# UNIT-II

**2. Principles of Network Applications:** At the core of network application development is writing programs that run on different end systems and communicate with each other over the network. In the Web application there are two distinct programs that communicate with each other: the browser program running in the user's host; and the Web server program running in the Web server host. When developing your new application, you need to write software that will run on multiple machines.

**2.1.1 Network Application Architectures**: The application architecture is designed by the application developer and dictates how the application is organized over the various end systems. In choosing the application architecture, an application developer will likely draw on one of the three predominant architectures used in modern network applications:

- the client-server architecture,
- the P2P architecture, and
- a hybrid of the client-server and P2P architectures.

**Client-server architecture**: In client-server architecture, there is an always-on host, called the server, which services requests from any other hosts, called clients. The client hosts can be either sometimes-on or always on. With the client-server architecture, clients do not directly communicate with each other. Another characteristic of the client server architecture is that the server has a fixed, well-known address, called an IP address. Because the server has a fixed, well-known address, and because the server is always on, a client can always contact the server by sending a packet to the server's address. Often in a client-server application, a single server host is incapable of keeping up with all the requests from its clients. For this reason, clusters of hosts--sometimes referred to as a server farm--are often used to create a powerful virtual server in client-server architectures.

**P2P architecture**: In a pure P2P architecture, there isn't an always-on server at the Centre of the application. Instead, arbitrary pairs of hosts, called peers, communicate directly with each other. Because the peers communicate without passing through some special server, the architecture is called peer-to-peer. One of the greatest strengths of the P2P architecture is it scalability. While each peer will generate workload by requesting files, each peer also adds service capacity to the system by responding to the requests of other peers. P2P file sharing is intrinsically scalable--each additional peer not only increases demand but also increases service capacity.

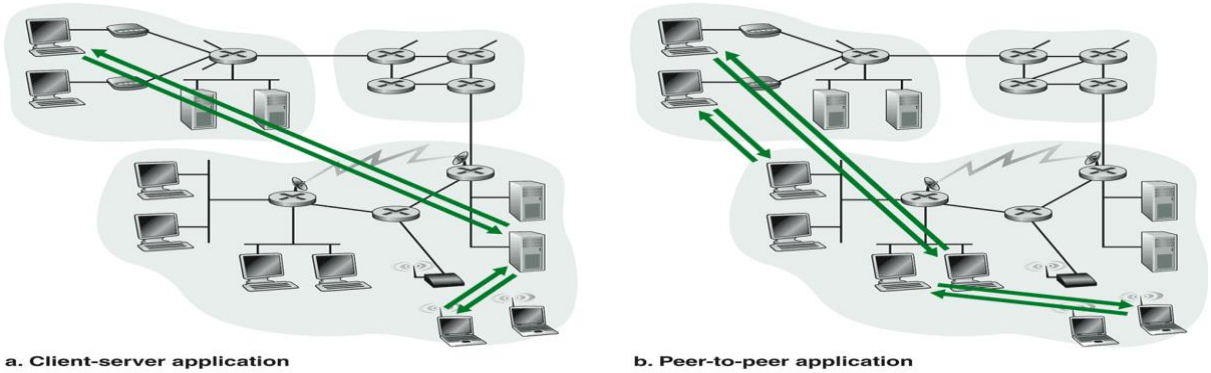



**Figure 2.2** ◆ (a) Client-server architecture; (b) P2P architecture.

**Hybrid architecture**: Many applications are organized as hybrids of the client-server and P2P architectures.

**2.1.2 Processes Communicating**: In the terminology of operating systems, it is not actually programs but processes that are communicating. A process can be thought of as a program that is running within an end 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. 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 process, the two processes were labelled one as the client and the other as the server. With the Web, a browser is a client process and a Web server is a server process. With P2P file sharing, the peer that is downloading the file is labelled as the client, and the peer that is uploading the file is labelled as the server.
- **Sockets**: Any message sent from one process to another must go through the underlying network. A process sends messages into, and receives messages from, the network through its socket. When a process wants to send a message to another process on another host, it shoves the message out its door (socket) and into the network. This sending process assumes that there is a transportation infrastructure on the other side of its door that will transport the message across the network to the door of the destination process. Once the message arrives at the destination host, the message passes, through the receiving process's door (socket), and the receiving process than acts on the message. 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 in the Internet.
- **Addressing Processes**: In order for a process on one host to send a message to a process on another host, the sending process must identify the receiving process. To identify the receiving process, two pieces of information need to be specified: (1) the name or address of the host and (2) an identifier that specifies the receiving process in the destination host. In internet applications, the destination host is identified by it IP address Since the IP address of any host connected to the public Internet must be globally unique, the assignment of IP addresses must be carefully managed. In addition to knowing the address of the host to which a message is destined, the sending host must also identify the receiving process running in the host. This information is needed because in general a host could be running many network applications. A destination port number serves this purpose. Popular applications have been assigned specific port numbers.

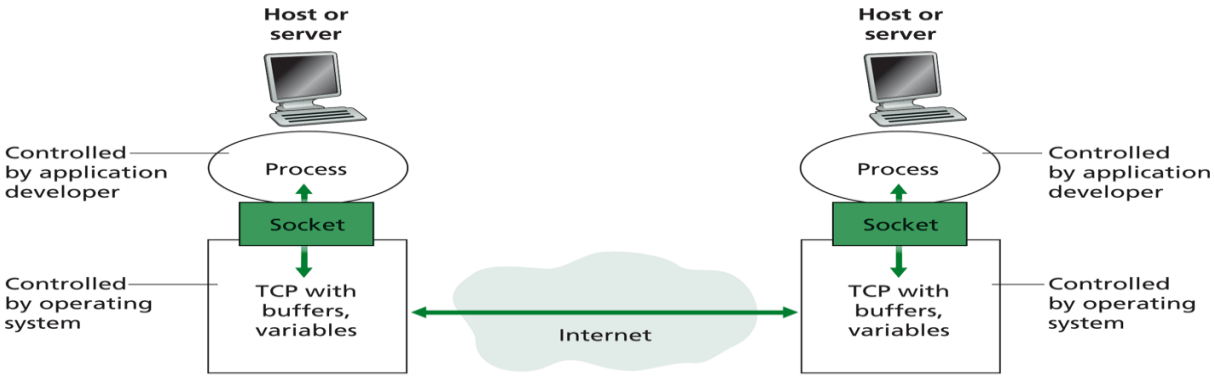



**Figure 2.3** ◆ Application processes, sockets, and underlying transport protocol

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

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

**2.2 The Web and HTTP:** The Web is the Internet application that caught the general public's eye. It dramatically changed how people interact inside and outside their work environments. It elevated the Internet from just one of many data networks to essentially the one and only data network. The Internet is yet another electronic communication technology with a major societal impact. E-mail and the Web, two of its most popular applications, have revolutionized how we live and work. In addition to being available on demand, the Web has many other wonderful features that people live and cherish.

**2.2.1 Overview of HTTP:**

- The **Hypertext Transfer Protocol** (HTTP), the Web's application-layer protocol is at the heart of the Web. HTTP is implemented in two programs: a client program and a server program. The client program and server program, executing on different end systems, talk to each other by exchanging HTTP messages. HTTP defines the structure of these messages and how the client and server exchange the message.
- 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 GIF image, a Java applet, an audio clip, and so on--that is addressable by a single URL. Most Web pages consist of a base HTML file and several referenced objects.
- A **browser** is a user agent for the Web; it displays the requested Web page to the user and provides numerous navigational and configuration features. Because Web browsers also implement the client side of HTTP, in the context of the Web, we will use the words browser and client interchangeably.

- A **Web server** houses Web objects, each addressable by a URL. Web servers also implement the server side of HTTP.
- HTTP defines how Web clients request Web pages from Web servers and how servers transfer Web pages to clients.
- 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.

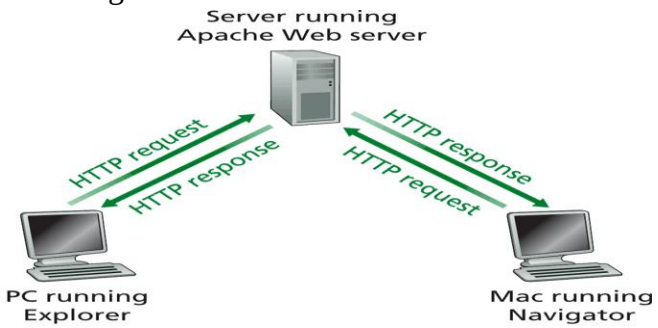



**Figure 2.6** ◆ HTTP request-response behavior

**2.2.2 Nonpersistent and Persistent Connections:** HTTP can use both nonpersistent connections and persistent connections. Although HTTP uses persistent connections in its default node, HTTP clients and servers can also be configured to use non-persistent connections instead.

