# KARPAGAM ACADEMY OF HIGHER EDUCATION

(Deemed to be University )

(Established under Section 3 of UGC Act,1956)

**Department of Computer Sciece, Computer Appication & Information Technology**

**Syllabus**

**Semester – V**

**16ITU502B NETWORK PROGRAMMING 4H – 4C**

---

**Instruction Hours / week: L: 4 T: 0 P: 0** **Marks:** Int : **40** Ext : **60** Total: **100**

**SCOPE**

This course is to master the fundamentals of communications networks by gaining a working knowledge of data transmission concepts, understanding the operation of all seven layers of OSI Model and the protocols used in each layer.

**OBJECTIVES**

- Various transmission media, their comparative study, fiber optics and wireless media
- Categories and topologies of networks (LAN and WAN) Layered architecture (OSI and TCP/IP) and protocol suites.
- TCP, UDP, SCTP protocols Ethernet and LAN administration.
- Details of IP operations in the INTERNET and associated routing principles

**UNIT-I**

Transport Layer Protocols: TCP, UDP, SCTP protocol.

**UNIT-II**

Socket Programming: Socket Introduction; TCP Sockets; TCP Client/Server Example ; signal handling

**UNIT-III**

I/O multiplexing using sockets; Socket Options; UDP Sockets; UDP client server example; Address lookup using sockets.

**UNIT-IV**

Network Applications: Remote logging; Email; WWW and HTTP.

**UNIT-V**

LAN administration: Linux and TCP/IP networking: Network Management and Debugging.

**Suggested Readings**

1. Richard Stevens, W., Bill Fenner., & Andrew, M. Rudoff. (2003). Unix Network Programming, The sockets Networking API, Vol. 1(3rd ed.). New Delhi: PHI.
2. Forouzan, B. A. (2003). Data Communications and Networking(4th ed.). New Delhi: THM Publishing Company Ltd.,
3. Nemeth Synder., & Hein. (2010). Linux Administration Handbook (2nd ed.), New Delhi: Pearson Education.
4. Steven, R. (1990). Unix Network Programming (2nd ed.). New Delhi: PHI.

**Web References**

1. https://www.tutorialspoint.com/unix_sockets/
2. https://www.udemy.com/learn-socket-programming-in-c-from-scratch/
3. https://www.binarytides.com/socket-programming-c-linux-tutorial/
4. https://www.networkmanagementsoftware.com/snmp-tutorial/
5. https://en.wikipedia.org/wiki/NetFlow

| **ESE Pattern** | | |
|---|---|---|
| 1 | Part - A | 20 X 1 = 20 Marks |
| 2 | Part - B | 5 X 2 = 10 Marks |
| 3 | Part – C | 5 X 6 = 30 Marks |
| **4** | **Total** | **60 Marks** |

# KARPAGAM ACADEMY OF HIGHER EDUCATION

**(Deemed to be University)**
**Established Under Section 3 of UGC Act, 1956)**
**Coimbatore – 641021, INDIA**
**Department of Computer Science, Applications & Information Technology**

**Subject Name: Computer Graphics** **Subject Code: 16ITU501A**

**Semester: V** **Class: III Bsc. IT**

**Staff: B.Bharathi**

| S.No | Topics | No. of Periods Required | Reference Materials |
|---|---|---|---|
| | **UNIT-I - Introduction** | | |
| 1 | Basic Elements of Computer Graphics: Refresh Cathode Ray Tubes, Raster Scan Displays | 1hr | SR2: 56-62 |
| 2 | Color CRT monitor, Direct View Storage Tubes, Flat panel display | 1hr | SR2:62-69 |
| 3 | Three Dimensional Viewing Devices, Stereoscopic and Virtual Reality Systems, LCD Display Unit, LED Display Unit | 1hr | SR2: 69-73¸ W1, W2, W3 |
| 4 | Applications of Graphics: Computer Aided Design | 1hr | SR2: 24-31 |
| 5 | Presentation Graphics, Computer Art | 1hr | SR2: 31-38 |
| 6 | Entertainment, Education & Training | 1hr | SR2: 38-45 |
| 7 | Visualization, Image Processing, Graphical User Interface | 1hr | SR2:45-55 |
| 8 | Recapitulation of Important Topics | 1hr | - |
| | **Total Hours** | **8 hrs** | |

**Suggested Reading**

**SR1:** J.D., Foley, Van Dan, A., & Feiner Hughes. (1990). Computer Graphics Principles & Practice (2nd ed.). Addison Wesley.

**SR2:** Hearn D. Baker. (2008). Computer Graphics. New Delhi: Prentice Hall of India.

**SR3:** Rogers, D.F. (1997). Procedural Elements for Computer Graphics. McGraw Hill.

**SR4:** Rogers, D.F. Adams. (1989). Mathematical Elements for Computer Graphics (2nd ed.). McGraw Hill.

**Web References**

**W1:**www.explainthatstuff.com/lcdtv.html

**W2:** www.explainthatstuff.com/how-oleds-and-leps-work.html

**W3:** www.electricalbasicprojects.com/led-vs-lcd-difference-between-led-and-lcd/

**Unit – II - Graphics Hardware**

| | | | |
|---|---|---|---|
| 1 | Architecture of Raster Scan Display: Video Controller, Display Processor | 1hr | SR2:73-76 |
| 2 | Random Scan System, Graphics monitor and Workstations | 1hr | SR2:76-80 |
| 3 | Raster & Random Scan Display Units | 1hr | SR2: 60-62 |
| 4 | Input & Output Devices: Keyboard, Mouse, Trackball & Space ball, Joystick | 1hr | SR2:80-84,W3 |
| 5 | Input & Output Devices: Data Glove, Digitizer, Image Scanner, Touch-panel, Light pen | 1hr | SR2:84-90,W3 |
| 6 | Input & Output Devices: Voice System, Hard-copy Devices | 1hr | SR2:90-95 |
| 7 | Graphics Software, Coordinate Representation, Graphic Functions, Software Standards, PHIGS Workstation | 1hr | SR2:95-99 |
| 8 | Recapitulation of Important Topics | 1hr | SR2:60-62 |
| | **Total Hours** | **8 hrs** | |

**Suggested Reading**

**SR1:** J.D., Foley, Van Dan, A., & Feiner Hughes. (1990). Computer Graphics Principles & Practice (2nd ed.). Addison Wesley.

**SR2:** Hearn D. Baker. (2008). Computer Graphics. New Delhi: Prentice Hall of India.

**SR3:** Rogers, D.F. (1997). Procedural Elements for Computer Graphics. McGraw Hill.

**SR4:** Rogers, D.F. Adams. (1989). Mathematical Elements for Computer Graphics (2nd ed.). McGraw Hill.

**Web References**

**W1:** https://www.tutorialspoint.com/computer_graphics/computer_graphics_basics.htm

**W2:** http://studentstudyhub.com/raster-scan-system-vs-random-scan-system/

**W3:** https://allaboutbasic.com/

## Unit - III Fundamental Techniques in Graphics

| | | | |
|---|---|---|---|
| 1 | Raster Scan Line, Raster Scan Circles, Ellipse | 1hr | SR1: 72-91,W1 |
| 2 | Filling polygon | 1hr | SR1:92-99 |
| 3 | Thick Primitive | 1hr | SR1:104-109 |
| 4 | Clipping Lines, Clipping Polygon | 1hr | SR1:111-127 |
| 5 | 2D: Transformation, Homogeneous Coordinates and Matrix Representation of 2D Transformation | 1hr | SR1:201-208,W2 |
| 6 | Composition of 2D Transformation, Window to View point Transformation, Efficiency, Matrix Representation of 3D Transformation | 1hr | SR1:208-217 |
| 7 | 3D: Projection, Specifying an Arbitrary 3D Viewing | 1hr | SR1:230-242,W3 |
| 8 | Planar Geometric Projection | 1hr | SR1:253-258 |
| 9 | Implementing Planar Geometric Projection | 1hr | SR1:258-279 |
| 10 | Recapitulation of Important Topics | 1hr | - |
| | **Total Hours** | **10 hrs** | |

**Suggested Reading**

**SR1:** J.D., Foley, Van Dan, A., & Feiner Hughes. (1990). Computer Graphics Principles & Practice (2nd ed.). Addison Wesley.

**SR2:** Hearn D. Baker. (2008). Computer Graphics. New Delhi: Prentice Hall of India.

**SR3:** Rogers, D.F. (1997). Procedural Elements for Computer Graphics. McGraw Hill.

**SR4:** Rogers, D.F. Adams. (1989). Mathematical Elements for Computer Graphics (2$^{nd}$ ed.). McGraw Hill.

**Web References**

**W1: :** https://www.tutorialspoint.com/computer_graphics

**W2:** http://www.it.hiof.no/~borres/j3d/math/twod/p-twod.html

**W3:** http://www.cs.iusb.edu/~danav/teach/c481/c481_07_intro3d.html

## UNIT-IV - Geometric Modeling

| | | | |
|---|---|---|---|
| 1 | Filling Polygon | 1hr | SR1:92-99 |
| 2 | Clipping Polygon | 1hr | SR1:124-127 |
| 3 | Polygon Meshes | 1hr | SR1:473-478 |
| 4 | Parametric Cubic Curves | 1hr | SR1:478-516,W1,W2 |
| 5 | Parametric Cubic Curves | 1hr | |
| 6 | Parametric Bi-cubic Surface | 1hr | SR1:516-528,W3 |
| 7 | Parametric Bi-cubic Surface | 1hr | |
| 8 | Quadric Surface | 1hr | SR1:528-529 |
| 9 | Recapitulation of Important Topics | 1hr | - |
| | **Total Hours** | **9 hrs** | |

**Suggested Reading**

**SR1:** J.D., Foley, Van Dan, A., & Feiner Hughes. (1990). Computer Graphics Principles & Practice (2nd ed.). Addison Wesley.

**SR2:** Hearn D. Baker. (2008). Computer Graphics. New Delhi: Prentice Hall of India.

**SR3:** Rogers, D.F. (1997). Procedural Elements for Computer Graphics. McGraw Hill.

**SR4:** Rogers, D.F. Adams. (1989). Mathematical Elements for Computer Graphics (2$^{nd}$ ed.). McGraw Hill.

**Web References**

**W1:** http://nptel.ac.in/courses/112102101/45

**W2:** https://www.tutorialspoint.com/computer_graphics/computer_graphics_curves.htm

**W3:** https://www.davidsalomon.name/CaS/CaS.html

| **UNIT-V - Virtual Surface Determination** | | | |
|---|---|---|---|
| 1 | Function of two Variables, Techniques of Efficient Visible-surface Algorithm, Algorithm for Visible-Line Determination | 1hr | SR1:651-658 |
| 2 | The Z-Buffer, List-Priority, Scan-Line Algorithm | 1hr | SR1:668-686 |
| 3 | Area Sub-Division Algorithm, Octrees | 1hr | SR1:686-698 |
| 4 | Illumination Model | 1hr | SR1:721-734 |
| 5 | Shading model of polygon, Surface Details, Shadows | 1hr | SR1:734-754 |
| 6 | Transparency | 1hr | SR1:754-758 |
| 7 | Inter-object Reflections, Illumination Models | 1hr | SR1:758-772 |
| 8 | Extending Light Sources, Spectra Sampling, Global Illumination Algorithm | 1hr | SR1:772-776 |
| 9 | Recursive Ray Tracing | 1hr | SR1:776-793 |
| 10 | Chromatic Color, Color Model for Graphics | 1hr | SR1:574-599 |
| 11 | Recapitulation of Important Topics | 1hr | - |
| 12 | Discussion of previous ESE QP | 1hr | - |
| 13 | Discussion of previous ESE QP | 1hr | - |
| | **Total Hours** | **13 hrs** | |
| | **Total Number of periods (10+10+13+11+16)** | **48 hrs** | |

**Suggested Reading**

**SR1:** J.D., Foley, Van Dan, A., & Feiner Hughes. (1990). Computer Graphics Principles & Practice (2nd ed.). Addison Wesley.

**SR2:** Hearn D. Baker. (2008). Computer Graphics. New Delhi: Prentice Hall of India.

**SR3:** Rogers, D.F. (1997). Procedural Elements for Computer Graphics. McGraw Hill.

**SR4:** Rogers, D.F. Adams. (1989). Mathematical Elements for Computer Graphics ($2^{nd}$ ed.). McGraw Hill.

**Web References**

**W1:** https://www.scripttutorial.com

**W2:** https://www.sketchpad.in

**W3:** https://www.tutorialspoint.com

## UNIT-I

## SYLLABUS

**Transport Layer Protocols:** TCP, UDP, SCTP protocol

### 1. Transport Layer Protocols

The transport layer is responsible for process-to-process delivery of the entire message. A process is an application program running on a host. Whereas the network layer oversees source-to-destination delivery of individual packets, it does not recognize any relationship between those packets. It treats each one independently, as though each piece belonged to a separate message, whether or not it does. The transport layer, on the other hand, ensures that the whole message arrives intact and in order, overseeing both error control and flow control at the source-to-destination level.

Computers often run several programs at the same time. For this reason, source to-destination delivery means delivery not only from one computer to the next but also from a specific process on one computer to a specific process on the other. The transport layer header must therefore include a type of address called a *service-point address* in the OSI model and port number or port addresses in the Internet and TCP/IP protocol suite.

A transport layer protocol can be either ***connectionless or connection-oriented***. A connectionless transport layer treats each segment as an independent packet and delivers it to the transport layer at the destination machine. A connection-oriented transport layer makes a connection with the transport layer at the destination machine first before delivering the packets. After all the data is transferred, the connection is terminated. In the transport layer, a message is normally divided into transmittable segments. A connectionless protocol, such as UDP, treats each segment separately.

A connection oriented protocol, such as TCP and SCTP, creates a relationship between the segments using sequence numbers. Like the data link layer, the transport layer may be responsible for flow and error control. However, flow and error control at this layer is performed end to end rather than across a single link. On the other hand, the other two protocols, TCP and SCTP, use sliding windows for flow control and an acknowledgment system for error control

### 2. TCP

The second transport layer protocol we are going to discuss is called **Transmission Control Protocol (TCP**). TCP, like UDP, is a process-to-process (program-to-program) protocol. TCP, therefore, like UDP, uses port numbers.

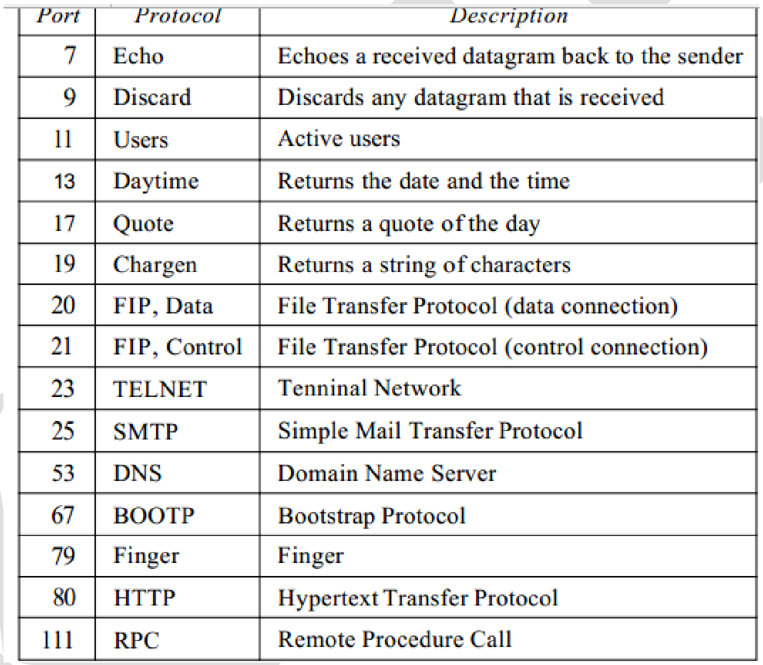
Unlike UDP, TCP is a connection oriented protocol; it creates a virtual connection between two TCPs to send data. In addition, TCP uses flow and error control mechanisms at the transport level. In brief, TCP is called a *connection-oriented, reliable* transport protocol. It adds connection-oriented and reliability features to the services of IP.

**TCP Services**

Before we discuss TCP in detail, let us explain the services offered by TCP to the processes at the application layer.

***Process-to-Process Communication*** TCP provides process-to-process communication using port numbers. Table 1 lists some well-known port numbers used by TCP.

**Table 1: Port numbers used by TCP**



| *Port* | *Protocol* | *Description* |
|---|---|---|
| 7 | Echo | Echoes a received datagram back to the sender |
| 9 | Discard | Discards any datagram that is received |
| 11 | Users | Active users |
| 13 | Daytime | Returns the date and the time |
| 17 | Quote | Returns a quote of the day |
| 19 | Chargen | Returns a string of characters |
| 20 | FIP, Data | File Transfer Protocol (data connection) |
| 21 | FIP, Control | File Transfer Protocol (control connection) |
| 23 | TELNET | Tenninal Network |
| 25 | SMTP | Simple Mail Transfer Protocol |
| 53 | DNS | Domain Name Server |
| 67 | BOOTP | Bootstrap Protocol |
| 79 | Finger | Finger |
| 80 | HTTP | Hypertext Transfer Protocol |
| 111 | RPC | Remote Procedure Call |

***Stream Delivery Service***

TCP is a stream-oriented protocol. In UDP, a process (an application program) sends messages, with predefined boundaries, to UDP for delivery. UDP adds its own header to each of these messages and delivers them to IP for transmission. Each message from the process is calIed a user datagram and becomes, eventually, one IP datagram. Neither IP nor UDP recognizes any relationship between the datagrams.

TCP, on the other hand, allows the sending process to deliver data as a stream of bytes and allows the receiving process to obtain data as a stream of bytes. TCP creates an environment in which the two processes seem to be connected by an imaginary "tube" that carries their data across the Internet.

This imaginary environment is depicted in Figure 1. The sending process produces (writes to) the stream of bytes, and the receiving process consumes (reads from) them.

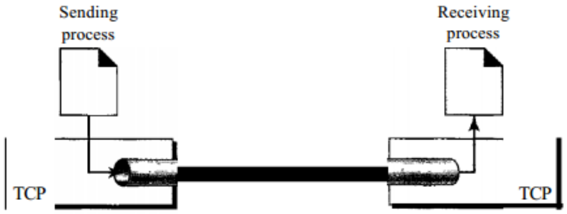


**Fig.1: Stream delivery**

**Sending and Receiving Buffers**

Because the sending and the receiving processes may not write or read data at the same speed, TCP needs buffers for storage. There are two buffers, the sending buffer and the receiving buffer, one for each direction. One way to implement a buffer is to use a circular array of I-byte locations as shown in Figure 2. For simplicity, we have shown two buffers of 20 bytes each; normally the buffers are hundreds or thousands of bytes, depending on the implementation. We also show the buffers as the same size, which is not always the case.

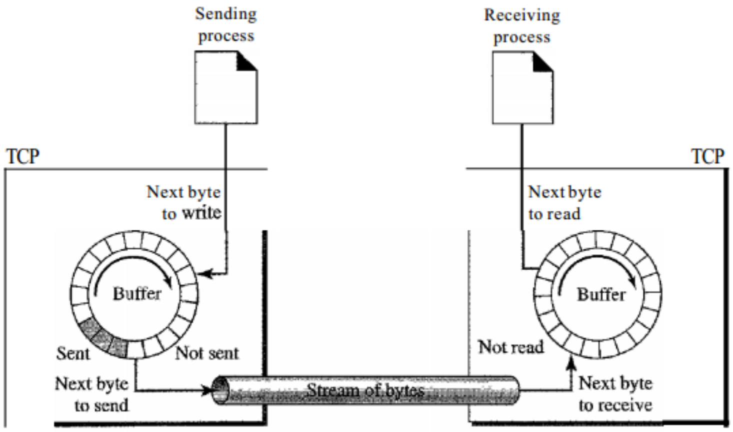


**Fig.2: Sending and receiving buffers**

Figure 2 shows the movement of the data in one direction. At the sending site, the buffer has three types of chambers. The white section contains empty chambers that can be filled by the sending process (producer). The gray area holds bytes that have been sent but not yet acknowledged. TCP keeps these bytes in the buffer until it receives an acknowledgment. The colored area contains bytes to be sent by the sending TCP.

However, as we will see later in this chapter, TCP may be able to send only part of this colored section. This could be due to the slowness of the receiving process or perhaps to congestion in the network. Also note that after the bytes in the gray chambers are acknowledged, the chambers are recycled and available for use by the sending process. This is why we show a circular buffer.

The operation of the buffer at the receiver site is simpler. The circular buffer is divided into two areas (shown as white and colored). The white area contains empty chambers to be filled by bytes received from the network. The colored sections contain received bytes that can be read by the receiving process. When a byte is read by the receiving process, the chamber is recycled and added to the pool of empty chambers.

**Segments:** Although buffering handles the disparity between the speed of the producing and consuming processes, we need one more step before we can send data. The IP layer, as a service provider for TCP, needs to send data in packets, not as a stream of bytes. At the transport layer, TCP groups a number of bytes together into a packet called a segment. TCP adds a header to each segment (for control purposes) and delivers the segment to the IP layer for transmission. The segments are encapsulated in IP datagrams and transmitted. This entire operation is transparent to the receiving process. Later we will see that segments may be received out of order, lost, or corrupted and resent. All these are handled by TCP with the receiving process unaware of any activities. Figure 3 shows how segments are created from the bytes in the buffers.

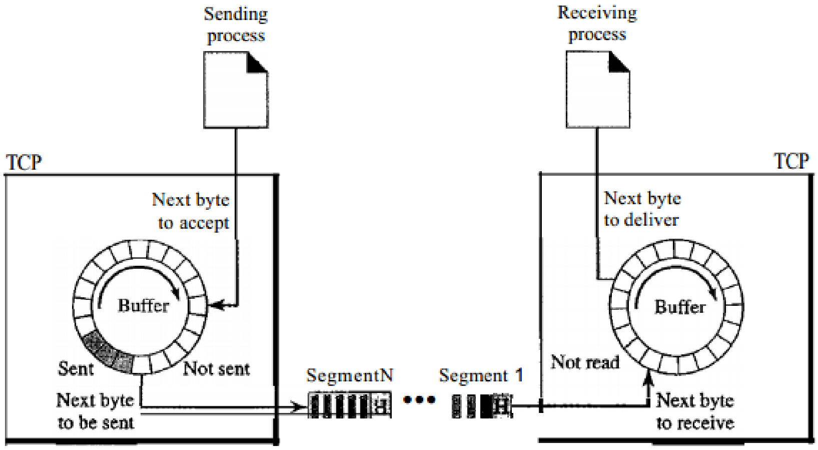


**Fig.3: TCP Segments**

Note that the segments are not necessarily the same size. In Figure 3, for simplicity, we show one segment carrying 3 bytes and the other carrying 5 bytes. In reality, segments carry hundreds, if not thousands, of bytes.

***Full-Duplex Communication***

TCP offers full-duplex service, in which data can flow in both directions at the same time. Each TCP then has a sending and receiving buffer, and segments move in both directions.

***Connection-Oriented Service***

TCP, unlike UDP, is a connection-oriented protocol. When a process at site A wants to send and receive data from another process at site B, the following occurs:

1. The two TCPs establish a connection between them.
2. Data are exchanged in both directions.
3. The connection is terminated.

Note that this is a virtual connection, not a physical connection. The TCP segment is encapsulated in an IP datagram and can be sent out of order, or lost, or corrupted, and then resent. Each may use a different path to reach the destination. There is no physical connection. TCP creates a stream-oriented environment in which it accepts the responsibility of delivering the bytes in order to the other site. The situation is similar to creating a bridge that spans multiple islands and passing all the bytes from one island to another in one single connection.

***Reliable Service*** TCP is a reliable transport protocol. It uses an acknowledgment mechanism to check the safe and sound arrival of data. We will discuss this feature further in the section on error control.

**TCP Features**

To provide the services mentioned in the previous section, TCP has several features that are briefly summarized in this section and discussed later in detail.

***Numbering System*** Although the TCP software keeps track of the segments being transmitted or received, there is no field for a segment number value in the segment header. Instead, there are two fields called the sequence number and the acknowledgment number. These two fields refer to the byte number and not the segment number.

***Byte Number*** TCP numbers all data bytes that are transmitted in a connection. Numbering is independent in each direction. When TCP receives bytes of data from a process, it stores them in the sending buffer and numbers them. The numbering does not necessarily start from O. Instead, TCP generates a random number between 0 and 232 - 1 for the number of the first byte. For example, if the random number happens to be 1057 and the total data to be sent are 6000 bytes, the bytes are numbered from 1057 to 7056. We will see that byte numbering is used for flow and error control.

***Sequence Number*** After the bytes have been numbered, TCP assigns a sequence number to each segment that is being sent. The sequence number for each segment is the number of the first byte carried in that segment.

***Flow Control***

TCP, unlike UDP, provides *flow control*. The receiver of the data controls the amount of data that are to be sent by the sender. This is done to prevent the receiver from being overwhelmed with data. The numbering system allows TCP to use a byte-oriented flow control.

***Error Control***

To provide reliable service, TCP implements an error control mechanism. Although error control considers a segment as the unit of data for error detection (loss or corrupted segments), error control is byte-oriented, as we will see later.

***Congestion Control***

TCP, unlike UDP, takes into account congestion in the network. The amount of data sent by a sender is not only controlled by the receiver (flow control), but is also determined by the level of congestion in the network.

**Segment** Before we discuss TCP in greater detail, let us discuss the TCP packets themselves. A packet in TCP is called a segment.

***Format*** The format of a segment is shown in Figure 4

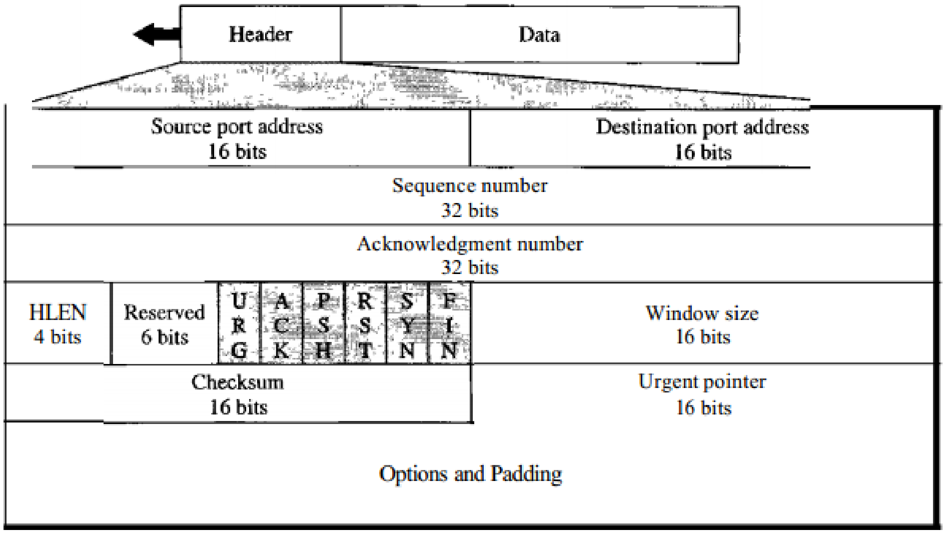


**Fig.4: TCP Segment Format**

The segment consists of a 20- to 60-byte header, followed by data from the application program. The header is 20 bytes if there are no options and up to 60 bytes if it contains options. We will discuss some of the header fields in this section.

***Source port address:*** This is a 16-bit field that defines the port number of the application program in the host that is sending the segment. This serves the same purpose as the source port address in the UDP header.

***Destination port address:*** This is a 16-bit field that defines the port number of the application program in the host that is receiving the segment. This serves the same purpose as the destination port address in the UDP header.

***Sequence number*** This 32-bit field defines the number assigned to the first byte of data contained in this segment. As we said before, TCP is a stream transport protocol. To ensure connectivity, each byte to be transmitted is numbered. The sequence number tells the destination which byte in this sequence comprises the first byte in the segment. During connection establishment, each party uses a random number generator to create an initial sequence number (ISN), which is usually different in each direction.

***Acknowledgment number:*** This 32-bit field defines the byte number that the receiver of the segment is expecting to receive from the other party. If the receiver of the segment has successfully received byte number *x* from the other party, it defines *x* + I as the acknowledgment number. Acknowledgment and data can be piggybacked together.

***Header length:*** This 4-bit field indicates the number of 4-byte words in the TCP header. The length of the header can be between 20 and 60 bytes. Therefore, the value of this field can be between 5 (5 x 4 =20) and 15 (15 x 4 =60).

***Reserved:*** This is a 6-bit field reserved for future use.

***Control:*** This field defines 6 different control bits or flags as shown in Figure 5. One or more of these bits can be set at a time.

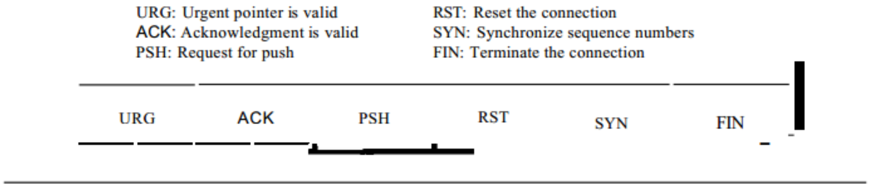
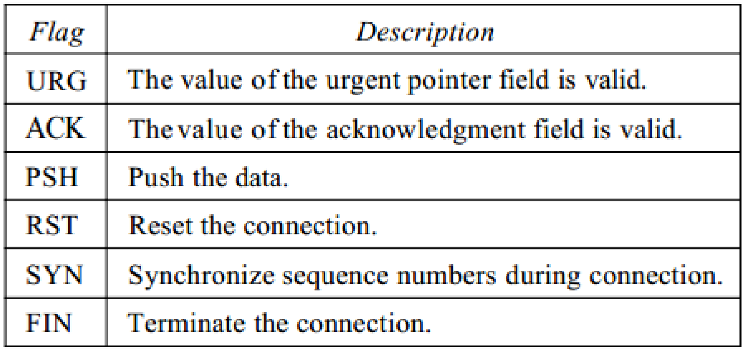

URG: Urgent pointer is valid
ACK: Acknowledgment is valid
PSH: Request for push
RST: Reset the connection
SYN: Synchronize sequence numbers
FIN: Terminate the connection
URG
ACK
PSH
RST
SYN
FIN

**Fig.5: Control field**

These bits enable flow control, connection establishment and termination, connection abortion, and the mode of data transfer in TCP. A brief description of each bit is shown in Table 2. We will discuss them further when we study the detailed operation of TCP later in the chapter.

**Table 2: Description of flags in the control field**



| *Flag* | *Description* |
|---|---|
| URG | The value of the urgent pointer field is valid. |
| ACK | The value of the acknowledgment field is valid. |
| PSH | Push the data. |
| RST | Reset the connection. |
| SYN | Synchronize sequence numbers during connection. |
| FIN | Terminate the connection. |

***Window size:*** This field defines the size of the window, in bytes, that the other party must maintain. Note that the length of this field is 16 bits, which means that the maximum size of the window is 65,535 bytes. This value is normally referred to as the receiving window (rwnd) and is determined by the receiver. The sender must obey the dictation of the receiver in this case.

***Checksum:*** This 16-bit field contains the checksum.. However, the inclusion of the checksum in the UDP datagram is optional, whereas the inclusion of the checksum for TCP is mandatory. The same pseudo-header, serving the same purpose, is added to the segment. For the TCP pseudoheader, the value for the protocol field is 6.

***Urgent pointer:*** This l6-bit field, which is valid only if the urgent flag is set, is used when the segment contains urgent data. It defines the number that must be added to the sequence number to obtain the number of the last urgent byte in the data section of the segment.

***Options****:* There can be up to 40 bytes of optional information in the TCP header.

**TCP Connection** TCP is connection-oriented. A connection-oriented transport protocol establishes a virtual path between the source and destination. All the segments belonging to a message are then sent over this virtual path. Using a single virtual pathway for the entire message facilitates the acknowledgment process as well as retransmission of damaged or lost frames. You may wonder how TCP, which uses the services of IP, a connectionless protocol, can be connection-oriented. The point is that a TCP connection is virtual, not physical. TCP operates at a higher level. TCP uses the services of IP to deliver individual segments to the receiver, but it controls the connection itself. If a segment is lost or corrupted, it is retransmitted. Unlike TCP, IP is unaware of this retransmission. If a segment arrives out of order, TCP holds it until the missing segments arrive; IP is unaware of this reordering. In TCP, connection-oriented transmission requires three phases: connection establishment, data transfer, and connection termination.

***Connection Establishment*** TCP transmits data in full-duplex mode. When two TCPs in two machines are connected, they are able to send segments to each other simultaneously. This implies that each party must initialize communication and get approval from the other party before any data are transferred.

**Three-Way Handshaking** The connection establishment in TCP is called three-way handshaking. In our example, an application program, called the client, wants to make a connection with another application program, called the server, using TCP as the transport layer protocol. The process starts with the server. The server program tells its TCP that it is ready to accept a connection. This is called a request for a *passive open.*

Although the server TCP is ready to accept any connection from any machine in the world, it cannot make the connection itself. The client program issues a request for an *active open.* A client that wishes to connect to an open server tells its TCP that it needs to be connected to that particular server. TCP can now start the three-way handshaking process as shown in Figure 6. To show the process, we use two time lines: one at each site.

Each segment has values for all its header fields and perhaps for some of its option fields, too. However, we show only the few fields necessary to understand each phase. We show the sequence number, the acknowledgment number, the control flags (only those that are set), and the window size, if not empty.

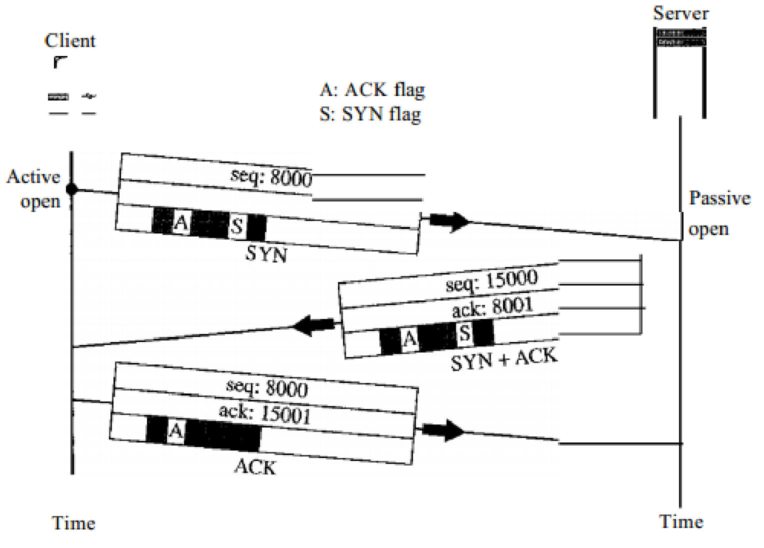


**Fig.6: Connection establishment using three way handshaking**

The three steps in this phase are as follows.

1. The client sends the first segment, a SYN segment, in which only the SYN flag is set. This segment is for synchronization of sequence numbers. It consumes one sequence number. When the data transfer starts, the sequence number is incremented by 1. We can say that the SYN segment carries no real data, but we can think of it as containing 1 imaginary byte.

2. The server sends the second segment, a SYN + ACK segment, with 2 flag bits set: SYN and ACK. This segment has a dual purpose. It is a SYN segment for communication in the other direction and serves as the acknowledgment for the SYN segment. It consumes one sequence number.

3. The client sends the third segment. This is just an ACK segment. It acknowledges the receipt of the second segment with the ACK flag and acknowledgment number field. Note that the sequence number in this segment is the same as the one in the SYN segment; the ACK segment does not consume any sequence numbers.

**Simultaneous Open** A rare situation, called a simultaneous open, may occur when both processes issue an active open. In this case, both TCPs transmit a SYN + ACK segment to each other, and one single connection is established between them.

**SYN Flooding Attack** The connection establishment procedure in TCP is susceptible to a serious security problem called the SYN flooding attack. This happens when a malicious attacker sends a large number of SYN segments to a server, pretending that each of them is corning from a different client by faking the source IP addresses in the datagrams. The server, assuming that the clients are issuing an active open, allocates the necessary resources, such as creating communication tables and setting timers. The TCP server then sends the SYN +ACK segments to the fake clients, which are lost. During this time, however, a lot of resources are occupied without being used. If, during this short time, the number of SYN segments is large, the server eventually runs out of resources and may crash. This SYN flooding attack belongs to a type of security attack known as a denial-of-service attack, in which an attacker monopolizes a system with so many service requests that the system collapses and denies service to every request. Some implementations of TCP have strategies to alleviate the effects of a SYN attack. Some have imposed a limit on connection requests during a specified period of time. Others filter out datagrams coming from unwanted source addresses. One recent strategy is to postpone resource allocation until the entire connection is set up, using what is called a cookie.

***Data Transfer*** After connection is established; bidirectional data transfer can take place. The client and server can both send data and acknowledgments. The acknowledgment is piggybacked with the data.

**Pushing Data** We saw that the sending TCP uses a buffer to store the stream of data coming from the sending application program. The sending TCP can select the segment size. The receiving TCP also buffers the data when they arrive and delivers them to the application program when the application program is ready or when it is convenient for the receiving TCP. This type of flexibility increases the efficiency of TCP.

However, on occasion the application program has no need for this flexibility_ For example, consider an application program that communicates interactively with another application program on the other end. The application program on one site wants to send a keystroke to the application at the other site and receive an immediate response. Delayed transmission and delayed delivery of data may not be acceptable by the application program. TCP can handle such a situation. The application program at the sending site can request a *push* operation. This means that the sending TCP must not wait for the window to be filled. It must create a segment and send it immediately. The sending TCP must also set the push bit (PSH) to let the receiving TCP know that the segment includes data that must be delivered to the receiving application program as soon as possible and not to wait for more data to come. Although the push operation can be requested by the application program, most current implementations ignore such requests. TCP can choose whether or not to use this feature.

**Urgent Data** TCP is a stream-oriented protocol. This means that the data are presented from the application program to TCP as a stream of bytes. Each byte of data has a position in the stream. However, on occasion an application program needs to send *urgent* bytes. This means that the sending application program wants a piece of data to be read out of order by the receiving application program. As an example, suppose that the sending application program is sending data to be processed by the receiving application program. When the result of processing comes back, the sending application program finds that everything is wrong. It wants to abort the process, but it has already sent a huge amount of data. If it issues an abort command (control + C), these two characters will be stored at the end of the receiving TCP buffer. It will be delivered to the receiving application program after all the data have been processed.

The solution is to send a segment with the URG bit set. The sending application program tells the sending TCP that the piece of data is urgent. The sending TCP creates a segment and inserts the urgent data at the beginning of the segment. The rest of the segment can contain normal data from the buffer. The urgent pointer field in the header defines the end of the urgent data and the start of normal data. When the receiving TCP receives a segment with the URG bit set, it extracts the urgent data from the segment, using the value of the urgent pointer, and delivers them, out of order, to the receiving application program.

***Connection Termination*** Any of the two parties involved in exchanging data (client or server) can close the connection, although it is usually initiated by the client. Most implementations today allow two options for connection termination: three-way handshaking and four-way handshaking with a half-close option.

**Three-Way Handshaking** Most implementations today allow *three-way handshaking* for connection termination as shown in Figure 6.

1. In a normal situation, the client TCP, after receiving a close command from the client process, sends the first segment, a FIN segment in which the FIN flag is set. Note that a FIN segment can include the last chunk of data sent by the client, or it can be just a control segment as shown in Figure 6. If it is only a control segment, it consumes only one sequence number.

2. The server TCP, after receiving the FIN segment, informs its process of the situation and sends the second segment, a FIN + ACK segment, to confirm the receipt of the FIN segment from the client and at the same time to announce the closing of the connection in the other direction. This segment can also contain the last chunk of data from the server. If it does not carry data, it consumes only one sequence number.

3. The client TCP sends the last segment, an ACK segment, to confirm the receipt of the FIN segment from the TCP server. This segment contains the acknowledgment number, which is 1 plus the sequence number received in the FIN segment from the server. This segment cannot carry data and consumes no sequence numbers.

**Half-Close** In TCP, one end can stop sending data while still receiving data. This is called a half-close. Although either end can issue a half-close, it is normally initiated by the client. It can occur when the server needs all the data before processing can begin. A good example is sorting. When the client sends data to the server to be sorted, the server needs to receive all the data before sorting can start. This means the client, after sending all the data, can close the connection in the outbound direction. However, the inbound direction must remain open to receive the sorted data. The server, after receiving the data, still needs time for sorting; its outbound direction must remain open.

