Module – 1
Application Layer

Syllabus Contents

Application Layer: Principles of Network Applications: Network Application Architectures, Processes Communicating, Transport Services Available to Applications, Transport Services Provided by the Internet, Application-Layer Protocols. The Web and HTTP: Overview of HTTP, Non-persistent and Persistent Connections, HTTP Message Format, User-Server Interaction: Cookies, Web Caching, The Conditional GET, File Transfer: FTP Commands & Replies, Electronic Mail in the Internet: SMTP, Comparison with HTTP, Mail Message Format, Mail Access Protocols, DNS; The Internet's Directory Service: Services Provided by DNS, Overview of How DNS Works, DNS Records and Messages, Peer-to-Peer Applications: P2P File Distribution, Distributed Hash Tables, Socket Programming: creating Network Applications: Socket Programming with UDP, Socket Programming with TCP.

1.1 Principles of Network Applications

- The core of network application development is writing programs that run on different end systems and communicate with each other over the network.

  Example 1: 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, smartphone etc.) and the **Web server program** running in the Web server host.

  Example 2: In a P2P file-sharing system there is a program in each host that participates in the file-sharing community. Here, the programs in the various hosts may be similar or identical.

- Thus, when developing new application, we need to write software that will run on multiple end systems. This software could be written, for example, in C, Java, or Python.
- Basic design—namely, confining application software to the end systems—as shown in Figure 1.1, has facilitated the rapid development and deployment of a vast array of network applications.

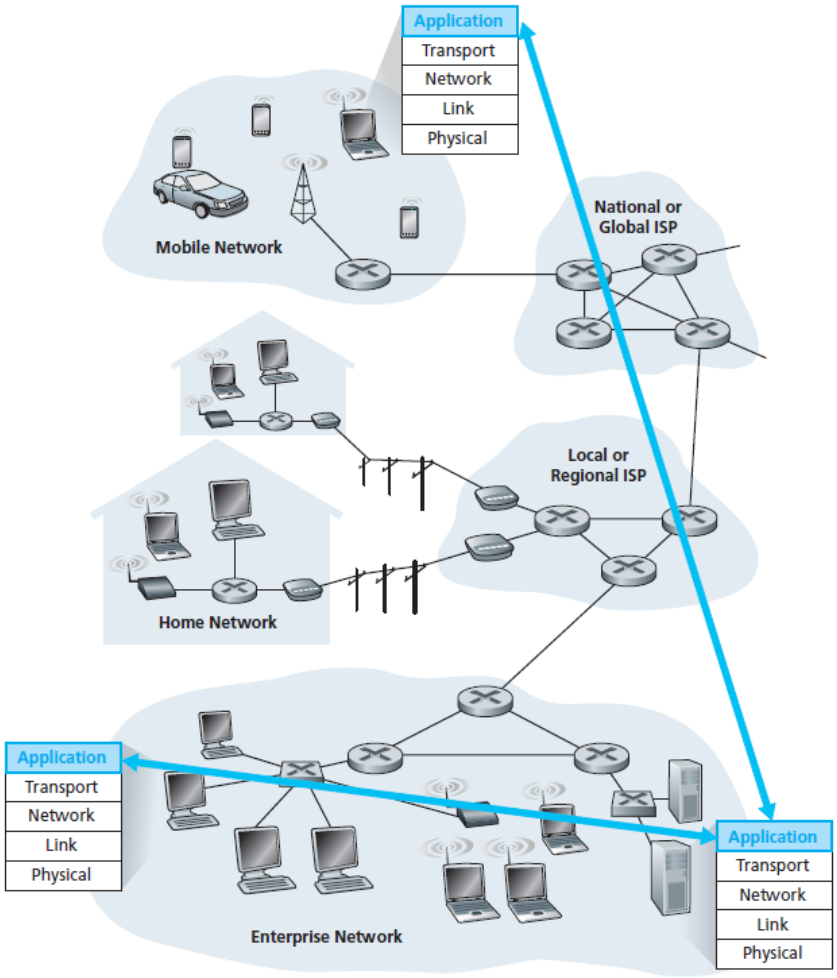

Figure. 1.1: Communication for a network application takes place between end systems at the application layer

1.1.1 Network Application Architectures

The **application architecture** is designed by the application developer and describes the structure of application over the various end systems.

The two predominant architectural paradigms used in modern network applications are:

1. **The client-server architecture**
2. **The peer-to-peer (P2P) architecture**

Client-server architecture:

- Here there is an always-on host, called the *server*, which services requests from many other hosts, called *clients*.

  Example: 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.

Characteristics of Client Server architecture:

- Clients do not directly communicate with each Other. For example, in the Web application, two browsers do not directly communicate.
- The server has a fixed, well-known address, called an IP address

- Since 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.
- Applications with a client-server architecture include the Web, FTP, Telnet, and e-mail.

The client-server architecture is shown in Figure 1.2 below.

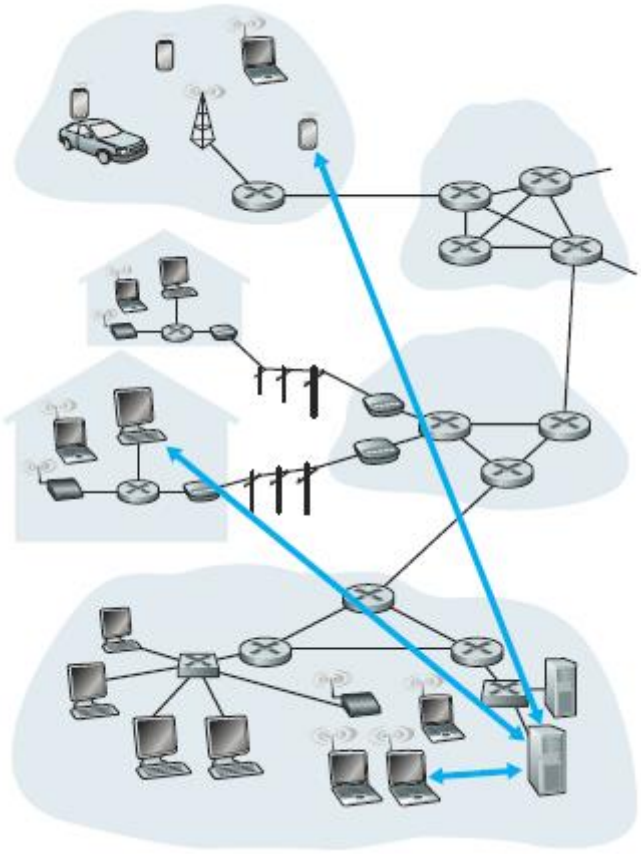

Figure 1.2: Client-server architecture

.

- In a client-server application, a single-server host is incapable of keeping up all the requests from clients.
- So, a **data center**, housing a large number of hosts, is often used to create a powerful virtual server.
- **Example**: Google has 30 to 50 data centers distributed around the world, which collectively handle search, YouTube, Gmail, and other services. A data center can have hundreds of thousands of servers, which must be powered and maintained.

Peer to Peer Architecture :

- In a **P2P architecture**, there is minimal or no always on dedicated servers in data centers.
- 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 2 peers residing in homes, universities, and offices.
- Because the peers communicate without passing through a dedicated server, the architecture is called peer-to-peer.

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

The P2P architecture is illustrated in Figure 1.3 below.

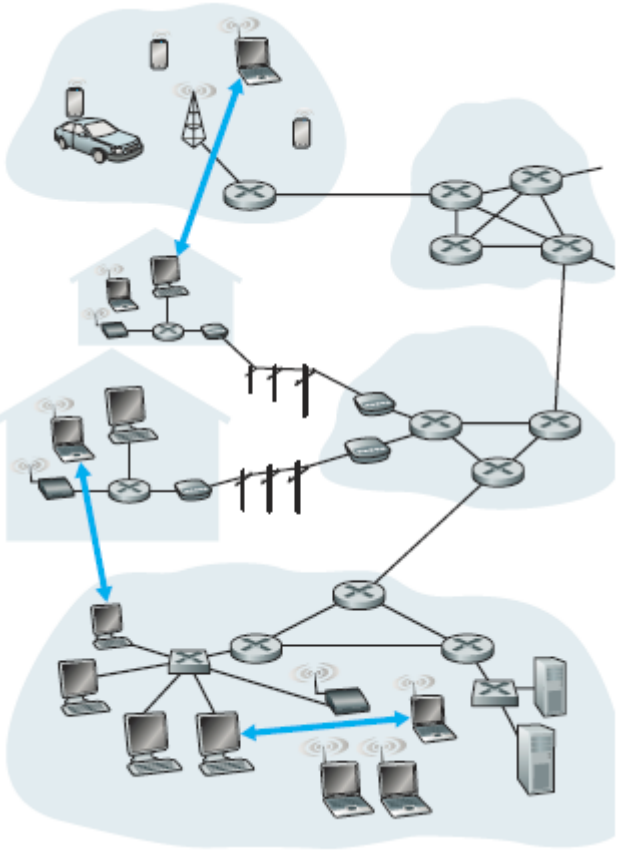

Figure 1.3: P2P architecture

Hybrid architectures:

- Combines both client-server and P2P elements.
- For example: Many instant messaging applications, servers track the IP addresses of users, but user-to-user messages are sent directly between user hosts (without passing through intermediate servers).

Advantages of P2P architecture:

- **Self-scalability:** 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.
- **Sost effective**: Since they do not require significant server infrastructure and server bandwidth unlike client-server paradigm.

Future P2P applications face three major challenges:

1. **ISP Friendly**: *R*esidential 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. Future P2P applications need to be designed so that they are friendly to ISPs.

2. **Security:** Because of highly distributed and open nature, P2P applications can be a challenge to secure.

3. **Incentives:** The success of future P2P applications depends on convincing users to volunteer bandwidth, storage and computation resources to the applications, which is the challenge of incentive design.

1.1.2 Processes Communicating:

- A process is a program under execution within a host 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 label one of the two processes as the **client** and the other process as the **server**.
- In the Web, a browser is a client process and a Web server is a server process.
- In 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 P2P file sharing, a process can be both a client and a server, i.e a process in a P2P file-sharing system can both upload and download files.

We define the client and server processes as follows:

- 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:

- Applications consist of pairs of communicating processes, with the two processes in each pair sending messages to each other.
- 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**.
    - When a process wants to send a message to another process on another host, it sends the message out of its socket. This sending process assumes that there is a transmission on the other side of its socket that will transmit the message to the socket of destination process.

- Once the message arrives at the destination host, the message passes through the receiving process's socket and the receiving process then acts on the message.

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

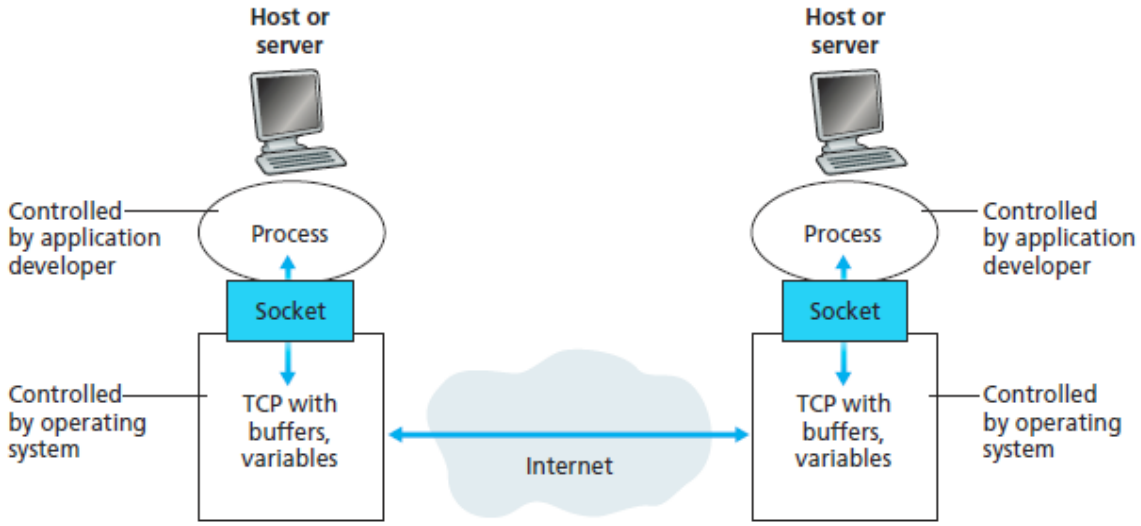

Figure 1.4: Application processes, sockets, and underlying transport protocol

- 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 control that the application developer has on the transport-layer side are:
  - Choice of transport protocol.
  - The ability to fix a few transport-layer parameters such as maximum buffer and maximum segment sizes.