**Nonpersistent Connections:** As the browser receives the Web page, it displays the page to the user. Two different browsers may interpret a Web page in somewhat different ways. HTTP has nothing to do with how a Web page is interpreted by a client. The HTTP specifications define only the communication protocol between the client HTTP program and the server HTTP program.

The round-trip time (RTT), which is the time it takes for a small packet to travel from one 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.

**Persistent Connections:** Nonpersistent connections have some shortcomings. First, a brand-new connection must be established and maintained for each requested object. For each of these connections, TCP buffers must be allocated and TCP variables must be kept in both the client and server. This can place a serious burden on the Web server, which may be serving request from hundreds of different clients simultaneously. Second, 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. Moreover, multiple Web pages residing on the same server can be sent from the server to the same client over a single persistent TCP connection**.**

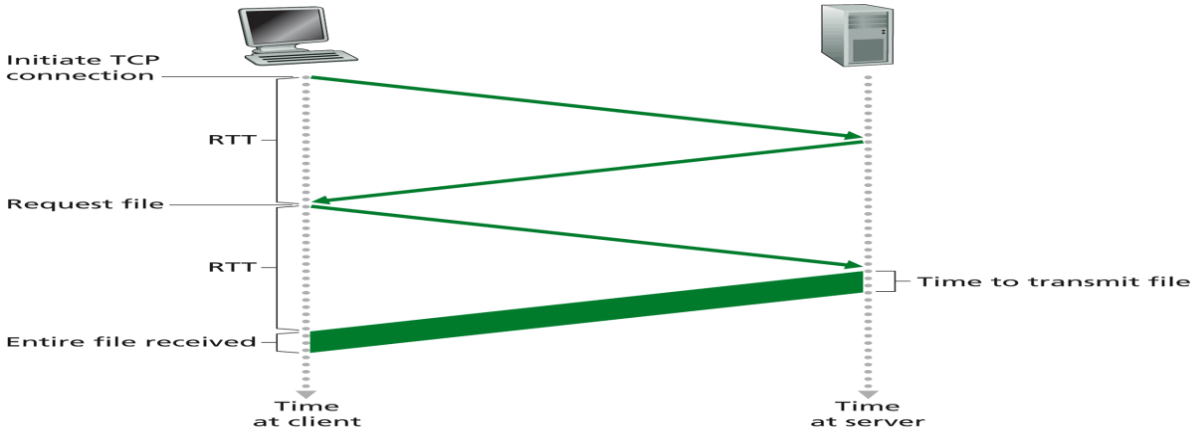



**Figure 2.7** ◆ Back-of-the-envelope calculation for requesting an HTML file

There are two versions of persistent connections: **without pipelining** and **with pipelining**. For the version without pipelining, the client issues a new request only when the previous response has been received. In this case, the client experiences one RTT in order to request and receive each of the referred objects. Although this is an improvement over nonpersistent's two RTT's per object, the RTT delay can be further reduced with pipelining. Another disadvantage of no pipelining is that after the server sends an object over the persistent TCP connection, the connection idles--does nothing--while it waits for another request to arrive. This idling wastes server resources.

The default mode of HTTP uses persistent connections with pipelining. With pipelining, the HTTP client issues a request as soon as it encounters a reference. Thus the HTTP client can make back-to-back requests for the referred objects; that is, it can make a new request before receiving a response to a previous request. When the server receives the back-to-back requests, it sends the objects back-to-back, with pipelining; it is possible for only one RTT to be expended for all the referenced objects. Furthermore, the pipelined TCP connection remains idle for a smaller fraction of time.

**2.2.3 HTTP Message Format:** The HTTP specifications include the definitions of the HTTP message formats. There are two types of HTTP messages, request messages and response messages.

**HTTP Request Message:**

- 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 fields, and the HTTP version field. The method field can take on several different values, including GET, POST, and HEAD. 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.
- An HTTP client often uses the POST method when the user fills out a form. With a POST message, the user is still requesting a Web page from the server, but the specific contents of the Web page depends 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.

- A request generated with a form does not necessarily use the POST method. Instead, HTML forms often use the GET method and include the inputted data in the requested URL. If a form uses the GET method, have two 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 to the requested object.

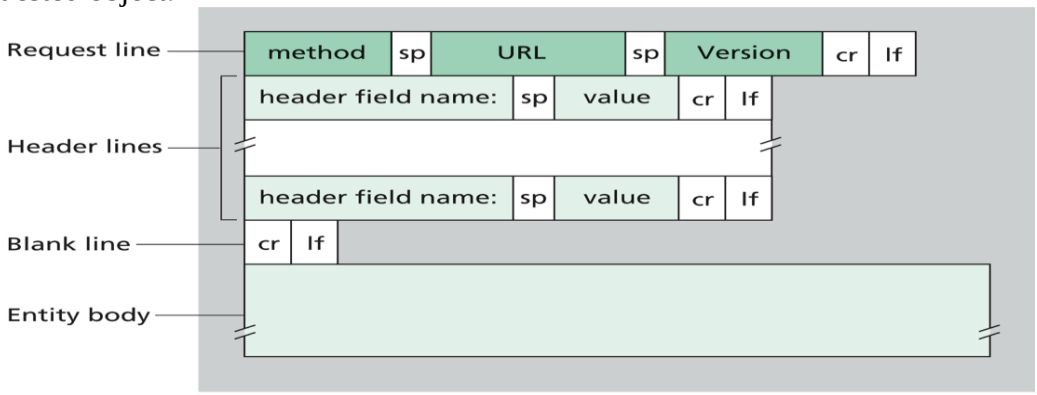



**Figure 2.8** ◆ General format of a request message

**HTTP Response Message**

- Response message has three sections: an initial status line, six header lines, and then the entity body. The entity body is the meat of the message--it contains the requested object itself. The status line has three fields: the protocol version field, a status code, and a corresponding status message.
- The server uses the Connection: close header line to tell the client header line to tell the client that it 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. Note that this is not the time when the object was created or modified; it is the time when the server retrieves the object from its file system, inserts the object into the response message, and sends the response message. The Server: header line indicates that the message was generated by Apache Web server; it is analogous to the User-agent: header line in the HTTP request message. The Last-Modified (also known as proxy servers): header line indicated 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 indicates that the object in the entity body is HTML text.

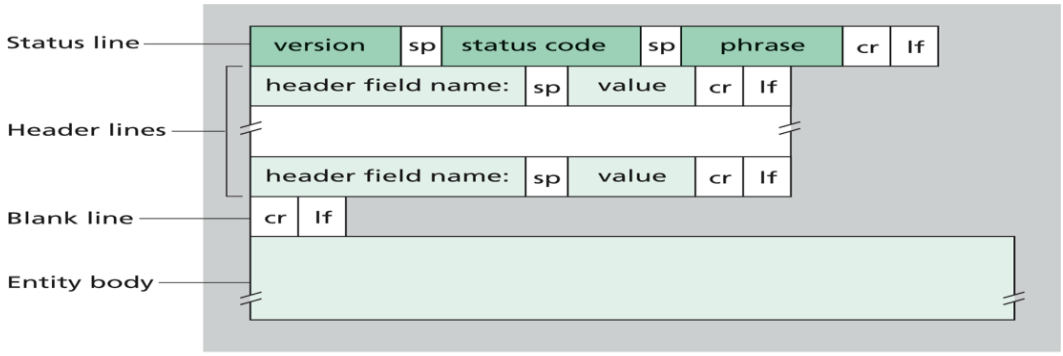



**Figure 2.9** ◆ General format of a response message

### 2.2.4 User-Server Interaction: Cookies

- Cookies allow sites to keep track of users. Although not all sites use cookies, most major portal, e-commerce, and advertising sites make extensive use of cookies.
- 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.
- Although cookies often simplify the Internet shopping experience for the user, they remain highly controversial because they can also be considered as an infringement on a user's privacy.