**Flow Control** TCP uses a sliding window, to handle flow control. The sliding window protocol used by TCP, however, is something between the *Go-Back-N* and Selective Repeat sliding window. The sliding window protocol in TCP looks like the Go-Back-N protocol because it does not use NAKs; it looks like Selective Repeat because the receiver holds the out-of-order segments until the missing ones arrive. There are two big differences between this sliding window and the one we used at the data link layer. First, the sliding window of TCP is byte-oriented. Second, the TCP's sliding window is of variable size. Figure 7 shows the sliding window in TCP. The window spans a portion of the buffer containing bytes received from the process. The bytes inside the window are the bytes that can be in transit; they can be sent without worrying about acknowledgment. The imaginary window has two walls: one left and one right. The window is *opened*, *closed*, or *shrunk*. These three activities, as we will see, are in the control of the receiver (and depend on congestion in the network), not the sender. The sender must obey the commands of the receiver in this matter. Opening a window means moving the right wall to the right. This allows more new bytes in the buffer that are eligible for sending. Closing the window means moving the left wall to the right. This means that some bytes have been acknowledged and the sender need not worry about them anymore. Sluinking the window means moving the right wall to the left. This is strongly discouraged and not allowed in some implementations because it means revoking the eligibility of some bytes for sending. This is a problem if the sender has already sent these bytes. Note that the left wall cannot move to the left because this would revoke some of the previously sent acknowledgments.

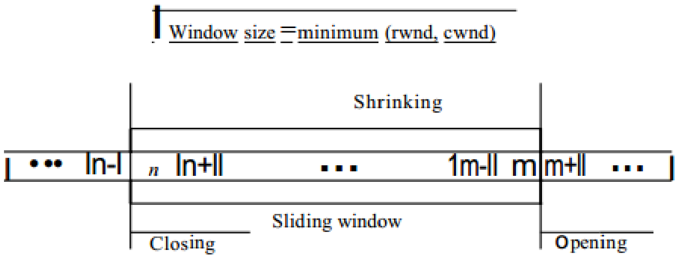


**Fig.7: Sliding Window**

The size of the window at one end is determined by the lesser of two values: *receiver window (rwnd)* or *congestion window (cwnd).* The *receiver window* is the value advertised by the opposite end in a segment containing acknowledgment. It is the number of bytes the other end can accept before its buffer overflows and data are discarded. The congestion window is a value determined by the network to avoid congestion.

**Error Control** TCP is a reliable transport layer protocol. This means that an application program that delivers a stream of data to TCP relies on TCP to deliver the entire stream to the application program on the other end in order, without error, and without any part lost or duplicated. TCP provides reliability using error control. Error control includes mechanisms for detecting corrupted segments, lost segments, out-of-order segments, and duplicated segments. Error control also includes a mechanism for correcting errors after they are detected. Error detection and correction in TCP is achieved through the use of three simple tools: checksum, acknowledgment, and time-out.

***Checksum*** Each segment includes a checksum field which is used to check for a corrupted segment. If the segment is corrupted, it is discarded by the destination TCP and is considered as lost. TCP uses a 16-bit checksum that is mandatory in every segment. ***Acknowledgment*** TCP uses acknowledgments to confirm the receipt of data segments. Control segments that carry no data but consume a sequence number are also acknowledged. ACK segments are never acknowledged.

***Retransmission*** The heart of the error control mechanism is the retransmission of segments. When a segment is corrupted, lost, or delayed, it is retransmitted. In modern implementations, a segment is retransmitted on two occasions: when a retransmission timer expires or when the sender receives three duplicate ACKs.

**Retransmission after RTO**

A recent implementation of TCP maintains one retransmission time-out (RTO) timer for all outstanding (sent, but not acknowledged) segments. When the timer matures, the earliest outstanding segment is retransmitted even though lack of a received ACK can be due to a delayed segment, a delayed ACK, or a lost acknowledgment. Note that no time-out timer is set for a segment that carries only an acknowledgment, which means that no such segment is resent. The value of RTO is dynamic in TCP and is updated based on the round-trip time (RTT) of segments. An RTI is the time needed for a segment to reach a destination and for an acknowledgment to be received. It uses a back-off strategy.

**Retransmission After Three Duplicate ACK Segments**

The previous rule about retransmission of a segment is sufficient if the value of RTO is not very large. Sometimes, however, one segment is lost and the receiver receives so many out-of-order segments that they cannot be saved (limited buffer size). To alleviate this situation, most implementations today follow the three-duplicate-ACKs rule and retransmit the missing segment immediately. This feature is referred to as fast retransmission, which we will see in an example shortly

**3. USER DATAGRAM PROTOCOL (UDP)**

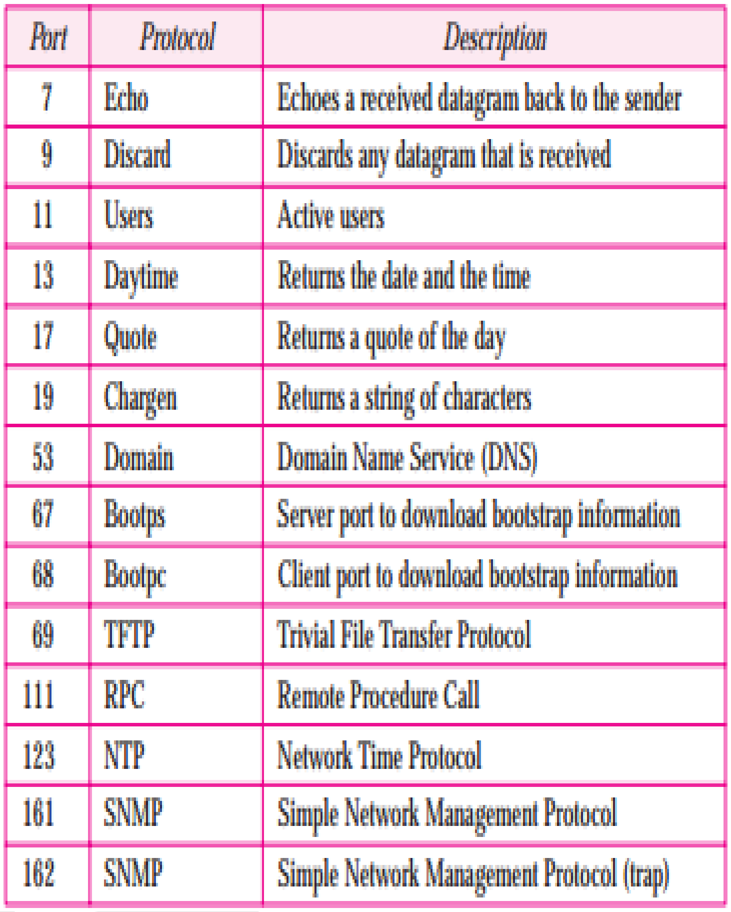
The User Datagram Protocol (UDP) is called a connectionless, unreliable transport protocol. It does not add anything to the services of IP except to provide process-to process communication instead of host-to-host communication. Also, it performs very limited error checking.

UDP is a very simple protocol using a minimum of overhead. If a process wants to send a small message and does not care much about reliability, it can use UDP. Sending a small message by using UDP takes much less interaction between the sender and receiver than using TCP or SCTP.

### Well-Known Ports for UDP

Table 3 shows some well-known port numbers used by UDP. Some port numbers can be used by both UDP and TCP.

**Table 3** ***Well-known ports used with UDP***



| Port | Protocol | Description |
|---|---|---|
| 7 | Echo | Echoes a received datagram back to the sender |
| 9 | Discard | Discards any datagram that is received |
| 11 | Users | Active users |
| 13 | Daytime | Returns the date and the time |
| 17 | Quote | Returns a quote of the day |
| 19 | Chargen | Returns a string of characters |
| 53 | Domain | Domain Name Service (DNS) |
| 67 | Bootps | Server port to download bootstrap information |
| 68 | Bootpc | Client port to download bootstrap information |
| 69 | TFTP | Trivial File Transfer Protocol |
| 111 | RPC | Remote Procedure Call |
| 123 | NTP | Network Time Protocol |
| 161 | SNMP | Simple Network Management Protocol |
| 162 | SNMP | Simple Network Management Protocol (trap) |

### User Datagram

UDP packets, called user datagrams, have a fixed-size header of 8 bytes. Figure 8 shows the format of a user datagram. The fields are as follows:

***Source port number:*** This is the port number used by the process running on the source host. It is 16 bits long, which means that the port number can range from 0 to 65,535. If the source host is the client (a client sending a request), the port number, in most cases, is an ephemeral port number requested by the process and chosen by the UDP software running on the source host. If the source host is the server (a server sending a response), the port number, in most cases, is a well-known port number

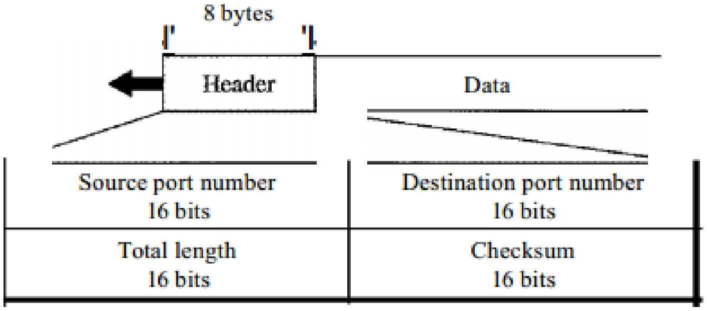


**Fig.8: User datagram format**

***Destination port number:*** This is the port number used by the process running on the destination host. It is also 16 bits long. If the destination host is the server (a client sending a request), the port number, in most cases, is a well-known port number. If the destination host is the client (a server sending a response), the port number, in most cases, is an ephemeral port number. In this case, the server copies the ephemeral port number it has received in the request packet.

***Length:*** This is a 16-bit field that defines the total length of the user datagram, header plus data. The 16 bits can define a total length of 0 to 65,535 bytes. However, the total length needs to be much less because a UDP user datagram is stored in an IP datagram with a total length of 65,535 bytes.

**UDP Operation**

UDP uses concepts common to the transport layer. These concepts will be discussed here briefly, and then expanded in the next section on the TCP protocol.

***Connectionless Services***

As mentioned previously, UDP provides a connectionless service. This means that each user datagram sent by UDP is an independent datagram. There is no relationship between the different user datagrams even if they are coming from the same source process and going to the same destination program. The user datagrams are not numbered. Also, there is no connection establishment and no connection termination, as is the case for TCP. This means that each user datagram can travel on a different path.

One of the ramifications of being connectionless is that the process that uses UDP cannot send a stream of data to UDP and expect UDP to chop them into different related user datagrams. Instead each request must be small enough to fit into one user datagram. Only those processes sending short messages should use UDP.

***Flow and Error Control***

UDP is a very simple, unreliable transport protocol. There is no flow control and hence no window mechanism. The receiver may overflow with incoming messages. There is no error control mechanism in UDP except for the checksum. This means that the sender does not know if a message has been lost or duplicated. When the receiver detects an error through the checksum, the user datagram is silently discarded. The lack of flow control and error control means that the process using UDP should provide these mechanisms.

***Encapsulation and Decapsulation***

To send a message from one process to another, the UDP protocol encapsulates and decapsulates messages in an IP datagram.

***Queuing***

We have talked about ports without discussing the actual implementation of them. In UDP, queues are associated with ports (see Figure 11).

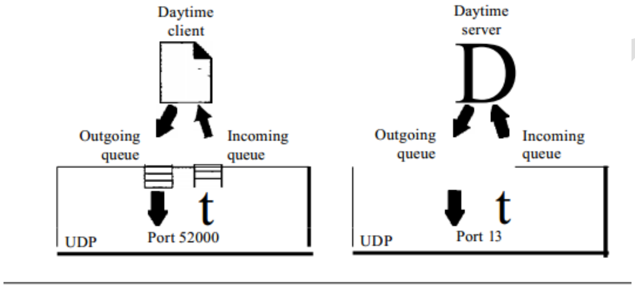


**Fig.11 Queues in UDP**

At the client site, when a process starts, it requests a port number from the operating system. Some implementations create both an incoming and an outgoing queue associated with each process. Other implementations create only an incoming queue associated with each process. Note that even if a process wants to communicate with multiple processes, it obtains only one port number and eventually one outgoing and one incoming queue. The queues opened by the client are, in most cases, identified by ephemeral port numbers.

The queues function as long as the process is running. When the process terminates, the queues are destroyed. The client process can send messages to the outgoing queue by using the source port number specified in the request. UDP removes the messages one by one and, after adding the UDP header, delivers them to IP. An outgoing queue can overflow. If this happens, the operating system can ask the client process to wait before sending any more messages. When a message arrives for a client, UDP checks to see if an incoming queue has been created for the port number specified in the destination port number field of the user datagram. If there is such a queue, UDP sends the received user datagram to the end of the queue. If there is no such queue, UDP discards the user datagram and asks the ICMP protocol to send a *port unreachable* message to the server. All the incoming messages for one particular client program, whether coming from the same or a different server, are sent to the same queue. An incoming queue can overflow. If this happens, UDP drops the user datagram and asks for a port unreachable message to be sent to the server. At the server site, the mechanism of creating queues is different. In its simplest form, a server asks for incoming and outgoing queues, using its well-known port, when it starts running. The queues remain open as long as the server is running.

When a message arrives for a server, UDP checks to see if an incoming queue has been created for the port number specified in the destination port number field of the user datagram. If there is such a queue, UDP sends the received user datagram to the end of the queue. If there is no such queue, UDP discards the user datagram and asks the ICMP protocol to send a port unreachable message to the client. All the incoming messages for one particular server, whether coming from the same or a different client, are sent to the same queue. An incoming queue can overflow. If this happens, UDP drops the user datagram and asks for a port unreachable message to be sent to the client. When a server wants to respond to a client, it sends messages to the outgoing queue, using the source port number specified in the request. UDP removes the messages one by one and, after adding the UDP header, delivers them to IP. An outgoing queue can overflow. If this happens, the operating system asks the server to wait before sending any more messages.

**Use of UDP**

The following lists some uses of the UDP protocol:

- UDP is suitable for a process that requires simple request-response communication with little concern for flow and error control. It is not usually used for a process such as FrP that needs to send bulk data.
- UDP is suitable for a process with internal flow and error control mechanisms. For example, the Trivial File Transfer Protocol (TFTP) process includes flow and error control. It can easily use UDP.
- UDP is a suitable transport protocol for multicasting. Multicasting capability is embedded in the UDP software but not in the TCP software.
- UDP is used for management processes such as SNMP .
- UDP is used for some route updating protocols such as Routing Information Protocol (RIP).

### 4. SCTP

Stream Control Transmission Protocol (SCTP) is a new reliable, message-oriented transport layer protocol. SCTP, however, is mostly designed for Internet applications that have recently been introduced. These new applications, such as IUA (ISDN over IP), M2UA and M3UA (telephony signaling), H.248 (media gateway control), H.323 (IP telephony), and SIP (IP telephony), need a more sophisticated service than TCP can provide. SCTP provides this enhanced performance and reliability.

We briefly compare UDP, TCP, and SCTP:

- UDP is a message-oriented protocol. A process delivers a message to UDP, which is encapsulated in a user datagram and sent over the network. UDP *conserves the message boundaries;* each message is independent of any other message. This is a desirable feature when we are dealing with applications such as IP telephony and transmission of real-time data, as we will see later in the text. However, UDP is unreliable; the sender cannot know the destiny of messages sent. A message can be lost, duplicated, or received out of order. UDP also lacks some other features, such as congestion control and flow control, needed for a friendly transport layer protocol.
- TCP is a byte-oriented protocol. It receives a message or messages from a process, stores them as a stream of bytes, and sends them in segments. There is no preservation of the message boundaries. However, TCP is a reliable protocol. The duplicate segments are detected, the lost segments are resent, and the bytes are delivered to the end process in order. TCP also has congestion control and flow control mechanisms.
- SCTP combines the best features of UDP and TCP. SCTP is a reliable message oriented protocol. It preserves the message boundaries and at the same time detects lost data, duplicate data, and out-of-order data. It also has congestion control and flow control mechanisms. Later we will see that SCTP has other innovative features unavailable in UDP and TCP.

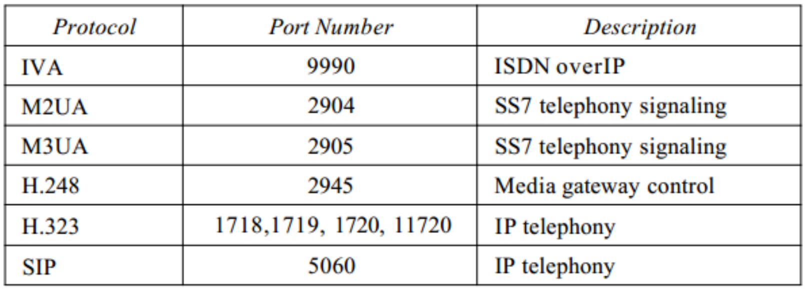
**SCTP Services**

Before we discuss the operation of SCTP, let us explain the services offered by SCTP to the application layer processes.

***Process-to-Process Communication*** SCTP uses all well-known ports in the TCP space.

***Multiple Streams*** We learned in the previous section that TCP is a stream-oriented protocol. Each connection between a TCP client and a TCP server involves one single stream. The problem with this approach is that a loss at any point in the stream blocks the delivery of the rest of the data. This can be acceptable when we are transferring text; it is not when we are sending real-time data such as audio or video. SCTP allows multistream service in each connection, which is called association in SCTP terminology. If one of the streams is blocked, the other streams can still deliver their data.The idea is similar to multiple lanes on a highway. Each lane can be used for a different type of traffic. For example, one lane can be used for regular traffic, another for car pools. If the traffic is blocked for regular vehicles, car pool vehicles can still reach their destinations. Figure 12 shows the idea of multiple-stream delivery.

**Table 4 lists some extra port numbers used by SCTP.**



| *Protocol* | *Port Number* | *Description* |
|---|---|---|
| IVA | 9990 | ISDN overIP |
| M2UA | 2904 | SS7 telephony signaling |
| M3UA | 2905 | SS7 telephony signaling |
| H.248 | 2945 | Media gateway control |
| H.323 | 1718,1719, 1720, 11720 | IP telephony |
| SIP | 5060 | IP telephony |

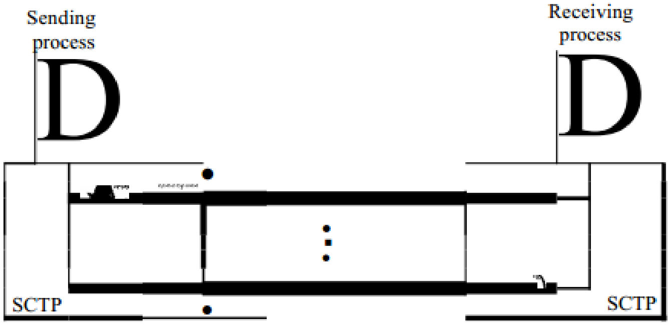


**Fig.12 Multiple-stream concept**

***Multi-homing***: A TCP connection involves one source and one destination IP address. This means that even if the sender or receiver is a multi-homed host (connected to more than one physical address with multiple IP addresses), only one of these IP addresses per end can be utilized during the connection. An SCTP association, on the other hand, supports multi-homing service. The sending and receiving host can define multiple IP addresses in each end for an association. In this fault-tolerant approach, when one path fails, another interface can be used for data delivery without interruption. This fault-tolerant feature is very helpful when we are sending and receiving a real-time payload such as Internet telephony. Figure 13 shows the idea of multi-homing.

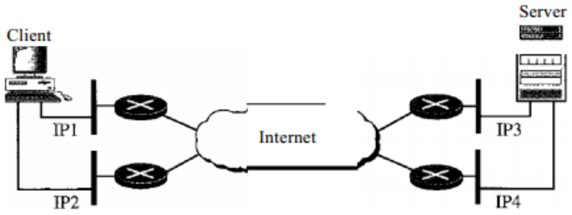


**Fig.13 Multihoming**

In Figure 13, the client is connected to two local networks with two IP addresses. The server is also connected to two networks with two IP addresses. The client and the server can make an association, using four different pairs of IP addresses. However, note that in the current implementations of SCTP, only one pair of IF addresses can be chosen for normal communication; the alternative is used if the main choice fails. In other words, at present, SCTP does not allow load sharing between different paths.

***Full-Duplex Communication*** Like TCP, SCTP offers full-duplex service, in which data can flow in both directions at the same time. Each SCTP then has a sending and receiving buffer, and packets are sent in both directions.

***Connection-Oriented Service*** Like TCP, SCTP is a connection-oriented protocol. However, in SCTP, a connection is called an association. When a process at site A wants to send and receive data from another process at site B, the following occurs:

1. The two SCTPs establish an association between each other.
2. Data are exchanged in both directions.
3. The association is terminated.

***Reliable Service*** SCTP, like TCP, is a reliable transport protocol. It uses an acknowledgment mechanism to check the safe and sound arrival of data. We will discuss this feature further in the section on error control.

**SCTP Features**

Let us first discuss the general features of SCTP and then compare them with those of TCP.

***Transmission Sequence Number:*** The unit of data in TCP is a byte. Data transfer in TCP is controlled by numbering bytes by using a sequence number. On the other hand, the unit of data in SCTP is a DATA chunk which may or may not have a one-to-one relationship with the message coming from the process because of fragmentation. Data transfer in SCTP is controlled by numbering the data chunks. SCTP uses a transmission sequence number (TSN) to number the data chunks. In other words, the TSN in SCTP plays the analogous role to the sequence number in TCP. TSNs are 32 bits long and randomly initialized between 0 and 232 - 1. Each data chunk must carry the corresponding TSN in its header.

***Stream Identifier***: In TCP, there is only one stream in each connection. In SCTP, there may be several streams in each association. Each stream in SCTP needs to be identified by using a stream identifier (SI). Each data chunk must carry the SI in its header so that when it arrives at the destination, it can be properly placed in its stream. The 51 is a 16-bit number starting from O.

***Stream Sequence Number:*** When a data chunk arrives at the destination SCTP, it is delivered to the appropriate stream and in the proper order. This means that, in addition to an SI, SCTP defines each data chunk in each stream with a stream sequence number (SSN).

***Packets:*** In TCP, a segment carries data and control information. Data are carried as a collection of bytes; control information is defined by six control flags in the header. The design of SCTP is totally different: data are carried as data chunks; control information is carried as control chunks. Several control chunks and data chunks can be packed together in a packet. A packet in SCTP plays the same role as a segment in TCP. Figure 14 compares a segment in TCP and a packet in SCTP. Let us briefly list the differences between an SCTP packet and a TCP segment:

1. The control information in TCP is part of the header; the control information in SCTP is included in the control chunks. There are several types of control chunks; each is used for a different purpose.

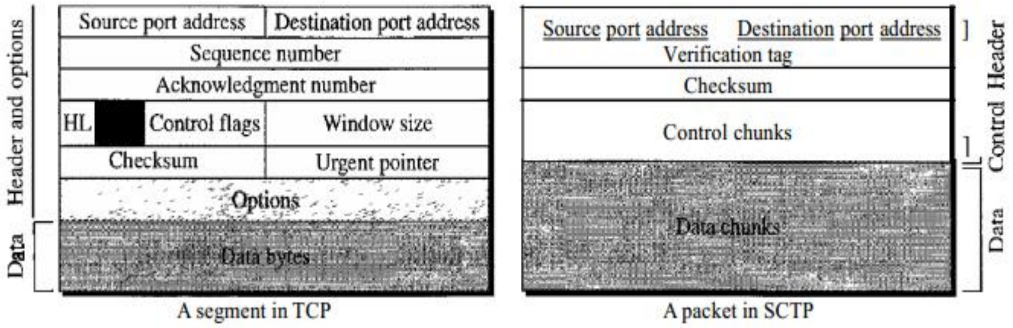


**Fig.14 Comparison between a TCP segment and an SCTP packet**

2. The data in a TCP segment treated as one entity; an SCTP packet can carry several data chunks; each can belong to a different stream.

3. The options section, which can be part of a TCP segment, does not exist in an SCTP packet. Options in SCTP are handled by defining new chunk types.

4. The mandatory part of the TCP header is 20 bytes, while the general header in SCTP is only 12 bytes.

The SCTP header is shorter due to the following:

a. An SCTP sequence number (TSN) belongs to each data chunk and hence is located in the chunk's header.

b. The acknowledgment number and window size are part of each control chunk.

c. There is no need for a header length field (shown as HL in the TCP segment) because there are no options to make the length of the header variable; the SCTP header length is fixed (12 bytes).

d. There is no need for an urgent pointer in SCTP.

5. The checksum in TCP is 16 bits; in SCTP, it is 32 bits.

6. The verification tag in SCTP is an association identifier, which does not exist in TCP. In TCP, the combination of IP and port addresses defines a connection; in SCTP we may have multihorning using different IP addresses. A unique verification tag is needed to define each association.

7. TCP includes one sequence number in the header, which defines the number of the first byte in the data section. An SCTP packet can include several different data chunks. TSNs, SIs, and SSNs define each data chunk.

8. Some segments in TCP that carry control information (such as SYN and FIN) need to consume one sequence number; control chunks in SCTP never use a TSN, SI, or SSN. These three identifiers belong only to data chunks, not to the whole packet.

***Acknowledgment Number*** TCP acknowledgment numbers are byte-oriented and refer to the sequence numbers. SCTP acknowledgment numbers are chunk-oriented. They refer to the TSN. A second difference between TCP and SCTP acknowledgments is the control information. Recall that this information is part of the segment header in TCP. To acknowledge segments that carry only control information, TCP uses a sequence number and acknowledgment number (for example, a SYN segment needs to be acknowledged by an ACK segment). In SCTP, however, the control information is carried by control chunks, which do not need a TSN. These control chunks are acknowledged by another control chunk of the appropriate type (some need no acknowledgment). For example, an INIT control chunk is acknowledged by an INIT ACK chunk. There is no need for a sequence number or an acknowledgment number.

***Flow Control***

Like TCP, SCTP implements flow control to avoid overwhelming the receiver.

***Error Control***

Like TCP, SCTP implements error control to provide reliability. TSN numbers and acknowledgment numbers are used for error control.

***Congestion Control***

Like TCP, SCTP implements congestion control to determine how many data chunks can be injected into the network.

***Chunks***

Control information or user data are carried in chunks. The first three fields are common to all chunks; the information field depends on the type of chunk. The important point to remember is that SCTP requires the information section to be a multiple of 4 bytes; if not, padding bytes (eight as) are added at the end of the section. See Table 6 for a list of chunks and their descriptions.

**An SCTP Association**

SCTP, like TCP, is a connection-oriented protocol. However, a connection in SCTP is called an *association* to emphasize multihoming.

***Association Establishment***

Association establishment in SCTP requires a four-way handshake. In this procedure, a process, normally a client wants to establish an association with another process, normally a server, using SCTP as the transport layer protocol. Similar to TCP, the SCTP server needs to be prepared to receive any association (passive open). Association establishment, however, is initiated by the client (active open). SCTP association establishment is shown in Figure 17. The steps, in a normal situation, are as follows:

1. The client sends the first packet, which contains an INIT chunk.
2. The server sends the second packet, which contains an INIT ACK chunk.
3. The client sends the third packet, which includes a COOKIE ECHO chunk. This is a very simple chunk that echoes, without change, the cookie sent by the server. SCTP allows the inclusion of data chunks in this packet.
4. The server sends the fourth packet, which includes the COOKIE ACK chunk that acknowledges the receipt of the COOKIE ECHO chunk. SCTP allows the inclusion of data chunks with this packet

***Data Transfer*** The whole purpose of an association is to transfer data between two ends. After the association is established, bidirectional data transfer can take place. The client and the server can both send data. Like TCP, SCTP supports piggybacking.

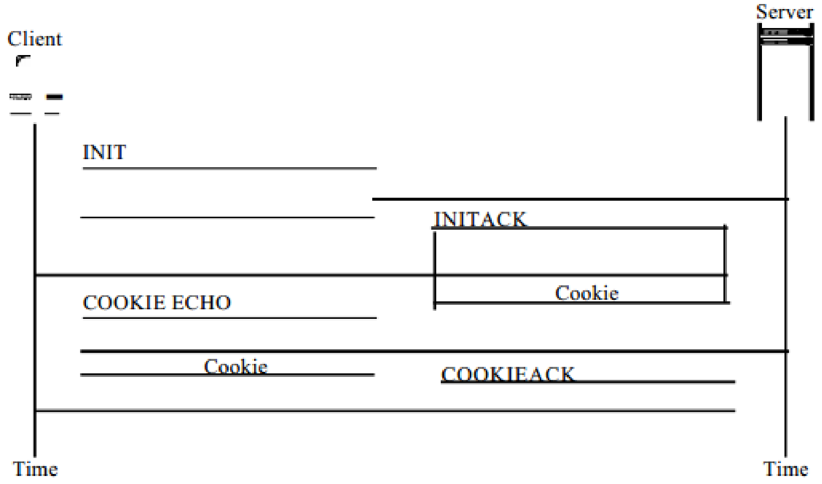


**Fig.17 Four-way handshaking**

There is a major difference, however, between data transfer in TCP and SCTP. TCP receives messages from a process as a stream of bytes without recognizing any boundary between them. The process may insert some boundaries for its peer use, but TCP treats that mark as part of the text. In other words, TCP takes each message and appends it to its buffer. A segment can carry parts of two different messages. The only ordering system imposed by TCP is the byte numbers. SCTP, on the other hand, recognizes and maintains boundaries. Each message coming from the process is treated as one unit and inserted into a DATA chunk unless it is fragmented (discussed later). In this sense, SCTP is like UDP, with one big advantage: data chunks are related to each other. A message received from a process becomes a DATA chunk, or chunks if fragmented, by adding a DATA chunk header to the message. Each DATA chunk formed by a message or a fragment of a message has one TSN. We need to remember that only DATA chunks use TSNs and only DATA chunks are acknowledged by SACK chunks.

**Multihoming Data Transfer:** We discussed the multihoming capability of SCTP, a feature that distinguishes SCTP from UDP and TCP. Multihoming allows both ends to define multiple IP addresses for communication. However, only one of these addresses can be defined as the primary address; the rest are alternative addresses. The primary address is defined during association establishment. The interesting point is that the primary address of an end is determined by the other end. In other words, a source defines the primary address for a destination.

**Multistream Delivery** One interesting feature of SCTP is the distinction between data transfer and data delivery. SCTP uses TSN numbers to handle data transfer, movement of data chunks between the source and destination. The delivery of the data chunks is controlled by SIs and SSNs. SCTP can support multiple streams, which means that the sender process can define different streams and a message can belong to one of these streams. Each stream is assigned a stream identifier (SI) which uniquely defines that stream.

**Fragmentation** Another issue in data transfer is fragmentation. Although SCTP shares this term with IP, fragmentation in IP and in SCTP belongs to different levels: the former at the network layer, the latter at the transport layer. SCTP preserves the boundaries of the message from process to process when creating a DATA chunk from a message if the size of the message (when encapsulated in an IP datagram) does not exceed the MTU of the path. The size of an IP datagram carrying a message can be determined by adding the size of the message, in bytes, to the four overheads: data chunk header, necessary SACK chunks, SCTP general header, and IP header. If the total size exceeds the MTU, the message needs to be fragmented.

***Association Termination*** In SCTP, like TCP, either of the two parties involved in exchanging data (client or server) can close the connection. However, unlike TCP, SCTP does not allow a halfclose situation. If one end closes the association, the other end must stop sending new data. If any data are left over in the queue of the recipient of the termination request, they are sent and the association is closed. Association **termination** uses three packets, as shown in Figure 18. Note that although the figure shows the case in which termination is initiated by the client, it can also be initiated by the server. Note that there can be several scenarios of association termination.

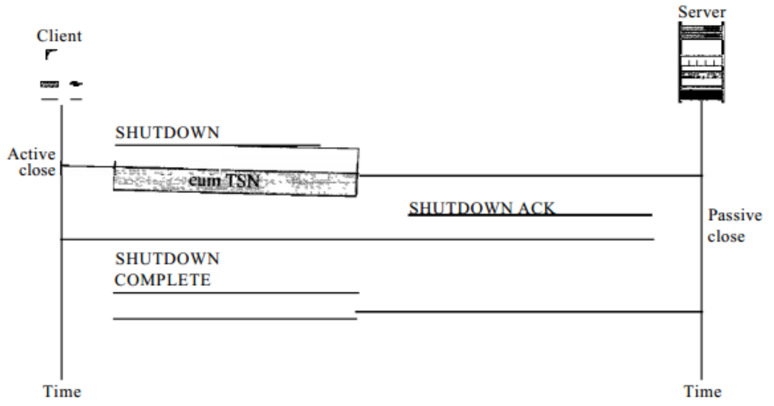


**Fig.18 Association Termination**

**Error Control** SCTP, like TCP, is a reliable transport layer protocol. It uses a SACK chunk to report the state of the receiver buffer to the sender. Each implementation uses a different set of entities and timers for the receiver and sender sites. We use a very simple design to convey the concept to the reader.

## Possible Questions

### 2 Mark Questions

1. Define a Network.
2. What are the Protocols supported in Transport Layer?
3. What is TCP?
4. What is an IP address?
5. What is a Port Number?

### 6 Mark Questions

1. Discuss in detail about OSI Reference Model
2. Write a detail note on a) UDP b) SCTP
3. What is TCP? How the Connection is established and terminated in TCP?
4. Write a detail note on a) TCP b) SCTP
5. What is SCTP? How the connection is established and terminated in SCTP?
6. Write a detail note on Port Number.
7. Differentiate TCP and UDP.
8. Discuss about the buffer size and limitations of the TCP.
9. Write a detail note on a) TCP b) UDP
10. Discuss about different applications and protocol supported in TCP/IP?

**KARPAGAM ACADEMY OF HIGHER EDUCATION**
**(Deemed University Established Under Section 3 of UGC Act, 1956)**
**Coimbatore – 641021, INDIA**
**Department of Computer Science, Applications & Information Technology**

Enable | Enlighten | Enrich
(Deemed to be University)
(Under Section 3 of UGC Act 1956)

**Unit - I**

| S.No | Questions | Opt1 | opt2 | opt3 | opt4 | Answer |
|---|---|---|---|---|---|---|
| 1 | The _____ layer is responsible for process-to-process delivery of the entire message | transport | data link | application | session | transport |
| 2 | TCP provides _____communication using port numbers | Host –to-host | process to process | port to port | interface to interface | process to process |
| 3 | At the transport layer, TCP groups a number of bytes together into a packet called | frames | bits | segments | datagrams | segments |
| 4 | The _____ function is used by a TCP client to establish a connection with a TCP server | bind | open | frame | connect | connect |
| 5 | The maximum size of the TCP header is | 40 bytes | 60 bytes | 80 bytes | 100 bytes | 60 bytes |
| 6 | In Stream Control Transmission Protocol control information and data information are carried in | Flow Chunks | Err-Control Chunk | same chunks | separate chunks | separate chunks |
| 7 | How many ports a computer may have | 256 | 128 | 65535 | 1024 | 65535 |
| 8 | In a TCP header source and destination header contains | 8 Bits | 16 Bits | 32 Bits | 128 Bits | 32 Bits |
| 9 | UDP and TCP are both_____ layer protocols | data link | network | transport | interface | transport |
| 10 | A ____ is an application program running on a host. | process | segment | port | interface | process |
| 11 | The must include port number or port addresses in the Internet and TCP/IP protocol suite | data link | application | session | transport layer | transport layer |
| 12 | A connection oriented protocol, such as____ , creates a relationship between the segments using sequence numbers | TCP and UDP | TCP and IP | TCP and SCTP | TCP and IGMP | TCP and SCTP |
| 13 | ___ uses flow and error control mechanisms at the transport leve | TCP | UDP | SCTP | IGMP | TCP |
| 14 | ____ is a stream-oriented protocol | UDP | SCTP | IGMP | TCP | TCP |
| 15 | TCP transmits data in ____mode | half-duplex | full-duplex | semi-duplex | transparent-duplex | full-duplex |
| 16 | The connection establishment in TCP is called | two way handshaking | multiway handshaking | threeway handshaking | fourway handshaking | threeway handshaking |
| 17 | The client program issues a request for an | active open | passive open | client open | server open | active open. |
| 18 | The ____ is called a connectionless, unreliable transport protocol | TCP | UDP | SCTP | SMCT | UDP |
| 19 | UDP packets, called user datagrams, have a fixed-size header of ___ bytes | 16 | 8 | 128 | 64 | 8 |
| 20 | ___is a unreliable transport protocol | TCP | IGMP | UDP | SCTP | UDP |
| 21 | the UDP protocol encapsulates and decapsulates messages in an ___ | IP datagram | segments | frames | packets | IP datagram |
| 22 | UDP is a suitable transport protocol for ___ | unicast | multicasting | boardcasting | multiway casting | multicasting |
| 23 | UDP is used for some route updating protocols such as | RIP | DIP | UIP | ARP | RIP |
| 24 | ___is a new reliable, message-oriented transport layer protocol. | TCP | IGMP | UDP | SCTP | SCTP |
| 25 | ___ combines the best features of UDP and TCP | TCP | IGMP | UDP | SCTP | SCTP |
| 26 | SCTP allows ____ service in each connection | bytestream | unistream | multistream | forwardstream | multistream |
| 27 | A connection in SCTP is ___called an to emphasize multihoming | distribution | association | identity | axioms | association |
| 28 | A connection in SCTP is called association to emphasize ___ | multistream | multihoming | multienvironment | multilayered | multihoming |
| 29 | The sending and receiving host can define multiple IP addresses is called as | multistream | multihoming | multienvironment | multilayered | multihoming |
| 30 | ___ does not allow load sharing between different paths | TCP | IGMP | UDP | SCTP | SCTP |
| 31 | TSN stands for | Transmission Sequence Number | Transfer Sequence Number | Traffic Sequence Number | Total Sequence Number | Transmission Sequence Number |
| 32 | The unit of data in TCP is a | bit | frame | byte | segments | byte |
| 33 | Data transfer in SCTP is controlled by numbering the | segment chunks | data chunks | frame chunks | datagram chunks | data chunks |
| 34 | TSNs are ___ bits long | 32 | 64 | 128 | 256 | 32 |
| 35 | Each stream in SCTP needs to be identified by using a | byte identifier | bit identifer | client identifier | stream identifier | stream identifier |
| 36 | SCTP defines each data chunk in each stream with a | byte sequence number | stream sequence number | segment sequence number | frame sequence number | stream sequence number |
| 37 | SSN stands | stream sequence number |  | string sequence number | stream segement number | stream sequence number |
| 38 | In TCP, a ___carries data and control information | frames | segment | packets | datagrams | segment |
| 39 | In SCTP, data are carried as | information chunks | frame chunks | data chunks | segement chunks | data chunks |
| 40 | n SCTP, control information is carried as | information chunks | control chunks | data chunks | segement chunks | control chunks |
| 41 | A ___ in SCTP plays the same role as a segment in TCP | packet | frames | datagrams | chunks | packet |
| 42 | SCTP uses a ___ chunk to report the state of the receiver buffer to the sender | TACK | PACK | QACK | SACK | SACK |
| 43 | Only ___ chunks use TSNs | FRAMES | DATA | SEGMENTS | PACKET | DATA |

## UNIT-I

## SYLLABUS

**Socket Programming:** Socket Introduction; TCP Sockets; TCP Client/Server Example ; signal handling

### 1. Socket Programming

Sockets allow communication between two different processes on the same or different machines.

**Where is Socket Used?**

A Unix Socket is used in a client-server application framework. A server is a process that performs some functions on request from a client. Most of the application-level protocols like FTP, SMTP, and POP3 make use of sockets to establish connection between client and server and then for exchanging data.

**Socket Types**

There are four types of sockets available to the users. The first two are most commonly used and the last two are rarely used.

Processes are presumed to communicate only between sockets of the same type but there is no restriction that prevents communication between sockets of different types.

- **Stream Sockets** − Delivery in a networked environment is guaranteed. If you send through the stream socket three items "A, B, C", they will arrive in the same order − "A, B, C". These sockets use TCP (Transmission Control Protocol) for data transmission. If delivery is impossible, the sender receives an error indicator. Data records do not have any boundaries.
- **Datagram Sockets** − Delivery in a networked environment is not guaranteed. They're connectionless because you don't need to have an open connection as in Stream Sockets − you build a packet with the destination information and send it out. They use UDP (User Datagram Protocol).
- **Raw Sockets** − These provide users access to the underlying communication protocols, which support socket abstractions. These sockets are normally datagram oriented, though their exact characteristics are dependent on the interface provided by the protocol. Raw sockets are not intended for the general user; they have been provided mainly for those interested in developing new communication protocols, or for gaining access to some of the more cryptic facilities of an existing protocol.
- Sequenced Packet Sockets − They are similar to a stream socket, with the exception that record boundaries are preserved. This interface is provided only as a part of the Network Systems (NS)

socket abstraction, and is very important in most serious NS applications. Sequenced-packet sockets allow the user to manipulate the Sequence Packet Protocol (SPP) or Internet Datagram Protocol (IDP) headers on a packet or a group of packets, either by writing a prototype header along with whatever data is to be sent, or by specifying a default header to be used with all outgoing data, and allows the user to receive the headers on incoming packets.

