Chapter 2: Application Layer

2.1 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. Importantly, you do not need to write software that runs on network-core devices, such as routers or link-layer switches. Even if you wanted to write application software for these network-core devices, you wouldn’t be able to do so. Network-core devices do not function at the application layer but instead function at lower layers - specifically at the network layer and below. 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.

2.1.1 Network Application Architectures

The application's architectures is distinctly **different** from the network architecture. From the application developer’s perspective, the network architecture is fixed and provides a specific set of services to applications. 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.

Two predominant architectureal paradigms used in modern network applications: the client-server architecture or the peer-to-peer (P2P) architecture.

In a **client-server architecture**, there is an always-on host, called the *server*, which services requests from many other hosts, called *clients*. Note that with the client-server architecture, **clients do not directly communicate with each other** (e.g. 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** (which we’ll discuss soon). Examples of client-server architecture: Web, FTP, Telnet and email.

Often in a client-server application, a single-server host is incapable of keeping up with all the requests from clients. For this reason, a **data center**, housing a large number of hosts, is often used to create a powerful virtual server.

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*. 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 (e.g. file sharing (e.g., BitTorrent), peer-assisted download acceleration (e.g., Xunlei), Internet Telephony (e.g., Skype), and IPTV (e.g.,Kankan and PPstream).

We mention that some applications have **hybrid architectures**, combining both client-server and P2P elements. For example, for many instant messaging applications, servers are used to track the IP addresses of users, but user-to-user messages are sent directly between user hosts (without passing through intermediate servers).

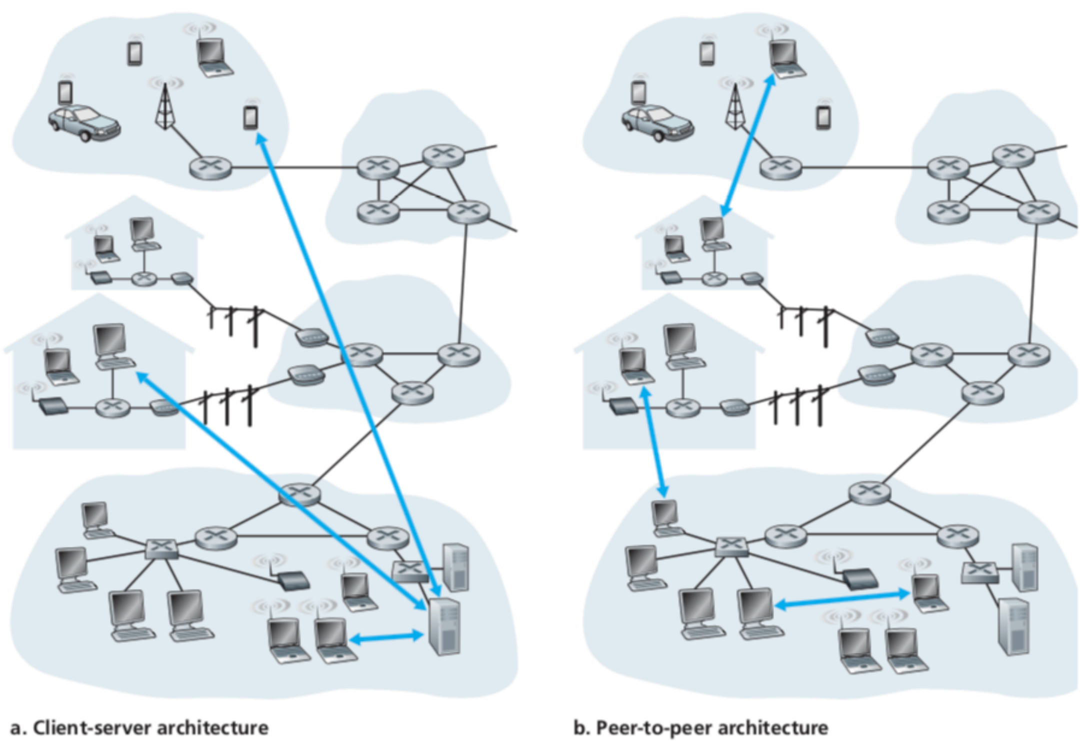


Figure 2.2 ♦ (a) Client-server architecture; (b) P2P architecture

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. P2P architectures are also **cost effective**, since they normally don't require significant server infrastructure and server bandwidth (in contrast with clients-server designs with datacenters).

The challenges P2P application will face in the nfuture:

- **ISP Friendly**. Most residential ISPs (including DSL and cable ISPs) have been dimensioned for "asymmetrical" bandwidth usage, that is, for much more downstream than upstream traffic. But P2P video streaming and file distribution applications shift upstream traffic from servers to residential ISPs, thereby putting significant stress on the ISPs.
- **Security**. Because of their highly distributed and open nature, P2P applications can be a challenge to secure.
- **Incentives**. The success of future P2P applications also depends on convincing users to volunteer bandwidth, storage, and computation resources to the applications, which is the

challenge of incentive design.

2.1.2 Processes Communicating

Processes helps programs running in multiple end systems and communicating with each other. 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 com municate with each other with interprocess communication. 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 Process

A network application consists of pairs of processes that send messages to each other over a network. 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 P2P file sharing, the peer that is downloading the file is labeled as the client, and the peer that is uploading the file is labeled as the server.

In some applications, such as in P2P file sharing, a process can be both a client and a server. 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 context of a communication session between a pair of processes, the process that initiates the communication (that is, initially contacts the other process at the beginning of the session) is labeled as the **client**. The process that waits to be contacted to begin the session is the **server**.*

The Interface Between the Process and the Computer Network

A process sends messages into, and receives messages from, the network through a software interface called a **socket**.

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.