### 2.2.5 Web Caching:

- A Web-cache--also called a proxy server--is a network entity that satisfies HTTP requests on the behalf of an origin Web server. The Web cache has its own disk storage and keeps copies of recently requested objects in this storage. 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**.**
- 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 is sends requests to and receives responses from an origin server, it is a client**.**

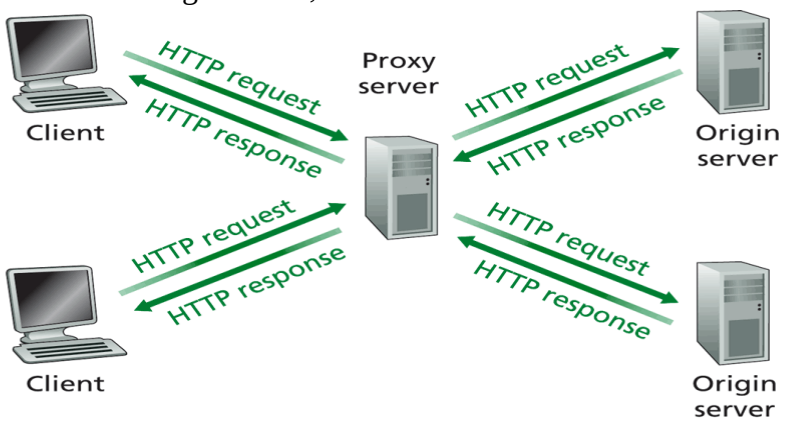



**Figure 2.10** ◆ Clients requesting objects through a Web cache

Web caching has seen development in the Internet for two reasons:

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. By reducing traffic, the institution does not have to upgrade bandwidth as quickly, thereby reducing costs. A Web caches can substantially reduce Web traffic in the Internet as a whole, thereby improving performance for all applications**.**

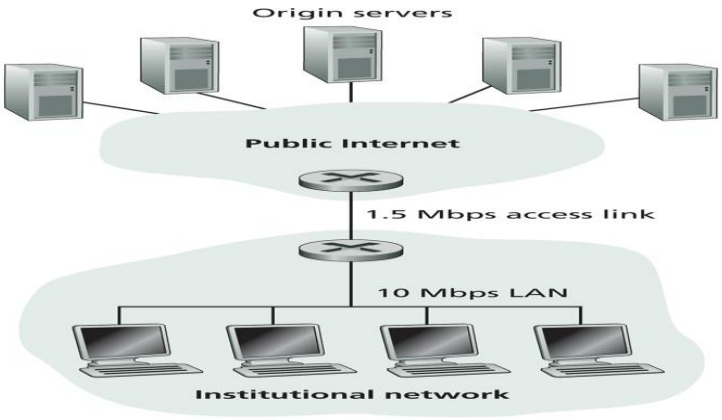



**Figure 2.11** ◆ Bottleneck between an institutional network and the Internet

### 2.2.6 The Conditional GET:

- Although caching can reduce user-perceived response times, it introduces a new problem--the copy of an object residing in the cache may be stale. The object housed 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 so-called conditional GET message if (1) the request message uses the GET method and (2) the request message includes an If-Modified-Since: header line.
- The cache forwards the object to the requesting browser but also caches the object locally. The cache also stores the last-modified date along with the object. Third, one week later, another browser request 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.

**2.3 File Transfer: FTP**: In a typically FTP session, the user is sitting in front of one 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 user identification and a password. After providing this authorization, the user can't transfer files from the local file system to the remote file system and vice versa. 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 get 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.

HTTP and FTP are both files transfer protocols and have many common characteristics; for example, they both run on top of TCP.

The two application-layer protocols have some important differences. FTP uses two parallel TCP connections to transfer a file, a **control connection** and a **data connection**.

- The control connection is used for sending control information between the two hosts. 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**. When a user starts an FTP session with a remote host, the client side of FTP first initiates a control TCP connection with the server side 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 sends also sends, over the side receives over the control connection a command for a file transfer, the server side initiates a TCP data connection to the data connection.

- Throughout a session, the FTP server must maintain state about the user. The server must associate the control connection with a specific user account and the server must keep track of the user's current directory as the user wanders about the remote directory tree. Keeping track of this state information for each on going user session significantly constrains the total number of sessions that FTP can maintain simultaneously

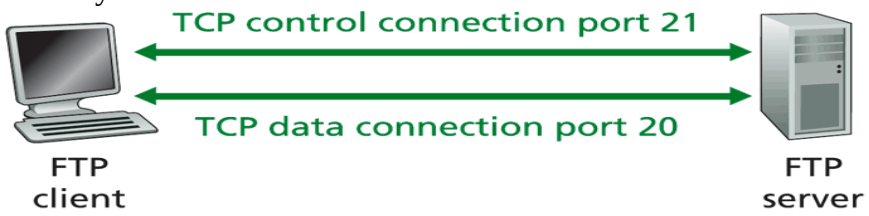



**Figure 2.14** ♦ Control and data connections

**2.3.1 FTP Commands and Replies:** The commands, from client to server, and replies, from server to client, are sent across the control connection is 7-bit ASCII format. Like HTTP commands, FTP commands are readable by people. In order to delineate successive commands, a carriage return and line feed end each command. Each command consists of four uppercase ASCII characters, some with optional arguments.

Some of the most common commands are:

- 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 user to send back a list of all the files in the current remote directory.
- RETR filename: Used to retrieve a file from the current directory of the remote host.
- STOR filename: Used to store a file into the current directory of the remote host.

There is typically a one-to-one correspondence between the command that the user issue and the FTP command sent across the control connection. Each command is followed by a reply, sent from server to client. The replies are three-digit numbers, with optional message following the number. This is similar in structure to the status code and phrase in the status line of the HTTP response message; the inventors of HTTP intentionally included this similarity in the HTTP response message.

**2.4 Electronic Mail in the Internet:**

Electronic mail has been around since the beginning of the Internet. It was the most popular application when the Internet was in its infancy, it has become more and more elaborate and powerful over the years, and it continues to evolve. It is one of the Internet's most important killer applications to date.

There are three major components: user agents, mail servers, and the Simple Mail Transfer Protocol (SMTP).

- Mail servers form the core of the e-mail infrastructure. Each recipient has a mailbox located in one of the mail servers.
- Servers hold the message in the message queue and attempt to transfer the message later. Reattempts are often done every 30 minutes or so; if there is no success after several days, the server removers the message and notifies the sender with an e-mail message.
- 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. As with most application-layer protocols, SMTP has two sides: a client side, which executes on the sender's mail server, and a server side, which executes on the recipient's mail server. Both the client and server sides of SMTP run on every mail server. When a mail server receives mail from the other mail servers it acts as an SMTP server.

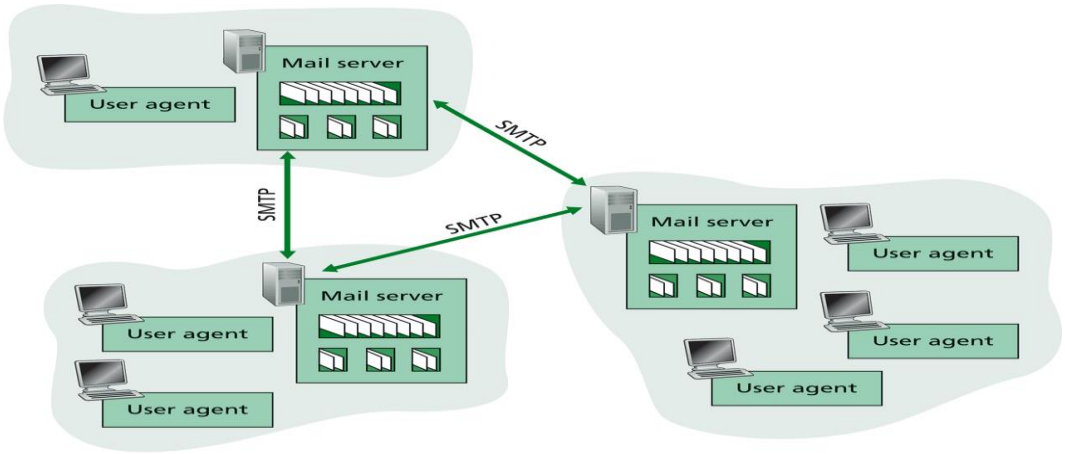



**Figure 2.15** ◆ A high-level view of the Internet e-mail system

### 2.4.1 SMTP:

- SMTP is at the heart of Internet electronic mail. SMTP transfers messages from senders' mail servers to the recipients' mail servers. SMTP is much older than HTTP. Although SMTP has numerous wonderful qualities it is nevertheless a legacy technology that possesses certain archaic characteristics.
- In the multimedia era, the 7-bit ASCII restriction is a bit of a pain it requires binary multimedia data to be enclosed to ASCII before being sent over SMTP; and it requires the corresponding ASCII message to be decoded back to binary after SMTP transport.
- It is important to observe that SMTP does not normally use intermediate mail servers for sending mail, even when the two mail servers are located at opposite ends of the world

### 2.5 DNS--The Internet's Directory Service:

- Just as humans can be identified in many ways, so too can Internet hosts. One identifier for a host is its hostname. Hostnames are mnemonic and are therefore appreciated by humans. However, hostnames provide little, if any, information about the location within the Internet of host.

- Because hostnames can consist of variable-length alphanumeric characters, they would be difficult to process by routers. For these reasons, hosts are also identified by so-called IP addresses.
- An IP address consists of four bytes and has a rigid hierarchical structure. An IP address looks like 121.7.106.83, where each period separates on to the bytes expressed in decimal notation from 0 to 255. An IP address is hierarchical because as we scan the address from left to right, we obtain more and more specific information about where the host is located in the Internet.

**2.5.1 Services Provided by DNS**

There are two ways to identify a host--by a hostname and by an IP address. People prefer the more mnemonic hostname identifier, while routers prefer fixed-length, hierarchically structured IP addresses. In order to reconcile these preferences, we need a directory service that translates hostnames to IP addresses. This is the main task of the Internet's domain name system (DNS). The DNS is (1) a distributed database implemented in a hierarchy of DNS servers and (2) an application-layer protocol that allows hosts to query the distributed database. The DNS servers are often UNIX machines running the Berkeley Internet Name Domain (BIND) software. The DNS protocol runs over UDP and uses port 53.