2. **Sockets Introduction**

**Socket Address Structures:**

The name of socket address structures begin with sockaddr_ and end with a unique suffix for each protocol suite.

**IPv4 Socket Address Structure:** *An IPv4 socket address structure, commonly called an "Internet socket address structure", is namedsockaddr_in and is defined by including the <netinet/in.h> header.*

```
struct in_addr
{
         in_addr_t   s_addr;          /* 32-bit IPv4 address */
        /* network byte ordered */
};
struct sockaddr_in
{
        uint8_t sin_len;      /* length of structure (16) */
        sa_family_t sin_family;   /* AF_INET */
        in_port_t sin_port;     /* 16-bit TCP or UDP port number */
        /* network byte ordered */
        struct in_addr  sin_addr;     /* 32-bit IPv4 address *
        /* network byte ordered */
        char sin_zero[8];  /* unused */
};
```

- **sin_len:** the length field. The four socket functions that pass a socket address structure from the process to the kernel,bind, connect, sendto, and sendmsg, all go through the sockargs function in a Berkeley-derived implementation. This function copies the socket address structure from the process and explicitly sets its sin_len member to the size of the structure that was passed as an argument to these four functions. The five socket functions that pass a socket address structure from the kernel to the process, accept, recvfrom, recvmsg, getpeername, andgetsockname, all set the sin_len member before returning to the process.

- **POSIX** requires only three members in the structure: sin_family, sin_addr, and sin_port. Almost all implementations add the sin_zero member so that all socket address structures are at least 16 bytes in size.
- The **in_addr_t** datatype must be an unsigned integer type of at least 32 bits, in_port_t must be an unsigned integer type of at least 16 bits, and sa_family_t can be any unsigned integer type. The latter is normally an 8-bit unsigned integer if the implementation supports the length field, or an unsigned 16-bit integer if the length field is not supported.
- Both the IPv4 address and the TCP or UDP port number are always stored in the structure in **network byte order**.
- The sin_zero member is unused. By convention, we always set the entire structure to 0 before filling it in.
- Socket address structures are used only on a given host: The structure itself is not communicated between different hosts

**Generic Socket Address Structure**

A socket address structures is always passed by reference when passed as an argument to any socket functions. But any socket function that takes one of these pointers as an argument must deal with socket address structures from any of the supported protocol families.

A generic socket address structure in the <sys/socket.h> header:

```
struct sockaddr
{
        uint8_t sa_len;
        sa_family_t  sa_family;    /* address family: AF_xxx value */
        char sa_data[14];  /* protocol-specific address */
};
```

The socket functions are then defined as taking a pointer to the generic socket address structure.

```
int bind(int, struct sockaddr *, socklen_t);
```

This requires that any calls to these functions must cast the pointer to the *protocol-specific socket address structure* to be a pointer to a *generic socket address structure*.

**For example:**

```
struct sockaddr_in  serv;      /* IPv4 socket address structure */
```

```
/* fill in serv{} */
bind(sockfd, (struct sockaddr *) &serv, sizeof(serv));
```

We have defined SA to be the string struct sockaddr, just to shorten the code that we must write to cast these pointers.

- From an application programmer 's point of view, the only use of these generic socket address structures is to cast pointers to protocol-specific structures.
- From the kernel's perspective, another reason for using pointers to generic socket address structures as arguments is that the kernel must take the caller's pointer, cast it to a struct sockaddr *, and then look at the value of sa_family to determine the type of the structure.

**IPv6 Socket Address Structure**

The IPv6 socket address is defined by including the <netinet/in.h> header:

```
struct in6_addr
{
uint8_t  s6_addr[16];        /* 128-bit IPv6 address */
 /* network byte ordered */
};
#define SIN6_LEN      /* required for compile-time tests */
struct sockaddr_in6
{
        uint8_t sin6_len;      /* length of this struct (28) */
        sa_family_t sin6_family;   /* AF_INET6 */
        in_port_t sin6_port;     /* transport layer port# */
        /* network byte ordered */
        uint32_t sin6_flowinfo; /* flow information, undefined */
        struct in6_addr sin6_addr;     /* IPv6 address */
         /* network byte ordered */
        uint32_t sin6_scope_id; /* set of interfaces for a scope */
};
```

- The SIN6_LEN constant must be defined if the system supports the length member for socket address structures.
- The IPv6 family is AF_INET6, whereas the IPv4 family is AF_INET

- The members in this structure are ordered so that if the sockaddr_in6 structure is 64-bit aligned, so is the 128-bit sin6_addr member.
- The sin6_flowinfo member is divided into two fields:
  - The low-order 20 bits are the flow label
  - The high-order 12 bits are reserved
- The sin6_scope_id identifies the scope zone in which a scoped address is meaningful, most commonly an interface index for a link-local address

**New Generic Socket Address Structure**

A new generic socket address structure was defined as part of the IPv6 sockets API, to overcome some of the shortcomings of the existing struct sockaddr. Unlike the struct sockaddr, the newstruct sockaddr_storage is large enough to hold any socket address type supported by the system. The sockaddr_storage structure is defined by including the <netinet/in.h> header:

```
struct sockaddr_storage
{
        uint8_t      ss_len;       /* length of this struct (implementation dependent) */
        sa_family_t  ss_family;    /* address family: AF_xxx value */
        /* implementation-dependent elements to provide:
        * a) alignment sufficient to fulfill the alignment requirements of
        *    all socket address types that the system supports.
        * b) enough storage to hold any type of socket address that the
        *    system supports.
        */
};
```

The sockaddr_storage type provides a generic socket address structure that is different from struct sockaddr in two ways:

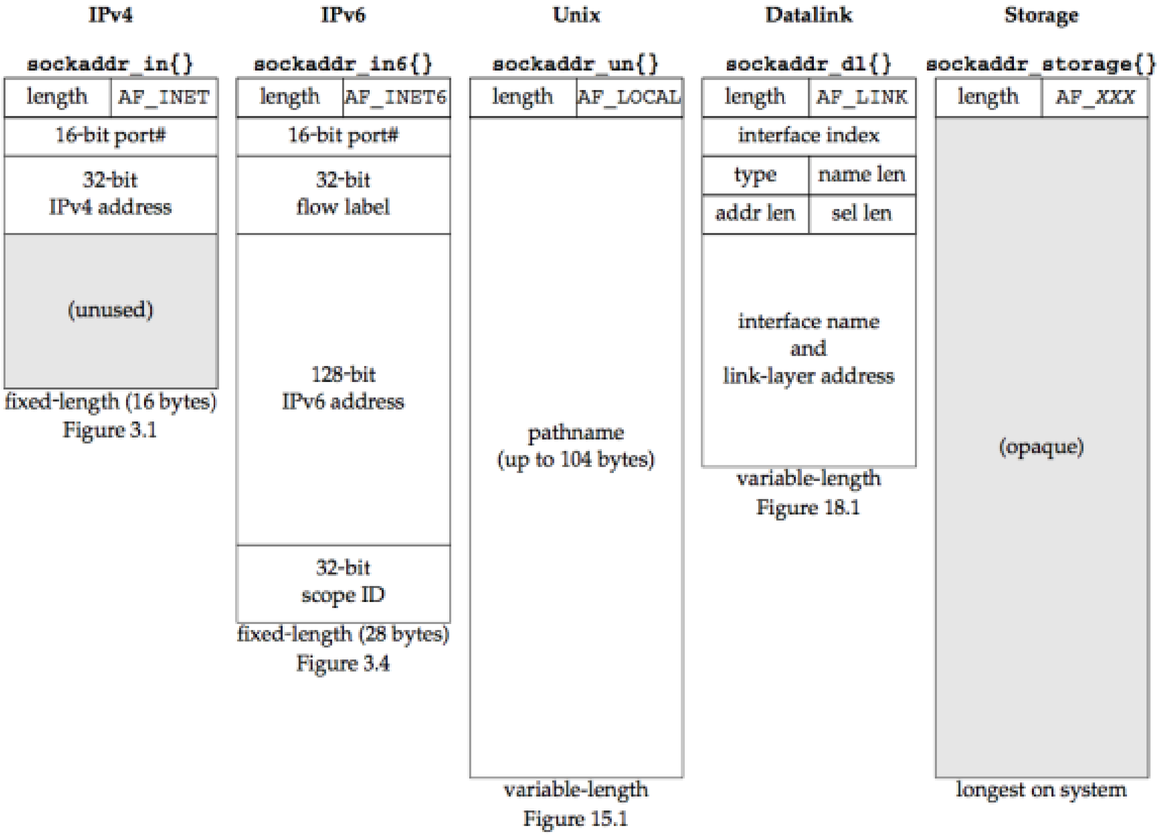
1. If any socket address structures that the system supports have alignment requirements, thesockaddr_storage provides the strictest alignment requirement.
2. The sockaddr_storage is large enough to contain any socket address structure that the system supports.

The fields of the sockaddr_storage structure are opaque to the user, except for ss_family and ss_len(if present). The sockaddr_storage must be cast or copied to the appropriate socket address structure for the address given in ss_family to access any other fields.

**Comparison of Socket Address Structures**

In this figure, we assume that:

- Socket address structures all contain a one-byte length field
- The family field also occupies one byte
- Any field that must be at least some number of bits is exactly that number of bits



To handle variable-length structures, whenever we pass a pointer to a socket address structure as an argument to one of the socket functions, we pass its length as another argument.

**Value-Result Arguments**

When a socket address structure is passed to any socket function, it is always passed by reference (a pointer to the structure is passed). The length of the structure is also passed as an argument.

The way in which the length is passed depends on which direction the structure is being passed:

1. From the **process to the kernel**
2. From the **kernel to the process**

**From process to kernel**

bind, connect, and sendto functions pass a socket address structure from the process to the kernel.

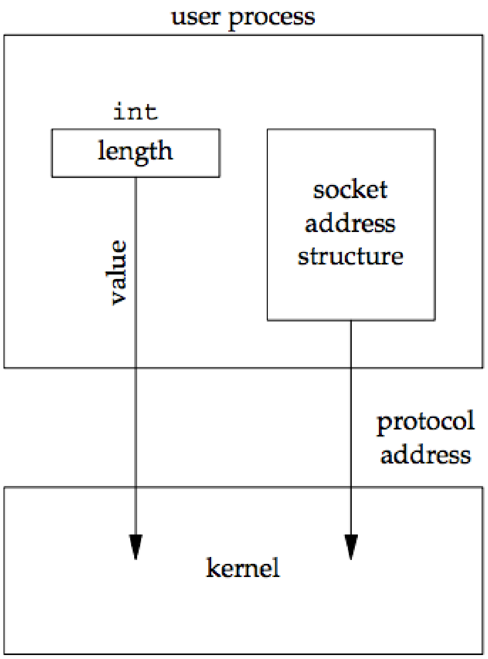
**Arguments to these functions:**

- The pointer to the socket address structure
- The integer size of the structure

**struct sockaddr_in serv;**

**/* fill in serv{} */**

**connect (sockfd, (SA *) &serv, sizeof(serv));**



The datatype for the size of a socket address structure is actually socklen_t and not int, but the POSIX specification recommends that socklen_t be defined as uint32_t.

**From kernel to process**

accept, recvfrom, getsockname, and getpeername functions pass a socket address structure from the kernel to the process.

**Arguments to these functions:**

- The pointer to the socket address structure
- The pointer to an integer containing the size of the structure.

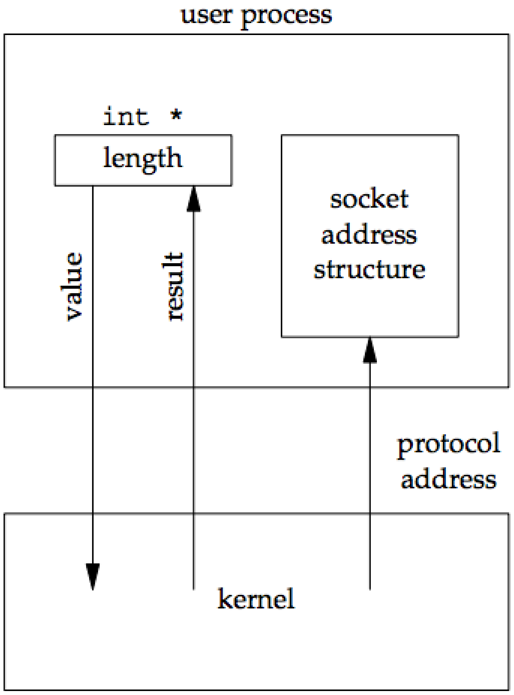
**struct sockaddr_un cli; /* Unix domain */**

**socklen_t len;**

**len = sizeof(cli); /* len is a value */**

**getpeername(unixfd, (SA *) &cli, &len);**

**/* len may have changed */**



**Value-result argument** In the above figure the size changes from an integer to be a pointer to an integer because the size is both a value when the function is called and a result when the function returns.

- As a **value**: it tells the kernel the size of the structure so that the kernel does not write past the end of the structure when filling it in
- As a **result**: it tells the process how much information the kernel actually stored in the structure

For two other functions that pass socket address structures, recvmsg and sendmsg, the length field is not a function argument but a structure member.

If the socket address structure is fixed-length, the value returned by the kernel will always be that fixed size: 16 for an IPv4 sockaddr_in and 28 for an IPv6 sockaddr_in6. But with a variable-length socket address structure (e.g., a Unix domain sockaddr_un), the value returned can be less than the maximum size of the structure.

Though the most common example of a value-result argument is the length of a returned socket address structure, we will encounter other value-result arguments in this text:

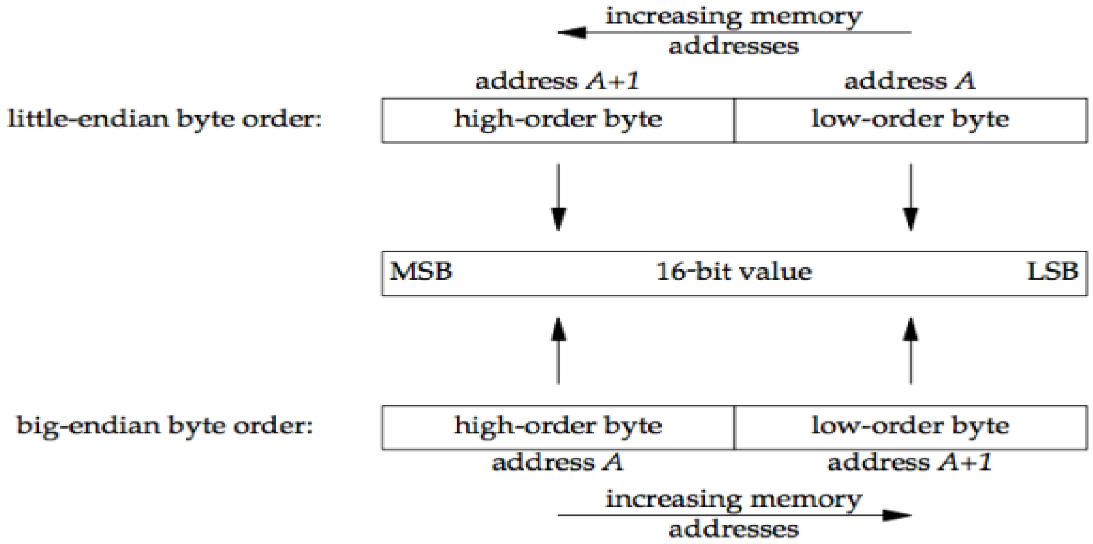
- The middle three arguments for the select function
- The length argument for the getsockopt function
- The msg_namelen and msg_controllen members of the msghdr structure, when used withrecvmsg

- The ifc_len member of the ifconf structure
- The first of the two length arguments for the sysctl function

**Byte Ordering Functions**

For a 16-bit integer that is made up of 2 bytes, there are two ways to store the two bytes in memory:

- **Little-endian** order: low-order byte is at the starting address.
- **Big-endian** order: high-order byte is at the starting address.



The figure shows the most significant bit (MSB) as the leftmost bit of the 16-bit value and the least significant bit (LSB) as the rightmost bit. The terms "little-endian" and "big-endian" indicate which end of the multibyte value, the little end or the big end, is stored at the starting address of the value.

Networking protocols must specify a **network byte order**. The sending protocol stack and the receiving protocol stack must agree on the order in which the bytes of these multibyte fields will be transmitted. The Internet protocols use big-endian byte ordering for these multibyte integers.

But, both history and the POSIX specification say that certain fields in the socket address structures must be maintained in network byte order. We use the following four functions to convert between these two byte orders:

***unp_htons.h***

```
#include <netinet/in.h>
uint16_t htons(uint16_t host16bitvalue);
uint32_t htonl(uint32_t host32bitvalue);
/* Both return: value in network byte order */
uint16_t ntohs(uint16_t net16bitvalue);
```

**uint32_t ntohl(uint32_t net32bitvalue);**

**/* Both return: value in host byte order */**

- h stands for *host*
- n stands for *network*
- s stands for *short* (16-bit value, e.g. TCP or UDP port number)
- l stands for *long* (32-bit value, e.g. IPv4 address)

When using these functions, we do not care about the actual values (big-endian or little-endian) for the host byte order and the network byte order. What we must do is call the appropriate function to convert a given value between the host and network byte order. On those systems that have the same byte ordering as the Internet protocols (big-endian), these four functions are usually defined as null macros.

We use the term "byte" to mean an 8-bit quantity since almost all current computer systems use 8-bit bytes. Most Internet standards use the term **octet** instead of byte to mean an 8-bit quantity.

Bit ordering is an important convention in Internet standards, such as the first 32 bits of the IPv4 header from RFC 791:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Version|  IHL |Type of Service|          Total Length         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

This represents four bytes in the order in which they appear on the wire; the leftmost bit is the most significant. However, the numbering starts with zero assigned to the most significant bit.

**Byte Manipulation Functions**

Two types functions differ in whether they deal with null-terminated C strings:

- The functions that operate on multibyte fields, without interpreting the data, and without assuming that the data is a null-terminated C string. These types of functions deal with socket address structures to manipulate fields such as IP addresses, which can contain bytes of 0, but are not C character strings.
    - The functions whose names begin with b (for byte) (from 4.2BSD)
    - The functions whose names begin with mem (for memory) (from ANSI C)

- The functions that deal with null-terminated C character strings (beginning with str (for string), defined by including the <string.h> header)

***unp_bzero.h***

**#include <strings.h>**

**void bzero(void *dest, size_t nbytes);**

**void bcopy(const void *src, void *dest, size_t nbytes);**

**int bcmp(const void *ptr1, const void *ptr2, size_t nbytes);**

**/* Returns: 0 if equal, nonzero if unequal */**

The memory pointed to by the const pointer is read but not modified by the function.

- bzero sets the specified number of bytes to 0 in the destination. We often use this function to initialize a socket address structure to 0.
- bcopy moves the specified number of bytes from the source to the destination.
- bcmp compares two arbitrary byte strings. The return value is zero if the two byte strings are identical; otherwise, it is nonzero

***unp_memset.h***

**#include <string.h>**

**void *memset(void *dest, int c, size_t len);**

**void *memcpy(void *dest, const void *src, size_t nbytes);**

**int memcmp(const void *ptr1, const void *ptr2, size_t nbytes);**

**/* Returns: 0 if equal, <0 or >0 if unequal (see text) */**

- memset sets the specified number of bytes to the value c in the destination
- memcpy is similar to bcopy, but the order of the two pointer arguments is swapped
- memcmp compares two arbitrary byte strings

**inet_aton, inet_addr, and inet_ntoa Functions**

These functions convert Internet addresses between ASCII strings (what humans prefer to use) and network byte ordered binary values (values that are stored in socket address structures).

***unp_inet_aton.h***

**#include <arpa/inet.h>**

**int inet_aton(const char *strptr, struct in_addr *addrptr);**

**/* Returns: 1 if string was valid, 0 on error */**

**in_addr_t inet_addr(const char *strptr);**

**/* Returns: 32-bit binary network byte ordered IPv4 address; INADDR_NONE if error */**

**char *inet_ntoa(struct in_addr inaddr);**

**/* Returns: pointer to dotted-decimal string */**

- inet_aton: converts the C character string pointed to by *strptr* into its 32-bit binary network byte ordered value, which is stored through the pointer *addrptr*
- inet_addr: does the same conversion, returning the 32-bit binary network byte ordered value as the return value. It is deprecated and any new code should use inet_aton instead
- inet_ntoa: converts a 32-bit binary network byte ordered IPv4 address into its corresponding dotted-decimal string.
    - The string pointed to by the return value of the function resides in static memory. This means the function is not reentrant.
    - This function takes a structure as its argument, not a pointer to a structure. (Functions that take actual structures as arguments are rare. It is more common to pass a pointer to the structure.)

**inet_pton and inet_ntop Functions**

These two functions are new with IPv6 and work with both IPv4 and IPv6 addresses. We use these two functions throughout the text. The letters "p" and "n" stand for *presentation* and *numeric*. The presentation format for an address is often an ASCII string and the numeric format is the binary value that goes into a socket address structure.

***unp_inet_pton.h***

**#include <arpa/inet.h>**

**int inet_pton(int family, const char *strptr, void *addrptr);**

**/* Returns: 1 if OK, 0 if input not a valid presentation format, -1 on error */**

**const char *inet_ntop(int family, const void *addrptr, char *strptr, size_t len);**

**/* Returns: pointer to result if OK, NULL on error */**

**Arguments:**

- *family*: is either AF_INET or AF_INET6. If *family* is not supported, both functions return an error witherrno set to EAFNOSUPPORT.

**Functions:**

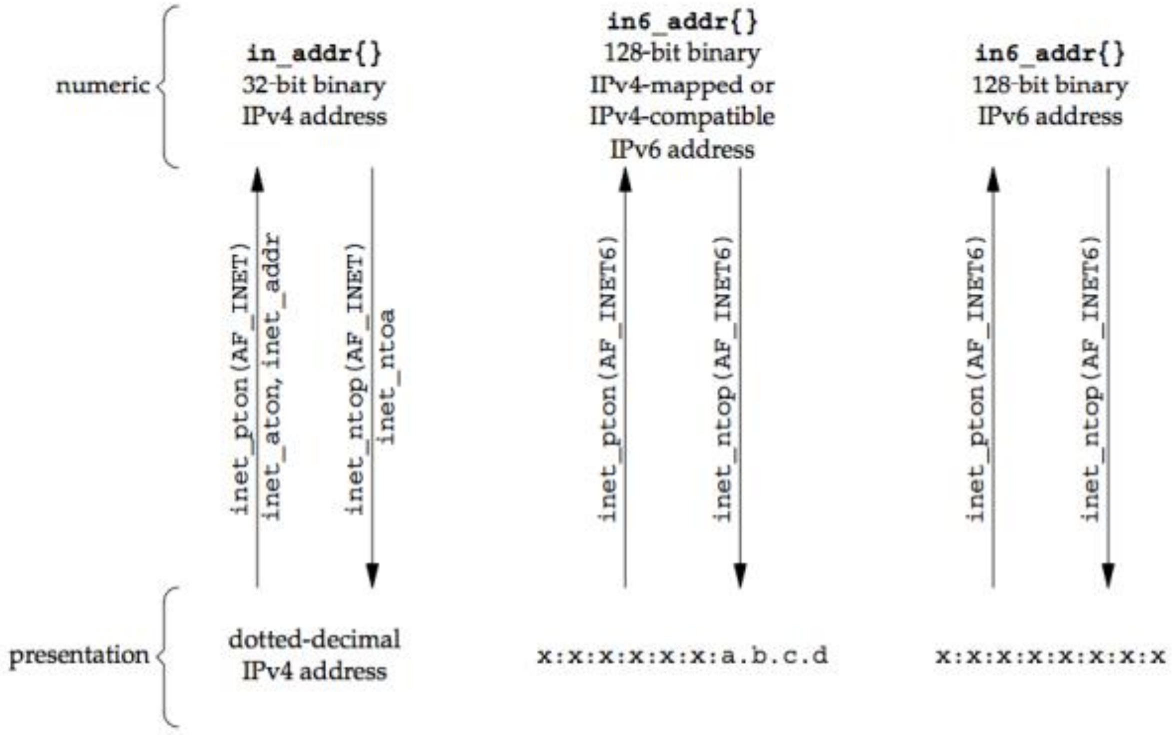
- inet_pton: converts the string pointed to by *strptr*, storing the binary result through the pointer*addrptr*. If successful, the return value is 1. If the input string is not a valid presentation format for the specified *family*, 0 is returned.
- inet_ntop does the reverse conversion, from numeric (*addrptr*) to presentation (*strptr*).

- *len* argument is the size of the destination. To help specify this size, the following two definitions are defined by including the <netinet/in.h> header.
- If *len* is too small to hold the resulting presentation format, including the terminating null, a null pointer is returned and errno is set to ENOSPC.
- The *strptr* argument to inet_ntop cannot be a null pointer. The caller must allocate memory for the destination and specify its size. On success, this pointer is the return value of the function.

Size definitions in <netinet/in.h> header for the *len* argument:

```
#define INET_ADDRSTRLEN       16       /* for IPv4 dotted-decimal */
#define INET6_ADDRSTRLEN      46       /* for IPv6 hex string */
```

The following figure summarizes the five functions on address conversion functions:



**Replacing inet_addr to inet_pton**

**Replace:**

foo.sin_addr.s_addr = inet_addr(cp);

**with**

inet_pton(AF_INET, cp, &foo.sin_addr);

**Replacing inet_ntoa to inet_ntop**

**Replace:**

```
ptr = inet_ntoa(foo.sin_addr);
```

***with***

```
char str[INET_ADDRSTRLEN];
ptr = inet_ntop(AF_INET, &foo.sin_addr, str, sizeof(str));
```

**sock_ntop and Related Functions**

A basic problem with inet_ntop is that it requires the caller to pass a pointer to a binary address. This address is normally contained in a socket address structure, requiring the caller to know the format of the structure and the address family.

**For IPv4:**

```
struct sockaddr_in   addr;
inet_ntop(AF_INET, &addr.sin_addr, str, sizeof(str));
```

**For IPv6:**

```
struct sockaddr_in6   addr6;
inet_ntop(AF_INET6, &addr6.sin6_addr, str, sizeof(str));
```

This (above) makes our code protocol-dependent.

To solve this, we will write our own function named sock_ntop that takes a pointer to a socket address structure, looks inside the structure, and calls the appropriate function to return the presentation format of the address.

***unp_sock_ntop.h***

```
#include "unp.h"
char *sock_ntop(const struct sockaddr *sockaddr, socklen_t addrlen);
/* Returns: non-null pointer if OK, NULL on error */
```

*sockaddr* points to a socket address structure whose length is *addrlen*. The function uses its own static buffer to hold the result and a pointer to this buffer is the return value. Notice that using static storage for the result prevents the function from being **re-entrant** or **thread-safe**.

***Related functions***

There are a few other functions that we define to operate on socket address structures, and these will simplify the portability of our code between IPv4 and IPv6.

- sock_bind_wild: binds the wildcard address and an ephemeral port to a socket.
- sock_cmp_addr: compares the address portion of two socket address structures.
- sock_cmp_port: compares the port number of two socket address structures.
- sock_get_port: returns just the port number.
- sock_ntop_host: converts just the host portion of a socket address structure to presentation format (not the port number)
- sock_set_addr: sets just the address portion of a socket address structure to the value pointed to by*ptr*.
- sock_set_port: sets just the port number of a socket address structure.
- sock_set_wild: sets the address portion of a socket address structure to the wildcard

**readn, writen, and readline Functions**

Stream sockets (e.g., TCP sockets) exhibit a behavior with the read and write functions that differs from normal file I/O. A read or write on a stream socket might input or output fewer bytes than requested, but this is not an error condition. The reason is that buffer limits might be reached for the socket in the kernel. All that is required to input or output the remaining bytes is for the caller to invoke the read or write function again. This scenario is always a possibility on a stream socket with read, but is normally seen with writeonly if the socket is nonblocking.

***unp_readn.h***

**#include "unp.h"**

**ssize_t readn(int filedes, void *buff, size_t nbytes);**

**ssize_t writen(int filedes, const void *buff, size_t nbytes);**

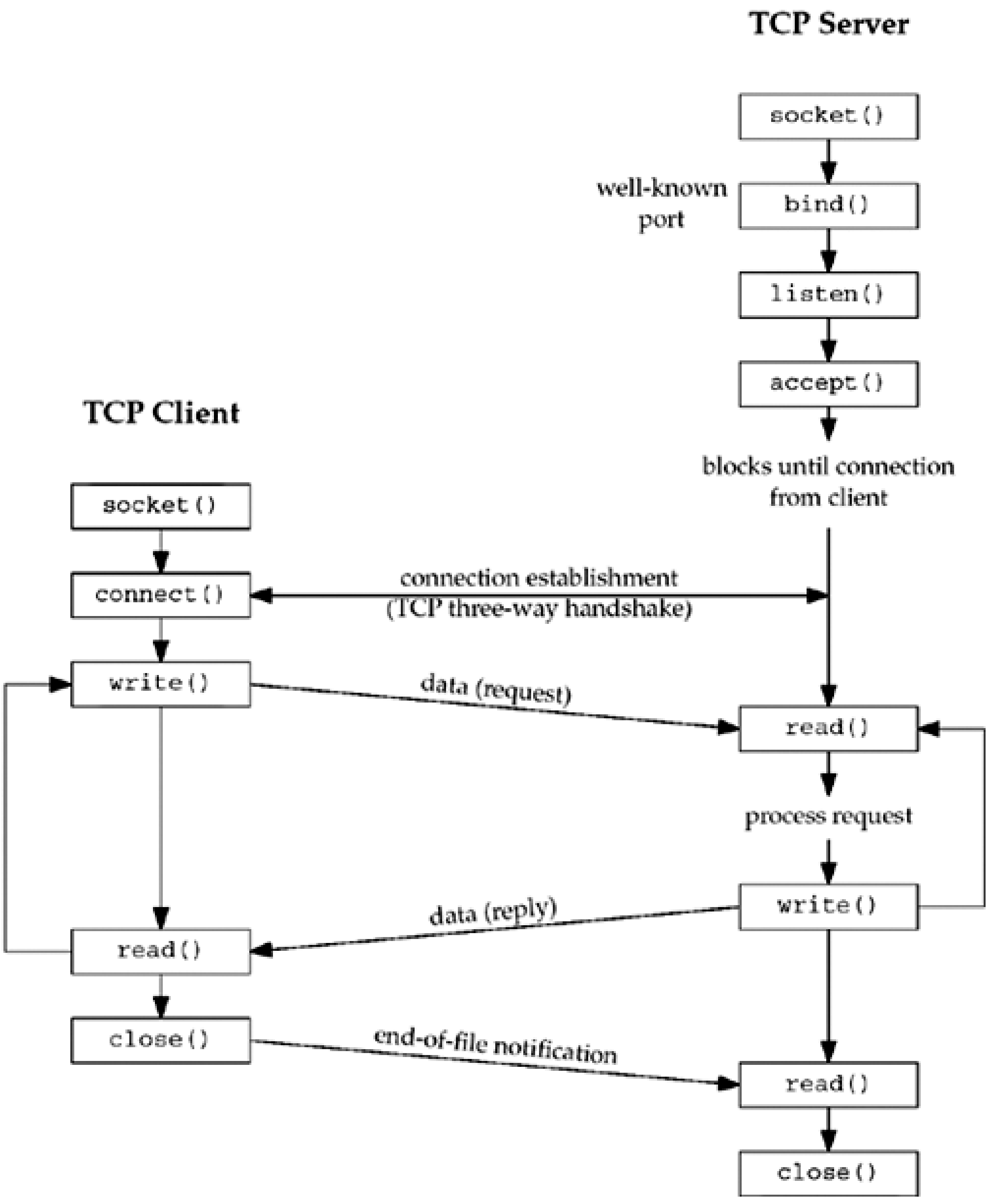
**ssize_t readline(int filedes, void *buff, size_t maxlen);**

**/* All return: number of bytes read or written, –1 on error */**

**Elementary TCP Sockets**

The elementary socket functions is required to write a complete TCP client and server, along with concurrent servers, a common Unix technique for providing concurrency when numerous clients are connected to the same server at the same time. Each client connection causes the server to fork a new process just for that client. In this chapter, we consider only the one-process-per-client model using fork.

The figure below shows a timeline of the typical scenario that takes place between a TCP client and server. First, the server is started, then sometime later, a client is started that connects to the server. We assume that the client sends a request to the server, the server processes the request, and the server sends a reply back to the client. This continues until the client closes its end of the connection, which sends an end-of-file notification to the server. The server then closes its end of the connection and either terminates or waits for a new client connection.



**socket Function**

To perform network I/O, the first thing a process must do is call the socket function, specifying the type of communication protocol desired (TCP using IPv4, UDP using IPv6, Unix domain stream protocol, etc.).

**#include <sys/socket.h>**

**int socket (int family, int type, int protocol);**

**/* Returns: non-negative descriptor if OK, -1 on error */**

**Arguments:**

- *family* specifies the protocol family and is one of the constants in the table below. This argument is often referred to as *domain* instead of *family*.
-

| ***family*** | **Description** |
|---|---|
| AF_INET | IPv4 protocols |
| AF_INET6 | IPv6 protocols |
| AF_LOCAL | Unix domain protocols |
| AF_ROUTE | Routing sockets) |
| AF_KEY | Key socket |

- The socket *type* is one of the constants shown in table below:

| ***type*** | **Description** |
|---|---|
| SOCK_STREAM | stream socket |
| SOCK_DGRAM | datagram socket |
| SOCK_SEQPACKET | sequenced packet socket |
| SOCK_RAW | raw socket |

- The *protocol* argument to the socket function should be set to the specific protocol type found in the table below, or 0 to select the system's default for the given combination of *family* and *type*.

| ***protocol*** | **Description** |
|---|---|

| ***protocol*** | **Description** |
|---|---|
| IPPROTO_TCP | TCP transport protocol |
| IPPROTO_UDP | UDP transport protocol |
| IPPROTO_SCTP | SCTP transport protocol |

Not all combinations of socket *family* and *type* are valid. The table below shows the valid combinations, along with the actual protocols that are valid for each pair. The boxes marked "Yes" are valid but do not have handy acronyms. The blank boxes are not supported.

| | | | | | |
|---|---|---|---|---|---|
| | AF_INET | AF_INET6 | AF_LOCAL | AF_ROUTE | AF_KEY |
| SOCK_STREAM | TCP/SCTP | TCP/SCTP | Yes | | |
| SOCK_DGRAM | UDP | UDP | Yes | | |
| SOCK_SEQPACKET | SCTP | SCTP | Yes | | |
| SOCK_RAW | IPv4 | IPv6 | | Yes | Yes |

On success, the socket function returns a small non-negative integer value, similar to a file descriptor. We call this a **socket descriptor**, or a *sockfd*. To obtain this socket descriptor, all we have specified is a protocol family (IPv4, IPv6, or Unix) and the socket type (stream, datagram, or raw). We have not yet specified either the local protocol address or the foreign protocol address.

***AF_xxx Versus PF_xxx***

The "AF_" prefix stands for "address family" and the "PF_" prefix stands for "protocol family." Historically, the intent was that a single protocol family might support multiple address families and that the PF_ value was used to create the socket and the AF_ value was used in socket address structures. But in actuality, a protocol family supporting multiple address families has never been supported and the <sys/socket.h>header defines the PF_ value for a given protocol to be equal to the AF_ value for that protocol. While there is no guarantee that this equality between the two will always be true, should anyone change this for existing protocols, lots of existing code would break.

To conform to existing coding practice, we use only the AF_ constants in this text, although you may encounter the PF_ value, mainly in calls to socket.

**connect Function**

The connect function is used by a TCP client to establish a connection with a TCP server.

**#include <sys/socket.h>**

**int connect(int sockfd, const struct sockaddr *servaddr, socklen_t addrlen);**

**/* Returns: 0 if OK, -1 on error */**

- *sockfd* is a socket descriptor returned by the socket function.
- The *servaddr* and *addrlen* arguments are a pointer to a socket address structure (which contains the IP address and port number of the server) and its size.

The client does not have to call bind before calling connect: the kernel will choose both an ephemeral port and the source IP address if necessary.

In the case of a TCP socket, the connect function initiates TCP's three-way handshake. The function returns only when the connection is established or an error occurs. There are several different error returns possible:

1. If the client TCP receives no response to its SYN segment, ETIMEDOUT is returned.
    - For example, in 4.4BSD, the client sends one SYN when connect is called, sends another SYN 6 seconds later, and sends another SYN 24 seconds later. If no response is received after a total of 75 seconds, the error is returned.
    - Some systems provide administrative control over this timeout.
2. If the server's response to the client's SYN is a reset (RST), this indicates that no process is waiting for connections on the server host at the port specified (the server process is probably not running). This is a **hard error** and the error ECONNREFUSED is returned to the client as soon as the RST is received. An RST is a type of TCP segment that is sent by TCP when something is wrong. Three conditions that generate an RST are:
    - When a SYN arrives for a port that has no listening server.
    - When TCP wants to abort an existing connection.
    - When TCP receives a segment for a connection that does not exist.
3. If the client's SYN elicits an ICMP "destination unreachable" from some intermediate router, this is considered a **soft error**. The client kernel saves the message but keeps sending SYNs with the same time between each SYN as in the first scenario. If no response is received after some fixed amount of time (75 seconds for 4.4BSD), the saved ICMP error is returned to the process as

either EHOSTUNREACHor ENETUNREACH. It is also possible that the remote system is not reachable by any route in the local system's forwarding table, or that the connect call returns without waiting at all. Note that network unreachables are considered obsolete, and applications should just treat ENETUNREACH andEHOSTUNREACH as the same error.

***Example: nonexistent host on the local subnet ****

We run the client daytimetcpcli and specify an IP address that is on the local subnet (192.168.1/24) but the host ID (100) is nonexistent. When the client host sends out ARP requests (asking for that host to respond with its hardware address), it will never receive an ARP reply.

```
solaris % daytimetcpcli 192.168.1.100
connect error: Connection timed out
```

We only get the error after the connect times out. Notice that our err_sys function prints the human-readable string associated with the ETIMEDOUT error.

***Example: no server process running ****

We specify a host (a local router) that is not running a daytime server:

```
solaris % daytimetcpcli 192.168.1.5
connect error: Connection refused
```

The server responds immediately with an RST.

***Example: destination not reachable on the Internet ****

Our final example specifies an IP address that is not reachable on the Internet. If we watch the packets withtcpdump, we see that a router six hops away returns an ICMP host unreachable error.

```
solaris % daytimetcpcli 192.3.4.5
connect error: No route to host
```

As with the ETIMEDOUT error, connect returns the EHOSTUNREACH error only after waiting its specified amount of time.

In terms of the TCP state transition diagram:

- connect moves from the CLOSED state (the state in which a socket begins when it is created by thesocket function) to the SYN_SENT state, and then, on success, to the ESTABLISHED state.
- If connect fails, the socket is no longer usable and must be closed. We cannot call connect again on the socket.

### bind Function

The bind function assigns a local protocol address to a socket. The protocol address is the combination of either a 32-bit IPv4 address or a 128-bit IPv6 address, along with a 16-bit TCP or UDP port number.

```
#include <sys/socket.h>

int bind (int sockfd, const struct sockaddr *myaddr, socklen_t addrlen);

/* Returns: 0 if OK,-1 on error */
```

- The second argument *myaddr* is a pointer to a protocol-specific addres
- The third argument *addrlen* is the size of this address structure.

With TCP, calling bind lets us specify a port number, an IP address, both, or neither.

- **Servers bind their well-known port when they start.** If a TCP client or server does not do this, the kernel chooses an ephemeral port for the socket when either connect or listen is called.
  - It is normal for a TCP client to let the kernel choose an ephemeral port, unless the application requires a reserved port.
  - However, it is rare for a TCP server to let the kernel choose an ephemeral port, since servers are known by their well-known port.

  Exceptions to this rule are Remote Procedure Call (RPC) servers. They normally let the kernel choose an ephemeral port for their listening socket since this port is then registered with the RPC port mapper. Clients have to contact the port mapper to obtain the ephemeral port before they can connect to the server. This also applies to RPC servers using UDP.
- **A process can bind a specific IP address to its socket.** The IP address must belong to an interface on the host.
  - For a TCP client, this assigns the source IP address that will be used for IP datagrams sent on the socket. Normally, a TCP client does not bind an IP address to its socket. The kernel chooses the source IP address when the socket is connected, based on the outgoing interface that is used, which in turn is based on the route required to reach the server
  - For a TCP server, this restricts the socket to receive incoming client connections destined only to that IP address. If a TCP server does not bind an IP address to its socket, the kernel uses the destination IP address of the client's SYN as the server's source IP address.

