UNIT-IV

TRANSPORT LAYER:

The transport layer in the TCP/IP suite is located between the application layer and the network layer. It provides services to the application layer and receives services from the network layer.

The transport layer acts as a liaison between a client program and a server program, a process-to-process connection. The transport layer is the heart of the TCP/IP protocol suite; it is the end-to-end logical vehicle for transferring data from one point to another in the Internet.

Introduction:

The transport layer is located between the application layer and the network layer. It provides a process-to-process communication between two application layers, one at the local host and the other at the remote host.

Communication is provided using a logical connection, which means that the two application layers, which can be located in different parts of the globe, assume that there is an imaginary direct connection through which they can send and receive messages.

THE TRANSPORT SERVICE:

Services provided to the upper layers:

The ultimate goal of the transport layer is to provide efficient, reliable, and cost-effective data transmission service to its users, normally processes in the application layer. To achieve this, the transport layer makes use of the services provided by the network layer. The software and/or hardware within the transport layer that does the work is called the **transport entity**.

The transport entity can be located in the operating system kernel, in a library package bound into network applications, in a separate user process, or even on the network interface card. The first two options are most common on the Internet. The (logical) relationship of the network, transport, and application layers is illustrated in Fig. 4.1.

Just as there are two types of network service, ***connection-oriented*** *and* ***connectionless****, there are also two types of transport service*. The ***connection-oriented transport service*** is similar to the connection-oriented network service in many ways. In both cases, connections have three phases: establishment, data transfer, and release. Addressing and flow control are also similar in both layers.

Furthermore, the ***connectionless transport service*** is also very similar to the connectionless network service. However, note that it can be difficult to provide a connectionless transport service on top of a connection-oriented network service, since it is inefficient to set up a connection to send a single packet and then tear (*meaning run/rip/rush*) it down immediately afterwards.

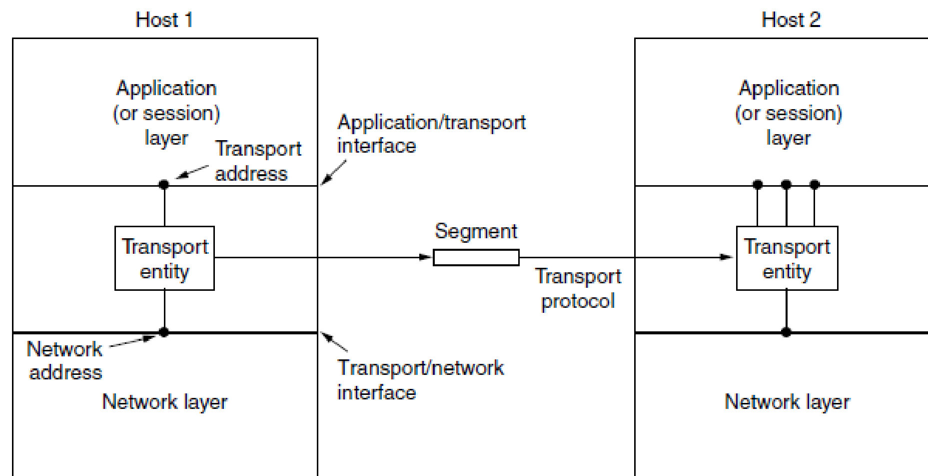

Figure 4.1: The network, transport, and application layers

Transport service primitives:

To allow users to access the transport service, the transport layer must provide some operations to application programs, that is, a transport service interface. Each transport service has its own interface.

The transport service is similar to the network service, but there are also some important differences. The main difference is that the network service is intended to model the service offered by real networks and all. Real networks can lose packets, so the network service is generally unreliable.

The connection-oriented transport service, in contrast, is reliable. Of course, real networks are not error-free, but that is precisely the purpose of the transport layer—to provide a reliable service on top of an unreliable network.

A second difference between the network service and transport service is whom the services are intended for. The network service is used only by the transport entities. Few users write their own transport entities, and thus few users or programs ever (*meaning always/forever/still*) see the bare network service.

Berkeley sockets: Let us now briefly inspect another set of transport primitives, the socket primitives as they are used for TCP. Sockets were first released as part of the Berkeley UNIX 4.2BSD software distribution in 1983. They quickly became popular.

The primitives are now widely used for Internet programming on many operating systems, especially UNIX-based systems, and there is a socket-style API for Windows called ‘‘winsock.’’ The primitives are listed in Fig. 4.2.

Primitive	Meaning
SOCKET	Create a new communication endpoint
BIND	Associate a local address with a socket
LISTEN	Announce willingness to accept connections; give queue size
ACCEPT	Passively establish an incoming connection
CONNECT	Actively attempt to establish a connection
SEND	Send some data over the connection
RECEIVE	Receive some data from the connection
CLOSE	Release the connection

Figure 4.2: The socket primitives for TCP

Note: An Example of Socket Programming: An Internet File Server

ELEMENTS OF TRANSPORT PROTOCOLS:

The transport service is implemented by a **transport protocol** used between the two transport entities. In some ways, transport protocols resemble the data link protocols. Both have to deal with error control, sequencing, and flow control, among other issues.

However, significant differences between the two also exist. These differences are due to major dissimilarities between the environments in which the two protocols operate, as shown in Fig. 4.3.

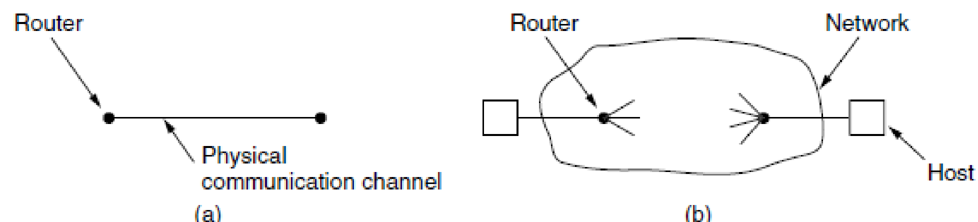

Figure 4.3: Environment of the (a) data link layer (b) transport layer

At the data link layer, two routers communicate directly via a physical channel, whether wired or wireless, whereas at the transport layer, this physical channel is replaced by the entire network.

For one thing, over point-to-point links such as wires or optical fiber, it is usually not necessary for a router to specify which router it wants to talk to—each outgoing line leads directly to a particular router. In the transport layer, explicit addressing of destinations is required.

For another thing, the process of establishing a connection over the wire of Fig. 4.3(a) is simple: the other end is always there (unless it has crashed, in which case it is not there). Either way, there is not much to do.

Even on wireless links, the process is not much different. Just sending a message is sufficient to have it reach all other destinations. If the message is not acknowledged due to an error, it can be resent. In the transport layer, initial connection establishment is complicated.

Addressing:

When an application (e.g., a user) process wishes to set up a connection to a remote application process, it must specify which one to connect to. (Connectionless transport has the same problem: to whom should each message be sent?) The method normally used is to define transport addresses to which processes can listen for connection requests.

In the Internet, these endpoints are called **ports**. We will use the generic term **TSAP** (**Transport Service Access Point**) to mean a specific endpoint in the transport layer. The analogous endpoints in the network layer (i.e., network layer addresses) are naturally called **NSAPs** (**Network Service Access Points**). IP addresses are examples of NSAPs.

Figure 4.4 illustrates the relationship between the NSAPs, the TSAPs, and a transport connection.

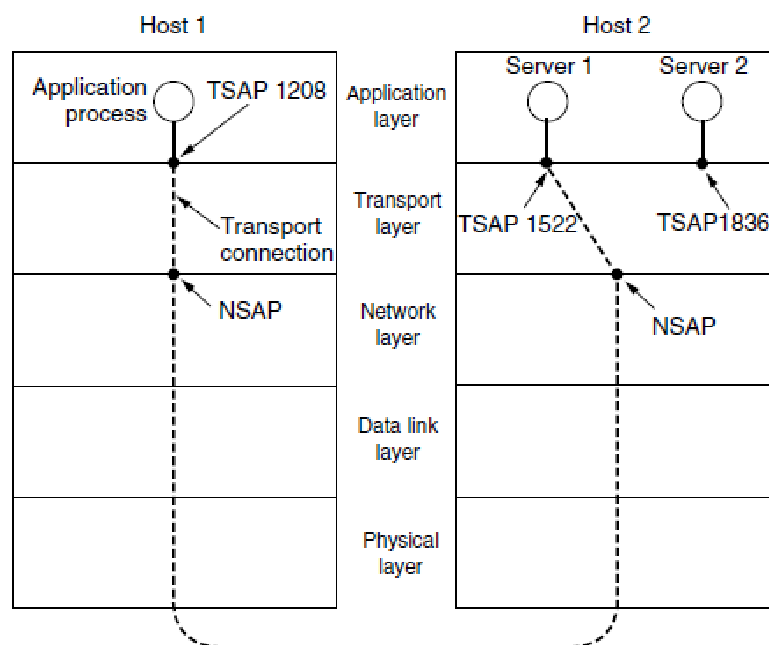

Figure 4.4: TSAPs, NSAPs, and Transport connections

Application processes, both clients and servers, can attach themselves to a local TSAP to establish a connection to a remote TSAP. These connections run through NSAPs on each host, as shown in figure 4.4.

A possible scenario for a transport connection is as follows:

1. A mail server process attaches itself to TSAP 1522 on host 2 to wait for an incoming call. A call such as our LISTEN might be used, for example.

2. An application process on host 1 wants to send an email message, so it attaches itself to TSAP 1208 and issues a CONNECT request.

    - The request specifies TSAP 1208 on host 1 as the source and TSAP 1522 on host 2 as the destination. This action ultimately results in a transport connection being established between the application process and the server.

3. The application process sends over the mail message.

4. The mail server responds to say that it will deliver the message.

5. The transport connection is released.

Connection Establishment:

Establishing a connection sounds easy, but it is actually surprisingly tricky. At first glance, it would seem sufficient for one transport entity to just send a CONNECTION REQUEST segment to the destination and wait for a CONNECTION ACCEPTED reply. The problem occurs when the network can lose, delay, corrupt, and duplicate packets. This behavior causes serious complications.

Imagine a network that is so congested that acknowledgements hardly ever get back in time and each packet times out and is retransmitted two or three times. Suppose that the network uses datagrams inside and that every packet follows a different route.

Some of the packets might get stuck in a traffic jam inside the network and take a long time to arrive. That is, they may be delayed in the network and pop out much later, when the sender thought that they had been lost.

<u>*The worst possible nightmare is as follows.*</u> *A user establishes a connection with a bank, sends messages telling the bank to transfer a large amount of money to the account of a not-entirely-trustworthy person. Unfortunately, the packets decide to take the scenic route to the destination and go off exploring a remote corner of the network.*

The sender then times out and sends them all again. This time the packets take the shortest route and are delivered quickly so the sender releases the connection.

Unfortunately, eventually the initial batch of packets finally come out of hiding and arrive at the destination in order, asking the bank to establish a new connection and transfer money (again). The bank has no way of telling that these are duplicates. It must assume that this is a second, independent transaction, and transfers the money again.

The crux (*meaning root*) of the problem is that the delayed duplicates are thought to be new packets. We cannot prevent packets from being duplicated and delayed. But if and when this happens, the packets must be rejected as duplicates and not processed as fresh packets.

The problem can be attacked in various ways, none of them very satisfactory. One way is to use throwaway transport addresses. In this approach, each time a transport address is needed, a new one is generated. When a connection is released, the address is discarded and never used again. Delayed duplicate packets then never find their way to a transport process and can do no damage.

Note: However, this approach makes it more difficult to connect with a process in the first place.

Another possibility is to give each connection a unique identifier (i.e., a sequence number incremented for each connection established) chosen by the initiating party and put in each segment, including the one requesting the connection.

After each connection is released, each transport entity can update a table listing obsolete connections as (peer transport entity, connection identifier) pairs. Whenever a connection request comes in, it can be checked against the table to see if it belongs to a previously released connection.