DNS is commonly employed by other application-layer protocols--including HTTP, SMTP, and FTP--to translate user-supplied host names to IP addresses. IN order for the user's host to be able to send an HTTP request message to the Web server the user's host must first obtain the IP address.

- **Host aliasing**: A host with a complicated hostname can have one or more alias names.
- **Mail server aliasing**: For obvious reasons, it is highly desirable that e-mail addresses be mnemonic.
- **Load distribution**: DNS is also used to perform load distribution among replicated servers, such as replicated Web servers.

The DNS is dated in several additional RFC's. It is a complex system, and we only touch upon key aspects of its operation here

**2.5.2 Overview of How DNS Works:**

The hostname that needs to be translated is the function call that an application call in order to perform the translation. DNS in the user's host then takes over, sending 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.

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

- **A single point of failure**: If the DNS server crashes, so does the entire Internet!
- **Traffic volume**: A single DNS server would have to handle all DNS queries.

- **Distant Centralized database**: A single DNS server can't be "close to" all the querying clients.
- **Maintenance**: The single DNS server would have to keep records for all Internet hosts.

In summary, a centralized database in a single DNS server simply doesn't scale.

**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 mapping for all of the host in the Internet. Instead, the mappings are distributed across--root DNS servers. To first approximate, there are three classes of DNS servers--root DNS servers, top-level domain (TLD) DNS a servers, and authoritative DNS servers--organized in a hierarchy. To understand how these three classes of servers interact, suppose a DNS client wants to determine the IP address for the hostname the following events will take place. The client first contacts one of the roots 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. Finally, the client contacts on of the authoritative servers for which returns the IP address for the hostname. The three classes of DNS servers are:

- Root DNS server. In the Internet there are 13 root DNS servers, most of which are located in North America.
- Top-Level Domain (TLD) servers. These servers are responsible for top-level domains such as com, org, net edu, and gov, and all of the country top-level domains such as uk, fr, ca, and jp.
- Authoritative DNS servers. Every organization with publically accessible hosts on the Internet must porvide publicly accessible DNS records that map the names of those hosts to IP addresses

The root, TLD, and authoritative DNS servers all belong to the hierarchy of DNS servers. There is another important type of DNS, called the local DNS server. A local DNS server does not strictly belong to the hierarchy of servers but is nevertheless central to the DNS architecture. Each 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. You can easily determine the IP address of your local DNS server by accessing network status windows in Windows or UNIX. A host's local DNS server is typically "close to" the host. For an institutional ISP, the local DNS server may be on the same LAN as the host; for a residential ISP, it is typically separated from the host by no more than a few routers. When a host makes a DNS query, the query is sent to the local DNS server, which acts a proxy, forwarding the query into the DNS server hierarchy.

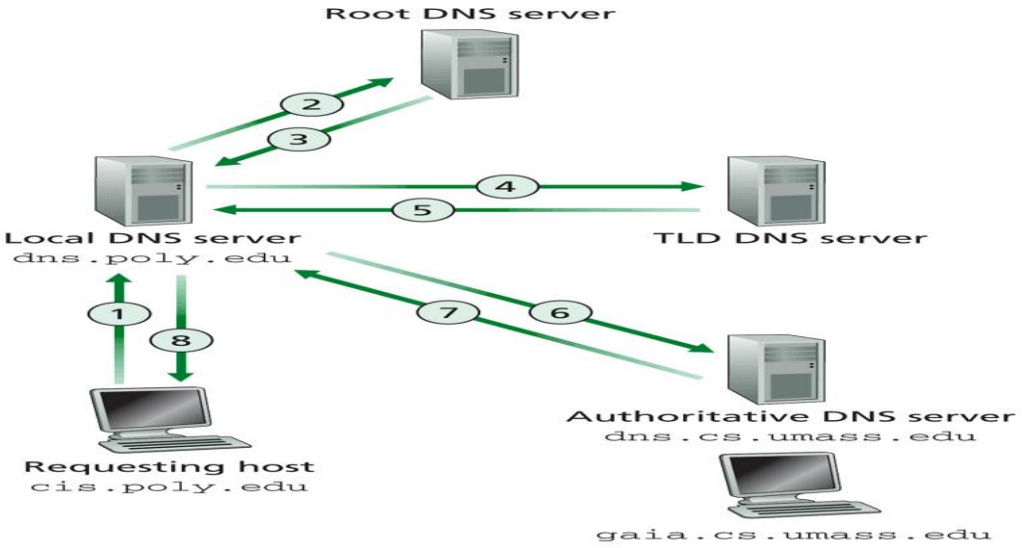



**Figure 2.20** ◆ Interaction of the various DNS servers

**DNS Caching:** DNS caching, a critically important feature of the DNS system. In truth, DNS extensively exploits DNS caching in order to improve the delay performance and to reduce the number of DNS messages ricocheting around the Internet. The idea behind DNS caching is very simple. In a query chain, when a DNS server receives a DNS reply it can cache the information in the reply in its local memory.

If a hostname/IP address pair is cached in a DNS server and another query arrives to the DNS server for the same hostname, the DNS server can provide the desired IP address, even if it is not authoritative for the hostname. Because hosts and mappings between hostnames and IP addresses are by no means permanent, DNS servers discard cached information after a period of time.

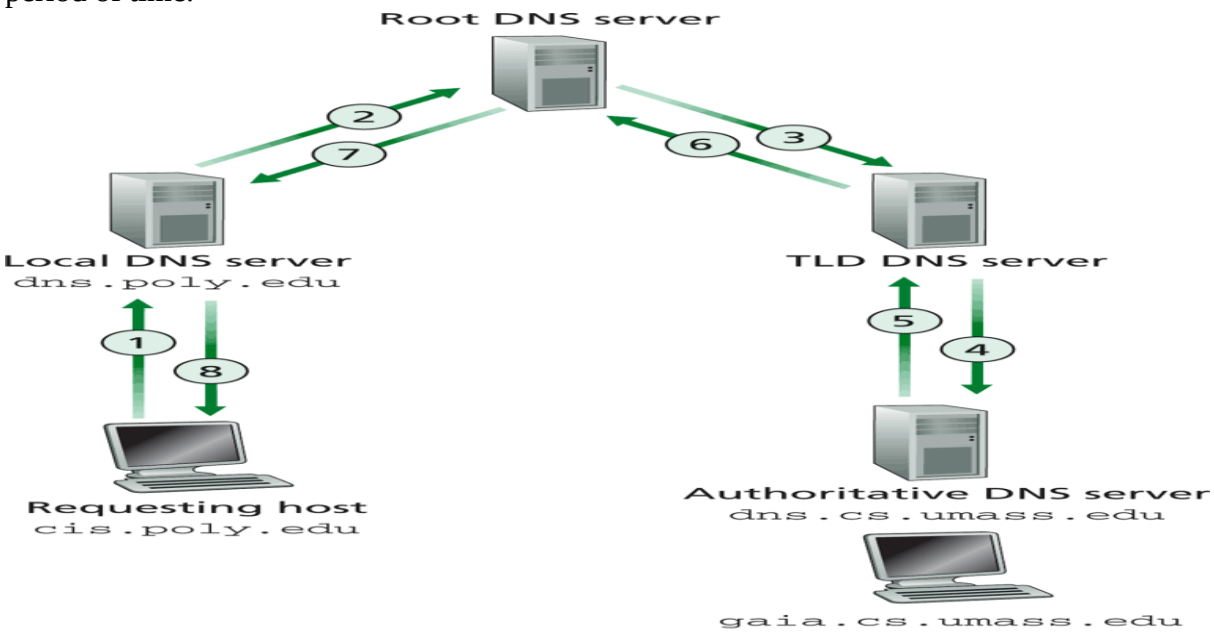



**Figure 2.21** ◆ Recursive queries in DNS

**3.3 Connectionless Transport: UDP:** UDP, Aside from the multiplexing/demultiplexing function and some light error checking, it adds nothing to IP. In fact, if the application developer chooses UDP instead of TCP, then the application is almost directly talking with IP. UDP takes messages from the application process, attaches source and destination port number fields for the multiplexing/demultiplexing service, adds two other small fields, and passes the resulting segment to the network layer. The network layer encapsulates the segment into an IP datagram and then makes a best-effort attempt to deliver the segment to the receiving host. If the segment arrives at the receiving host, UDP uses the port numbers and the IP destination address to deliver the segment's data to the correct application process. Note that with UDP there is no handshaking between sending and receiving transport-layer entities before sending a segment. For this reason, UDP is said to be connectionless

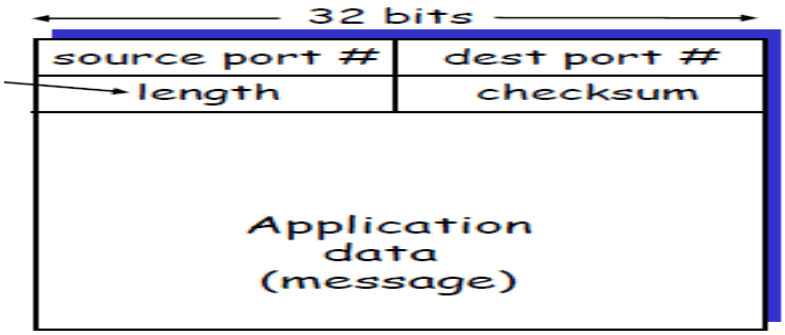
DNS is an example of an application-layer protocol that uses UDP. When the DNS application in a host wants to make a query, it constructs a DNS query message and passes the message to a UDP socket without performing any handshaking; UDP adds header fields to the message and passes the resulting segment to the network layer. The network layer encapsulates the UDP segment into a datagram and sends the datagram to a name server. The DNS application at the querying host then waits for a reply to its query. If it doesn't receive a reply it either tries sending the query to another nameserver, or it informs the invoking application that it can't get a reply.