As mentioned, calling bind lets us specify the IP address, the port, both, or neither. The following table summarizes the values to which we set sin_addr and sin_port, or sin6_addr and sin6_port, depending on the desired result.

| **IP address** | **Port** | **Result** |
|---|---|---|
| Wildcard | 0 | Kernel chooses IP address and port |
| Wildcard | nonzero | Kernel chooses IP address, process specifies port |
| Local IP address | 0 | Process specifies IP address, kernel chooses port |
| Local IP address | nonzero | Process specifies IP address and port |

- If we specify a port number of 0, the kernel chooses an ephemeral port when bind is called.
- If we specify a wildcard IP address, the kernel does not choose the local IP address until either the socket is connected (TCP) or a datagram is sent on the socket (UDP).

***Wildcard Address and INADDR_ANY ****

With IPv4, the *wildcard* address is specified by the constant INADDR_ANY, whose value is normally 0. This tells the kernel to choose the IP address.

```
struct sockaddr_in   servaddr;
servaddr.sin_addr.s_addr = htonl (INADDR_ANY);     /* wildcard */
```

While this works with IPv4, where an IP address is a 32-bit value that can be represented as a simple numeric constant (0 in this case), we cannot use this technique with IPv6, since the 128-bit IPv6 address is stored in a structure. In C we cannot represent a constant structure on the right-hand side of an assignment. To solve this problem, we write:

```
struct sockaddr_in6    serv;
serv.sin6_addr = in6addr_any;     /* wildcard */
```

The system allocates and initializes the in6addr_any variable to the constant IN6ADDR_ANY_INIT. The<netinet/in.h> header contains the extern declaration for in6addr_any.

The value of INADDR_ANY (0) is the same in either network or host byte order, so the use of htonl is not really required. But, since all the INADDR_constants defined by the <netinet/in.h> header are defined in host byte order, we should use htonl with any of these constants.

If we tell the kernel to choose an ephemeral port number for our socket (by specifying a 0 for port number),bind does not return the chosen value. It cannot return this value since the second argument to bind has the const qualifier. To obtain the value of the ephemeral port assigned by the kernel, we must callgetsockname to return the protocol address.

***Binding a non-wildcard IP address ****

A common example of a process binding a non-wildcard IP address to a socket is a host that provides Web servers to multiple organizations:

- First, each organization has its own domain name, such as www.organization.com.
- Next, each organization's domain name maps into a different IP address, but typically on the same subnet.

For example, if the subnet is 198.69.10, the first organization's IP address could be 198.69.10.128, the next 198.69.10.129, and so on. All these IP addresses are then *aliased* onto a single network interface (using thealias option of the ifconfig command on 4.4BSD, for example) so that the IP layer will accept incoming datagrams destined for any of the aliased addresses. Finally, one copy of the HTTP server is started for each organization and each copy binds only the IP address for that organization.

An alternative technique is to run a single server that binds the wildcard address. When a connection arrives, the server calls getsockname to obtain the destination IP address from the client, which in our discussion above could be 198.69.10.128, 198.69.10.129, and so on. The server then handles the client request based on the IP address to which the connection was issued.

One advantage in binding a non-wildcard IP address is that the demultiplexing of a given destination IP address to a given server process is then done by the kernel.

**listen Function**

The listen function is called only by a TCP server and it performs two actions:

1. The listen function converts an unconnected socket into a passive socket, indicating that the kernel should accept incoming connection requests directed to this socket. In terms of the TCP state transition diagram ,the call to listen moves the socket from the CLOSED state to the LISTEN state.
   - When a socket is created by the socket function (and before calling listen), it is assumed to be an active socket, that is, a client socket that will issue a connect.
2. The second argument *backlog* to this function specifies the maximum number of connections the kernel should queue for this socket.

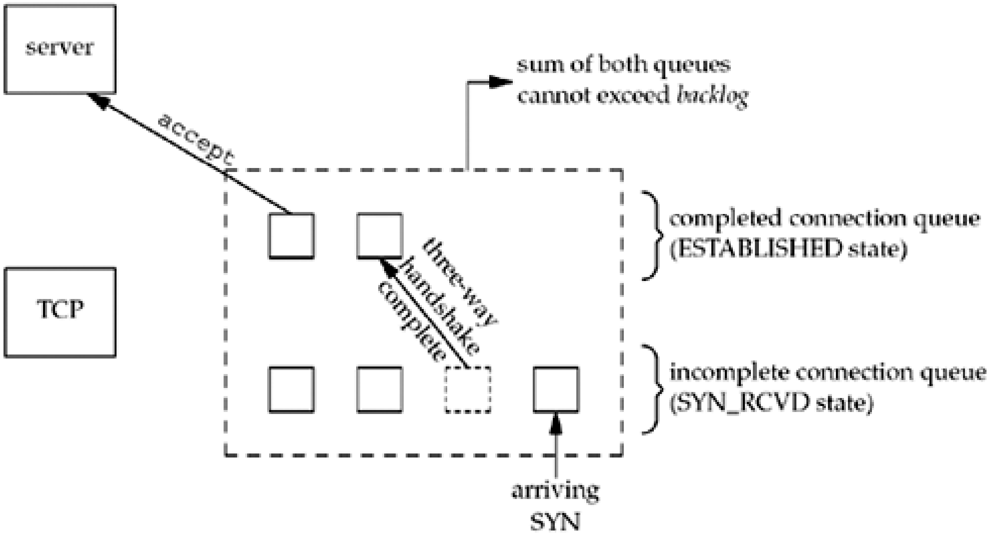
This function is normally called after both the socket and bind functions and must be called before calling the accept function.

***Connection queues ****

To understand the *backlog* argument, we must realize that for a given listening socket, the kernel maintains two queues:

1. An **incomplete connection queue**, which contains an entry for each SYN that has arrived from a client for which the server is awaiting completion of the TCP three-way handshake. These sockets are in theSYN_RCVD state
2. A **completed connection queue**, which contains an entry for each client with whom the TCP three-way handshake has completed. These sockets are in the ESTABLISHED state.

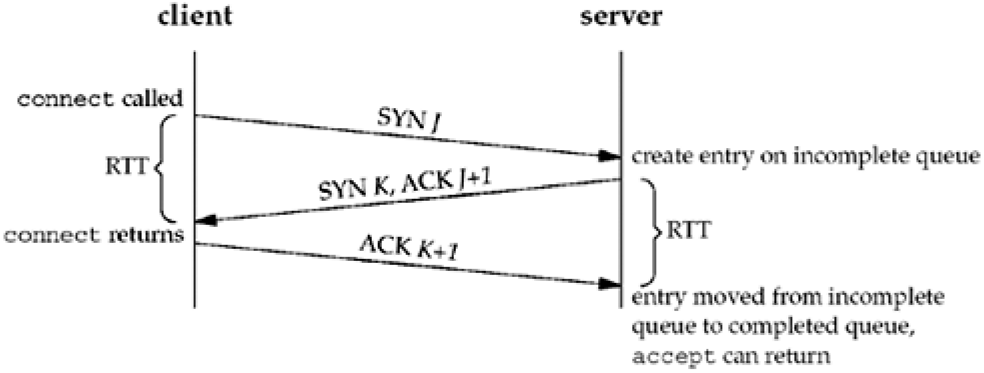
These two queues are depicted in the figure below:



When an entry is created on the incomplete queue, the parameters from the listen socket are copied over to the newly created connection. The connection creation mechanism is completely automatic; the server process is not involved.

***Packet exchanges during conenction establishment ****

The following figure depicts the packets exchanged during the connection establishment with these two queues:


client
server
connect called
RTT
connect returns
SYN J
SYN K, ACK J+1
ACK K+1
create entry on incomplete queue
RTT
entry moved from incomplete queue to completed queue, accept can return

- When a SYN arrives from a client, TCP creates a new entry on the incomplete queue and then responds with the second segment of the three-way handshake: the server's SYN with an ACK of the client's SYN.
- This entry will remain on the incomplete queue, until:
  - The third segment of the three-way handshake arrives (the client's ACK of the server's SYN), or
  - The entry times out. (Berkeley-derived implementations have a timeout of 75 seconds for these incomplete entries.)
- If the three-way handshake completes normally, the entry moves from the incomplete queue to the end of the completed queue.
- When the process calls accept:
  - The first entry on the completed queue is returned to the process, or
  - If the queue is empty, the process is put to sleep until an entry is placed onto the completed queue.

***The* backlog *argument ****

Several points to consider when handling the two queues:

- **Sum of both queues**. The *backlog* argument to the listen function has historically specified the maximum value for the sum of both queues.
- **Multiplied by 1.5**. Berkeley-derived implementations add a fudge factor to the *backlog*: It is multiplied by 1.5.
  - If the *backlog* specifies the maximum number of completed connections the kernel will queue for a socket, then the reason for the fudge factor is to take into account incomplete connections on the queue.

- **Do not specify value of 0** for *backlog*, as different implementations interpret this differently .If you do not want any clients connecting to your listening socket, close the listening socket.
- **One RTT**. If the three-way handshake completes normally (no lost segments and no retransmissions), an entry remains on the incomplete connection queue for one RTT.
- **Configurable maximum value**. Many current systems allow the administrator to modify the maximum value for the *backlog*. Historically, sample code always shows a *backlog* of 5 (which is adequate today).
- **What value should the application specify for the *backlog*** (5 is often inadequate)? There is no easy answer to this.
    - HTTP servers now specify a larger value, but if the value specified is a constant in the source code, to increase the constant requires recompiling the server.
    - Another method is to allow a command-line option or an environment variable to override the default. It is always acceptable to specify a value that is larger than supported by the kernel, as the kernel should silently truncate the value to the maximum value that it supports, without returning an error.

**SYN Flooding ***

**SYN flooding** is a type of attack (the attacker writes a program to send SYNs at a high rate to the victim) that attempts to fill the incomplete connection queue for one or more TCP ports. Additionally, the source IP address of each SYN is set to a random number (called **IP spoofing**) so that the server's SYN/ACK goes nowhere.This also prevents the server from knowing the real IP address of the attacker. By filling the incomplete queue with bogus SYNs, legitimate SYNs are not queued, providing a denial of service to legitimate clients.

The listen's *backlog* argument should specify the maximum number of completed connections for a given socket that the kernel will queue. The purpose ofto limit completed connections is to stop the kernel from accepting new connection requests for a given socket when the application is not accepting them. If a system implements this interpretation, then the application need not specify huge *backlog* values just because the server handles lots of client requests or to provide protection against SYN flooding. The kernel handles lots of incomplete connections, regardless of whether they are legitimate or from a hacker. But even with this interpretation, scenarios do occur where the traditional value of 5 is inadequate.

**accept Function**

accept is called by a TCP server to return the next completed connection from the front of the completed connection queue. If the completed connection queue is empty, the process is put to sleep (assuming the default of a blocking socket).

```
#include <sys/socket.h>
int accept (int sockfd, struct sockaddr *cliaddr, socklen_t *addrlen);
/* Returns: non-negative descriptor if OK, -1 on error */
```

The *cliaddr* and *addrlen* arguments are used to return the protocol address of the connected peer process (the client). *addrlen* is a value-result argument :

- Before the call, we set the integer value referenced by **addrlen* to the size of the socket address structure pointed to by *cliaddr*;
- On return, this integer value contains the actual number of bytes stored by the kernel in the socket address structure.

If successful, accept returns a new descriptor automatically created by the kernel. This new descriptor refers to the TCP connection with the client.

- The **listening socket** is the first argument (*sockfd*) to accept (the descriptor created by socket and used as the first argument to both bind and listen).
- The **connected socket** is the return value from accept the connected socket.

It is important to differentiate between these two sockets:

- A given server normally creates only one listening socket, which then exists for the lifetime of the server.
- The kernel creates one connected socket for each client connection that is accepted (for which the TCP three-way handshake completes).
- When the server is finished serving a given client, the connected socket is closed.

This function returns up to three values:

- An integer return code that is either a new socket descriptor or an error indication,
- The protocol address of the client process (through the *cliaddr* pointer),
- The size of this address (through the *addrlen* pointer).

If we are not interested in having the protocol address of the client returned, we set both *cliaddr* and *addrlen*to null pointers..

### Concurrent Servers

When a client request can take longer to service, we do not want to tie up a single server with one client; we want to handle multiple clients at the same time. The simplest way to write a concurrent server under Unix is to fork a child process to handle each client.

### close Function

The normal Unix close function is also used to close a socket and terminate a TCP connection.

```
#include <unistd.h>
int close (int sockfd);
/* Returns: 0 if OK, -1 on error */
```

The default action of close with a TCP socket is to mark the socket as closed and return to the process immediately. The socket descriptor is no longer usable by the process: It cannot be used as an argument toread or write. But, TCP will try to send any data that is already queued to be sent to the other end, and after this occurs, the normal TCP connection termination sequence takes place.

### getsockname and getpeername Functions

- getsockname returns the local protocol address associated with a socket.
- getpeername returns the foreign protocol address associated with a socket.

```
#include <sys/socket.h>
int getsockname(int sockfd, struct sockaddr *localaddr, socklen_t *addrlen);
int getpeername(int sockfd, struct sockaddr *peeraddr, socklen_t *addrlen);
/* Both return: 0 if OK, -1 on error */
```

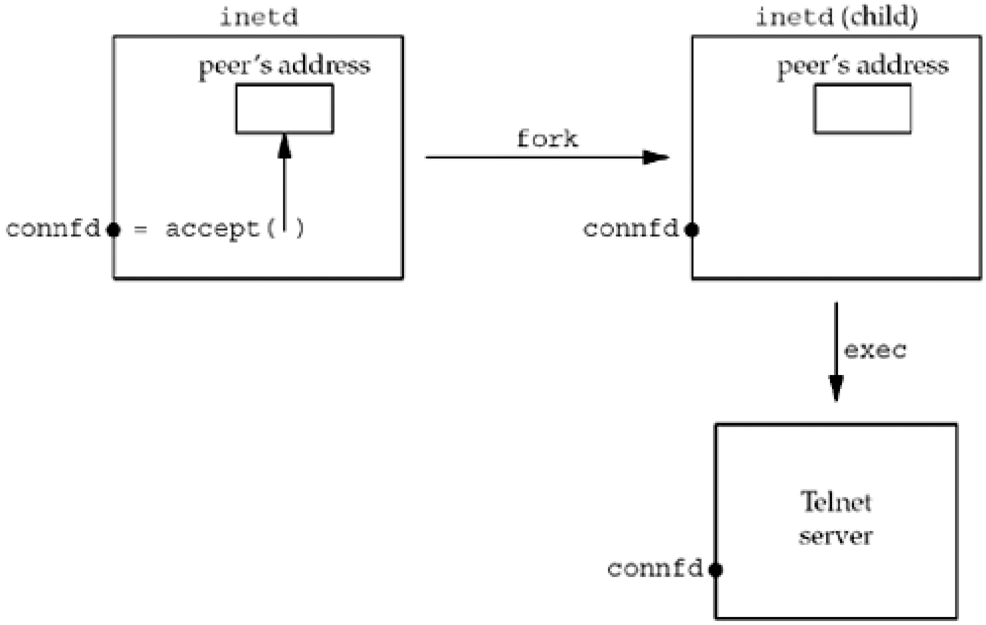
The *addrlen* argument for both functions is value-result argument: both functions fill in the socket address structure pointed to by localaddr or peeraddr.

The term "name" in the function name is misleading. These two functions return the protocol address associated with one of the two ends of a network connection, which for IPV4 and IPV6 is the combination of an IP address and port number. These functions have nothing to do with domain names.

These two functions are required for the following reasons:

- After connect successfully returns in a TCP client that does not call bind, getsockname returns the local IP address and local port number assigned to the connection by the kernel.
- After calling bind with a port number of 0 (telling the kernel to choose the local port number),getsockname returns the local port number that was assigned.

- getsockname can be called to obtain the address family of a socket.
- In a TCP server that binds the wildcard IP address, once a connection is established with a client (accept returns successfully), the server can call getsockname to obtain the local IP address assigned to the connection. The socket descriptor argument to getsocknamemust be that of the connected socket, and not the listening socket.
- When a server is execed by the process that calls accept, the only way the server can obtain the identity of the client is to call getpeername. For example, inetd forks and execs a TCP server (follwing figure):
    - inetd calls accept, which return two values: the connected socket descriptor (connfd, return value of the function) and the "peer's address" (an Internet socket address structure) that contains the IP address and port number of the client.
    - fork is called and a child of inetd is created, with a copy of the parent's memory image, so the socket address structure is available to the child, as is the connected socket descriptor (since the descriptors are shared between the parent and child).
    - When the child execs the real server (e.g. Telnet server that we show), the memory image of the child is replaced with the new program file for the Telnet server (the socket address structure containing the peer's address is lost), and the connected socket descriptor remains open across the exec. One of the first function calls performed by the Telnet server is getpeername to obtain the IP address and port number of the client.



In this example, the Telnet server must know the value of connfd when it starts. There are two common ways to do this.

1. The process calling exec pass it as a command-line argument to the newly execed program.
2. A convention can be established that a certain descriptor is always set to the connected socket before calling exec.

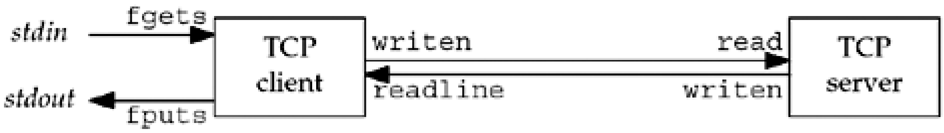
The second one is what inetd does, always setting descriptors 0, 1, and 2 to be the connected socket.

**TCP Client/Server Example**

We will now use the elementary functions from the previous sessions to write a complete TCP client/server example. Our simple example is an echo server that performs the following steps:

1. The client reads a line of text from its standard input and writes the line to the server.
2. The server reads the line from its network input and echoes the line back to the client.
3. The client reads the echoed line and prints it on its standard output.

The figure below depcits this simple client/server:



Despite two arrows between the client and server in the above figure, it is really a full-duplex TCP connection.fgets and fputs functions are from the standard I/O library. writen and readline functions were shown in the above sessions.

The echo client/server is a valid, simple example of a network application. To expand this example into your own application, all you need to do is change what the server does with the input it receives from its clients.

Besides running the client/server in normal mode (type in a line and watch it echo), we examine lots of boundary conditions:

- What happens when the client and server are started?
- What happens when the client terminates normally?
- What happens to the client if the server process terminates before the client is done?
- What happens to the client if the server host crashes?

In all these examples, we have "hard-coded" protocol-specific constants such as addresses and ports. There are two reasons for this:

- We must understand exactly what needs to be stored in the protocol-specific address structures
- We have not yet covered the library functions that can make this more portable

**TCP Echo Server: main Function**

Our TCP client and server follow the flow of functions that we diagrammed in Figure 4.1. The below code is the concurrent server program:

***tcpcliserv/tcpserv01.c***

```
#include    "unp.h"
int
main(int argc, char **argv)
{
    int             listenfd, connfd;
    pid_t           childpid;
    socklen_t       clilen;
    struct sockaddr_in  cliaddr, servaddr;
    listenfd = Socket(AF_INET, SOCK_STREAM, 0);
    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family      = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port        = htons(SERV_PORT);
    Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));
    Listen(listenfd, LISTENQ);
    for ( ; ; ) {
        clilen = sizeof(cliaddr);
        connfd = Accept(listenfd, (SA *) &cliaddr, &clilen);
        if ( (childpid = Fork()) == 0) {    /* child process */
            Close(listenfd);    /* close listening socket */
            str_echo(connfd);   /* process the request */
            exit(0);
        }
        Close(connfd);          /* parent closes connected socket */
    }
}
```

The above code does the following:

- **Create socket, bind server's well-known port**
    - A TCP socket is created.
    - An Internet socket address structure is filled in with the wildcard address (INADDR_ANY) and the server's well-known port (SERV_PORT, which is defined as 9877 in our unp.h header). Binding the wildcard address tells the system that we will accept a connection destined for any local interface, in case the system is multihomed. The socket is converted into a listening socket by listen.
- **Wait for client connection to complete**
    - The server blocks in the call to accept, waiting for a client connection to complete.
- **Concurrent server**
    - For each client, fork spawns a child, and the child handles the new client. The child closes the listening socket and the parent closes the connected socket.

**TCP Echo Server: str_echo Function**

The function str_echo performs the server processing for each client: It reads data from the client and echoes it back to the client.

***lib/str_echo.c***

```
#include    "unp.h"
void
str_echo(int sockfd)
{
   ssize_t     n;
   char       buf[MAXLINE];
again:
   while ( (n = read(sockfd, buf, MAXLINE)) > 0)
      Writen(sockfd, buf, n);
   if (n < 0 && errno == EINTR)
      goto again;
   else if (n < 0)
      err_sys("str_echo: read error");
}
```

The above code does the following:

- **Read a buffer and echo the buffer**
    - read reads data from the socket and the line is echoed back to the client by writen. If the client closes the connection (the normal scenario), the receipt of the client's FIN causes the child's read to return 0. This causes the str_echo function to return, which terminates the child.

**TCP Echo Client: main Function:** tcpcliserv/tcpcli01.c

```
#include    "unp.h"
int
main(int argc, char **argv)
{
   int              sockfd;
   struct sockaddr_in  servaddr;
   if (argc != 2)
      err_quit("usage: tcpcli <IPaddress>");
   sockfd = Socket(AF_INET, SOCK_STREAM, 0);
   bzero(&servaddr, sizeof(servaddr));
   servaddr.sin_family = AF_INET;
   servaddr.sin_port = htons(SERV_PORT);
   Inet_pton(AF_INET, argv[1], &servaddr.sin_addr);
   Connect(sockfd, (SA *) &servaddr, sizeof(servaddr));
   str_cli(stdin, sockfd);     /* do it all */
   exit(0);
}
```

The above code does the following:

- **Create socket, fill in Internet socket address structure**
    - A TCP socket is created and an Internet socket address structure is filled in with the server's IP address and port number. The server's IP address is taken from the command-line argument and the server's well-known port (SERV_PORT) is from our unp.h header.
- **Connect to server**
    - connect establishes the connection with the server. The function str_cli handles the rest of the client processing.

**TCP Echo Client: str_cli Function**

The str_cli function handles the client processing loop: It reads a line of text from standard input, writes it to the server, reads back the server's echo of the line, and outputs the echoed line to standard output.

**lib/str_cli.c**

```
#include    "unp.h"
void
str_cli(FILE *fp, int sockfd)
{
   char    sendline[MAXLINE], recvline[MAXLINE];
   while (Fgets(sendline, MAXLINE, fp) != NULL) {
                 Writen(sockfd, sendline, strlen(sendline));
       if (Readline(sockfd, recvline, MAXLINE) == 0)
           err_quit("str_cli: server terminated prematurely");
       Fputs(recvline, stdout);
   }
}
```

The above code does the following:

- **Read a line, write to server**
    - fgets reads a line of text and writen sends the line to the server.
- **Read echoed line from server, write to standard output**
    - readline reads the line echoed back from the server and fputs writes it to standard output.
- **Return to main**
    - The loop terminates when fgets returns a null pointer, which occurs when it encounters either an end-of-file (EOF) or an error. Our Fgets wrapper function checks for an error and aborts if one occurs, so Fgets returns a null pointer only when an end-of-file is encountered.

**Normal Startup**

Although the TCP example is small, it is essential that we understand:

- How the client and server start and end,
- What happens when something goes wrong:
    - the client host crashes,

- the client process crashes,
- network connectivity is lost

Only by understanding these boundary conditions, and their interaction with the TCP/IP protocols, can we write robust clients and servers that can handle these conditions.

***Start the server in the background***

First, we start the server in the background:

```
linux % tcpserv01 &
[1] 17870
```

When the server starts, it calls socket, bind, listen, and accept, blocking in the call to accept.

***Run netstat***

Before starting the client, we run the netstat program to verify the state of the server's listening socket.

```
linux % netstat -a
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address      Foreign Address      State
tcp       0      0 *:9877              *:*                 LISTEN
```

This command shows the status of all sockets on the system. We must specify the -a flag to see listening sockets.

In the output, a socket is in the LISTEN state with a wildcard for the local IP address and a local port of 9877.netstat prints an asterisk for an IP address of 0 (INADDR_ANY, the wildcard) or for a port of 0.

***Start the client on the same host***

We then start the client on the same host, specifying the server's IP address of 127.0.0.1 (the loopback address). We could have also specified the server's normal (nonloopback) IP address.

```
linux % tcpcli01 127.0.0.1
```

The client calls socket, and connect which causes TCP's three-way handshake. When the three-way handshake completes, connect returns in the client and accept returns in the server. The connection is established. The following steps then take place:

1. The client calls str_cli, which will block in the call to fgets.
2. When accept returns in the server, it calls fork and the child calls str_echo. This function callsreadline, which calls read, which blocks while waiting for a line to be sent from the client.

3. The server parent, on the other hand, calls accept again, and blocks while waiting for the next client connection.

Notes from the previous three steps:

- All three processes are asleep (blocked): client, server parent, and server child.
- We purposely list the client step first, and then the server steps when the three-way handshake completes. This is because accept returns one-half of the RTT after connect returns :
    - On the client side, connect returns when the second segment of the handshake is received
    - On the server side, accept does not return until the third segment of the handshake is received

***Run netstat after connection completes***

Since we are running the client and server on the same host, netstat now shows two additional lines of output, corresponding to the TCP connection:

```
linux % netstat -a
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address         Foreign Address        State
tcp        0      0 local host:9877       localhost:42758        ESTABLISHED
tcp        0      0 local host:42758      localhost:9877         ESTABLISHED
tcp        0      0 *:9877                *:*                    LISTEN
```

- The first ESTABLISHED line corresponds to the server child's socket, since the local port is 9877.
- The second ESTABLISHED lines is the client's socket, since the local port is 42758

If we were running the client and server on different hosts, the client host would display only the client's socket, and the server host would display only the two server sockets.

***Run ps to check process status and relationship***

```
linux % ps -t pts/6 -o pid,ppid,tty,stat,args,wchan
  PID  PPID TT       STAT COMMAND          WCHAN
22038 22036 pts/6    S    -bash            wait4
17870 22038 pts/6    S    ./tcpserv01      wait_for_connect
19315 17870 pts/6    S    ./tcpserv01      tcp_data_wait
19314 22038 pts/6    S    ./tcpcli01 127.0 read_chan
```

Very specific arguments to ps are used:

- The TT column (pts/6): client and server are run from the same window, pseudo-terminal number 6.
- The PID and PPID columns show the parent and child relationships.
    - The first tcpserv01 line is the parent and the second tcpserv01 line is the child since the PPID of the child is the parent's PID.
    - The PPID of the parent is the shell (bash).
- The STAT column for all three of our network processes is "S", meaning the process is sleeping (waiting for something).
- The WCHAN column specifies the condition when a process is asleep.
    - Linux prints wait_for_connect when a process is blocked in either accept or connect,tcp_data_wait when a process is blocked on socket input or output, or read_chan when a process is blocked on terminal I/O.
    - In ps(1), WCHAN column indicates the name of the kernel function in which the process is sleeping, a "-" if the process is running, or a "*" if the process is multi-threaded and ps is not displaying threads.

**Normal Termination**

At this point, the connection is established and whatever we type to the client is echoed back.

```
linux % tcpcli01 127.0.0.1   # we showed this line earlier
hello, world              # we now type this
hello, world              # and the line is echoed
good bye
good bye
^D                        # Control-D is our terminal EOF character
```

If we immediately execute netstat, we have:

```
linux % netstat -a | grep 9877
tcp     0     0 *:9877            *:*              LISTEN
tcp     0     0 localhost:42758     localhost:9877    TIME_WAIT
```

This time we pipe the output of netstat into grep, printing only the lines with our server's well-known port:

- The client's side of the connection (since the local port is 42758) enters the TIME_WAIT state
- The listening server is still waiting for another client connection.

The following steps are involved in the normal termination of client and server:

1. When we type our EOF character, fgets returns a null pointer and the function str_cli returns.
2. str_cli returns to the client main function which terminates by calling exit.
3. Part of process termination is the closing of all open descriptors, so the client socket is closed by the kernel. This sends a FIN to the server, to which the server TCP responds with an ACK. This is the first half of the TCP connection termination sequence. At this point, the server socket is in the CLOSE_WAIT state and the client socket is in the FIN_WAIT_2 state
4. When the server TCP receives the FIN, the server child is blocked in a call to read , and read then returns 0. This causes the str_echo function to return to the server child main.
5. The server child terminates by calling exit.
6. All open descriptors in the server child are closed.
    - The closing of the connected socket by the child causes the final two segments of the TCP connection termination to take place: a FIN from the server to the client, and an ACK from the client.
7. Finally, the SIGCHLD signal is sent to the parent when the server child terminates.
    - This occurs in this example, but we do not catch the signal in our code, and the default action of the signal is to be ignored. Thus, the child enters the zombie state. We can verify this with the pscommand.

```
linux % ps -t pts/6 -o pid,ppid,tty,stat,args,wchan
  PID  PPID TT      STAT COMMAND         WCHAN
22038 22036 pts/6    S    -bash           read_chan
17870 22038 pts/6    S    ./tcpserv01      wait_for_connect
19315 17870 pts/6    Z    [tcpserv01 <defu do_exit
```

The STAT of the child is now Z (for zombie).

We need to clean up our zombie processes and doing this requires dealing with Unix signals. The next section will give an overview of signal handling.

### POSIX Signal Handling

A **signal** is a notification to a process that an event has occurred. Signals are sometimes called **software interrupts**. Signals usually occur asynchronously, which means that a process doesn't know ahead of time exactly when a signal will occur.

Signals can be sent:

- By one process to another process (or to itself)
- By the kernel to a process.
  - For example, whenever a process terminates, the kernel send a SIGCHLD signal to the parent of the terminating process.

Every signal has a **disposition**, which is also called the **action** associated with the signal. We set the disposition of a signal by calling the sigaction function and we have three choices for the disposition:

1. **Catching a signal**. We can provide a function called a **signal handler** that is called whenever a specific signal occurs. The two signals SIGKILL and SIGSTOP cannot be caught. Our function is called with a single integer argument that is the signal number and the function returns nothing. Its function prototype is therefore:

2. **void handler (int signo);**

   For most signals, we can call sigaction and specify the signal handler to catch it. A few signals,SIGIO, SIGPOLL, and SIGURG, all require additional actions on the part of the process to catch the signal.

3. **Ignoring a signal**. We can ignore a signal by setting its disposition to SIG_IGN. The two signals SIGKILL and SIGSTOP cannot be ignored.
4. **Setting the default disposition for a signal**. This can be done by setting its disposition to SIG_DFL. The default is normally to terminate a process on receipt of a signal, with certain signals also generating a core image of the process in its current working directory. There are a few signals whose default disposition is to be ignored: SIGCHLD and SIGURG (sent on the arrival of out-of-band data) are two that we will encounter in this text.

### signal Function

The POSIX way to establish the disposition of a signal is to call the sigaction function, which is complicated in that one argument to the function is a structure (struct sigaction) that we must allocate and fill in.

An easier way to set the disposition of a signal is to call the signal function. The first argument is the signal name and the second argument is either a pointer to a function or one of the constants SIG_IGN orSIG_DFL.

However, signal is an historical function that predates POSIX. Different implementations provide different signal semantics when it is called, providing backward compatibility, whereas POSIX explicitly spells out the semantics when sigaction is called.

The solution is to define our own function named signal that just calls the POSIX sigaction function. This provides a simple interface with the desired POSIX semantics. We include this function in our own library, along with our err_XXX functions and our wrapper functions.

**lib/signal.c**

```
#include    "unp.h"
Sigfunc *
signal(int signo, Sigfunc *func)
{
    struct sigaction    act, oact;
    act.sa_handler = func;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;
    if (signo == SIGALRM) {
#ifdef  SA_INTERRUPT
        act.sa_flags |= SA_INTERRUPT;   /* SunOS 4.x */
#endif
    } else {
#ifdef  SA_RESTART
        act.sa_flags |= SA_RESTART;     /* SVR4, 44BSD */
#endif
    }
    if (sigaction(signo, &act, &oact) < 0)
        return(SIG_ERR);
    return(oact.sa_handler);
}
/* end signal */
```

```
Sigfunc *
Signal(int signo, Sigfunc *func)    /* for our signal() function */
{
   Sigfunc *sigfunc;

   if ( (sigfunc = signal(signo, func)) == SIG_ERR)
       err_sys("signal error");
   return(sigfunc);
}
```

***Simplify function prototype using typedef***

The normal function prototype for signal is complicated by the level of nested parentheses.

```
void (*signal (int signo, void (*func) (int))) (int);
```

To simplify this, we define the Sigfunc type in our unp.h header as

```
typedef    void    Sigfunc(int);
```

stating that signal handlers are functions with an integer argument and the function returns nothing (void). The function prototype then becomes

```
Sigfunc *signal (int signo, Sigfunc *func);
```

A pointer to a signal handling function is the second argument to the function, as well as the return value from the function.

***Set handler***

The sa_handler member of the sigaction structure is set to the *func* argument.

***Set signal mask for handler***

POSIX allows us to specify a set of signals that will be blocked when our signal handler is called. Any signal that is blocked cannot be delivered to a process. We set the sa_mask member to the empty set, which means that no additional signals will be blocked while our signal handler is running. POSIX guarantees that the signal being caught is always blocked while its handler is executing.

***Set SA_RESTART flag***

SA_RESTART is an optional flag. When the flag is set, a system call interrupted by this signal will be automatically restarted by the kernel.

If the signal being caught is not SIGALRM, we specify the SA_RESTART flag, if defined. This is because the purpose of generating the SIGALRM signal is normally to place a timeout on an I/O operation, in which case, we want the blocked system call to be interrupted by the signal.

***Call sigaction***

We call sigaction and then return the old action for the signal as the return value of the signal function.

Throughout this text, we will use the signal function from the above definition.

**Handling SIGCHLD Signals**

The zombie state is to maintain information about the child for the parent to fetch later, which includes:

- process ID of the child,
- termination status,
- information on the resource utilization of the child.

If a parent process of zombie children terminates, the parent process ID of all the zombie children is set to 1 (the init process), which will inherit the children and clean them up (init will wait for them, which removes the zombie).

***Handling Zombies***

Zombies take up space in the kernel and eventually we can run out of processes. Whenever we forkchildren, we must wait for them to prevent them from becoming zombies. We can establish a signal handler to catch SIGCHLD and call wait within the handler. We establish the signal handler by adding the following function call after the call to listen (in server's main function; it must be done before forking the first child and needs to be done only once.):

```
Signal (SIGCHLD, sig_chld);
```

The signal handler, the function sig_chld, is defined below:

```
#include    "unp.h"
void
sig_chld(int signo)
{
   pid_t   pid;
   int     stat;
```

```
    pid = wait(&stat);
    printf("child %d terminated\n", pid);
    return;
}
```

Note that calling standard I/O functions such as printf in a signal handler is not recommended. We callprintf here as a diagnostic tool to see when the child terminates.

**Compiling and running the program on Solaris** *

This program (tcpcliserv/tcpserv02.c) is compiled on Solaris 9 and uses the signal function from the system library.

```
solaris % tcpserv02 &            # start server in background
[2] 16939
solaris % tcpcli01 127.0.0.1       # then start client in foreground
hi there                         # we type this
hi there                         # and this is echoed
^D                               # we type our EOF character
child 16942 terminated           # output by printf in signal handler
accept error: Interrupted system call # main function aborts
```

The sequence of steps is as follows:

1. We terminate the client by typing our EOF character. The client TCP sends a FIN to the server and the server responds with an ACK.
2. The receipt of the FIN delivers an EOF to the child's pending readline. The child terminates.
3. The parent is blocked in its call to accept when the SIGCHLD signal is delivered. The sig_chldfunction executes (our signal handler), wait fetches the child's PID and termination status, andprintf is called from the signal handler. The signal handler returns.
4. Since the signal was caught by the parent while the parent was blocked in a slow system call (accept), the kernel causes the accept to return an error of EINTR (interrupted system call). The parent does not handle this error, so it aborts.

From this example, we know that when writing network programs that catch signals, we must be cognizant of interrupted system calls, and we must handle them. In this example, the signal function provided in the standard C library does not cause an interrupted system call to be automatically restarted by the kernel. Some other systems automatically restart the interrupted system call. If we run the same example under 4.4BSD, using its library version of the signal function, the kernel restarts the interrupted system call and accept does not return an error. To handle this potential problem between different operating systems is one reason we define our own version of the signal function.

As part of the coding conventions used, we always code an explicit return in our signal handlers, even though this is unnecessary for a function returning void. This reads as a reminder that the return may interrupt a system call.

***Handling Interrupted System Calls***

The term "slow system call" is used to describe any system call that can block forever, such as accept. That is, the system call need never return. Most networking functions fall into this category. Examples are:

- accept: there is no guarantee that a server's call to accept will ever return, if there are no clients that will connect to the server.
- read: the server's call to read in server's str_echo function will never return if the client never sends a line for the server to echo.

Other examples of slow system calls are reads and writes of pipes and terminal devices. A notable exception is disk I/O, which usually returns to the caller (assuming no catastrophic hardware failure).

When a process is blocked in a slow system call and the process catches a signal and the signal handler returns, the system call can return an error of EINT. Some kernels automatically restart some interrupted system calls. For portability, when we write a program that catches signals (most concurrent servers catch SIGCHLD), we must be prepared for slow system calls to return EINTR.

To handle an interrupted accept, we change the call to accept in server's main function, the beginning of the for loop, to the following:

```
for ( ; ; ) {
    clilen = sizeof (cliaddr);
    if ( (connfd = accept (listenfd, (SA *) &cliaddr, &clilen)) < 0) {
        if (errno == EINTR)
            continue;        /* back to for () */
        else
            err_sys ("accept error");
    }
```

Restarting the interrupted system call is fine for:

- accept
- read
- write
- select
- open

However, there is one function that we cannot restart: connect. If this function returns EINTR, we cannot call it again, as doing so will return an immediate error. When connect is interrupted by a caught signal and is not automatically restarted, we must call select to wait for the connection to complete.

**wait and waitpid Functions**

We can call wait function to handle the terminated child.

```
#include <sys/wait.h>
pid_t wait (int *statloc);
pid_t waitpid (pid_t pid, int *statloc, int options);
/* Both return: process ID if OK, 0 or–1 on error */
```

wait and waitpid both return two values: the return value of the function is the process ID of the terminated child, and the termination status of the child (an integer) is returned through the statloc pointer. There are three macros that we can call that examine the termination status:

- WIFEXITED: tells if the child terminated normally
- WIFSIGNALED: tells if the child was killed by a signal
- WIFSTOPPED: tells if the child was just stopped by job control

Additional macros let us then fetch the exit status of the child, or the value of the signal that killed the child, or the value of the job-control signal that stopped the child. We will use the WIFEXITED and WEXITSTATUSmacros for this purpose.

If there are no terminated children for the process calling wait, but the process has one or more children that are still executing, then wait blocks until the first of the existing children terminates.

waitpid has more control over which process to wait for and whether or not to block:

- The *pid* argument specifies the process ID that we want to wait for. A value of -1 says to wait for the first of our children to terminate.

- The *options* argument specifies additional options. The most common option is WNOHANG, which tells the kernel not to block if there are no terminated children.

***Difference between wait and waitpid***

The following example illustrates the difference between the wait and waitpid functions when used to clean up terminated children.

We modify our TCP client as below, which establishes five connections with the server and then uses only the first one (sockfd[0]) in the call to str_cli. The purpose of establishing multiple connections is to spawn multiple children from the concurrent server.

**tcpcliserv/tcpcli04.c**

```
#include    "unp.h"
int
main(int argc, char **argv)
{
    int               i, sockfd[5];
    struct sockaddr_in  servaddr;
    if (argc != 2)
        err_quit("usage: tcpcli <IPaddress>");
    for (i = 0; i < 5; i++) {
        sockfd[i] = Socket(AF_INET, SOCK_STREAM, 0);
        bzero(&servaddr, sizeof(servaddr));
        servaddr.sin_family = AF_INET;
        servaddr.sin_port = htons(SERV_PORT);
        Inet_pton(AF_INET, argv[1], &servaddr.sin_addr);
        Connect(sockfd[i], (SA *) &servaddr, sizeof(servaddr));
    }
    str_cli(stdin, sockfd[0]);      /* do it all */
    exit(0);
}
```

When the client terminates, all open descriptors are closed automatically by the kernel (we do not call close, only exit), and all five connections are terminated at about the same time. This causes five FINs to be sent, one on each connection, which in turn causes all five server children to terminate at about the same time. This causes five SIGCHLD signals to be delivered to the parent at about the same time. This causes the problem under discussion.

