Principles of Network Applications

At the core of network application development is writing programs that run on different end systems and communicate with each other over the network. For example, in the Web application there are two distinct programs that communicate with each other: the browser program running in the user's host (desktop, laptop, tablet, Smartphone, and so on); and the Web server program running in the Web server host.

As another example, in a P2P file sharing system there is a program in each host that participates in the file-sharing community. In this case, the programs in the various hosts may be similar or identical. Thus, when developing your new application, you need to write software that will run on multiple end systems. This software could be written, for example, in C, Java, or Python. Importantly, you do not need to write software that runs on network core devices, such as routers or link-layer switches. This basic design—namely, confining application software to the end systems as shown in Figure 2.1 has facilitated the rapid development and deployment of a vast array of network applications.

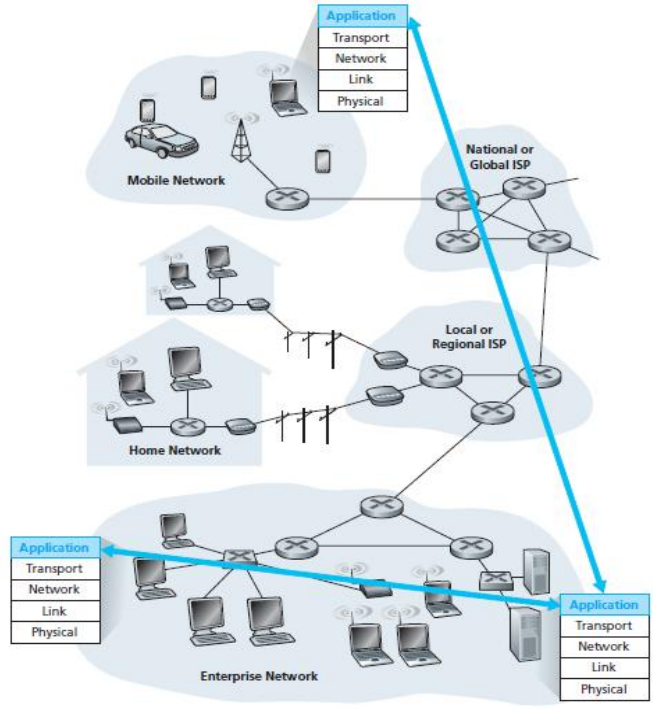


Figure 2.1 Communication for a network application takes place between end systems at the application layer

Network Application Architectures

The application architecture, on the other hand, is designed by the application developer and dictates how the application is structured over the various end systems. In choosing the application architecture, an application developer will likely draw on one of the two predominant architectural paradigms used in modern network applications:

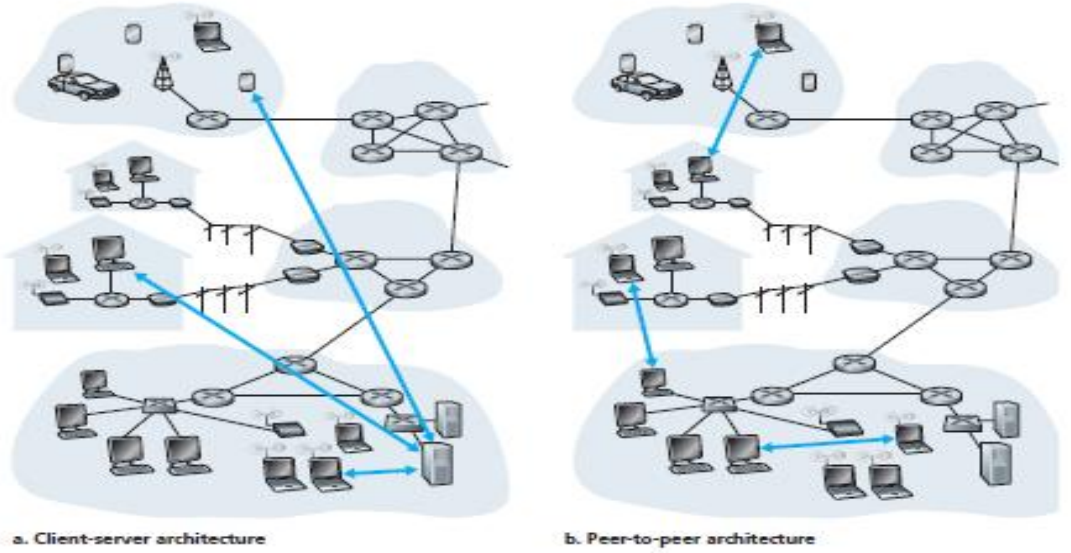
- client-server architecture
- peer-to-peer (P2P) architecture
- hybrid architecture

❖ client-server architecture:

- In a **client-server architecture**, there is an always-on host, called the *server*, which services requests from many other hosts, called *clients*.
- A classic example is the Web application for which an always-on Web server services requests from browsers running on client hosts. When a Web server receives a request for an object from a client host, it responds by sending the requested object to the client host.
- In client-server architecture, clients do not directly communicate with each other; for example, in the Web application, two browsers do not directly communicate.
- Another characteristic of the client-server architecture is that the server has a fixed, well-known address, called an IP address because the server has a fixed, well-known address, and because the server is always on, a client can always contact the server by sending a packet to the server's IP address.
- Some of the better-known applications with client-server architecture include the Web, FTP, Telnet, and e-mail.

❖ P2p architecture:

- In a P2P architecture, there is minimal (or no) reliance on dedicated servers in data centers. Instead the application exploits direct communication between pairs of intermittently connected hosts, called **peers**.
- The peers are not owned by the service provider, but are instead desktops and laptops controlled by users, with most of the peers residing in homes, universities, and offices. Because the peers communicate without passing through a dedicated server the architecture is called peer-to-peer.
- Many of today's most popular and traffic-intensive applications are based on P2P architectures. These applications include file sharing (e.g., BitTorrent), peer-assisted download acceleration (e.g., Xunlei), Internet Telephony (e.g., Skype), and IPTV (e.g.,Kankan and PPstream).
- One of the most compelling features of P2P architectures is their self-scalability. For example, in a P2P file-sharing application, although each peer generates workload by requesting files, each peer also adds service capacity to the system by distributing files to other peers.



❖ **Hybrid of client-server and P2P**

- voice-over-IP P2P application
- centralized server: finding address of remote party:
- client-client connection: direct (not through server)Instant messaging
- chatting between two users is P2P
- centralized service: client presence detection/location user registers its IP address with central server when it comes online user contacts central server to find IP addresses of buddies

Processes Communicating

A process can be thought of as a program that is running within an end system. When processes are running on the same end system, they can communicate with each other with inter-process communication, using rules that are governed by the end system's operating system.

Processes on two different end systems communicate with each other by exchanging **messages** across the computer network. A sending process creates and sends messages into the network, a receiving process receives these messages and possibly responds by sending messages back.

Client and Server Processes

A network application consists of pairs of processes that send messages to each other over a network. For example, in the Web application a client browser process exchanges messages with a Web server process. In a P2P file-sharing system, a file is transferred from a

process in one peer to a process in another peer. For each pair of communicating processes, we typically label one of the two processes as the **client** and the other process as the **server**.

With the Web, a browser is a client process and a Web server is a server process. With P2P file sharing, the peer that is downloading the file is labelled as the client, and the peer that is uploading the file is labelled as the server.

Indeed, a process in a P2P file-sharing system can both upload and download files. Nevertheless, in the context of any given communication session between a pair of processes, we can still label one process as the client and the other process as the server.

In the Web, a browser process initializes contact with a Web server process; hence the browser process is the client and the Web server process is the server.

The Interface Between the Process and the Computer Network (sockets)

Any message sent from one process to another must go through the underlying network. A process sends messages into, and receives messages from, the network through a software interface called a **socket**.

A process is analogous to a house and its socket is analogous to its door. When a process wants to send a message to another process on another host, it shoves the message out its door (socket). This sending process assumes that there is a transportation infrastructure on the other side of its door that will transport the message to the door of the destination process. Once the message arrives at the destination host, the message passes through the receiving process's door (socket), and the receiving process then acts on the message.

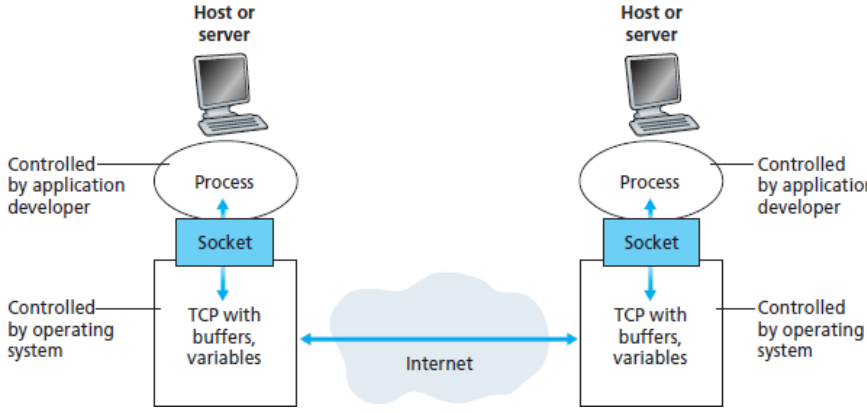


Figure 2.3 illustrates socket communication between two processes that communicate over the Internet.

As shown in this figure, a socket is the interface between the application layer and the transport layer within a host. It is also referred to as the **Application Programming Interface (API)** between the application and the network, since the socket is the programming interface with which network applications are built.
The application developer has control of everything on the application-layer side of the socket but has little control of the transport-layer side of the socket.

The only control that the Application developer has on the transport-layer side is

- ✓ The choice of transport protocol
- ✓ Perhaps the ability to fix a few transport-layer parameters such as maximum buffer and maximum segment sizes.

Addressing Processes

In order to send postal mail to a particular destination, the destination needs to have an address. Similarly, in order for a process running on one host to send packets to a process running on another host, the receiving process needs to have an address. To identify the receiving process, two pieces of information need to be specified:

- ✓ Address of the host
- ✓ Identifier that specifies the receiving process in the destination host.

This information is needed because in general a host could be running many network applications. A destination **port number** serves this purpose. Popular applications have been assigned specific port numbers. For example, a Web server is identified by port number 80. A mail server process (using the SMTP protocol) is identified by port number 25.

Transport services available to applications

We can broadly classify the possible services along four dimensions: **reliable data transfer, throughput, timing, and security.**

1. Reliable Data Transfer

- Packets can get lost within a computer network. For example, packet can overflow a buffer in a router, or can be discarded by a host or router after having some of its bits corrupted. For many applications—such as electronic mail, file transfer, remote host access, Web document transfers, and financial applications—data loss can have devastating consequences.

- Thus, to support these applications, something has to be done to guarantee that the data sent by one end of the application is delivered correctly and completely to the other end of the application. If a protocol provides such a guaranteed data delivery service, it is said to provide **reliable data transfer**.

- When a transport protocol provides this service, the sending process can just pass its data into the socket and know with complete confidence that the data will arrive without errors at the receiving process.

- When a transport-layer protocol doesn't provide reliable data transfer, some of the data sent by the sending process may never arrive at the receiving process. This may be acceptable for **loss-tolerant applications**, most notably multimedia applications such as conversational audio/video that can tolerate some amount of data loss. In these multimedia applications, lost data might result in a small glitch in the audio/video—not a crucial impairment.

2. Throughput

- Throughput is the rate at which the sending process can deliver bits to the receiving process. Because other sessions will be sharing the bandwidth along the network path, and because these other sessions will be coming and going, the available throughput can fluctuate with time.
- These observations lead to another natural service that a transport-layer protocol could provide, namely, guaranteed available throughput at some specified rate. With such a service, the application could request a guaranteed throughput of ***r* bits/sec,** and the transport protocol would then ensure that the available throughput is always **at least *r* bits/sec**. Such a guaranteed through-put service would appeal to many applications.

- Applications that have throughput requirements are said to be **bandwidth-sensitive applications**. Many current multimedia applications are bandwidth sensitive, although some multimedia applications may use adaptive coding techniques to encode digitized voice or video at a rate that matches the currently available throughput.
- While bandwidth-sensitive applications have specific throughput requirements, **elastic applications** can make use of as much, or as little, throughput as happens to be available. Electronic mail, file transfer, and Web transfers are all elastic applications.

3. Timing

- A transport-layer protocol can also provide timing guarantees. As with throughput guarantees, timing guarantees can come in many shapes and forms. An example guarantee might be that every bit that the sender pumps into the socket arrives at the receiver's socket no more than 100 msec later.
- Such a service would be appealing to interactive real-time applications, such as Internet telephony, virtual environments, teleconferencing, and multiplayer games, all of which require tight timing constraints on data delivery in order to be effective.
- Long delays in Internet telephony, for example, tend to result in unnatural pauses in the conversation.

4. Security

- Finally, a transport protocol can provide an application with one or more security services. For example, in the sending host, a transport protocol can encrypt all data transmitted by the sending process, and in the receiving host, the transport-layer protocol can decrypt the data before delivering the data to the receiving process.
- A transport protocol can also provide other security services in addition to confidentiality, including data integrity and end-point authentication.

Application	Data Loss	Throughput	Time-Sensitive
File transfer/download	No loss	Elastic	No
E-mail	No loss	Elastic	No
Web documents	No loss	Elastic (few kbps)	No
Internet telephony/ Video conferencing	Loss-tolerant	Audio: few kbps–1Mbps Video: 10 kbps–5 Mbps	Yes: 100s of msec
Streaming stored audio/video	Loss-tolerant	Same as above	Yes: few seconds
Interactive games	Loss-tolerant	Few kbps–10 kbps	Yes: 100s of msec
Instant messaging	No loss	Elastic	Yes and no

Figure 2.4 ♦ Requirements of selected network applications

Transport Services Provided by the Internet

The Internet (and, more generally, TCP/IP networks) makes two transport protocols available to applications, UDP and TCP. When you (as an application developer) create a new network application for the Internet, one of the first decisions you have to make is whether to use UDP or TCP. Each of these protocols offers a different set of services to the invoking applications.

❖ TCP Services

The TCP service model includes a connection-oriented service and a reliable data transfer service. When an application invokes TCP as its transport protocol, the application receives both of these services from TCP.

- **Connection-oriented service**

TCP has the client and server exchange transport-layer control information with each other *before* the application-level messages begin to flow. This so-called handshaking procedure alerts the client and server, allowing them to prepare for an onslaught of packets.

After the handshaking phase, a **TCP connection** is said to exist between the sockets of the two processes. The connection is a full-duplex connection in that the two processes can send

messages to each other over the connection at the same time. When the application finishes sending messages, it must tear down the connection.

- **Reliable data transfer service**

The communicating processes can rely on TCP to deliver all data sent without error and in the proper order. When one side of the application passes a stream of bytes into a socket, it can count on TCP to deliver the same stream of bytes to the receiving socket, with no missing or duplicate bytes.

TCP also includes a congestion-control mechanism, a service for the general welfare of the Internet rather than for the direct benefit of the communicating processes. The TCP congestion-control mechanism throttles a sending process (client or server) when the network is congested between sender and receiver. TCP congestion control also attempts to limit each TCP connection to its fair share of network bandwidth.