Unfortunately, this scheme has a basic flaw: it requires each transport entity to maintain a certain amount of history information indefinitely. This history must persist at both the source and destination machines. Otherwise, if a machine crashes and loses its memory, it will no longer know which connection identifiers have already been used by its peers.

Instead, we need to take a different tack to simplify the problem. Rather than allowing packets to live forever within the network, we devise a mechanism to kill off aged packets that are still hobbling about.

Packet lifetime can be restricted to a known maximum using one (or more) of the following techniques:

1. Restricted network design.
2. Putting a hop counter in each packet.
3. Timestamping each packet.

TCP uses three-way handshake to establish connections in the presence of delayed duplicate control segments as shown in figure 4.5.

Connection Release:

Releasing a connection is easier than establishing one. There are two styles of terminating a connection: ***asymmetric release*** and ***symmetric release***.

Asymmetric release is the way the telephone system works: when one party hangs up, the connection is broken.

Symmetric release treats the connection as two separate unidirectional connections and requires each one to be released separately.

Asymmetric release is abrupt and may result in data loss. Consider the scenario of Fig. 4.6. After the connection is established, host 1 sends a segment that arrives properly at host 2. Then host 1 sends another segment.

Unfortunately, host 2 issues a DISCONNECT before the second segment arrives. The result is that the connection is released and data are lost.

Symmetric release does the job when each process has a fixed amount of data to send and clearly knows when it has sent it. In other situations, determining that all the work has been done and the connection should be terminated is not so obvious.

One can envision a protocol in which host 1 says ‘‘I am done. Are you done too?’’ If host 2 responds: ‘‘I am done too. Goodbye, the connection can be safely released.’’

In practice, we can avoid this quandary (*meaning dilemma/difficulty*) by foregoing the need for agreement and pushing the problem up to the transport user, letting each side independently decide when it is done. This is an easier problem to solve.

Figure 4.7 illustrates four scenarios of releasing using a three-way handshake. While this protocol is not infallible, it is usually adequate. In Fig. 4.7(a), we see the normal case in which one of the users sends a DR (DISCONNECTION REQUEST) segment to initiate the connection release.

When it arrives, the recipient sends back a DR segment and starts a timer, just in case its DR is lost. When this DR arrives, the original sender sends back an ACK segment and releases the connection.

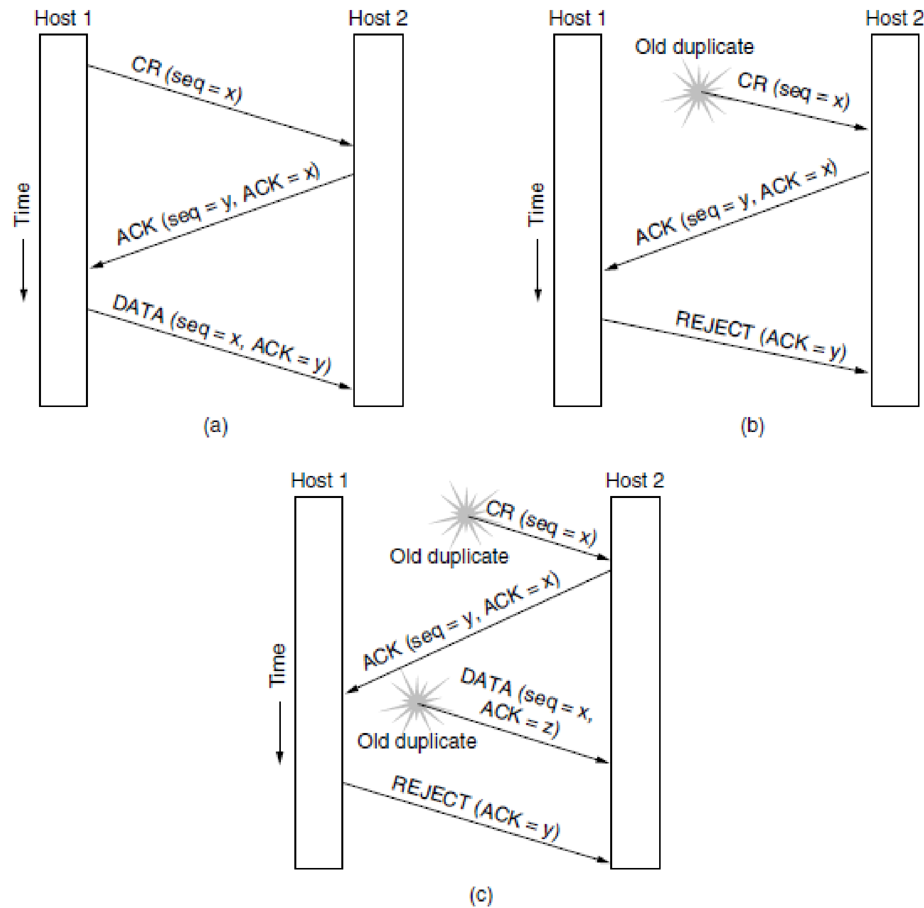

Figure 4.5: Three protocol scenarios for establishing a connection using a three-way handshake. CR denotes Connection Request. (a) normal operation. (b) old duplicate connection request appearing out of nowhere. (c) duplicate connection request and duplicate ack.

Finally, when the ACK segment arrives, the receiver also releases the connection. Releasing a connection means that the transport entity removes the information about the connection from its table of currently open connections and signals the connection's owner (the transport user) somehow.

If the final ACK segment is lost, as shown in Fig. 4.7(b), the situation is saved by the timer. When the timer expires, the connection is released anyway. Now consider the case of the second DR being lost.

The user initiating the disconnection will not receive the expected response, will time out, and will start all over again. In Fig. 4.7(c), we see how this works, assuming that the second time no segments are lost and all segments are delivered correctly and on time.

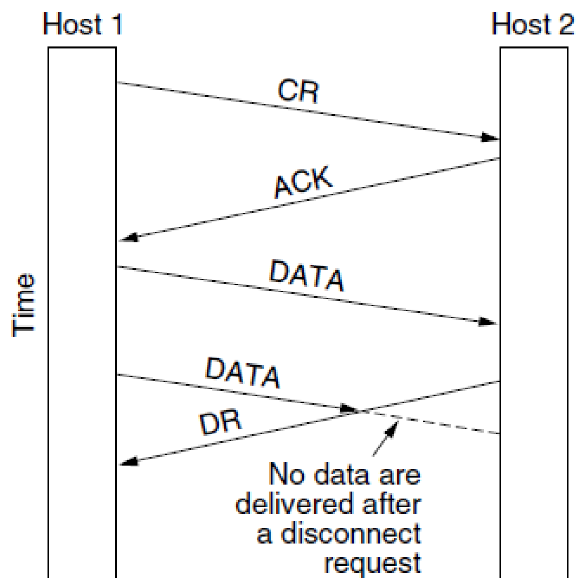

Figure 4.6: Abrupt disconnection with loss of data

Our last scenario, Fig. 4.7(d), is the same as Fig. 4.7(c) except that now we assume all the repeated attempts to retransmit the DR also fail due to lost segments. After *N* retries, the sender just gives up and releases the connection. Meanwhile, the receiver times out and also exits.

Error control and Flow control:

Error control is ensuring that the data is delivered with the desired level of reliability, usually that all of the data is delivered without any errors. Flow control is keeping a fast transmitter from overrunning a slow receiver.

MULTIPLEXING:

Multiplexing, or sharing several conversations over connections, virtual circuits, and physical links plays a role in several layers of the network architecture. In the transport layer, the need for multiplexing can arise in a number of ways. For example, if only one network address is available on a host, all transport connections on that machine have to use it.

When a segment comes in, some way is needed to tell which process to give it to. This situation, called **multiplexing**, is shown in Fig. 4.8(a). In this figure, four distinct transport connections all use the same network connection (e.g., IP address) to the remote host.

Multiplexing can also be useful in the transport layer for another reason. Suppose, for example, that a host has multiple network paths that it can use. If a user needs more bandwidth or more reliability than one of the network paths can provide, a way out is to have a connection that distributes the traffic among multiple network paths on a round-robin basis, as indicated in Fig. 4.8(b).

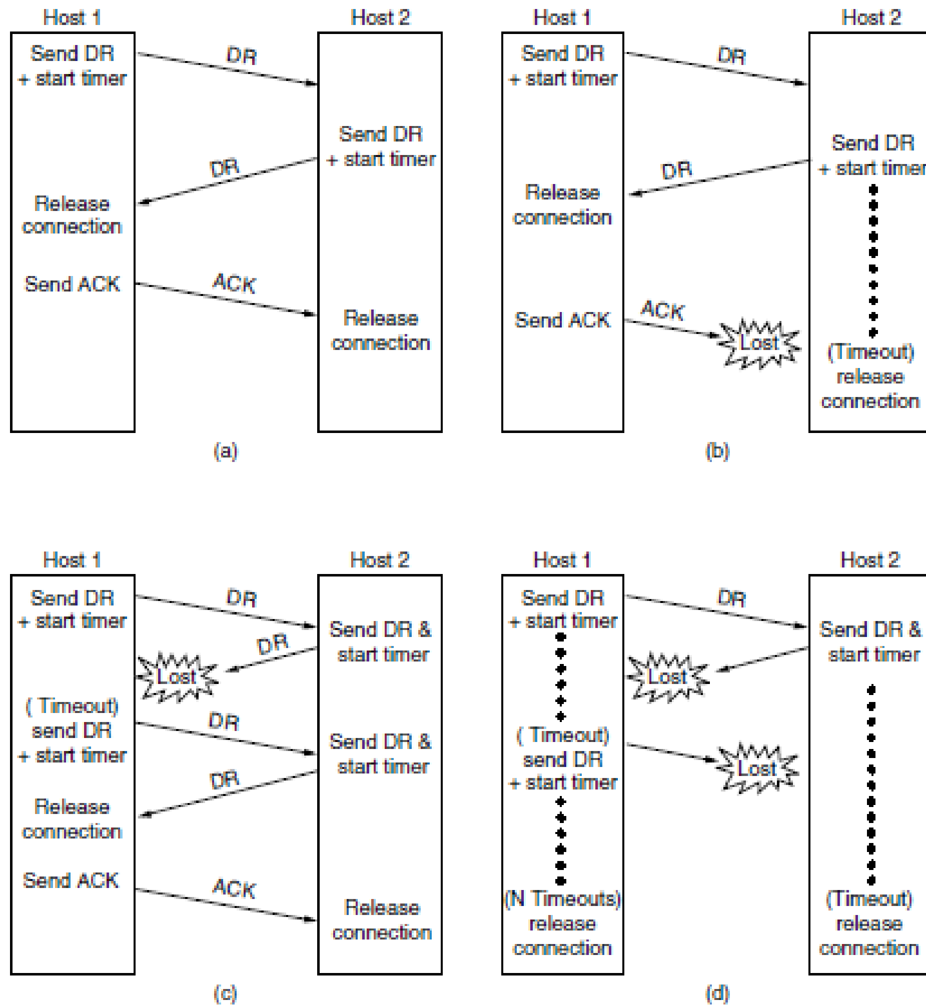

Figure 4.7: Four protocol scenarios for releasing a connection. (a) normal case of three-way handshake. (b) final ACK lost. (c) response lost. (d) response lost and subsequent DRs lost.