We first run the server **(tcpcliserv/tcpserv03.c)** in the background and then our new client:

```
linux % tcpserv03 &
[1] 20419
linux % tcpcli04 127.0.0.1
hello                   # we type this
hello                   # and it is echoed
^D                      # we then type our EOF character
child 20426 terminated      # output by server
```

Only one printf is output, when we expect all five children to have terminated. If we execute ps, we see that the other four children still exist as zombies.

```
PID TTY         TIME CMD
20419 pts/6    00:00:00 tcpserv03
20421 pts/6    00:00:00 tcpserv03 <defunct>
20422 pts/6    00:00:00 tcpserv03 <defunct>
20423 pts/6    00:00:00 tcpserv03 <defunct>
```

Establishing a signal handler and calling wait from that handler are insufficient for preventing zombies. The problem is that all five signals are generated before the signal handler is executed, and the signal handler is executed only one time because Unix signals are normally not queued.This problem is nondeterministic. Dependent on the timing of the FINs arriving at the server host, the signal handler is executed two, three or even four times.

The correct solution is to call waitpid instead of wait. The code below shows the version of oursig_chld function that handles SIGCHLD correctly. This version works because we call waitpid within a loop, fetching the status of any of our children that have terminated, with the WNOHANG option, which tells waitpid not to block if there are running children that have not yet terminated. We cannot call wait in a loop, because there is no way to prevent wait from blocking if there are running children that have not yet terminated.

**tcpcliserv/sigchldwaitpid.c**

```
#include    "unp.h"
```

```
void
sig_chld(int signo)
{
    pid_t   pid;
    int     stat;
    while ( (pid = waitpid(-1, &stat, WNOHANG)) > 0)
        printf("child %d terminated\n", pid);
    return;
}
```

The code below shows the final version of our server. It correctly handles a return of EINTR from acceptand it establishes a signal handler (code above) that calls waitpid for all terminated children.

**tcpcliserv/tcpserv04.c**

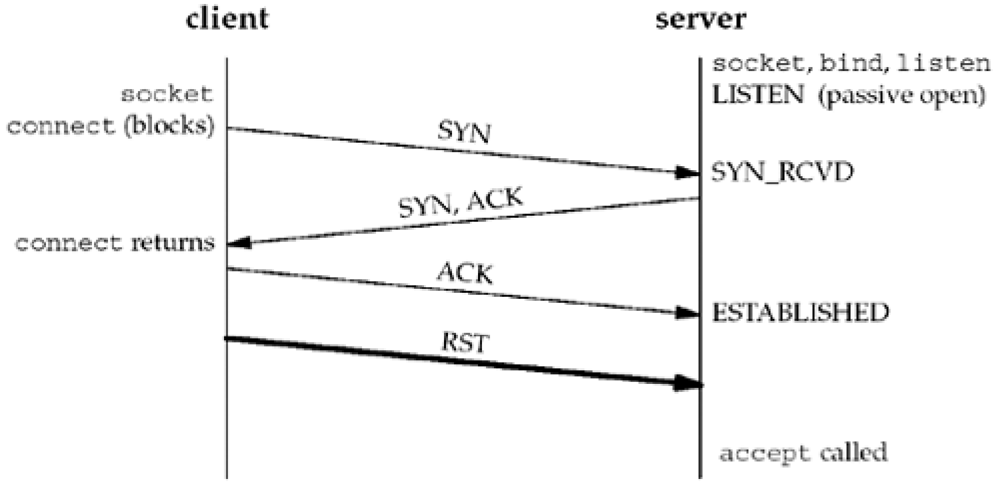
```
#include    "unp.h"
int
main(int argc, char **argv)
{
    int                listenfd, connfd;
    pid_t              childpid;
    socklen_t          clilen;
    struct sockaddr_in  cliaddr, servaddr;
    void               sig_chld(int);
    listenfd = Socket(AF_INET, SOCK_STREAM, 0);
    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family      = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port        = htons(SERV_PORT);
    Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));
    Listen(listenfd, LISTENQ);
    Signal(SIGCHLD, sig_chld);  /* must call waitpid() */
    for ( ; ; ) {
        clilen = sizeof(cliaddr);
        if ( (connfd = accept(listenfd, (SA *) &cliaddr, &clilen)) < 0) {
            if (errno == EINTR)
                continue;      /* back to for() */
            else
                err_sys("accept error");
        }
        if ( (childpid = Fork()) == 0) {    /* child process */
            Close(listenfd);    /* close listening socket */
            str_echo(connfd);   /* process the request */
            exit(0);
        }
        Close(connfd);         /* parent closes connected socket */
    }
}
```

The purpose of this section has been to demonstrate three scenarios that we can encounter with network programming:

- We must catch the SIGCHLD signal when forking child processes.
- We must handle interrupted system calls when we catch signals.
- A SIGCHLD handler must be coded correctly using waitpid to prevent any zombies from being left around.

**Connection Abort before accept Returns**

There is another condition similar to the interrupted system call that can cause accept to return a nonfatal error, in which case we should just call accept again. The sequence of packets shown below has been seen on busy servers (typically busy Web servers), where the server receives an RST for an ESTABLISHEDconnection before accept is called.



The three-way handshake completes, the connection is established, and then the client TCP sends an RST (reset). On the server side, the connection is queued by its TCP, waiting for the server process to call accept when the RST arrives. Sometime later, the server process calls accept.

An easy way to simulate this scenario is to start the server, have it call socket, bind, and listen, and then go to sleep for a short period of time before calling accept. While the server process is asleep, start the client and have it call socket and connect. As soon as connect returns, set the SO_LINGER socket option to generate the RST and terminate.

**Termination of Server Process**

We will now start our client/server and then kill the server child process, which simulates the crashing of the server process. We must be careful to distinguish between the crashing of the server *process* and the crashing of the server *host*.

The following steps take place:

1. We start the server and client and type one line to the client to verify that all is okay. That line is echoed normally by the server child.
2. We find the process ID of the server child and kill it. As part of process termination, all open descriptors in the child are closed. This causes a FIN to be sent to the client, and the client TCP responds with an ACK. This is the first half of the TCP connection termination.
3. The SIGCHLD signal is sent to the server parent and handled correctly.
4. Nothing happens at the client. The client TCP receives the FIN from the server TCP and responds with an ACK, but the problem is that the client process is blocked in the call to fgets waiting for a line from the terminal.
5. Running netstat at this point shows the state of the sockets.

```
6.     linux % netstat -a | grep 9877
7.     tcp    0    0 *:9877            *:*              LISTEN
8.     tcp    0    0 localhost:9877    localhost:43604  FIN_WAIT2
9.     tcp    1    0 localhost:43604   localhost:9877   CLOSE_WAIT
```

10. We can still type a line of input to the client. Here is what happens at the client starting from Step 1:

```
11.    linux % tcpcli01 127.0.0.1  # start client
12.    hello            # the first line that we type
13.    hello            # is echoed correctly  we kill the server child on the server host
14.    another line       # we then type a second line to the client
15.    str_cli : server terminated prematurely
```

When we type "another line," str_cli calls writen and the client TCP sends the data to the server. This is allowed by TCP because the receipt of the FIN by the client TCP only indicates that the server process has closed its end of the connection and will not be sending any more data. The receipt of the FIN does not tell the client TCP that the server process has terminated (which in this case, it has).

When the server TCP receives the data from the client, it responds with an RST since the process that had that socket open has terminated. We can verify that the RST was sent by watching the packets withtcpdump.

16. The client process will not see the RST because it calls readline immediately after the call to writen and readline returns 0 (EOF) immediately because of the FIN that was received in Step 2. Our client is not expecting to receive an EOF at this point (str_cli) so it quits with the error message "server terminated prematurely."
17. When the client terminates (by calling err_quit in str_cli), all its open descriptors are closed.
    - If the readline happens before the RST is received (as shown in this example), the result is an unexpected EOF in the client.
    - If the RST arrives first, the result is an ECONNRESET ("Connection reset by peer") error return from readline.

The problem in this example is that the client is blocked in the call to fgets when the FIN arrives on the socket. The client is really working with two descriptors,the socket and the user input. Instead of blocking on input from only one of the two sources, it should block on input from either source.

**SIGPIPE Signal**

The rules are:

- When a process writes to a socket that has received an RST, the SIGPIPE signal is sent to the process. The default action of this signal is to terminate the process, so the process must catch the signal to avoid being involuntarily terminated.
- If the process either catches the signal and returns from the signal handler, or ignores the signal, the write operation returns EPIPE.

We can simulate this from the client by performing two writes to the server (which has sent FIN to the client) before reading anything back, with the first write eliciting the RST (causing the server to send an RST to the client). We must use two writes to obtain the signal, because the first write elicits the RST and the second write elicits the signal. It is okay to write to a socket that has received a FIN, but it is an error to write to a socket that has received an RST.

We modify our client as below:

**tcpcliserv/str_cli11.c**

```
#include    "unp.h"
void
str_cli(FILE *fp, int sockfd)
{
    char    sendline[MAXLINE], recvline[MAXLINE];
    while (Fgets(sendline, MAXLINE, fp) != NULL) {


        Writen(sockfd, sendline, 1);
        sleep(1);
        Writen(sockfd, sendline+1, strlen(sendline)-1);
        if (Readline(sockfd, recvline, MAXLINE) == 0)
            err_quit("str_cli: server terminated prematurely");
        Fputs(recvline, stdout);
    }
}
```

The writen is called two times. The intent is for the first writen to elicit the RST and then for the secondwriten to generate SIGPIPE.

Run the program on the Linux host:

```
linux % tcpclill 127.0.0.1
hi there     # we type this line
hi there     # this is echoed by the server
             # here we kill the server child
bye          # then we type this line
Broken pipe  # this is printed by the shell
```

We start the client, type in one line, see that line echoed correctly, and then terminate the server child on the server host. We then type another line ("bye") and the shell tells us the process died with a SIGPIPE signal.

The recommended way to handle SIGPIPE depends on what the application wants to do when this occurs:

- If there is nothing special to do, then setting the signal disposition to SIG_IGN is easy, assuming that subsequent output operations will catch the error of EPIPE and terminate.
- If special actions are needed when the signal occurs (writing to a log file perhaps), then the signal should be caught and any desired actions can be performed in the signal handler.
- If multiple sockets are in use, the delivery of the signal will not tell us which socket encountered the error. If we need to know which write caused the error, then we must either ignore the signal or return from the signal handler and handle EPIPE from the write.

**Crashing of Server Host**

To simulate what happens when the server host crashes, we must run the client and server on different hosts. We then start the server, start the client, type in a line to the client to verify that the connection is up, disconnect the server host from the network, and type in another line at the client. This also covers the scenario of the server host being unreachable when the client sends data (i.e., some intermediate router goes down after the connection has been established).

The following steps take place:

1. When the server host crashes (which means it is not shut down by an operator), nothing is sent out on the existing network connections.

2. We type a line of input to the client, it is written by writen (str_cli), and is sent by the client TCP as a data segment. The client then blocks in the call to readline, waiting for the echoed reply.
3. With tcpdump, we will see the client TCP continually retransmitting the data segment, trying to receive an ACK from the server. Berkeley-derived implementations retransmit the data segment 12 times, waiting for around 9 minutes before giving up. When the client TCP finally gives up (assuming the server host has not been rebooted during this time, or the server host is still unreachable), an error is returned to the client process's readline. The error can be one of the following:
   - If the server host crashed and there were no responses at all to the client's data segments, the error is ETIMEDOUT.
   - If some intermediate router determined that the server host was unreachable and responded with an ICMP "destination unreachable" message, the error is either EHOSTUNREACH orENETUNREACH.

To detect that the peer is down or unreachable quicker than 9 minutes, we can place a timeout on the call toreadline.This example detects that the server host has crashed only when we send data to that host. If we want to detect the crashing of the server host even if we are not actively sending it data, another technique is required: SO_KEEPALIVE socket option.

**Crashing and Rebooting of Server Host**

In the following example, we will establish a connection between the client and server and then assume the server host crashes and reboots. The easiest way to simulate this is to establish the connection, disconnect the server from the network, shut down the server host and then reboot it, and then reconnect the server host to the network. We do not want the client to see the server host shut down.

As stated in the previous section, if the client is not actively sending data to the server when the server host crashes, the client is not aware that the server host has crashed. The following steps take place:

1. We start the server and then the client. We type a line to verify that the connection is established.
2. The server host crashes and reboots.
3. We type a line of input to the client, which is sent as a TCP data segment to the server host.
4. When the server host reboots after crashing, its TCP loses all information about connections that existed before the crash. Therefore, the server TCP responds to the received data segment from the client with an RST.
5. Our client is blocked in the call to readline when the RST is received, causing readline to return the error ECONNRESET.

If it is important for our client to detect the crashing of the server host, even if the client is not actively sending data, then some other technique, such as the SO_KEEPALIVE socket option or some client/server heartbeat function, is required.

**Shutdown of Server Host**

This section discusses what happens if the server host is shut down by an operator while our server process is running on that host.

When a Unix system is shut down, the following steps happen:

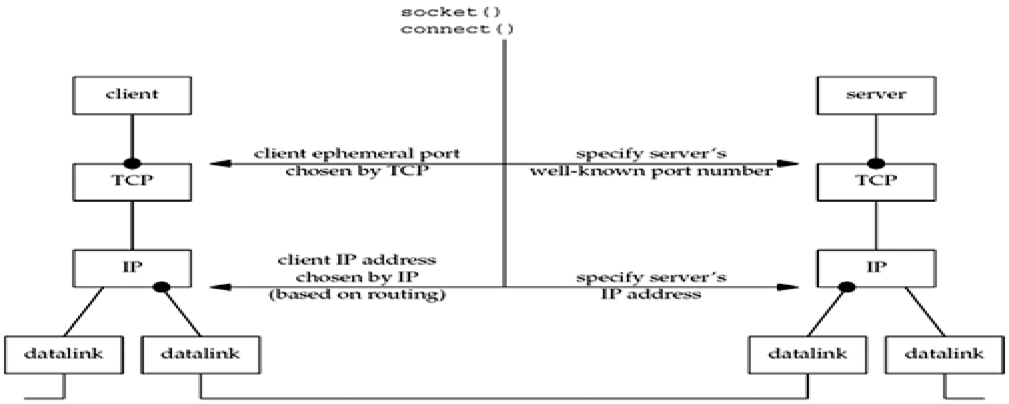
1. The init process normally sends the SIGTERM signal to all processes (we can catch this signal).
2. The init waits some fixed amount of time (often between 5 and 20 seconds).
3. The init sends the SIGKILL signal (which we cannot catch) to any processes still running.

This gives all running processes a short amount of time to clean up and terminate. When the process terminates, all open descriptors are closed (the sequence of steps are same to Termination of Server Process). We must use the select or poll function in our client to have the client detect the termination of the server process as soon as it occurs.

**Summary of TCP Example**

Before any TCP client and server can communicate with each other, each end must specify the socket pair for the connection: the local IP address, local port, foreign IP address, and foreign port. These four values are shown as bullets in the two figures below.

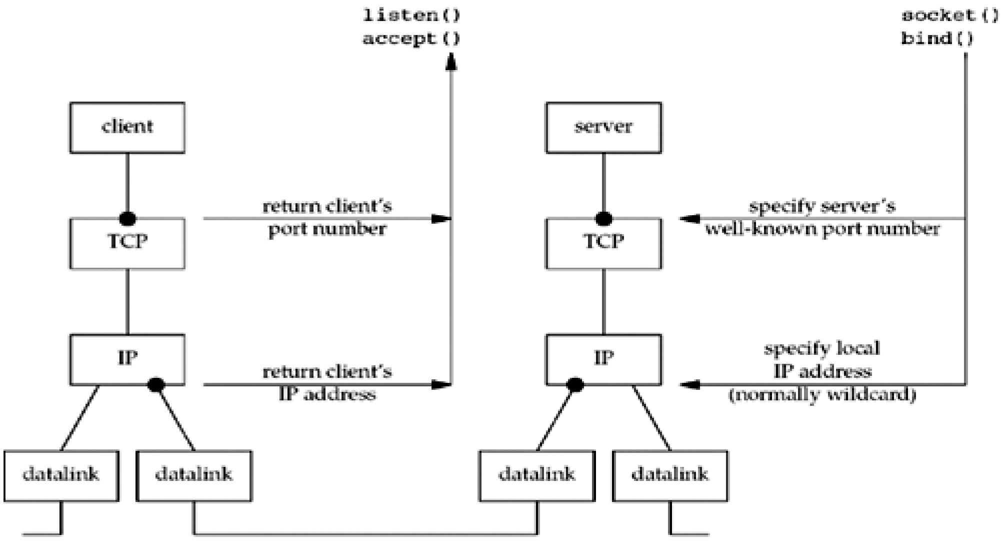
***Client's perspective***



- connect. The foreign IP address and foreign port must be specified by the client in the call toconnect. The two local values are normally chosen by the kernel as part of the connect function.
- bind. The client has the option of specifying either or both of the local values, by calling bind before connect, but this is not common.

- getsockname. The client can obtain the two local values chosen by the kernel by callinggetsockname after the connection is established.

***Server's perspective***


listen()
accept()
socket()
bind()
client
TCP
IP
datalink
datalink
return client's port number
return client's IP address
server
TCP
IP
datalink
datalink
specify server's well-known port number
specify local IP address (normally wildcard)

- bind. The local port (the server's well-known port) is specified by bind. Normally, the server also specifies the wildcard IP address in this call.
- getsockname. If the server binds the wildcard IP address on a multihomed host, it can determine the local IP address by calling getsockname after the connection is established.
- accept. The two foreign values are returned to the server by accept.
- getpeername. If another program is execed by the server that calls accept, that program can callgetpeername to determine the client's IP address and port, if necessary.

**Possible Questions**

**2 Mark Questions**

1. What is a socket pair?
2. What is the syntax of bind function?
3. What is the syntax of fork function?
4. What is the return value of fork function?
5. What is the use of listen function?

**6 Mark Questions**

1. Discuss about the IPv4 Socket Structure in Detail
2. Write a client server program that simulate Echo Client
3. Discuss in detail about fork and exec function in detail with examples.
4. What is the use of Byte Order function? Illustrate it.
5. Discuss about getpeername and getsockname function.
6. Write a program that depicts the usage of readn and written .
7. Write a simple client server program that simulates the Day-time server.
8. Discuss about concurrent server in detail.
9. What are functions used to create a passive connection. Explain with example.
10. Discuss about the IPv6 Socket Structure

**KARPAGAM ACADEMY OF HIGHER EDUCATION**
**(Deemed University Established Under Section 3 of UGC Act, 1956)**
**Coimbatore – 641021, INDIA**
**Department of Computer Science, Applications & Information Technology**

**Unit - II**

| S.No | Questions | Opt1 | Opt2 | Opt3 | Opt4 | Answer |
|---|---|---|---|---|---|---|
| 1 | TCP Sockets are also called as | virtual ports | realibale ports | communicable ports | transfer ports | virtual ports |
| 2 | Sockets is the combination of ____ and IP address together | serial number | port number | byte number | acknowledgement number | port number |
| 3 | The_____ function assigns a local protocol address to a socket | connect | close | bind | frame | bind |
| 4 | ____ is called by a TCP server to return the next completed connection from the front of the completed connection queue | connect | close | accept | frame | accept |
| 5 | Stream Sockets use | TCP | UDP | IGMP | IP | TCP |
| 6 | Datagram Sockets use | TCP | UDP | IGMP | IP | UDP |
| 7 | The name of socket address structures begin with ___ | sock_addr | socketaddress_ | sock_address | sockaddr_ | sockaddr_ |
| 8 | An socket address structure, commonly called an Internet socket address structure | IPv6 | IPv4 | Unix | datalink | IPv4 |
| 9 | IPv4 address and the TCP or UDP port number are always stored in the structure in | network byte order | host byte order | binary byte order | datalink byte order | network byte order |
| 10 | AF_INET stands for | Address family | Argument Family | Arrays Family | Acknowldegement family | Address family |
| 11 | PF_INET stands for | Protocol family | Process family | Productive family | Progress Family | Protocol family |
| 12 | When a socket address structure is passed to any socket function, it is always passed by | value | arguments | reference | pointers | reference |
| 13 | bind, connect, and ___ functions pass a socket address structure from the process to the kernel | receiveto | sendto | frameto | ackto | sendto |
| 14 | ___functions pass a socket address structure from the kernel to the process. | getformname | getlinkname | getdataname | getpeername | getpeername |
| 15 | Little-endian order is a ___ byte at the starting address | high-order | low-order | precision | string | low-order |
| 16 | Big-endian order is a _____ byte is at the starting address. | high-order | low-order | precision | string | high-order |
| 17 | sockaddr points to a socket address structure whose length is | lengthaddress | addresslength | addrlen | addr_len | addrlen |
| 18 | sock_get_port returns just the ___ | socket number | port number | host number | packet number | port number |
| 19 | To perform network I/O, the first thing a process must do is call ___ the function | socket | bind | connect | open | socket |
| 20 | The _____ and addrlen arguments are a pointer to a socket address structure | pointeraddr | socklen | servaddr | clientaddr | servaddr |
| 21 | The ____ function assigns a local protocol address to a socket | client | bind | connect | server | bind |
| 22 | With IPv4, the wildcard address is specified by the constant | ADDR_ANY | IPv4_ADDR | IPv4_ADDRESS | INADDR_ANY | INADDR_ANY |
| 23 | The ____ function converts an unconnected socket into a passive socket | socket | bind | listen | connect | listen |
| 24 | ___flooding is a type of attack that attempts to fill the incomplete connection queue for one or more TCP ports | ASYN | SYN | QUEUE | SYN_ATTK | SYN |
| 25 | _____is called by a TCP server to return the next completed connection from the front of the completed connection queue. | accept | bind | listen | connect | accept |
| 26 | The ____ socket is the return value from accept the connected socket | bind | connected | listen | connect | connected |
| 27 | ____ returns the local protocol address associated with a socket | getformname | getsockname | getdataname | getpeername | getsockname |
| 28 | The function _____ performs the server processing for each client | str_client | str_server | str_echo | str_process | str_echo |
| 29 | _____function establishes the connection with the server | bind | connect | client | server | connect |
| 30 | The ____ function handles the client processing loop | str_client | str_cli | str_echo | str_client | str_cli |
| 31 | ____reads a line of text | fline | getline | fgets | ftext | fgets |
| 32 | ___sends the line to the server | writeline | writeserver | writen | serverwrite | writen |
| 33 | ____reads the line echoed back from the server | echoread | readline | echoline | echoserver | readline |
| 34 | ___writes it to standard output | foutput | fwrite | fset | fputs | fputs |
| 35 | A ___ is a notification to a process that an event has occurred | fire | signal | trigger | start | signal |
| 36 | ____are sometimes called software interrupts | Signals | fire | trigger | start | signal |
| 37 | Every signal has a disposition, which is also called the ____ associated with the signal | fire | signal | trigger | action | action |
| 38 | We can ignore a signal by setting its disposition to ___ | SIGNAL | SIG_IGN | SIGNAL_IGN | SIGNAL_SET | SIG_IGN |
| 39 | ____ allows us to specify a set of signals | POSIX | SET | SIGNALS | ALLOW | POSIX |
| 40 | _____tells if the child terminated normally | WIFSIGNALED | WIFEXITED | WIFSTOPPED | WIFCLOSED | WIFEXITED |
| 41 | ___tells if the child was killed by a signal | WIFSIGNALED | WIFEXITED | WIFSTOPPED | WIFCLOSED | WIFSIGNALED |
| 42 | ___tells if the child was just stopped by job control | WIFSIGNALED | WIFEXITED | WIFSTOPPED | WIFCLOSED | WIFSTOPPED |
| 43 | ____argument specifies the process ID | parg | pid | sid | aid | pid |
| 44 | When a process writes to a socket that has received an RST, the____ signal is sent to the process | SIG | SOCK | WRITE | SIGPIPE | SIGPIPE |

## UNIT-III

## SYLLABUS

**I/O multiplexing using sockets:** Socket Options; UDP Sockets; UDP client server example; Address lookup using sockets.

### 1. Introduction (I/O Model):

The basic function of I/O modeling is select and poll. The five I/O models available to us under UNIX are:

- blocking I/O
- nonblocking I/O
- I/O multiplexing (select and poll)
- signal driven I/O (SIGIO)
- asynchronous I/O (the POSIX aio_functions)

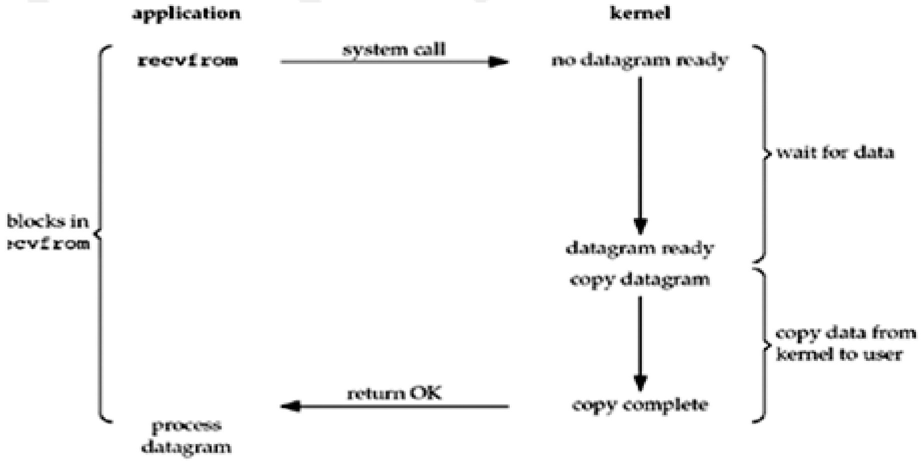
There are normally two distinct phases for an input operation:

1. Waiting for the data to be ready
2. Copying the data from the kernel to the process

For an input operation on a socket, the first step normally involves waiting for data to arrive on the network. When the packet arrives, it is copied into a buffer within the kernel. The second step is copying this data from the kernel's buffer into our application buffer

***Blocking I/O Model***

The most prevalent model for I/O is the blocking I/O model. By default, all sockets are blocking. Using a datagram socket for our examples, we have the scenario shown in figure.



**Blocking I/O Model**

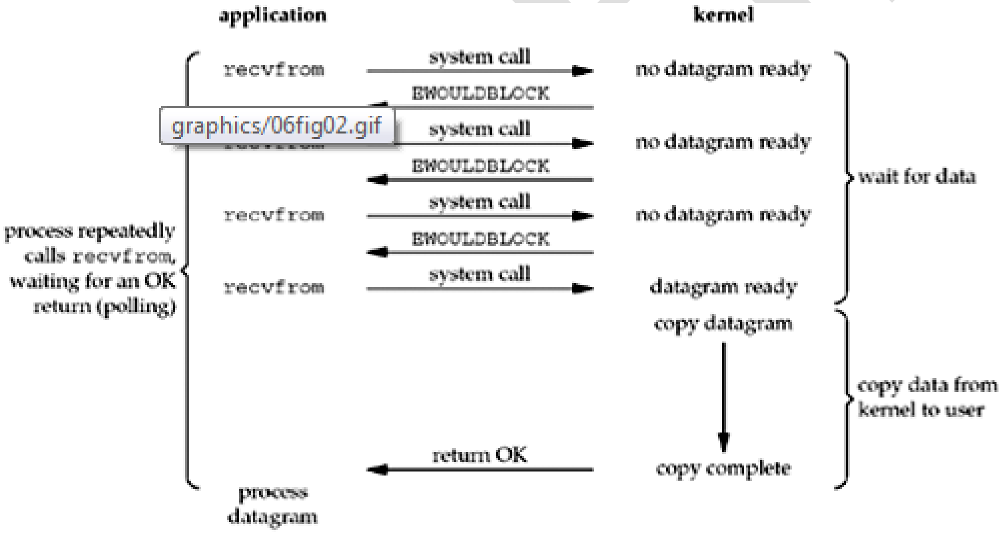
We use UDP for this example instead of TCP because with UDP, the concept of data being "ready" to read is simple: either an entire datagram has been received or it has not. With TCP it gets more complicated, as additional variables such as the socket's low-water mark come into play.

In the examples in this section, we also refer to recvfrom as a system call because we are differentiating between our application and the kernel. Regardless of how recvfrom is, there is normally a switch from running in the application to running in the kernel, followed at some time later by a return to the application.

In figure, the process calls recvfrom and the system call does not return until the datagram arrives and is copied into our application buffer, or an error occurs. The most common error is the system call being interrupted by a signal

**Non-Blocking I/O Model:**

When we set a socket to be nonblocking, we are telling the kernel "when an I/O operation that I request cannot be completed without putting the process to sleep, do not put the process to sleep, but return an error instead."



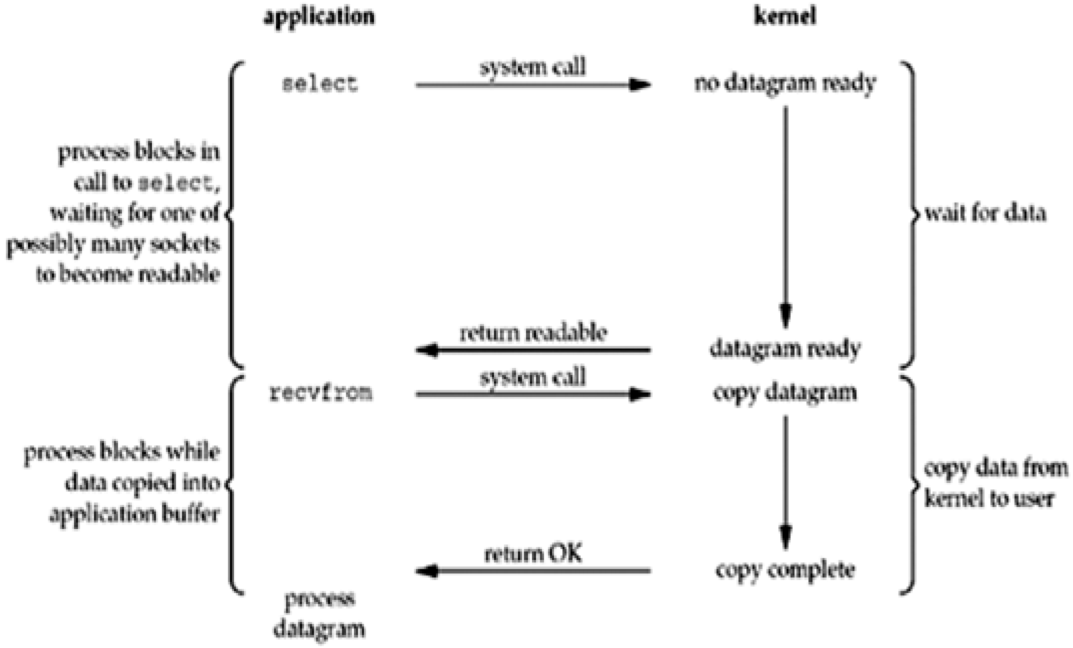
Non-Blocking I/O Model

The first three times that we call recvfrom, there is no data to return, so the kernel immediately returns an error of EWOULDBLOCK instead. The fourth time we call recvfrom, a datagram is ready, it is copied into our application buffer, and recvfrom returns successfully. We then process the data.

When an application sits in a loop calling recvfrom on a nonblocking descriptor like this, it is called polling. The application is continually polling the kernel to see if some operation is ready. This is often a waste of CPU time, but this model is occasionally encountered, normally on systems dedicated to one function.

**I/O Multiplexing Model:**

With I/O multiplexing, we call select or poll and block in one of these two system calls, instead of blocking in the actual I/O system call. We block in a call to select, waiting for the datagram socket to be readable. When select returns that the socket is readable, we then call recvfrom to copy the datagram into our application buffer.Comparing I/O Multiplexing to Basic I/O , there does not appear to be any advantage, and in fact, there is a slight disadvantage because using select requires two system calls instead of one. But the advantage in using select. Another closely related I/O model is to use multithreading with blocking I/O. That model very closely resembles the model described above, except that instead of using select to block on multiple file descriptors, the program uses multiple threads (one per file descriptor), and each thread is then free to call blocking system calls like recvfrom.


application
kernel
select
system call
no datagram ready
process blocks in call to select, waiting for one of possibly many sockets to become readable
wait for data
return readable
datagram ready
recvfrom
system call
copy datagram
process blocks while data copied into application buffer
copy data from kernel to user
return OK
copy complete
process datagram

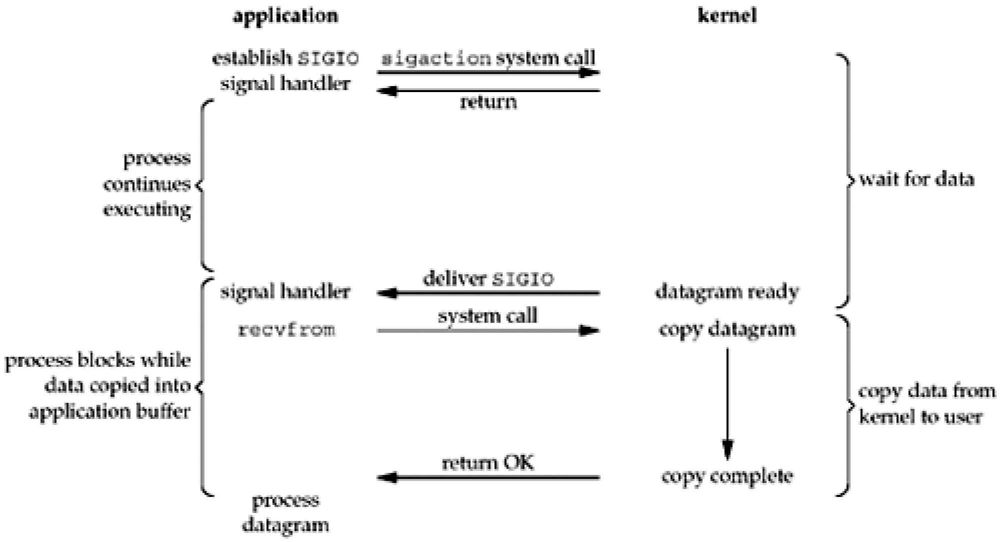
I/O Multiplexing

**Signal Driven I/O Model:**

We can also use signals, telling the kernel to notify us with the SIGIO signal when the descriptor is ready.

We first enable the socket for signal-driven I/O and install a signal handler using the sigaction system call. The return from this system call is immediate and our process continues; it is not blocked. When the datagram is ready to be read, the SIGIO signal is generated for our process. We can either read the datagram from the signal handler by calling recvfrom and then notify the main loop that the data is ready to be processed or we can notify the main loop and let it read the datagram.

Regardless of how we handle the signal, the advantage to this model is that we are not blocked while waiting for the datagram to arrive. The main loop can continue executing and just wait to be notified by the signal handler that either the data is ready to process or the datagram is ready to be read.



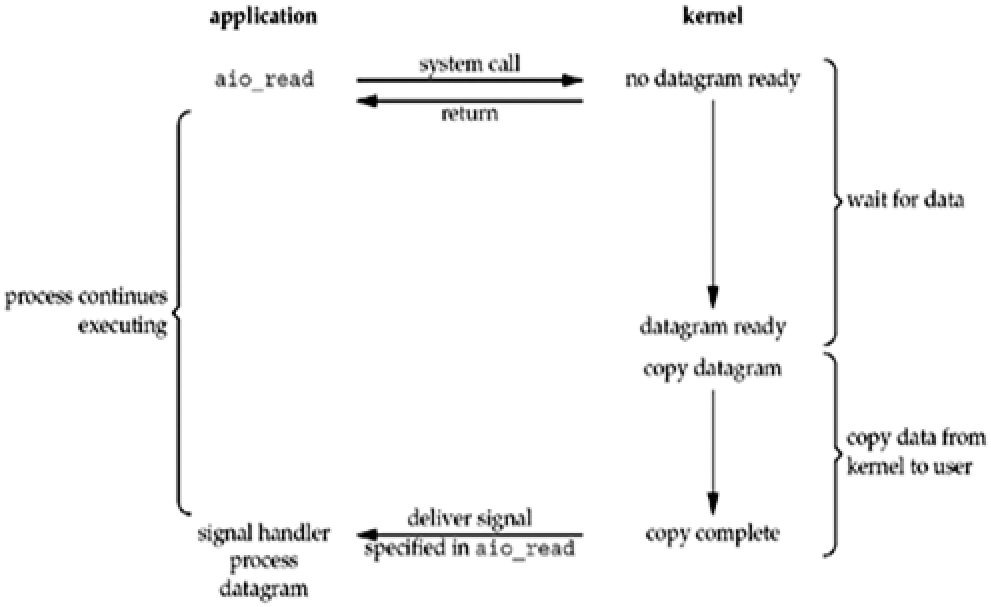
**Signal Driven I/O Model**

**Asynchronous I/O Model:**

Asynchronous I/O is defined by the POSIX specification, and various differences in the real-time functions that appeared in the various standards which came together to form the current POSIX specification have been reconciled. In general, these functions work by telling the kernel to start the operation and to notify us when the entire operation (including the copy of the data from the kernel to our buffer) is complete. The main difference between this model and the signal-driven I/O model in the previous section is that with signal-driven I/O, the kernel tells us when an I/O operation can be initiated, but with asynchronous I/O, the kernel tells us when an I/O operation is complete.

We call aio_read (the POSIX asynchronous I/O functions begin with aio_ or lio_) and pass the kernel the descriptor, buffer pointer, buffer size (the same three arguments for read), file offset (similar to lseek), and how to notify us when the entire operation is complete. This system call returns immediately and our process is not blocked while waiting for the I/O to complete. We assume in this example that we ask the kernel to generate some signal when the operation is complete. This signal is not generated until the data has been copied into our application buffer, which is different from the signal-driven I/O model.

As of this writing, few systems support POSIX asynchronous I/O. We are not certain, for example, if systems will support it for sockets. Our use of it here is as an example to compare against the signal-driven I/O model.


application
kernel
aio_read
system call
return
no datagram ready
wait for data
process continues executing
datagram ready
copy datagram
copy data from kernel to user
signal handler process datagram
deliver signal specified in aio_read
copy complete

Asynchronous I/O Model

***Synchronous I/O versus Asynchronous I/O***

***POSIX defines these two terms as follows:***

- A synchronous I/O operation causes the requesting process to be blocked until that I/O operation completes.
- An asynchronous I/O operation does not cause the requesting process to be blocked.

Using these definitions, the first four I/O models—blocking, nonblocking, I/O multiplexing, and signal-driven I/O—are all synchronous because the actual I/O operation (recvfrom) blocks the process. Only the asynchronous I/O model matches the asynchronous I/O definition.

### select Function

This function allows the process to instruct the kernel to wait for any one of multiple events to occur and to wake up the process only when one or more of these events occurs or when a specified amount of time has passed.

As an example, we can call select and tell the kernel to return only when:

- Any of the descriptors in the set {1, 4, 5} are ready for reading
- Any of the descriptors in the set {2, 7} are ready for writing
- Any of the descriptors in the set {1, 4} have an exception condition pending
- 10.2 seconds have elapsed

That is, we tell the kernel what descriptors we are interested in (for reading, writing, or an exception condition) and how long to wait. The descriptors in which we are interested are not restricted to sockets; any descriptor can be tested using select.

Berkeley-derived implementations have always allowed I/O multiplexing with any descriptor. SVR3 originally limited I/O multiplexing to descriptors that were STREAMS devices.

```
#include <sys/select.h>

#include <sys/time.h>

int select(int maxfdp1, fd_set *readset, fd_set *writeset, fd_set *exceptset, const struct timeval *timeout);

Returns: positive count of ready descriptors, 0 on timeout, –1 on error
```

A timeval structure specifies the number of seconds and microseconds.

```
struct timeval  {
  long   tv_sec;         /* seconds */
  long   tv_usec;        /* microseconds */
};
```

There are three possibilities:

1. Wait forever— Return only when one of the specified descriptors is ready for I/O. For this, we specify the timeout argument as a null pointer.

2. Wait up to a fixed amount of time— Return when one of the specified descriptors is ready for I/O, but do not wait beyond the number of seconds and microseconds specified in the timeval structure pointed to by the timeout argument.
3. Do not wait at all— Return immediately after checking the descriptors. This is called polling. To specify this, the timeout argument must point to a timeval structure and the timer value (the number of seconds and microseconds specified by the structure) must be 0.

The wait in the first two scenarios is normally interrupted if the process catches a signal and returns from the signal handler.