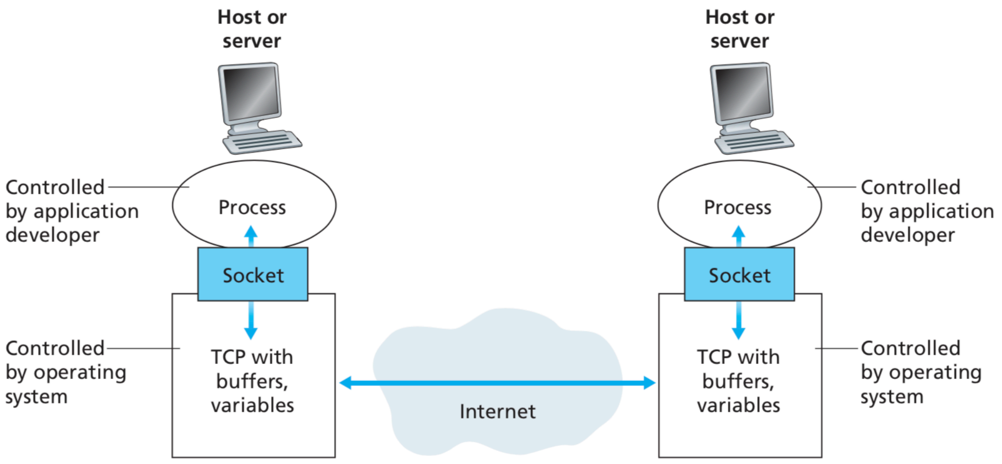

Host or server
Host or server
Controlled by application developer
Process
Socket
Controlled by operating system
TCP with buffers, variables
Internet
Process
Socket
TCP with buffers, variables
Controlled by application developer
Controlled by operating system

Figure 2.3 ♦ Application processes, sockets, and underlying transport protocol

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:

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

Addressing Processes

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:

1. The address of the host. In the Internet, the host is identified by its **IP address**, a 32-bit quantity that we can think of as uniquely identifies the host.
2. An identifier that specifies the receiving process in the destination host: the destination **port number**. Popular applications have been assigned specific port number (web server -> 80, mail server process -> 25).

2.1.3 Transport Services Available to Applications

What are the services that a transport-layer protocol can offer to applications invoking it?

Reliable Data Transfer

For some applications, such as email, file transfer, remote host access, Web document transfer, data loss can have devastating consequences. If a protocal provides a guarantee that the data sent by one end of the application is delivered correctly and completely to the other side, it is said to provide **reliable data transfer**.

One important service that a transport-layer protocol can potentially provide to an application is process-to-process 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.

Throughput

The throughput is the rate at which the sending process can deliver bits to the receiving process. The available throughput can fluctuate due to the flucutation of the shared bandwidth. These observations lead to another natural service that a transport-layer protocol could provide, namely, guaranteed available throughput at some specified rate. Applications that have throughput requirements are said to be **bandwidth-sensitive applications** (e.g. many current multimedia applications).

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 (e.g. email, file transfer, Web transfer).

Timing

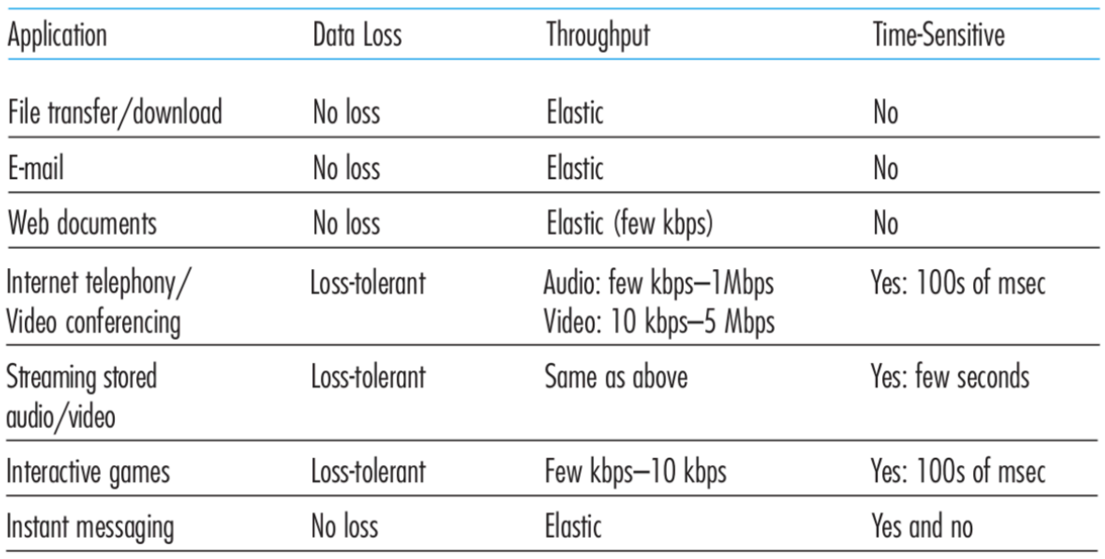
A transport-layer protocol can also provide timing guarantees. As with throughput guarantees, timing guarantees can come in many shapes and forms. For example, guarantee that every bit that the sender pumps into the socket arrives at the receiver's socket no more than 100 msec later -> be favored by Internet telephony, virtual environments, teleconferencing and multiplayer games.

Security

A transport protocol can provide an application with one or more security services. For exmaple, it could encrypt the data when it is send and decrypt it when it is received. Or it can also provide other security services in addition to confidentiality, including data integrity and end-point authentication.

2.1.4 Transport Services Provided by the Internet

The Internet provides two transport protocols available to applications: UDP and TCP. Before we create a new network application, the first thing we should do is to decide whether to use UDP or TCP. The following figure shows the service requirements for some selected applications:



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

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**. 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.

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.

SECURING TCP

Neither TCP nor UDP provide any encryption—the data that the sending process passes into its socket is the same data that travels over the network to the destination process. Thus, the Internet community has developed an enhancement for TCP, called **Secure Sockets Layer (SSL)**, which not only does everything that traditional TCP does but also provides critical process-to-process security service such as encryption, data integrity and end-point authentication. It is not a third Internet transport protocol, but instead is an enhancement of TCP, the enhancements being implemented in the application layer in both client and server side.

SSL has its own socket API, similar to the traditional one. If we send processes passing cleartext data to the SSL, it will encrypt it and when the TCP socket receives it, it will be passes to SSL for decrypt.

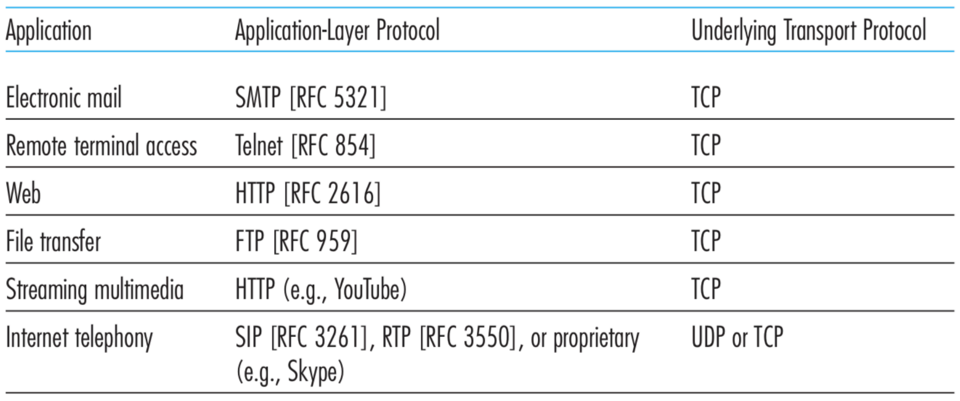
UDP Services

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**. UDP provides *no* guarantee that the message will ever reach the receiving process, and messages may also arrive out of order.
- UDP does not include a congestion-control mechanism. The sending side of UDP can pump data into the layer below (the network layer) at any rate it pleases.