Many applications are better suited for UDP for the following reasons:

- **No connection establishment**: TCP uses a three-way handshake before it starts to transfer data. UDP just blasts away without any formal preliminaries. Thus UDP does not introduce any delay to establish a connection. This is probably the principal reason why DNS runs over UDP rather than TCP—DNS would be much slower if it ran over TCP. HTTP uses TCP rather than UDP, since reliability is critical for Web pages with text
- **No connection state**: TCP maintains connection state in the end systems. This connection state includes receive and send buffers, congestion-control parameters, and sequence and acknowledgment number parameters. This state information is needed to implement TCP's reliable data transfer service and to provide congestion control. UDP, on the other hand, does not maintain connection state and does not track any of these parameters.
- **Small packet header overhead**: The TCP segment has 20 bytes of header overhead in every segment, whereas UDP only has 8 bytes of overhead.
- **Unregulated send rate**: TCP has a congestion control mechanism that throttles the sender when one or more links between sender and receiver become excessively congested. This throttling can have a severe impact on real-time applications, which can tolerate some packet loss but require a minimum send rate. On the other hand, the speed at which UDP sends data is only constrained by the rate at which the application generates data, the capabilities of the source (CPU, clock rate, and so on) and the access bandwidth to the Internet.

**3.3.1 UDP Segment Structure:** In the UDP segment structure, the application data occupies the data field of the UDP datagram. For example, for DNS, the data field contains either a

query message or a response message. For a streaming audio application, audio samples fill the data field. The UDP header has only four fields, each consisting of two bytes. The port numbers allow the destination host to pass the application data to the correct process running on the destination (that is, the demultiplexing function). The checksum is used by the receiving host to check if errors have been introduced into the segment. In truth, the checksum is also calculated over a few of the fields in the IP header in addition to the UDP segment. The length field specifies the length of the UDP segment, including the header, in bytes




UDP segment format

**3.3.2 UDP checksum:** Its goal is to detect "errors" (e.g., flipped bits) in transmitted segment

At Sender:

Treat segment contents as sequence of 16-bit integers

Checksum: addition (1's complement sum) of segment contents

Sender puts checksum value into UDP checksum field

At Receiver:

Compute checksum of received segment

Check if computed checksum equals checksum field value:

NO - error detected

YES - no error detected.

## Internet Checksum Example

- Note
  - When adding numbers, a carryout from the most significant bit needs to be added to the result
- Example: add two 16-bit integers

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |
| | | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| wraparound | (1) | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 |
| sum | | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 |
| checksum | | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 |

**3.5 Connection-Oriented Transport: TCP:** TCP provides multiplexing, demultiplexing, and error detection in exactly the same manner as UDP. Nevertheless, TCP and UDP differ in many ways. The most fundamental difference is that UDP is connectionless, while TCP is connection-oriented. UDP is connectionless because it sends data without ever establishing a connection.

**3.51. The TCP Connection**: TCP is connection-oriented because before one application process can begin to send data to another, the two processes must first "handshake" with each other--that is, they must send some preliminary segments to each other to establish the parameters of the ensuing data transfer. As part of the TCP connection establishment, both sides of the connection will initialize many TCP "state variables". Because the TCP protocol runs only in the end systems and not in the intermediate network elements (routers and bridges), the intermediate network elements do not maintain TCP connection state. A TCP connection provides for full duplex data transfer. If there is a TCP connection between process A on one host and process B on another host, then application-level data can flow from A to B at the same time as application-level data flows from B to A. A TCP connection is also always point-to-point, that is, between a single sender and a single receiver. So called "multicasting" , the transfer of data from one sender to many receivers in a single send operation--is not possible with TCP.

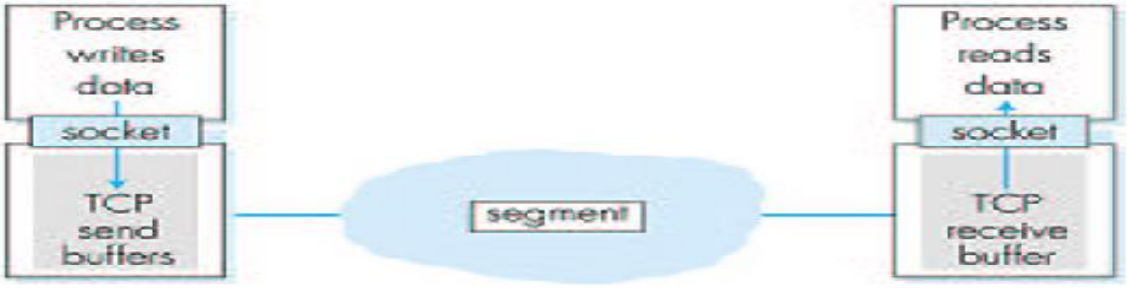



**Figure 3.27:** TCP send and receive buffers

Suppose a process running in one host wants to initiate a connection with another process in another host. The client application process first informs the client TCP that it wants to establish a connection to a process in the server, the server responds with a second special TCP segment; and finally the client responds again with a third special segment. The first two segments contain no "payload," that is, no application-layer data; the third of these segments may carry a payload. Because three segments are sent between the two hosts, this connection establishment procedure is often referred to as a three-way handshake. The client process passes a stream of data through the socket. Once the data passes through the door, the data is now in the hands of TCP running in the client. As shown in Figure 3.27, TCP directs this data to the connection's send buffer, which is one of the buffers that are set aside during the initial three-way handshake. From time to time, TCP will "grab" chunks of data from the send buffer. TCP encapsulates each chunk of client data with a TCP header, thereby forming TCP segments. The segments are passed down to the network layer, where they are separately encapsulated within network-layer IP datagrams. The IP datagrams are then sent into the network. When TCP receives a segment at the other end, the segment's data is placed in the TCP connection's receive buffer. The application reads the stream of data from this buffer.

**3.5.2 TCP Segment Structure:** The TCP segment consists of header fields and a data field. The data field contains a chunk of application data**.** The MSS limits the maximum size of a

segment's data field. When TCP sends a large file, such as an encoded image as part of a Web page, it typically breaks the file into chunks of size MSS (except for the last chunk, which will often be less than the MSS). Interactive applications, however, often transmit data chunks that are smaller than the MSS

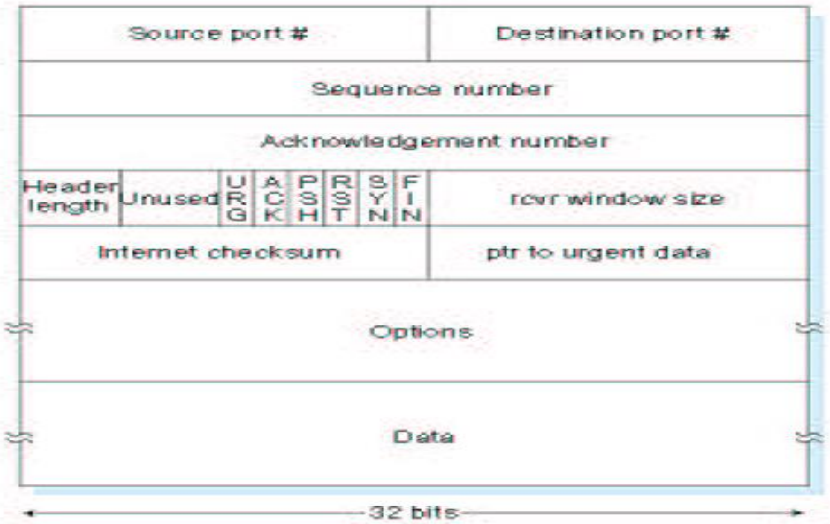


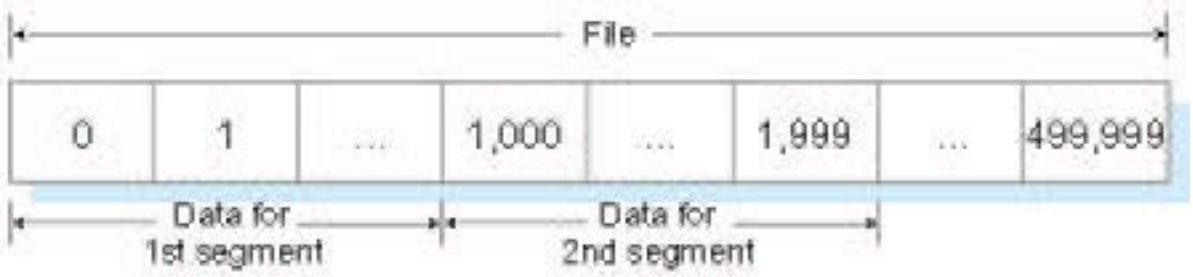

**Figure 3.28:** TCP segment structure

A TCP segment header also contains the following fields:

- The 32-bit **sequence number** field and the 32-bit **acknowledgment number** field are used by the TCP sender and receiver in implementing a reliable data-transfer service, as discussed below.
- The 16-bit **window-size** field is used for flow control. We will see shortly that it is used to indicate the number of bytes that a receiver is willing to accept.
- The 4-bit **length field** specifies the length of the TCP header in 32-bit words. The TCP header can be of variable length due to the TCP options field, discussed below. (Typically, the options field is empty, so that the length of the typical TCP header is 20 bytes.)
- The optional and variable length **options field** is used when a sender and receiver negotiate the maximum segment size (MSS) or as a window scaling factor for use in high-speed networks. A time stamping option is also defined.
- The **flag field** contains 6 bits. The **ACK** bit is used to indicate that the value carried in the acknowledgment field is valid. The **RST, SYN, and FIN** bits are used for connection setup and teardown. When the **PSH** bit is set, this is an indication that the receiver should pass the data to the upper layer immediately. Finally, the **URG** bit is used to indicate that there is data in this segment that the sending-side upper layer entity has marked as "urgent." The location of the last byte of this urgent data is indicated by the 16-bit urgent data pointer. TCP must inform the receiving-side upper-layer entity when urgent data exists and pass it a pointer to the end of the urgent data.

**Sequence Numbers and Acknowledgment Numbers:** Two of the most important fields in the TCP segment header are the sequence number field and the acknowledgment number field. The sequence number for a segment is the byte-stream number of the first byte in the segment. Suppose that the data stream consists of a file consisting of 500,000 bytes, that the MSS is 1,000 bytes, and that the first byte of the data stream is numbered zero. As shown in

Figure 3.29, TCP constructs 500 segments out of the data stream. The first segment gets assigned sequence number 0, the second segment gets assigned sequence number 1000, and the third segment gets assigned sequence number 2000, and so on. Each sequence number is inserted in the sequence number field in the header of the appropriate TCP segment.




The **acknowledgment number** that host A puts in its segment is the sequence number of the next byte host A is expecting from host B. Suppose that host A has received all bytes numbered 0 through 535 from B and suppose that it is about to send a segment to host B. In other words, host A is waiting for byte 536 and all the subsequent bytes in host B's data stream. So host A puts 536 in the acknowledgment number field of the segment it sends to B.

**3.6 Principles of Congestion Control:** packet loss typically results from the overflowing of router buffers as the network becomes congested. Packet retransmission thus treats a symptom of network congestion (the loss of a specific transport-layer segment) but does not treat the cause of network congestion--too many sources attempting to send data at too high a rate. To treat the cause of network congestion, mechanisms are needed to throttle senders in the face of network congestion.

**3.6.1 The Causes and the Costs of Congestion:** The study of congestion control by examining three increasingly complex scenarios in which congestion occurs.

**Scenario 1: Two Senders, a Router with Infinite Buffers:** Two hosts (A and B) each have a connection that shares a single hop between source and destination

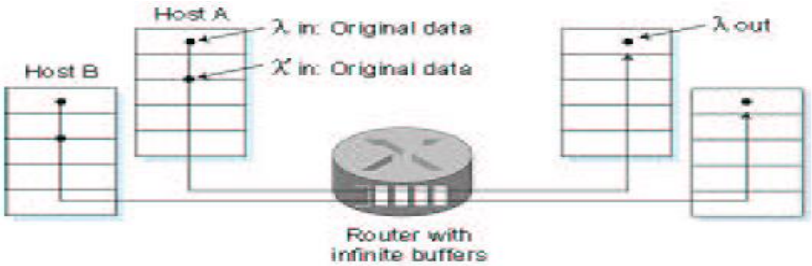



**Figure 3.41:** Congestion scenario 1: Two connections sharing a single hop with infinite buffers

Let's assume that the application in Host A and Host B is sending data into the connection at an average rate of $\lambda_{in}$ bytes/sec. These data are "original" in the sense that each unit of data is sent into the socket only once. The underlying transport-level protocol is a simple one. Data is encapsulated and sent; no error recovery (for example, retransmission), flow control, or congestion control is performed. Packets from hosts A and B pass through a router and over a shared outgoing link of capacity R. The router has buffers that allow it to store incoming packets when the packet-arrival rate exceeds the outgoing link's capacity. The router has an infinite amount of buffer space. In Figure 3.42, the left graph plots the per-connection

throughput (number of bytes per second at the receiver) as a function of the connection sending rate. For a sending rate between 0 and R/2, the throughput at the receiver equals the sender's sending rate--everything sent by the sender is received at the receiver with a finite delay. When the sending rate is above R/2, however, the throughput is only R/2. This upper limit on throughput is a consequence of the sharing of link capacity between two connections. The link simply cannot deliver packets to a receiver at a steady-state rate that exceeds R/2. No matter how high Hosts A and B set their sending rates, they will each never see a throughput higher than R/2. The right-hand graph in Figure 3.42, however, shows the consequences of operating near link capacity. As the sending rate approaches R/2 (from the left), the average delay becomes larger and larger. When the sending rate exceeds R/2, the average number of queued packets in the router is unbounded, and the average delay between source and destination becomes infinite.

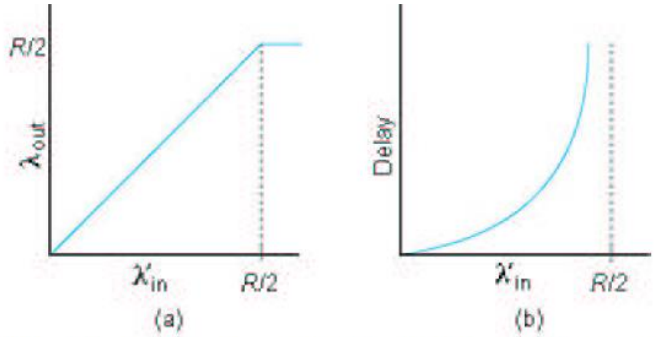



**Figure 3.42:** Congestion scenario 1: Throughput and delay as a function of host sending rate

**Scenario 2: Two Senders, a Router with Finite Buffers:** scenario1 is modified in the following two ways: First, the amount of router buffering is assumed to be finite. Second, we assume that each connection is reliable. If a packet containing a transport-level segment is dropped at the router, it will eventually be retransmitted by the sender. Let us again denote the rate at which the application sends original data into the socket by $\lambda_{in}$ bytes/sec. The rate at which the transport layer sends segments (containing original data or retransmitted data) into the network will be denoted $\lambda_{in}$ bytes/sec. $\lambda_{in}$' is sometimes referred to as the offered load to the network. The performance realized under scenario 2 will now depend strongly on how retransmission is performed. First, consider the unrealistic case that Host A is able to somehow (magically!) determine whether or not a buffer is free in the router and thus sends a packet only when a buffer is free. In this case, no loss would occur, λin would be equal to λin', and the throughput of the connection would be equal to λin.