❖ UDP Services

- UDP is a no-frills, lightweight transport protocol, providing minimal services. UDP is connectionless, so there is no handshaking before the two processes start to communicate.
- UDP provides an unreliable data transfer service—that is, when a process sends a message into a UDP socket, UDP provides *no* guarantee that the message will ever reach the receiving process. Furthermore, messages that do arrive at the receiving process may arrive out of order.
- UDP does not include a congestion-control mechanism, so the sending side of UDP can pump data into the layer below (the network layer) at any rate it pleases. (Note, however, that the actual end-to-end throughput may be less than this rate due to the limited transmission

Application	Application-Layer Protocol	Underlying Transport Protocol
Electronic mail	SMTP [RFC 5321]	TCP
Remote terminal access	Telnet [RFC 854]	TCP
Web	HTTP [RFC 2616]	TCP
File transfer	FTP [RFC 959]	TCP
Streaming multimedia	HTTP (e.g., YouTube)	TCP
Internet telephony	SIP [RFC 3261], RTP [RFC 3550], or proprietary (e.g., Skype)	UDP or TCP

Figure 2.5 ♦ Popular Internet applications, their application-layer protocols, and their underlying transport protocols

=.

The Web and HTTP

Until the early 1990s the Internet was used primarily by researchers, academics, and university students to log in to remote hosts, to transfer files from local hosts to remote hosts and vice versa, to receive and send news, and to receive and send electronic mail. The Web was the first Internet application that caught the general public's eye. It dramatically changed, and continues to change, how people interact inside and outside their work environments.

Overview of HTTP

The **Hypertext Transfer Protocol (HTTP)**, the Web's application-layer protocol, is at the heart of the Web. HTTP is implemented in two programs: a client program and a server program. The client program and server program, executing on different end systems, talk to each other by exchanging HTTP messages.

- **Web page** (also called a document) consists of objects. An **object** is simply a file such as an HTML file, a JPEG image, a Java applet, or a video clip—that is addressable by a single URL.
- Most Web pages consist of a **base HTML file** and several referenced objects. For example, if a Web page contains HTML text and five JPEG images, then the Web page has six objects: the base HTML file plus the five images.

Each URL has two components: i. **The hostname of the server that houses the object**

ii. **The object's path name. For example, the URL**

Example: **http://www.someSchool.edu/someDepartment/picture.gif**

www.someSchool.edu for a hostname and **/some Department/ picture.gif** for a path name.

Because **Web browsers** (such as Internet Explorer and Firefox) implement the client side of HTTP, in the context of the Web. **Web servers**, which implement the server side of HTTP. Popular Web servers include Apache and Microsoft Internet Information Server.

HTTP defines how Web clients request Web pages from Web servers and how servers transfer Web pages to clients.

The general idea is illustrated in **Figure 2.6.** When a user requests a Web page the browser sends HTTP request messages for the objects in the page to the server. The server receives the requests and responds with HTTP response messages that contain the objects.

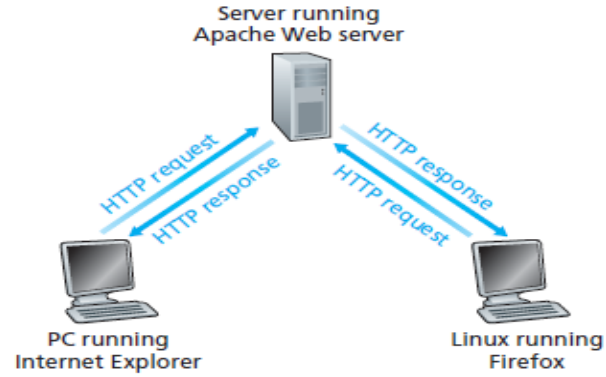


Figure 2.6 ♦ HTTP request-response behavior

- HTTP uses TCP as its underlying transport protocol (rather than running on top of UDP). The HTTP client first initiates a TCP connection with the server. Once the connection is established, the browser and the server processes access TCP through their socket interfaces.
- On the client side the socket interface is the door between the client process and the TCP connection; on the server side it is the door between the server process and the TCP connection.
- The client sends HTTP request messages into its socket interface and receives HTTP response messages from its socket interface. Similarly, the HTTP server receives request messages from its socket interface and sends response messages into its socket interface. Once the client sends a message into its socket interface, the message is out of the client's hands and is "in the hands" of TCP.
- It is important to note that the server sends requested files to clients without storing any state information about the client. If a particular client asks for the same object twice in a period of a few seconds, the server does not respond by saying that it just served the object to the client; instead, the server resends the object, as it has completely forgotten what it did earlier.
- Because an HTTP server maintains no information about the clients, HTTP is said to be a **stateless protocol**.

Non-Persistent and Persistent Connections

In many Internet applications, the client and server communicate for an extended period of time, with the client making a series of requests and the server responding to each of the requests. Depending on the application and on how the application is being used, the series of requests may be made back-to-back, periodically at regular intervals, or intermittently.

When this client-server interaction is taking place over TCP, the application developer needs to make an important decision—

- Should each request/response pair be sent over a *separate* TCP connection?
- Should all of the requests and their corresponding responses be sent over the *same* TCP connection?

In the former approach, the application is said to use **Non-persistent connections**; and in the latter approach, **persistent connections**. HTTP, which can use both non-persistent connections and persistent connections. Although HTTP uses persistent connections in its default mode.

HTTP with Non-Persistent Connections

The steps of transferring a Web page from server to client for the case of non-persistent connections. Let's suppose the page consists of a base HTML file and 10 JPEG images, and that all 11 of these objects reside on the same server.

Further suppose the URL for the base HTML file is
http://www.someSchool.edu/someDepartment/home.index

The steps below illustrate the use of non-persistent connections, where each TCP connection is closed after the server sends the object—the connection does not persist for other objects.

1. The HTTP client process initiates a TCP connection to the server www.someSchool.edu on port number 80, which is the default port number for HTTP. Associated with the TCP connection, there will be a socket at the client and a socket at the server.
2. The HTTP client sends an HTTP request message to the server via its socket. The request message includes the path name **/some Department/home. Index**
3. The HTTP server process receives the request message via its socket, retrieves the object **some Department/home. Index** from its storage (RAM or disk), encapsulates the object in an HTTP response message, and sends the response message to the client via its socket.
4. The HTTP server process tells TCP to close the TCP connection. (But TCP doesn't actually terminate the connection until it knows for sure that the client has received the response message intact.)
5. The HTTP client receives the response message. The TCP connection terminates. The message indicates that the encapsulated object is an HTML file. The client extracts the file from the response message, examines the HTML file, and finds references to the 10 JPEG objects.

Thus, in this example, when a user requests the Web page, 11 TCP connections are generated. Parallel connections can be set to one, in which case the 10 connections are established serially.

The use of parallel connections shortens the response time. The **round-trip time (RTT)**, which is the time it takes for a small packet to travel from client to server and then back to the client. The RTT includes **packet-propagation delays, packet-queuing delays in intermediate routers and switches, and packet-processing delays.**

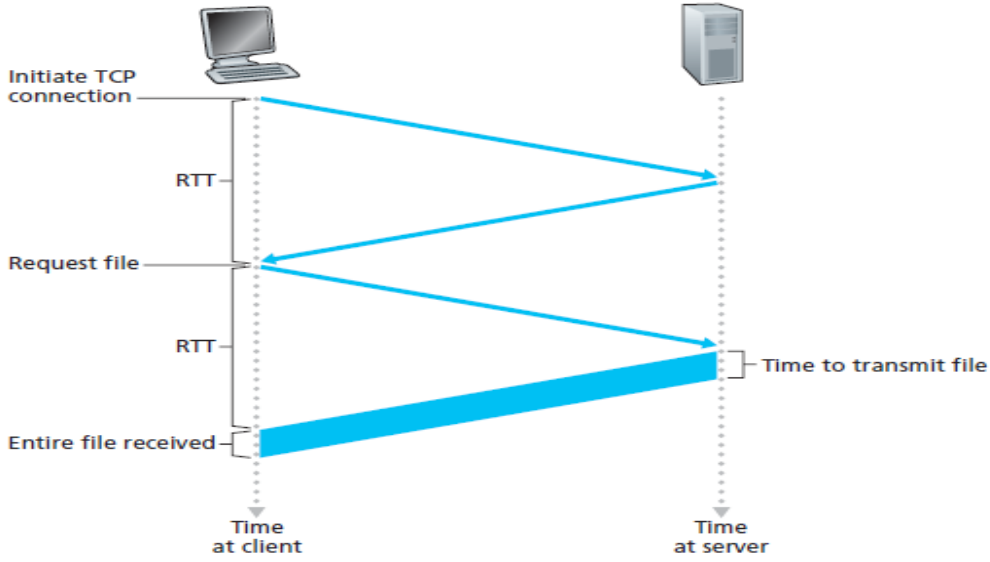


Figure 2.7 ♦ Back-of-the-envelope calculation for the time needed to request and receive an HTML file

- This causes the browser to initiate a TCP connection between the browser and the Web server; this involves a "three-way handshake" the client sends a small TCP segment to the server, the server acknowledges and responds with a small TCP segment, and, finally, the client acknowledges back to the server.
- The first two parts of the three-way handshake take one RTT. After completing the first two parts of the hand-shake, the client sends the HTTP request message combined with the third part of the three-way handshake (the acknowledgment) into the TCP connection.
- Once the request message arrives at the server, the server sends the HTML file into the TCP connection. This HTTP request/response eats up another RTT.
- Thus, roughly, the total response time is **two RTTs plus the transmission time at the server of the HTML file**

HTTP with Persistent Connections

Non-persistent connections have some shortcomings.

- First, a brand-new connection must be established and maintained for each requested object. For each of these connections, TCP buffers must be allocated and TCP variables must be kept in both the client and server. This can place a significant burden on the Web server, which may be serving requests from hundreds of different clients simultaneously.
- Second, as we just described, each object suffers a delivery delay of two RTTs— one RTT to establish the TCP connection and one RTT to request and receive an object.

  - With persistent connections, the server leaves the TCP connection open after sending a response. Subsequent requests and responses between the same client and server can be sent over the same connection. These requests for objects can be made back-to-back, without waiting for replies to pending requests (pipelining). Typically, the HTTP server closes a connection when it isn't used for a certain time (a configurable timeout interval). When the server receives the back-to-back requests it sends the objects back-to-back.

The default mode of HTTP uses **persistent connections with pipelining**.

HTTP Message Format

There are two types of HTTP messages, request messages and response messages, both of which are discussed below.

HTTP Request Message

GET /somedir/page.html HTTP/1.1

Host: www.someschool.edu

Connection: close

User-agent: Mozilla/5.0

Accept-language: fr

- The first line of an HTTP request message is called the **request line**; the subsequent lines are called the **header lines**. The request line has three fields: the method field, the URL field, and the HTTP version field. The method field can take on several different values, including `GET`, `POST`, `HEAD`, `PUT`, and `DELETE`.

- The **`GET`** method is used when the browser requests an object, with the requested object identified in the URL field.

- The header line **`Host: www.someschool.edu`** specifies the host on which the object resides. You might think that this header line is unnecessary, as there is already a TCP connection in place to the host.

- **`Connection: close`** header line, the browser is telling the server that it doesn't want to bother with persistent connections; it wants the server to close the connection after sending the requested object.

- The **`User-agent:`** header line specifies the user agent, that is, the browser type that is making the request to the server. Here the user agent is Mozilla/5.0, a Firefox browser.

- Finally, the **`Accept-language:`** header indicates that the user prefers to receive a French version of the object, if such an object exists on the server; otherwise, the server should send its default version.

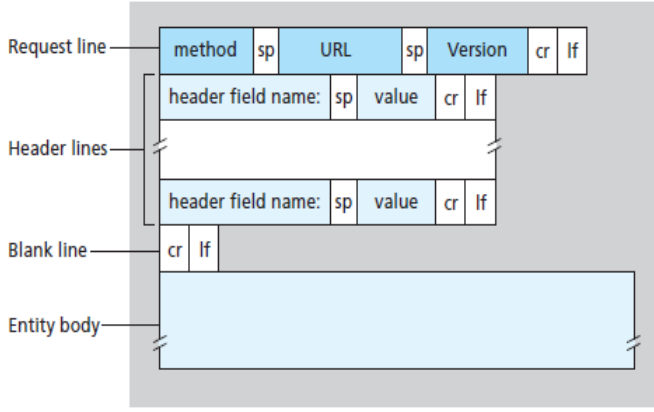


Figure 2.8 ♦ General format of an HTTP request message

HTTP Response Message

HTTP/1.1 200 OK
Connection: close
Date: Tue, 09 Aug 2011 15:44:04 GMT
Server: Apache/2.2.3 (CentOS)
Last-Modified: Tue, 09 Aug 2011 15:11:03 GMT
Content-Length: 6821
Content-Type: text/html (data data data data data ...)

It has three sections: an initial **status line**, six **header lines**, and then the **entity body**. The entity body is the meat of the message—it contains the requested object itself (represented by data data data data data ...)

- The status line has three fields: the protocol version field, a status code, and a corresponding status message. In this example, the status line indicates that the server is using HTTP/1.1 and that everything is OK
- **Connection: close** header line to tell the client that it is going to close the TCP connection after sending the message.
- **Date: header line** indicates the time and date when the HTTP response was created and sent by the server. it is the time when the server retrieves the object from its file system, inserts the object into the response message, and sends the response message.
- **The Server:** header line indicates that the message was generated by an Apache Web server; it is analogous to the User-agent: header line in the HTTP request message. The Last-Modified: header line indicates the time and date when the object was created or last modified.
- **The Last-Modified:** header, which we will soon cover in more detail, is critical for object caching, both in the local client and in network cache servers (also known as proxy servers).
- **The Content-Length:** header line indicates the number of bytes in the object being sent.

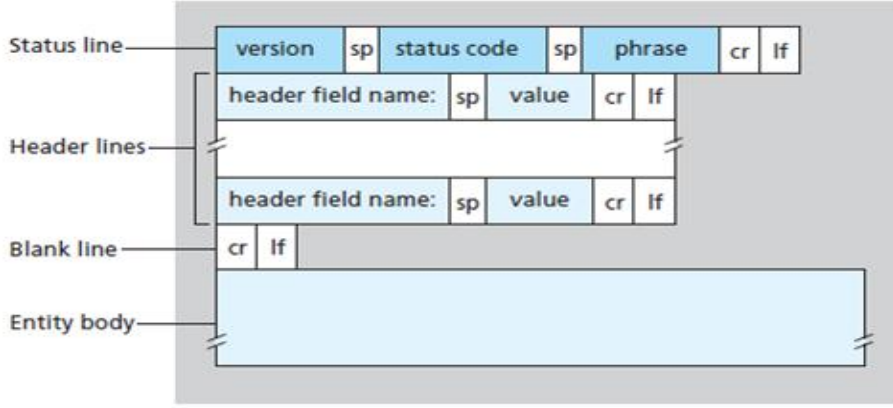


Figure 2.9 ♦ General format of an HTTP response message

- **Content-Type:** header line indicates that the object in the entity body is HTML text.