Services Not Provided by Internet Transport Protocals

TCP and UDP doesn't provide throughput or timing guarantees, services not provided by today's Internet transport protocols. We therefore design applications to cope, to the greatest extent possible, with this lack of guarantee.



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

2.1.5 Application-Layer Protocols

An **application-layer protocol** defines how an application's processes, running on different end systems, pass messages to each other. In particular, an application-layer protocol defines:

- The types of messages exchanged, for example, request messages and response messages.
- The syntax of the various message types, such as the fields in the message and how the fields are delineated.
- The semantics of the fields, that is, the meaning of the information in the fields.
- Rules for determining when and how a process sends messages and responds to messages.

Network application vs. application-layer protocals: An application-layer protocol is only one piece of a network application.

2.2 The Web and HTTP

Internet was primarily used by researcheres, academics and university students. A major application, the World Wide Web, was the first Internet application that caught the general public's eyes. It dramatically changed, and continues to change, how people interact inside and outside their work environments.

What appeals to user is that the Web operates *on demand* : users receive what they want when they want it. It doesn't like television or broadcast radio that force users to listen what they don't want.

It is enormously easy for an individual to make information available over the web. Hyperlinks and search engines help us navigate through the ocean of web sites.

2.2.1 Overview of HTTP

The **HyperText Transfer Protocol (HTTP)**, the Web's application-layer protocol, 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. **HTTP defines the structure of these messages and how the client and server exchange the messages.**

A **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. The base HTML file references the other objects in the page with the objects' URLs.

Each **URL** has two components: **the hostname of the server** that houses the object and **the object's path name**. For example, the URL:

```
http://www.someSchool.edu/someDepartment/picture.gif
```

has www.someSchool.edu for a hostname and /someDepartment/picture.gif for a path name.

Web browsers (such as Internet Explorer and Firefox) implement the client side of HTTP.

Web servers, which implement the server side of HTTP, house Web objects, each addressable by a URL.

HTTP defines how Web clients request Web pages from Web servers and how servers transfer Web pages to clients. Below illustrates the basic idea of the interaction between server and client.

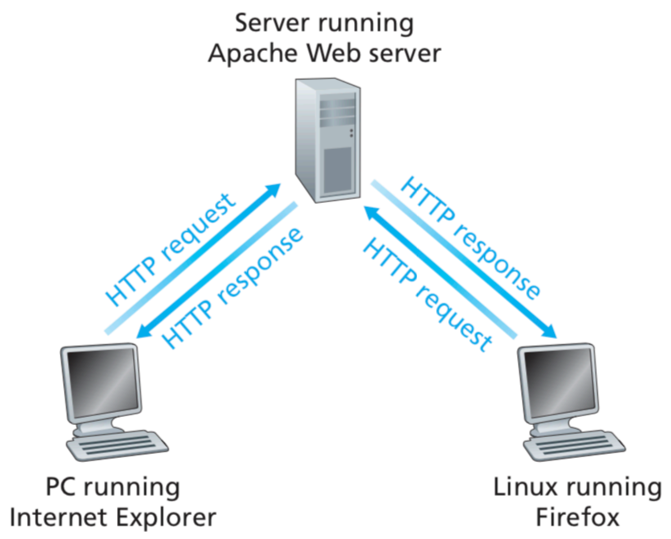


Figure 2.6 ♦ HTTP request-response behavior

HTTP uses TCP as its underlying transport protocol. 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. As described in Section 2.1, 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.

HTTP need not worry about lost data (since it uses TCP as its underlying protocol) or the details of how TCP recovers from loss or reordering of data within the network.

HTTP is said to be a **stateless protocol** because an HTTP server maintains no information about the clients. (e.g. if client A asks an object from server twice, the server doesn't respond that it just served the object to the client before. It is completely forgotten what it did and will resend the object to A).

Web uses the **client-server application architecture**, which has a fixed IP address, and it services requests from potentially millions of different browsers.

2.2.2 Non-Persistent and Persistent Connections

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, or should all of the requests and their cor- responding responses be sent over the *same* TCP connection?

Non-persistent connections: each request/response pair be sent over a separate TCP connection.

Persistent connections: each request/response pair be sent over the same TCP connection.

By default HTTP uses persistent connections, but HTTP server and clients can be configured to use non-persistent connection instead.