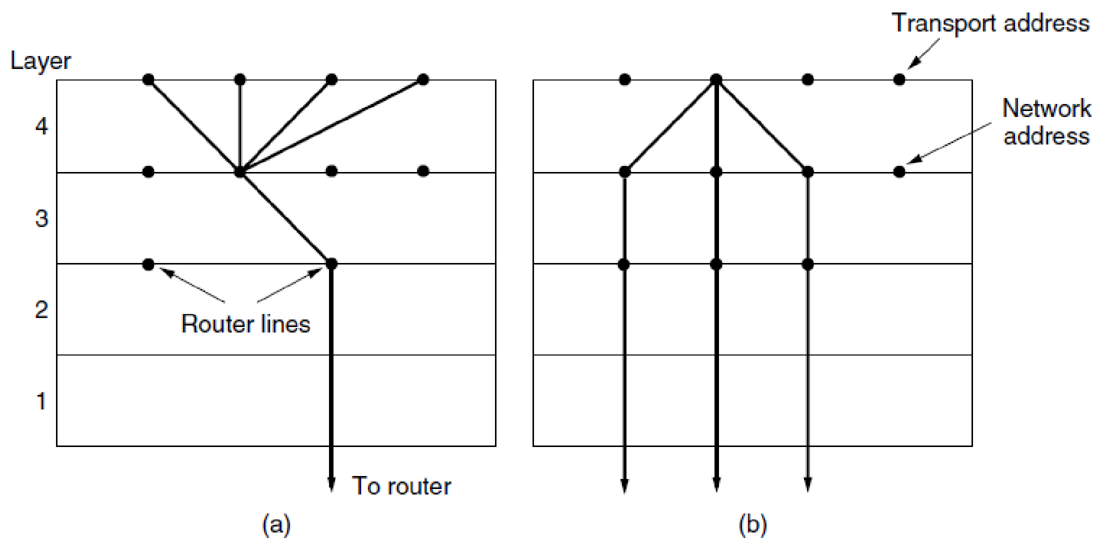

Figure 4.8: (A) Multiplexing (B) Inverse Multiplexing

CRASH RECOVERY:

If hosts and routers are subject to crashes or connections are long-lived (e.g., large software or media downloads), recovery from these crashes becomes an issue.

If the transport entity is entirely within the hosts, recovery from network and router crashes is straightforward. The transport entities expect lost segments all the time and know how to cope with them by using retransmissions.

A more troublesome problem is how to recover from host crashes. In particular, it may be desirable for clients to be able to continue working when servers crash and quickly reboot.

CONGESTION CONTROL:

If the transport entities on many machines send too many packets into the network too quickly, the network will become congested, with performance degraded as packets are delayed and lost.

Controlling congestion to avoid this problem is the combined responsibility of the network and transport layers. Congestion occurs at routers, so it is detected at the network layer.

However, congestion is ultimately caused by traffic sent into the network by the transport layer. The only effective way to control congestion is for the transport protocols to send packets into the network more slowly.

DESIRABLE BANDWIDTH ALLOCATION:

Before we describe how to regulate traffic, we must understand what we are trying to achieve by running a congestion control algorithm. That is, we must specify the state in which a good congestion control algorithm will operate the network.

The goal is more than to simply avoid congestion. It is to find a good allocation of bandwidth to the transport entities that are using the network. A good allocation will deliver good performance because it uses all the available bandwidth but avoids congestion, it will be fair across competing transport entities, and it will quickly track changes in traffic demands.

Efficiency and Power:

An efficient allocation of bandwidth across transport entities will use all of the network capacity that is available. However, it is not quite right to think that if there is a 100-Mbps link, five transport entities should get 20 Mbps each. They should usually get less than 20 Mbps for good performance.

Max-Min Fairness:

In the preceding discussion, we did not talk about how to divide bandwidth between different transport senders. This sounds like a simple question to answer—give all the senders an equal fraction of the bandwidth—but it involves several considerations.

Perhaps the first consideration is to ask what this problem has to do with congestion control.

A second consideration is what a fair portion means for flows in a network. It is simple enough if *N* flows use a single link, in which case they can all have 1/*N* of the bandwidth (although efficiency will dictate that they use slightly less if the traffic is bursty).

But what happens if the flows have different, but overlapping, network paths? For example, one flow may cross three links, and the other flows may cross one link. The three-link flow consumes more network resources. It might be fairer in some sense to give it less bandwidth than the one-link flows.

The form of fairness that is often desired for network usage is **max-min fairness**. An allocation is max-min fair if the bandwidth given to one flow cannot be increased without decreasing the bandwidth given to another flow with an allocation that is no larger.

Convergence:

A final criterion is that the congestion control algorithm converge quickly to a fair and efficient allocation of bandwidth. The discussion of the desirable operating point above assumes a static network environment.

However, connections are always coming and going in a network, and the bandwidth needed by a given connection will vary over time too. Because of the variation in demand, the ideal operating point for the network varies over time.

A good congestion control algorithm should rapidly converge to the ideal operating point, and it should track that point as it changes over time. If the convergence is too slow, the algorithm will never be close to the changing operating point. If the algorithm is not stable, it may fail to converge to the right point in some cases, or even oscillate around the right point.

Regulating the sending rate:

Now it is time to regulate the sending rates to obtain a desirable bandwidth allocation. The sending rate may be limited by two factors.

- The first is flow control, in the case that there is insufficient buffering at the receiver.
- The second is congestion, in the case that there is insufficient capacity in the network.

In Fig. 4.9, we see this problem illustrated hydraulically. In Fig. 4.9(a), we see a thick pipe leading to a small-capacity receiver. This is a flow-control limited situation. As long as the sender does not send more water than the bucket can contain, no water will be lost.

In Fig. 4.9(b), the limiting factor is not the bucket capacity, but the internal carrying capacity of the network. If too much water comes in too fast, it will back up and some will be lost (in this case, by overflowing the funnel).

The way that a transport protocol should regulate the sending rate depends on the form of the feedback returned by the network. Different network layers may return different kinds of feedback. The feedback may be explicit or implicit, and it may be precise or imprecise.

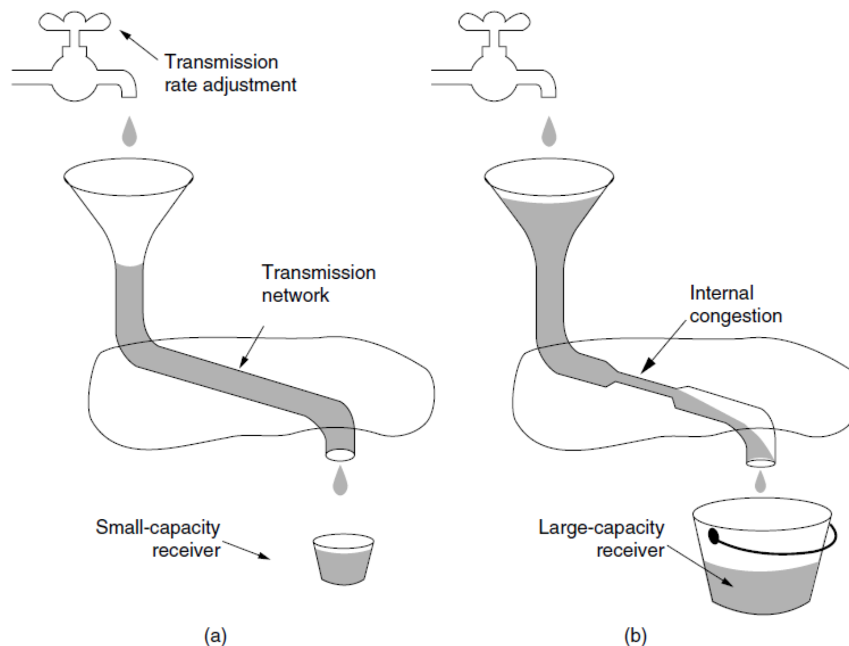

Figure 4.9: (a) a fast network feeding a low-capacity receiver. (b) a slow network feeding a high-capacity receiver.

Wireless issues:

Transport protocols such as TCP that implement congestion control should be independent of the underlying network and link layer technologies. That is a good theory, but in practice there are issues with wireless networks. The main issue is that packet loss is often used as a congestion signal, including by TCP.

Wireless networks lose packets all the time due to transmission errors. To function well, the only packet losses that the congestion control algorithm should observe are losses due to insufficient bandwidth, not losses due to transmission errors. One solution to this problem is to mask the wireless losses by using retransmissions over the wireless link.

THE INTERNET TRANSPORT PROTOCOLS:

UDP:

The Internet has two main protocols in the transport layer, a connectionless protocol and a connection-oriented one. The protocols complement each other.

The connectionless protocol is UDP. It does almost nothing beyond sending packets between applications, letting applications build their own protocols on top as needed.

The connection-oriented protocol is TCP. It does almost everything. It makes connections and adds reliability with retransmissions, along with flow control and congestion control, all on behalf of the applications that use it.

INTRODUCTION TO UDP:

The Internet protocol suite supports a connectionless transport protocol called **UDP** (**User Datagram Protocol**).

UDP provides a way for applications to send encapsulated IP datagrams without having to establish a connection. UDP is described in RFC 768.

UDP transmits **segments** consisting of an 8-byte header followed by the payload. The header is shown in Fig. 4.10. The two **ports** serve to identify the endpoints within the source and destination machines.

When a UDP packet arrives, its payload is handed to the process attached to the destination port. This attachment occurs when the BIND primitive or something similar is used.

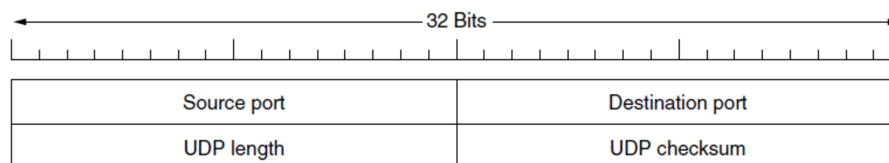

Figure 4.10: the UDP header

Think of ports as mailboxes that applications can rent to receive packets. In fact, the main value of UDP over just using raw IP is the addition of the source and destination ports.

Without the port fields, the transport layer would not know what to do with each incoming packet. With them, it delivers the embedded segment to the correct application.

The source port is primarily needed when a reply must be sent back to the source. By copying the *Source port* field from the incoming segment into the *Destination port* field of the outgoing segment, the process sending the reply can specify which process on the sending machine is to get it.

The *UDP length* field includes the 8-byte header and the data. The minimum length is 8 bytes, to cover the header. The maximum length is 65,515 bytes, which is lower than the largest number that will fit in 16 bits because of the size limit on IP packets.

An optional *Checksum* is also provided for extra reliability. It checksums the header, the data, and a conceptual IP pseudoheader. When performing this computation, the *Checksum* field is set to zero and the data field is padded out with an additional zero byte if its length is an odd number.

The checksum algorithm is simply to add up all the 16-bit words in one's complement and to take the one's complement of the sum.

Remote procedure call:

In a certain sense, sending a message to a remote host and getting a reply back is a lot like making a function call in a programming language. The idea behind RPC is to make a remote procedure call look as much as possible like a local one.

In the simplest form, to call a remote procedure, the client program must be bound with a small library procedure, called the **client stub**, that represents the server procedure in the client's address space.

Similarly, the server is bound with a procedure called the **server stub**. These procedures hide the fact that the procedure call from the client to the server is not local. The actual steps in making an RPC are shown in Fig. 4.12.

- Step 1 is the client calling the client stub. This call is a local procedure call, with the parameters pushed onto the stack in the normal way.
- Step 2 is the client stub packing the parameters into a message and making a system call to send the message. Packing the parameters is called **marshaling**.
- Step 3 is the operating system sending the message from the client machine to the server machine.