The three middle arguments, readset, writeset, and exceptset, specify the descriptors that we want the kernel to test for reading, writing, and exception conditions. There are only two exception conditions currently supported:

1. The arrival of out-of-band data for a socket.
2. The presence of control status information to be read from the master side of a pseudo-terminal that has been put into packet mode.

```
void FD_ZERO(fd_set *fdset);                    /* clear all bits in fdset */

void FD_SET(int fd, fd_set *fdset);             /* turn on the bit for fd in fdset */

void FD_CLR(int fd, fd_set *fdset);             /* turn off the bit for fd in fdset */

int FD_ISSET(int fd, fd_set *fdset);            /* is the bit for fd on in fdset ? */
```

We allocate a descriptor set of the fd_set datatype, we set and test the bits in the set using these macros, and we can also assign it to another descriptor set across an equals sign (=) in C.

```
fd_set rset;
FD_ZERO(&rset);       /* initialize the set: all bits off */
FD_SET(1, &rset);     /* turn on bit for fd 1 */
FD_SET(4, &rset);     /* turn on bit for fd 4 */
FD_SET(5, &rset);     /* turn on bit for fd 5 */
```

The maxfdp1 argument specifies the number of descriptors to be tested. Its value is the maximum descriptor to be tested plus one (hence our name of maxfdp1). The descriptors 0, 1, 2, up through and including maxfdp1–1 are tested.

The constant FD_SETSIZE, defined by including <sys/select.h>, is the number of descriptors in the fd_set datatype. Its value is often 1024, but few programs use that many descriptors.

select modifies the descriptor sets pointed to by the readset, writeset, and exceptset pointers. These three arguments are value-result arguments. When we call the function, we specify the values of the descriptors that we are interested in, and on return, the result indicates which descriptors are ready. We use the FD_ISSET macro on return to test a specific descriptor in an fd_set structure.

Any descriptor that is not ready on return will have its corresponding bit cleared in the descriptor set. To handle this, we turn on all the bits in which we are interested in all the descriptor sets each time we call select.

**str_cli Function**

This function, handles the client processing loop: It reads a line of text from standard input, writes it to the server, reads back the server's echo of the line, and outputs the echoed line to standard output.

***str_cli function: client processing loop.***

*lib/str_cli.c*

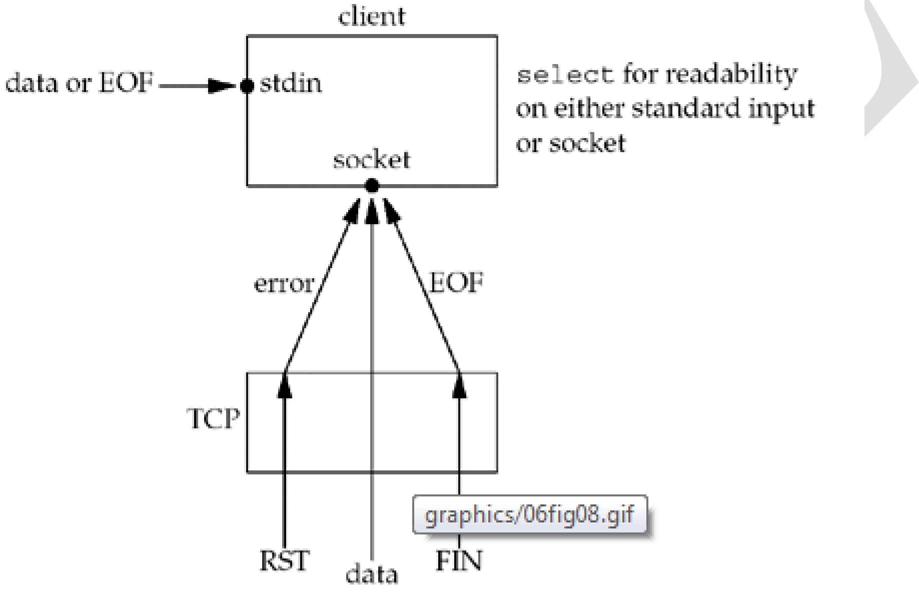
```
#include    "unp.h"
void
str_cli(FILE *fp, int sockfd)
{
        char    sendline[MAXLINE], recvline[MAXLINE];
        while (Fgets(sendline, MAXLINE, fp) != NULL) {
        Writen(sockfd, sendline, strlen (sendline));
        if (Readline(sockfd, recvline, MAXLINE) == 0)
                err_quit("str_cli: server terminated prematurely");
        Fputs(recvline, stdout);
    }}
```

### *Read a line, write to server*

*6–7* fgets reads a line of text and writen sends the line to the server. Read echoed line from server, write to standard output *8–10* readline reads the line echoed back from the server and fputs writes it to standard output.

### *Return to main*

*11–12* The loop terminates when fgets returns a null pointer, which occurs when it encounters either an end-of-file (EOF) or an error. Our Fgets wrapper function checks for an error and aborts if one occurs, so Fgets returns a null pointer only when an end-of-file is encountered.

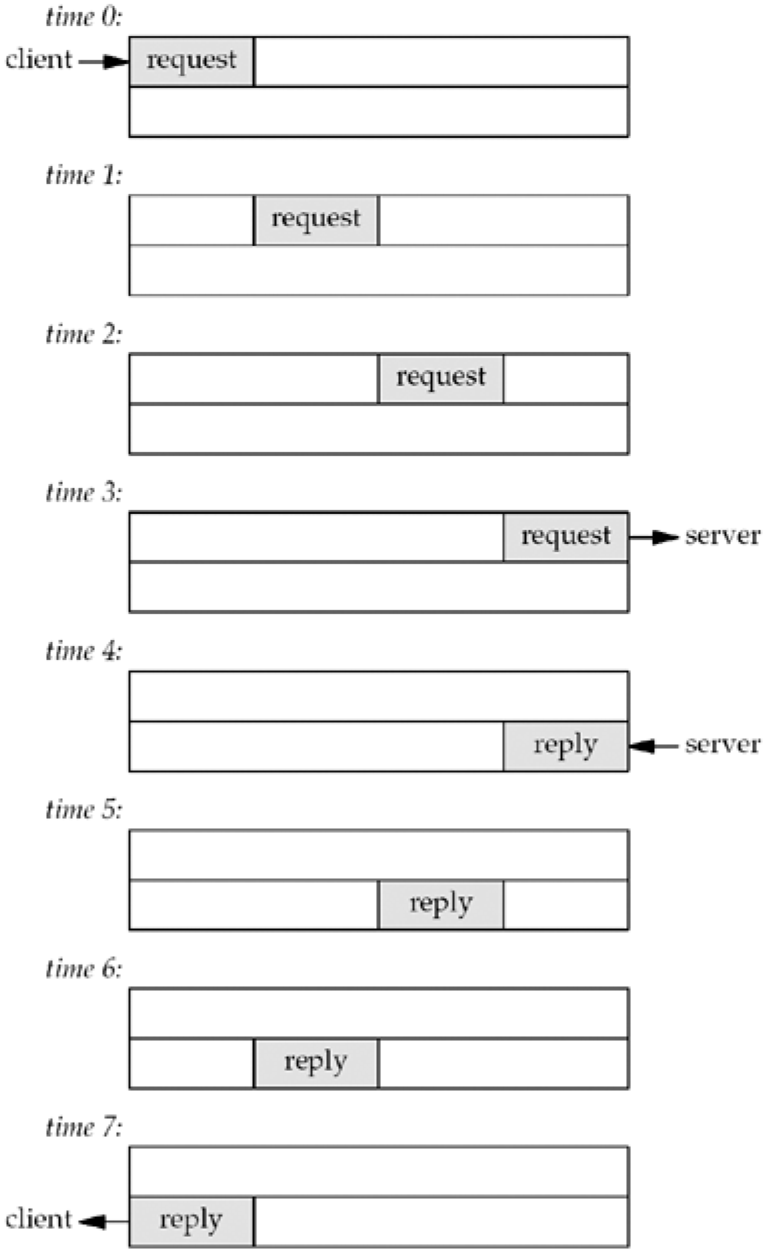


**Conditions Handled by select & str_cli**

### Batch Input and Buffering

The ping program is an easy way to measure RTTs. If we run ping to the host connix.com from our host solaris, the average RTT over 30 measurements is 175 ms. If we take the first 2,000 lines of the Solaris termcap file, the resulting file size is 98,349 bytes, for an average of 49 bytes per line. If we add the sizes of the IP header (20 bytes) and the TCP header (20), the average TCP segment will be about 89 bytes, nearly the same as the ping packet sizes. We can therefore estimate that the total clock time will be around 350 seconds for 2,000 lines (2,000x0.175sec).

If we consider the network between the client and server as a full-duplex pipe, with requests going from the client to the server and replies in the reverse direction, then figure shows our stop-and-wait mode.

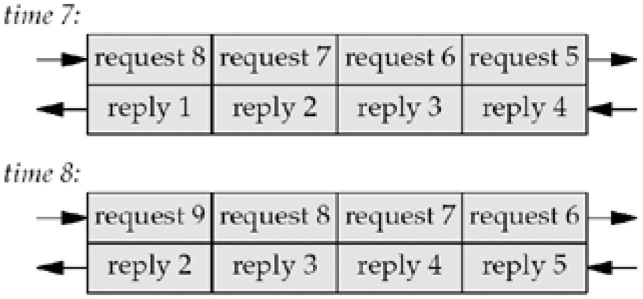


**Timeline of Stop and Wait Mode**

A request is sent by the client at time 0 and we assume an RTT of 8 units of time. The reply sent at time 4 is received at time 7. We also assume that there is no server processing time and that the size of the request is the same as the reply. We show only the data packets between the client and server, ignoring the TCP acknowledgments that are also going across the network.

Since there is a delay between sending a packet and that packet arriving at the other end of the pipe, and since the pipe is full-duplex, in this example, we are only using one-eighth of the pipe's capacity. This stop-and-wait mode is fine for interactive input, but since our client reads from standard input and writes to standard output, and since it is trivial under the Unix shells to redirect the input and output, we can easily run our client in a batch mode. When we redirect the input and output, however, the resulting output file is always smaller than the input file (and they should be identical for an echo server).

To see what's happening, realize that in a batch mode, we can keep sending requests as fast as the network can accept them. The server processes them and sends back the replies at the same rate. This leads to the full pipe at time 7, as shown below



time 7:

| | | | |
|---|---|---|---|
| request 8 | request 7 | request 6 | request 5 |
| reply 1 | reply 2 | reply 3 | reply 4 |

time 8:

| | | | |
|---|---|---|---|
| request 9 | request 8 | request 7 | request 6 |
| reply 2 | reply 3 | reply 4 | reply 5 |

**Filling the Pipe between Client a Server**

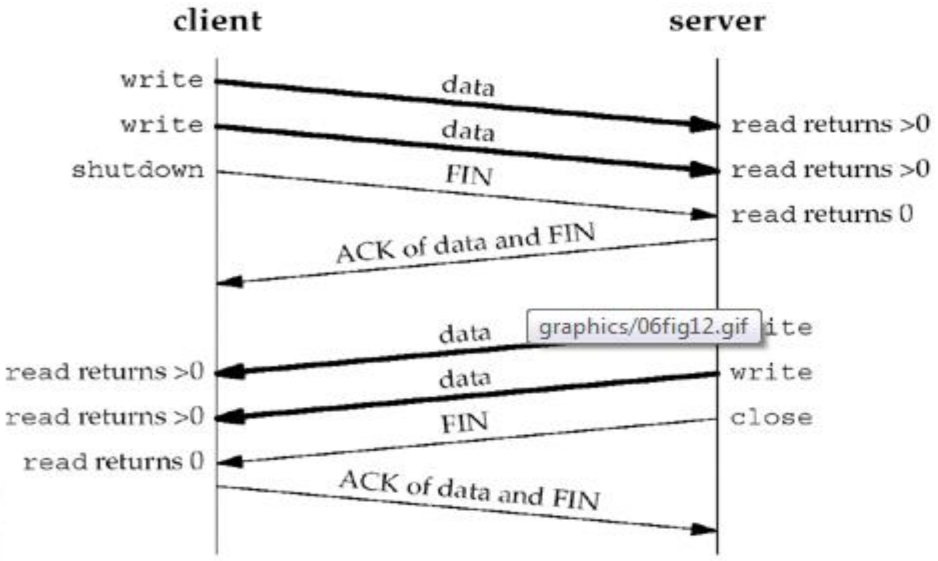
Here we assume that after sending the first request, we immediately send another, and then another. We also assume that we can keep sending requests as fast as the network can accept them, along with processing replies as fast as the network supplies them.

There are numerous subtleties dealing with TCP's bulk data flow that we are ignoring here, such as its slow-start algorithm, which limits the rate at which data is sent on a new or idle connection, and the returning ACKs. These are all covered in Chapter 20 of TCPv1.

***shutdown Function***

The normal way to terminate a network connection is to call the close function. But, there are two limitations with close that can be avoided with shutdown:

- close decrements the descriptor's reference count and closes the socket only if the count reaches 0. With shutdown, we can initiate TCP's normal connection termination, regardless of the reference count.
- close terminates both directions of data transfer, reading and writing. Since a TCP connection is full-duplex, there are times when we want to tell the other end that we have finished sending, even though that end might have more data to send us. This is the scenario we encountered in the previous section with batch input to our str_cli function.



**Calling Shutdown to close half of TCP Connection**

| |
|---|
| #include <sys/socket.h> |
| int shutdown(int sockfd, int howto); |
| Returns: 0 if OK, –1 on error |

The action of the function depends on the value of the howto argument.

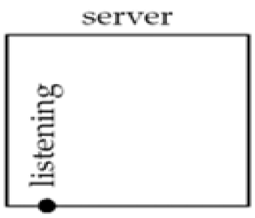
SHUT_RD: The read half of the connection is closed— No more data can be received on the socket and any data currently in the socket receive buffer is discarded. The process can no longer issue any of the read functions on the socket. Any data received after this call for a TCP socket is acknowledged and then silently discarded

SHUT_WR: The write half of the connection is closed— In the case of TCP, this is called a half-close. Any data currently in the socket send buffer will be sent, followed by TCP's normal connection termination sequence. As we mentioned earlier, this closing of the write half is done regardless of whether or not the socket descriptor's reference count is currently greater than 0. The process can no longer issue any of the write functions on the socket.

SHUT_RDWR: The read half and the write half of the connection are both closed. This is equivalent to calling shutdown twice: first with SHUT_RD and then with SHUT_WR.

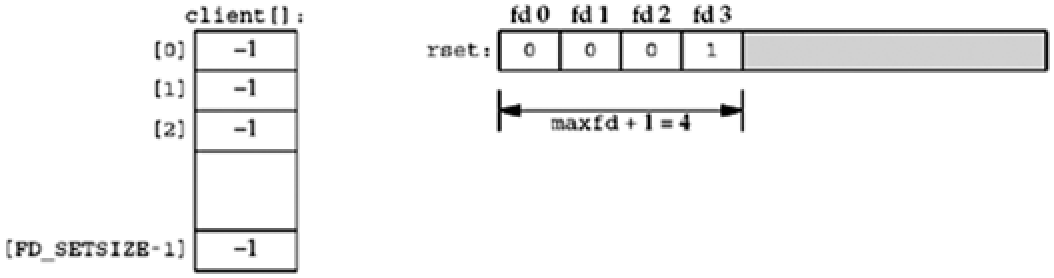
**2. TCP Echo Server**

Echo Server can be written using select function to handle any number of clients instead of forking one child per client. The Data-structure used in keeping track of the client plays a vital role. The following fig shows the state of server before the first client has established the connection.



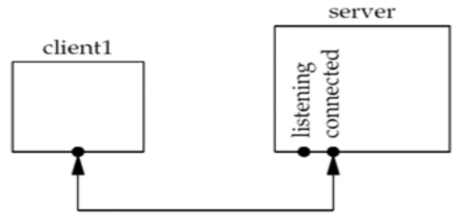
**TCP before First Client Connection**

The server has a single listening descriptor, which we show as a bullet. The server maintains only a read descriptor set, which we show in following figure. We assume that the server is started in the foreground, so descriptors 0, 1, and 2 are set to standard input, output, and error. Therefore, the first available descriptor for the listening socket is 3. We also show an array of integers named client that contains the connected socket descriptor for each client. All elements in this array are initialized to –1. The only nonzero entry in the descriptor set is the entry for the listening sockets and the first argument to select will be 4.



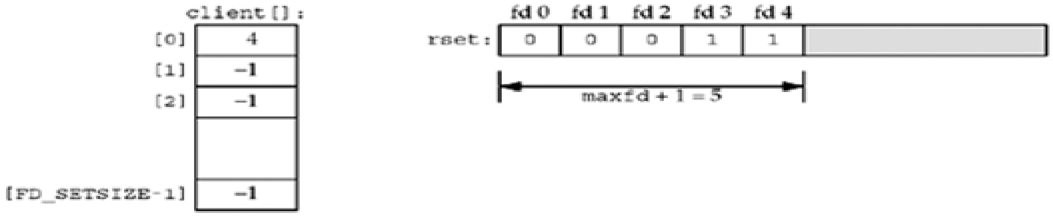
**Data Structure of TCP Server**

When the first client establishes a connection with our server, the listening descriptor becomes readable and our server calls accept. The new connected descriptor returned by accepts will be 4, given the assumptions of this example.



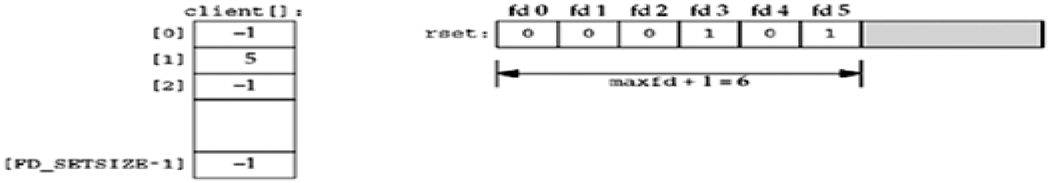
TCP Server After First Client Established Connection

From this point on, our server must remember the new connected socket in its client array, and the connected socket must be added to the descriptor set. These updated data structures are shown in figure.



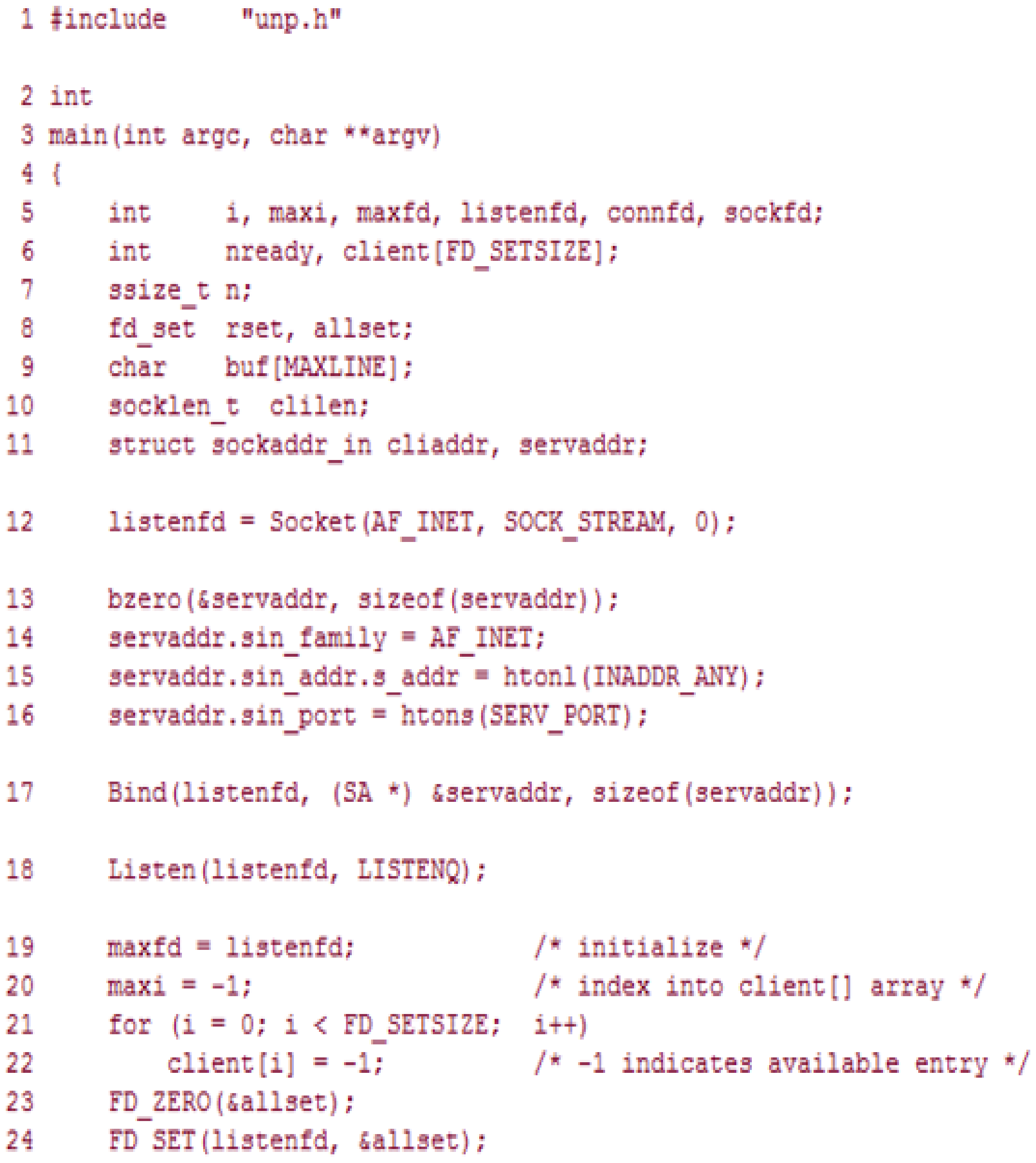
DS after Second Connection Establishment

Next, we assume the first client terminates its connection. The client TCP sends a FIN, which makes descriptor 4 in the server readable. When our server reads this connected socket, read returns 0. We then close this socket and update our data structures accordingly. The value of client [0] is set to –1 and descriptor 4 in the descriptor set is set to 0. This is shown in figure.



DS After First Client Terminate

In summary, as clients arrive, we record their connected socket descriptor in the first available entry in the client array (i.e., the first entry with a value of –1). We must also add the connected socket to the read descriptor set. The variable maxi is the highest index in the client array that is currently in use and the variable maxfd (plus one) is the current value of the first argument to select. The only limit on the number of clients that this server can handle is the minimum of the two values FD_SETSIZE and the maximum number of descriptors allowed for this process by the kernel.

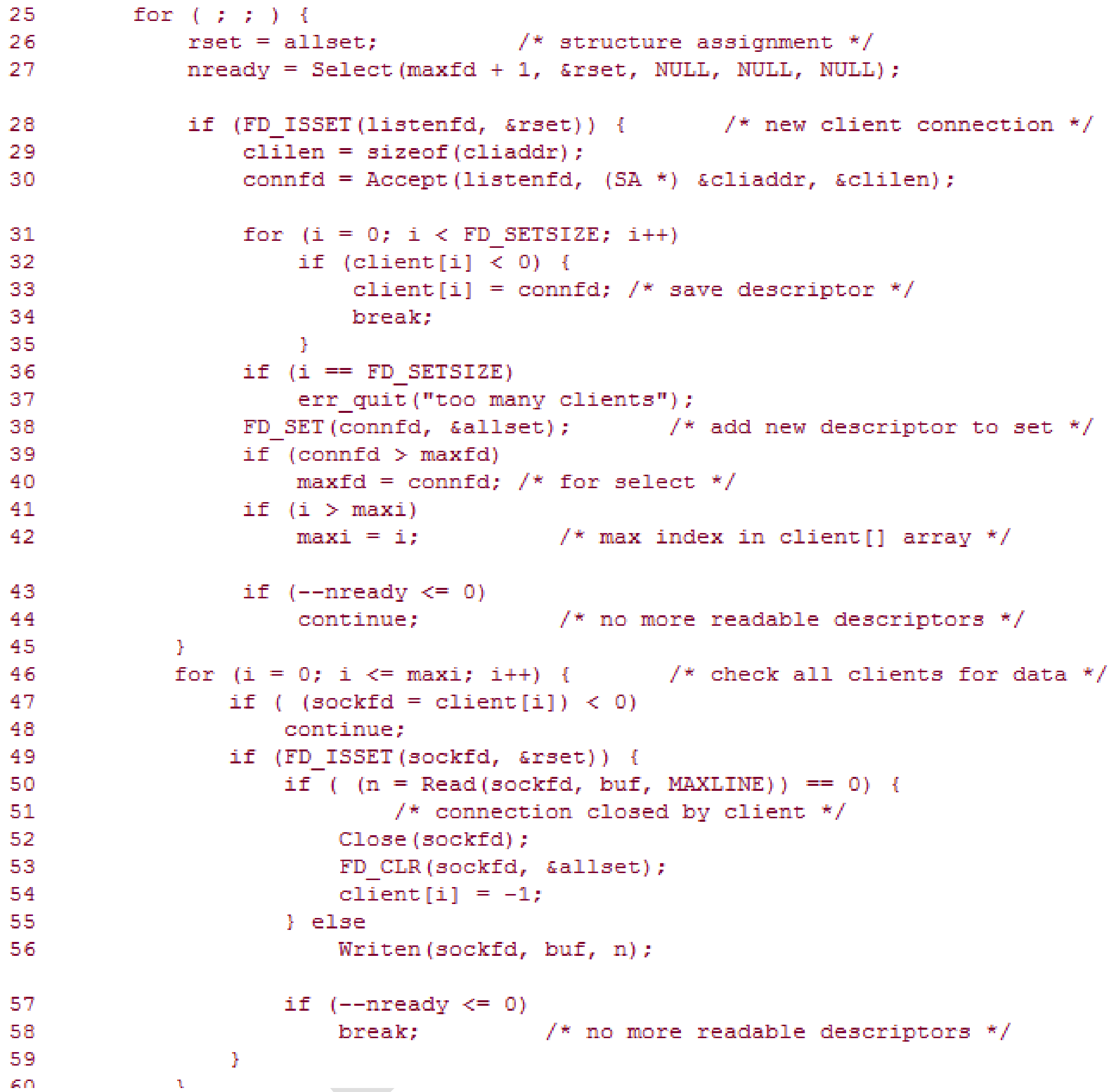


```
#include     "unp.h"

int
main(int argc, char **argv)
{
    int     i, maxi, maxfd, listenfd, connfd, sockfd;
    int     nready, client[FD_SETSIZE];
    ssize_t n;
    fd_set  rset, allset;
    char    buf[MAXLINE];
    socklen_t  clilen;
    struct sockaddr_in cliaddr, servaddr;

    listenfd = Socket(AF_INET, SOCK_STREAM, 0);

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(SERV_PORT);

    Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));

    Listen(listenfd, LISTENQ);

    maxfd = listenfd;            /* initialize */
    maxi = -1;                   /* index into client[] array */
    for (i = 0; i < FD_SETSIZE;  i++)
        client[i] = -1;          /* -1 indicates available entry */
    FD_ZERO(&allset);
    FD_SET(listenfd, &allset);
```

TCP Server using Single Process and Select Initialization

**Create listening socket and initialize for select**

12–24 the steps to create the listening socket are the same as seen earlier: socket, bind, and listen. We initialize our data structures assuming that the only descriptor that we will select on initially is the listening socket.



```
25        for ( ; ; ) {
26            rset = allset;          /* structure assignment */
27            nready = Select(maxfd + 1, &rset, NULL, NULL, NULL);

28            if (FD_ISSET(listenfd, &rset)) {       /* new client connection */
29                clilen = sizeof(cliaddr);
30                connfd = Accept(listenfd, (SA *) &cliaddr, &clilen);

31                for (i = 0; i < FD_SETSIZE; i++)
32                    if (client[i] < 0) {
33                        client[i] = connfd; /* save descriptor */
34                        break;
35                    }
36                if (i == FD_SETSIZE)
37                    err_quit("too many clients");
38                FD_SET(connfd, &allset);       /* add new descriptor to set */
39                if (connfd > maxfd)
40                    maxfd = connfd; /* for select */
41                if (i > maxi)
42                    maxi = i;           /* max index in client[] array */

43                if (--nready <= 0)
44                    continue;           /* no more readable descriptors */
45           }
46           for (i = 0; i <= maxi; i++) {       /* check all clients for data */
47               if ( (sockfd = client[i]) < 0)
48                   continue;
49               if (FD_ISSET(sockfd, &rset)) {
50                   if ( (n = Read(sockfd, buf, MAXLINE)) == 0) {
51                           /* connection closed by client */
52                       Close(sockfd);
53                       FD_CLR(sockfd, &allset);
54                       client[i] = -1;
55                   } else
56                       Writen(sockfd, buf, n);

57                   if (--nready <= 0)
58                       break;          /* no more readable descriptors */
59               }
60           }
```

***Block in select***

26–27 select waits for something to happen: either the establishment of a new client connection or the arrival of data, a FIN, or an RST on an existing connection.

***accept new connections***

28–45 If the listening socket is readable, a new connection has been established. We call accept and update our data structures accordingly. We use the first unused entry in the client array to record the connected socket. The number of ready descriptors is decremented, and if it is 0, we can avoid the next for loop. This lets us use the return value from select to avoid checking descriptors that are not ready.

***Check existing connections***

46–60 A test is made for each existing client connection as to whether or not its descriptor is in the descriptor set returned by select. If so, a line is read from the client and echoed back to the client. If the client closes the connection, read returns 0 and we update our data structures accordingly. We never decrement the value of maxi, but we could check for this possibility each time a client closes its connection.

**pselect Function:**

The pselect function was invented by POSIX and is now supported by many of the Unix variants.

| |
|---|
| #include <sys/select.h> |
| #include <signal.h> |
| #include <time.h> |
| int pselect (int maxfdp1, fd_set *readset, fd_set *writeset, fd_set *exceptset, const struct timespec *timeout, const sigset_t *sigmask); |
| Returns: count of ready descriptors, 0 on timeout, –1 on error |

pselect contains two changes from the normal select function:

1. pselect uses the timespec structure, another POSIX invention, instead of the timeval structure.

   ```
   struct timespec
   {
               time_t tv_sec;      /* seconds */
               long   tv_nsec;     /* nanoseconds */
   };
   ```

   The difference in these two structures is with the second member: The tv_nsec member of the newer structure specifies nanoseconds, whereas the tv_usec member of the older structure specifies microseconds.
2. pselect adds a sixth argument: a pointer to a signal mask. This allows the program to disable the delivery of certain signals, test some global variables that are set by the handlers for these now-disabled signals, and then call pselect, telling it to reset the signal mask.

**Poll Function:**

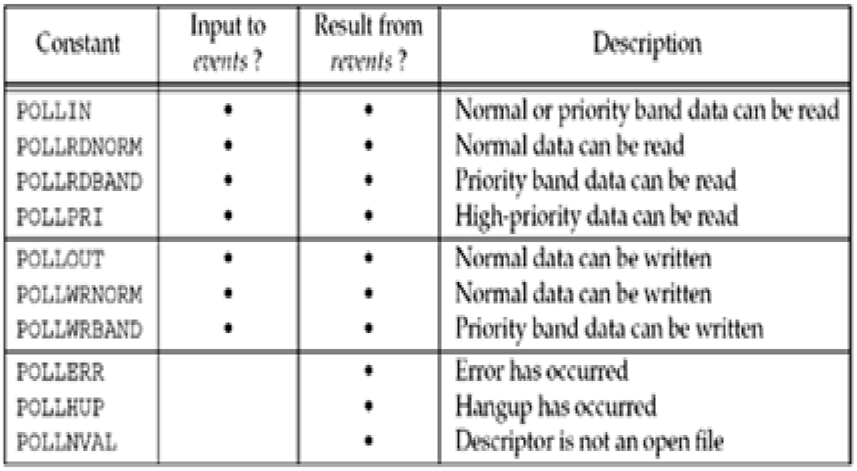
The poll function originated with SVR3 and was originally limited to STREAMS devices. SVR4 removed this limitation, allowing poll to work with any descriptor. poll provides functionality that is similar to select, but poll provides additional information when dealing with STREAMS devices.

| |
|---|
| #include <poll.h> |
| int poll (struct pollfd *fdarray, unsigned long nfds, int timeout); |
| Returns: count of ready descriptors, 0 on timeout, –1 on error |

The first argument is a pointer to the first element of an array of structures. Each element of the array is a pollfd structure that specifies the conditions to be tested for a given descriptor, fd.

```
struct pollfd
{
        int     fd;       /* descriptor to check */
        short   events;   /* events of interest on fd */
        short   revents;  /* events that occurred on fd */
};
```

The conditions to be tested are specified by the events member, and the function returns the status for that descriptor in the corresponding revents member. (Having two variables per descriptor, one a value and one a result, avoids value-result arguments. Recall that the middle three arguments for select are value-result.) Each of these two members is composed of one or more bits that specify a certain condition.



| Constant | Input to *events* ? | Result from *revents* ? | Description |
|---|---|---|---|
| POLLIN | • | • | Normal or priority band data can be read |
| POLLRDNORM | • | • | Normal data can be read |
| POLLRDBAND | • | • | Priority band data can be read |
| POLLPRI | • | • | High-priority data can be read |
| POLLOUT | • | • | Normal data can be written |
| POLLWRNORM | • | • | Normal data can be written |
| POLLWRBAND | • | • | Priority band data can be written |
| POLLERR | | • | Error has occurred |
| POLLHUP | | • | Hangup has occurred |
| POLLNVAL | | • | Descriptor is not an open file |

**I/P Events and Return Events for poll**

There are three classes of data identified by poll: normal, priority band, and high-priority. These terms come from the STREAMS-based implementations. POLLIN can be defined as the logical OR of POLLRDNORM and POLLRDBAND. The POLLIN constant exists from SVR3 implementations that predated the priority bands in SVR4, so the constant remains for backward compatibility. Similarly, POLLOUT is equivalent to POLLWRNORM, with the former predating the latter.

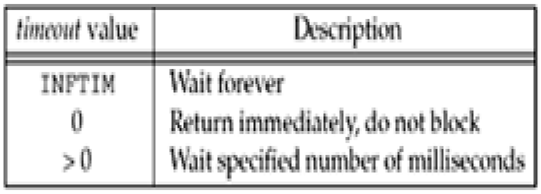
With regard to TCP and UDP sockets, the following conditions cause poll to return the specified revent. Unfortunately, POSIX leaves many holes (i.e., optional ways to return the same condition) in its definition of poll.

- All regular TCP data and all UDP data is considered normal.
- TCP's out-of-band data is considered priority band.
- When the read half of a TCP connection is closed (e.g., a FIN is received), this is also considered normal data and a subsequent read operation will return 0.
- The presence of an error for a TCP connection can be considered either normal data or an error (POLLERR). In either case, a subsequent read will return –1 with errno set to the appropriate value. This handles conditions such as the receipt of an RST or a timeout.
- The availability of a new connection on a listening socket can be considered either normal data or priority data. Most implementations consider this normal data.
- The completion of a non-blocking connect is considered to make a socket writable.

The number of elements in the array of structures is specified by the nfds argument.

Historically, this argument has been an unsigned long, which seems excessive. An unsigned int would be adequate. Unix 98 defines a new datatype for this argument: nfds_t.

The timeout argument specifies how long the function is to wait before returning. A positive value specifies the number of milliseconds to wait. Figure shows the possible values for the timeout argument.



| *timeout* value | Description |
|---|---|
| INFTIM | Wait forever |
| 0 | Return immediately, do not block |
| > 0 | Wait specified number of milliseconds |

timeout values for polls

### 3. Socket Options

There are various ways to get and set the options that affect a socket:

- The getsockopt and setsockopt functions

- The fcntl function
- The ioctl function

**getsockopt and setsockopt Functions:**

These two functions apply only to sockets:

```
#include <sys/socket.h>

int getsockopt(int sockfd, int level, int optname, void *optval, socklen_t *optlen);

int setsockopt(int sockfd, int level, int optname, const void *optval socklen_t optlen);

Both return: 0 if OK,–1 on error
```

sockfd must refer to an open socket descriptor. level specifies the code in the system that interprets the option: the general socket code or some protocol-specific code (e.g., IPv4, IPv6, TCP, or SCTP). optval is a pointer to a variable from which the new value of the option is fetched by setsockopt, or into which the current value of the option is stored by getsockopt. The size of this variable is specified by the final argument, as a value for setsockopt and as a value-result for getsockopt.

There are two basic types of options: binary options that enable or disable a certain feature (flags), and options that fetch and return specific values that we can either set or examine (values). The column labeled "Flag" specifies if the option is a flag option. When calling getsockopt for these flag options, *optval is an integer. The value returned in *optval is zero if the option is disabled, or nonzero if the option is enabled. Similarly, setsockopt requires a nonzero *optval to turn the option on, and a zero value to turn the option off. If the "Flag" column does not contain a "•," then the option is used to pass a value of the specified datatype between the user process and the system.

**Socket States**

Some socket options have timing considerations about when to set or fetch the option versus the state of the socket. We mention these with the affected options.The following socket options are inherited by a connected TCP socket from the listening socket:

SO_DEBUG, SO_DONTROUTE, SO_KEEPALIVE, SO_LINGER, SO_OOBINLINE, SO_RCVBUF, SO_RCVLOWAT, SO_SNDBUF, SO_SNDLOWAT, TCP_MAXSEG, and TCP_NODELAY.

This is important with TCP because the connected socket is not returned to a server by accept until the three-way handshake is completed by the TCP layer. To ensure that one of these socket options is set for the connected socket when the three-way handshake completes, we must set that option for the listening socket.

**Generic Socket Options**

***SO_BROADCAST* Socket Option**

This option enables or disables the ability of the process to send broadcast messages. Broadcasting is supported for only datagram sockets and only on networks that support the concept of a broadcast message (e.g., Ethernet, token ring, etc.). You cannot broadcast on a point-to-point link or any connection-based transport protocol such as SCTP or TCP.

***SO_DEBUG* Socket Option**

This option is supported only by TCP. When enabled for a TCP socket, the kernel keeps track of detailed information about all the packets sent or received by TCP for the socket. These are kept in a circular buffer within the kernel that can be examined with the trpt program. Pages 916–920 of TCPv2 provide additional details and an example that uses this option.

***SO_DONTROUTE* Socket Option**

This option specifies that outgoing packets are to bypass the normal routing mechanisms of the underlying protocol. For example, with IPv4, the packet is directed to the appropriate local interface, as specified by the network and subnet portions of the destination address. If the local interface cannot be determined from the destination address (e.g., the destination is not on the other end of a point-to-point link, or is not on a shared network), ENETUNREACH is returned

***SO_ERROR* Socket Option**

When an error occurs on a socket, the protocol module in a Berkeley-derived kernel sets a variable named so_error for that socket to one of the standard Unix Exxx values. This is called the pending error for the socket. The process can be immediately notified of the error in one of two ways:

1. If the process is blocked in a call to select on the socket, for either readability or writability, select returns with either or both conditions set.
2. If the process is using signal-driven I/O, the SIGIO signal is generated for either the process or the process group.

***SO_KEEPALIVE* Socket Option**

When the keep-alive option is set for a TCP socket and no data has been exchanged across the socket in either direction for two hours, TCP automatically sends a keep-alive probe to the peer. This probe is a TCP segment to which the peer must respond. One of three scenarios results:

1. The peer responds with the expected ACK. The application is not notified (since everything is okay). TCP will send another probe following another two hours of inactivity.

2. The peer responds with an RST, which tells the local TCP that the peer host has crashed and rebooted. The socket's pending error is set to ECONNRESET and the socket is closed.
3. There is no response from the peer to the keep-alive probe. Berkeley-derived TCPs send 8 additional probes, 75 seconds apart, trying to elicit a response. TCP will give up if there is no response within 11 minutes and 15 seconds after sending the first probe.

   HP-UX 11 treats the keep-alive probes in the same way as it would treat data, sending the second probe after a retransmission timeout and doubling the timeout for each packet until the configured maximum interval, with a default of 10 minutes.

### SO_LINGER Socket Option

This option specifies how the close function operates for a connection-oriented protocol (e.g., for TCP and SCTP, but not for UDP). By default, close returns immediately, but if there is any data still remaining in the socket send buffer, the system will try to deliver the data to the peer. The SO_LINGER socket option lets us change this default. This option requires the following structure to be passed between the user process and the kernel. It is defined by including <sys/socket.h>.

```
struct linger {
          int   l_onoff;        /* 0=off, nonzero=on */
          int   l_linger;       /* linger time, POSIX specifies units as seconds */
  };
```

Calling setsockopt leads to one of the following three scenarios, depending on the values of the two structure members:

1. If l_onoff is 0, the option is turned off. The value of l_linger is ignored and the previously discussed TCP default applies: close returns immediately.
2. If l_onoff is nonzero and l_linger is zero, TCP aborts the connection when it is closed. That is, TCP discards any data still remaining in the socket send buffer and sends an RST to the peer, not the normal four-packet connection termination sequence.
3. If l_onoff is nonzero and l_linger is nonzero, then the kernel will linger when the socket is closed. That is, if there is any data still remaining in the socket send buffer, the process is put to sleep until either: (i) all the data is sent and acknowledged by the peer TCP, or (ii) the linger time expires. If the socket has been set to nonblocking it will not wait for the close to complete, even if the linger time is nonzero. When using this feature of the SO_LINGER option, it is important for the application to check the return value from

close, because if the linger time expires before the remaining data is sent and acknowledged, close returns EWOULDBLOCK and any remaining data in the send buffer is discarded.