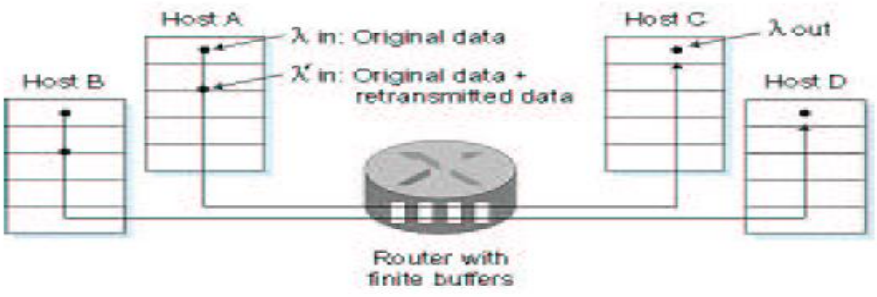



**Figure 3.43:** Scenario 2: Two hosts (with retransmissions) and a router with finite buffers

This case is shown by the upper curve in Figure 3.44(a). Note that the average host sending rate cannot exceed R/2 under this scenario, since packet loss is assumed never to occur. Consider next the slightly more realistic case that the sender retransmits only when a packet is known for certain to be lost. In this case, the performance might look something like that shown in Figure 3.44(b). To appreciate what is happening here, consider the case that the offered load, $\lambda_{in}$' (the rate of original data transmission plus retransmissions), equals 0.5R. According to Figure 3.44(b), at this value of the offered load, the rate at which data are delivered to the receiver application is R/3. Thus, out of the 0.5R units of data transmitted, 0.333R bytes/sec (on average) is original data and 0.266R bytes per second (on average) are retransmitted data. Finally, let us consider the case that the sender may timeout prematurely and retransmit a packet that has been delayed in the queue, but not yet lost. In this case, both the original data packet and the retransmission may both reach the receiver. Of course, the receiver needs but one copy of this packet and will discard the retransmission. In this case, the "work" done by the router in forwarding the retransmitted copy of the original packet was "wasted," as the receiver will have already received the original copy of this packet. The router would have better used the link transmission capacity to send a different packet instead. The lower curve in Figure 3.44(a) shows the throughput versus offered load when each packet is assumed to be forwarded (on average) twice by the router. Since each packet is forwarded twice, the throughput achieved will be given by the line segment in Figure 3.44(a) with the asymptotic value of R/4.

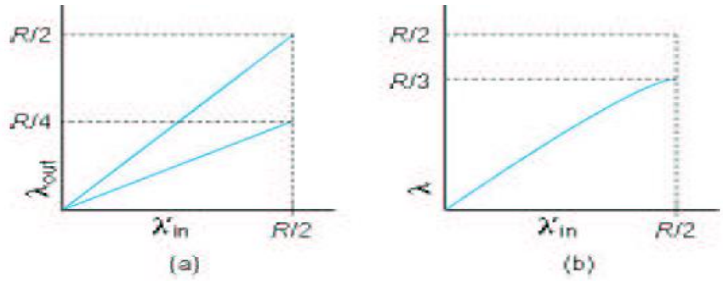



**Figure 3.44:** Scenario 2 performance

**Scenario 3: Four Senders, Routers with Finite Buffers, and Multihop Paths:** In our final congestion scenario, four hosts transmit packets, each over overlapping two-hop paths, as shown in Figure 3.45. We again assume that each host uses a timeout/ retransmission mechanism to implement a reliable data transfer service, that all hosts have the same value of in, and that all router links have capacity R bytes/sec. Let us consider the connection from Host A to Host C, passing through Routers R1 and R2. The A-C connection shares router R1 with the D-B connection and shares router R2 with the B-D connection. For extremely small values of in, buffer overflows are rare (as in congestion scenarios 1 and 2), and the throughput approximately equals the offered load. For slightly larger values of $\lambda_{in}$, the corresponding throughput is also larger, as more original data is being transmitted into the network and delivered to the destination, and overflows are still rare. Thus, for small values of $\lambda_{in}$, an increase in in results $\lambda_{in}$ an increase in $\lambda_{out}$

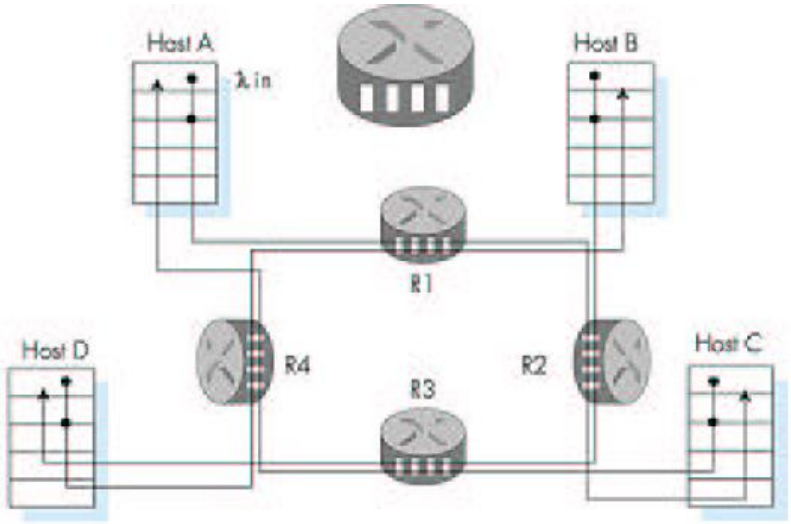



**Figure 3.45:** Four senders, routers with finite buffers, and multihop paths

Having considered the case of extremely low traffic, let us next examine the case that in (and hence $\lambda_{in}$') is extremely large. Consider router R2. The A-C traffic arriving to router R2 can have an arrival rate at R2 that is at most R, the capacity of the link from R1 to R2, regardless of the value of in. If $\lambda_{in}$' is extremely large for all connections (including the B-D connection), then the arrival rate of BD traffic at R2 can be much larger than that of the A-C traffic. Because the A-C and B-D traffic must compete at router R2 for the limited amount of buffer space, the amount of A-C traffic that successfully gets through R2(that is, is not lost due to buffer overflow) becomes smaller and smaller as the offered load from B-D gets larger and larger. In the limit, as the offered load approaches infinity, an empty buffer at R2 is immediately filled by a BD packet, and the throughput of the A-C connection at R2 goes to zero. This, in turn, implies that the A-C end-end throughput goes to zero in the limit of heavy traffic. These considerations give rise to the offered load versus throughput trade off shown in Figure 3.46. The reason for the eventual decrease in throughput with increasing offered load is evident when one considers the amount of wasted "work" done by the network

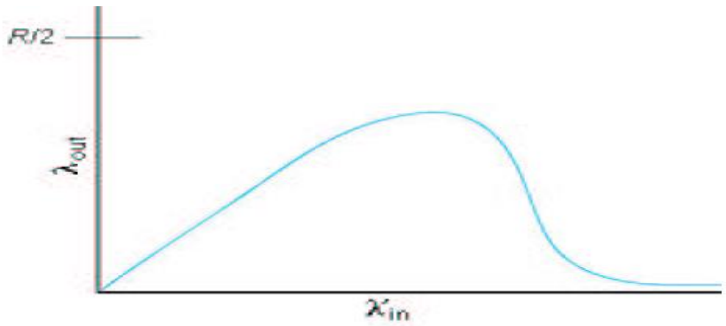



**Figure 3.46:** Scenario 3 performance with finite buffers and multihop paths

**3.6.2 Approaches toward Congestion Control:** At the broadest level, we can distinguish among congestion-control approaches based on whether or not the network layer provides any explicit assistance to the transport layer for congestion-control purposes:

- **End-end congestion control**: In an end-end approach toward congestion control, the network layer provides no explicit support to the transport layer for congestion-control purposes. Even the presence of congestion in the network must be inferred by

the end systems based only on observed network behaviour (for example, packet loss and delay). TCP segment loss (as indicated by a timeout or a triple duplicate acknowledgment) is taken as an indication of network congestion and TCP decreases its window size accordingly.

- **Network-assisted congestion control**: With network-assisted congestion control, network-layer components (that is, routers) provide explicit feedback to the sender regarding the congestion state in the network. This feedback may be as simple as a single bit indicating congestion at a link. Congestion information is typically fed back from the network to the sender in one of two ways, as shown in Figure 3.47. Direct feedback may be sent from a network router to the sender. This form of notification typically takes the form of a choke packet (essentially saying, "I'm congested!"). The second form of notification occurs when a router marks/updates a field in a packet flowing from sender to receiver to indicate congestion. Upon receipt of a marked packet, the receiver then notifies the sender of the congestion indication. Note that this latter form of notification takes at least a full round-trip time.

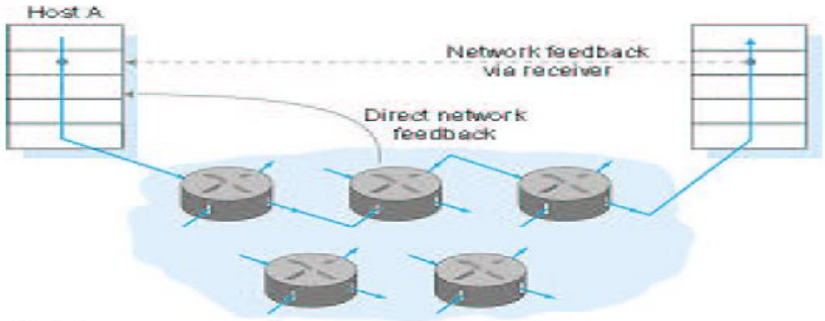



**Figure 3.47:** Two feedback pathways for network-indicated congestion information

**3.7 TCP Congestion Control:** TCP must use end-to-end congestion control rather than network-assisted congestion control, since the IP layer provides no explicit feedback to the end systems regarding network congestion. The approach taken by TCP is to have each sender limit the rate at which it sends traffic into its connection as a function of perceived network congestion. But this approach raises three questions. First, how does a TCP sender limit the rate at which it sends traffic into its connection? Second, how does a TCP sender perceive that there is congestion on the path between itself and the destination? And third, what algorithm should the sender use to change its send rate as a function of perceived end-to-end congestion?