Once the application developer chooses a transport protocol, the application is built using the transport-layer services provided by that protocol.

Addressing Processes

- 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

(2) An identifier that specifies the receiving process in the destination host.

- In the Internet, the host is identified by its **IP address**.
- An IP address is a 32-bit quantity that uniquely identifies the host.
- The **port numbers** identifies the source process and destination process running on the end systems.
- Popular applications have been assigned specific port numbers. For example, a Web server is identified by port number 80. A mail server process is identified by port number 25.

1.1.3 Transport Services Available to Applications

- A socket is the interface between the application process and the transport-layer protocol.
- The application at the sending side pushes messages through the socket.
- At the other side of the socket, the transport-layer protocol has the responsibility of getting the messages to the socket of the receiving process.

Applications requires the services provided by the available transport-layer protocols and thus need to select the protocol with the services that best match application's needs.

The following transport layer services can be offered to applications:

- ✓ Reliable data transfer
- ✓ Throughput
- ✓ Timing
- ✓ Security.

Reliable Data Transfer

- Important service that a transport-layer protocol can potentially provide to an application is process-to-process reliable data transfer.
- Packets can get lost within a computer network.
- 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, **reliable data transfer** guarantee that the data sent by one end of the application is delivered correctly and completely to the other end of the application.
- 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.
- **Loss-tolerant applications**, such as multimedia applications. Eg. Conversational audio/video can tolerate some amount of data loss. These applications do not require reliable data transfer service.

Throughput

- Available **throughput** in the context of a communication session between two processes along a network path is **the rate at which the sending process can deliver bits to the receiving process.**
- Transport-layer protocol could provide, 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 throughput service would appeal to many applications. For example, if an Internet telephony application encodes voice at 32 kbps, it needs to send data into the network and have data delivered to the receiving application at this rate.
- Applications that have throughput requirements are said to be **bandwidth-sensitive applications**.
- **Elastic applications** can make use little throughput as happens to be available. Electronic mail, file transfer, and Web transfers are all elastic applications.

Timing

- A transport-layer protocol can also provide timing guarantees.
- 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. Eg. Internet telephony, virtual environments, teleconferencing and multiplayer games 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; in a multiplayer game or virtual interactive environment, a long delay between taking an action and seeing the response from the environment makes the application feel less realistic.

Security

- 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.
- Such a service would provide confidentiality between the two processes
- A transport protocol can also provide other security services in addition to confidentiality, including data integrity and end-point authentication.

1.1.4 Transport Services Provided by the Internet

- The Internet makes two transport protocols available to applications, **UDP** and **TCP**.
- When application developer creates a new network application for the Internet, developer must choose either UDP or TCP.
- Each of these protocols offers a different set of services to the invoking applications. Figure 1.5 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 1.5: Requirements of selected network applications

TCP Services

The TCP service model includes:

- **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 is called **handshaking** procedure.
  - 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 which 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. 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 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.

Services Not Provided by Internet Transport Protocols

- Throughput or timing guarantees—may not be provided by today's Internet transport protocols.
- Today's Internet can provide satisfactory service to time-sensitive applications, but it cannot provide any timing or throughput guarantees.

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 1.6: Popular Internet applications, their application-layer protocols, and their underlying transport protocols

Figure 1.6 indicates the transport protocols used by some popular Internet applications. We see that e-mail, remote terminal access, the Web, and file transfer use TCP. These applications have chosen TCP primarily because TCP provides reliable data transfer, guaranteeing that all data will eventually get to its destination. Because Internet telephony applications (such as Skype) can often tolerate some loss but require a minimal rate to be effective, developers of Internet telephony applications usually prefer to run their applications over UDP, thereby circumventing TCP's congestion control mechanism and packet overheads.

1.1.5 Application-Layer Protocols

- Network processes communicate with each other by sending messages into sockets.
- An **application-layer protocol** defines how an application's processes, running on different end systems, pass messages to each other.

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.

- Some application-layer protocols are specified in RFCs and are therefore in the public domain. For example, the Web's application-layer protocol, HTTP (the HyperText Transfer Protocol [RFC 2616]), is available as an RFC.
- Many other application-layer protocols are proprietary and intentionally not available in the public domain. For example, Skype uses proprietary application-layer protocols.

The difference between Network applications and application-layer protocols is illustrated below. An application-layer protocol is only one piece of a network application.

Example:

The Web is a client-server application that allows users to obtain documents from Web servers on demand. The Web application consists of many components, including a standard for document formats (that is, HTML), Web browsers (for example, Firefox and Microsoft Internet Explorer), Web servers (for example, Apache and Microsoft servers) and an application-layer protocol. The Web's application-layer protocol, HTTP, defines the format and sequence of messages exchanged between browser and Web server. Thus, HTTP is only one piece of the Web application.

1.2 The Web and HTTP

A major application—the World Wide Web was first internet application developed.

2.2.1 Overview of HTTP

The **HyperText Transfer Protocol (HTTP)** is the Web's application-layer protocol. It is defined in [RFC 1945] and [RFC 2616].

- HTTP is implemented in two programs: a client program and a server program.
- The client program and server program, executing on different end systems, communicate with 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**, implement the server side of HTTP, contains Web objects, each addressable by a URL. 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.

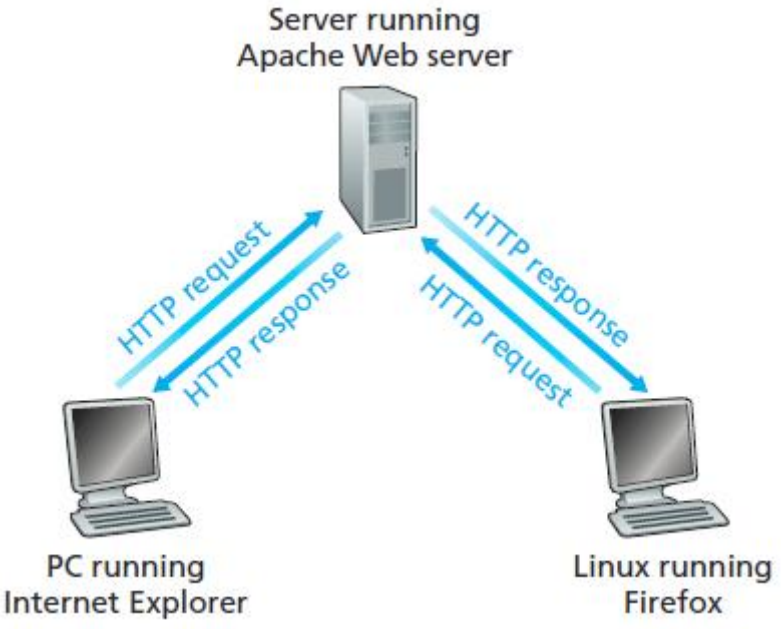

Figure 1.7: HTTP request-response behavior

- When a user requests a Web page (for example, clicks on a hyperlink), 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. 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.
- On the client side the socket interface is the door between the client process and the TCP connection;
- On the server side socket 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.
- The TCP provides a reliable data transfer service to HTTP. This implies that each HTTP request message sent by a client process eventually arrives intact at the server; similarly, each HTTP response message sent by the server process eventually arrives intact at the client.
- 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 resends the object.
- An HTTP server maintains no information about the clients, HTTP is said to be a **stateless protocol**.
- The Web uses the client-server application architecture. A Web server is always on, with a fixed IP address, and it services requests from potentially millions of different browsers.

1.2.2 Non-Persistent and Persistent Connections

- When the client-server interaction is taking place over TCP, the application can use separate TCP connections to send control information and messages. Such TCP connections are called **non-persistent connections**;

- The application may use single TCP connection to send control information and messages, Such TCP connection is called **persistent connections**;
- HTTP uses persistent connections in its default mode, HTTP clients and servers can be configured to use non-persistent connections as well.

HTTP with Non-Persistent Connections

The following steps describes transferring a Web page from server to client for the case of non-persistent connections. Let us 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.

URL for the base HTML file is:

http://www.someSchool.edu/someDepartment/home.index

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 ***/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.

- As the browser receives the Web page, it displays the page to the user. Two different browsers may interpret a Web page in different ways.
- The steps above 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. Thus, in this example, when a user requests the Webpage, 11 TCP connections are generated.
- The amount of time that elapses from when a client requests the base HTML file until the entire file is received by the client is called **round-trip time(RTT)**.
- It 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.

Example:

When a user clicks on a hyperlink, as shown in Figure 1.8, 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.

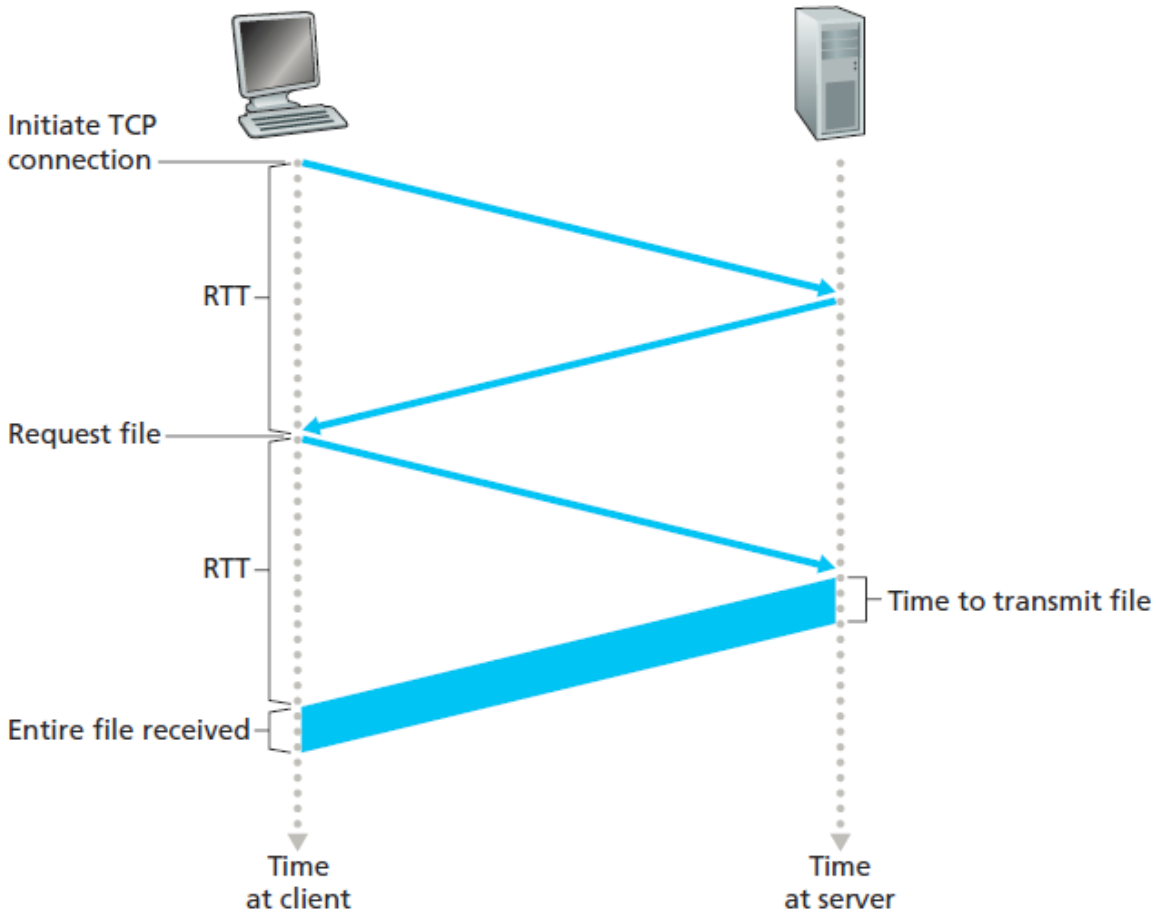

Figure 1.8: Back-of-the-envelope calculation for the time needed to request and receive an HTML file

HTTP with Persistent Connections

Non-persistent connections have some shortcomings.

- 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.

- 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 can be sent over a single persistent TCP connection. 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 .

1.2.3 HTTP Message Format

The HTTP specifications [RFC 1945; RFC 2616] include the definitions of the HTTP message formats.

There are two types of HTTP messages : **Request messages** and **Response messages**.