The status code and associated phrase indicate the result of the request. Some common status codes and associated phrases include:

- **200 OK:** Request succeeded and the information is returned in the response.
- **301 Moved Permanently:** Requested object has been permanently moved the new URL is specified in Location: header of the response message. The client software will automatically retrieve the new URL.
- **400 Bad Request:** This is a generic error code indicating that the request could not be understood by the server.
- **404 Not Found:** The requested document does not exist on this server.
- **505 HTTP Version Not Supported:** The requested HTTP protocol version is not supported by the server.

User-Server Interaction: Cookies

HTTP server is stateless. This simplifies server design and has permitted engineers to develop high-performance Web servers that can handle thousands of simultaneous TCP connections. However, it is often desirable for a Web site to identify users, either because the server wishes to restrict user access or because it wants to serve content as a function of the user identity. For these purposes, HTTP uses cookies. Cookies, allow sites to keep track of users. Most major commercial Web sites use cookies today.

Cookie technology has four components:

- cookie header line in the HTTP response message;
- cookie header line in the HTTP request message;
- cookie file kept on the user's end system and managed by the user's browser
- back-end database at the Web site. .

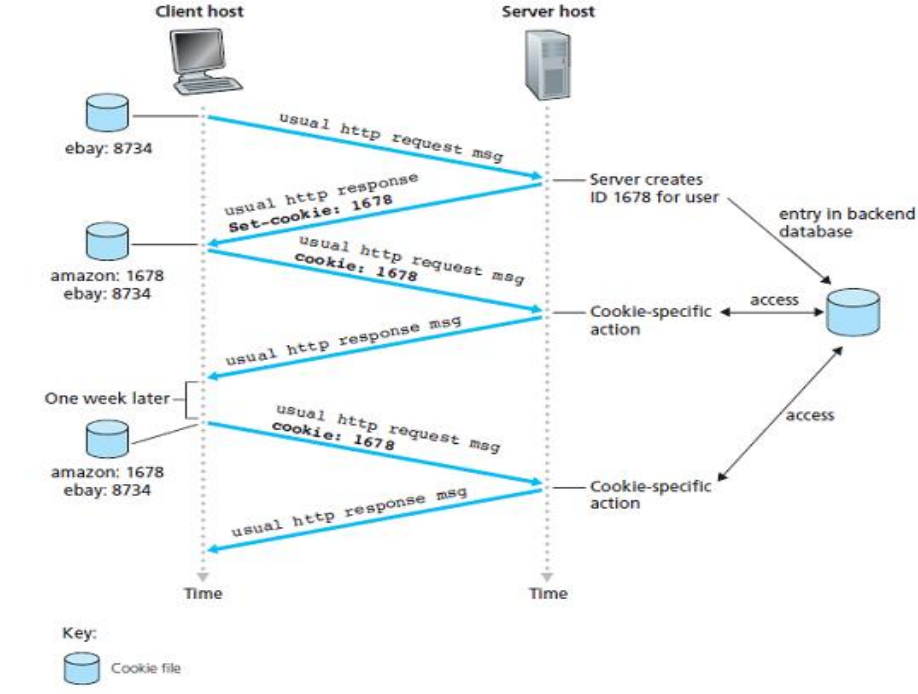


Figure 2.10 ♦ Keeping user state with cookies

Suppose Susan, who always accesses the Web using Internet Explorer from her home PC, contacts Amazon.com for the first time. Let us suppose that in the past she has already visited the eBay site. When the request comes into the Amazon Web server, the server creates a unique identification number and creates an entry in its back-end database that is indexed by the identification number.

The Amazon Web server then responds to Susan's browser, including in the HTTP response a Set-cookie: header, which contains the identification number. For example, the header line might be: **Set-cookie: 1678**

When Susan's browser receives the HTTP response message, it sees the Setcookie: header. The browser then appends a line to the special cookie file that it manages. This line includes the hostname of the server and the identification number in the Set-cookie: header. Note that the cookie file already has an entry for eBay.

Since Susan has visited that site in the past. As Susan continues to browse the Amazon site, each time she requests a Web page, her browser consults her cookie file, extracts her identification number for this site, and puts a cookie header line that includes the identification number in the HTTP request. Specifically, each of her HTTP requests to the Amazon server includes the header line: **Cookie: 1678**

Web Caching

A **Web cache**—also called a **proxy server**—is a network entity that satisfies HTTP requests on the behalf of an origin Web server. The Web cache has its own disk storage and keeps copies of recently requested objects in this storage. As shown in Figure a user's browser can be configured so that all of the user's HTTP requests are first directed to the Web cache.

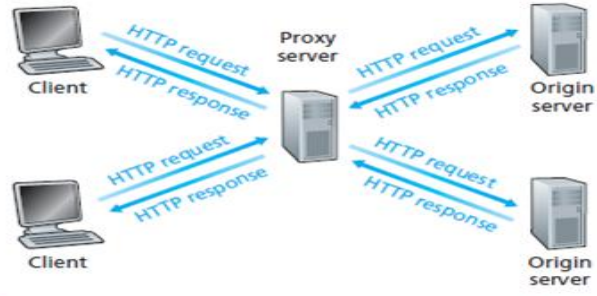


Figure 2.11 ♦ Clients requesting objects through a Web cache

Once a browser is configured, each browser request for an object is first directed to the Web cache. As an example, suppose a browser is requesting the object **http://www.someschool.edu/campus.gif.**

Here is what happens:

- The browser establishes a TCP connection to the Web cache and sends an HTTP request for the object to the Web cache.
- The Web cache checks to see if it has a copy of the object stored locally. If it does, the Web cache returns the object within an HTTP response message to the client browser.
- If the Web cache does not have the object, the Web cache opens a TCP connection to the origin server, that is, to www.someschool.edu. The Web cache then sends an HTTP request

for the object into the cache-to-server TCP connection. After receiving this request, the origin server sends the object within an HTTP response to the Web cache.

- When the Web cache receives the object, it stores a copy in its local storage and sends a copy, within an HTTP response message, to the client browser.

Web caching has seen deployment in the Internet for two reasons.

- ✓ First, a Web cache can substantially reduce the response time for a client request, if the cache has the requested object, then the cache will be able to deliver the object rapidly to the client.
- ✓ Second Web caches can substantially reduce traffic on an institution's access link to the Internet. By reducing traffic, the institution does not have to upgrade bandwidth as quickly, thereby reducing costs.

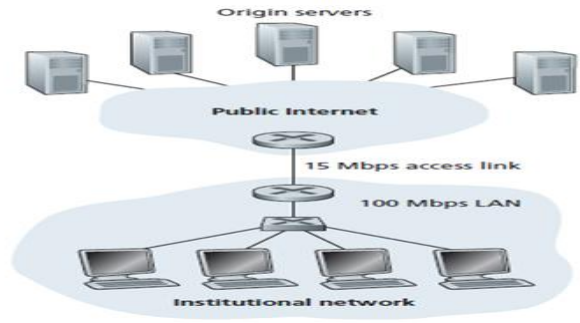


Fig: Bottleneck between an institutional network and the Internet

This figure shows two networks—the institutional network and the rest of the public Internet. The institutional network is a high-speed LAN. A router in the institutional network and a router in the Internet are connected by a 15 Mbps link. The origin servers are attached to the Internet but are located all over the globe. Suppose that the average object size is 1 Mbits and that the average request rate from the institution's browsers to the origin servers is 15 requests per second.

The total response time—that is, the time from the browser's request of an object until its receipt of the object—is the sum of the LAN delay, the access delay (that is, the delay between the two routers), and the Internet delay. Let's now do a very crude calculation to estimate this delay.

The traffic intensity on the LAN is:

(15 requests/sec) *(1 Mbits/request)/ (100 Mbps) = 0.15

whereas the **traffic intensity on the access link is:**

15 requests/sec) * (1 Mbits/request)/(15 Mbps) = 1

The delay on a link becomes very large and grows without bound. Thus, the average response time to satisfy requests is going to be on the order of minutes.

One possible solution is to increase the access rate from 15 Mbps to, say, 100 Mbps. This will lower the traffic intensity on the access link to 0.15, which translates to negligible delays between the two routers. In this case, the total response time will roughly be two seconds, that is, the Internet delay. But this solution also means that the institution must upgrade its access link from 15 Mbps to 100 Mbps, a costly proposition.

Now consider the alternative solution of not upgrading the access link but instead installing a Web cache in the institutional network. This solution is illustrated in Figure 2.13.

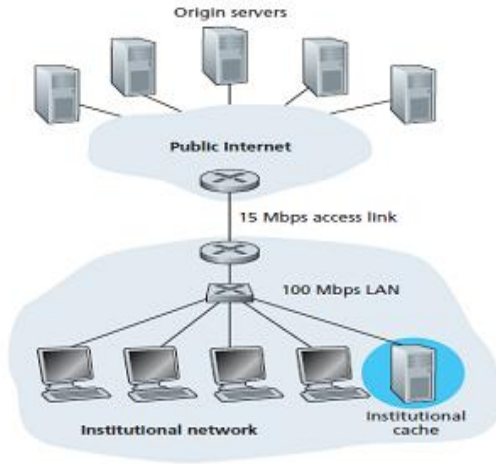


Figure 2.13 ♦ Adding a cache to the institutional network

Hit rates—the fraction of requests that are satisfied by a cache—typically range from 0.2 to 0.7 in practice. For illustrative purposes, let's suppose that the cache provides a hit rate of 0.4 for this institution.

This delay is negligible compared with the two second Internet delay.

Given these considerations, average delay therefore is

0.4 _ (0.01 seconds) + 0.6 _ (2.01 seconds) which is just slightly greater than 1.2 seconds

The Conditional GET

Although caching can reduce user-perceived response times, it introduces a new problem the copy of an object residing in the cache may be stale. In other words, the object housed in the Web server may have been modified since the copy was cached at the client. Fortunately, HTTP has a mechanism that allows a cache to verify that its objects are up to date. This mechanism is called the **conditional GET**.

An HTTP request message is a so-called conditional GET message if

(1) the request message uses the GET method and
(2) the request message includes an If-Modified- Since: header line.

To illustrate how the conditional GET operates, let's walk through an example.

- First, on the behalf of a requesting browser, a proxy cache sends a request message to a Web server:

  GET /fruit/kiwi.gif HTTP/1.1
  Host: www.exotiquecuisine.com

- Second, the Web server sends a response message with the requested object to the

  cache:HTTP/1.1 200 OK
  Date: Sat, 8 Oct 2011 15:39:29
  Server: Apache/1.3.0 (Unix)
  Last-Modified: Wed, 7 Sep 2011 09:23:24
  Content-Type: image/gif
  (data data data data data ...)

The cache forwards the object to the requesting browser but also caches the object locally. Importantly, the cache also stores the last-modified date along with the object.

- Third, one week later, another browser requests the same object via the cache, and the object is still in the cache. Since this object may have been modified at the Web server in the past week, the cache performs an up-to-date check by issuing a conditional GET. Specifically, the cache sends:

GET /fruit/kiwi.gif HTTP/1.1
Host: www.exotiquecuisine.com
If-modified-since: Wed, 7 Sep 2011 09:23:24

Note that the value of the If-modified-since: header line is exactly equal to the value of the Last-Modified: header line that was sent by the server one week ago.

This conditional GET is telling the server to send the object only if the object has been modified since the specified date. Suppose the object has not been modified since 7 Sep 2011 09:23:24

- Then, fourth, the Web server sends a response message to the cache:

  HTTP/1.1 304 Not Modified
  Date: Sat, 15 Oct 2011 15:39:29
  Server: Apache/1.3.0 (Unix) (empty entity body

File Transfer: FTP

In a typical FTP session, the user is sitting in front of one host (the local host) and wants to transfer files to or from a remote host. In order for the user to access the remote account, the user must provide a user identification and a password.

After providing this authorization information, the user can transfer files from the local file system to the remote file system and vice versa.

As shown in Figure 2.14, the user interacts with FTP through an FTP user agent.

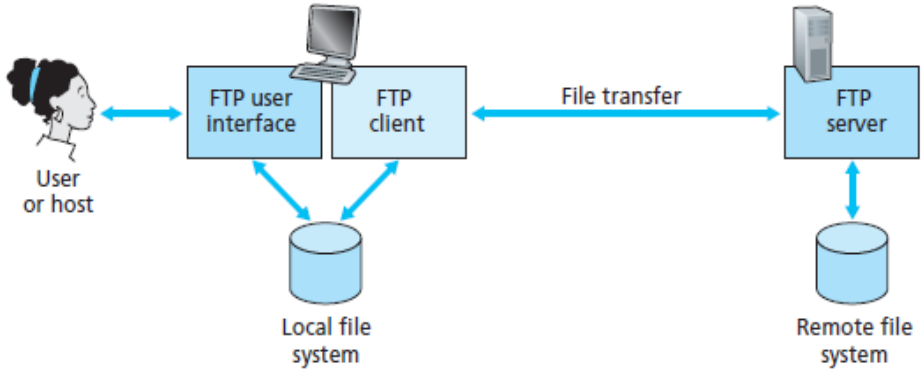


Figure 2.14 ♦ FTP moves files between local and remote file systems

- The user first provides the hostname of the remote host, causing the FTP client process in the local host to establish a TCP connection with the FTP server process in the remote host.
- The user then provides the user identification and password, which are sent over the TCP connection as part of FTP commands. Once the server has authorized the user, the user copies one or more files stored in the local file system into the remote file system (or vice versa).
- FTP uses two parallel TCP connections to transfer a file, a **control connection** and a **data connection**.

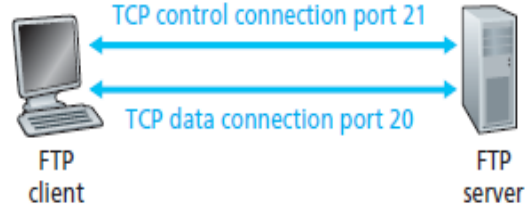


Figure 2.15 ♦ Control and data connections

- ✓ **control connection** is used for sending control information between the two host information such as user identification, password, commands to change remote directory, and commands to "put" and "get" files.
- ✓ **data connection** is used to actually send a file. Because FTP uses a separate control connection, FTP is said to send its control information **out-of-band**.

- Throughout a session, the FTP server must maintain **state** about the user. In particular, the server must associate the control connection with a specific user account, and the server must keep track of the user's current directory as the user wanders about the remote directory tree.
- When a user starts an FTP session with a remote host, the client side of FTP (user) first initiates a control TCP connection with the server side (remote host) on server port number21.

- The client side of FTP sends the user identification and password over this control connection. The client side of FTP also sends, over the control connection, commands to change the remote directory. When the server side receives a command for a file transfer over the control connection (either to, or from, the remote host), the server side initiates a TCP data connection to the client side.

- FTP sends exactly one file over the data connection and then closes the data connection. If, during the same session, the user wants to transfer another file, FTP opens another data connection. Thus, with FTP, the control connection remains open throughout the duration of the user session, but a new data connection is created for each file transferred within a session (that is, the data connections are non-persistent).

FTP Commands and Replies

Some of the more common commands are given below:

• **USER username:** Used to send the user identification to the server.