First is each side of a TCP connection consists of a receive buffer, a send buffer, and several variables (LastByteRead, rwnd, and so on). The TCP congestion-control mechanism operating at the sender keeps track of an additional variable, the congestion window. The congestion window, denoted cwnd, imposes a constraint on the rate at which a TCP sender can send traffic into the network. Specifically, the amount of unacknowledged data at a sender may not exceed the minimum of cwnd and rwnd, that is:

LastByteSent – LastByteAcked <= min {cwnd, rwnd}

Roughly, at the beginning of every RTT, the constraint permits the sender to send cwnd bytes of data into the connection; at the end of the RTT the sender receives acknowledgments for the data. Thus the sender's send rate is roughly cwnd/RTT bytes/sec. By adjusting the value of cwnd, the sender can therefore adjust the rate at which it sends data into its connection.

Second a TCP sender perceives that there is congestion on the path between itself and the destination with a "loss event" at a TCP sender as the occurrence of either a timeout or the receipt of three duplicate ACKs from the receiver. Because TCP uses acknowledgments to trigger (or clock) its increase in congestion window size, TCP is said to be self-clocking. Given the mechanism of adjusting the value of cwnd to control the sending rate, the critical question remains: How should a TCP sender determine the rate at which it should send? TCP answers these questions using the following guiding principles:

- A lost segment implies congestion, and hence, the TCP sender's rate should be decreased when a segment is lost.
- An acknowledged segment indicates that the network is delivering the sender's segments to the receiver, and hence, the sender's rate can be increased when an ACK arrives for a previously unacknowledged segment.
- Bandwidth probing: Given ACKs indicating a congestion-free source-to-destination path and loss events indicating a congested path, TCP's strategy for adjusting its transmission rate are to increase its rate in response to arriving ACKs until a loss event occurs, at which point, the transmission rate is decreased. The TCP sender thus increases its transmission rate to probe for the rate that at which congestion onset begins, backs off from that rate, and then to begins probing again to see if the congestion onset rate has changed.

The algorithm has three major components: (1) slow start, (2) congestion avoidance, and (3) fast recovery.

**Slow Start:** When a TCP connection begins, the value of cwnd is typically initialized to a small value of 1 MSS resulting in an initial sending rate of roughly MSS/RTT. TCP sends the first segment into the network and waits for an acknowledgment. When this acknowledgment arrives, the TCP sender increases the congestion window by one MSS and sends out two maximum-sized segments. These segments are then acknowledged, with the sender increasing the congestion window by 1 MSS for each of the acknowledged segments, giving a congestion window of 4 MSS, and so on. This process results in a doubling of the sending rate every RTT. Thus, the TCP send rate starts slow but grows exponentially during the slow start phase. But when should this exponential growth end? Slow start provides several answers to this question. First, if there is a loss event (i.e., congestion) indicated by a timeout, the TCP sender sets the value of cwnd to 1 and begins the slow start process anew. It also sets the value of a second state variable, ssthresh (shorthand for "slow start threshold") to cwnd/2—half of the value of the congestion window value when congestion was detected. Thus, when the value of cwnd equals ssthresh, slow start ends and TCP transitions into congestion avoidance mode. The final way in which slow start can end is if three duplicate ACKs are detected, in which case TCP performs a fast retransmit and enters the fast recovery state.

**Congestion Avoidance**: On entry to the congestion-avoidance state, the value of cwnd is approximately half its value when congestion was last encountered—congestion could be just around the corner! Thus, rather than doubling the value of cwnd every RTT, TCP adopts a more conservative approach and increases the value of cwnd by just a single MSS every RTT.

**Fast Recovery**: In fast recovery, the value of cwnd is increased by 1 MSS for every duplicate ACK received for the missing segment that caused TCP to enter the fast-recovery state. Eventually, when an ACK arrives for the missing segment, TCP enters the congestion-avoidance state after deflating cwnd. If a timeout event occurs, fast recovery transitions to the slow-start state after performing the same actions as in slow start and congestion avoidance: The value of cwnd is set to 1 MSS, and the value of ssthresh is set to half the value of cwnd when the loss event occurred.

Figure 3.53 illustrates the evolution of TCP's congestion window for both Reno and Tahoe. In this figure, the threshold is initially equal to 8 MSS. For the first eight transmission rounds, Tahoe and Reno take identical actions. The congestion window climbs exponentially fast during slow start and hits the threshold at the fourth round of transmission. The congestion window then climbs linearly until a triple duplicate-ACK event occurs, just after transmission round 8.

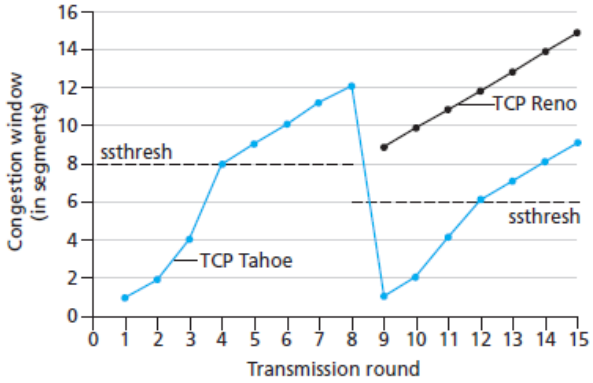



**Figure 3.53** ♦ Evolution of TCP's congestion window (Tahoe and Reno)

TCP's congestion control consists of linear (additive) increase in cwnd of 1 MSS per RTT and then a halving (multiplicative decrease) of cwnd on a triple duplicate-ACK event. For this reason, TCP congestion control is often referred to as an additive-increase, multiplicative decrease (AIMD) form of congestion control.

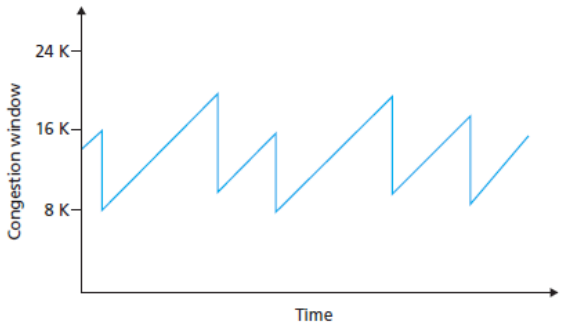



**Figure 3.54** ♦ Additive-increase, multiplicative-decrease congestion control

**Does TCP Ensure Fairness?:** The goal of TCP's congestion control mechanism is to share a bottleneck link's bandwidth evenly among the TCP connections traversing that link. Let's consider the simple case of two TCP connections sharing a single link with transmission rate R, as shown in Figure 3.7-2. We'll assume that the two connections have the same MSS and RTT (so that if they have the same congestion window size, then they have the same throughput), that they have a large amount of data to send, and that no other TCP connections

or UDP datagrams traverse this shared link. Also, we'll ignore the slow start phase of TCP, and assume the TCP connections are operating in congestion avoidance mode (additive increase, multiplicative decrease) at all times.

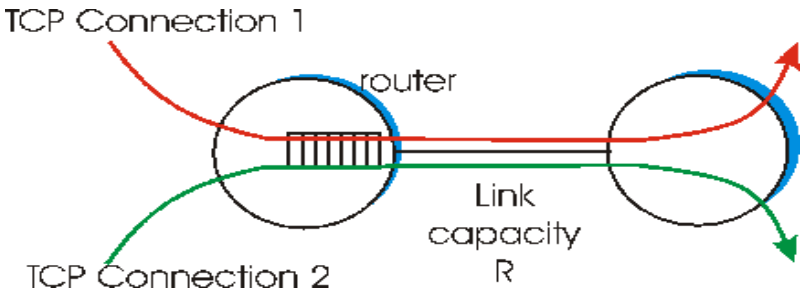



Figure 3.7-3 plots the throughput realized by the two TCP connections. If TCP is to equally share the link bandwidth between the two connections, then the realized throughput should fall along the 45 degree arrow ("equal bandwidth share") emanating from the origin. Ideally, the sum of the two throughputs should equal R (certainly, each connection receiving an equal, but zero, share of the link capacity is not a desirable situation!), so the goal should be to have the achieved throughputs fall somewhere near the intersection of the "equal bandwidth share" line and the "full bandwidth utilization" line in the above figure.

- Many network applications run over TCP rather than UDP because they want to make use of TCP's reliable transport service. But an application developer choosing TCP gets not only reliable data transfer but also TCP congestion control. But even if we could force UDP traffic to behave fairly, the fairness problem would still not be completely solved. This is because there is nothing to stop an application running over TCP from using multiple parallel connections.