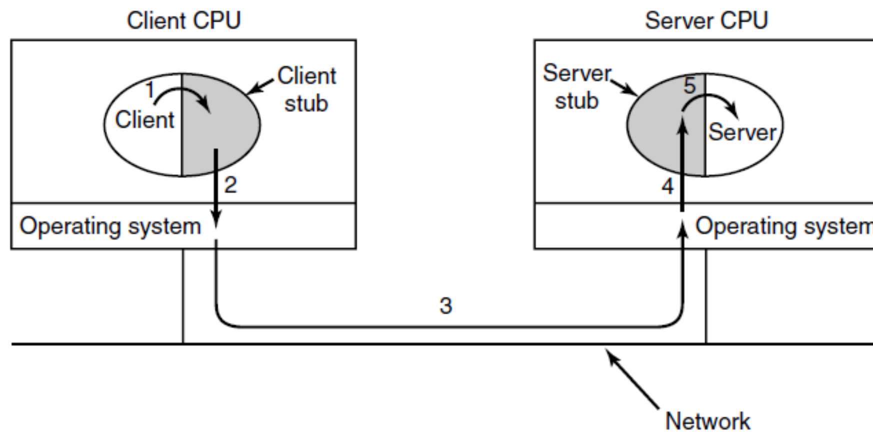

Figure 4.12: Steps in making a remote procedure call, the stubs are shaded

- Step 4 is the operating system passing the incoming packet to the server stub.
- Finally, step 5 is the server stub calling the server procedure with the unmarshaled parameters.

The reply traces the same path in the other direction.

The key item to note here is that the client procedure, written by the user, just makes a normal (i.e., local) procedure call to the client stub, which has the same name as the server procedure. Since the client procedure and client stub are in the same address space, the parameters are passed in the usual way.

Similarly, the server procedure is called by a procedure in its address space with the parameters it expects. To the server procedure, nothing is unusual.

Real-Time Transport Protocols

Client-server RPC is one area in which UDP is widely used. Another one is for real-time multimedia applications.

In particular, as Internet radio, Internet telephony, music-on-demand, videoconferencing, video-on-demand, and other multimedia applications became more commonplace, people have discovered that each application was reinventing more or less the same real-time transport protocol. Thus was **RTP** (**Real-time Transport Protocol**) born.

It is described in RFC 3550 and is now in widespread use for multimedia applications. There are two aspects of real-time transport . The first is the RTP protocol for transporting audio and video data in packets. The second is the processing that takes place, mostly at the receiver, to play out the audio and video at the right time.

RTP—The Real-Time Transport Protocol:

The basic function of RTP is to multiplex several real-time data streams onto a single stream of UDP packets. The UDP stream can be sent to a single destination (unicasting) or to multiple destinations (multicasting).

Because RTP just uses normal UDP, its packets are not treated specially by the routers unless some normal IP quality-of-service features are enabled. In particular, there are no special guarantees about delivery, and packets may be lost, delayed, corrupted, etc.

The RTP format contains several features to help receivers work with multimedia information. The RTP header is illustrated in Fig. 4.13. It consists of three 32-bit words and potentially some extensions.

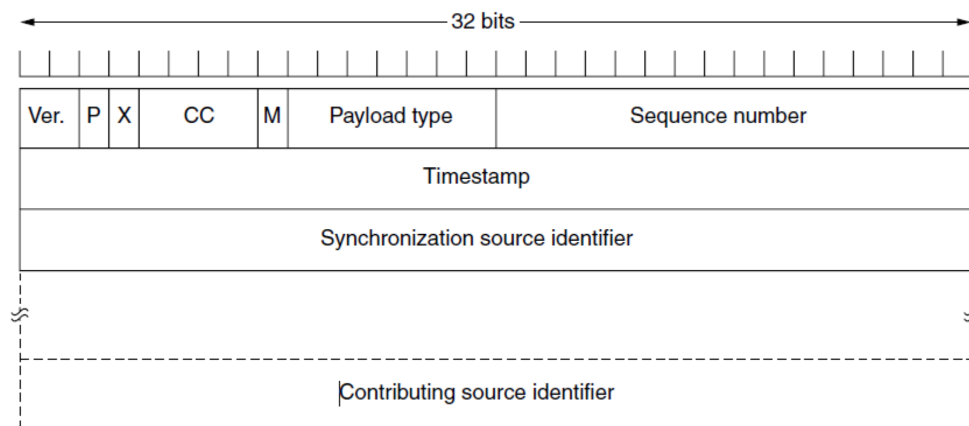

Figure 4.13: The RTP header

The first word contains the Version field, which is already at 2.

The *P* bit indicates that the packet has been padded to a multiple of 4 bytes.

The X bit indicates that an extension header is present.

The CC field tells how many contributing sources are present, from 0 to 15.

The M bit is an application-specific marker bit. It can be used to mark the start of a video frame, the start of a word in an audio channel, or something else that the application understands.

The Payload type field tells which encoding algorithm has been used (e.g., uncompressed 8-bit audio, MP3, etc.).

The Sequence number is just a counter that is incremented on each RTP packet sent. It is used to detect lost packets.

The Timestamp is produced by the stream's source to note when the first sample in the packet was made.

The Synchronization source identifier tells which stream the packet belongs to. It is the method used to multiplex and demultiplex multiple data streams onto a single stream of UDP packets.

Finally, the Contributing source identifiers, if any, are used when mixers are present.

RTCP—The Real-time Transport Control Protocol

RTP has a little sister protocol (little sibling protocol?) called **RTCP** (**Realtime Transport Control Protocol**). It is defined along with RTP in RFC 3550 and handles feedback, synchronization, and the user interface. It does not transport any media samples.

THE INTERNET TRANSPORT PROTOCOLS:

TCP

UDP is a simple protocol and it has some very important uses, such as clientserver interactions and multimedia, but for most Internet applications, reliable, sequenced delivery is needed. UDP cannot provide this, so another protocol is required. It is called TCP and is the main workhorse of the Internet.

Introduction to TCP:

TCP (**Transmission Control Protocol**) was specifically designed to provide a reliable end-to-end byte stream over an unreliable internetwork. An internetwork differs from a single network because different parts may have wildly different topologies, bandwidths, delays, packet sizes, and other parameters.

TCP was designed to dynamically adapt to properties of the internetwork and to be robust in the face of many kinds of failures. TCP was formally defined in RFC 793 in September 1981.

As time went on, many improvements have been made, and various errors and inconsistencies have been fixed. To give you a sense of the extent of TCP, the important RFCs are now RFC 793 plus: clarifications and bug fixes in RFC 1122; extensions for high-performance in RFC 1323.

Selective acknowledgements in RFC 2018; congestion control in RFC 2581; repurposing of header fields for quality of service in RFC 2873; improved retransmission timers in RFC 2988; and explicit congestion notification in RFC 3168. The IP layer gives no guarantee that datagrams will be delivered properly, nor any indication of how fast datagrams may be sent.

It is up to TCP to send datagrams fast enough to make use of the capacity but not cause congestion, and to time out and retransmit any datagrams that are not delivered. Datagrams that do arrive may well do so in the wrong order; it is also up to TCP to reassemble them into messages in the proper sequence.

The TCP Service Model:

TCP service is obtained by both the sender and the receiver creating end points, called **sockets.** Each socket has a socket number (address) consisting of the IP address of the host and a 16-bit number local to that host, called a **port**. A port is the TCP name for a TSAP.

For TCP service to be obtained, a connection must be explicitly established between a socket on one machine and a socket on another machine. A socket may be used for multiple connections at the same time. In other words, two or more connections may terminate at the same socket.

Port numbers below 1024 are reserved for standard services that can usually only be started by privileged users (e.g., root in UNIX systems). They are called **well-known ports**.

For example, any process wishing to remotely retrieve mail from a host can connect to the destination host's port 143 to contact its IMAP daemon. The list of well-known ports is given at *www.iana.org*. Over 700 have been assigned. A few of the better-known ones are listed in Fig. 4.14.

Port	Protocol	Use
20, 21	FTP	File transfer
22	SSH	Remote login, replacement for Telnet
25	SMTP	Email
80	HTTP	World Wide Web
110	POP-3	Remote email access
143	IMAP	Remote email access
443	HTTPS	Secure Web (HTTP over SSL/TLS)
543	RTSP	Media player control
631	IPP	Printer sharing

Figure 4.14: Some assigned ports

All TCP connections are full duplex and point-to-point. Full duplex means that traffic can go in both directions at the same time. Point-to-point means that each connection has exactly two end points. TCP does not support multicasting or broadcasting.

A TCP connection is a byte stream, not a message stream. Message boundaries are not preserved end to end.

The TCP Protocol:

A key feature of TCP, and one that dominates the protocol design, is that every byte on a TCP connection has its own 32-bit sequence number. When the Internet began, the lines between routers were mostly 56-kbps leased lines, so a host blasting away at full speed took over 1 week to cycle through the sequence numbers.

The sending and receiving TCP entities exchange data in the form of segments. A **TCP segment** consists of a fixed 20-byte header (plus an optional part) followed by zero or more data bytes. The TCP software decides how big segments should be.

It can accumulate data from several writes into one segment or can split data from one write over multiple segments. Two limits restrict the segment size. First, each segment, including the TCP header, must fit in the 65,515- byte IP payload. Second, each link has an **MTU** (**Maximum Transfer Unit**).

Each segment must fit in the MTU at the sender and receiver so that it can be sent and received in a single, unfragmented packet. However, it is still possible for IP packets carrying TCP segments to be fragmented when passing over a network path for which some link has a small MTU.

If this happens, it degrades performance and causes other problems. Instead, modern TCP implementations perform **path MTU discovery** by using the technique outlined in RFC 1191. This technique uses ICMP error messages to find the smallest MTU for any link on the path. TCP then adjusts the segment size downwards to avoid fragmentation.

The basic protocol used by TCP entities is the sliding window protocol with a dynamic window size. When a sender transmits a segment, it also starts a timer. When the segment arrives at the destination, the receiving TCP entity sends back a segment (with data if any exist, and otherwise without) bearing an acknowledgement number equal to the next sequence number it expects to receive and the remaining window size.

If the sender’s timer goes off before the acknowledgement is received, the sender transmits the segment again.

The TCP Segment Header:

Figure 4.15 shows the layout of a TCP segment. Every segment begins with a fixed-format, 20-byte header. The fixed header may be followed by header options. After the options, if any, up to 65,535 − 20 − 20 = 65,495 data bytes may follow, where the first 20 refer to the IP header and the second to the TCP header.

Segments without any data are legal and are commonly used for acknowledgements and control messages.

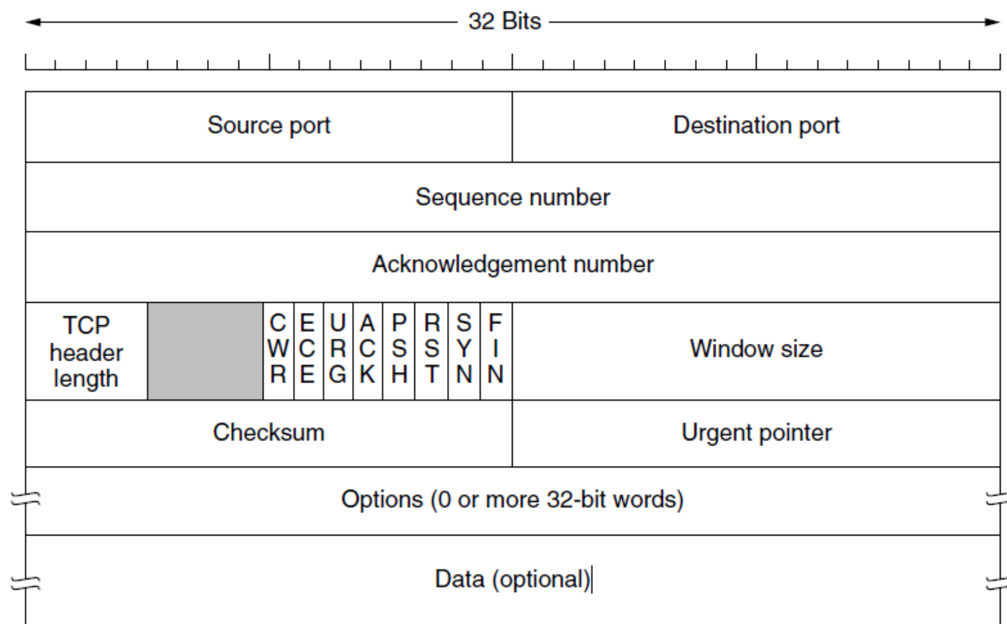

Figure 4.15: The TCP Header

The *Source port* and *Destination port* fields identify the local end points of the connection. The source and destination end points together identify the connection. This connection identifier is called a **5 tuple** because it consists of five pieces of information: the protocol (TCP), source IP and source port, and destination IP and destination port.