• **PASS password:** Used to send the user password to the server.

• **LIST:** Used to ask the server to send back a list of all the files in the current remote directory. The list of files is sent over a (new and non-persistent) data connection rather than the control TCP connection.

• **RETR filename:** Used to retrieve (that is, get) a file from the current directory of the remote host. This command causes the remote host to initiate a data connection and to send the requested file over the data connection.

• **STOR filename:** Used to store (that is, put) a file into the current directory of the remote host. There is typically a one-to-one correspondence between the command that the user issues and the FTP command sent across the control connection.

REPLIES

Each command is followed by a reply, sent from server to client. The replies are three-digit numbers, with an optional message following the number. Some typical replies, along with their possible messages are as follows:

- **331 Username OK, password required**
- **125 Data connection already open; transfer starting**
- **425 Can't open data connection**
- **452 Error writing file**

Electronic Mail in the Internet [E-MAIL]

Electronic mail has been around since the beginning of the Internet. It was the most popular application when the Internet was in its infancy [Segaller 1998], and has become more and more elaborate and powerful over the years.

We see from this diagram that it has three major components: **user agents**, **mail servers**, and the **Simple Mail Transfer Protocol (SMTP)**.

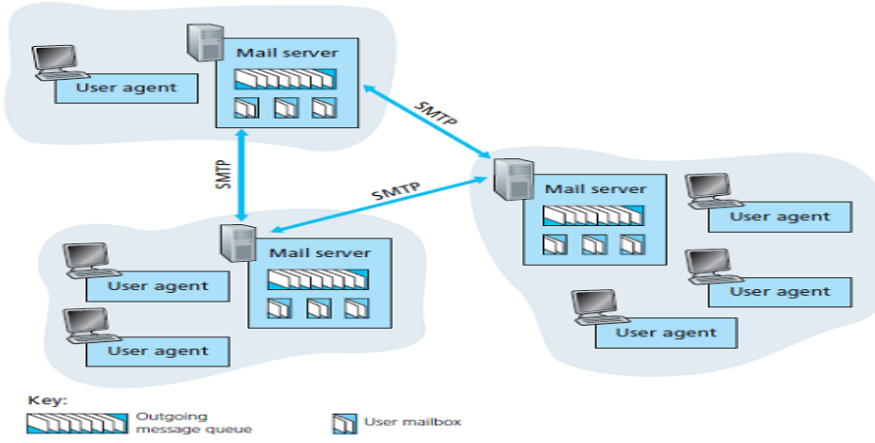


Figure 2.16 ◆ A high-level view of the Internet e-mail system

Figure 2.16 presents a high-level view of the Internet mail system.

- Alice, sending an e-mail message to a recipient, Bob. User agents allow users to read, reply to, forward, save, and compose messages. Microsoft Outlook and Apple Mail are examples of user agents for e-mail.
- When Alice is finished composing her message, her user agent sends the message to her mail server, where the message is placed in the mail server's outgoing message queue. When Bob wants to read a message, his user agent retrieves the message from his mailbox in his mail server.
- Mail servers form the core of the e-mail infrastructure. Each recipient, such as Bob, has a **mailbox** located in one of the mail servers. Bob's mailbox manages and maintains the messages that have been sent to him.

- A typical message starts its journey in the sender's user agent, travels to the sender's mail server, and travels to the recipient's mail server, where it is deposited in the recipient's mailbox.
- When Bob wants to access the messages in his mailbox, the mail server containing his mailbox authenticates Bob (with usernames and passwords). Alice's mail server must also deal with failures in Bob's mail server.
- If Alice's server cannot deliver mail to Bob's server, Alice's server holds the message in a **message queue** and attempts to transfer the message later. Reattempts are often done every 30 minutes or so; if there is no success after several days, the server removes the message and notifies the sender (Alice) with an e-mail message.

SMTP

SMTP, defined in RFC 5321, is at the heart of Internet electronic mail. As mentioned above, SMTP transfers messages from senders' mail servers to the recipients' mail servers. SMTP is much older than HTTP. This restriction made sense in the early 1980s when transmission capacity was scarce and no one was e-mailing large attachments or large image, audio, or video files. But today, in the multimedia era, the 7-bit ASCII restriction is a bit of a pain—it requires binary multimedia data to be encoded to ASCII before being sent over SMTP; and it requires the corresponding ASCII message to be decoded back to binary after SMTP transport.

To illustrate the basic operation of SMTP, let's walk through a common scenario. Suppose Alice wants to send Bob a simple ASCII message.

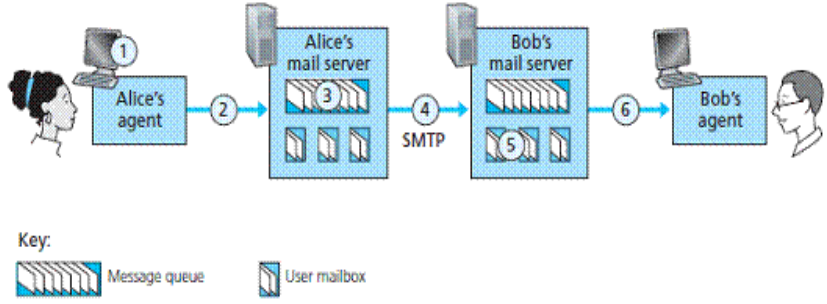


Figure 2.17 ♦ Alice sends a message to Bob

1. Alice invokes her user agent for e-mail, provides Bob's e-mail address (for example, bob@someschool.edu), composes a message, and instructs the user agent to send the message.
2. Alice's user agent sends the message to her mail server, where it is placed in a message queue.
3. The client side of SMTP, running on Alice's mail server, sees the message in the message queue. It opens a TCP connection to an SMTP server, running on Bob's mail server.
4. After some initial SMTP handshaking, the SMTP client sends Alice's message into the TCP connection.

5. At Bob's mail server, the server side of SMTP receives the message. Bob's mail server then places the message in Bob's mailbox.
6. Bob invokes his user agent to read the message at his convenience.

First, the client SMTP (running on the sending mail server host) has TCP establish a connection to port 25 at the server SMTP (running on the receiving mail server host).
If the server is down, the client tries again later. Once the SMTP client and server have introduced themselves to each other, the client sends the message.

SMTP can count on the reliable data transfer service of TCP to get the message to the server without errors. The client then repeats this process over the same TCP connection if it has other messages to send to the server; otherwise, it instructs TCP to close the connection.

Let's next take a look at an example transcript of messages exchanged between an SMTP client (C) and an SMTP server (S). The hostname of the client is crepes.fr and the hostname of the server is hamburger.edu. The ASCII text lines prefaced with C: are exactly the lines the client sends into its TCP socket, and the ASCII text lines prefaced with S: are exactly the lines the server sends into its TCP socket.
The following transcript begins as soon as the TCP connection is established.

S: 220 hamburger.edu
C: HELO crepes.fr
S: 250 Hello crepes.fr, pleased to meet you
C: MAIL FROM: <alice@crepes.fr>
S: 250 alice@crepes.fr ... Sender ok
C: RCPT TO: <bob@hamburger.edu>
S: 250 bob@hamburger.edu ... Recipient ok
C: DATA
S: 354 Enter mail, end with "." on a line by itself
C: Do you like ketchup?
C: How about pickles?
C: .
S: 250 Message accepted for delivery
C: QUIT
S: 221 hamburger.edu closing connection

In the example above, the client sends a message ("Do you like ketchup?
How about pickles?") from mail server crepes.fr to mail server hamburger. edu. As part of the dialogue, the client issued five commands: **HELO (an abbreviation for HELLO), MAIL FROM, RCPT TO, DATA, and QUIT.** These commands are self-explanatory. If the sending mail server has several messages to send to the same receiving mail server, it can send all of the messages over the same TCP connection. For each message, the

client begins the process with new MAIL FROM: crepes.fr, designates the end of message with an isolated period, and issues QUIT only after all messages have been sent.

It is highly recommended that you use Telnet to carry out a direct dialogue with an SMTP server. To do this, issue telnet server Name 25 where server Name is the name of a local mail server. When you do this, you are simply establishing a TCP connection between your local host and the mail server. After typing this line, you should immediately receive the 220 reply from the server.

Difference between HTTP and SMTP

Let's Both protocols are used to transfer files from one host to another:

- ✓ HTTP transfers files (also called objects) from a Web server to a Web client (typically a browser);
- ✓ SMTP transfers files (that is, e-mail messages) from one mail server to another mail server. When transferring the files, both HTTP and SMTP use persistent connections.
- ✓

- First, **HTTP** is mainly a **pull protocol** someone loads information on a Web server and users use HTTP to pull the information from the server at their convenience. On the other hand, **SMTP** is primarily a **push protocol**—the sending mail server pushes the file to the receiving mail server.
- A second difference, is that **SMTP** requires each message, including the body of each message, to be in **7-bit ASCII format**. If the message contains characters that are not 7-bit ASCII (for example, French characters with accents) or contains binary data (such as an image file), then the message has to be encoded into 7-bit ASCII. **HTTP** data does not impose this restriction.
- A third important difference concerns how a document consisting of text and images (along with possibly other media types) is handled. HTTP encapsulates each object in its own HTTP response message. Internet mail places all of the message's objects into one message.

Mail Access Protocols

Given that Bob (the recipient) executes his user agent on his local PC, it is natural to consider placing a mail server on his local PC as well. With this approach, This could be done simply by having Alice's user agent send the message directly to Bob's mail server. And this could be done with SMTP—indeed; SMTP has been designed for pushing e-mail from one host to another. However, typically the sender's user agent does not dialogue directly with the recipient's mail server.

Instead, as shown in Figure 2.18, Alice's user agent uses SMTP to push the e-mail message into her mail server, then Alice's mail server uses SMTP (as an SMTP client) to relay the e-mail message to Bob's mail server.

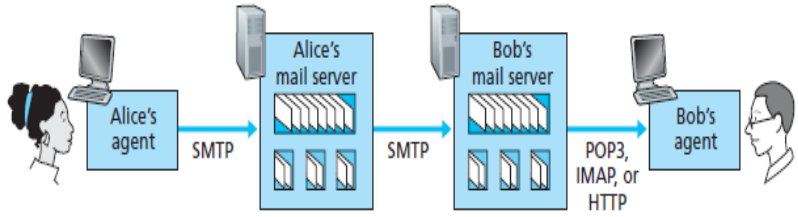


Figure 2.18 ♦ E-mail protocols and their communicating entities

There are currently a number of popular mail access protocols, including
Post Office Protocol—Version 3 (POP3), **Internet Mail Access Protocol (IMAP)**, **and HTTP.**

1.POP3

POP3 is an extremely simple mail access protocol. It is defined in [RFC 1939], which is short and quite readable. Because the protocol is so simple, its functionality is rather limited. POP3 begins when the user agent (the client) opens a TCP connection to the mail server (the server) on port 110.
With the TCP connection established,POP3 progresses through three phases:
Authorization, transaction, and update.

- During the first phase, authorization, the user agent sends a username and a password (in the clear) to authenticate the user.
- During the second phase, transaction, the user agent retrieves messages; also during this phase, the user agent can mark messages for deletion, remove deletion marks, and obtain mail statistics.
- The third phase, update, occurs after the client has issued the quit command, ending the POP3 session; at this time, the mail server deletes the messages that were marked for deletion.

In a POP3 transaction, the user agent issues commands, and the server responds to each command with a reply.
There are two possible responses: **+OK** used by the server to indicate that the previous command was fine; and **-ERR,** used by the server to indicate that something was wrong with the previous command.
The authorization phase has two principal commands: user <username> and pass <password>. To illustrate these two commands, we suggest that you Telnet directly into a POP3 server, using port 110, and issue these commands. Suppose that mail Server is the name of your mail server. You will see something like:
telnet mailServer 110
+OK POP3 server ready

user bob
+OK
pass hungry
+OK user successfully logged on

Note that after the authorization phase, the user agent employed only four commands**: list, retr, dele, and quit.** After processing the quit command, the POP3 server enters the update phase and removes messages 1 and 2 from the mailbox. A problem with this **download-and-delete mode** is that the recipient, Bob, may be nomadic and may want to access his mail messages from multiple machines, for example, his office PC, his home PC, and his portable computer.

- The **download and- delete mode** partitions Bob's mail messages over these three machines; in particular, if Bob first reads a message on his office PC, he will not be able to reread the message from his portable at home later in the evening.
- **In the download-and keep mode,** the user agent leaves the messages on the mail server after downloading them. In this case, Bob can reread messages from different machines; he can access a message from work and access it again later in the week from home.

2.IMAP:

With POP3 access, once Bob has downloaded his messages to the local machine, he can create mail folders and move the downloaded messages into the folders. Bob can then delete messages, move messages across folders, and search for messages (by sender name or subject). But this paradigm—namely, folders and messages in the local machine—poses a problem for the nomadic user, who would prefer to maintain a folder hierarchy on a remote server that can be accessed from any computer. This is not possible with POP3—the POP3 protocol does not provide any means for a user to create remote folders and assign messages to folders.

To solve this and other problems, the IMAP protocol, Like POP3, IMAP is a mail access protocol. It has many more features than POP3, but it is also significantly more complex. An **IMAP** server will associate each message with a folder; when a message first arrives at the server, it is associated with the recipient's INBOX folder.

The recipient can then move the message into a new, user-created folder, read the message, delete the message, and so on. The IMAP protocol provides commands to allow users to create folders and move messages from one folder to another. IMAP also provides commands that allow users to search remote folders for messages matching specific criteria. Note that, unlike POP3, an IMAP server maintains user state information across IMAP sessions—for example, the names of the folders and which messages are associated with which folders.

Another important feature of IMAP is that it has commands that permit a user agent to obtain components of messages. For example, a user agent can obtain just the message header of a message or just one part of a multipart MIME message.
This feature is useful when there is a low-bandwidth connection (for example, a

slow-speed modem link) between the user agent and its mail server. With a low bandwidth connection, the user may not want to download all of the messages in its mailbox, particularly avoiding long messages that might contain, for example, an audio or video clip.

3.Web-Based E-Mail

More and more users today are sending and accessing their e-mail through their Web browsers. Hotmail introduced Web-based access in the mid 1990s. Now Web-based e-mail is also provided by Google, Yahoo!, as well as just about every major university and corporation.

With this service, the user agent is an ordinary Web browser, and the user communicates with its remote mailbox via HTTP. When a recipient, such as Bob, wants to access a message in his mailbox, the e-mail message is sent from Bob's mail server to Bob's browser using the HTTP protocol rather than the POP3 or IMAP protocol. When a sender, such as Alice, wants to send an e-mail message, the e-mail message is sent from her browser to her mail server over HTTP rather than over SMTP. Alice's mail server, however, still sends messages to, and receives messages from, other mail servers using SMTP.

DNS—The Internet's Directory Service

We human beings can be identified in many ways. For example, we can be identified by the names that appear on our birth certificates. We can be identified by oursocial security numbers. We can be identified by our driver's license numbers.

Just as humans can be identified in many ways, so too can Internet hosts. One identifier for a host is its **hostname**. Hostnames—such as cnn.com, www.yahoo.com gaia.cs.umass.edu and cis.poly.edu—are mnemonic and are therefore appreciated by humans. However, hostnames provide little, if any, information about the location within the Internet of the host.