HTTP with Non-Persistent Connections

Let's walk through an example 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
```

What happens next is:

1. The HTTP client process initiates a TCP connection to the server www.someSchool.edu on port number 80, which is the default port num- ber 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 /someDepartment/home.index.
3. The HTTP server process receives the request message via its socket, retrieves the object /someDepartment/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.
6. The first four steps are then repeated for each of the referenced JPEG objects.

The HTTP specifications ([RFC 1945] and [RFC 2616]) define only the communication protocol between the client HTTP program and the server HTTP program, and it doesn't care how web browser interpret a Web page, i.e. Safari and Chrome may interpret the receiving Web page in different manner.

In this process, **each TCP connection is closed after the server sends the object**—the connection does not persist for other objects. Note that each TCP connection transports exactly one request message and one response message. Thus, in this example, when a user requests the Web page, 11 TCP connections are generated.

In the steps above, we actually vague about whether the client obtain 10 JPEGs over 10 series TCP connections, or whether some of the JPEGs are obtained over parallel TCP connections. Indeed, users can configure modern browsers to control the degree of parallelism. In their default modes, most browsers open 5 to 10 parallel TCP connections, and each of these connections handles one request-response transaction.

Round-trip time (RTT) is the time it takes for a small packet to travel from client to server and then back to the client, which includes packet-propagation delays, packet-queuing delays in intermediate routers and switches, and packet-processing delays.

Now consider what happens when a user clicks on a hyperlink. As shown in Figure 2.7, 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 handshake, 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.**

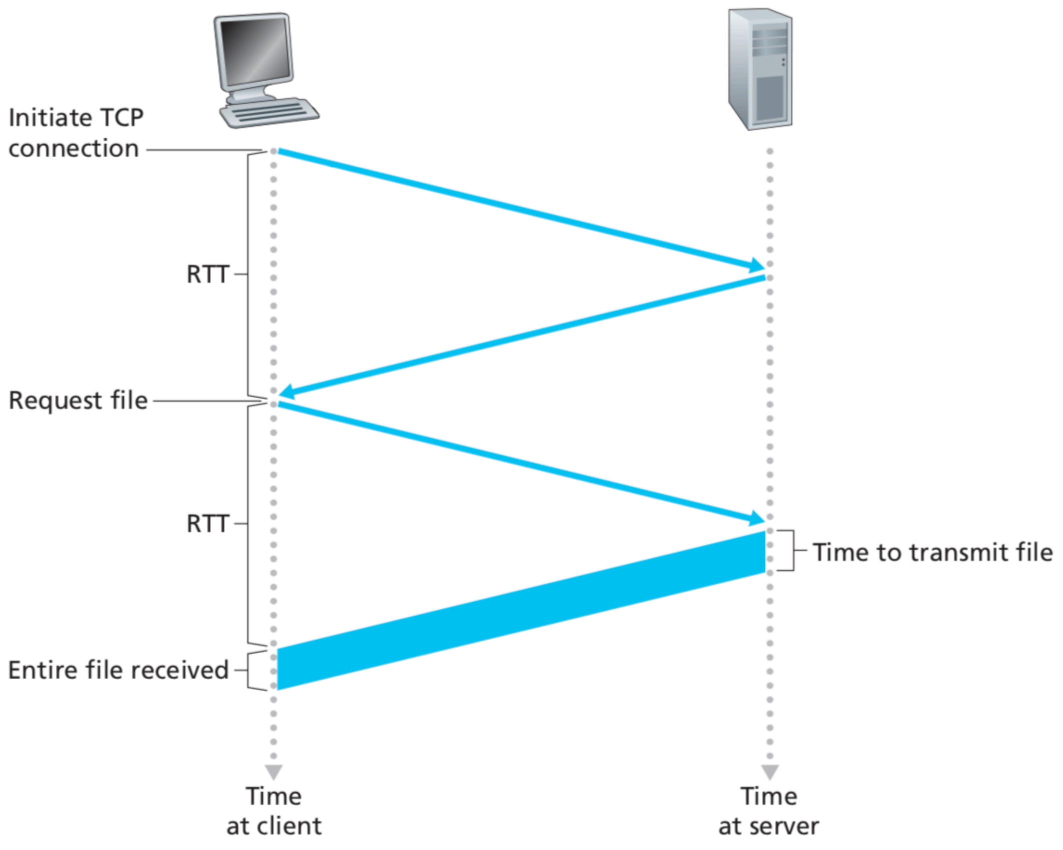


Figure 2.7 ♦ Back-of-the-envelope calculation for the time needed to request and receive an HTML file

HTTP with Persistent Connections

Non-persistent connections have some shortcomings:

1. **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.
2. **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. In particular, an entire Web page (in the example above, the base HTML file and the 10 images) can be sent over a single persistent TCP connection.

Moreover, multiple Web pages residing on the same server can be sent from the server to the same client over a single persistent TCP 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.**

2.2.3 HTTP Message Format

Two types of HTTP messages: request messages and response messages.

HTTP Request Messages

```
GET /somedir/page.html HTTP/1.1
Host: www.someschool.edu
Connection: close
User-agent: Mozilla/5.0
Accept-language: fr
```

- Ordinary ASCII text.
- First line: **request line**
- Other lines: **header lines**
- The first line has three fields: the method field, the URL field, and the HTTP version field.
    - method field possible values: `GET, POST, HEAD, PUT, DELETE`
    - The great majority of HTTP request messages use the GET method. The GET method is used when the browser requests an object, with the requested object identified in the URL field.
- `Connection: close` means 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 general format of a request message, as shown in Figure 2.8 (cr -> carriage return, lf -> line feed):

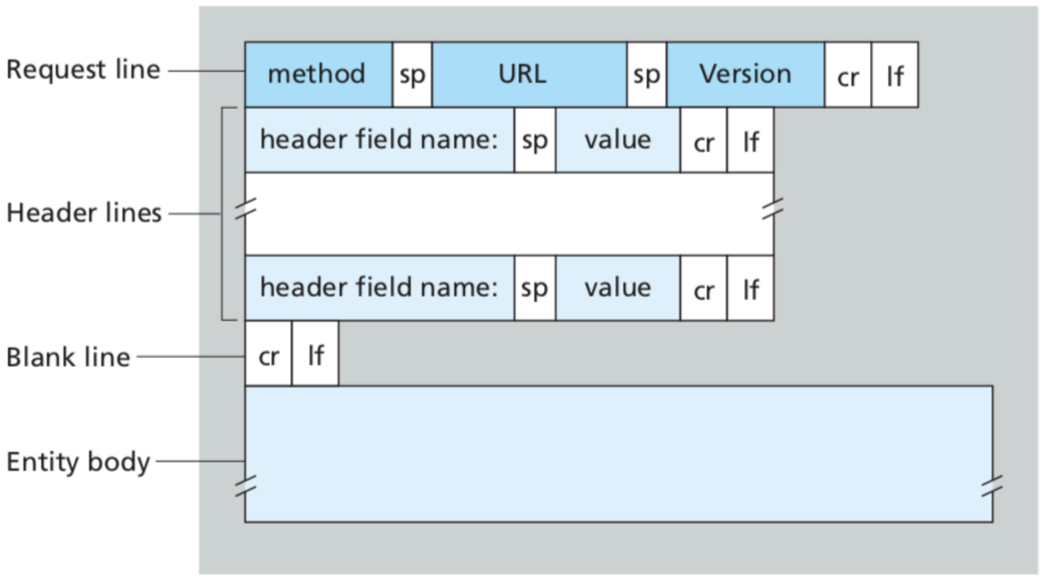


Figure 2.8 ♦ General format of an HTTP request message

After header lines, there is an entity body. The entity body is empty with the `GET` method, but is used with the `POST` method. The `POST` method is usually used when the user fill out a form. With a POST message, the user is still requesting a Web page from the server, but the specific contents of the Web page depend on what the user entered into the form fields.

We would be remiss if we didn't mention that a request generated with a form does not necessarily use the `POST` method. Instead, HTML forms often use the `GET` method and include the inputted data (in the form fields) in the requested URL:

```
www.somesite.com/animalsearch?monkeys&bananas.
```

- The `HEAD` method is similar to the `GET` method. When a server receives a request with the `HEAD` method, it responds with an HTTP message but it leaves out the requested object. Application developers often use the `HEAD` method for debugging.
- The `PUT` method is often used in conjunction with Web publishing tools. It allows a user to upload an object to a specific path (directory) on a specific Web server. The `PUT` method is also used by applications that need to upload objects to Web servers.
- The `DELETE` method allows a user, or an application, to delete an object on a Web server.

HTTP Response Messages

