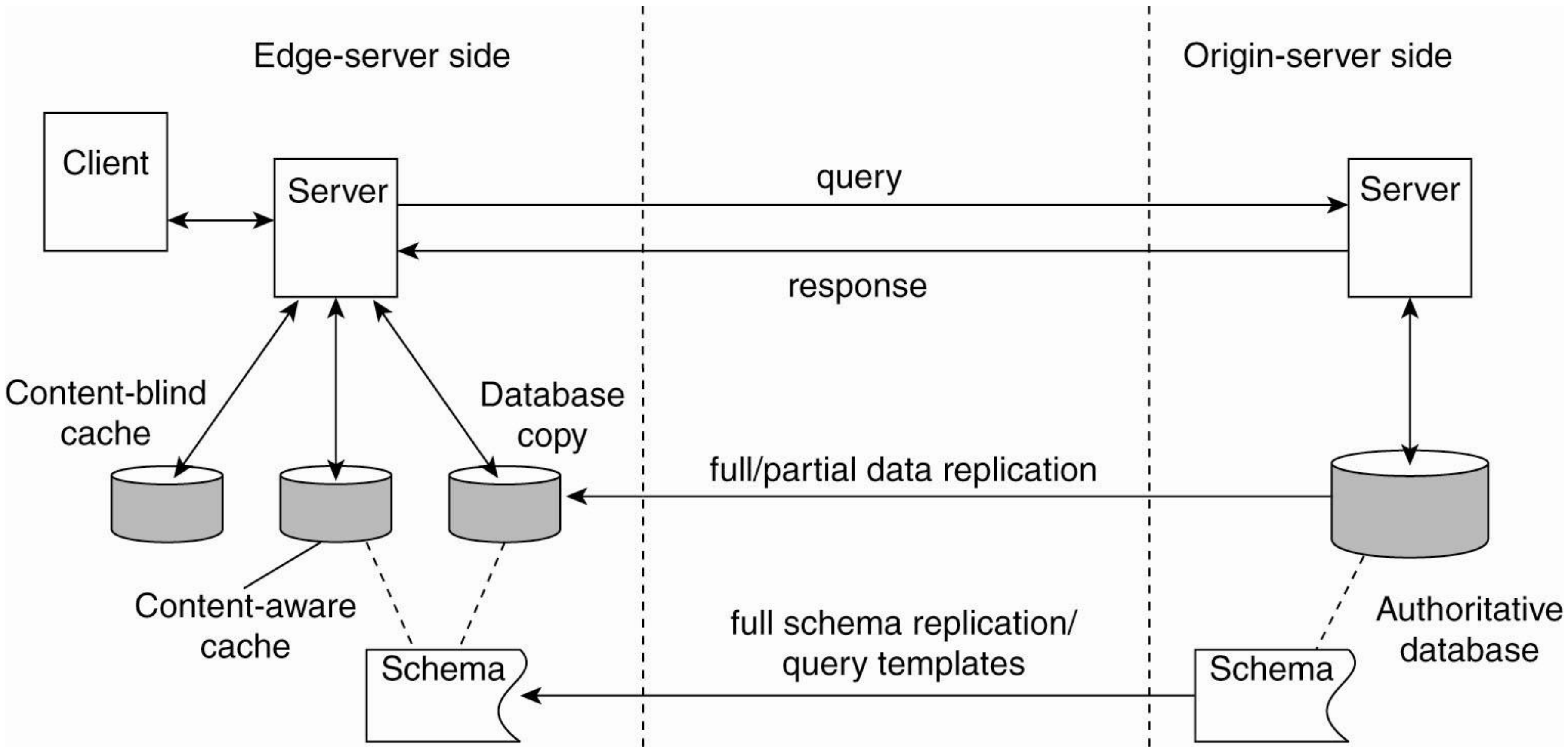


Figure 12-21. Alternatives for caching and replication with Web applications.

Replication of Web Applications

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.**

Second 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**(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**.

Replication of Web Applications
Context-aware caches

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 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**.

Replication of Web Applications
Context-aware caches

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 non-eligible 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.

Replication of Web Applications
Context-blind caches

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.

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**.