A hostname such as www.eurecom.fr, which ends with the country code .fr, tells us that the host is probably in France, but doesn't say much more.) Furthermore, because hostnames can consist of variable length alphanumeric characters, they would be difficult to process by routers. For these reasons, hosts are also identified by so-called **IP addresses**.

An IP address consists of four bytes and has a rigid hierarchical structure. An IP address looks like 121.7.106.83, where each period separates one of the bytes expressed in decimal notation from 0 to 255. An IP address is hierarchical because as we scan the address from left to right, we obtain more and more specific information about where the host is located in the Internet (that is, within which network, in the network of networks).

Similarly, when we scan a postal address from bottom to top, we obtain more and more specific information about where the addressee is located.People prefer the more mnemonic

hostname identifier, while routers prefer fixed-length, hierarchically structured IP addresses. In order to reconcile these preferences, we need a directory service that translates hostnames to IP addresses. This is the main task of the Internet's **domain name system (DNS)**. "
The DNS is

- a distributed database implemented in a hierarchy of **DNS servers**
- an application-layer protocol that allows hosts to query the distributed database.

The DNS protocol runs over UDP and uses port 53.
DNS is commonly employed by other application-layer protocols—including HTTP, SMTP, and FTP—to translate user-supplied hostnames to IP addresses.

As an example, consider what happens when a browser (that is, an HTTP client), running on some user's host, requests the URL www.someschool.edu/index.html
In order for the user's host to be able to send an HTTP request message to the Web server **www.someschool.edu**, the user's host must first obtain the IP address of **www.someschool.edu.** This is done as follows.

1. The same user machine runs the client side of the DNS application.
2. The browser extracts the hostname, www.someschool.edu, from the URL and passes the hostname to the client side of the DNS application.
3. The DNS client sends a query containing the hostname to a DNS server.
4. The DNS client eventually receives a reply, which includes the IP address for the hostname.
5. Once the browser receives the IP address from DNS, it can initiate a TCP connection to the HTTP server process located at port 80 at that IP address.

Services Provided By DNS

1)Translation : Translating hostnames to IP addresses.

2)Host aliasing:A host with a complicated hostname can have one or more alias names. For example, a hostname such as **relay1.west-coast.enterprise.com** could have, say, two aliases such as **enterprise.com** and www.enterprise.com. In this case, the hostname **relay1.westcoast. enterprise.com** is said to be a **canonical hostname**.
Alias hostnames,when present, are typically more mnemonic than canonical hostnames. DNS can be invoked by an application to obtain the canonical hostname for a supplied alias hostname as well as the IP address of the host.

3)Mail server aliasing:
For obvious reasons, it is highly desirable that e-mail addresses be mnemonic. For example, if Bob has an account with Hotmail, Bob's e-mail address might be as simple as **bob@hotmail.com.** However, the hostname of the Hotmail mail server is more complicated and much less mnemonic than simply hotmail.com (for example, the canonical hostname might be something like **relay1.west-coast.hotmail.com** DNS can be invoked by a mail application to obtain the canonical hostname for a supplied
alias hostname as well as the IP address of the host.

4)Load distribution: DNS is also used to perform load distribution among replicated servers, such as replicated Web servers. Busy sites, such as cnn.com, are replicated over multiple servers, with each server running on a different end system and each having a different IP address. For replicated Web servers, a *set* of IP addresses is thus associated with one canonical hostname.

Overview of How DNS Works

A simple design for DNS would have one DNS server that contains all the mappings.
In this centralized design, clients simply direct all queries to the single DNS server, and the DNS server responds directly to the querying clients. Although the simplicity of this design is attractive, it is inappropriate for today's Internet, with its vast (and growing) number of hosts. The problems with a centralized design include:

- ✓ **A single point of failure:** If the DNS server crashes, so does the entire Internet!
- ✓ **Traffic volume:** A single DNS server would have to handle all DNS queries (for all the HTTP requests and e-mail messages generated from hundreds of millions ofhosts).
- ✓ **Distant centralized database.** A single DNS server cannot be "close to" all the querying clients. If we put the single DNS server in New York City, then all queries from Australia must travel to the other side of the globe, perhaps over slow and congested links. This can lead to significant delays.
- ✓ **Maintenance.** The single DNS server would have to keep records for all Internet hosts. Not only would this centralized database be huge, but it would have to be updated frequently to account for every new host.

A Distributed, Hierarchical Database:

In order to deal with the issue of scale, the DNS uses a large number of servers, organized in a hierarchical fashion and distributed around the world. No single DNS server has all of the mappings for all of the hosts in the Internet. Instead, the mappings are distributed across the DNS servers. To a first approximation, there are three classes of DNS servers **root DNS servers, top-level domain (TLD) DNS servers, and authoritative DNS servers** organized in a hierarchy as shown in Figure 2.19.

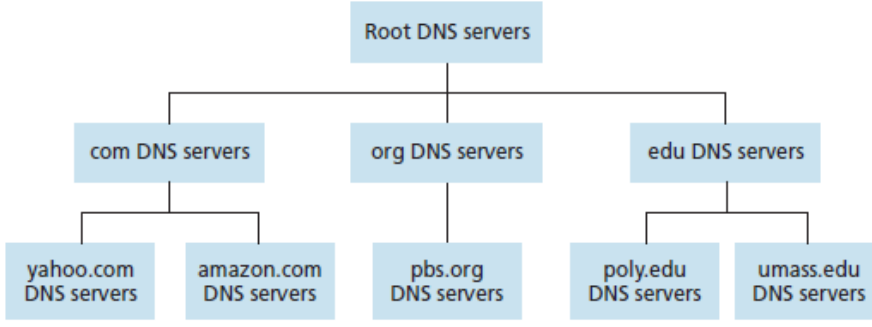


Figure 2.19 ♦ Portion of the hierarchy of DNS servers

To understand how these three classes of servers interact, suppose a DNS client wants to determine the IP address for the hostname www.amazon.com. To a first approximation, the following events will take place.

- ✓ The client first contacts one of the root servers, which returns IP addresses for TLD servers for the top-level domain com.
- ✓ The client then contacts one of these TLD servers, which returns the IP address of an authoritative server for amazon.com.
- ✓ Finally, the client contacts one of the authoritative servers for amazon.com, which returns the IP address for the hostname www.amazon.com.

- **Root DNS servers:** In the Internet there are 13 root DNS servers (labeled A through M), most of which are located in North America. An October 2006 map of the root DNS servers is shown in Figure 2.20; a list of the current root DNS servers is available via [Root-servers 2012]. Although we have referred to each of the 13 root DNS servers as if it were a single server, each "server" is actually a network of replicated servers, for both security and reliability purposes. All together, there are 247 root servers as of fall 2011.
- **Top-level domain (TLD) servers:** These servers are responsible for top-level domains such as **com, org, net, edu and gov** and all of the country top-level domains such as **uk, fr, ca, and jp.**

- **Authoritative DNS servers.** Every organization with publicly accessible hosts (such as Web servers and mail servers) on the Internet must provide publicly accessible DNS records that map the names of those hosts to IP addresses. An organization's authoritative DNS server houses these DNS records.

  A host's local DNS server is typically "close to" the host. For an institutional ISP, the local DNS server may be on the same LAN as the host; for a residential ISP, it is typically separated from the host by no more than a few routers. When a host makes a DNS query, the query is sent to the local DNS server, which acts a proxy

  2 types of queries

  (1) Iterative queries

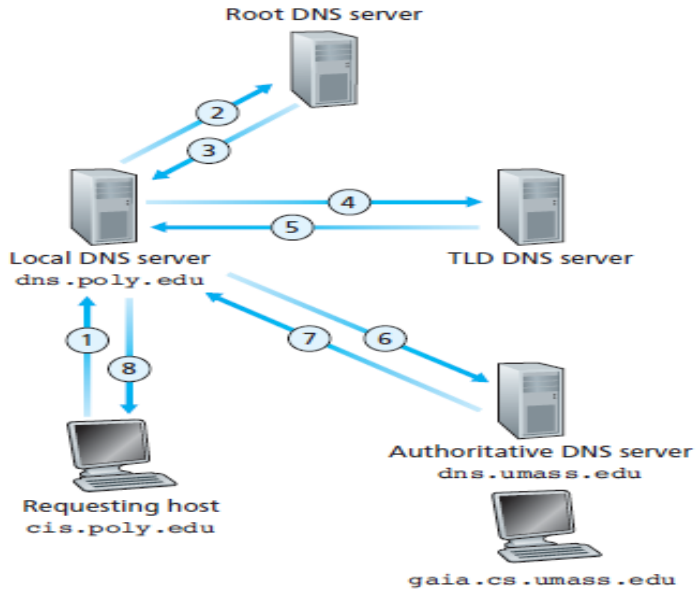
  (2) recursive queries

➢ **Iterative queries:**

An iterative name query is one in which a DNS client allows the DNS server to return the best answer it can give based on its cache or zone data.

Iterative DNS queries are once in which a DNS server is queried and returns an answer without querying other DNS servers ,even if it cannot provide a definitive answer.

Iterative queries are also called as "non-recursive queries"



Let's take a look at a simple example. Suppose the host **cis.poly.edu** desires the IP address of **gaia.cs.umass.edu.** Also suppose that Polytechnic's local DNS server is called **dns.poly.edu** and that an authoritative DNS server for **gaia.cs.umass.edu** is called **dns.umass.edu.**

The host cis.poly.edu first sends a DNS query message to its local DNSserver **dns.poly.edu.** The query message contains the hostname to be translated namely **gaia.cs.umass.edu.**The local DNS server forwards the query message to a root DNS server. The root DNS server takes note of the **edu** suffix and returns to the local DNS server a list of IP addresses for TLD servers responsible for **edu.**
The local DNS server then resends the query message to one of these TLD servers.
The TLD server takes note of the **umass.edu** suffix and responds with the IP address of the authoritative DNS server for the University of Massachusetts, namely, **dns.umass.edu.** Finally, the local DNS server resends the query message directly to dns.umass.edu, which responds with the IP address of **gaia.cs.umass.edu.**

Our previous example assumed that the TLD server knows the authoritative DNS server for the hostname. The query sent from cis.poly.edu to dns.poly.edu is a recursive query, since the query asks dns.poly.edu to obtain the mapping on its behalf. But the subsequent three queries are iterative since all of the replies are directly returned to dns.poly.edu. In theory, any DNS query can be iterative or recursive.

Recursive queries:

The DNS clients requires that DNS server responds to the client with either the requested resources record or an error messages stating that the record or domain name doesn't exist.

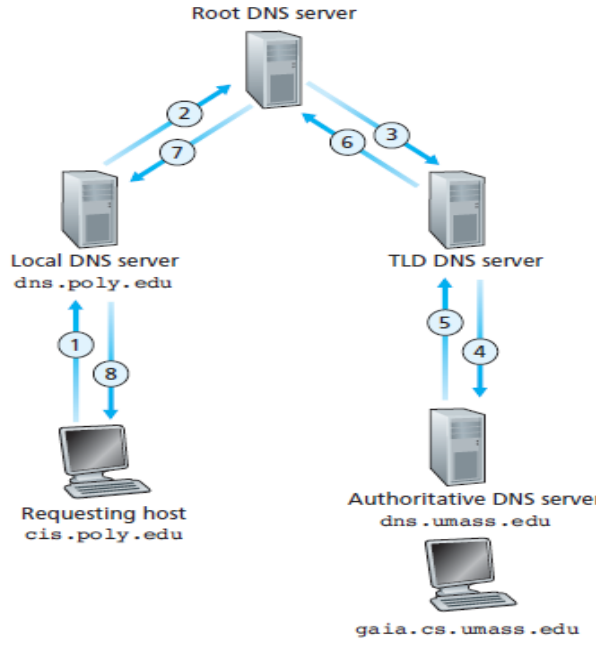


Figure 2.22 ♦ Recursive queries in DNS

DNS query chain for which all of the queries are recursive.

DNS Caching

In a query chain, when a DNS server receives a DNS reply (containing, for example, a mapping from a hostname to an IP address), it can cache the mapping in its local memory. For example, in Figure 2.21, each time the local DNS server dns.poly.edu receives a reply from some DNS server, it can cache any of the information contained in the reply. If a hostname/IP address pair is cached in a DNS server and another query arrives to the DNS server for the same hostname, the DNS server can provide the desired IP address, even if it is not authoritative for the hostname. Because hosts and mappings between hostnames and IP addresses are by no means permanent, DNS servers discard cached information after a period of time (often set to two days).

As an example, suppose that a host apricot.poly.edu queries dns.poly.edu for the IP address for the hostname cnn.com. Furthermore, suppose that a few hours later, another Polytechnic University host, say, kiwi.poly.fr, also queries dns.poly.edu with the same hostname. Because of caching, the local DNS server will be able to immediately return the IP address of cnn.com to this second requesting host without having to query any other DNS servers. A local DNS

server can also cache the IP addresses of TLD servers, thereby allowing the local DNS server to bypass the root DNS servers in a query chain .

DNS Records and Messages

The DNS servers that together implement the DNS distributed database store **resource records (RRs)**, including RRs that provide hostname-to-IP address mappings. Each DNS reply message carries one or more resource records. A resource record is a four-tuple that contains the following fields: **(Name, Value, Type, TTL)**

TTL is the time to live of the resource record; it determines when a resource should be removed from a cache. In the example records given below, we ignore the TTL field. The meaning of Name and Value depend on Type:

• **If Type=A:** then Name is a hostname and Value is the IP address for the hostname.
Thus, a Type A record provides the standard hostname-to-IP address mapping.
EX: (relay1.bar.foo.com, 145.37.93.126, A) is a Type A record.

• **If Type=NS,** then Name is a domain (such as foo.com) and Value is the hostname of an authoritative DNS server that knows how to obtain the IP addresses for hosts in the domain. This record is used to route DNS queries further along in the query chain.
Example: (foo.com, dns.foo.com, NS) is a Type NS record.

• **If Type=CNAME**: then Value is a canonical hostname for the alias hostname Name. This record can provide querying hosts the canonical name for a hostname.
Example: (foo.com, relay1.bar.foo.com, CNAME) is a CNAME record.

• **If Type=MX:**then Value is the canonical name of a mail server that has an alias hostname Name.
Example: (foo.com, mail.bar.foo.com, MX) is an MX record.
MX records allow the hostnames of mail servers to have simple aliases. To obtain the canonical name for the mail server, a DNS client would query for an MX record; to obtain the canonical name for the other server, the DNS client would query for the CNAME record.

If a DNS server is authoritative for a particular hostname, then the DNS server will contain a Type A record for the hostname. (Even if the DNS server is not authoritative, it may contain a Type A record in its cache.)
If a server is not authoritative for a hostname, then the server will contain a Type NS record for the domain that includes the hostname; it will also contain a Type A record that provides the IP address of the DNS server in the Value field of the NS record.

As an example, suppose an edu TLD server is not authoritative for the host gaia.cs.umass.edu. Then this server will contain a record for a domain that includes the host gaia.cs.umass.edu, for example, (umass.edu, dns.umass.edu, NS). The edu TLD server would also contain a Type

Arecord, which maps the DNS server dns.umass.edu to an IP address, for example, (dns.umass.edu, 128.119.40.111, A).