### 4. IPv4 Socket Options

These socket options are processed by IPv4 and have a level of IPPROTO_IP. We defer discussion of the multicasting socket options.

***IP_HDRINCL Socket Option:*** If this option is set for a raw IP socket we must build our own IP header for all the datagrams we send on the raw socket. Normally, the kernel builds the IP header for datagrams sent on a raw socket, but there are some applications (notably traceroute) that build their own IP header to override values that IP would place into certain header fields.

***IP_HDRINCL Socket Option:***If this option is set for a raw IP socket, we must build our own IP header for all the datagrams we send on the raw socket. Normally, the kernel builds the IP header for datagrams sent on a raw socket, but there are some applications (notably traceroute) that build their own IP header to override values that IP would place into certain header fields.

When this option is set, we build a complete IP header, with the following exceptions:

- IP always calculates and stores the IP header checksum.
- If we set the IP identification field to 0, the kernel will set the field.
- If the source IP address is INADDR_ANY, IP sets it to the primary IP address of the outgoing interface.
- Setting IP options is implementation-dependent. Some implementations take any IP options that were set using the IP_OPTIONS socket option and append these to the header that we build, while others require our header to also contain any desired IP options.
- Some fields must be in host byte order, and some in network byte order. This is implementation-dependent, which makes writing raw packets with IP_HDRINCL not as portable as we'd like.

***IP_OPTIONS Socket Option:*** Setting this option allows us to set IP options in the IPv4 header. This requires intimate knowledge of the format of the IP options in the IP header.

***IP_RECVDSTADDR Socket Option:*** This socket option causes the destination IP address of a received UDP datagram to be returned as ancillary data by recvmsg.

**IP_RECVIF Socket Option:** This socket option causes the index of the interface on which a UDP datagram is received to be returned as ancillary data by recvmsg.

**IP_TOS Socket Option:** This option lets us set the type-of-service (TOS) field (which contains the DSCP and ECN fields, in the IP header for a TCP, UDP, or SCTP socket. If we call getsockopt for this option, the current value that would be placed into the DSCP and ECN fields in the IP header (which defaults to 0) is returned. There is no way to fetch the value from a received IP datagram. An application can set the DSCP to a value negotiated with the network service provider to receive prearranged services, e.g., low delay for IP telephony or higher throughput for bulk data transfer. he diffserv architecture, defined in RFC 2474 [Nichols et al. 1998], provides for only limited backward compatibility with the historical TOS field definition (from RFC 1349 [Almquist 1992]). Application that set IP_TOS to one of the contents from <netinet/ip.h>, for instance, IPTOS_LOWDELAY or IPTOS_THROUGHPUT, should instead use a user-specified DSCP value. The only TOS values that diffserv retains are precedence levels 6 ("internetwork control") and 7 ("network control"); this means that applications that set IP_TOS to IPTOS_PREC_NETCONTROL or IPTOS_PREC_INTERNETCONTROL will work in a diffserv network.

**IP_TTL Socket Option:** With this option, we can set and fetch the default TTL that the system will use for unicast packets sent on a given socket. 4.4BSD, for example, uses the default of 64 for both TCP and UDP sockets (specified in the IANA's "IP Option Numbers" registry [IANA]) and 255 for raw sockets. As with the TOS field, calling getsockopt returns the default value of the field that the system will use in outgoing datagrams—there is no way to obtain the value from a received datagram.

**5. IPv6 Socket Option**

These socket options are processed by IPv6 and have a level of IPPROTO_IPV6. Note that many of these options make use of ancillary data with the recvmsg function.

***IPV6_CHECKSUM Socket Option:*** This socket option specifies the byte offset into the user data where the checksum field is located. If this value is non-negative, the kernel will: (i) compute and store a checksum for all outgoing packets, and (ii) verify the received checksum on input, discarding packets with an invalid checksum. This option affects all IPv6 raw sockets, except ICMPv6 raw sockets. (The kernel always calculates and stores the checksum for ICMPv6 raw sockets.) If a value of -1 is specified (the default), the kernel will not calculate and store the checksum for outgoing packets on this raw socket and will not verify the checksum for received packets.

***IPV6_DONTFRAG Socket Option:*** Setting this option disables the automatic insertion of a fragment header for UDP and raw sockets. When this option is set, output packets larger than the MTU of the outgoing interface will be dropped. No error needs to be returned from the system call that sends the packet, since the packet might exceed the path MTU en-route. Instead, the application should enable the IPV6_RECVPATHMTU option.

***IPV6_NEXTHOP Socket Option:*** This option specifies the next-hop address for a datagram as a socket address structure, and is a privileged operation.

***IPV6_PATHMTU Socket Option:*** This option cannot be set, only retrieved. When this option is retrieved, the current MTU as determined by path-MTU discovery is returned.

**IPV6_RECVDSTOPTS Socket Option:** Setting this option specifies that any received IPv6 destination options are to be returned as ancillary data by recvmsg. This option defaults to OFF.

**IPV6_RECVHOPLIMIT Socket Option:** Setting this option specifies that the received hop limit field is to be returned as ancillary data by recvmsg. This option defaults to OFF. There is no way with IPv4 to obtain the received TTL field.

***IPV6_RECVHOPOPTS Socket Option:*** Setting this option specifies that any received IPv6 hop-by-hop options are to be returned as ancillary data by recvmsg. This option defaults to OFF.

***IPV6_RECVPATHMTU Socket Option:*** Setting this option specifies that the path MTU of a path is to be returned as ancillary data by recvmsg (without any accompanying data) when it changes.

***IPV6_RECVPKTINFO Socket Option:*** Setting this option specifies that the following two pieces of information about a received IPv6 datagram are to be returned as ancillary data by recvmsg: the destination IPv6 address and the arriving interface index.

***IPV6_RECVRTHDR Socket Option:*** Setting this option specifies that a received IPv6 routing header is to be returned as ancillary data by recvmsg. This option defaults to OFF.

***IPV6_RECVTCLASS Socket Option:*** Setting this option specifies that the received traffic class (containing the DSCP and ECN fields) is to be returned as ancillary data by recvmsg. This option defaults to OFF.

***IPV6_UNICAST_HOPS* Socket Option:** This IPv6 option is similar to the IPv4 IP_TTL socket option. Setting the socket option specifies the default hop limit for outgoing datagrams sent on the socket, while fetching the socket option returns the value for the hop limit that the kernel will use for the socket. The actual hop limit field from a received IPv6 datagram is obtained by using the IPV6_RECVHOPLIMIT socket option.

**IPV6_USE_MIN_MTU Socket Option:** Setting this option to 1 specifies that path MTU discovery is not to be performed and that packets are sent using the minimum IPv6 MTU to avoid fragmentation. Setting it to 0 causes path MTU discovery to occur for all destinations. Setting it to–1 specifies that path MTU discovery is performed for unicast destinations but the minimum MTU is used when sending to multicast destinations. This option defaults to –1.

**IPV6_V6ONLY Socket Option:** Setting this option on an AF_INET6 socket restricts it to IPv6 communication only. This option defaults to OFF, although some systems have an option to turn it ON by default.

***IPV6_XXX* Socket Options:** Most of the IPv6 options for header modification assume a UDP socket with information being passed between the kernel and the application using ancillary data with recvmsg and sendmsg. A TCP socket fetches and stores these values using getsockopt and setsockopt instead. The socket option is the same as the type of the ancillary data, and the buffer contains the same information as would be present in the ancillary data.

### 6. TCP Socket Options

There are two socket options for TCP. We specify the level as IPPROTO_TCP.

**TCP_MAXSEG Socket Option:**This socket option allows us to fetch or set the MSS for a TCP connection. The value returned is the maximum amount of data that our TCP will send to the other end; often, it is the MSS announced by the other end with its SYN, unless our TCP chooses to use a smaller value than the peer's announced MSS. If this value is fetched before the socket is connected, the value returned is the default value that will be used if an MSS option is not received from the other end. Also be aware that a value smaller than the returned value can actually be used for the connection if the timestamp option, for example, is in use, because this option occupies 12 bytes of TCP options in each segment.

***TCP_NODELAY* Socket Option:** If set, this disables TCP's Nagle algorithm. By default, this algorithm is enabled. The purpose of the Nagle algorithm is to reduce the number of small packets on a WAN. The algorithm states that if a given connection has outstanding data (i.e., data that our TCP has sent, and for which it is currently awaiting an acknowledgment), then no small packets will be sent on the connection in response to a user write operation until the existing data is acknowledged. The definition of a "small" packet is any packet smaller than the MSS. TCP will always send a full-sized packet if possible; the purpose of the Nagle algorithm is to prevent a connection from having multiple small packets outstanding at any time.

The two common generators of small packets are the Rlogin and Telnet clients, since they normally send each keystroke as a separate packet. On a fast LAN, we normally do not notice the Nagle algorithm with these clients, because the time required for a small packet to be acknowledged is typically a few milliseconds—far less than the time between two successive characters that we type.

The Nagle algorithm often interacts with another TCP algorithm: the delayed ACK algorithm. This algorithm causes TCP to not send an ACK immediately when it receives data; instead, TCP will wait some small amount of time (typically 50–200 ms) and only then send the ACK. The hope is that in this small amount of time, there will be data to send back to the peer, and the ACK can piggyback with the data, saving one TCP segment. This is normally the case with the Rlogin and Telnet clients, because the servers typically echo each character sent by the client, so the ACK of the client's character piggybacks with the server's echo of that character.

The problem is with other clients whose servers do not generate traffic in the reverse direction on which ACKs can piggyback. These clients can detect noticeable delays because the client TCP will not send any data to the server until the server's delayed ACK timer expires. These clients need a way to disable the Nagle algorithm, hence the TCP_NODELAY option.

Another type of client that interacts badly with the Nagle algorithm and TCP's delayed ACKs is a client that sends a single logical request to its server in small pieces. For example, assume a client sends a 400-byte request to its server, but this is a 4-byte request type followed by 396 bytes of request data. If the client performs a 4-byte write followed by a 396-byte write, the second write will not be sent by the client TCP until the server TCP acknowledges the 4-byte write.

Also, since the server application cannot operate on the 4 bytes of data until it receives the remaining 396 bytes of data, the server TCP will delay the ACK of the 4 bytes of data (i.e., there will not be any data from the server to the client on which to piggyback the ACK). There are three ways to fix this type of client:

- Use writev instead of two calls to write. A single call to writev ends up with one call to TCP output instead of two calls, resulting in one TCP segment for our example. This is the preferred solution.
- Copy the 4 bytes of data and the 396 bytes of data into a single buffer and call write once for this buffer.
- Set the TCP_NODELAY socket option and continue to call write two times. This is the least desirable solution, and is harmful to the network, so it generally should not even be considered.

### 7. SCTP Socket Options

The relatively large number of socket options for SCTP (17 at present writing) reflects the finer grain of control SCTP provides to the application developer. We specify the level as IPPROTO_SCTP.

Several options used to get information about SCTP require that data be passed into the kernel (e.g., association ID and/or peer address). While some implementations of getsockopt support passing data both into and out of the kernel, not all do. The SCTP API defines a sctp_opt_info function that hides this difference. On systems on which getsockopt does support this, it is simply a wrapper around getsockopt. Otherwise, it performs the required action, perhaps using a custom ioctl or a new system call. We recommend always using sctp_opt_info when retrieving these options for maximum portability.

***SCTP_ADAPTION_LAYER Socket Option:*** During association initialization, either endpoint may specify an adaption layer indication. This indication is a 32-bit unsigned integer that can be used by the two applications to coordinate any local application adaption layer. This option allows the caller to fetch or set the adaption layer indication that this endpoint will provide to peers. When fetching this value, the caller will only retrieve the value the local socket will provide to all future peers. To retrieve the peer's adaption layer indication, an application must subscribe to adaption layer events.

***SCTP_ASSOCINFO* Socket Option:** The SCTP_ASSOCINFO socket option can be used for three purposes:

(i) to retrieve information about an existing association,

(ii) to change the parameters of an existing association, and/or

(iii) to set defaults for future associations.

When retrieving information about an existing association, the sctp_opt_info function should be used instead of getsockopt. This option takes as input the sctp_assocparams structure.

```
struct sctp_assocparams {
  sctp_assoc_t sasoc_assoc_id;
  u_int16_t sasoc_asocmaxrxt;
  u_int16_t sasoc_number_peer_destinations;
  u_int32_t sasoc_peer_rwnd;
  u_int32_t sasoc_local_rwnd;
  u_int32_t sasoc_cookie_life;
};
```

These fields have the following meaning:

- sasoc_assoc_id holds the identification for the association of interest. If this value is set to 0 when calling the setsockopt function, then sasoc_asocmaxrxt and sasoc_cookie_life represent values that are to be set as defaults on the socket. Calling getsockopt will return association-specific information if the association ID is supplied; otherwise, if this field is 0, the default endpoint settings will be returned.
- sasoc_asocmaxrxt holds the maximum number of retransmissions an association will make without acknowledgment before giving up, reporting the peer unusable and closing the association.
- sasoc_number_peer_destinations holds the number of peer destination addresses. It cannot be set, only retrieved.
- sasoc_peer_rwnd holds the peer's current calculated receive window. This value represents the total number of data bytes that can yet be sent. This field is dynamic; as the local

endpoint sends data, this value decreases. As the remote application reads data that has been received, this value increases. This value cannot be changed by this socket option call.

- sasoc_local_rwnd represents the local receive window the SCTP stack is currently reporting to the peer. This value is dynamic as well and is influenced by the SO_SNDBUF socket option. This value cannot be changed by this socket option call.
- sasoc_cookie_life represents the number of milliseconds for which a cookie, given to a remote peer, is valid. Each state cookie sent to a peer has a lifetime associated with it to prevent replay attacks. The default value of 60,000 milliseconds can be changed by setting this option with a sasoc_assoc_id value of 0.

***SCTP_AUTOCLOSE* Socket Option:** This option allows us to fetch or set the auto close time for an SCTP endpoint. The auto close time is the number of seconds an SCTP association will remain open when idle. Idle is defined by the SCTP stack as either endpoint sending or receiving user data. The default is for the auto close function to be disabled. The auto close option is intended to be used in the one-to-many-style SCTP interface. When this option is set, the integer passed to the option is the number of seconds before an idle connection should be closed; a value of 0 disables auto close. Only future associations created by this endpoint will be affected by this option; existing associations retain their current setting. Autoclose can be used by a server to force the closing of idle associations without the server needing to maintain additional state. A server using this feature needs to carefully assess the longest idle time expected on all its associations. Setting the auto close value smaller than needed results in the premature closing of associations.

***SCTP_DEFAULT_SEND_PARAM* Socket Option:** SCTP has many optional send parameters that are often passed as ancillary data or used with the sctp_sendmsg function call (which is often implemented as a library call that passes ancillary data for the user).

An application that wishes to send a large number of messages, all with the same parameters, can use this option to set up the default parameters and thus avoid using ancillary data or the sctp_sendmsg call.

This option takes as input the sctp_sndrcvinfo structure.

```
struct sctp_sndrcvinfo {
  u_int16_t sinfo_stream;
  u_int16_t sinfo_ssn;
  u_int16_t sinfo_flags;
  u_int32_t sinfo_ppid;
  u_int32_t sinfo_context;
  u_int32_t sinfo_timetolive;
  u_int32_t sinfo_tsn;
  u_int32_t sinfo_cumtsn;
  sctp_assoc_t sinfo_assoc_id;
};
```

These fields are defined as follows:

- sinfo_stream specifies the new default stream to which all messages will be sent.
- sinfo_ssn is ignored when setting the default options. When receiving a message with the recvmsg function or sctp_recvmsg function, this field will hold the value the peer placed in the stream sequence number (SSN) field in the SCTP DATA chunk.
- sinfo_flags dictates the default flags to apply to all future message sends.
- sinfo_pid provides the default value to use when setting the SCTP payload protocol identifier in all data transmissions.
- sinfo_context specifies the default value to place in the sinfo_context field, which is provided as a local tag when messages that could not be sent to a peer are retrieved.
- sinfo_timetolive dictates the default lifetime that will be applied to all message sends. The lifetime field is used by SCTP stacks to know when to discard an outgoing message due to excessive delay (prior to its first transmission). If the two endpoints support the partial reliability option, then the lifetime is also used to specify how long a message is valid after its first transmission.
- sinfo_tsn is ignored when setting the default options. When receiving a message with the recvmsg function or sctp_recvmsg function, this field will hold the value the peer placed in the transport sequence number (TSN) field in the SCTP DATA chunk.
- sinfo_cumtsn is ignored when setting the default options. When receiving a message with the recvmsg function or sctp_recvmsg function, this field will hold the current cumulative TSN the local SCTP stack has associated with its remote peer.

- sinfo_assoc_id specifies the association identification that the requester wishes the default parameters to be set against. For one-to-one sockets, this field is ignored.

***SCTP_DISABLE_FRAGMENTS Socket Option:***

SCTP normally fragments any user message that does not fit in a single SCTP packet into multiple DATA chunks. Setting this option disables this behavior on the sender. When disabled by this option, SCTP will return the error EMSGSIZE and not send the message. The default behavior is for this option to be disabled; SCTP will normally fragment user messages. This option may be used by applications that wish to control message sizes, ensuring that every user application message will fit in a single IP packet. An application that enables this option must be prepared to handle the error case (i.e., its message was too big) by either providing application-layer fragmentation of the message or a smaller message.

***SCTP_EVENTS Socket Option:*** This socket option allows a caller to fetch, enable, or disable various SCTP notifications. An SCTP notification is a message that the SCTP stack will send to the application. The message is read as normal data, with the msg_flags field of the recvmsg function being set to MSG_NOTIFICATION. An application that is not prepared to use either recvmsg or sctp_recvmsg should not enable events. Eight different types of events can be subscribed to by using this option and passing an sctp_event_subscribe structure. Any value of 0 represents a non-subscription and a value of 1 represents a subscription.

***SCTP_GET_PEER_ADDR_INFO Socket Option:*** This option retrieves information about a peer address, including the congestion window, smoothed RTT and MTU. This option may only be used to retrieve information about a specific peer address. The caller provides a sctp_paddrinfo structure with the spinfo_address field filled in with the peer address of interest, and should use sctp_opt_info instead of getsockopt for maximum portability. The sctp_paddrinfo structure has the following format:

```
struct sctp_paddrinfo
 {
        sctp_assoc_t spinfo_assoc_id;
        struct sockaddr_storage spinfo_address;
        int32_t spinfo_state;
        u_int32_t spinfo_cwnd;
        u_int32_t spinfo_srtt;
        u_int32_t spinfo_rto;
        u_int32_t spinfo_mtu;
};
```

***SCTP_GET_PEER_ADDR_INFO Socket Option:*** This option retrieves information about a peer address, including the congestion window, smoothed RTT and MTU. This option may only be used to retrieve information about a specific peer address. The caller provides a sctp_paddrinfo structure with the spinfo_address field filled in with the peer address of interest, and should use sctp_opt_info instead of getsockopt for maximum portability.

The sctp_paddrinfo structure has the following format:

```
struct sctp_paddrinfo
 {
        sctp_assoc_t spinfo_assoc_id;
        struct sockaddr_storage spinfo_address;
        int32_t spinfo_state;
        u_int32_t spinfo_cwnd;
        u_int32_t spinfo_srtt;
        u_int32_t spinfo_rto;
        u_int32_t spinfo_mtu;
};
```

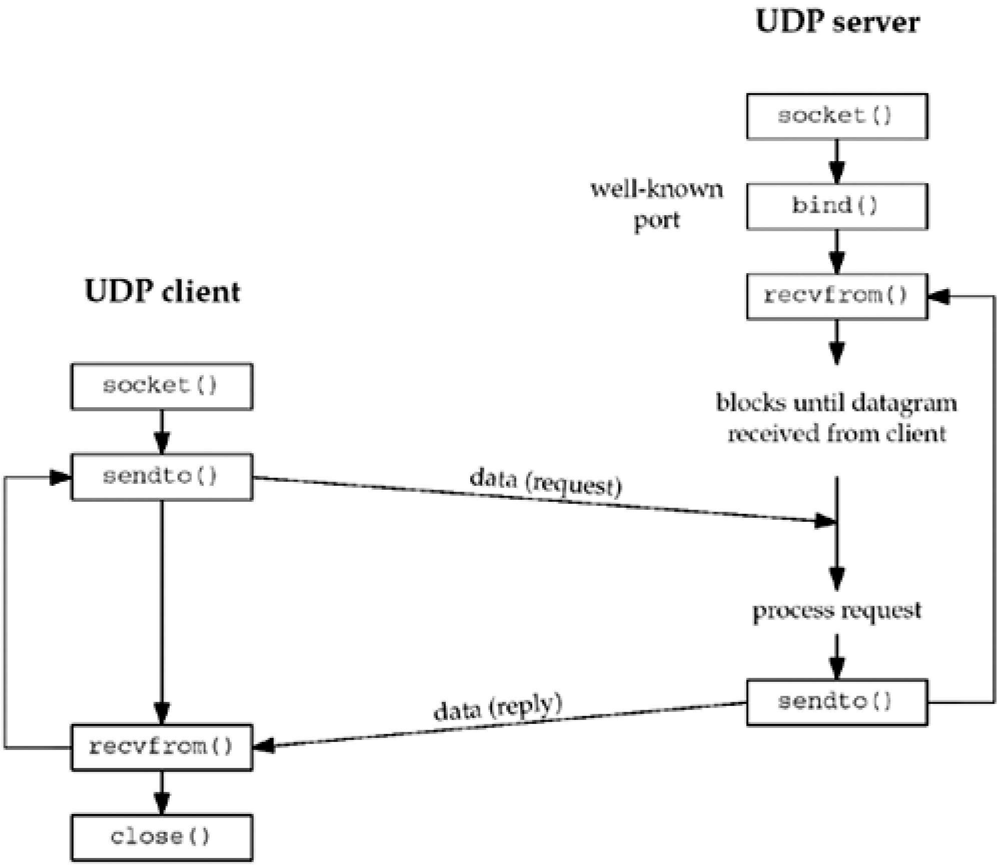
***SCTP_MAXBURST Socket Option:*** This socket option allows the application to fetch or set the maximum burst size used when sending packets. When an SCTP implementation sends data to a peer, no more than SCTP_MAXBURST packets are sent at once to avoid flooding the network with packets. An implementation may apply this limit by either: (i) reducing its congestion window to the current flight size plus the maximum burst size times the path MTU, or (ii) using this value as a separate micro-control, sending at most maximum burst packets at any single send opportunity.

***SCTP_NODELAY Socket Option:*** If set, this option disables SCTP's Nagle algorithm. This option is OFF by default (i.e., the Nagle algorithm is ON by default). SCTP's Nagle algorithm works identically to TCP's except that it is trying to coalesce multiple DATA chunks as opposed to simply coalescing bytes on a stream. For a further discussion of the Nagle algorithm, see TCP_MAXSEG.

## 8. UDP Socket

### Introduction

There are some fundamental differences between applications written using TCP versus those that use UDP. These are because of the differences in the two transport layers: UDP is a connectionless, unreliable, datagram protocol, quite unlike the connection-oriented, reliable byte stream provided by TCP. The Figure shows the function calls for a typical UDP client/server. The client does not establish a connection with the server. Instead, the client just sends a datagram to the server using the sendto function (described in the next section), which requires the address of the destination (the server) as a parameter. Similarly, the server does not accept a connection from a client. Instead, the server just calls the recvfrom function, which waits until data arrives from some client. recvfrom returns the protocol address of the client, along with the datagram, so the server can send a response to the correct client.



**Socket Function for UDP Client Server**

**recvfrom & sendto Functions:**

These two functions are similar to the standard read and write functions, but three additional arguments are required.

| |
|---|
| #include <sys/socket.h> |
| ssize_t recvfrom(int sockfd, void *buff, size_t nbytes, int flags, struct sockaddr *from, socklen_t *addrlen); |
| ssize_t sendto(int sockfd, const void *buff, size_t nbytes, int flags, const struct sockaddr *to, socklen_t |

| |
|---|
| addrlen); |
| Both return: number of bytes read or written if OK, –1 on error |

The first three arguments, sockfd, buff, and nbytes, are identical to the first three arguments for read and write: descriptor, pointer to buffer to read into or write from, and number of bytes to read or write. The to argument for sendto is a socket address structure containing the protocol address (e.g., IP address and port number) of where the data is to be sent. The size of this socket address structure is specified by addrlen. The recvfrom function fills in the socket address structure pointed to by from with the protocol address of who sent the datagram. The number of bytes stored in this socket address structure is also returned to the caller in the integer pointed to by addrlen. Note that the final argument to sendto is an integer value, while the final argument to recvfrom is a pointer to an integer value (a value-result argument).

The final two arguments to recvfrom are similar to the final two arguments to accept: The contents of the socket address structure upon return tell us who sent the datagram (in the case of UDP) or who initiated the connection (in the case of TCP). The final two arguments to sendto are similar to the final two arguments to connect: We fill in the socket address structure with the protocol address of where to send the datagram (in the case of UDP) or with whom to establish a connection (in the case of TCP).

Both functions return the length of the data that was read or written as the value of the function. In the typical use of recvfrom, with a datagram protocol, the return value is the amount of user data in the datagram received.

**8. UDP Echo Client Server**

**UDP Client:**

```
#include <stdio.h>
#include <errno.h>
#include <sys/socket.h>
#include <resolv.h>
#include <netinet/in.h>
#include <sys/types.h>
#include<string.h>
```

```
main(int argc, char * argv[])
{
        char buffer[50]="";
        struct sockaddr_in servaddr;
        sockfd = socket(AF_INET, SOCK_DGRAM, 0);
        bzero(&servaddr, sizeof(servaddr));
        servaddr.sin_family = AF_INET;
        servaddr.sin_addr.s_addr = INADDR_ANY;
        servaddr.sin_port = htons(7802);
        connect(sockfd,(struct sockaddr*)&servaddr,sizeof(servaddr));
        scanf("%s",buffer);
        send(sockfd,buffer,sizeof(buffer),0);
        recv(sockfd,buffer,sizeof(buffer),0);
}
```

**UDP Server:**

```
#include <sys/socket.h>
#include <sys/types.h>
#include <string.h>
#include<netinet/in.h>
#include <stdio.h>
#include<stdlib.h>
main()
{
        char buffer[100]="";
        int sockfd, len;
        struct sockaddr_in saddr;
        sockfd = socket(AF_INET, SOCK_DGRAM, 0);
        bzero(&saddr, sizeof(saddr));
        saddr.sin_family = AF_INET;
        saddr.sin_addr.s_addr = INADDR_ANY;
        saddr.sin_port = htons(7802);
        bind(sockfd, (struct sockaddr *)&saddr, sizeof(saddr));
        recv(sockfd,buffer,sizeof(buffer),0);
        send(sockfd,buffer,sizeof(buffer),0);
}
```

**Lost Datagrams**

Our UDP client/server example is not reliable. If a client datagram is lost (say it is discarded by some router between the client and server), the client will block forever in its call to recv in the function dg_cli, waiting for a server reply that will never arrive. Similarly, if the client datagram arrives at the server but the server's reply is lost, the client will again block forever in its call to recv. A typical way to prevent this is to place a timeout on the client's call to recv.

**9. Address Conversion**

All the examples so far in this text have used numeric addresses for the hosts (e.g., 206.6.226.33) and numeric port numbers to identify the servers (e.g., port 13 for the standard daytime server and port 9877 for our echo server). We should, however, use names instead of numbers for numerous reasons: Names are easier to remember; the numeric address can change but the name can remain the same; and with the move to IPv6, numeric addresses become much longer, making it much more error-prone to enter an address by hand.

**gethostbyname()**

Host computers are normally known by human-readable names. All the examples that we have shown so far in this book have intentionally used IP addresses instead of names, so we know exactly what goes into the socket address structures for functions such as connect and sendto, and what is returned by functions such as accept and recvfrom. But, most applications should deal with names, not addresses.

The non-null pointer returned by this function points to the following hostent structure:

```
#include <netdb.h>

struct hostent *gethostbyname (const char *hostname);

                    Returns: non-null pointer if OK,NULL on error with h_errno set
```

```
struct hostent
 {          char  *h_name;       /* official (canonical) name of host */
            char **h_aliases;    /* pointer to array of pointers to alias names */
            int    h_addrtype;   /* host address type: AF_INET */
            int    h_length;     /* length of address: 4 */
            char **h_addr_list;  /* ptr to array of ptrs with IPv4 addrs */  };
```

The most basic function that looks up a hostname is gethostbyname. If successful, it returns a pointer to a hostent structure that contains all the IPv4 addresses for the host. However, it is limited in that it can only return IPv4 addresses. In terms of the DNS, gethostbyname performs a query for an A record. This function can return only IPv4 addresses

**gethostbyaddr Function:**The function gethostbyaddr takes a binary IPv4 address and tries to find the hostname corresponding to that address. This is the reverse of gethostbyname.

```
#include <netdb.h>

struct hostent *gethostbyaddr (const char *addr, socklen_t len, int family);

Returns: non-null pointer if OK, NULL on error with h_errno set
```

This function returns a pointer to the same hostent structure that we described with gethostbyname. The field of interest in this structure is normally h_name, the canonical hostname. The addr argument is not a char*, but is really a pointer to an in_addr structure containing the IPv4 address. len is the size of this structure: 4 for an IPv4 address. The family argument is AF_INET. In terms of the DNS, gethostbyaddr queries a name server for a PTR record in the in-addr.arpa domain.

**getservbyname and getservbyport Functions**

Services, like hosts, are often known by names, too. If we refer to a service by its name in our code, instead of by its port number, and if the mapping from the name to port number is contained in a file (normally /etc/services), then if the port number changes, all we need to modify is one line in the /etc/services file instead of having to recompile the applications. The next function, getservbyname, looks up a service given its name.

```
#include <netdb.h>

struct servent *getservbyname (const char *servname, const char *protoname);

                                        Returns: non-null pointer if OK, NULL on error
```

This function returns a pointer to the following structure:

```
struct servent
{           char   *s_name;      /* official service name */
            char  **s_aliases;   /* alias list */
             int     s-port;      /* port number, network-byte order */
            char   *s_proto;     /* protocol to use */ };
```

The service name servname must be specified. If a protocol is also specified (protoname is a non-null pointer), then the entry must also have a matching protocol. Some Internet services are provided using either TCP or UDP.

The next function, getservbyport, looks up a service given its port number and an optional protocol.

| |
|---|
| #include <netdb.h> |
| struct servent *getservbyport (int port, const char *protoname); |
| Returns: non-null pointer if OK, NULL on error |

The port value must be network byte ordered. Typical calls to this function could be as follows:

```
struct servent *sptr;
sptr = getservbyport (htons (53), "udp"); /* DNS using UDP */
sptr = getservbyport (htons (21), "tcp"); /* FTP using TCP */
sptr = getservbyport (htons (21), NULL);  /* FTP using TCP */
sptr = getservbyport (htons (21), "udp"); /* this call will fail */
```

The last call fails because there is no service that uses port 21 with UDP.

**getaddrinfo Function**

The gethostbyname and gethostbyaddr functions only support IPv4. The API for resolving IPv6 addresses went through several iterations. These sockaddr structures can then be used by the socket functions directly. In this way, the getaddrinfo function hides all the protocol dependencies in the library function, which is where they belong. The application deals only with the socket address structures that are filled in by getaddrinfo. This function is defined in the POSIX specification.

| |
|---|
| #include <netdb.h> |
| int getaddrinfo (const char *hostname, const char *service, const struct addrinfo *hints, struct addrinfo **result) ; |
| Returns: 0 if OK, nonzero on error |

This function returns through the result pointer a pointer to a linked list of addrinfo structures, which is defined by including <netdb.h>.

```
struct addrinfo {
   int       ai_flags;       /* AI_PASSIVE, AI_CANONNAME */
   int       ai_family;       /* AF_xxx */
   int         ai_socktype;       /* SOCK_xxx */
   int         ai_protocol;       /* 0 or IPPROTO_xxx for IPv4 and IPv6 */
   socklen_t   ai_addrlen;        /* length of ai_addr */
   char        *ai_canonname;     /* ptr to canonical name for host */
   struct sockaddr   *ai_addr;    /* ptr to socket address structure */
   struct addrinfo   *ai_next;    /* ptr to next structure in linked list */
};
```

The hostname is either a hostname or an address string (dotted-decimal for IPv4 or a hex string for IPv6). The service is either a service name or a decimal port number string hints is either a null pointer or a pointer to an addrinfo structure that the caller fills in with hints about the types of information the caller wants returned. For example, if the specified service is provided for both TCP and UDP (e.g., the domain service, which refers to a DNS server), the caller can set the ai_socktype member of the hints structure to SOCK_DGRAM. The only information returned will be for datagram sockets. The members of the hints structure that can be set by the caller are:

- ai_flags (zero or more AI_XXX values OR'ed together)
- ai_family (an AF_xxx value)
- ai_socktype (a SOCK_xxx value)
- ai_protocol

### Possible Questions

## 2 Mark Questions

1. What is a socket?
2. What is ARP?
3. What is RARP?
4. What is a datagram?
5. What is a segment?

### 6 Mark Questions

1. Discuss about Socket in general.
2. How are Posix signal handled in C? Discuss about it in detail.
3. Illustrate wait and waitpid function.
4. Discuss about IPv4 Socket options.
5. Discuss about IPv6 Socket Options.
6. Write a detail note on TCP Socket options.
7. Discuss about SCTP Socket Option in detail.
8. Write a program that uses UDP recfrom and sendto to transfer data between client & server.
9. Discuss in short about any 4 Address conversion functions
10. Discuss about Socket options in general.

**KARPAGAM ACADEMY OF HIGHER EDUCATION**
**(Deemed University Established Under Section 3 of UGC Act, 1956)**
**Coimbatore – 641021, INDIA**
**Department of Computer Science, Applications & Information Technology**

**Unit - III**

| S.No | Questions | Opt1 | Opt2 | Opt3 | Opt4 | Answer |
|---|---|---|---|---|---|---|
| 1 | When an application sits in a loop calling recvfrom on a nonblocking descriptor like this, it is called | discarding | polling | synchronous | timeour | polling |
| 2 | The signal-driven I/O model uses signals, telling the kernel to notify us with the ____signal when the descriptor is ready | SIGIO | SICCIO | SIOID | SIDCOO | SIGIO |
| 3 | read the datagram from the signal handler by calling ____ | getfrom | receivedfrom | recvfrom | reservedfrom | recvfrom |
| 4 | A _____ operation causes the requesting process to be blocked until that I/O operation completes | asynchronous I/O | signal I/O | designalling | synchronous I/O | synchronous I/O |
| 5 | The _____ function allows the process to instruct the kernel | select | deselect | poll | time | select |
| 6 | The ________argument tells the kernel how long to wait for one of the specified descriptors to become ready | timein | comein | timeout | comout | timeout |
| 7 | A ______structure specifies the number of seconds and microseconds | timeval | timefunc | timeinterval | timeread | timeval |
| 8 | _____will Return only when one of the specified descriptors is ready for I/O | Wait | Wait signal | Wait forever | Waiting | Wait forever |
| 9 | The normal way to terminate a network connection is to call the ____ function | close | shut | shutdown | restart | close |
| 10 | The close function decrements the descriptor's reference count and closes the socket only if the count reaches____ | 1 | 2 | 3 | 0 | 0 |
| 11 | _____terminates both directions of data transfer, reading and writing | shut | close | shutdown | restart | close |
| 12 | pselect uses the ____ structure instead of the timeval structure | time | timing | timespec | timefunc | timespec |
| 13 | The ____member of the newer structure specifies nanoseconds | tv_sec | tv_nsec | n_sec | sec_nano | tv_nsec |
| 14 | ______ member of the older structure specifies microseconds. | tv_micro | micro_sec | tv_usec | sec_tv | tv_usec |
| 15 | poll function provides additional information when dealing with ____ devices. | STREAMS | INPUT | OUTPUT | FILES | STREAMS |
| 16 | ____ specifies the presence of an error for a TCP connection can be considered either normal data or an error | POLLERROR | POLLING | ERRORPOLL | POLLERR | POLLERR |
| 17 | The constant INFTIM (wait forever) is defined to be a _____ value. | negative | positive | real | natural | negative |
| 18 | ______must refer to an open socket descriptor | sockdes | sockfd | sockfile | sockopen | sockfd |
| 19 | ____is a pointer to a variable from which the new value of the option is fetched by setsockopt | optimal | optional | optval | optionalvalue | optval |
| 20 | The size of this variable is specified by the final argument _____ | length | optlen | varlen | finallen | optlen |
| 21 | Generic socket options are _____ | protocol-independent | platform dependent | protocol-dependent | cross platform | protocol-independent |
| 22 | ______Socket option enables or disables the ability of the process to send broadcast messages | SO_SOCKETS | SO_BROADCAST | SO-UNICAST | SO_MULTICAST | SO_BROADCAST |
| 23 | SO_DEBUG Socket Option is supported only by _____ | TCP | UDP | SCTP | IGMP | TCP |
| 24 | _____ Socket Option specifies that outgoing packets are to bypass the normal routing mechanisms of the underlying protocol | SO_ROUTE | SO_DONT | SO_PASS | SO_DONTROUTE | SO_DONTROUTE |
| 25 | _____ Socket Option is one that can be fetched but cannot be set | SO_ERR | SO_ERROR | SO_FETCH | SO_BURST | SO_ERROR |
| 26 | _____ Socket Option cannot be set, only retrieved | IPV6_RECVTCLASS | IPV6_PATHMTU | IPV6_UNICAST_HOPS | IPV6_MSG | IPV6_PATHMTU |
| 27 | ____Socket Option specifies that the received traffic class is to be returned as ancillary data by recvmsg | IPV6_RECVTCLASS | IPV6_PATHMTU | IPV6_UNICAST_HOPS | IPV6_MSG | IPV6_RECVTCLASS |
| 28 | ______Socket Option is similar to the IPv4 IP_TTL socket option | IPV6_RECVTCLASS | IPV6_PATHMTU | IPV6_UNICAST_HOPS | IPV6_MSG | IPV6_UNICAST_HOPS |
| 29 | _____Socket Option allows us to fetch or set the MSS for a TCP connection | TCP_MAXSEG | TCP_MAXIMUM | TCP_SEGEMENT | TCP_MIN | TCP_MAXSEG |
| 30 | The ______ socket option can be used to retrieve information about an existing association | SCTP_ASSOCIATION | SCTP_INDEXING | SCTP_MAXBURST | SCTP_ASSOCINF | SCTP_ASSOCINF |
| 31 | The____ time is the number of seconds an SCTP association will remain open when idle | autoopen | autoclose | autoin | autoout | autoclose |
| 32 | _______ Socket Option allows a caller to fetch, enable, or disable various SCTP notifications | SCTP_ASSOCIATION | SCTP_EVENTS | SCTP_MAXBURST | SCTP_ASSOCINF | SCTP_EVENTS |
| 33 | _____ Socket Option allows the application to fetch or set the maximum burst size used when sending packets | SCTP_ASSOCIATION | SCTP_TIME | SCTP_MAXBURST | SCTP_ASSOCINF | SCTP_MAXBURST |
| 34 | _____Socket Option allows the application to fetch or set the maximum fragment size used during SCTP fragmentation | SCTP_ASSOCIATION | SCTP_MAXSEG | SCTP_MAXBURST | SCTP_ASSOCINF | SCTP_MAXSEG |
| 35 | SCTP_NODELAY Socket Option if set, this option disables SCTP's ____ algorithm | Nagle | Fourier | Aprior | Filter | Nagle |
| 36 | _____Socket Option will retrieve the current state of an SCTP association | SCTP_ASSOCIATION | SCTP_STATUS | SCTP_MAXBURST | SCTP_ASSOCINF | SCTP_STATUS |
| 37 | The to argument for ____ is a socket address structure containing the protocol address of where the data is to be sent. | sendto | sendprotocol | senddata | sendaddr | sendto |
| 38 | The size of the socket address structure is specified by _____ | addresslength | addrlen | sizelen | socklen | addrlen |
| 39 | We create a UDP socket by specifying the second argument to socket as ____ | SOCK_UDP | SOCK_ARGU | SOCK_DGRAM | SOCK_SECOND | SOCK_DGRAM |

## UNIT-IV

## SYLLABUS

**Network Applications:** Remote logging; Email; WWW and HTTP

### 1. Introduction

The main task of the Internet and its TCP/IP protocol suite is to provide services for users. Although there are some specific client/server programs, it would be impossible to write a specific client-server program for each demand. The better solution is a general-purpose client-server program that lets a user access any application program on a remote computer; in other words, allow the user to log on to a remote computer.