```
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 ...)
```

- First line: **Status line**.
    - the protocol version field
    - a status code
    - a corresponding status message
- Six **header lines**:
    - `Connection: close` tells the client that it is going to close the TCP connection after sending the message.
    - `Date` indicates the time and date when the HTTP response was created and sent by the server. Note that this is not the time when the object was created or last modified; 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.
    - `Server`, which is analogous to to the User-agent, the header line in the HTTP request message
    - `Last-Modified` is critical for object caching, both in the local client and in network cache servers (also known as proxy servers).
    - `Content-Length` indicates the number of bytes in the object being sent.
    - `Content-Type` indicates that the object in the entity body is HTML text.

  Note: These six lines are only a few of the total number of header lines.
- **Entity body**: the meat of the message—it contains the requested object itself (represented by data data data data data ...).

General format of a response message:

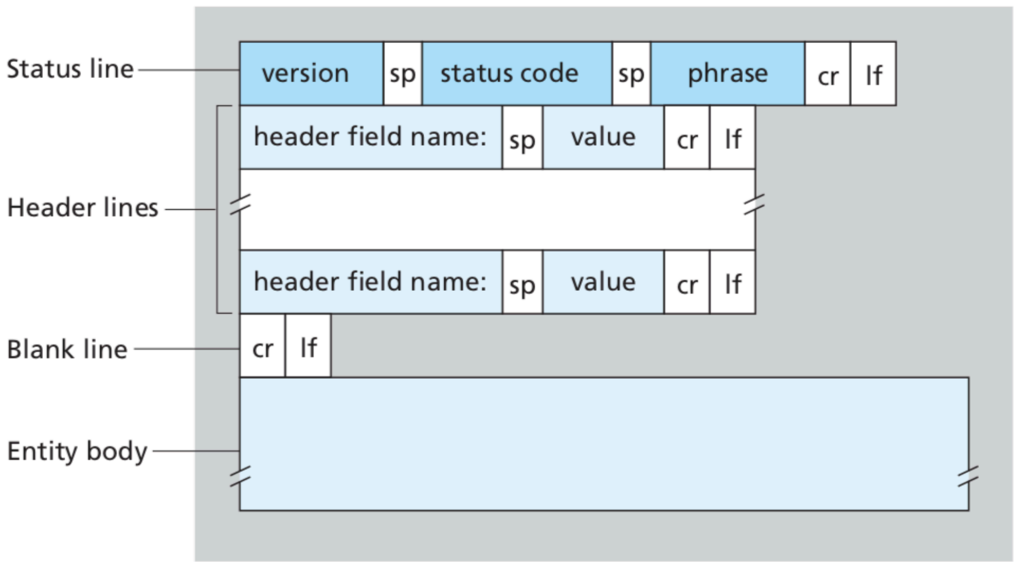


Figure 2.9 ♦ General format of an HTTP response message

Some common status codes:

`200 OK`: request succeeded and information returned `301 Moved Permanently`: the object has moved, the new location is specified in the header of the response `400 Bad Request`: generic error code: request can not be understood by the server `404 Not Found`: The requested document doesn't exist on the server `505 HTTP Version Not Supported`: The requested HTTP protocol version is not supported by the server

2.2.4 User-Server Interaction: Cookies

The HTTP server is *stateless* as we mentioned before. 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. Thus, HTTP uses **cookies**, which allow sites to keep tract of users.

Cookie has 4 components:

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

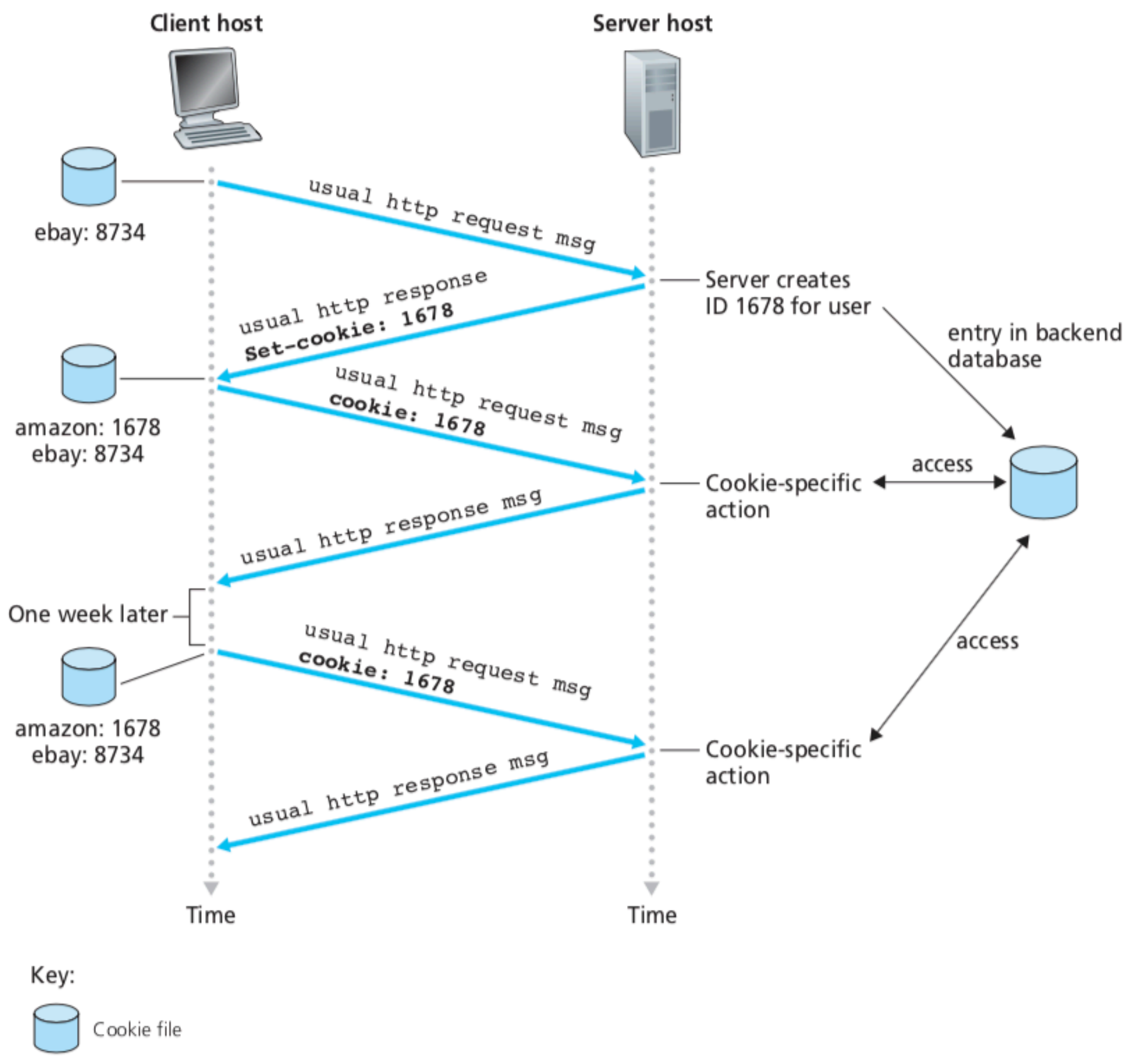


Figure 2.10 ♦ Keeping user state with cookies

User connects to website using cookies:

- Server creates a unique identification number and creates an entry in its back-end database indexed by the identification number-server responds to user's browser including in the header: `Set-cookie: identification number`
- The browser will append a line to the special cookie file that includes the hostname of the server and the identification number in the header
- Each time the browser will request a page, it will consult the cookie file, extract the identification number for the site and put a cookie header line including the identification number

We see that cookies can be used to identify a userand the server can track the user's activity. But that's why cookies are also controversial because they can also be considered as an invasion of privacy: a Web site can learn a lot about a user and potentially sell this information to a third party.

Hence, **cookies can be used to create a user session layer on top of stateless HTTP**.

2.2.5 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.

1. The browser establishes a TCP connection to the Web cache and sends an HTTP request for the object to the Web cache.
2. 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.
3. If it doesn't, the Web caches opens a TCP connection to the original server, then it sends an HTTP requests for the object into the cache-to-server TCP connection. After receiving this request, the origin server sends the object within the HTTP response to the Web cache.
4. The Web cache receives the object, stores a copy in its local storage and then sends a copy, within an HTTP response message, to the client browser (over the existing TCP connection between the client browser and the Web cache).

Thus, the **cache is both a server and a client** at the same time.

Typically a Web cache is purchased and installed by an ISP. Advantages for the use of web cache:

1. It can substantially reduce the response time for a client request, particularly if the bottleneck bandwidth between the client and the origin server is much less than the bottleneck bandwidth between the client and the cache.
2. Web caches can substantially reduce traffic on an institution's access link to the Internet, thus the company doesn't need to upgrade the bandwidth thereby reducing the cost.
3. Web caches can substantially reduce Web traffic in the Internet as a whole, thereby improving performance for all applications.

Check the case study in pg112 to gain a deeper understanding of the benefits of caches.

Through the use of **Content Distribution Networks (CDNs)**, Web caches are increasingly playing an important role in the Internet. A CDN company installs many geographically distributed caches throughout the Internet, thereby localizing much of the traffic.

2.2.6 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**. 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
2. the request message includes an `If-Modified-Since:` header line.

How the conditional `GET` works:

1. On the behalf of a requesting browser, a proxy cache sends a request message to a Web server.
2. The Web server sends a response message with the requested object to the cache. The cache forwards the object to the requesting browser but also caches the object locally.
3. One week later, another browser requests the same object via the cache, and the object is still in the cache. But this object may be modified in the server. The cache performs an up-

to-date check by issuing a conditional `GET`, which tells the server to send the object only if the object has been modified since the specified date.
4. The Web server sends a response message to the cache. If not modified, the Web server still sends a response message but does not include the requested object in the response message in response to conditional `GET`.

2.3 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.