DNS Messages

These are the only two kinds of DNS messages. Furthermore, both query and reply messages have the same format, as shown in Figure 2.23.

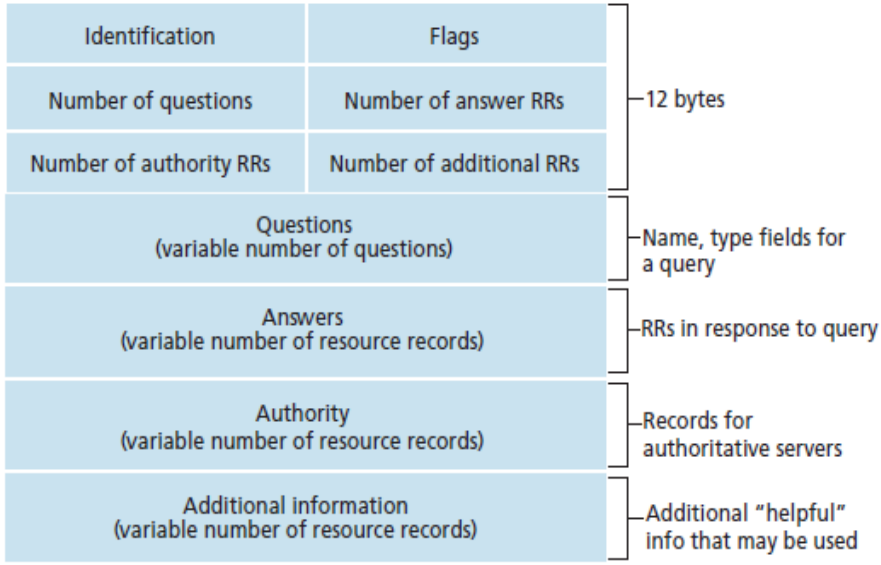


Figure 2.23 ♦ DNS message format

The semantics of the various fields in a DNS message are as follows:

- The first 12 bytes is the *header section*, which has a number of fields. The first field is a 16-bit number that identifies the query. This identifier is copied into the reply message to a query, allowing the client to match received replies with sent queries.

- There are a number of flags in the flag field.
  - ✓ A 1-bit query/reply flag indicates whether the message is a query (0) or a reply (1). A1-bit authoritative flag is set in a reply message when a DNS server is an authoritative server for a queried name.
  - ✓ A 1-bit recursion-desired flag is set when a client (host or DNS server) desires that the DNS server perform recursion when it doesn't have the record.
  - ✓ A 1-bit recursion available field is set in a reply if the DNS server supports recursion.
- The **question section** contains information about the query that is being made. This section includes
  (1) a name field that contains the name that is being queried, and

(2) a type field that indicates the type of question being asked about the name—for example, a host address associated with a name (Type A) or the mail server for a name (Type MX).

- In a reply from a DNS server, the **answer section** contains the resource records for the name that was originally queried. Recall that in each resource record there is the Type (for example, A, NS, CNAME, and MX), the Value, and the TTL. A reply can return multiple RRs in the answer, since a hostname can have multiple IP addresses (for example, for replicated Web servers, as discussed earlier in this section).
- The **authority section** contains records of other authoritative servers.
- The **additional section** contains other helpful records. For example, the answer field in a reply to an MX query contains a resource record providing the canonical hostname of a mail server. The additional section contains a Type A record providing the IP address for the canonical hostname of the mail server.

Inserting Records into the DNS Database:

The first thing you'll surely want to do is register the domain name networkutopia.com at a registrar. A **registrar** is a commercial entity that verifies the uniqueness of the domain name, enters the domain name into the DNS database (as discussed below), and collects a small fee from you for its services. Prior to 1999, a single registrar, Network Solutions, had a monopoly on domain name registration for com, net, and org domains. But now there are many registrars competing for customers, and the Internet Corporation for Assigned Names and Numbers (ICANN) accredits the various registrars. A complete list of accredited registrars is available at http://www.internic.net.

When you register the domain name networkutopia.com with some registrar, you also need to provide the registrar with the names and IP addresses of your primary and secondary authoritative DNS servers. Suppose the names and IP addresses a **dns1.networkutopia.com, dns2.networkutopia.com, 212.212.212.1, and 212.212.212.2.**

For each of these two authoritative DNS servers, the registrar would then make sure that a Type NS and a Type A record are entered into the TLD com servers. Specifically, for the primary authoritative server for networkutopia.com, the registrar would insert the following two resource records into the DNS system:

(networkutopia.com, dns1.networkutopia.com, NS)
(dns1.networkutopia.com, 212.212.212.1, A)

Peer-to-Peer Applications

In P2P architecture, there is minimal (or no) reliance on always-on infrastructure servers. Instead, pairs of intermittently connected hosts, called peers, communicate directly with each other.

The peers are not owned by a service provider, but are instead desktops and laptops controlled by users. In this section we'll examine two different applications that are particularly well-suited for P2P designs. The first is file distribution, where the application distributes a file from a single source to a large number of peers. File distribution is a nice place to start our investigation of P2P, as it clearly exposes the self-scalability of P2P architectures.

P2P File Distribution

Distributing a large file from a single server to a large number of hosts (called peers). The file might be a new version of the Linux operating system, a software patch for an existing operating system or application, an MP3 music file, or an MPEG video file. In client-server file distribution, the server must send a copy of the file to each of the peers—placing an enormous burden on the server and consuming a large amount of server bandwidth.

In P2P file distribution, each peer can redistribute any portion of the file it has received to any other peers, thereby assisting the server in the distribution process. As of 2012, the most popular P2P file distribution protocol is BitTorrent. Originally developed by Bram Cohen, there are now many different independent BitTorrent clients conforming to the BitTorrent protocol, just as there are a number of Web browser clients that conform to the HTTP protocol. In this subsection, we first examine the selfscalability of P2P architectures in the context of file distribution. We then describe BitTorrent in some detail, highlighting its most important characteristics and features.

Scalability of P2P Architectures:

To compare client-server architectures with peer-to-peer architectures, and illustrate the inherent self-scalability of P2P, we now consider a simple quantitative model for distributing a file to a fixed set of peers for both architecture types.

As shown in Figure 2.24, the server and the peers are connected to the Internet with access links. Denote the upload rate of the server's access link by *us*, the upload rate of the *i*th peer's access link by **ui,** and the download rate of the *i*th peer's access link by **di.**

Also denote the size of the file to be distributed (in bits) by *F* and the number of peers that want to obtain a copy of the file by *N*.

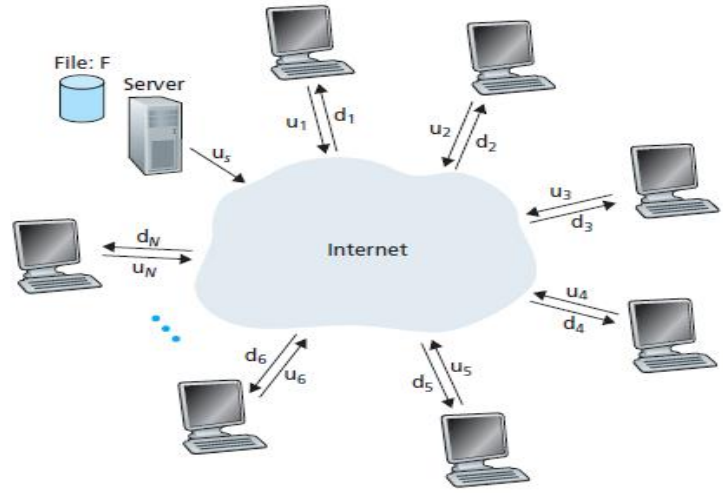


Figure 2.24 ♦ An illustrative file distribution problem

The **distribution time** is the time it takes to get a copy of the file to all *N* peers. We also suppose that the server and clients are not participating in any other network applications, so that all of their upload and download access bandwidth can be fully devoted to distributing this file. Let's first determine the distribution time for the client-server architecture, which we denote by *Dcs*. In the client-server architecture, none of the peers aids in distributing the file.

We make the following observations:

- The server must transmit one copy of the file to each of the *N* peers. Thus the server must transmit *NF* bits. Since the server's upload rate is *us*, the time to distribute the file must be at least *NF/us*.
- Let *d*min denote the download rate of the peer with the lowest download rate, thatis, *d*min = min{*d*1,*d*p,...,*dN*}. The peer with the lowest download rate cannot obtain all *F* bits of the file in less than *F/d*min seconds. Thus the minimum distribution time is at least *F/d*min. Putting these two observations together, we obtain

$d_{cs} = \max \{ NF/u_s, F/\min(d_i) \}$

This provides a lower bound on the minimum distribution time for the client-server architecture. In the homework problems you will be asked to show that the server can schedule its transmissions so that the lower bound is actually achieved. So let's take this lower bound provided above as the actual distribution time, that is,

$$d_{cs} = \max \{ NF/u_s, F/\min(d_i) \} \text{----------} (2.1)$$

We see from Equation 2.1 that for *N* large enough, the client-server distribution time is given by *NF/u*s. Thus, the distribution time increases linearly with the number of peers *N*.

So, for example, if the number of peers from one week to the next increases a thousand-fold from a thousand to a million, the time required to distribute the file to all peers increases by 1,000.

Let's now go through a similar analysis for the P2P architecture, where each peer can assist the server in distributing the file. In particular, when a peer receives some file data, it can use its own upload capacity to redistribute the data to other peers.

Calculating the distribution time for the P2P architecture is somewhat more complicated than for the client-server architecture, since the distribution time depends on how each peer distributes portions of the file to the other peers

To this end, we first make the following observations:

• At the beginning of the distribution, only the server has the file. To get this file into the community of peers, the server must send each bit of the file at least once into its access link. Thus, the minimum distribution time is at least *F/us*. (Unlike the client-server scheme, a bit sent once by the server may not have to be sent by the server again, as the peers may redistribute the bit among themselves.)

• As with the client-server architecture, the peer with the lowest download rate cannot obtain all *F* bits of the file in less than *F/d*min seconds. Thus the minimum distribution time is at least *F/d*min.

• Finally, observe that the total upload capacity of the system as a whole is equal to the upload rate of the server plus the upload rates of each of the individual peers, that is, *u*total = *us* + *u*1 + … + *uN*.

The system must deliver (upload) *F* bits to each of the *N* peers, thus delivering a total of *NF* bits. This cannot be done at a rate faster than *u*total. Thus, the minimum distribution time is also at least ***NF/(us + u1 + … + uN).***

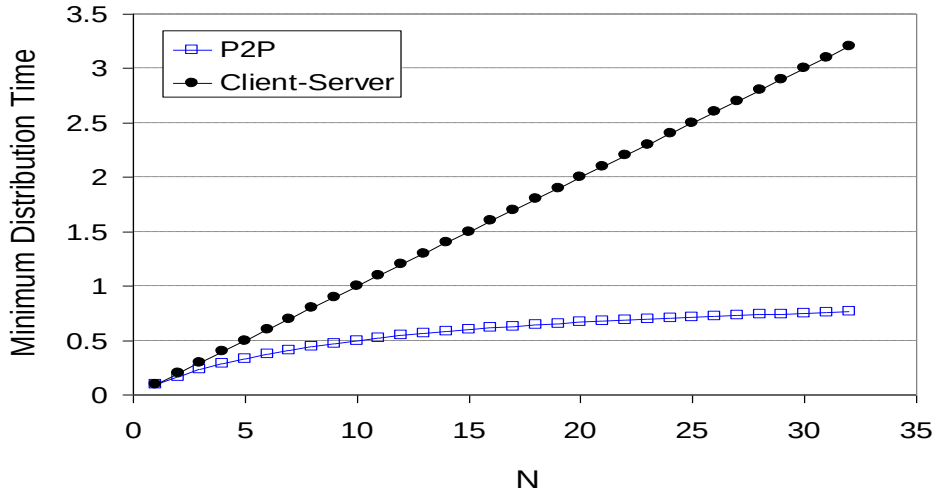
Putting these three observations together, we obtain the minimum distribution time for P2P, denoted by *D*P2P.

$$\mathbf{d_{P2P} = max\ \{\ F/u_s,\ F/min(d_i)\ ,\ NF/(u_s + Su_i)\ \}}\text{----------}(2.2)$$

Equation 2.2 provides a lower bound for the minimum distribution time for the P2P architecture.

Figure 2.25 compares the minimum distribution time for the client-server and P2P architectures assuming that all peers have the same upload rate *u*. we have set *F/u* = 1 hour, *us* = 10*u*, and *d*min ≥ *us*.

Thus, a peer can transmit the entire file in one hour, the server transmission rate is 10 times the peer upload rate, and (for simplicity) the peer download rates are set large enough so as not to have an effect. However, for the P2P architecture, the minimal distribution time is not only always less than the distribution time of the client-server architecture; it is also less than one hour for *any* number of peers *N*.



BitTorrent

BitTorrent is a popular P2P protocol for file distribution.In BitTorrent lingo, the collection of all peers participating in the distribution of a particular file is called a **torrent.**

Peers in a torrent download equal-size *chunks* of the file from one another, with a typical chunk size of 256 KBytes.

When a peer first joins a torrent, it has no chunks. Over time it accumulates more and more chunks. While it downloads chunks it also uploads chunks to other peers. Once a peer has acquired the entire file, it may (selfishly) leave the torrent, or (altruistically) remain in the torrent and continue to upload chunks to other peers.

Also, any peer may leave the torrent at any time with only a subset of chunks, and later rejoin the torrent. Let's now take a closer look at how BitTorrent operates. Each torrent has an infrastructure node called a **tracker.**

When a peer joins a torrent, it registers itself with the tracker and periodically informs the tracker that it is still in the torrent. In this manner, the tracker keeps track of the peers that are participating in the torrent. A given torrent may have fewer than ten or more than a thousand peers participating at any instant of time.

As shown in Figure 2.26, when a new peer, Alice, joins the torrent, the tracker randomly selects a subset of peers (for concreteness, say 50) from the set of participating peers, and sends the IP addresses of these 50 peers to Alice. Possessing this list of peers, Alice attempts to establish concurrent TCP connections with all the peers on this list.