The *Sequence number* and *Acknowledgement number* fields perform their usual functions.

The *Sequence number* and *Acknowledgement number* fields perform their usual functions.

The *TCP header length* tells how many 32-bit words are contained in the TCP header. This information is needed because the *Options* field is of variable length, so the header is, too.

Now come eight 1-bit flags. *CWR* and *ECE* are used to signal congestion when ECN (Explicit Congestion Notification) is used. *CWR* is set to signal *Congestion Window Reduced* from the TCP sender to the TCP receiver so that it knows the sender has slowed down and can stop sending the *ECN-Echo*.

URG is set to 1 if the *Urgent pointer* is in use. The *Urgent pointer* is used to indicate a byte offset from the current sequence number at which urgent data are to be found.

The *ACK* bit is set to 1 to indicate that the *Acknowledgement number* is valid. This is the case for nearly all packets. If *ACK* is 0, the segment does not contain an acknowledgement, so the *Acknowledgement number* field is ignored.

The *PSH* bit indicates PUSHed data. The receiver is hereby kindly requested to deliver the data to the application upon arrival and not buffer it until a full buffer has been received (which it might otherwise do for efficiency).

The *RST* bit is used to abruptly reset a connection that has become confused due to a host crash or some other reason.

The *SYN* bit is used to establish connections. The *FIN* bit is used to release a connection.

The *Window size* field tells how many bytes may be sent starting at the byte acknowledged.

A *Checksum* is also provided for extra reliability. The *Options* field provides a way to add extra facilities not covered by the regular header.

TCP Connection Establishment:

Connections are established in TCP by means of the three-way handshake. To establish a connection, one side, say, the server, passively waits for an incoming connection by executing the LISTEN and ACCEPT primitives in that order, either specifying a specific source or nobody in particular.

The other side, say, the client, executes a CONNECT primitive, specifying the IP address and port to which it wants to connect, the maximum TCP segment size it is willing to accept, and optionally some user data (e.g., a password). The CONNECT primitive sends a TCP segment with the *SYN* bit on and *ACK* bit off and waits for a response.

When this segment arrives at the destination, the TCP entity there checks to see if there is a process that has done a LISTEN on the port given in the *Destination port* field. If not, it sends a reply with the *RST* bit on to reject the connection.

TCP Connection Release

Although TCP connections are full duplex, to understand how connections are released it is best to think of them as a pair of simplex connections. Each simplex connection is released independently of its sibling.

To release a connection, either party can send a TCP segment with the *FIN* bit set, which means that it has no more data to transmit. When the *FIN* is acknowledged, that direction is shut down for new data.

Data may continue to flow indefinitely in the other direction, however. When both directions have been shut down, the connection is released.

TCP Congestion Control:

The network layer detects congestion when queues grow large at routers and tries to manage it, if only by dropping packets. It is up to the transport layer to receive congestion feedback from the network layer and slow down the rate of traffic that it is sending into the network.

In the Internet, TCP plays the main role in controlling congestion, as well as the main role in reliable transport. That is why it is such a special protocol.

PERFORMANCE PROBLEMS IN COMPUTER NETWORKS

Some performance problems, such as congestion, are caused by temporary resource overloads. If more traffic suddenly arrives at a router than the router can handle, congestion will build up and performance will suffer.

Performance also degrades when there is a structural resource imbalance. For example, if a gigabit communication line is attached to a low-end PC, the poor host will not be able to process the incoming packets fast enough and some will be lost. These packets will eventually be retransmitted, adding delay, wasting bandwidth, and generally reducing performance.

Overloads can also be synchronously triggered. As an example, if a segment contains a bad parameter , in many cases the receiver will thoughtfully send back an error notification.

Another tuning issue is setting timeouts. When a segment is sent, a timer is set to guard against loss of the segment. If the timeout is set too short, unnecessary retransmissions will occur, clogging the wires. If the timeout is set too long, unnecessary delays will occur after a segment is lost.

NETWORK PERFORMANCE MEASUREMENT:

When a network performs poorly, its users often complain to the folks running it, demanding improvements. To improve the performance, the operators must first determine exactly what is going on. To find out what is really happening, the operators must make measurements.

Measurements can be made in different ways and at many locations (both in the protocol stack and physically). The most basic kind of measurement is to start a timer when beginning some activity and see how long that activity takes.

Other measurements are made with counters that record how often some event has happened (e.g., number of lost segments).

Measuring network performance and parameters has many potential pitfalls. We list a few of them here. Any systematic attempt to measure network performance should be careful to avoid these.

1) Make Sure That the Sample Size Is Large Enough

Do not measure the time to send one segment, but repeat the measurement, say, one million times and take the average.

2) Make Sure That the Samples Are Representative

Ideally, the whole sequence of one million measurements should be repeated at different times of the day and the week to see the effect of different network conditions on the measured quantity.

3) Caching Can Wreak Havoc with Measurements

Repeating a measurement many times will return an unexpectedly fast answer if the protocols use caching mechanisms.

4) Be Sure That Nothing Unexpected Is Going On during Your Tests

Making measurements at the same time that some user has decided to run a video conference over your network will often give different results than if there is no video conference.

5) Be Careful When Using a Coarse-Grained Clock

Computer clocks function by incrementing some counter at regular intervals.

6) Be Careful about Extrapolating the Results

Suppose that you make measurements with simulated network loads running from 0 (idle) to 0.4 (40% of capacity).

UNIT-V

INTRODUCTION TO APPLICATION LAYER:

INTRODUCTION:

The application layer provides services to the user. Communication is provided using a logical connection, which means that the two application layers assume that there is an imaginary direct connection through which they can send and receive messages.

Providing Services:

All communication networks that started before the Internet were designed to provide services to network users. Most of these networks, however, were originally designed to provide one specific service. For example, the telephone network was originally designed to provide voice service: to allow people all over the world to talk to each other. This network, however, was later used for some other services, such as facsimile (fax), enabled by users adding some extra hardware at both ends.

The Internet was originally designed for the same purpose: to provide service to users around the world. The layered architecture of the TCP/IP protocol suite, however, makes the Internet more flexible than other communication networks such as postal or telephone networks.

Each layer in the suite was originally made up of one or more protocols, but new protocols can be added or some protocols can be removed or replaced by the Internet authorities. However, if a protocol is added to each layer, it should be designed in such a way that it uses the services provided by one of the protocols at the lower layer.

If a protocol is removed from a layer, care should be taken to change the protocol at the next higher layer that supposedly uses the services of the removed protocol. The application layer, however, is somewhat different from other layers in that it is the highest layer in the suite.

The protocols in this layer do not provide services to any other protocol in the suite; they only receive services from the protocols in the transport layer. This means that protocols can be removed from this layer easily. New protocols can be also added to this layer as long as the new protocols can use the services provided by one of the transport-layer protocols.

Standard and Nonstandard Protocols:

To provide smooth operation of the Internet, the protocols used in the first four layers of the TCP/IP suite need to be standardized and documented.

Standard Application-Layer Protocols:

There are several application-layer protocols that have been standardized and documented by the Internet authority, and we are using them in our daily interaction with the Internet.

Each standard protocol is a pair of computer programs that interact with the user and the transport layer to provide a specific service to the user.

Nonstandard Application-Layer Protocols:

A programmer can create a nonstandard application-layer program if she can write two programs that provide service to the user by interacting with the transport layer.

Application-Layer Paradigms

It should be clear that to use the Internet we need two application programs to interact with each other: one running on a computer somewhere in the world, the other running on another computer somewhere else in the world. The two programs need to send messages to each other through the Internet infrastructure.

However, we have not discussed what the relationship should be between these programs.

Should both application programs be able to request services and provide services, or should the application programs just do one or the other?

Two paradigms have been developed during the lifetime of the Internet to answer this question: the client-server paradigm and the peer-to-peer paradigm.

Traditional Paradigm: Client-Server:

The traditional paradigm is called the **client-server paradigm**. It was the most popular paradigm until a few years ago. In this paradigm, the service provider is an application program, called the server process; it runs continuously, waiting for another application program, called the client process, to make a connection through the Internet and ask for service.

There are normally some server processes that can provide a specific type of service, but there are many clients that request service from any of these server processes. The server process must be running all the time; the client process is started when the client needs to receive service.

New Paradigm: Peer-to-Peer:

A new paradigm, called the **peer-to-peer paradigm** (often abbreviated *P2P paradigm*) has emerged to respond to the needs of some new applications.

In this paradigm, there is no need for a server process to be running all the time and waiting for the client processes to connect. The responsibility is shared between peers.

A computer connected to the Internet can provide service at one time and receive service at another time. A computer can even provide and receive services at the same time.

CLIENT-SERVER PROGRAMMING:

In a client-server paradigm, communication at the application layer is between two running application programs called ***processes****:* a *client* and a *server*.

A client is a running program that initializes the communication by sending a request; a server is another application program that waits for a request from a client.

The server handles the request received from a client, prepares a result, and sends the result back to the client. This definition of a server implies that a server must be running when a request from a client arrives, but the client needs to be run only when it is needed.

This means that if we have two computers connected to each other somewhere, we can run a client process on one of them and the server on the other. However, we need to be careful that the server program is started before we start running the client program.

Application Programming Interface:

A client process communicate with a server process with the help of a computer program which is normally written in a computer language with a predefined set of instructions that tells the computer what to do.

A computer language has a set of instructions for mathematical operations, a set of instructions for string manipulation, a set of instructions for input/output access, and so on.

If we need a process to be able to communicate with another process, we need a new set of instructions to tell the lowest four layers of the TCP/IP suite to open the connection, send and receive data from the other end, and close the connection. A set of instructions of this kind is normally referred to as an **application programming interface (API)**.

An interface in programming is a set of instructions between two entities. In this case, one of the entities is the process at the application layer and the other is the *operating system* that encapsulates the first four layers of the TCP/IP protocol suite.

Several APIs have been designed for communication. One of the most common one is: **socket interface**. The socket interface is a set of instructions that provide communication between the application layer and the operating system, as shown in Figure 5.1.

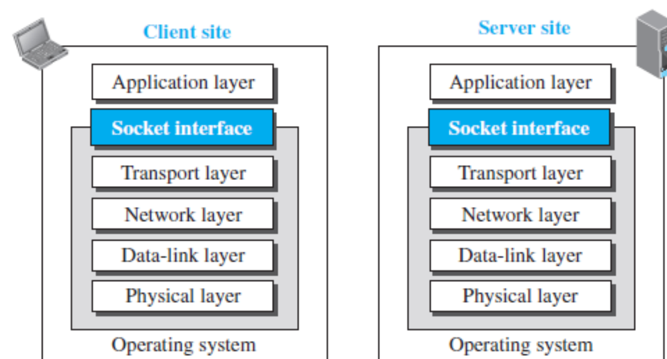

FIGURE 5.1: Position Of The Socket Interface

It is a set of instructions that can be used by a process to communicate with another process. The idea of sockets allows us to use the set of all instructions already designed in a programming language for other sources and sinks.

For example, in most computer languages, like C, C++, or Java, we have several instructions that can read and write data to other sources and sinks such as a keyboard (a source), a monitor (a sink), or a file (source and sink). We can use the same instructions to read from or write to sockets.

Sockets:

Although a socket is supposed to behave like a terminal or a file, it is not a physical entity like them; it is an abstraction. It is an object that is created and used by the application program.