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).

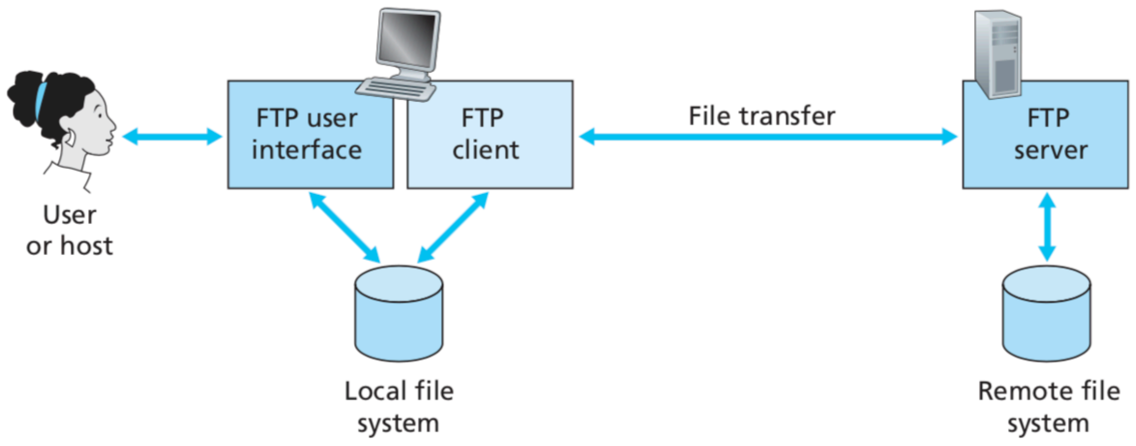


Figure 2.14 ♦ FTP moves files between local and remote file systems

HTTP and FTP are both file transfer protocols and both run on top of TCP. However, the most striking difference is that FTP uses two parallel TCP connections to transfer a file, a **control connection** and a **data connection**. The control connection is used for sending control information between the two hosts—information such as user identification, password, commands to change remote directory, and commands to “put” and “get” files. The 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**. HTTP then sends request and response header lines into the same TCP connection that carries the transferred file itself. For this reason, HTTP is said to send its control information **in-band**. **SMTP, the main protocol for electronic mail, also sends control information in-band.**

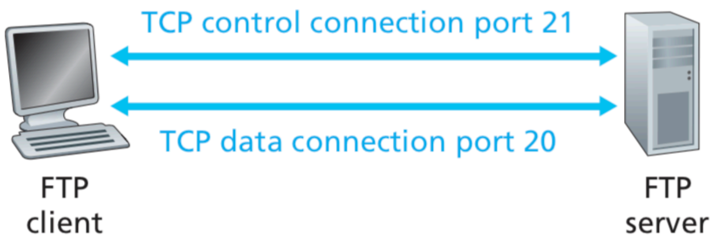


Figure 2.15 ♦ Control and data connections

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 number 21**. 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 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**).

Throughout a session, the FTP server must maintain **state** about the user, whereas HTTP is **stateless**—it does not have to keep track of any user state. Keeping track of this state information for each ongoing user session significantly constrains the total number of sessions that FTP can maintain simultaneously.

2.3.1 FTP Commands and Replies

The commands are in **7-bit ASCII format**, which is readable by people, and followed by a reply from server.

Each command consists of four uppercase ASCII characters, some with optional arguments. Some of the more common commands are given below:

- `USER username`: Send the user identification to the server.
- `PASS password`: Send the user password to the server.
- `LIST`: 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`: Retrieve 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`: Store 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.

The replies are **three-digit numbers**, with an optional message following the number.

Some typical replies:

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

2.4 Electronic Mail in the Internet

E-mail is an asynchronous communication medium. Modern e-mail has many powerful features, including messages with attachments, hyperlinks, HTML-formatted text, and embedded photos.

Three major components of Internet mail system: **user agents**, **mail servers** and the **Simple Mail Transfer Protocol (SMTP)**.

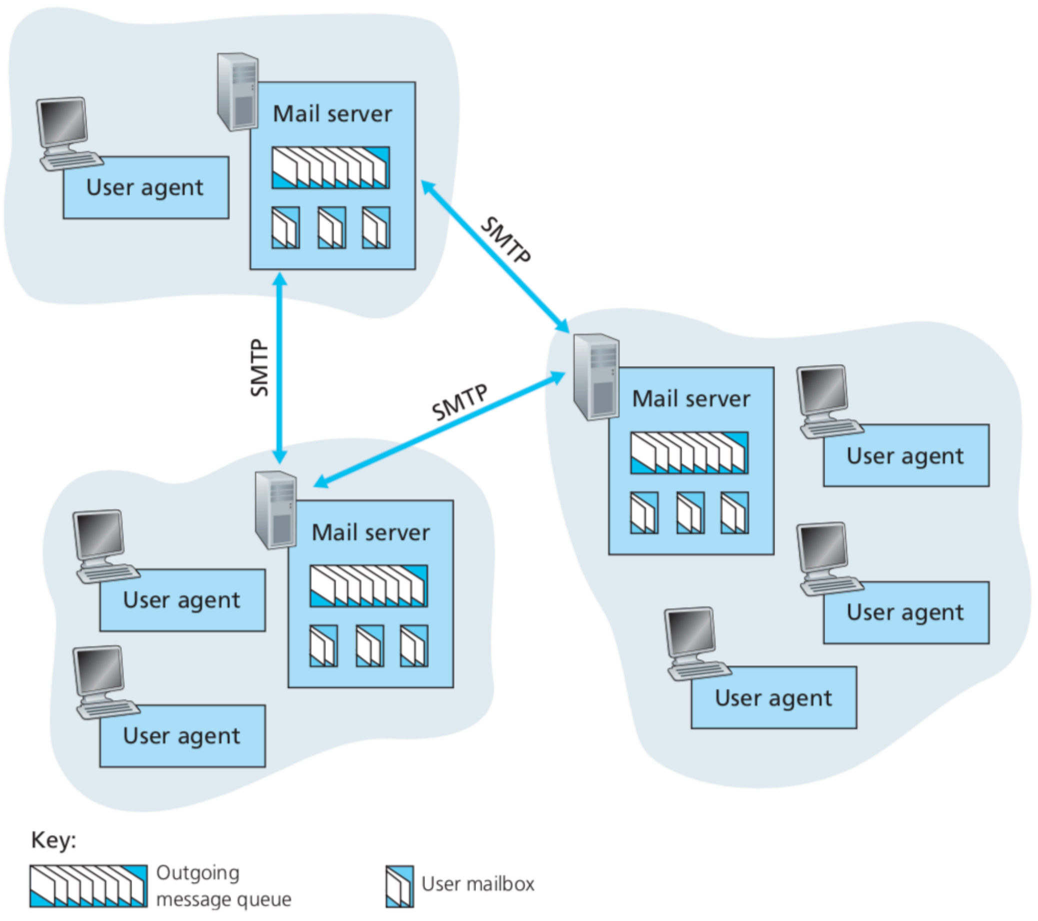


Figure 2.16 ♦ A high-level view of the Internet e-mail system

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. The mail server containing receivers' mailbox authenticates receiver (with usernames and passwords) when receiver wants to access mailbox.

If sender’s server can't deliver mail to receiver’s server, sender’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 no success after several days, the server removes the message and notifies the sender with an e-mail message.

SMTP is the principal application-layer protocol for Internet electronic mail, which uses the reliable data transfer service of TCP to transfer mail from the sender’s mail server to the recipient’s mail server. Like most application-layer protocols, SMTP has two sides: client side, which executes on the sender’s mail server, and a server side, which executes on the recipient’s mail server. Both the client and server sides of SMTP run on every mail server. When a mail server sends mail to other mail servers, it acts as an SMTP client. When a mail server receives mail from other mail servers, it acts as an SMTP server.

2.4.1 SMTP

SMTP, defined in RFC 5321, is at the heart of Internet electronic mail. **SMTP transfers messages from senders’ mail servers to the recipients’ mail servers. SMTP uses persistent connections** and is much older than HTTP.

Although SMTP has numerous wonderful qualities, it is nevertheless a legacy technology that possesses certain archaic characteristics. For example, it restricts the body (not just the headers) of all mail messages to simple 7-bit ASCII. This restriction made sense back to 1980s when transmission capacity was scarse, but today when it is 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. **HTTP does not require multimedia data to be ASCII encoded before transfer.**

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

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.

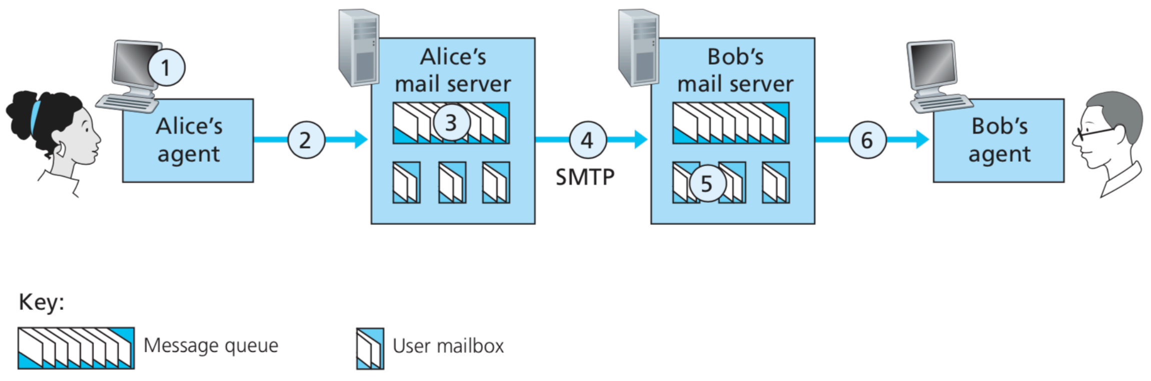


Figure 2.17 ♦ Alice sends a message to Bob

It is important to observe that **SMTP does not normally use intermediate mail servers for sending mail**, even when the two mail servers are located at opposite ends of the world. Suppose Alice's mail server is in Hongkong and Bob's server is in St. Louis, the TCP connection is a direct connection between the Hong Kong and St. Louis servers. Even if Bob's server is down, the message remains in Alice's mail server and waits for a new attempt—the message does not get placed in some intermediate mail server.

A closer look at how SMTP transfers a message from a sending mail server to a receiving mail server.

1. 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.
2. Once this connection is established, the server and client perform some application-layer handshaking - SMTP clients and servers introduce themselves before transferring information. During this SMTP handshaking phase, the SMTP client indicates the e-mail address of the sender (the person who generated the message) and the e-mail address of the recipient.
3. 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.
4. The client then repeats this process over the same TCP connection if it has other messages to send to the server; otherwise (**persistent connection**), 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).