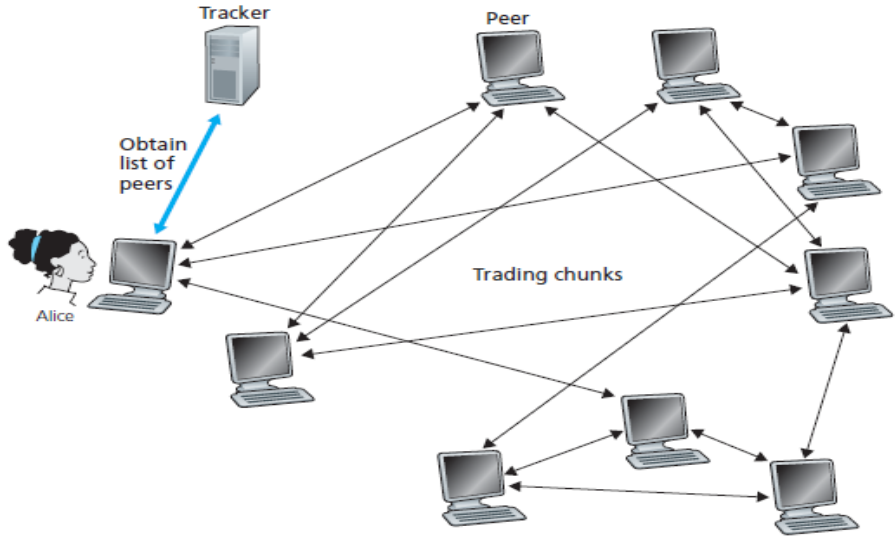


Figure 2.26 ♦ File distribution with BitTorrent

Let's call all the peers with which Alice succeeds in establishing a TCP connection "neighboring peers." (In Figure 2.26, Alice is shown to have only three neighbouring peers. Normally, she would have many more.) As time evolves, some of these peers may leave and other peers (outside the initial 50) may attempt to establish TCP connections with Alice. So a peer's neighboring peers will fluctuate over time.

At any given time, each peer will have a subset of chunks from the file, with different peers having different subsets. Periodically, Alice will ask each of her neighboring peers (over the TCP connections) for the list of the chunks they have. If Alice has *L* different neighbors, she will obtain *L* lists of chunks. With this knowledge, Alice will issue requests (again over the TCP connections) for chunks she currently does not have.

So at any given instant of time, Alice will have a subset of chunks and will know which chunks her neighbors have. With this information, Alice will have two important decisions to make.

- ✓ First, which chunks should she request first from her neighbours?
- ✓ second, to which of her neighbours should she send requested chunks?

In deciding which chunks to request, Alice uses a technique called **rarest first**. The idea is to determine, from among the chunks she does not have, the chunks that are the rarest among her neighbors (that is, the chunks that have the fewest repeated copies among her neighbors) and then request those rarest chunks first.

To determine which requests she responds to, BitTorrent uses a **clever trading algorithm.** The basic idea is that Alice gives priority to the neighbors that are currently supplying her data ***at the highest rate*.**

Specifically, for each of her neighbors, Alice continually measures the rate at which she receives bits and determines the four peers that are feeding her bits at the highest rate. She then reciprocates by sending chunks to these same four peers. Every 10 seconds, she recalculates the rates and possibly modifies the set of four peers.

In BitTorrent lingo, these four peers are said to be **unchoked**. Importantly, every 30 seconds, she also picks one additional neighbour at random and sends it chunks. Let's call the randomly chosen peer Bob. In BitTorrent lingo, Bob is said to be **optimistically unchoked**.

Because Alice is sending data to Bob, she may become one of Bob's top four uploaders, in which case Bob would start to send data to Alice. If the rate at which Bob sends data to Alice is high enough, Bob could then, in turn, become one of Alice's top four uploaders. In other words, every 30 seconds, Alice will randomly choose a new trading partner and initiate trading with that partner.
If the two peers are satisfied with the trading, they will put each other in their top four lists and continue trading with each other until one of the peers finds a better partner. The effect is that peers capable of uploading at compatible rates tend to find each other.

Distributed Hash Tables (DHTs)

In this section, we will consider how to implement a simple database in a P2P network.
Let's begin by describing a centralized version of this simple database, which will simply contain (key, value) pairs.
For example,

- ✓ the keys could be social security numbers and the values could be the corresponding human names; in this case, an example key-value pair is **(156-45-7081, Johnny Wu).**
- ✓ Or the keys could be content names (e.g., names of movies, albums, and software), and the value could be the IP address at which the content is stored; in this case, an example key-value pair is (**Led Zeppelin IV, 128.17.123.38).**

We query the database with a key. If there are one or more key-value pairs in the database that match the query key, the database returns the corresponding values.

- ➢ So, for example, if the database stores social security numbers and their corresponding human names, we can query with a specific social security number, and the database returns the name of the human who has that social security number.
- ➢ Or, if the database stores content names and their corresponding IP addresses, we can query with a specific content name, and the database returns the IP addresses that store the specific content.

In the P2P system, each peer will only hold a small subset of the totality of the (key, value) pairs. We'll allow any peer to query the distributed database with a particular key.
The distributed database will then locate the peers that have the corresponding (key, value) pairs and return the key-value pairs to the querying peer. Any peer will also be allowed to insert new key-value pairs into the database. Such a distributed database is referred to as a **distributed hash table (DHT).**

Before describing how we can create a DHT, let's first describe a specific example DHT service in the context of P2P file sharing. In this case, a key is the content name and the value is the IP address of a peer that has a copy of the content. So, if Bob and Charlie each have a copy of the latest Linux distribution, then the
DHT database will include the following two key-value pairs: **(Linux, IPBob) and (Linux, IPCharlie).**

More specifically, since the DHT database is distributed over the peers, some peer, say Dave, will be responsible for the key "Linux" and will have the corresponding key-value pairs. Now suppose Alice wants to obtain a copy of Linux. Clearly, she first needs to know which peers have a copy of Linux before she can begin to download it. To this end, she queries the DHT with "Linux" as the key.The DHT then determines that the peer Dave is responsible for the key "Linux." The DHT then contacts peer Dave, obtains from Dave the key-value pairs (Linux, IPBob) and (Linux, IPCharlie), and passes them on to Alice. Alice can then download the latest Linux distribution from either IPBob or IPCharlie.

Now let's return to the general problem of designing a DHT for general keyvalue pairs. In this design, the querying peer sends its query to all other peers, and the peers containing the (key, value) pairs that match the key can respond with their matching pairs. Such an approach is completely unscalable,of course, as it would require each peer to not only know about all other peers (possibly millions of such peers!) but even worse, have each query sent to *all* peers.
We now describe an elegant approach to designing a DHT. To this end, let's first assign an identifier to each peer, where each identifier is an integer in the range [0, 2*n*-1] for some fixed *n*. Note that each such identifier can be expressed by an *n*-bit representation. Let's also require each key to be an integer in the same range. To create integers out of such keys, we will use a hash function that maps each key (e.g., social security number) to an integer in the range [0, 2*n*_1].
Let's now consider the problem of storing the (key, value) pairs in the DHT. The central issue here is defining a rule for assigning keys to peers. Given that each peer has an integer identifier and that each key is also an integer in the same range, a natural approach is to assign each (key, value) pair to the peer whose identifier is the **closest** to the key.

To implement such a scheme, we'll need to define what is meant by "closest," for which many conventions are possible.
For convenience, let's define the closest peer as the **closest successor of the key.** To gain some insight here, let's take a look at a specific example. Suppose n=4 so that all the peer and key identifiers are in the range [0, 15].
Further suppose that there are eight peers in the system with identifiers 1, 3, 4, 5, 8, 10, 12, and 15. Finally, suppose we want to store the (key, value) pair (11, Johnny Wu) in one of the eight peers. But in which peer? Using our closest convention, since peer 12 is the closest successor for key 11,

Now suppose a peer, Alice, wants to insert a (key, value) pair into the DHT. Conceptually, this is straightforward: She first determines the peer whose identifier is closest to the key; she then sends a message to that peer, instructing it to store the (key, value) pair.

But how does Alice determine the peer that is closest to the key? If Alice were to keep track of all the peers in the system (peer IDs and corresponding IP addresses), she could locally determine the closest peer. But such an approach requires e*ach* peer to keep track of *all* other peers in the DHT—which is completely impractical for a large-scale system with millions of peers.

Circular DHT:

To address this problem of scale, let's now consider organizing the peers into a circle. In this circular arrangement, each peer only keeps track of its immediate successor and immediate predecessor (modulo 2n).
In this example, n is again 4 and there are the same eight peers from the previous example. Each peer is only aware of its immediate successor and predecessor; for example, peer 5 knows the IP address and identifier for peers 8 and 4 but does not necessarily know anything about any other peers that may be in the DHT. This circular arrangement of the peers is a special case of an **overlay network**.

In an overlay network, the peers form an abstract logical network which resides above the "underlay" computer network consisting of physical links, routers, and hosts. The links in an overlay network are not physical links, but are simply virtual liaisons between pairs of peers. In the overlay in Figure 2.27(a), there are eight peers and eight overlay links; in the overlay in Figure 2.27(b) there are eight peers and 16 overlay links.

Using the circular overlay in Figure 2.27(a), now suppose that peer 3 wants to determine which peer in the DHT is responsible for key 11. Using the circular overlay, the origin peer (peer 3) creates a message saying "Who is responsible for key 11?" and sends this message clockwise around the circle.

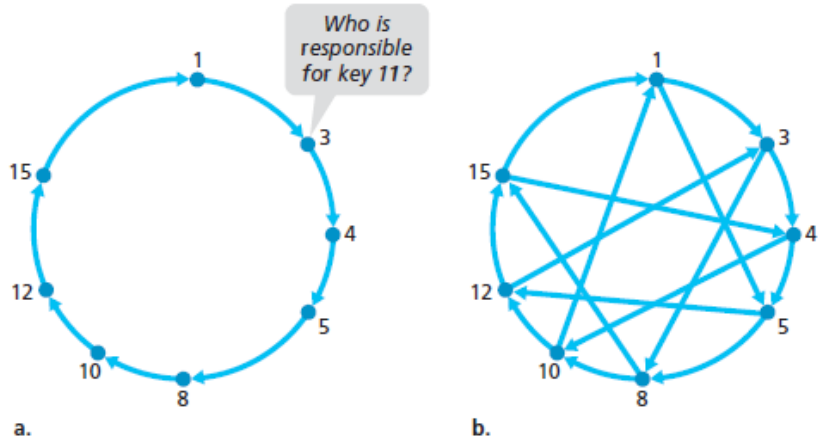


Figure 2.27 ♦ (a) A circular DHT. Peer 3 wants to determine who is responsible for key 11. (b) A circular DHT with shortcuts

Whenever a peer receives such a message, because it knows the identifier of its successor and predecessor, it can determine whether it is responsible for (that is, closest to) the key in question. If a peer is not responsible for the key, it simply sends the message to its successor.

So, for example, when peer 4 receives the message asking about key 11, it determines that it is not responsible for the key (because its successor is closer to the key), so it just passes the message along to peer 5. This process continues until the message arrives at peer 12, who determines that it is the closest peer to key 11. At this point, peer 12 can send a message back to the querying peer, peer 3, indicating that it is responsible for key 11.

The circular DHT provides a very elegant solution for reducing the amount of overlay information each peer must manage. In particular, each peer needs only to be aware of two peers, its immediate successor and its immediate predecessor. But this solution introduces yet a new problem.

Although each peer is only aware of two neighboring peers, to find the node responsible for a key (in the worst case), all N nodes in the DHT will have to forward a message around the circle; $N/2$ messages are sent on average.

Thus, in designing a DHT, there is tradeoff between the number of neighbors each peer has to track and the number of messages that the DHT needs to send to resolve a single query.

On one hand, if each peer tracks all other peers (mesh overlay), then only one message is sent per query, but each peer has to keep track of *N* peers. On the other hand, with a circular DHT, each peer is only aware of two peers, but *N*/2 messages are sent on average for each query. but add "shortcuts" so that each peer not only keeps track of its immediate successor and predecessor, but also of a relatively small number of shortcut peers scattered about the circle.

An example of such a circular DHT with some shortcuts is shown in Figure 2.27(b). Shortcuts are used to expedite the routing of query messages. Specifically, when a peer receives a message that is querying for a key, it forwards the message to the neighbor (successor neighbor or one of the shortcut neighbors) which is the closet to the key. Thus, in **Figure 2.27(b),** when peer 4 receives the message asking about key 11, it determines that the closet peer to the key (among its neighbors) is its shortcut neighbor 10 and then forwards the message directly to peer 10. Clearly,shortcuts can ignificantly reduce the number of messages used to process a query.

Peer Churn:

In P2P systems, a peer can come or go without warning. Thus, when designing a DHT, we also must be concerned about maintaining the DHT overlay in the presence of such peer churn. To get a big-picture understanding of how this could be accomplished, let's once again consider the circular DHT in Figure 2.27(a).

To handle peer churn, we will now require each peer to track (that is, know the IP address of) its first and second successors; for example, peer 4 now tracks both peer 5 and peer 8. We also require each peer to periodically verify that its two successors are alive (for example, by periodically sending ping messages to them and asking for responses).

Let's now consider how the DHT is maintained when a peer abruptly leaves. For example, suppose peer 5 in Figure 2.27(a) abruptly leaves. In this case, the two peers preceding the departed peer (4 and 3) learn that 5 has departed, since it no longer responds to ping messages. Peers 4 and 3 thus need to update their successor state information. Let's consider how peer 4 updates its state:

- Peer 4 replaces its first successor (peer 5) with its second successor (peer 8).
- Peer 4 then asks its new first successor (peer 8) for the identifier and IP address of its immediate successor (peer 10). Peer 4 then makes peer 10 its second successor.

Having briefly addressed what has to be done when a peer leaves, let's now consider what happens when a peer wants to join the DHT. Let's say a peer with identifier 13 wants to join the DHT, and at the time of joining, it only knows about peer 1's existence in the DHT. Peer 13 would first send peer 1 a message, saying "what will be 13's predecessor and successor?"

This message gets forwarded through the DHT until it reaches peer 12, who realizes that it will be 13's predecessor and that its current successor, peer 15, will become 13's successor. Next, peer 12 sends this predecessor and successor information to peer 13. Peer 13 can now join the DHT by making peer 15 its successor and by notifying peer 12 that it should change its immediate successor to 13.

Socket Programming: Creating Network Applications

Now that we've looked at a number of important network applications, let's explore how network application programs are actually created. that a typical network application consists of a pair of programs a client program and a server program residing in two different end systems.

When these two programs are executed, a client process and a server process are created, and these processes communicate with each other by reading from, and writing to, sockets. When creating a network application, the developer's main task is therefore to write the code for both the client and server programs. There are two types of network applications.

Indeed, many of today's network applications involve communication between client and server programs that have been created by independent developers—for example, a Firefox browser communicating with an Apache Web server, or a BitTorrent client communicating with BitTorrent tracker.

Socket Programming with UDP

That processes running on different machines communicate with each other by sending messages into sockets. We said that each process is analogous to a house and the process's socket is analogous to a door. The application resides on one side of in the door in the house; the transport-layer protocol resides on the other side of the door in the outside world.

The application developer has control of everything on the application-layer side of the socket; however, it has little control of the transport-layer side. Now let's take a closer look at the interaction between two communicating processes that use UDP sockets. Before the sending process can push a packet of data out the socket door, when using UDP, it must first attach a destination address to the packet.

After the packet passes through the sender's socket, the Internet will use this destination address to route the packet through the Internet to the socket in the receiving process. When the packet arrives at the receiving socket, the receiving process will retrieve the packet through the socket, and then inspect the packet's contents and take appropriate action.