HTTP Request Message

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 message consists of five lines, each followed by a carriage return and a line feed Message has five lines, a request message can have many more lines or as few as one line.
- 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 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. In this example, the browser is requesting the object */somedir/page.html*. The version is HTTP/1.1.

The header line Host: *www.someschool.edu* specifies the host on which the object resides.

By including the Connection: close header line, the browser is telling the server that it doesn't want 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. This header line is useful because the server can actually send different versions of the same object to different types of user agents.

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.

General format of a request message, as shown in Figure 1.9. After the header lines, there is an "entity body."

- The entity body is empty with the GET method, but is used with the POST method.
- An HTTP client uses the POST method when the user fills out a form—for example, when a user provides search words to a search engine.
- 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.
- If the value of the method field is POST, then the entity body contains what the user entered into the form fields.

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.

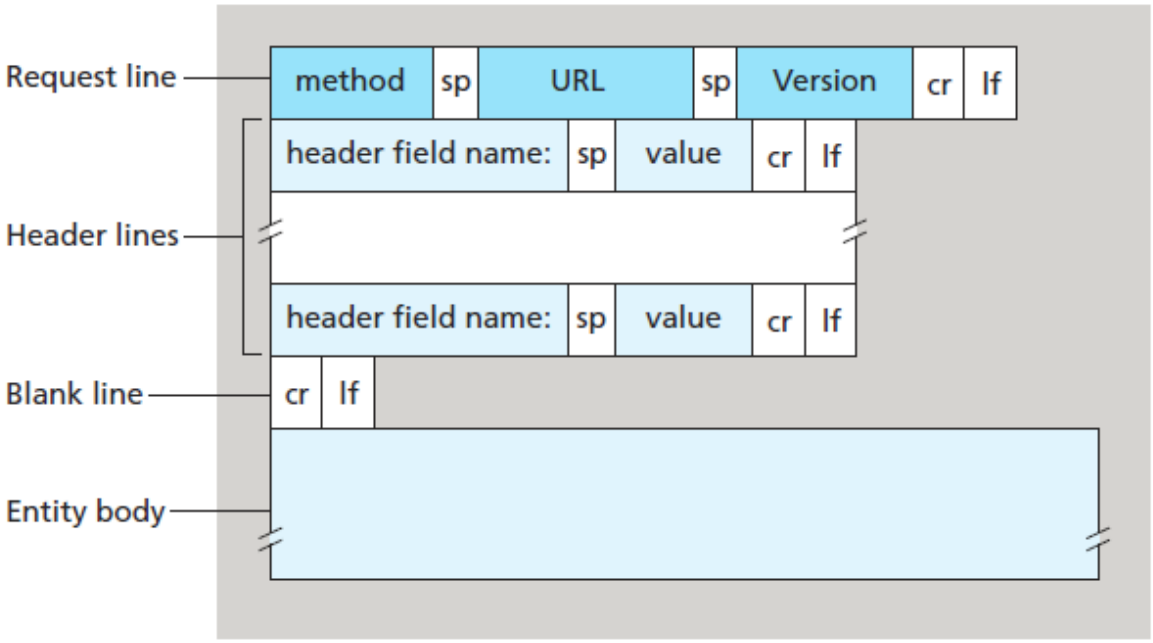

Figure 1.9: General format of an HTTP request message

HTTP Response 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 ...)

Response message has three sections: an initial **status line**, six **header lines** and the **entity body**.

The **entity body** of the message contains the requested object. The **status line** has three fields: the protocol version field, a status code and a corresponding status message.

The **status line** indicates that the server is using HTTP/1.1 and OK indicates that the server has found and is sending the requested object.

The server uses the **Connection: close** header line to indicate the client that server is going to close the TCP connection after sending the message.

The **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**;
The **Last-Modified**: header line indicates the time and date when the object was created or last modified.
The **Content-Length**: header line indicates the number of bytes in the object being sent.
The **Content-Type**: header line indicates that the object in the entity body is HTML text.

The general format of a response message is shown in Figure 1.10.

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;
- *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.

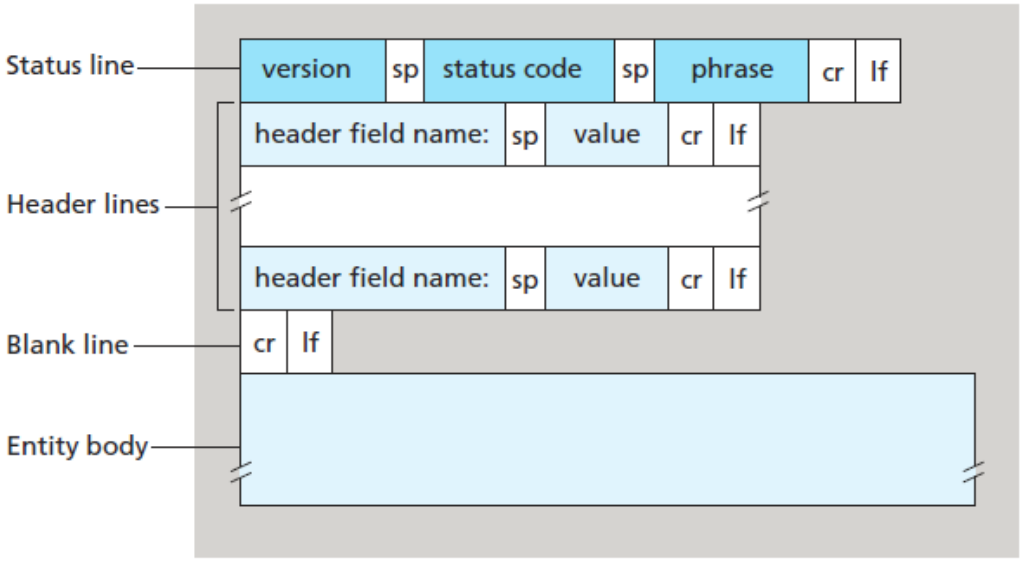

Figure 1.10: General format of an HTTP response message

1.2.4 User – Server Interaction: Cookies

HTTP uses cookies to restrict user access. Cookies allow sites to keep track of users.

Cookie technology has four 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.

Example:

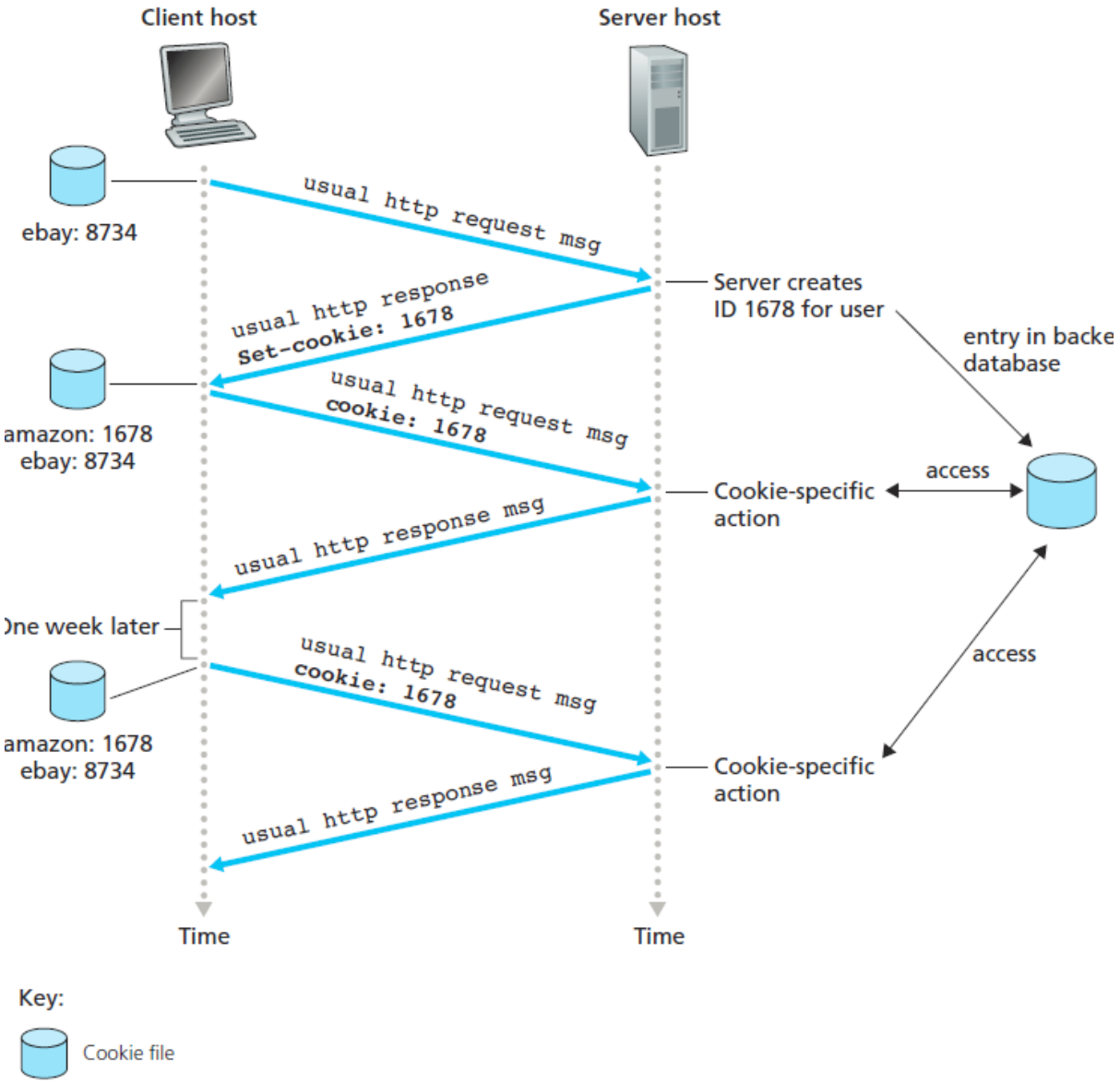

Figure 1.11: Keeping user state with cookies

- User always accesses the Web using Internet Explorer from her home PC and contacts say Amazon.com for the first time.
- Suppose that in the past user 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 users browser, including in the HTTP response a Set-cookie: header, which contains the identification number.

  Set-cookie: 1678
- Browser receives the HTTP response message, it sees the Set-cookie: 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.
- Cookie file already has entry for eBay, since user has visited that site in the past.
- As user 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.***
- The Amazon server is able to track user's activity at the Amazon site. Server knows exactly which pages user 1678 visited, order of visit and number of times.
- Amazon uses cookies to provide its shopping cart service—Amazon can maintain a list of all of user's intended purchases, so that user can pay for them collectively at the end of the session.
- If user returns to Amazon's site, say, one week later, her browser will continue to put the header line Cookie: 1678 in the request messages.
- Amazon also recommends products to user based on Web pages she has visited at Amazon in the past.

1.2.5 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.
- A user's browser can be configured so that all of the user's HTTP requests are first directed to the Web cache.
- Once a browser is configured, each browser request for an object is first directed to the Web cache.

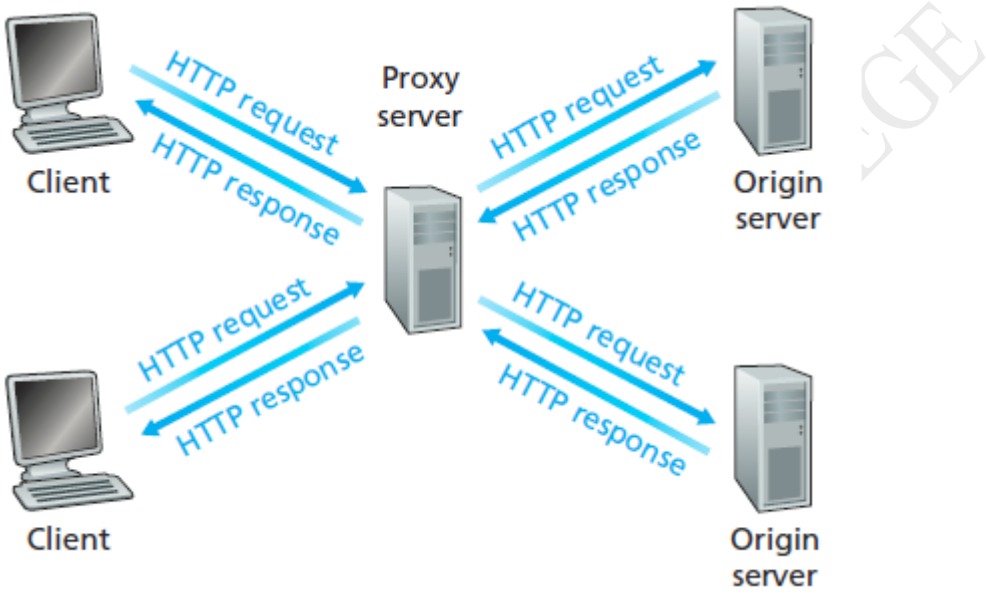

Figure 1.12: Clients requesting objects through a Web cache

Example: Suppose a browser is requesting the object http://www.someschool.edu/campus.gif. Here is what happens:

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 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.
4. The Web cache 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.
5. 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.