Security (1)

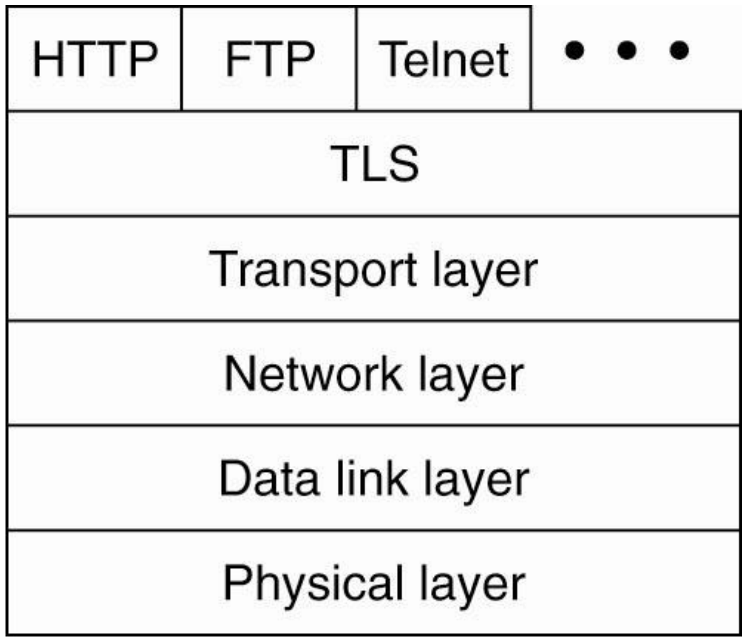


Figure 12-22. The position of TLS in the Internet protocol stack.

Security (2)

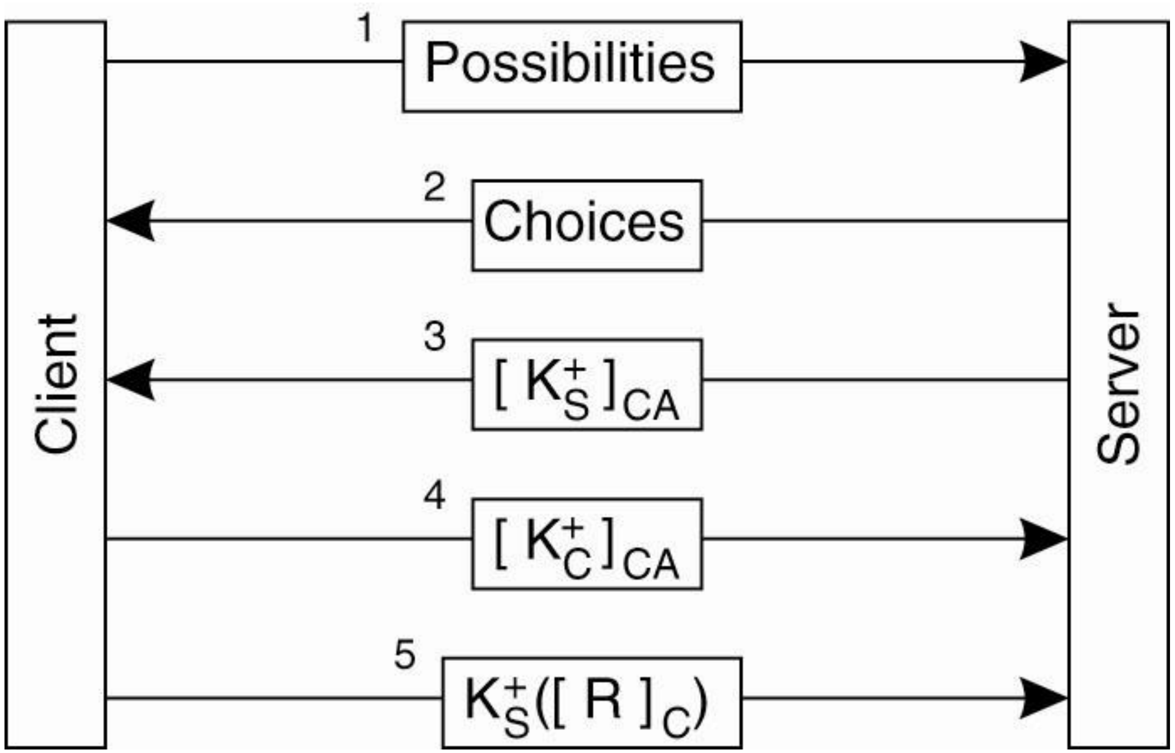


Figure 12-23. TLS with mutual authentication.