```
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
```

2.4.2 Comparison with HTTP

The similarities shared by HTTP and SMTP:

1. **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.
2. **Both persistent HTTP and SMTP use persistent connections when transferring the files.**

The differences between HTTP and SMTP:

1. 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. In particular, the TCP connection is initiated by the machine that wants to **receive** the file. On the other hand, SMTP is primarily a **push protocol**—the sending mail server pushes the file to the receiving mail server. In particular, the TCP connection is initiated by the machine that wants to **send** the file.
2. **SMTP requires each message, including the body of each message, to be in 7-bit ASCII format, while HTTP data does not impose this restriction.** 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.
3. The way 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.

2.4.3 Mail Message Formats

When an e-mail message is sent from one person to another, a header containing peripheral information precedes the body of the message itself. This peripheral information is contained in a series of header lines, which are defined in RFC 5322. The header lines and the body of the message are separated by a blank line (that is, by CRLF). RFC 5322 specifies the exact format for mail header lines as well as their semantic interpretations.

Every header must have a `From: header line` and a `To: header line`; a header may include a `Sub- ject: header line` as well as other optional header lines.

It is important to note that these header lines are *different* from the SMTP commands: The com-mands in that section were part of the SMTP handshaking protocol; the header lines examined in this section are part of the mail message itself.

A typical message header looks like this:

```
From: alice@crepes.fr
To: bob@hamburger.edu
Subject: Searching for the meaning of life.
```

After the message header, a blank line follows; then the message body (in ASCII) follows.

2.4.4 Mail Access Protocols

Today, mail access uses a client-server architecture—the typical user reads e-mail with a client that executes on the user's end system, for example, on an office PC, a laptop, or a smartphone. By executing a mail client on a local PC, users enjoy a rich set of features, including the ability to view multimedia messages and attachments.

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, Alice's mail server would dialogue directly with Bob's PC. Recall that a mail server manages mailboxes and runs the client and server sides of SMTP. If Bob's mail server were to reside on his local PC, then Bob's PC would have to remain always on, and connected to the Internet, in order to receive new mail, which can arrive at any time.

This is impractical for many Internet users. Instead, a typical user runs a user agent on the local PC but accesses its mailbox stored on an always-on shared mail server. This mail server is shared with other users and is typically maintained by the user's ISP (for example, university or company).

Let's consider the path an e-mail message takes when it is sent from Alice to Bob. 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.

Why the two-step procedure? Primarily because without relaying through Alice's mail server, Alice's user agent doesn't have any recourse to an unreachable destination mail server. By having Alice first deposit the e-mail in her own mail server, Alice's mail server can repeatedly try to send the message to Bob's mail server, say every 30 minutes, until Bob's mail server becomes operational. (And if Alice's mail server is down, then she has the recourse of complaining to her system administrator!)

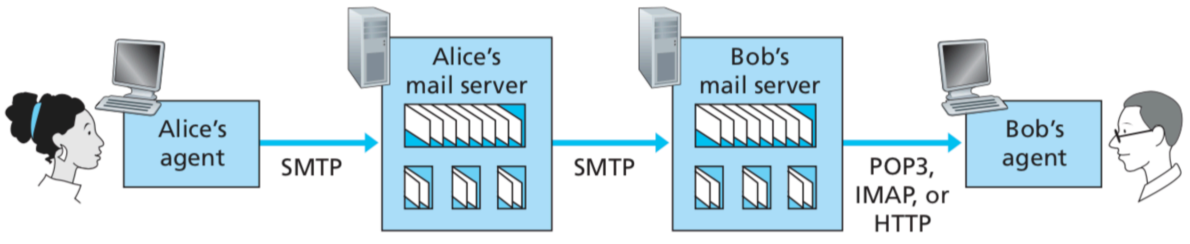


Figure 2.18 ♦ E-mail protocols and their communicating entities

From figure 2.18 we can see that SMTP does two jobs:

1. Transfer mail from the sender's mail server to the recipient's mail server.

2. Transfer mail from the sender's user agent to the sender's mail server.

A mail access protocol, such as POP3, then is used to transfer mail from the recipient's mail server to the recipient's user agent.

But how does a recipient like Bob, running a user agent on his local PC, obtain his messages, which are sitting in a mail server within Bob's ISP? Note we cannot use SMTP to obtain the messages because obtaining the messages is a pull operation, whereas SMTP is a push protocol. This is solovd by introducing a special mail access protocol that transfers messages from Bob's mail server to his local PC. There are currently a number of popular mail access protocols, including **Post Office Protocol—Version 3 (POP3)**, **Internet Mail Access Protocol (IMAP)**, and **HTTP**.

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 authorization, the user agent sends a username and a password (in the clear) to authenticate the user.
- During the 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:

1. `+OK` (sometimes followed by server-to-client data), used by the server to indicate that the previous command was fine.
2. `-ERR`, used by the server to indicate that something was wrong with the previous command (If you misspell a command, the POP3 server will reply with an -ERR message).

The authorization phase has two principal commands: `user <username>` and `pass <password>`. We can see the following example:

```
telnet mailServer 110 +OK POP3 server ready user bob
+OK
pass hungry
+OK user successfully logged on
```

In transaction phase, a user agent using POP3 can often be configured (by the user) to "**download and delete**" or to "**download and keep**." The sequence of commands issued by a POP3 user agent depends on which of these two modes the user agent is operating in.

In the download-and-delete mode, the user agent will issue the `list`, `retr`, and `dele` commands. The user agent first asks the mail server to list the size of each of the stored messages. The user agent then retrieves and deletes each message from the server. 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 he cannot read a message from his laptop after reading it in the PC. 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 the message from his laptop.

During a POP3 session between a user agent and the mail server, **the POP3 server maintains some state information**; in particular, it keeps track of which user messages have been marked deleted. **However, the POP3 server does not carry state information across POP3 sessions**. This lack of state information across sessions greatly simplifies the implementation of a POP3 server.

IMAP