- A cache is both a server and a client at the same time. When it receives requests from and sends responses to a browser, it is a server. When it sends requests to and receives responses from an origin server, it is a client.
- A Web cache is purchased and installed by an ISP. For example, a university might install a cache on its campus network and configure all of the campus browsers to point to the cache.

Advantages:

1. A Web cache 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.
3. By reducing traffic, the institution does not have to upgrade bandwidth as quickly, thereby reducing costs.
4. Improves performance for all applications.

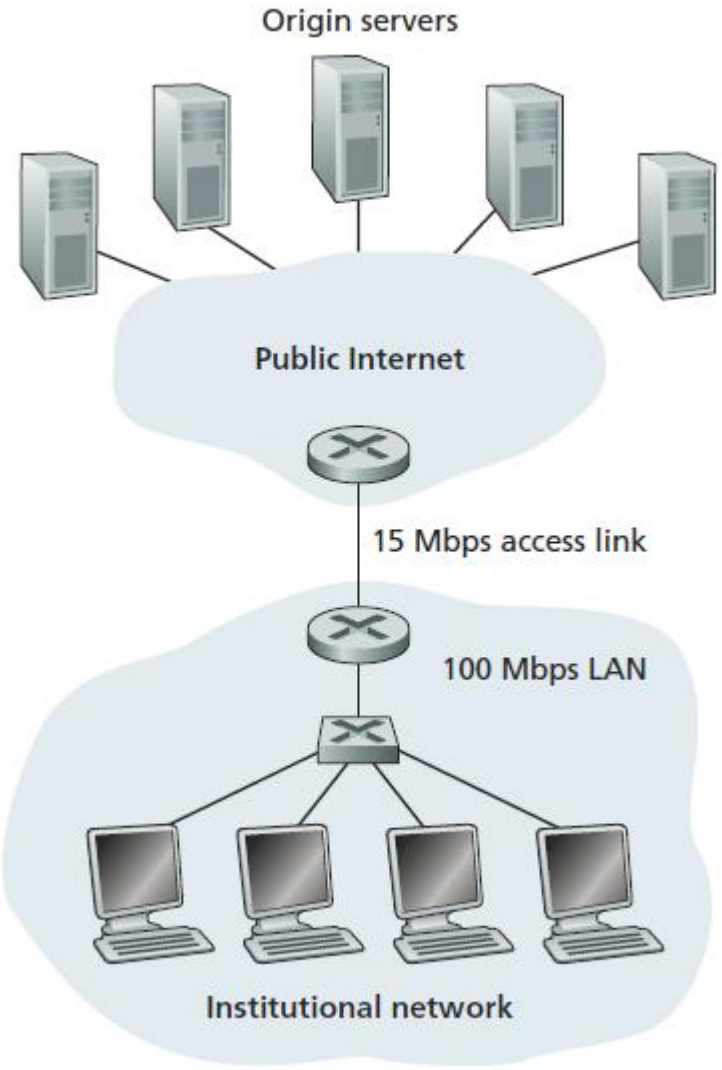

Figure 1.13: Bottleneck between an institutional network and the Internet

This figure shows two networks—the institutional network and 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.

Suppose that the HTTP request messages are negligibly small and thus create no traffic in the networks or in the access link (from institutional router to Internet router).

The amount of time it takes from when the router on the Internet side of the access link in Figure 1.13 forwards an HTTP request until it receives the response is two seconds on average. This delay is "Internet delay."

The total response time i.e 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.

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 (from the Internet router to institution router) is :

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

A traffic intensity of 0.15 on a LAN results in, at most, tens of milliseconds of delay; hence, we can neglect the LAN delay.

As the traffic intensity approaches 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, if not more, which is unacceptable for the institution's users.

Solutions :

1. Increase the access rate from 15 Mbps to, say, 100Mbps. This will lower the traffic intensity on the access link to 0.15, which translates to negligible delays between the two routers. The total response time will roughly be two seconds, that is, the Internet delay. But the institution must upgrade its access link from 15 Mbps to 100 Mbps, a costly proposition.

2. Install a Web cache in the institutional network. Hit rates—the fraction of requests that are satisfied by a cache—typically range from 0.2 to 0.7

Example :

Suppose that the cache provides a hit rate of 0.4 for this institution. Because the clients and the cache are connected to the same high-speed LAN, 40 percent of the requests will be satisfied almost immediately, say, within 10 milliseconds, by the cache.

The remaining 60 percent of the requests still need to be satisfied by the origin servers. But with only 60 percent of the requested objects passing through the access link, the traffic intensity on the access link is reduced from 1.0 to 0.6.

Traffic intensity less than 0.8 corresponds to a small delay, say, tens of milliseconds, on a 15 Mbps link. This delay is **negligible** compared with the two second Internet delay.

Average delay therefore is

0.4 *(0.01 seconds) + 0.6 * (2.01 seconds)

which is just slightly greater than 1.2 seconds. Thus, this second solution provides an even lower response time than the first solution, and it doesn't require the institution to upgrade its link to the Internet.

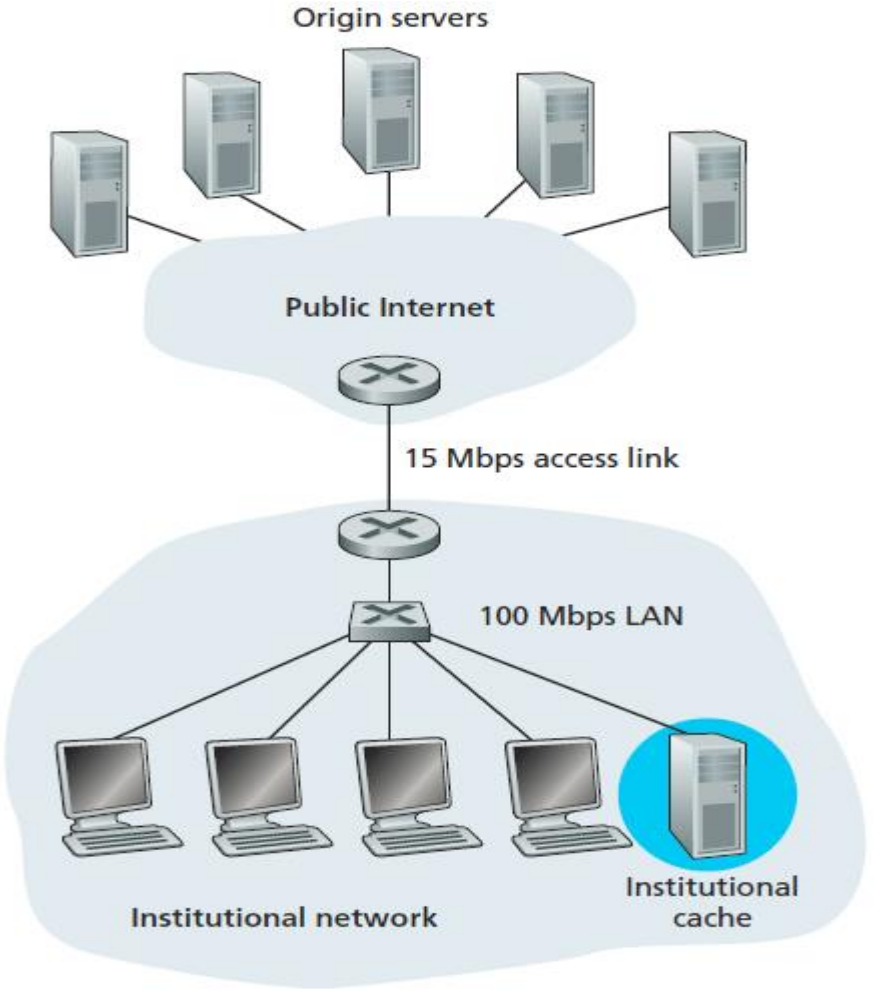

Figure 1.14: Adding a cache to the institutional network

The institution needs to install a Web cache in its work.

The Conditional GET :

The copy of an object residing in the cache may be outdated. The object stored in the Web server may have been modified since the copy was cached at the client.

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 called conditional GET message if

(1) the request message uses the GET method

(2) the request message includes an If-Modified-Since: header line.

Example:

- 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

- 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.

- A 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.

  The cache sends:

  GET /fruit/kiwi.gif HTTP/1.1

  Host: www.exotiquecuisine.com

  If-modified-since: Wed, 7 Sep 2011 09:23:24

  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 1 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.

- 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)

The Web server still sends a response message but does not include the requested object in the response message. Including the requested object would only waste bandwidth and increase user-perceived response time, particularly if the object is large.

304 Not Modified in the status line, tells the cache that it can go ahead and forward its (the proxy cache's) cached copy of the object to the requesting browser.

1.3 File Transfer FTP

- In a 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 1.15, the user interacts with FTP through an FTP user agent.
- 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).

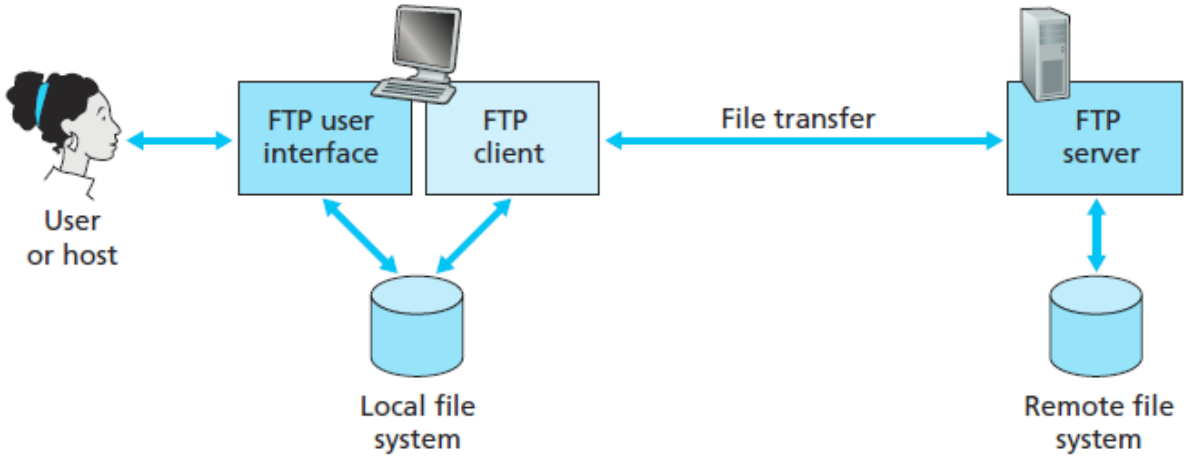

Figure 1.15: FTP moves files between local and remote file systems

- HTTP and FTP are both file transfer protocols and have a common characteristic i.e ., both run on top of TCP. However, the two application-layer protocols have some important differences.

Differences between FTP and HTTP:

Table 1.1: Differences between FTP and HTTP

FTP	HTTP
Uses two parallel TCP connections to transfer a file, a **control connection** and a **data connection**	sends request and response header lines into the same TCP connection
Because FTP uses a separate control connection, FTP is said to send its control information **out-of-band**	HTTP is said to send its control information **in-band**
FTP uses TCP's port number 20 and 21.	HTTP uses TCP's port number 80.

- 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**.

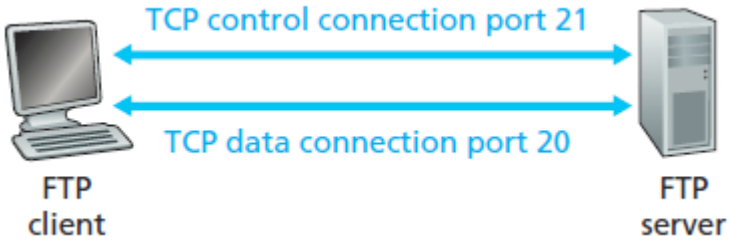

Figure 1.16: Control and data connections

- 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, during the same session, the user wants to transfer another file, FTP opens another data connection.
- FTP keeps 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.
- Throughout a session, the FTP server must maintain **state** about the user.
- HTTP, on the other hand, is stateless—it does not have to keep track of any user state.