- By including the destination IP address in the packet, the routers in the Internet will be able to route the packet through the Internet to the destination host.
- But because a host may be running many network application processes, each with one or more sockets, it is also necessary to identify the particular socket in the destination host. When a socket is created, an identifier, called a **port number**, is assigned to it.
- The sender's source address— consisting of the IP address of the source host and the port number of the source socket—are also attached to the packet. However, attaching the source address to the packet is typically *not* done by the UDP application code. instead it is automatically done by the underlying operating system

Figure 2.28 highlights the main socket-related activity of the client and server that communicate over the UDP transport service.

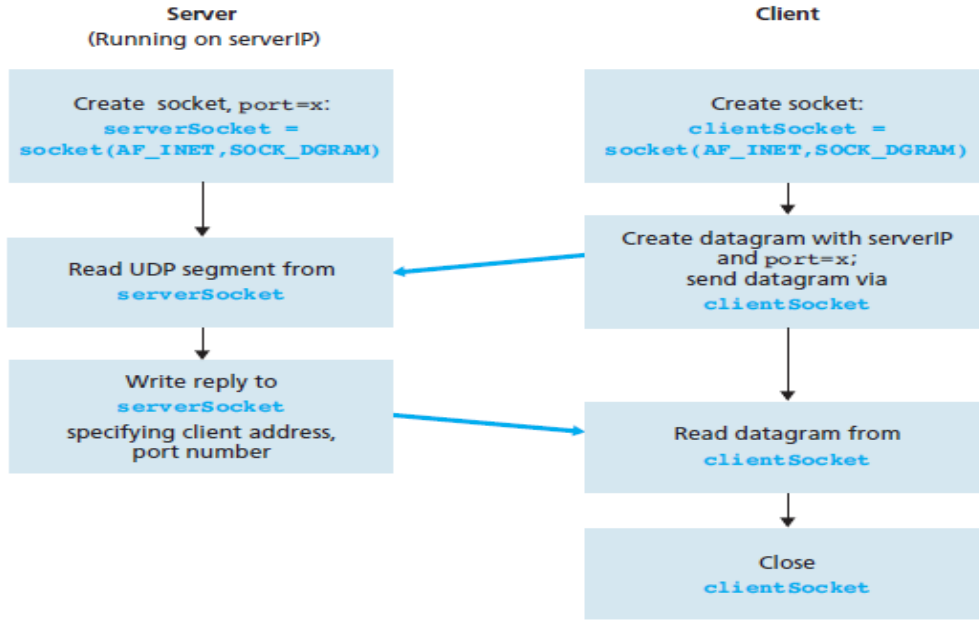


Figure 2.28 ♦ The client-server application using UDP

1. The client reads a line of characters (data) from its keyboard and sends the data to the server.
2. The server receives the data and converts the characters to uppercase.
3. The server sends the modified data to the client.
4. The client receives the modified data and displays the line on its screen.

UDPClient.py
Here is the code for the client side of the application:

```
from socket import *
serverName = 'hostname'
serverPort = 12000
clientSocket = socket(socket.AF_INET, socket.SOCK_DGRAM)
message = raw_input('Input lowercase sentence:')
clientSocket.sendto(message,(serverName, serverPort))
modifiedMessage, serverAddress = clientSocket.recvfrom(2048)
print modifiedMessage
clientSocket.close()
```

Now let's take a look at the various lines of code in UDPClient.py. from socket import *

The socket module forms the basis of all network communications in Python. By including this line, we will be able to create sockets within our program.


```
serverName = 'hostname'
serverPort = 12000
```

The first line sets the string server Name to hostname. Here, we provide a string containing either the IP address of the server (e.g., **"128.138.32.126"**) or the host-name of the server (e.g., "cis.poly.edu").

If we use the hostname, then a DNS lookup will automatically be performed to get the IP address.) The second line sets the inte-ger variable serverPort to 12000.

clientSocket = socket(socket.AF_INET, socket.SOCK_DGRAM)

This line creates the client's socket, called clientSocket. The first parameter indicates the address family; in particular, AF_INET indicates that the underlying network is using IPv4. (Do not worry about this now—we will discuss IPv4 in Chapter 4.) The second parameter indicates that the socket is of type SOCK_DGRAM, which means it is a UDP socket (rather than a TCP socket).

message = raw_input("Input lowercase sentence:")

raw_input() is a built-in function in Python. When this command is executed, the user at the client is prompted with the words "Input data:" The user then uses her keyboard to input a line, which is put into the variable message.

Now that we have a socket and a message, we will want to send the message through the socket to the destination host.

clientSocket.sendto(message,(serverName, serverPort))

In the above line, the method sendto() attaches the destination address (serverName, serverPort) to the message and sends the resulting packet into the process's socket, clientSocket. (As mentioned earlier, the source address is also attached to the packet, although this is done automatically rather than explicitly by the code.) Sending a client-to-server message via a UDP socket is that simple! After sending the packet, the client waits to receive data from the server.

modifiedMessage, serverAddress = clientSocket.recvfrom(2048)

With the above line, when a packet arrives from the Internet at the client's socket, the packet's data is put into the variable modifiedMessage and the packet's source address is put into the variable serverAddress. The variable serverAddress contains both the server's IP address and the server's port number. The method recvfrom also takes the buffer size 2048 as input.

print modifiedMessage: This line prints out modifiedMessage on the user's display. It should be the original line that the user typed, but now capitalized.

clientSocket.close(): This line closes the socket. The process then terminates.

UDPServer.py

Let's now take a look at the server side of the application:

from socket import *

serverPort = 12000

serverSocket = socket(AF_INET, SOCK_DGRAM)

serverSocket.bind((', serverPort)) print ***"The server is ready to receive"*** **while 1:**

message, clientAddress = serverSocket.recvfrom(2048)

modifiedMessage = message.upper()

serverSocket.sendto(modifiedMessage, clientAddress)

The first line of code that is significantly different from UDPClient is:

serverSocket.bind((" ", serverPort)): It binds (that is, assigns) the port number 12000 to the server's socket. Thus in UDPServer, the code (written by the application developer) is explicitly. assigning a port number to the socket. In this manner, when anyone sends a packet to port 12000 at the IP address of the server, UDPServer waits for a packet to arrive.

message, clientAddress = serverSocket.recvfrom(2048):

When a packet arrives at the server's socket, the packet's data is put into the variable message and the packet's source address is put into the variable clientAddress.

The variable clientAddress contains both the client's IP address and the client's port number. Here, UDPServer *will* make use of this address information, as it provides a return address, similar to the return address with ordinary postal mail. With this source address information, the server now knows to where it should direct its reply.

modifiedMessage = message.upper()

This line is the heart of our simple application. It takes the line sent by the client and uses the method upper() to capitalize it.

serverSocket.sendto(modifiedMessage, clientAddress)

This last line attaches the client's address (IP address and port number) to the capitalized message, and sends the resulting packet into the server's socket.

The Internet will then deliver the packet to this client address. After the server sends the packet, it remains in the while loop, waiting for another UDP packet to arrive (from any client running on any host).

Socket Programming with TCP

- Unlike UDP, TCP is a connection-oriented protocol. This means that before the client and server can start to send data to each other, they first need to handshake and establish a TCP connection.

- One end of the TCP connection is attached to the client socket and the other end is attached to a server socket. When creating the TCP connection, we associate with it the client socket address (IP address and port number) and the server socket address (IP address and port number). With the TCP connection established, when one side wants to send data to the other side, it just drops the data into the TCP connection via its socket.

This is different from UDP, for which the server must attach a destination address to the packet before dropping it into the socket.

- First, as in the case of UDP, the TCP server must be run-ning as a process before the client attempts to initiate contact. Second, the server program must have a special door—more precisely, a special socket—that welcomes some initial contact from a client process running on an arbitrary host. Using our house/door analogy for a process/socket, we will sometimes refer to the client's initial contact as "knocking on the welcoming door."

- With the server process running, the client process can initiate a TCP connection to the server. This is done in the client program by creating a TCP socket. When the client creates its TCP socket, it specifies the address of the welcoming socket in the server, namely, the IP address of the server host and the port number of the socket.

- After creating its socket, the client initiates a three-way handshake and establishes a TCP connection with the server. The three-way handshake, which takes place within the transport layer, is completely invisible to the client and server pro-grams.

- During the three-way handshake, the client process knocks on the welcoming door of the server process. When the server "hears" the knocking, it creates a new door— more precisely, a *new* socket that is dedicated to that particular client.

- The newly created socket dedicated to the client making the connection is called connec-tionSocket.

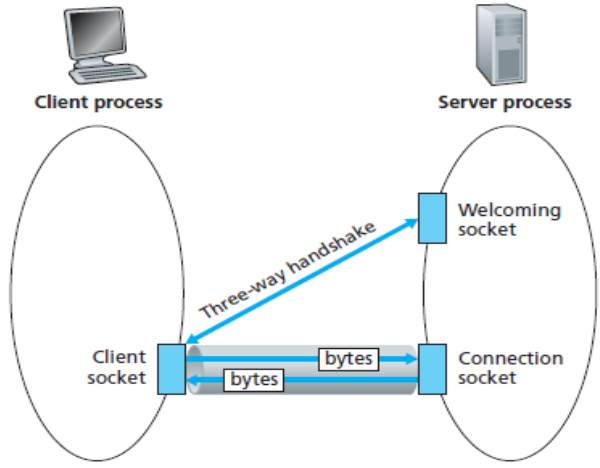


Figure 2.29 ♦ The `TCPServer` process has two sockets

We use the same simple client-server application to demonstrate socket programming with TCP: The client sends one line of data to the server, the server capitalizes the line and sends it back to the client. Figure 2.30 highlights the main socket-related activity of the client and server that communicate over the TCP transport service.

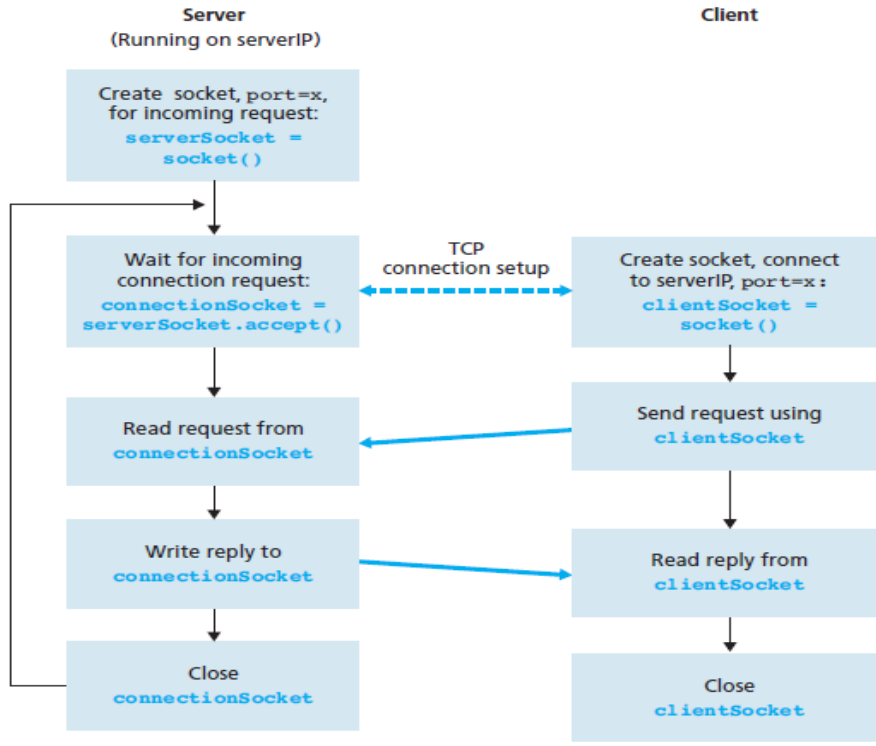


Figure 2.30 ♦ The client-server application using TCP

TCPClient.py

Here is the code for the client side of the application:

```
from socket import *
serverName = 'servername'
serverPort = 12000
clientSocket = socket(AF_INET, SOCK_STREAM)
```

clientSocket.connect((serverName,serverPort))

sentence = raw_input('Input lowercase sentence:')

clientSocket.send(sentence) modifiedSentence = clientSocket.recv(1024)
print 'From Server:', modifiedSentence clientSocket.close()

Let's now take a look at the various lines in the code that differ significantly from the UDP implementation. The first such line is the creation of the client socket.

```
clientSocket = socket(AF_INET, SOCK_STREAM)
```

This line creates the client's socket, called `clientSocket`. The first parameter again indicates that the underlying network is using IPv4. The second parameter indicates that the socket is of type `SOCK_STREAM`, which means it is a TCP socket (rather than a UDP socket.

`clientSocket.connect((serverName,serverPort))`: client can send data to the server (or vice versa) using a TCP socket, a TCP connection must first be established between the client and server. The above line initiates the TCP connection between the client and server. The parameter of the `connect ()` method is the address of the server side of the con-nection. After this line of code is executed, the three-way handshake is performed and a TCP connection is established between the client and server.

sentence = raw_input('Input lowercase sentence:'): The string `sentence` continues to gather characters until the user ends the line by typing a carriage return. The next line of code is also very different from UDPClient:

`clientSocket.send(sentence)`: It sends the string `sentence` through the client's socket and into the TCP connection. Note that the program does *not* explicitly create a packet and attach the destination address to the packet, as was the case with UDP sockets. Instead the client program simply drops the bytes in the string `sentence` into the TCP con-nection. The client then waits to receive bytes from the server.

```
modifiedSentence = clientSocket.recv(2048)
```

When characters arrive from the server, they get placed into the string `modified-Sentence`. Characters continue to accumulate in **`modifiedSentence`** until the line ends with a carriage return character. After printing the capitalized sentence, we close the client's socket:

```
clientSocket.close()
```

This last line closes the socket and, hence, closes the TCP connection between the client and the server.

TCPServer.py

Now let's take a look at the server program.

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind((‘’,serverPort))
serverSocket.listen(1)
print ‘The server is ready to receive’ while 1:
connectionSocket, addr = serverSocket.accept()
sentence = connectionSocket.recv(1024)
capitalizedSentence = sentence.upper()
connectionSocket.send(capitalizedSentence)
connectionSocket.close()
```


Let's now take a look at the lines that differ significantly from UDPServer and TCP Client. As with TCPClient, the server creates a TCP socket with:

serverSocket=socket(AF_INET,SOCK_STREAM)

Similar to UDPServer, we associate the server port number, serverPort, with this socket:

serverSocket.bind((‘’,serverPort))

But with TCP, serverSocket will be our welcoming socket. After establish-ing this welcoming door, we will wait and listen for some client to knock on the door:

serverSocket.listen(1): The server listen for TCP connection requests from the client. The parameter specifies the maximum number of queued connections (at least 1).

connectionSocket, addr = serverSocket.accept(): When a client knocks on this door, the program invokes the accept() method for serverSocket, which creates a new socket in the server, called connec tionSocket, dedicated to this particular client. The client and server then complete the handshaking, creating a TCP connection between the client's clientSocket and the server's connectionSocket.

With the TCP connection established, the client and server can now send bytes to each other over the connection. With TCP, all bytes sent from one side not are not only guaranteed to arrive at the other side but also guaranteed arrive in order.

connectionSocket.close(): In this program, after sending the modified sentence to the client, we close the connection socket. But since serverSocket remains open, another client can now knock on the door and send the server a sentence to modify.