Socket Addresses:

The interaction between a client and a server is two-way communication. In a two-way communication, we need a pair of addresses: local (sender) and remote (receiver). The local address in one direction is the remote address in the other direction and vice versa.

Since communication in the client-server paradigm is between two sockets, we need a pair of **socket addresses** for communication: a local socket address and a remote socket address. However, we need to define a socket address in terms of identifiers used in the TCP/IP protocol suite.

A socket address should first define the computer on which a client or a server is running. Socket address should be a combination of an IP address (32 bit) and a port number (16 bit).

Since a socket defines the end-point of the communication, we can say that a socket is identified by a pair of socket addresses, a local and a remote.

Finding Socket Addresses: How can a client or a server find a pair of socket addresses for communication? The situation is different for each site.

Server Site: The server needs a local (server) and a remote (client) socket address for communication.

Local Socket Address The local (server) socket address is provided by the operating system. The operating system knows the IP address of the computer on which the server process is running. The port number of a server process, however, needs to be assigned.

If the server process is a standard one defined by the Internet authority, a port number is already assigned to it. For example, the assigned port number for a Hypertext Transfer Protocol (HTTP) is the integer 80, which cannot be used by any other process.

Remote Socket Address The remote socket address for a server is the socket address of the client that makes the connection. Since the server can serve many clients, it does not know beforehand the remote socket address for communication.

The server can find this socket address when a client tries to connect to the server. The client socket address, which is contained in the request packet sent to the server, becomes the remote socket address that is used for responding to the client.

Client Site: The client also needs a local (client) and a remote (server) socket address for communication.

Local Socket Address The local (client) socket address is also provided by the operating system. The operating system knows the IP address of the computer on which the client is running. The port number, however, is a 16-bit temporary integer that is assigned to a client process each time the process needs to start the communication.

The port number, however, needs to be assigned from a set of integers defined by the Internet authority and called the ephemeral (temporary) port numbers. The operating system, however, needs to guarantee that the new port number is not used by any other running client process.

Remote Socket Address Finding the remote (server) socket address for a client, however, needs more work. When a client process starts, it should know the socket address of the server it wants to connect to.

Using Services of the Transport Layer:

A pair of processes provide services to the users of the Internet, human or programs. A pair of processes, however, need to use the services provided by the transport layer for communication because there is no physical communication at the application layer.

WORLD WIDE WEB AND HTTP:

World Wide Web:

The idea of the Web was first proposed by Tim Berners-Lee in 1989. The Web today is a repository of information in which the documents, called ***web pages,*** are distributed all over the world and related documents are linked together.

The popularity and growth of the Web can be related to two terms in the above statement: *distributed* and *linked*. Distribution allows the growth of the Web. Each web server in the world can add a new web page to the repository and announce it to all Internet users without overloading a few servers.

Linking allows one web page to refer to another web page stored in another server somewhere else in the world. The linking of web pages was achieved using a concept called ***hypertext,*** which was introduced many years before the advent of the Internet.

The idea was to use a machine that automatically retrieved another document stored in the system when a link to it appeared in the document. The Web implemented this idea electronically to allow the linked document to be retrieved when the link was clicked by the user.

Today, the term *hypertext,* coined to mean linked text documents, has been changed to ***hypermedia,*** to show that a web pagecan be a text document, an image, an audio file, or a video file.

Architecture:

The WWW today is a distributed client-server service, in which a client using a browser can access a service using a server. However, the service provided is distributed over many locations called *sites.* Each site holds one or more web pages.

Each web page, however, can contain some links to other web pages in the same or other sites. In other words, a web page can be simple or composite. A simple web page has no links to other web pages; a composite web page has one or more links to other web pages. Each web page is a file with a name and address.

Web Client (Browser): A variety of vendors offer commercial **browsers** that interpret and display a web page, and all of them use nearly the same architecture. Each browser usually consists of three parts: a controller, client protocols, and interpreters (figure 5.2).

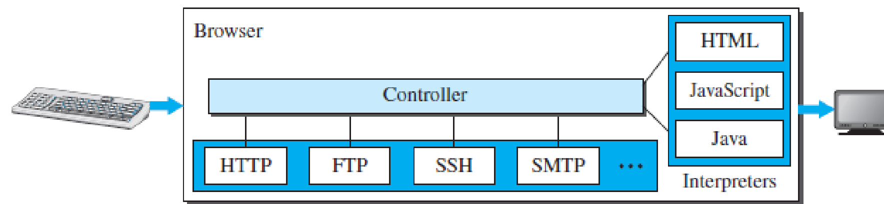

Figure 5.2: Browser

The controller receives input from the keyboard or the mouse and uses the client programs to access the document. After the document has been accessed, the controller uses one of the interpreters to display the document on the screen.

The client protocol can be one of the protocols described later, such as HTTP or FTP. The interpreter can be HTML, Java, or JavaScript, depending on the type of document. Some commercial browsers include Internet Explorer, Netscape Navigator, and Firefox.

Web Server: The web page is stored at the server. Each time a request arrives, the corresponding document is sent to the client. To improve efficiency, servers normally store requested files in a cache in memory; memory is faster to access than a disk.

A server can also become more efficient through multithreading or multiprocessing. In this case, a server can answer more than one request at a time. Some popular web servers include Apache and Microsoft Internet Information Server.

Uniform Resource Locator (URL):

A web page, as a file, needs to have a unique identifier to distinguish it from other web pages. To define a web page, we need three identifiers: *host, port,* and *path*.

However, before defining the web page, we need to tell the browser what clientserver application we want to use, which is called the *protocol*. This means we need four identifiers to define the web page.

The first is the type of vehicle to be used to fetch the web page; the last three make up the combination that defines the destination object (web page).

Protocol. The first identifier is the abbreviation for the client-server program that we need in order to access the web page.

Although most of the time the protocol is HTTP (HyperText Transfer Protocol), we can also use other protocols such as FTP (File Transfer Protocol).

Host. The host identifier can be the IP address of the server or the unique name given to the server. IP addresses can be defined in dotted decimal notation.

Port. The port, a 16-bit integer, is normally predefined for the client-server application.

Path. The path identifies the location and the name of the file in the underlying operating system. The format of this identifier normally depends on the operating system.

To combine these four pieces together, the **uniform resource locator (URL)** has been designed; it uses three different separators between the four pieces as shown below:

protocol://host/path	Used most of the time
protocol://host:port/path	Used when port number is needed

Web Documents:

The documents in the WWW can be grouped into three broad categories: static, dynamic, and active.

Static Documents:

Static documents are fixed-content documents that are created and stored in a server. The client can get a copy of the document only. In other words, the contents of the file are determined when the file is created, not when it is used.

Static documents are prepared using one of several languages: *HyperText Markup Language* (HTML), *Extensible Markup Language* (XML), *Extensible Style Language* (XSL), and *Extensible Hypertext Markup Language* (XHTML).

Dynamic Documents:

A **dynamic document** is created by a web server whenever a browser requests the document. When a request arrives, the web server runs an application program or a script that creates the dynamic document.

The server returns the result of the program or script as a response to the browser that requested the document. Because a fresh document is created for each request, the contents of a dynamic document may vary from one request to another. A very simple example of a dynamic document is the retrieval of the time and date from a server.

Active Documents:

For many applications, we need a program or a script to be run at the client site. These are called ***active documents*.** For example, suppose we want to run a program that creates animated graphics on the screen or a program that interacts with the user.

HyperText Transfer Protocol (HTTP):

The **HyperText Transfer Protocol (HTTP)** is used to define how the client-server programs can be written to retrieve web pages from the Web. An HTTP client sends a request; an HTTP server returns a response. The server uses the port number 80; the client uses a temporary port number. HTTP uses the services of TCP, which, as discussed before, is a connection-oriented and reliable protocol.

Nonpersistent versus Persistent Connections:

If the web pages, objects to be retrieved, are located on different servers, we do not have any other choice than to create a new TCP connection for retrieving each object. However, if some of the objects are located on the same server, we have two choices: to retrieve each object using a new TCP connection or to make a TCP connection and retrieve them all. The first method is referred to as a *nonpersistent connection,* the second as a *persistent connection*.

Nonpersistent Connections

In a **nonpersistent connection**, one TCP connection is made for each request/response.

The following lists the steps in this strategy:

1. The client opens a TCP connection and sends a request.

2. The server sends the response and closes the connection.

3. The client reads the data until it encounters an end-of-file marker; it then closes the connection.

Persistent Connections

HTTP version 1.1 specifies a **persistent connection** by default. In a persistent connection, the server leaves the connection open for more requests after sending a response.

The server can close the connection at the request of a client or if a time-out has been reached. The sender usually sends the length of the data with each response. However, there are some occasions when the sender does not know the length of the data.

This is the case when a document is created dynamically or actively. In these cases, the server informs the client that the length is not known and closes the connection after sending the data so the client knows that the end of the data has been reached. Time and resources are saved using persistent connections.

Only one set of buffers and variables needs to be set for the connection at each site. The round trip time for connection establishment and connection termination is saved.

Message Formats:

The HTTP protocol defines the format of the request and response messages. Each message is made of four sections. The first section in the request message is called the *request line;* the first section in the response message is called the *status line*.

The other three sections have the same names in the request and response messages. However, the similarities between these sections are only in the names; they may have different contents. We discuss each message type separately.

Request Message:

There are three fields in this line separated by one space and terminated by two characters (carriage return and line feed). The fields are called *method, URL*, and *version*.

The method field defines the request types. Several methods are defined like GET, PUT, HEAD, POST, TRACE, DELETE, etc. The URL defines the address and name of the corresponding web page. The version field gives the version of the protocol; the most current version of HTTP is 1.1.

Response Message:

A response message consists of a status line, header lines, a blank line, and sometimes a body. The first line in a response message is called the *status line*. There are three fields in this line separated by spaces and terminated by a carriage return and line feed.

The first field defines the version of HTTP protocol, currently 1.1. The status code field defines the status of the request. It consists of three digits. Whereas the codes in the 100 range are only informational, the codes in the 200 range indicate a successful request.

The codes in the 300 range redirect the client to another URL, and the codes in the 400 range indicate an error at the client site. Finally, the codes in the 500 range indicate an error at the server site.

The status phrase explains the status code in text form. After the status line, we can have zero or more *response header* lines. Each header line sends additional information from the server to the client.

Web Caching: Proxy Servers:

HTTP supports **proxy servers**. A proxy server is a computer that keeps copies of responses to recent requests. The HTTP client sends a request to the proxy server. The proxy server checks its cache.

If the response is not stored in the cache, the proxy server sends the request to the corresponding server. Incoming responses are sent to the proxy server and stored for future requests from other clients.

The proxy server reduces the load on the original server, decreases traffic, and improves latency. However, to use the proxy server, the client must be configured to access the proxy instead of the target server.

HTTP Security:

HTTP per se does not provide security. HTTP can be run over the Secure Socket Layer (SSL). In this case, HTTP is referred to as HTTPS. HTTPS provides confidentiality, client and server authentication, and data integrity.

FTP:

File Transfer Protocol (FTP) is the standard protocol provided by TCP/IP for copying a file from one host to another. Although transferring files from one system to another seems simple and straightforward, some problems must be dealt with first.

Although we can transfer files using HTTP, FTP is a better choice to transfer large files or to transfer files using different formats. Figure 5.3 shows the basic model of FTP. The client has three components: the user interface, the client control process, and the client data transfer process. The server has two components: the server control process and the server data transfer process.