FTP Commands and Replies:

- The commands, from client to server, and replies, from server to client, are sent across the control connection in 7-bit ASCII format.
- Each command consists of four uppercase ASCII characters, some with optional arguments. 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.

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

1.4 Electronic Mail in the Internet:

A high-level view of the Internet mail system. It has three major components: **user agents**, **mail servers** and the **Simple Mail Transfer Protocol (SMTP)**.

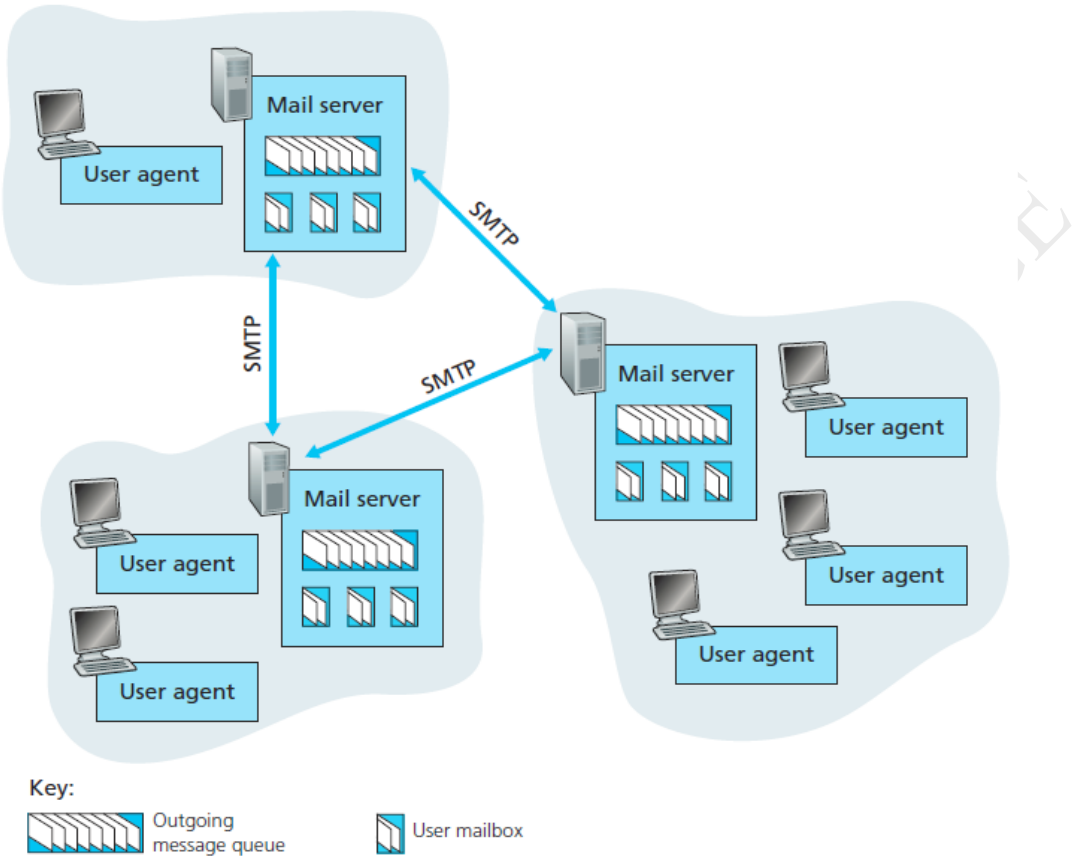

Figure 1.17: A high-level view of the Internet e-mail system

In the context of a sender, 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 for every 30 minutes.
- 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 is the principal application-layer protocol for Internet electronic mail. It uses the reliable data transfer service of TCP to transfer mail from the sender's mail server to the recipient's mail server.

- SMTP has two sides:
  (1) A client side, which executes on the sender's mail server
  (2) 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.
- SMTP restricts the body of all mail messages to simple 7-bit ASCII. It requires binary multimedia data to be encoded to ASCII before being sent over SMTP and the corresponding ASCII message to be decoded back to binary after SMTP transport.

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.

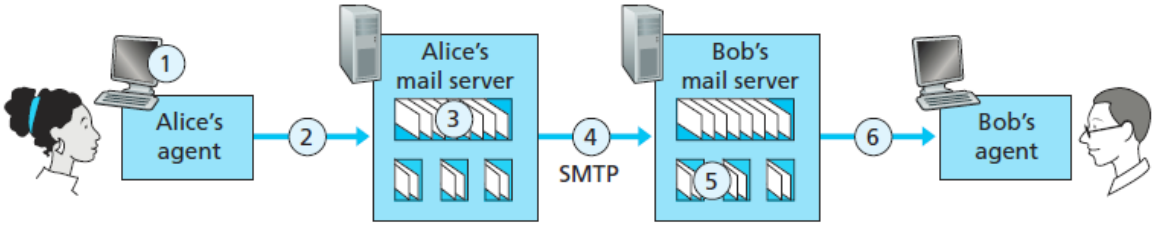

Figure 1.18: Alice sends a message to Bob

- 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.
- 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 this connection is established, the server and client perform some application-layer handshaking before transferring information from one to another, 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 and the e-mail address of the recipient.
- 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.

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?                    // Msg from client
C: How about pickles?                          // Msg
C: .                            //End of msg
S: 250 Message accepted for delivery
C: QUIT
S: 221 hamburger.edu closing connection
```

The client issues five commands: HELO (an abbreviation for HELLO), MAIL FROM, RCPT TO, DATA, and QUIT.

The server issues replies to each command, with each reply having a reply code and some explanation.

SMTP uses persistent connections: 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 a new MAIL FROM: *crepes.fr*, designates the end of message with an isolated period and issues QUIT only after all messages have been sent.

2.4.2 Comparison with HTTP :

- 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 persistent HTTP and SMTP use persistent connections. Thus, the two protocols have common characteristics.

Three important differences between HTTP and SMTP are :

- HTTP is a **pull protocol**—someone loads information on a Web server and users use HTTP to pull the information from the server. In particular, the TCP connection is initiated by the machine that wants to receive the file.

SMTP is 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.

- 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 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.

- HTTP encapsulates each object in its own HTTP response message.

SMTP 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.

The header lines and the body of the message are separated by a blank line.

Each header line contains readable text, consisting of a keyword followed by a colon followed by a value.

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

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 :

SMTP delivers the message from Alice's mail server to Bob's mail server, the message is placed in Bob's mailbox. Bob reads his mail by logging onto the server host and then executing a mail reader that runs on that host.

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.

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.

Instead, a typical user runs a user agent on the local PC but accesses its mailbox stored on an always-on shared mail server.

Mail server is shared with other users and is maintained by the user's ISP .

SMTP has been designed for pushing e-mail from one host to another. The sender's user agent does not interact directly with the recipient's mail server.

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.

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.

The SMTP RFC defines how the SMTP commands can be used to relay a message across multiple SMTP servers.

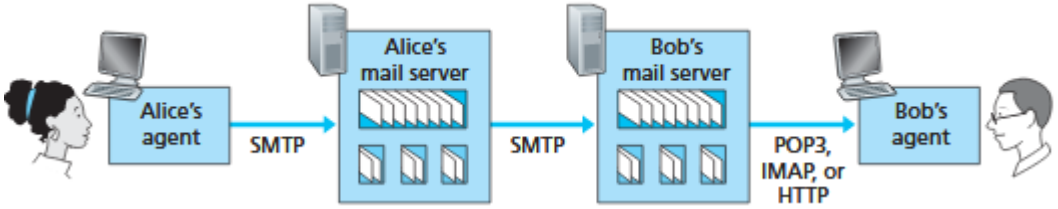

Fig 2.18: E-mail Protocols and their communicating entities

POP3 (POST OFFICE PROTOCOL) :

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

POP3 progresses through three phases: authorization, transaction, and update.

During the first phase, authorization, the user agent sends a username and a password 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>.

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. As an example, suppose the user has two messages in his or her mailbox. In the dialogue below, C:(standing for client) is the user agent and S: (standing for server) is the mail server.

The transaction:

C: list

S: 1 498

S: 2 912

S: .

C: retr 1

S: (blah blah ...

S: .....

S: .....blah)

S: .

C: dele 1

C: retr 2

S: (blah blah ...

S: .....

S: .....blah)

S: .

C: dele 2

C: quit

S: +OK POP3 server signing off

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. 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 disadvantage 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.

Bob can reread messages from different machines; he can access a message from work and access it again later in the week from home.

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.

IMAP (Internet Mail Access Protocol)

With POP3 access, once Bob(receiver) has downloaded his messages to the local machine, he can create mail folders and move the downloaded messages into the folders.

Bob can delete messages, move messages across folders, and search for messages (by sender name or subject).

But 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 a disadvantage of POP3—the POP3 protocol does not provide any means for a user to create remote folders and assign messages to folders.

To overcome this disadvantage, the IMAP protocol was invented. IMAP is a mail access protocol.

- 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.
- IMAP server maintains user state information across IMAP sessions—the names of the folders and which messages are associated with which folders.
- 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 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.

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.

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.

2.5 DNS (DOMAIN NAME SYSTEM) – The Internet's directory service :

Two ways to identify Hosts are : Hostname and IP address. Hostnames are mnemonic identifier. They are easily understandable by humans. Eg. www.Yahoo.com . Every hostname has unique IP address. Eg. 121.7.106.83 . An IP address consists of four bytes and has a rigid hierarchical structure. Each period separates one of the bytes expressed in decimal notation from 0 to 255. They are understandable by routers.

2.5.1 Services Provided by DNS :

DNS is 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 :

(1) a distributed database implemented in a hierarchy of **DNS servers**.

(2) an application-layer protocol that allows hosts to query the distributed database.

The DNS protocol runs over UDP and uses port 53.

DNS is employed by other application-layer protocols—including HTTP, SMTP, and FTP—to translate user-supplied hostnames to IP addresses.

Example, consider 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.

DNS adds an additional delay for sending query and receiving reply.

DNS provides a few other important **services** in addition to translating hostnames to IP addresses:

• **Host aliasing.** A host with a complicated hostname can have one or more alias names. For example, a hostname such as relay1.west-coast.hotmail.com could have, say, two

aliases such as hotmail.com and www.hotmail.com. The hostname relay1.westcoast.hotmail.com is a **canonical hostname**.

Alias hostnames are mnemonic. 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.

• **Mail server aliasing.** E-mail addresses needs to be mnemonic. For example, if Bob has an account with Hotmail, Bob's e-mail address : bob@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.

• **Load distribution.** DNS is also used to perform load distribution among replicated servers, such as replicated Web servers. Busy sites, such as google.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.

The DNS database contains this set of IP addresses. When clients make a DNS query for a name mapped to a set of addresses, the server responds with the entire set of IP addresses, but **rotates** the ordering of the addresses within each reply. Because a client typically sends its HTTP request message to the IP address that is listed first in the set.

DNS rotation distributes the traffic among the replicated servers.

DNS rotation is also used for e-mail so that multiple mail servers can have the same alias name.

2.5.2 Overview of How DNS works ?

DNS in the user's host sends a query message into the network. All DNS query and reply messages are sent within UDP datagrams to port 53. After a delay, ranging from milliseconds to seconds, DNS in the user's host receives a DNS reply message that provides the desired mapping. This mapping is then passed to the invoking application. Network has large number of DNS servers distributed around the globe.

DNS design :

A simple design for DNS would have one DNS server that contains all the mappings. In centralized design, clients simply direct all queries to the single DNS server and the DNS server responds directly to the querying clients.

The problems with a centralized design include:

1.A single point of failure. If the DNS server crashes, entire communication is lost.

2.Traffic volume. A single DNS server would have to handle all DNS queries from million clients.

3.Distant centralized database. A single DNS server cannot be "close to"(nearer to) all the querying clients. This can lead to significant delays.

4.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.

Three classes of DNS servers are—**Root** DNS servers, **Top-level domain** (TLD) DNS servers and **Authoritative** DNS servers—organized in a hierarchy as shown in Figure.

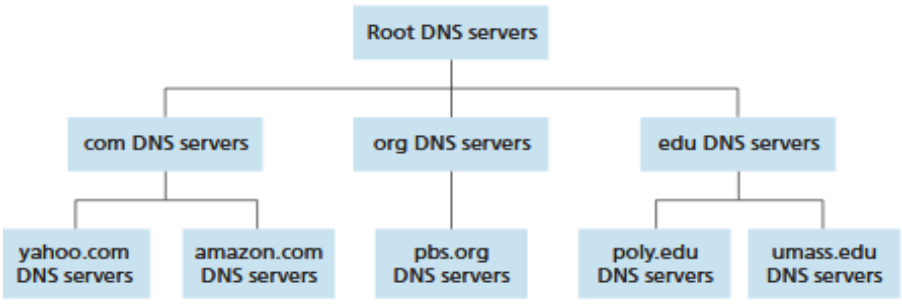

Fig 2.19 Hierarchy of DNS Servers

Suppose a DNS client wants to determine the IP address for the hostname www.amazon.com.

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 (labelled A through M), most of which are located in North America.

Each of 13 root server is actually a network of replicated servers, for both security and reliability purposes. All together, there are around 247 root servers.

• **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.

• **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. An organization can pay & have its own authoritative DNS server, to have these records stored.

Most universities and large companies implement and maintain their own primary and secondary (backup) authoritative DNS server.

There is another important type of DNS server called the **local DNS server**.

- A local DNS server does not strictly belong to the hierarchy of servers but is central to the DNS architecture.

- Each ISP—such as a university, an academic department, an employee's company, or a residential ISP—has a local DNS server .
- When a host connects to an ISP, the ISP provides the host with the IP addresses of one or more of its local DNS servers.
- Determine the IP address of local DNS server by accessing network status windows in Windows or UNIX.
- A host's local DNS server is typically close (nearer) to the host. Eg. For an institutional ISP, the local DNS server may be on the same LAN as the host;
- When a host makes a DNS query, the query is sent to the local DNS server, which acts a proxy, forwarding the query into the DNS server hierarchy.

Example: Suppose the host **cis.poly.edu** desires the IP address of **gaia.cs.umass.edu** and local DNS server is called **dns.poly.edu** and that an authoritative DNS server for **gaia.cs.umass.edu** is called **dns.umass.edu**.

As shown in Figure, the host cis.poly.edu first sends a DNS query message to its local DNS server, **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.

In this example, in order to obtain the mapping for one hostname, eight DNS messa

ges were sent: four query messages and four reply messages.

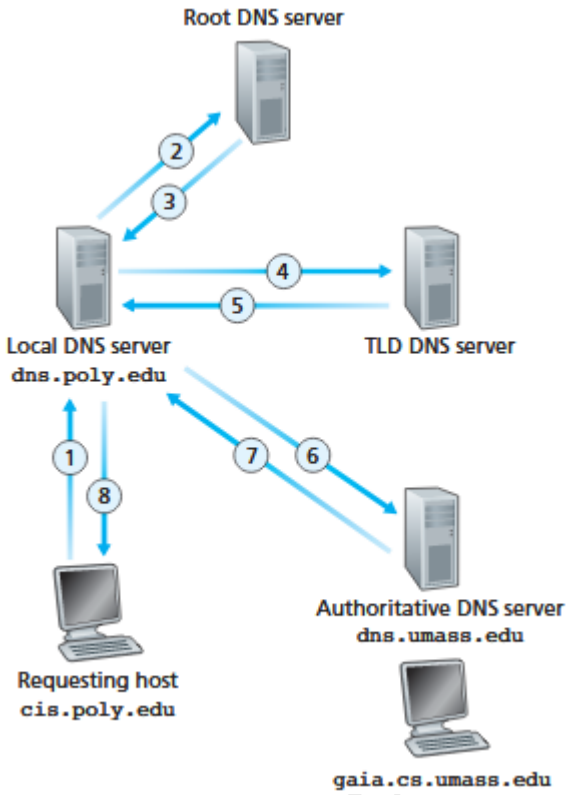

Fig 2.21 Interaction of the various DNS Servers

For example, suppose the University of Massachusetts has a DNS server for the university, called dns.umass.edu. Suppose that each of the departments at the University of Massachusetts has its own DNS server, and that each departmental DNS server is authoritative for all hosts in the department.

When the intermediate DNS server, dns.umass.edu, receives a query for a host with a hostname ending with cs.umass.edu, it returns to dns.poly.edu the IP address of dns.cs.umass.edu, which is authoritative for all hostnames ending with cs.umass.edu.

The local DNS server dns.poly.edu then sends the query to the authoritative DNS server, which returns the desired mapping to the local DNS server, which in turn returns the mapping to the requesting host. Hence, a total of 10 DNS messages are sent.

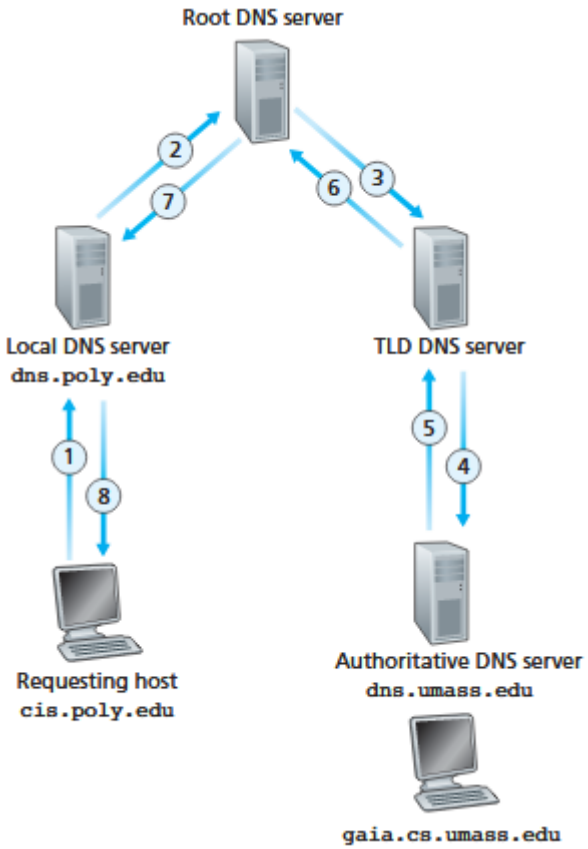

Fig 2.22 Recursive queries in DNS

The example shown in Figure makes use of both recursive queries and iterative queries. 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.

DNS Caching

DNS caching is used to improve the delay performance and to reduce the number of DNS messages moving around the Internet.

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 the above Figure, 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.

DNS servers discard cached information after a period of time.

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.

2.5.3 DNS Records and Messages

The DNS servers that 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.

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.

Example, (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.

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.

Example, suppose an edu TLD server is not authoritative for the host gaia.cs.umass.edu.

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

DNS query and reply messages have the same format, as shown in Figure below.

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.

In the header, there are also four number-of fields. These fields indicate the number of occurrences of the four types of data sections that follow the header.

• 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

(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.

A reply can return multiple RRs in the answer, since a hostname can have multiple IP addresses.

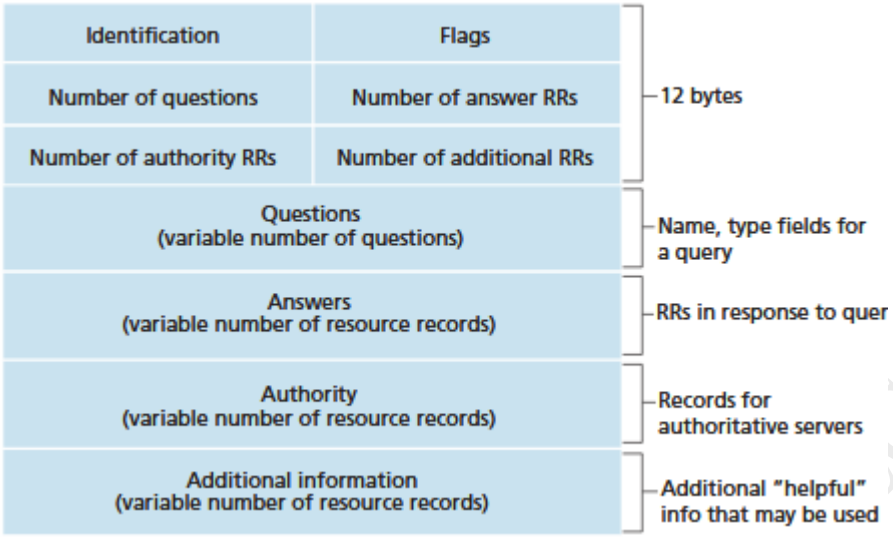

Fig 2.23 DNS Message format

• 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.

A new startup company called Network Utopia is setup.

The following steps are followed to setup DNS server :

- 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 and collects a small fee from company for its services. .
- During the registration we need to provide the registrar with the names and IP addresses of the primary and secondary authoritative DNS servers belonging to the company.
- Suppose the names and IP addresses are 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)