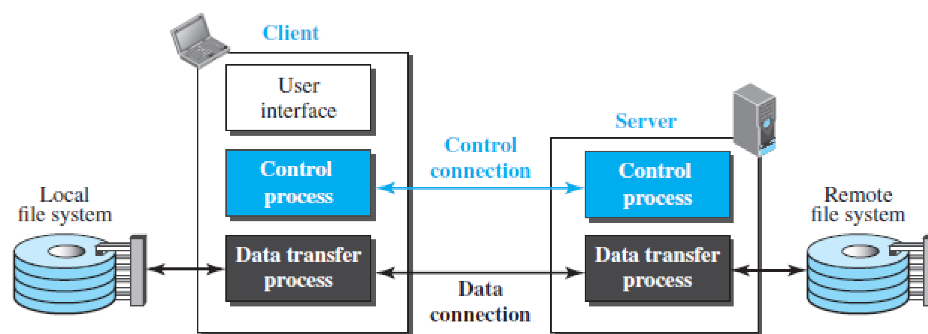

Figure 5.3: FTP

The control connection is made between the control processes. The data connection is made between the data transfer processes. Separation of commands and data transfer makes FTP more efficient. The control connection uses very simple rules of communication. We need to transfer only a line of command or a line of response at a time. The data connection, on the other hand, needs more complex rules due to the variety of data types transferred.

Two Connections

The two connections in FTP have different lifetimes. The control connection remains connected during the entire interactive FTP session. The data connection is opened and then closed for each file transfer activity.

FTP uses two well-known TCP ports: port 21 is used for the control connection, and port 20 is used for the data connection.

Control Connection:

During this control connection, commands are sent from the client to the server and responses are sent from the server to the client. Commands, which are sent from the FTP client control process, are in the form of ASCII uppercase, which may or may not be followed by an argument. Some of the most common commands are shown in table below:

Command	Argument(s)	Description
ABOR		Abort the previous command
CDUP		Change to parent directory
CWD	Directory name	Change to another directory
DELE	File name	Delete a file
LIST	Directory name	List subdirectories or files
MKD	Directory name	Create a new directory
PASS	User password	Password
PASV		Server chooses a port
PORT	Port identifier	Client chooses a port
PWD		Display name of current directory
QUIT		Log out of the system

Every FTP command generates at least one response. A response has two parts: a three-digit number followed by text. The numeric part defines the code; the text part defines needed parameters or further explanations. The first digit defines the status of the command. The second digit defines the area in which the status applies. The third digit provides additional information.

Code	Description	Code	Description
125	Data Connection Open	250	Request file action OK
150	File Status OK	331	User name OK; password is needed
200	Command OK	425	Cannot open data connection

ELECTRONIC MAIL:

Electronic mail (or e-mail) allows users to exchange messages. The nature of this application, however, is different from other applications discussed so far. In an application such as HTTP or FTP, the server program is running all the time, waiting for a request from a client. When the request arrives, the server provides the service. There is a request and there is a response.

In the case of electronic mail, the situation is different. First, e-mail is considered a one-way transaction. When Alice sends an email to Bob, she may expect a response, but this is not a mandate. Bob may or may not respond. If he does respond, it is another one-way transaction.

Second, it is neither feasible nor logical for Bob to run a server program and wait until someone sends an e-mail to him. Bob may turn off his computer when he is not using it.

This means that the idea of client/server programming should be implemented in another way: using some intermediate computers (servers). The users run only client programs when they want and the intermediate servers apply the client/server paradigm

Architecture:

To explain the architecture of e-mail, we give a common scenario as shown in Figure 5.4.

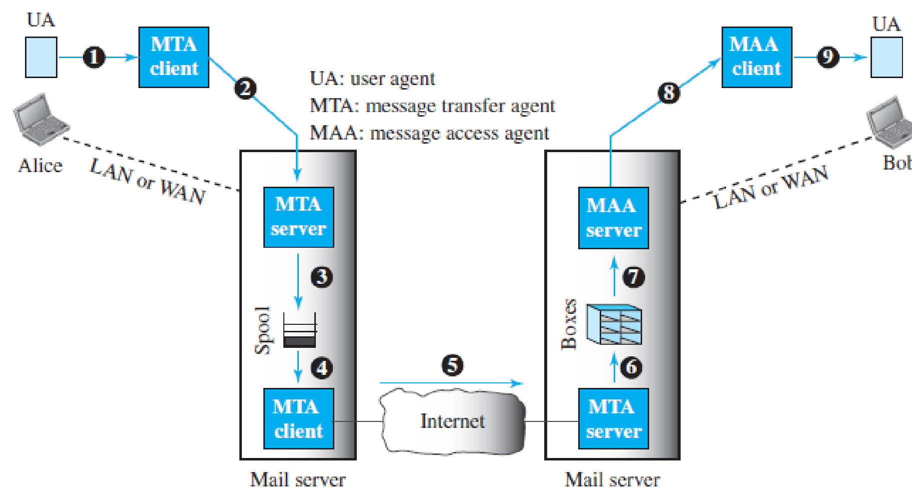

Figure 5.4: Common Scenario

In the common scenario, the sender and the receiver of the e-mail, Alice and Bob respectively, are connected via a LAN or a WAN to two mail servers. The administrator has created one mailbox for each user where the received messages are stored.

A *mailbox* is part of a server hard drive, a special file with permission restrictions. Only the owner of the mailbox has access to it. The administrator has also created a queue (spool) to store messages waiting to be sent.

A simple e-mail from Alice to Bob takes nine different steps. Alice and Bob use three different *agents*: a **user agent (UA),** a **message transfer agent (MTA),** and a **message access agent (MAA).** When Alice needs to send a message to Bob, she runs a UA program to prepare the message and send it to her mail server.

The mail server at her site uses a queue (spool) to store messages waiting to be sent. The message, however, needs to be sent through the Internet from Alice's site to Bob's site using an MTA. Here two message transfer agents are needed: one client and one server.

Like most client-server programs on the Internet, the server needs to run all the time because it does not know when a client will ask for a connection. The client, on the other hand, can be triggered by the system when there is a message in the queue to be sent.

The user agent at the Bob site allows Bob to read the received message. Bob later uses an MAA client to retrieve the message from an MAA server running on the second server.

User Agent: The first component of an electronic mail system is the **user agent (UA).** It provides service to the user to make the process of sending and receiving a message easier.

A user agent is a software package (program) that composes, reads, replies to, and forwards messages. It also handles local mailboxes on the user computers.

Message Transfer Agent: SMTP: Based on the common scenario, we can say that the e-mail is one of those applications that needs three uses of client-server paradigms to accomplish its task. It is important that we distinguish these three when we are dealing with e-mail.

The formal protocol that defines the MTA client and server in the Internet is called ***Simple Mail Transfer Protocol (SMTP).*** SMTP is used two times, between the sender and the sender's mail server and between the two mail servers. SMTP simply defines how commands and responses must be sent back and forth.

Message Access Agent: POP and IMAP: The first and second stages of mail delivery use SMTP. However, SMTP is not involved in the third stage because SMTP is a *push* protocol; it pushes the message from the client to the server.

On the other hand, the third stage needs a *pull* protocol; the client must pull messages from the server. The direction of the bulk data is from the server to the client. The third stage uses a message access agent.

Currently two message access protocols are available: Post Office Protocol, version 3 (POP3) and Internet Mail Access Protocol, version 4 (IMAP4).

POP3:

Post Office Protocol, version 3 (POP3) is simple but limited in functionality. The client POP3 software is installed on the recipient computer; the server POP3 software is installed on the mail server.

Mail access starts with the client when the user needs to download its e-mail from the mailbox on the mail server. The client opens a connection to the server on TCP port 110. It then sends its user name and password to access the mailbox. The user can then list and retrieve the mail messages, one by one.

POP3 has two modes: the *delete* mode and the *keep* mode. In the delete mode, the mail is deleted from the mailbox after each retrieval. In the keep mode, the mail remains in the mailbox after retrieval.

IMAP4:

Another mail access protocol is **Internet Mail Access Protocol, version 4 (IMAP4).** IMAP4 is similar to POP3, but it has more features; IMAP4 is more powerful and more complex.

POP3 is deficient in several ways. It does not allow the user to organize her mail on the server; the user cannot have different folders on the server. In addition, POP3 does not allow the user to partially check the contents of the mail before downloading.

IMAP4 provides the following extra functions:

- A user can check the e-mail header prior to downloading.
- A user can search the contents of the e-mail for a specific string of characters prior to downloading.
- A user can partially download e-mail. This is especially useful if bandwidth is limited and the e-mail contains multimedia with high bandwidth requirements.
- A user can create, delete, or rename mailboxes on the mail server.
- A user can create a hierarchy of mailboxes in a folder for e-mail storage.

TELNET:

A server program can provide a specific service to its corresponding client program. For example, the FTP server is designed to let the FTP client store or retrieve files on the server site. However, it is impossible to have a client/server pair for each type of service we need; the number of servers soon becomes intractable which is not scalable.

Another solution is to have a specific client/server program for a set of common scenarios, but to have some generic client/server programs that allow a user on the client site to log into the computer at the server site and use the services available there.

For example, if a student needs to use the Java compiler program at her university lab, there is no need for a Java compiler client and a Java compiler server. The student can use a client logging program to log into the university server and use the compiler program at the university. We refer to these generic client/server pairs as ***remote logging*** applications.

One of the original remote logging protocols is **TELNET,** which is an abbreviation for *TErminaL NETwork.* Although TELNET requires a logging name and password, it is vulnerable to hacking because it sends all data including the password in plaintext (not encrypted).

A hacker can eavesdrop and obtain the logging name and password. Because of this security issue, the use of TELNET has diminished in favor of another protocol, Secure Shell (SSH).

Although TELNET is almost replaced by SSH, we briefly discuss TELNET here for two reasons:

1. The simple plaintext architecture of TELNET allows us to explain the issues and challenges related to the concept of remote logging, which is also used in SSH when it serves as a remote logging protocol.

2. Network administrators often use TELNET for diagnostic and debugging purposes.

Local versus Remote Logging:

When a user logs into a local system, it is called *local logging*. As a user types at a terminal or at a workstation running a terminal emulator, the keystrokes are accepted by the terminal driver.

The terminal driver passes the characters to the operating system. The operating system, in turn, interprets the combination of characters and invokes the desired application program or utility.

However, when a user wants to access an application program or utility located on a remote machine, she performs *remote logging.* Here the TELNET client and server programs come into use. The user sends the keystrokes to the terminal driver where the local operating system accepts the characters but does not interpret them.

The characters are sent to the TELNET client, which transforms the characters into a universal character set called *Network Virtual Terminal* (NVT) characters and delivers them to the local TCP/IP stack.

The commands or text, in NVT form, travel through the Internet and arrive at the TCP/IP stack at the remote machine. Here the characters are delivered to the operating system and passed to the TELNET server, which changes the characters to the corresponding characters understandable by the remote computer.

However, the characters cannot be passed directly to the operating system because the remote operating system is not designed to receive characters from a TELNET server; it is designed to receive characters from a terminal driver.

The solution is to add a piece of software called a *pseudoterminal driver*, which pretends that the characters are coming from a terminal. The operating system then passes the characters to the appropriate application program.

NVT uses two sets of characters, one for data and one for control. Both are 8-bit bytes. For data, NVT normally uses what is called *NVT ASCII*. This is an 8-bit character set in which the seven lowest order bits are the same as US ASCII and the highest order bit is 0.

To send control characters between computers (from client to server or vice versa), NVT uses an 8-bit character set in which the highest order bit is set to 1.

Options: TELNET lets the client and server negotiate options before or during the use of the service.

User Interface:

The operating system (UNIX, for example) defines an interface with user-friendly commands. An example of such a set of commands can be found in Table beow:

Command Name	**Meaning**
open	Connect to a remote computer
close	Close the connections
display	Show the operating parameters
mode	Change to line or character mode
Quit	Exit TELNET
send	Send special characters

SECURE SHELL (SSH):

Although **Secure Shell (SSH)** is a secure application program that can be used today for several purposes such as remote logging and file transfer, it was originally designed to replace TELNET.

There are two versions of SSH: SSH-1 and SSH-2, which are totally incompatible. The first version, SSH-1, is now deprecated because of security flaws in it. In this section, we discuss only SSH-2.