- Type A resource record for Web server www.networkutopia.com and the Type MX resource record for mail server mail.networkutopia.com are entered into authoritative DNS servers.
- Finally, people will be able to visit company Web site and send e-mail to the employees at the company.

Eg. Suppose Alice in Australia wants to view the Web page www.networkutopia.com .

- Alice's host will first send a DNS query to her local DNS server.
- The local DNS server will then contact a TLD com server.
- TLD server contains the Type NS and Type A resource records, because the registrar had these resource records inserted into all of the TLD com servers.
- The TLD com server sends a reply to Alice's local DNS server, with the reply containing the two resource records.
- The local DNS server then sends a DNS query to 212.212.212.1, asking for the Type A record corresponding to www.networkutopia.com.
- This record provides the IP address of the desired Web server, say, 212.212.71.4, which the local DNS server passes back to Alice's host.
- Alice's browser can now initiate a TCP connection to the host 212.212.71.4 and send an HTTP request over the connection.

2.6 Peer-to-Peer Applications

P2P architecture has 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.

2.6.1 P2P File Distribution

In client-server file distribution, the server must send a copy of the large 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.

The most popular P2P file distribution protocol is BitTorrent.

Scalability of P2P Architectures

Consider a simple model for distributing a file to a fixed set of peers for both architecture types. As shown in Figure, the server and the peers are connected to the Internet with access links.

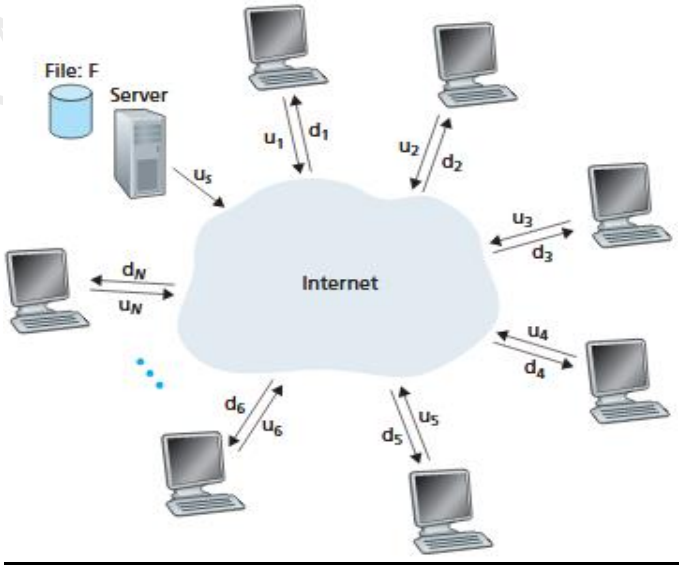

Fig 2.24 An illustrative file distribution problem

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*.

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*.

The **distribution time** is the time it takes to get a copy of the file to all *N* peers.

Distribution time for both client-server and P2P architectures is calculated.

Distribution time for the **client-server architecture** is denoted by *Dcs*.

In the client-server architecture, none of the peers aids in distributing the file.

• 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, that is, *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

$$D_{cs} \geq \max\left\{\frac{NF}{u_s}, \frac{F}{d_{min}}\right\}.$$

This provides a lower bound on the minimum distribution time for the client-server architecture.

let's take this lower bound provided above as the actual distribution time, that is,

$$D_{cs} = \max\left\{\frac{NF}{u_s}, \frac{F}{d_{min}}\right\}$$

For *N* large file, the client-server distribution time is given by *NF/u*s. Thus, the distribution time increases linearly with the number of peers *N*.

In P2P architecture, 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.

The distribution time depends on how each peer distributes portions of the file to the other peers.

• 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*.

• 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/dmin seconds. Thus the minimum distribution time is at least F/dmin.

• Finally, 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 + $u1$ + … + 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 $NF/(us + u1 + \ldots + uN)$.

Putting these three observations together, we obtain the minimum distribution time for P2P, denoted by D_{P2P}.

$$D_{P2P} \geq \max\left\{\frac{F}{u_s}, \frac{F}{d_{min}}, \frac{NF}{u_s + \sum_{i=1}^{N} u_i}\right\}$$

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

Take the lower bound provided by Equation as the actual minimum distribution time, that is,

$$D_{P2P} = \max\left\{\frac{F}{u_s}, \frac{F}{d_{min}}, \frac{NF}{u_s + \sum_{i=1}^{N} u_i}\right\}$$

Figure below compares the minimum distribution time for the client-server and P2P architectures assuming that all peers have the same upload rate u.

In Figure, F/u = 1 hour, us = 10u, 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 the peer download rates are set large enough so as not to have an effect.

For the client-server architecture, the distribution time increases linearly and without bound as the number of peers increases. 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. Thus, applications with the P2P architecture can be self-scaling.

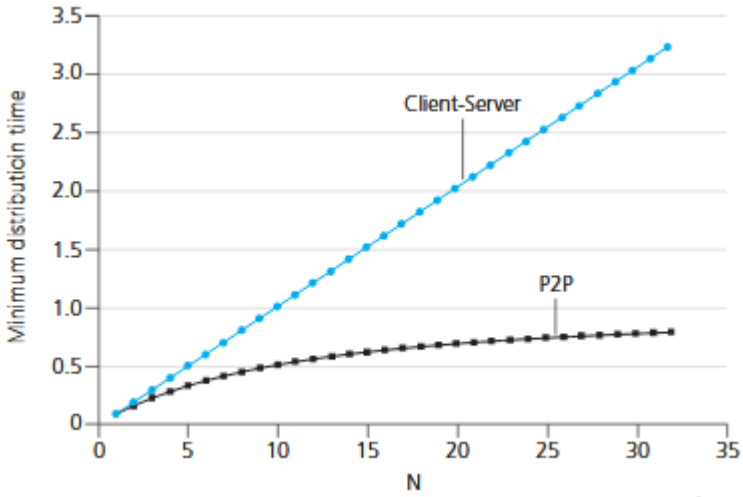

Fig 2.25 Distribution time for P2P and client-server architectures

BitTorrent

BitTorrent is a popular P2P protocol for file distribution.

In BitTorrent, 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 a peer downloads chunks it also uploads chunks to other peers. Once a peer has acquired the entire file, it may leave the torrent or remain in the torrent and continue to upload chunks to other peers.

Any peer may leave the torrent at any time with only a subset of chunks, and later rejoin the torrent.

Operation of BitTorrent:

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.

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, when a new peer, Alice, joins the torrent, the tracker randomly selects a subset of peers (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.

All the peers with which Alice succeeds in establishing a TCP connection "neighboring peers." In Figure, Alice is shown to have only three neighboring peers.

As time evolves, some of these peers may leave and other peers may attempt to establish TCP connections with Alice.

A peer's neighbouring 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 neighbours have.

With this information, Alice will have two important decisions to make:

First, which chunks should she request first from her neighbors?

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 neighbours and then request those rarest chunks first.

In this manner, the rarest chunks get more quickly redistributed.

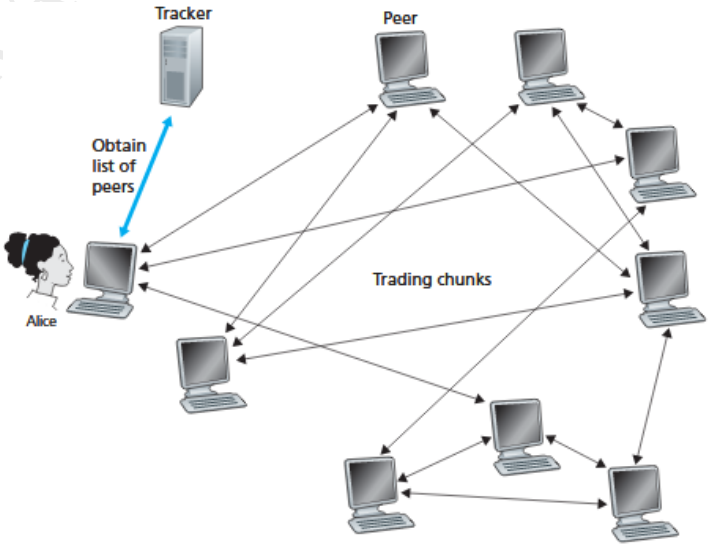

Fig2.26 File Distribution with Bit Torrent

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, these four peers are said to be unchoked.

Every 30 seconds, she also picks one additional neighbour at random and sends it chunks.

Let the randomly chosen peer be Bob. In BitTorrent, 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.

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.

The random neighbor selection also allows new peers to get chunks, so that they can have something to trade.

All other neighboring peers besides these five peers (four "top" peers and one probing peer) are "choked," that is, they do not receive any chunks from Alice.

2.6.2 Distributed Hash Tables (DHTs)

Consider the implementation of a simple centralized database in a P2P network, which will contain (key, value) pairs. For example, the keys could be voterid and the values could be the corresponding human names;

An example key-value pair is (156-45-7081, Johnny) 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; an example key-value pair is (Linux, 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 voterid and their corresponding human names, we can query with a specific voterid and the database returns the name of the human who has that voterid.

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.

Consider a distributed, P2P version of this database that stores the (key, value) pairs over millions of peers.

In the P2P system, each peer will only hold a small subset of the totality of the (key, value) pairs.

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).

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, IP of Bob) and (Linux, IP of Charlie).

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.

Suppose Alice wants to obtain a copy of Linux. 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.

Design of DHT :

Approach 1:

Randomly scatter the (key, value) pairs across all the peers and have each peer maintain a list of the IP addresses of all participating peers.

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, 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.

Approach 2 :

Assign an identifier to each peer, where each identifier is an integer in the range [0, 2*n*_ 1] for some fixed *n*.

Let each key to be an integer in the same range. To create integers out of alphabetic keys, we will use a hash function that maps each key (e.g., voterid) to an integer in the range [0, 2*n*_1]. A hash function is a many-to-one function for which two different inputs can have the same output (same integer), but the likelihood of the having the same output is extremely small.

The hash function is assumed to be available to all peers in the system. Henceforth, "key," refers to the hash of the original key.

For example, if the original key is "Linux," the key used in the DHT will be the integer that equals the hash of "Linux".

Storing (key,value) pair in 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.

Assign each (key, value) pair to the peer whose identifier is the *closest* to the key.

Closest peer(nearest peer) as the *closest successor of the key*.

Example. Suppose *n* _ 4 so that all the peer and key identifiers are in the range [0, 15]. 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) in one of the eight peers.

since peer 12 is the closest successor for key 11, we therefore store the pair

(11, Johnny Wu) in the peer 12.

If the key is exactly equal to one of the peer identifiers, the (key, value) pair is stored in that matching peer;

Circular DHT

The peers are recognized in the form of a circle. In this circular arrangement, each peer only keeps track of its immediate successor and immediate predecessor (modulo $2n$).

An example of such a circle is shown in Figure.

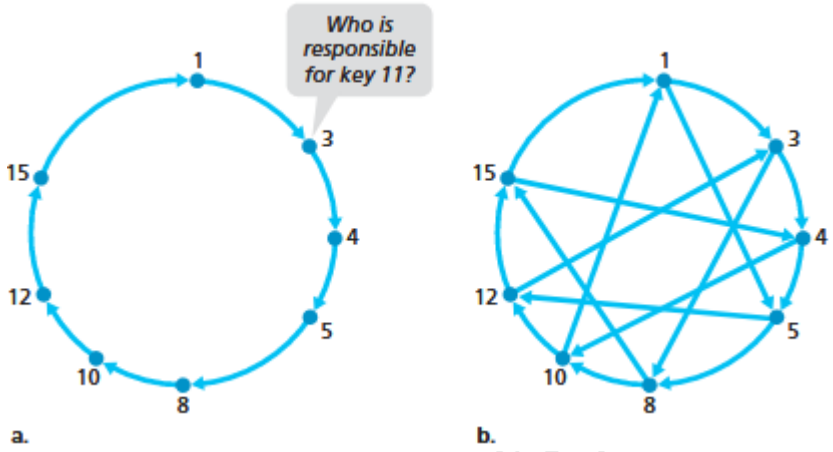

Fig 2.27 (a) Circular DHT. Peer 3 wants to determine who is responsible for key 11. (b) A circular DHT with shortcuts

In this example, n is 4 and there are 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 .

This circular arrangement of the peers is a special case of an overlay network.

The links in an overlay network are not physical links, but are simply virtual links.

In the overlay in Figure (a), there are eight peers and eight overlay links; in the overlay in Figure (b) there are eight peers and 16 overlay links.

A single overlay link typically uses many physical links and physical routers in the underlay network.

Using the circular overlay in Figure (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.

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.

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.

One such refinement is to use the circular overlay as a foundation, 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 (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 (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 significantly reduce the number of messages used to process a query.

Peer Churn

In P2P systems, a peer can come or go without warning.

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.

Let's now consider how the DHT is maintained when a peer abruptly leaves. For example, suppose peer 5 in Figure (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.

Peer 4 updates its state in the following way:

1. Peer 4 replaces its first successor (peer 5) with its second successor (peer 8).

2. 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.

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.

2.7 Socket Programming: Creating Network Applications

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.

2.7.1 Socket Programming with UDP

Client-server programs that use UDP; in the following section, processes running on different machines communicate with each other by sending messages into sockets.

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.

TCP is connection oriented and provides a reliable bytestream channel through which data flows between two end systems. UDP is connectionless and sends independent packets of data from one end system to the other, without any guarantees about delivery.

Processes running on different machines communicate with each other by sending messages into sockets.

The application resides on one side of the socket; the transport-layer protocol resides on

the other side of the socket.

Before the sending process can push a packet of data out the socket, 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.

The destination host's IP address is part of the destination address. 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. Hence, The packet's destination address also includes the socket's port number.

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.

The following simple client-server application demonstrates socket programming for both UDP and TCP:

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.

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

UDP client, sends a simple application-level message to the server. In order for the server to

be able to receive and reply to the client's message, it must be ready and running— that is, it must be running as a process before the client sends its message.

The client program is called UDPClient.py, and the server program is called UDPServer.py.

For this application, 12000 is arbitrarily chosen for the server port number.

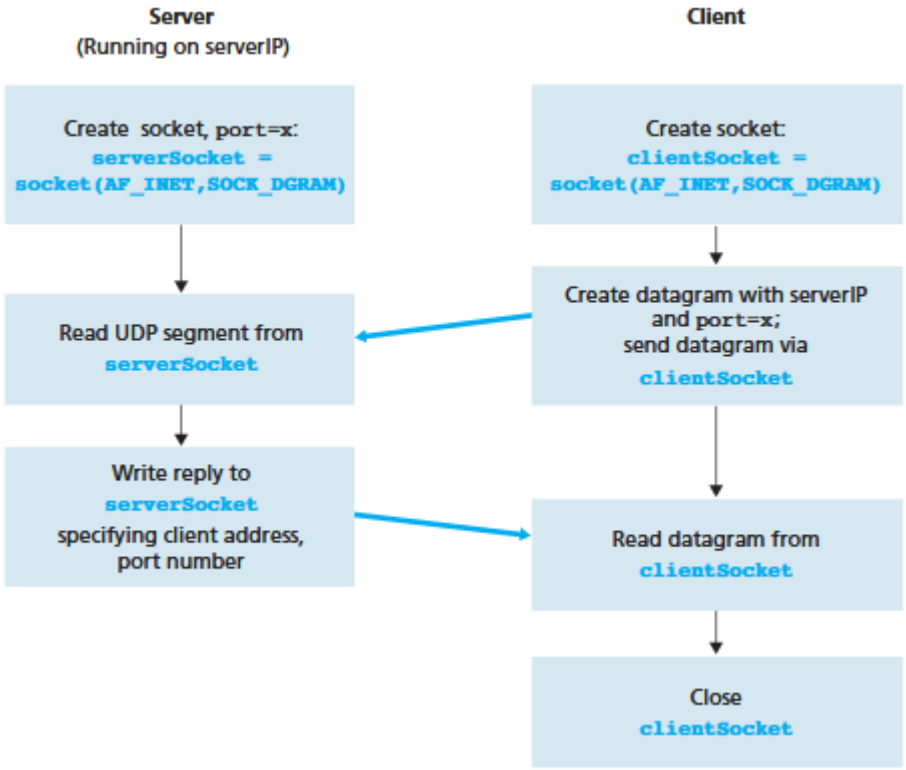

Fig 2.28 The client-server application using UDP

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 serverName to hostname. Here, we provide a string containing either the IP address of the server (e.g., "128.138.32.126") or the hostname 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 integer 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.

The second parameter indicates that the socket is of type SOCK_DGRAM, which means it is a UDP socket .

Port number of the client socket is specified by the operating system.

```
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.

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 program UDPClient doesn't actually need this server address information, since it already knows the server address from the outset; but this line of Python provides the server address.

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

The server side of the application is:

```
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)

It also imports the socket module, also sets the integer variable serverPort to 12000, and also creates a socket of type SOCK_DGRAM (a UDP socket).

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

serverSocket.bind((*''*, serverPort))

The above line 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, that packet will be directed to this socket.

UDPServer then enters a while loop; the while loop will allow UDPServer to receive and process packets from clients indefinitely. In the while loop, 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()

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 .

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.

In order for the server to be able to react to the client's initial contact, the server has to be ready.

This implies two things.

- First, as in the case of UDP, the TCP server must be running 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.

Client's initial contact is "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 programs.

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. In example below, the welcoming door is a TCP socket object that is called serverSocket; the newly created socket dedicated to the client making the connection is called connectionSocket.

The initial point of contact for all clients wanting to communicate with the server is server socket and each newly created server-side connection socket that is subsequently created for communicating with each client.

From the application's perspective, the client's socket and the server's connection socket are directly connected by a pipe. As shown in Figure below, the client process can send arbitrary bytes into its socket, and TCP guarantees that the server process will receive (through the connection socket) each byte in the order sent. TCP thus provides a reliable service between the client and server processes.

The client process not only sends bytes into but also receives bytes from its socket; similarly, the server process not only receives bytes from but also sends bytes into its connection socket.

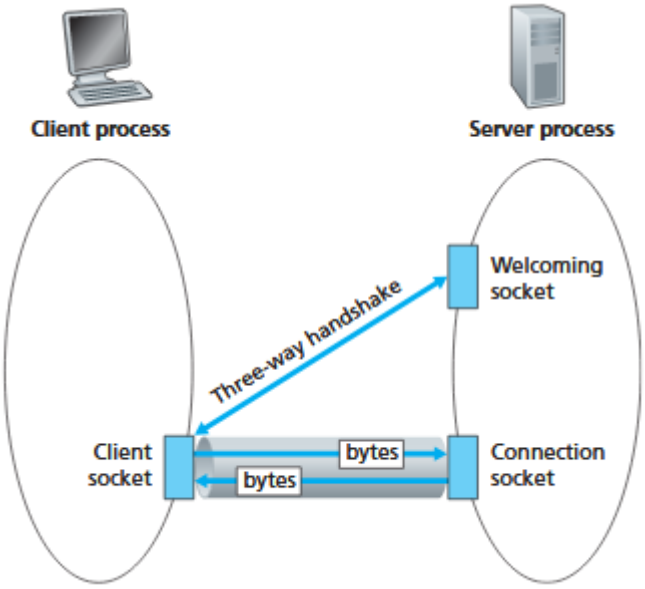

Fig 2.29 The TCPServer process has two sockets

The client sends one line of data to the server, the server capitalizes the line and sends it back to the client. Figure highlights the main socket-related activity of the client and server that communicate over the TCP transport service.

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()
```

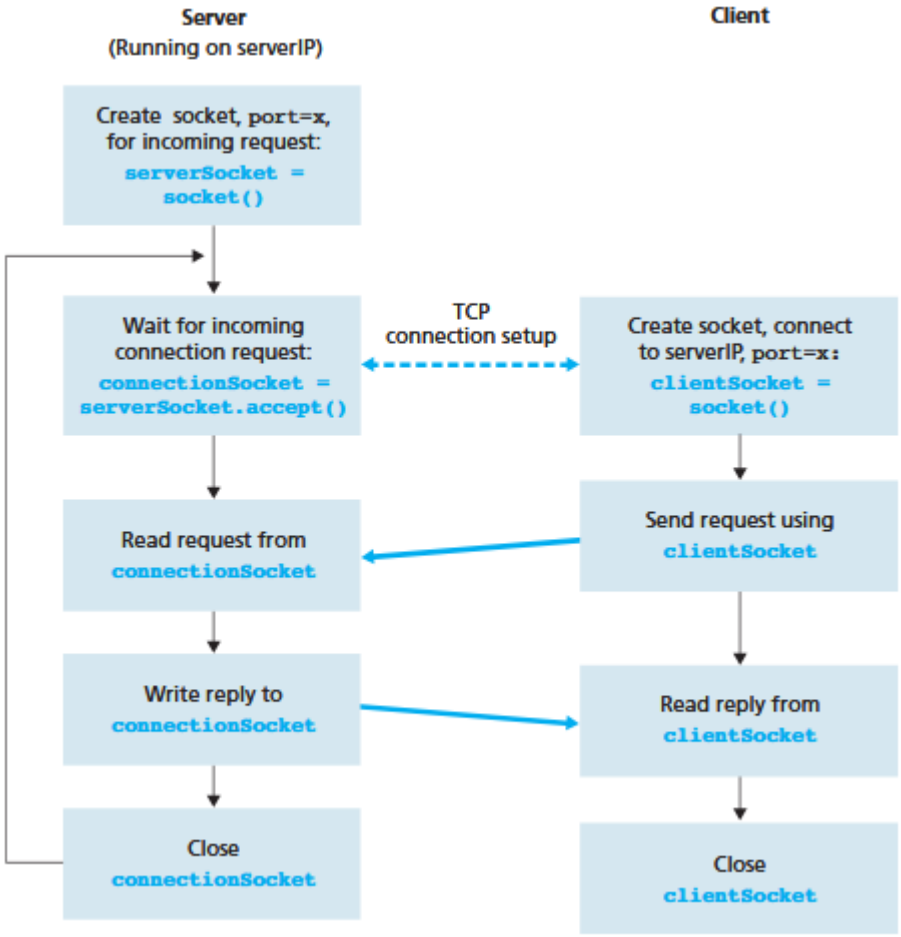

Fig 2.30 The Client-server Application using TCP

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).

UDPClient:

clientSocket.connect((serverName,serverPort))

Before the 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 connection.

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:’)

As with UDPClient, the above obtains a sentence from the user. 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)

The above line sends the string sentence through the client’s socket and into the TCP connection.

The client program simply drops the bytes in the string sentence into the TCP connection.

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. It causes TCP in the client to send a TCP message to TCP in 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 TCPClient.

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 establishing this welcoming door, we will wait and listen for some client to knock on the door:

serverSocket.listen(1)

This line has 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 connectionSocket, 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.

Question Bank

1. List five nonproprietary Internet applications and the application-layer protocols that they use.

Ans :

1. The Web: HTTP
2. Remote login: Telnet
3. Network News: NNTP
4. e-mail: SMTP.
5. File transfer: FTP

2. What is the difference between network architecture and application architecture?

Ans : Network arch : 5 layered internet arch(TCP/IP) or 7 layered OSI.- Fixed for all applications and cant be changed by developer.

Application arch : Client – server , Peer to peer (designed by developer specific for applications)

3. For a communication session between a pair of processes, which process is the client and which is the server?(pg.no 6)

4. For a P2P file-sharing application, do you agree with the statement, "There is no notion of client and server sides of a communication session"? Why or why not? (pg.no 4 & 6)

5. What information is used by a process running on one host to identify a process running on another host? (P.no 6&7)

6. Suppose you wanted to do a transaction from a remote client to a server as fast as possible. Would you use UDP or TCP? Why? (p.no 11 & 12)

Ans : UDP

7. List the four broad classes of services that a transport protocol can provide. For each of the service classes, indicate if either UDP or TCP (or both) provides such a service.(P.no 8-12)

8. What is meant by a handshaking protocol?

9. Why do HTTP, FTP, SMTP, and POP3 run on top of TCP rather than on UDP?

10. Consider an e-commerce site that wants to keep a purchase record for each of its customers. Describe how this can be done with cookies.

11. What is meant by a handshaking protocol?(P.No 17,18)

12. Why do HTTP, FTP, SMTP, and POP3 run on top of TCP rather than on UDP?(P.no 15)

13. Describe how Web caching can reduce the delay in receiving a requested object. Will Web caching reduce the delay for all objects requested by a user or for only some of the objects? Why?(p.no 24-26) Refer problem in p.no 27

Ans:Only 40% requests may be satisfied .

14. Why is it said that FTP sends control information "out-of-band"? (P.no 30,31)

15. Suppose Alice, with a Web-based e-mail account (such as Hotmail or gmail), sends a message to Bob, who accesses his mail from his mail server using POP3. Discuss how the message gets from Alice's host to Bob's host. Be sure to list the series of application-layer protocols that are used to move the message between the two hosts.(P.no 37,38,39)

16. From a user's perspective, what is the difference between the download-and delete

mode and the download-and-keep mode in POP3?(p.no 38,39)

17. Is it possible for an organization's Web server and mail server to have exactly the same alias for a hostname (for example, foo.com)? What would be the type for the RR that contains the hostname of the mail server?(p.no 42 & 47) – Ans :MX record

18. In BitTorrent, suppose Alice provides chunks to Bob throughout a 30-second interval. Will Bob necessarily return the favor and provide chunks to Alice in this same interval? Why or why not?(p.no 55) – Ans : not necessary

19. Consider a new peer Alice that joins BitTorrent without possessing any chunks. Without any chunks, she cannot become a top-four uploader for any of the other peers, since she has nothing to upload. How then will Alice get her first chunk?(p.No 54,55)

20. What is an overlay network? Does it include routers? What are the edges in the overlay network?(p.no 58)

21. Consider a DHT with a mesh overlay topology (that is, every peer tracks all peers in the system). What are the advantages and disadvantages of such a design? What are the advantages and disadvantages of a circular DHT (with no shortcuts)?(p.no 58,59)

22. List at least four different applications that are naturally suitable for P2P architectures. (*Hint:* File distribution and instant messaging are two.)

Ans : Bit torrent , skype, emule, ares galaxy ,kaaza

23. UDP server described needed only one socket, whereas the TCP server needed two sockets. Why? If the TCP server were to support *n* simultaneous connections, each from a different client host, how many sockets would the TCP server need?

Ans: No Handshaking in UDP. TCP server needs (n+1) sockets .

24. For the client-server application over TCP, why must the server program be executed before the client program? For the client server application over UDP, why may the client program be executed before the server program?

Ans : Connection socket is required for each client. Since , TCP server creates it, server program should be executed before client.

25. What is the difference between MAIL FROM: in SMTP and From: in the mail message itself? (p.no 36)

26. How does SMTP mark the end of a message body? How about HTTP? Can HTTP use the same method as SMTP to mark the end of a message body? Explain.(p.no 35,36)

Problems :

1. Consider an HTTP client that wants to retrieve a Web document at a given URL. The IP address of the HTTP server is initially unknown. What transport and application-layer protocols besides HTTP are needed in this scenario? (DNS,TCP)

2. Consider accessing your e-mail with POP3.(p.no 38,39)

a. Suppose you have configured your POP mail client to operate in the download-and-delete mode. Complete the following transaction:

C: list

S: 1 498

S: 2 912

S: .

C: retr 1

S: blah blah ...

S: .....blah

S: .

?

?

b. Suppose you have configured your POP mail client to operate in the download-and-keep mode. Complete the following transaction. (p.no 38,39) – without dele command.

C: list

S: 1 498

S: 2 912

S: .

C: retr 1

S: blah blah ...

S: .....blah

S: .

?

?

3. Suppose you can access the caches in the local DNS servers of your department. Can you propose a way to roughly determine the Web servers (outside your department) that are most popular among the users in your department? Explain.

4. Suppose that your department has a local DNS server for all computers in the department. You are an ordinary user (i.e., not a network/system administrator). Can you determine if an external Web site was likely accessed from a computer in your department a couple of seconds ago? Explain.(p.no 49,50)

5. Consider distributing a file of $F = 15$ Gbits to N peers. The server has an upload rate of $us = 30$ Mbps, and each peer has a download rate of $di = 2$ Mbps and an upload rate of u. For N

= 10, 100, and 1,000 and u = 300 Kbps, 700 Kbps, and 2 Mbps, prepare a chart giving the minimum distribution time for each of the combinations of N and u for both client-server distribution and P2P distribution.(p.no 52,53)

P27. In the circular DHT example in Section 2.6.2, suppose that peer 3 learns that peer 5 has left. How does peer 3 update its successor state information? Which peer is now its first successor? Its second successor? In the circular DHT example in Section 2.6.2, suppose that a new peer 6 wants to join the DHT and peer 6 initially only knows peer 15's IP address.

What steps are taken?(p.no 59)