Components: SSH is an application-layer protocol with three components.

SSH Transport-Layer Protocol (SSH-TRANS):

Since TCP is not a secured transport-layer protocol, SSH first uses a protocol that creates a secured channel on top of the TCP. This new layer is an independent protocol referred to as SSH-TRANS.

When the procedure implementing this protocol is called, the client and server first use the TCP protocol to establish an insecure connection. Then they exchange several security parameters to establish a secure channel on top of the TCP. The services provided by this protocol are:

1. Privacy or confidentiality of the message exchanged.
2. Data integrity, which means that it is guaranteed that the messages exchanged between the client and server are not changed by an intruder.
3. Server authentication, which means that the client is now sure that the server is the one that it claims to be.
4. Compression of the messages, which improves the efficiency of the system and makes attack more difficult.

SSH Authentication Protocol (SSH-AUTH):

After a secure channel is established between the client and the server and the server is authenticated for the client, SSH can call another procedure that can authenticate the client for the server. The client authentication process in SSH is very similar to what is done in Secure Socket Layer (SSL).

This layer defines a number of authentication tools similar to the ones used in SSL. Authentication starts with the client, which sends a request message to the server. The request includes the user name, server name, the method of authentication, and the required data. The server responds with either a success message, which confirms that the client is authenticated, or a failed message, which means that the process needs to be repeated with a new request message.

SSH Connection Protocol (SSH-CONN):

After the secured channel is established and both server and client are authenticated for each other, SSH can call a piece of software that implements the third protocol, SSHCONN.

One of the services provided by the SSH-CONN protocol is multiplexing. SSH-CONN takes the secure channel established by the two previous protocols and lets the client create multiple logical channels over it. Each channel can be used for a different purpose, such as remote logging, file transfer, and so on.

Applications:

Although SSH is often thought of as a replacement for TELNET, SSH is, in fact, a general-purpose protocol that provides a secure connection between a client and server.

SSH for Remote Logging:

Several free and commercial applications use SSH for remote logging. Among them, we can mention PuTTy, by Simon Tatham, which is a client SSH program that can be used for remote logging. Another application program is Tectia, which can be used on several platforms.

SSH for File Transfer:

One of the application programs that is built on top of SSH for file transfer is the *Secure File Transfer Program* (*sftp*). The *sftp* application program uses one of the channels provided by the SSH to transfer files. Another common application is called *Secure Copy* (*scp*). This application uses the same format as the UNIX copy command, *cp,* to copy files.

DOMAIN NAME SYSTEM (DNS):

Since the Internet is so huge today, a central directory system cannot hold all the mapping. In addition, if the central computer fails, the whole communication network will collapse.

A better solution is to distribute the information among many computers in the world. In this method, the host that needs mapping can contact the closest computer holding the needed information. This method is used by the **Domain Name System (DNS)**.

Figure 5.5 shows how TCP/IP uses a DNS client and a DNS server to map a name to an address. A user wants to use a file transfer client to access the corresponding file transfer server running on a remote host. The user knows only the file transfer server name, such as *afilesource.com.*

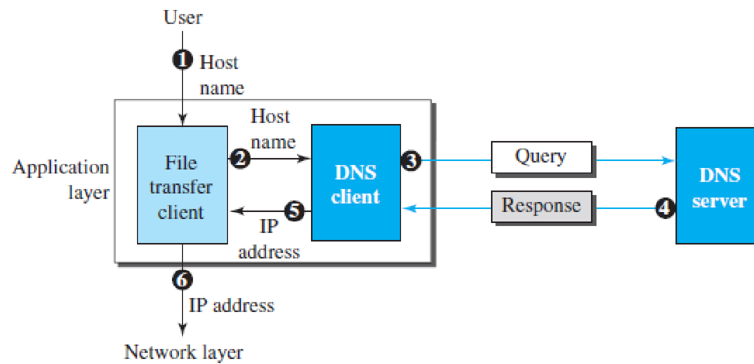

Figure 5.5: Purpose of DNS

Name Space:

A **name space** that maps each address to a unique name can be organized in two ways: flat or hierarchical. In a *flat name space,* a name is assigned to an address.

A name in this space is a sequence of characters without structure. The names may or may not have a common section; if they do, it has no meaning. The main disadvantage of a flat name space is that it cannot be used in a large system such as the Internet because it must be centrally controlled to avoid ambiguity and duplication.

In a *hierarchical name space,* each name is made of several parts. The first part can define the nature of the organization, the second part can define the name of an organization, the third part can define departments in the organization, and so on. In this case, the authority to assign and control the name spaces can be decentralized.

A central authority can assign the part of the name that defines the nature of the organization and the name of the organization. The responsibility for the rest of the name can be given to the organization itself.

The organization can add suffixes (or prefixes) to the name to define its host or resources. The management of the organization need not worry that the prefix chosen for a host is taken by another organization because, even if part of an address is the same, the whole address is different.

Domain Name Space:

To have a hierarchical name space, a **domain name space** was designed. In this design the names are defined in an inverted-tree structure with the root at the top. The tree can have only 128 levels: level 0 (root) to level 127 (see Figure 5.6).

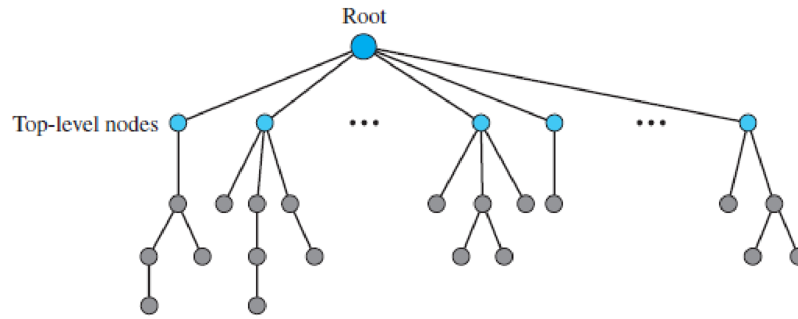

Figure 5.6: Domain name space

Label:

Each node in the tree has a **label**, which is a string with a maximum of 63 characters. The root label is a null string (empty string). DNS requires that children of a node (nodes that branch from the same node) have different labels, which guarantees the uniqueness of the domain names.

Domain Name:

Each node in the tree has a domain name. A full **domain name** is a sequence of labels separated by dots (**.**). The domain names are always read from the node up to the root.

The last label is the label of the root (null). This means that a full domain name always ends in a null label, which means the last character is a dot because the null string is nothing. Figure 5.7 shows some domain names.

Domain:

A **domain** is a subtree of the domain name space. The name of the domain is the name of the node at the top of the subtree. Figure 5.8 shows some domains. Note that a domain may itself be divided into domains.

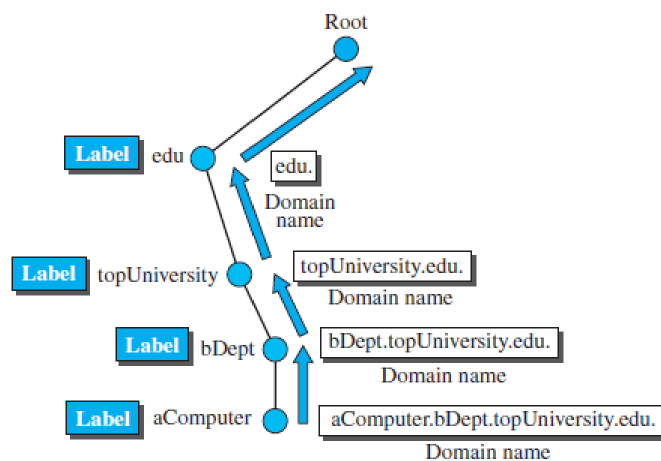

Figure 5.7: Domain names and labels

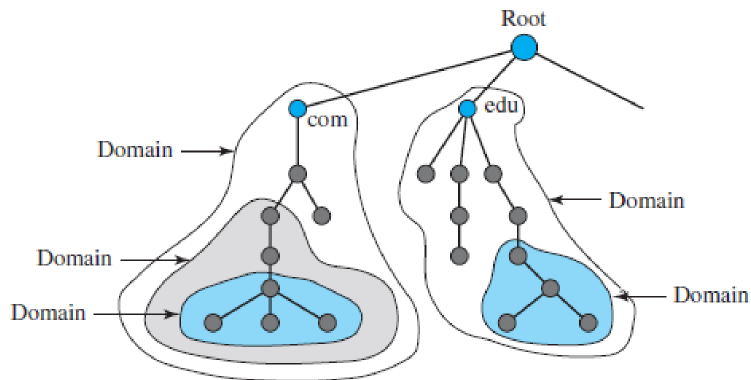

Figure 5.8: Domains

SNMP:

Several network management standards have been devised during the last few decades. The most important one is **Simple Network Management Protocol (SNMP),** used by the Internet.

SNMP is a framework for managing devices in an internet using the TCP/IP protocol suite. It provides a set of fundamental operations for monitoring and maintaining an internet. SNMP uses the concept of manager and agent. That is, a manager, usually a host, controls and monitors a set of agents, usually routers or servers (see Figure 5.9).

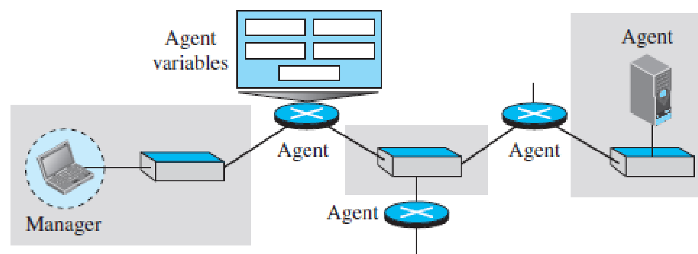

Figure 5.9: SNMP concept

SNMP is an application-level protocol in which a few manager stations control a set of agents. The protocol is designed at the application level so that it can monitor devices made by different manufacturers and installed on different physical networks.

In other words, SNMP frees management tasks from both the physical characteristics of the managed devices and the underlying networking technology. It can be used in a heterogeneous internet made of different LANs and WANs connected by routers made by different manufacturers.

Managers and Agents: A management station, called a *manager,* is a host that runs the SNMP client program. A managed station, called an *agent,* is a router (or a host) that runs the SNMP server program.

Management is achieved through simple interaction between a manager and an agent. The agent keeps performance information in a database. The manager has access to the values in the database.

For example, a router can store in appropriate variables the number of packets received and forwarded. The manager can fetch and compare the values of these two variables to see if the router is congested or not.

The manager can also make the router perform certain actions. For example, a router periodically checks the value of a reboot counter to see when it should reboot itself. It reboots itself, for example, if the value of the counter is 0. The manager can use this feature to reboot the agent remotely at any time. It simply sends a packet to force a 0 value in the counter.

Agents can also contribute to the management process. The server program running on the agent can check the environment and, if it notices something unusual, it can send a warning message (called a *Trap*) to the manager. In other words, management with SNMP is based on three basic ideas:

1. A manager checks an agent by requesting information that reflects the behavior of the agent.
2. A manager forces an agent to perform a task by resetting values in the agent database.
3. An agent contributes to the management process by warning the manager of an unusual situation.

Management Components: To do management tasks, SNMP uses two other protocols: **Structure of Management Information (SMI)** and **Management Information Base (MIB).**

Role of SNMP: SNMP has some very specific roles in network management. It defines the format of the packet to be sent from a manager to an agent and vice versa. It also interprets the result and creates statistics (often with the help of other management software).

Role of SMI: To use SNMP, we need rules for naming objects. This is particularly important because the objects in SNMP form a hierarchical structure. Part of a name can be inherited from the parent. We also need rules to define the types of objects.

Role of MIB: MIB creates a set of objects defined for each entity in a manner similar to that of a database (mostly metadata in a database, names & types without values).