### 2. TELNET

TELNET is an abbreviation for TErminaL NETwork. It is the standard TCP/IP protocol for virtual terminal service as proposed by ISO. TELNET enables the establishment of a connection to a remote system in such a way that the local terminal appears to be a terminal at the remote system.

**Time-Sharing Environment**

TELNET was designed at a time when most operating systems, such as UNIX, were operating in a time-sharing environment. In such an environment, a large computer supports multiple users. The interaction between a user and the computer occurs through a terminal, which is usually a combination of keyboard, monitor, and mouse. Even a microcomputer can simulate a terminal with a terminal emulator. In a time-sharing environment, all of the processing must be done by the central computer. When a user types a character on the keyboard, the character is usually sent to the computer and echoed to the monitor.

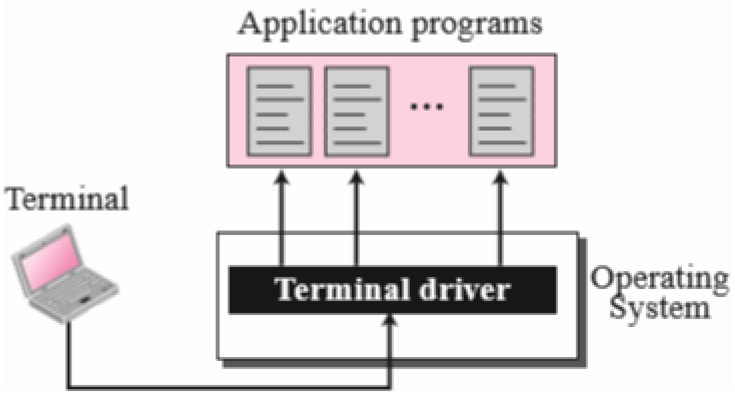
**Login**

In a time-sharing environment, users are part of the system with some right to access resources. Each authorized user has an identification and probably a password. The user identification defines the user as part of the system. To access the system, the user logs into the system with a user id or login name. The system also includes password checking to prevent an unauthorized user from accessing the resources.

**Local Login**

When a user logs into a local time-sharing system, it is called local login. 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.

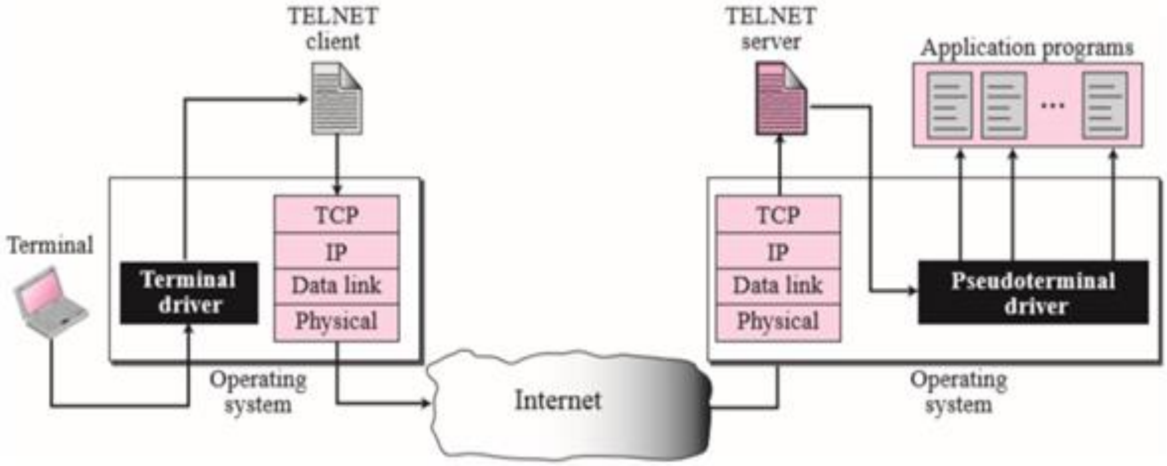
The mechanism, however, is not as simple as it seems because the operating system may assign special meanings to special characters. For example, in UNIX some combinations of characters have special meanings, such as the combination of the control character with the character z, which means suspend; the combination of the control character with the character c, which means abort; and so on.



**Local Login**

### Remote Login

When a user wants to access an application program or utility located on a remote machine, he or she performs remote login. 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 to a universal character set called Network Virtual Terminal (NVT) characters and delivers them to the local TCP/IP stack

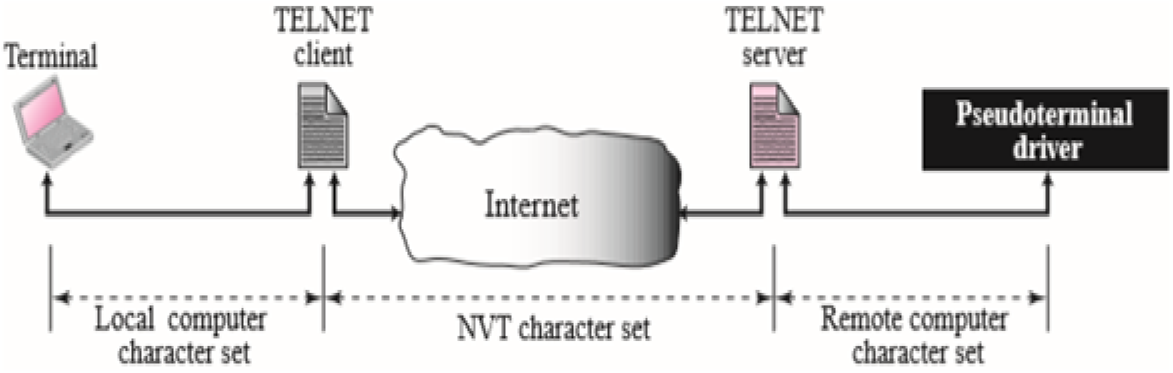


**Remote Login**

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 pseudo-terminal driver, which pretends that the characters are coming from a terminal. The operating system then passes the characters to the appropriate application program.

**Network Virtual Terminal (NVT)**

The mechanism to access a remote computer is complex. This is because every computer and its operating system accepts a special combination of characters as tokens. For example, the end-of-file token in a computer running the DOS operating system is Ctrl+z, while the UNIX operating system recognizes Ctrl+d. We are dealing with heterogeneous systems. If we want to access any remote computer in the world, we must first know what type of computer we will be connected to, and we must also install the specific terminal emulator used by that computer. TELNET solves this problem by defining a universal interface called the Network Virtual Terminal (NVT) character set. Via this interface, the client TELNET translates characters (data or commands) that come from the local terminal into NVT form and delivers them to the network. The server TELNET, on the other hand, translates data and commands from NVT form into the form acceptable by the remote computer.



Concept of NVT

**NVT Character Set**

NVT uses two sets of characters, one for data and one for control. Both are 8-bit bytes.

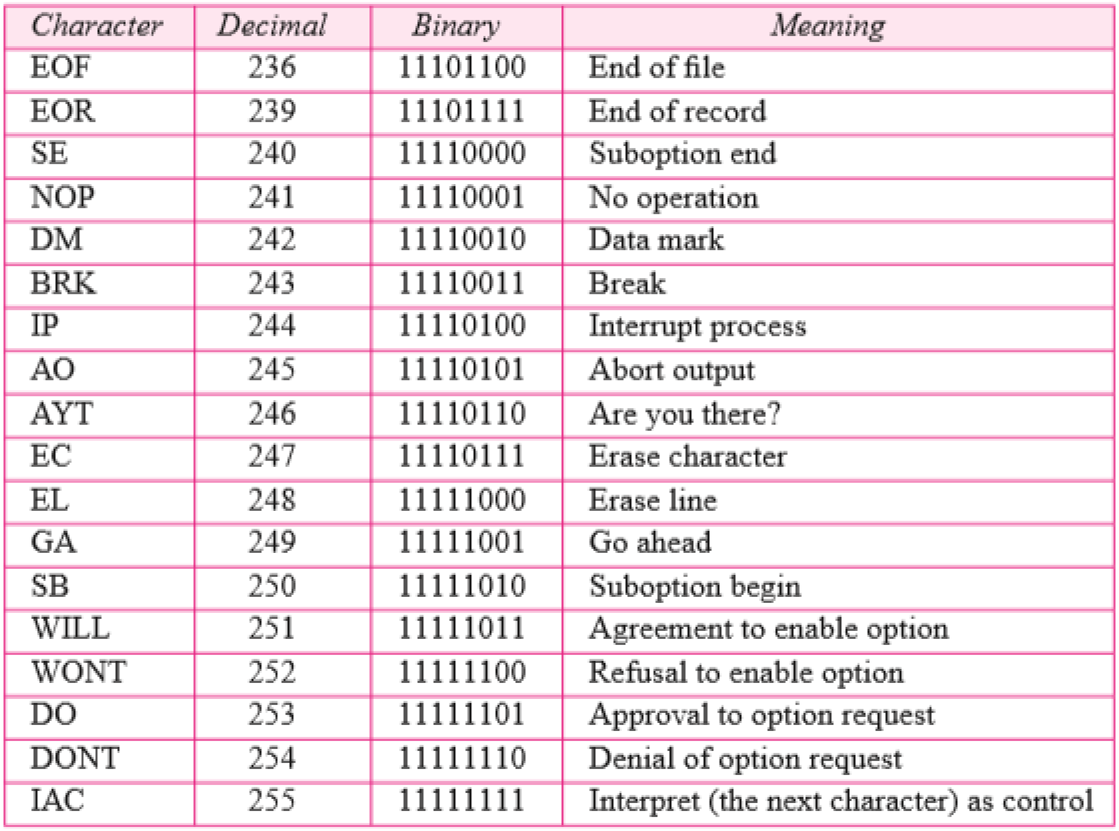
a. Data Character b. Control Character

**Data Characters**

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. Although it is possible to send an 8-bit ASCII (with the highest order bit set to be 0 or 1), this must first be agreed upon between the client and the server using option negotiation.

**Control Characters**

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. The below lists some of the control characters and their meanings.



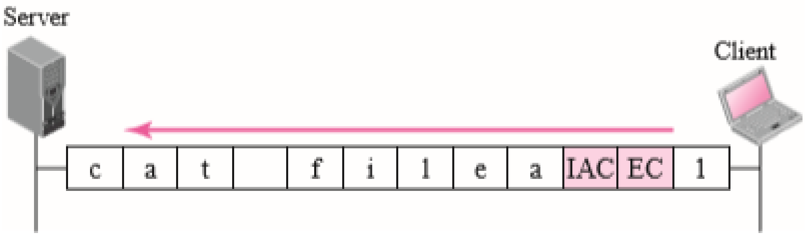
| *Character* | *Decimal* | *Binary* | *Meaning* |
|---|---|---|---|
| EOF | 236 | 11101100 | End of file |
| EOR | 239 | 11101111 | End of record |
| SE | 240 | 11110000 | Suboption end |
| NOP | 241 | 11110001 | No operation |
| DM | 242 | 11110010 | Data mark |
| BRK | 243 | 11110011 | Break |
| IP | 244 | 11110100 | Interrupt process |
| AO | 245 | 11110101 | Abort output |
| AYT | 246 | 11110110 | Are you there? |
| EC | 247 | 11110111 | Erase character |
| EL | 248 | 11111000 | Erase line |
| GA | 249 | 11111001 | Go ahead |
| SB | 250 | 11111010 | Suboption begin |
| WILL | 251 | 11111011 | Agreement to enable option |
| WONT | 252 | 11111100 | Refusal to enable option |
| DO | 253 | 11111101 | Approval to option request |
| DONT | 254 | 11111110 | Denial of option request |
| IAC | 255 | 11111111 | Interpret (the next character) as control |

**Embedding**

TELNET uses only one TCP connection. The server uses the well-known port 23 and the client uses an ephemeral port. The same connection is used for sending both data and control characters.

TELNET accomplishes this by embedding the control characters in the data stream. However, to distinguish data from control characters, each sequence of control characters is preceded by a special control character called interpret as control (IAC).

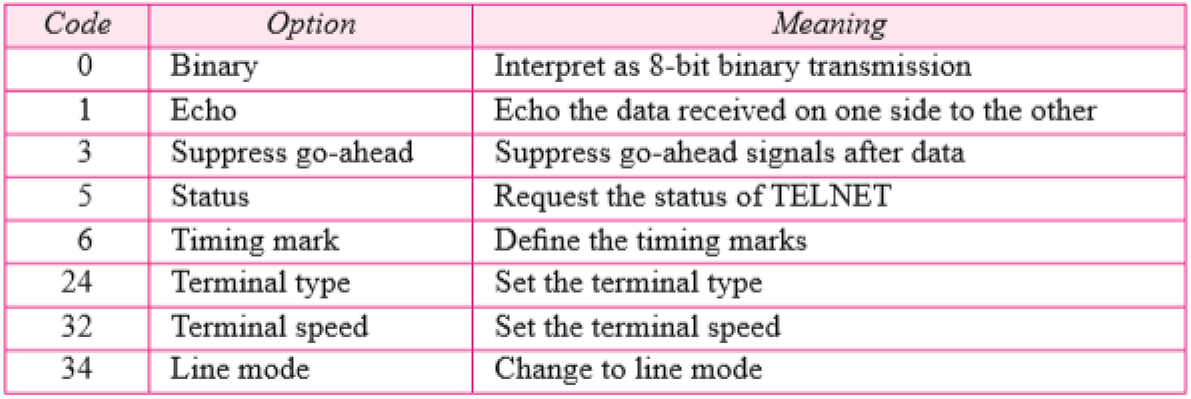
For example, imagine a user wants a server to display a file (file1) on a remote server he uses “cat file1” command. However, the name of the file has been mistyped (filea instead of file1). The user uses the backspace key to correct this situation.” cat filea<backspace>1”. However, in the default implementation of TELNET, the user cannot edit locally; the editing is done at the remote server. The backspace character is translated into two remote characters (IAC EC), which is embedded in the data and sent to the remote server. What is sent to the server is shown in Figure.



**Embedding**

### Options

TELNET lets the client and server negotiate options before or during the use of the ser- vice. Options are extra features available to a user with a more sophisticated terminal. Users with simpler terminals can use default features. Some control characters dis- cussed previously are used to define options. The table shows some common options.



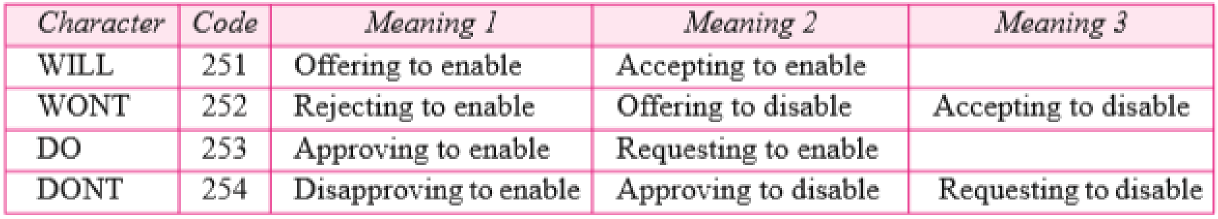
| Code | Option | Meaning |
|---|---|---|
| 0 | Binary | Interpret as 8-bit binary transmission |
| 1 | Echo | Echo the data received on one side to the other |
| 3 | Suppress go-ahead | Suppress go-ahead signals after data |
| 5 | Status | Request the status of TELNET |
| 6 | Timing mark | Define the timing marks |
| 24 | Terminal type | Set the terminal type |
| 32 | Terminal speed | Set the terminal speed |
| 34 | Line mode | Change to line mode |

- **Binary:** This option allows the receiver to interpret every 8-bit character received, except IAC, as binary data. When IAC is received, the next character or characters are interpreted as commands. However, if two consecutive IAC characters are received, the first is discarded and the second is interpreted as data. **Echo:** This option allows the server to echo data received from the client. This means that every character sent by the client to the sender will be echoed back to the screen of the client terminal. In this case, the user terminal usually does not echo characters when they are typed but waits until it receives them from the server.

- **Suppress go-ahead:** This option suppresses the go-ahead (GA) character (see section on Modes of Operation).
- **Status:** This option allows the user or the process running on the client machine to get the status of the options being enabled at the server site.
- **Timing mark:**
- This option allows one party to issue a timing mark that indicates all previously received data has been processed.
- **Terminal type:**
- This option allows the client to send its terminal type.
- **Terminal speed:** This option allows the client to send its terminal speed.
- **Line mode:** This option allows the client to switch to the line mode

**Option Negotiation**

To use any of the options mentioned in the previous section first requires option negotiation between the client and the server. Four control characters are used for this purpose; these are shown in Table



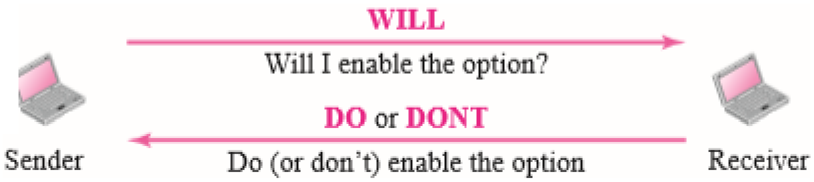
| *Character* | *Code* | *Meaning 1* | *Meaning 2* | *Meaning 3* |
|---|---|---|---|---|
| WILL | 251 | Offering to enable | Accepting to enable | |
| WONT | 252 | Rejecting to enable | Offering to disable | Accepting to disable |
| DO | 253 | Approving to enable | Requesting to enable | |
| DONT | 254 | Disapproving to enable | Approving to disable | Requesting to disable |

**Enabling an Option**

Some options can only be enabled by the server, some only by the client, and some by both. An option is enabled either through an offer or a request.

**Offer to Enable**

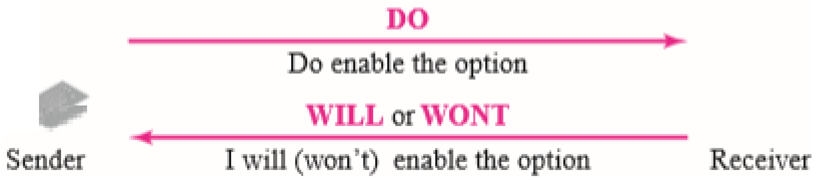
A party can offer to enable an option if it has the right to do so. The offering can be approved or disapproved by the other party. The offering party sends the WILL command, which means "Will I enable the option?" The other party sends either the DO command, which means "Please do," or the DONT command, which means "Please don't."



**Offer to Enable an option**

### Request to Enable

A party can request from the other party the enabling of an option. The request can be accepted or refused by the other party. The requesting party sends the DO command, which means “Please do enable the option.” The other party sends either the WILL command, which means “I will,” or the WONT command, which means “I won’t.”



**Request to Enable**

### Disabling an Option

An option that has been enabled can be disabled by one of the parties. An option is disabled either through an offer or a request.

### Offer to Disable

A party can offer to disable an option. The other party must approve the offering; it cannot be disapproved. The offering party sends the WONT command, which means “I won’t use this option anymore.” The answer must be the DONT com- mand, which means “Don’t use it anymore.”

### Request to Disable:

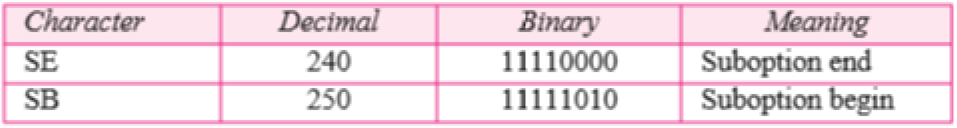
A party can request from another party the disabling of an option. The other party must accept the request; it cannot be rejected. The requesting party sends the DONT command, which means “Please don’t use this option anymore.” The answer must be the WONT command, which means “I won’t use it anymore.

### Symmetry

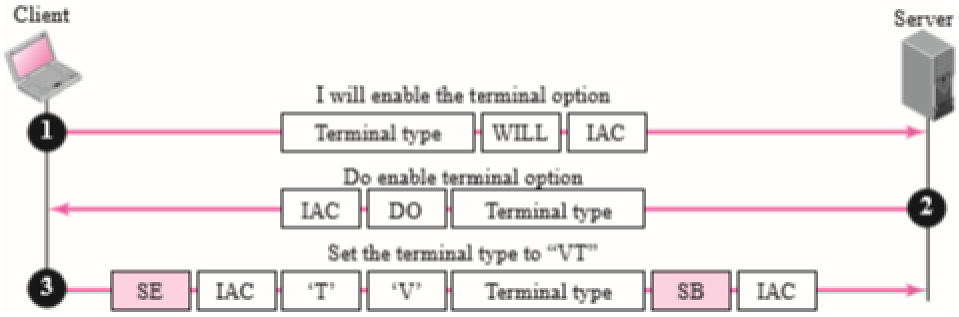
One interesting feature of TELNET is its symmetric option negotiation in which the client and server are given equal opportunity. This means that, at the beginning of con- nection, it is assumed that both sides are using a default TELNET implementation with no options enabled. If one party wants an option enabled, it can offer or request. The other party has the right to approve the offer or reject the request if the party is not capable of using the option or does not want to use the option. This allows for the expansion of TELNET.

**Sub-option Negotiation**

Some options require additional information. For example, to define the type or speed of a terminal, the negotiation includes a string or a number to define the type or speed. In either case, the two sub-option characters indicated in the table are needed for sub-option negotiation.



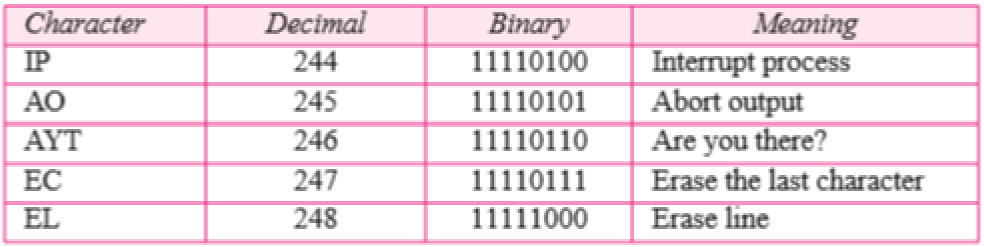
| *Character* | *Decimal* | *Binary* | *Meaning* |
|---|---|---|---|
| SE | 240 | 11110000 | Suboption end |
| SB | 250 | 11111010 | Suboption begin |


Client
Server
I will enable the terminal option
1
Terminal type
WILL
IAC
Do enable terminal option
2
IAC
DO
Terminal type
Set the terminal type to "VT"
3
SE
IAC
'T'
'V'
Terminal type
SB
IAC

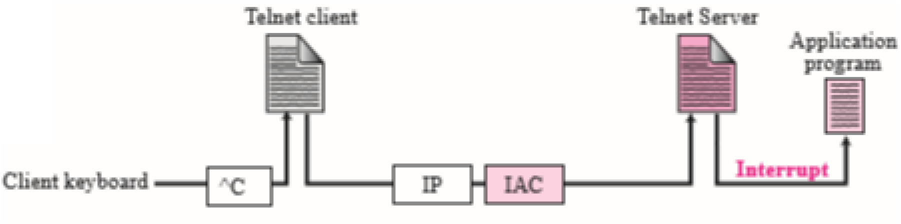
**Sub Option Negotiation**

**Controlling the Server**

Some control characters can be used to control the remote server. When an application program is running on the local computer, special characters are used to interrupt (abort) the program (for example, Ctrl+c), or erase the last character typed (for example, delete key or backspace key), and so on. However, when a program is running on a remote computer, these control characters are sent to the remote machine. The user still types the same sequences, but they are changed to special characters and sent to the server. Table below shows some of the characters that can be sent to the server to control the application program that is running there.



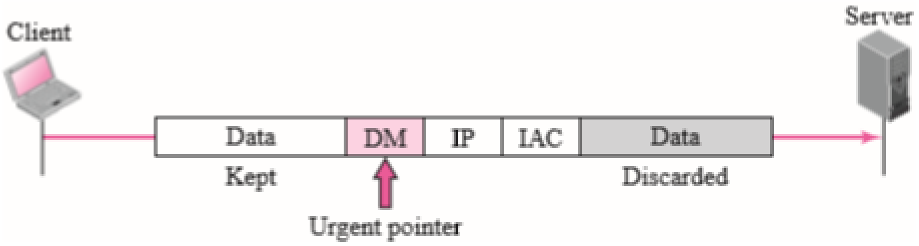
| *Character* | *Decimal* | *Binary* | *Meaning* |
|---|---|---|---|
| IP | 244 | 11110100 | Interrupt process |
| AO | 245 | 11110101 | Abort output |
| AYT | 246 | 11110110 | Are you there? |
| EC | 247 | 11110111 | Erase the last character |
| EL | 248 | 11111000 | Erase line |



**Example of interrupting the program**

**Out-of-Band Signaling**

To make control characters effective in special situations, TELNET uses out-of-band signaling. In out-of-band signaling, the control characters are preceded by IAC and are sent to the remote process. Imagine a situation in which an application program running at the server site has gone into an infinite loop and does not accept any more input data. The user wants to interrupt the application program, but the program does not read data from the buffer. The TCP at the server site has found that the buffer is full and has sent a segment specifying that the client window size should be zero. In other words, the TCP at the server site is announcing that no more regular traffic is accepted. To remedy such a situation, an urgent TCP segment should be sent from the client to the server. The urgent segment overrides the regular flow-control mechanism. Although TCP is not accepting normal segments, it must accept an urgent segment.



When a TELNET process (client or server) wants to send an out-of-band sequence of characters to the other process (client or server), it embeds the sequence in the data stream and inserts a special character called a DM (data mark). However, to inform the other party, it creates a TCP segment with the urgent bit set and the urgent pointer pointing to the DM character. When the receiving process receives the data, it reads the data and discards any data preceding the control characters (IAC and IP, for example). When it reaches the DM character, the remaining data are handled normally. In other words, the DM character is used as a synchronization character that switches the receiving process from the urgent mode to the normal mode and resynchronizes the two ends. In this way, the control character (IP) is delivered out of band to the operating system, which uses the appropriate function to interrupt the running application program.

### 3. Email

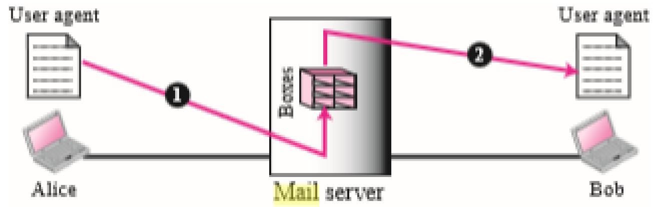
One of the most popular Internet services is electronic mail (e-mail). At the beginning of the Internet era, the messages sent by electronic mail were short and consisted of text only; they let people exchange quick memos. Today, electronic mail is much more complex. It allows a message to include text, audio, and video. It also allows one message to be sent to one or more recipients.

**Architecture**

To explain the architecture of e-mail, we give four scenarios. We begin with the simplest situation and add complexity as we proceed. The fourth scenario is the most common in the exchange of e-mail.

**First Scenario**

In the first scenario, the sender and the receiver of the e-mail are users (or application programs) on the same mail server; they are directly connected to a shared mail server. The administrator has created one mailbox for each user where the received messages are stored. A mailbox is part of a local hard drive, a special file with permission restrictions. Only the owner of the mailbox has access to it. When Alice needs to send a message to Bob, she runs a user agent (UA) program to prepare the message and store it in Bob’s mailbox. The message has the sender and recipient mailbox addresses (names of files). Bob can retrieve and read the contents of his mailbox at his convenience using a user agent. The following figure shows the scenario.

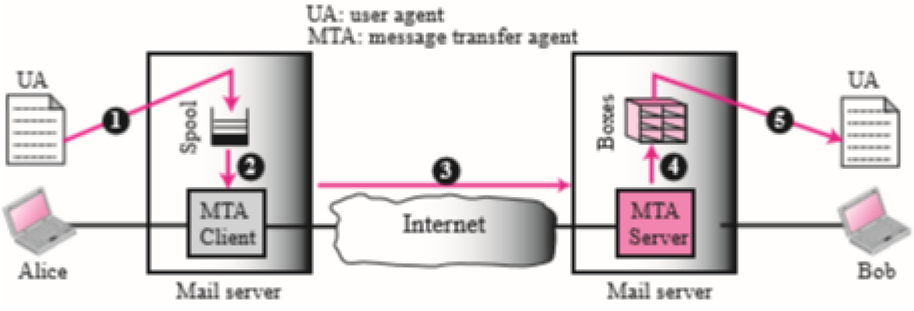


**First Scenario**

**Second Scenario**

In the second scenario, the sender and the receiver of the e-mail are users (or application programs) on two different mail servers. The message needs to be sent over the Internet. Here we need user agents (UAs) and message transfer agents (MTAs) as shown in Figure.

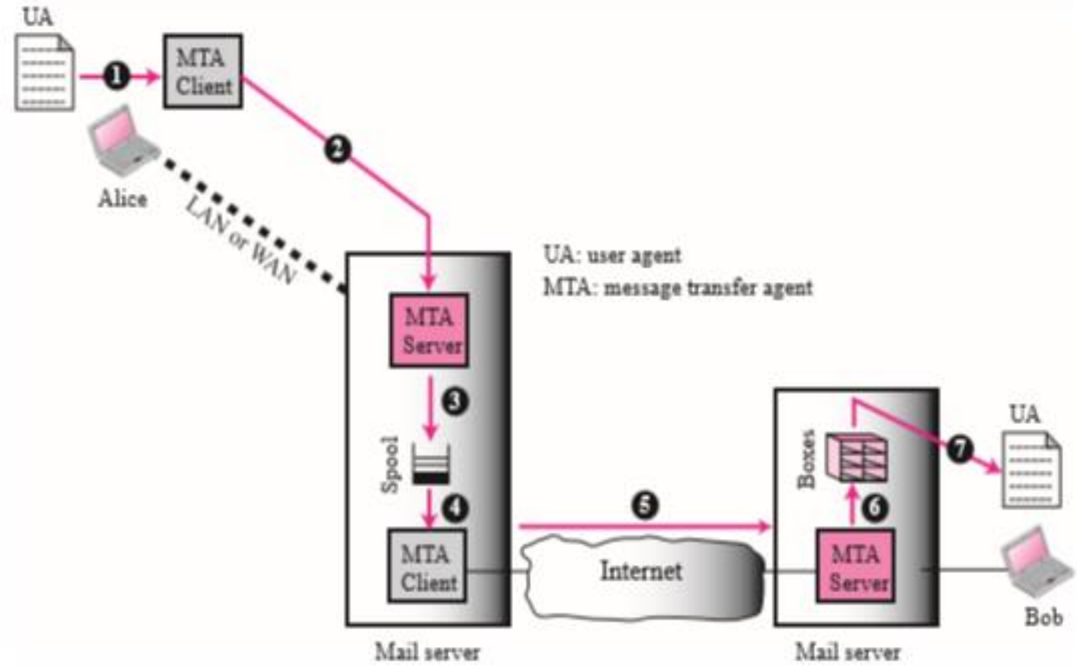
Alice needs to use a user agent program to send her message to the mail server at her own site. The mail server at her site uses a queue (spool) to store messages waiting to be sent. Bob also needs a user agent program to retrieve messages stored in the mail- box of the system at his site. The message, however, needs to be sent through the Inter- net from Alice’s site to Bob’s site. 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 of 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.



**Second Scenario**

### Third Scenario

The Following Figure shows the third scenario. Bob, as in the second scenario, is directly connected to his mail server. Alice, however, is separated from her mail server. Alice is either connected to the mail server via a point-to-point WAN—such as a dial-up modem, a DSL, or a cable modem—or she is connected to a LAN in an organization that uses one mail server for handling e-mails; all users need to send their messages to this mail server.

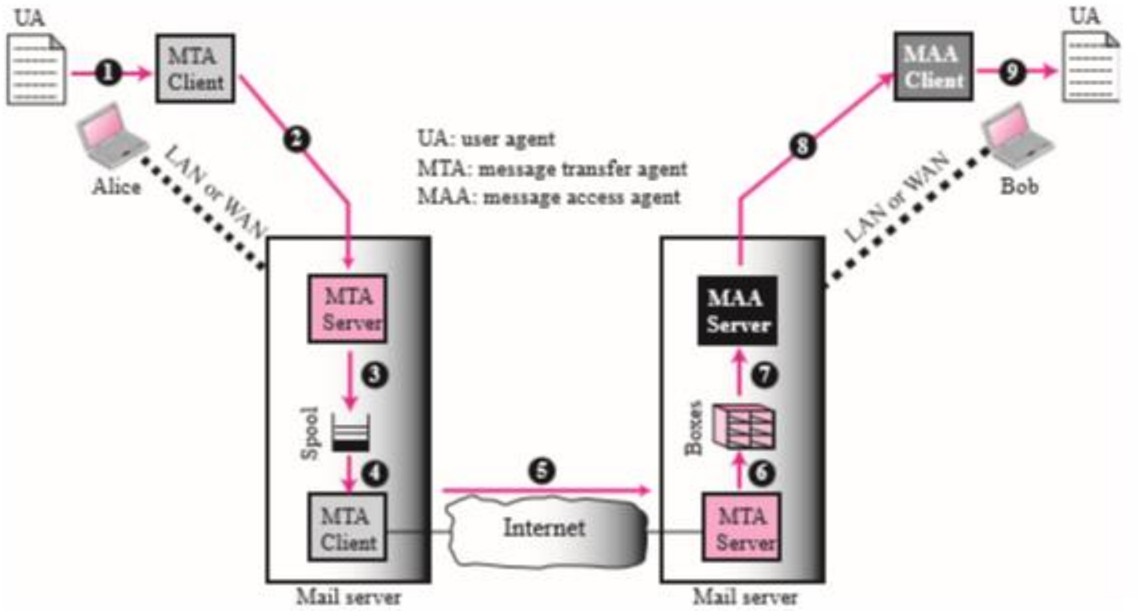


**Third Scenario**

Alice still needs a user agent to prepare her message. She then needs to send the message through the LAN or WAN. This can be done through a pair of message transfer agents (client and server). Whenever Alice has a message to send, she calls the user agent which, in turn, calls the MTA client. The MTA client establishes a connection with the MTA server on the system, which is running all the time. The system at Alice's site queues all messages received. It then uses an MTA client to send the messages to the system at Bob's site; the system receives the message and stores it in Bob's mailbox.

**Fourth Scenario**

In the fourth and most common scenario, Bob is also connected to his mail server by a WAN or a LAN. After the message has arrived at Bob's mail server, Bob needs to retrieve it. Here, we need another set of client-server agents, which we call message access agents (MAAs). Bob uses an MAA client to retrieve his messages. The client sends a request to the MAA server, which is running all the time, and requests the transfer of the messages. The situation is shown in Figure 23.4. There are two important points we need to emphasize here.



First, Bob cannot bypass the mail server and use the MTA server directly. To use the MTA server directly, Bob would need to run the MTA server all the time because he does not know when a message will arrive. This implies that Bob must keep his computer on all the time if he is connected to his system through a LAN. If he is connected through a WAN, he must keep the connection up all the time. Neither of these situations is feasible today. Second, note that Bob needs another pair of client-server programs: message access programs. This is because an MTA client-server program is a push program: the client pushes the message to the server. Bob needs a pull program. The client needs to pull the message from the 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.

**Services Provided by a User Agent**

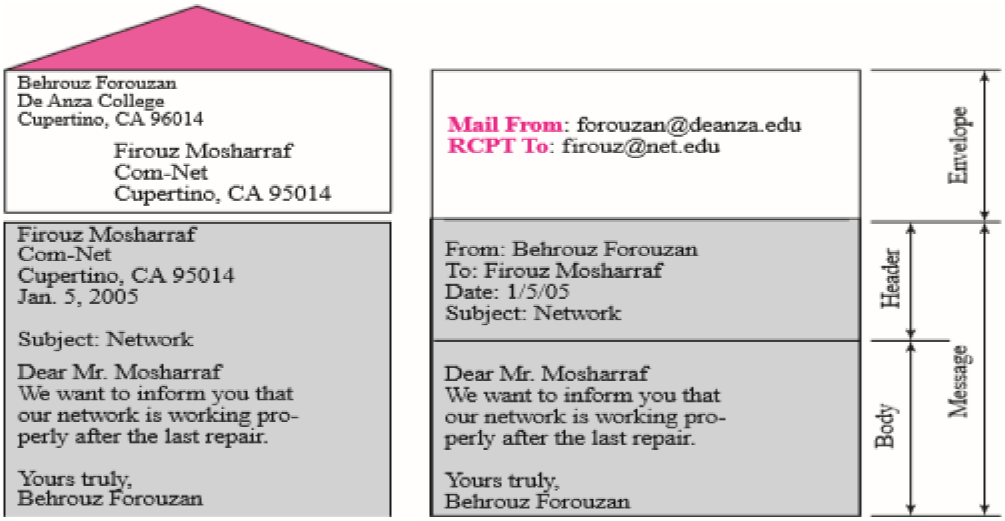
A user agent is a software package (program) that composes, reads, replies to, and for- wards messages. It also handles local mailboxes on the user computers.

**User Agent Types**

There are two types of user agents: command-driven and GUI-based. Command-driven user agents belong to the early days of electronic mail. They are still present as the underlying user agents in servers. A command-driven user agent normally accepts a one- character command from the keyboard to perform its task. For example, a user can type the character r, at the command prompt, to reply to the sender of the message, or type the character R to reply to the sender and all recipients. Modern user agents are GUI-based. They contain graphical user interface (GUI) components that allow the user to interact with the software by using both the keyboard and the mouse. They have graphical components such as icons, menu bars, and windows that make the services easy to access.

**Sending Mail**

To send mail, the user, through the UA, creates mail that looks very similar to postal mail. It has an envelope and a message.



Behrouz Forouzan
De Anza College
Cupertino, CA 96014

Firouz Mosharraf
Com-Net
Cupertino, CA 95014

Firouz Mosharraf
Com-Net
Cupertino, CA 95014
Jan. 5, 2005

Subject: Network

Dear Mr. Mosharraf
We want to inform you that our network is working pro-perly after the last repair.

Yours truly,
Behrouz Forouzan

Mail From: forouzan@deanza.edu
RCPT To: firouz@net.edu

From: Behrouz Forouzan
To: Firouz Mosharraf
Date: 1/5/05
Subject: Network

Dear Mr. Mosharraf
We want to inform you that our network is working pro-perly after the last repair.

Yours truly,
Behrouz Forouzan

**Format of an Email**

**Envelope**

The envelope usually contains the sender address, the receiver address, and other information.

**Message**

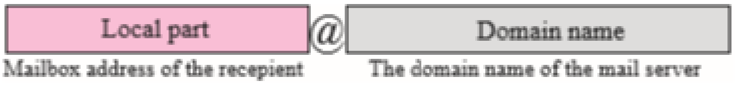
The message contains the header and the body. The header of the message defines the sender, the receiver, the subject of the message, and some other information. The body of the message contains the actual information to be read by the recipient.

**Receiving Mail**

The user agent is triggered by the user (or a timer). If a user has mail, the UA informs the user with a notice. If the user is ready to read the mail, a list is displayed in which each line contains a summary of the information about a particular message in the mail- box. The summary usually includes the sender mail address, the subject, and the time the mail was sent or received. The user can select any of the messages and display its contents on the screen.

**Addresses**

To deliver mail, a mail handling system must use an addressing system with unique addresses. In the Internet, the address consists of two parts: a local part and a domain name, separated by an @ sign.


Local part
@
Domain name
Mailbox address of the recepient
The domain name of the mail server

**Format of Email Address**

**Local Part**

The local part defines the name of a special file, called the user mailbox, where all of the mail received for a user is stored for retrieval by the message access agent.

**Domain Name**

The second part of the address is the domain name. An organization usually selects one or more hosts to receive and send e-mail; they are sometimes called mail servers or exchangers.

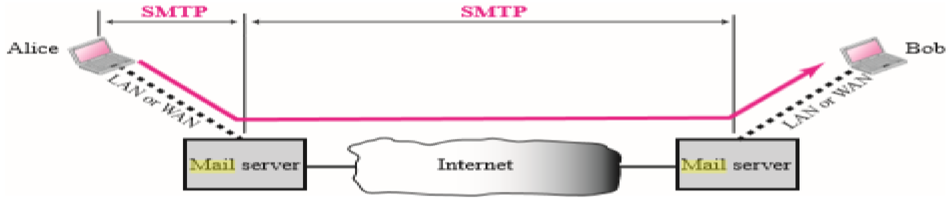
**Mailing List or Group List**

Electronic mail allows one name, an alias, to represent several different e-mail addresses; this is called a mailing list. Every time a message is to be sent, the sys- tem checks the recipient's name against the alias database; if there is a mailing list for the defined alias, separate messages, one for each entry in the list, must be pre- pared and handed to the MTA. If there is no mailing list for the alias, the name itself is the receiving address and a single message is delivered to the mail transfer entity.

### SMTP - Mail Agent

The actual mail transfer is done through message transfer agents (MTAs). To send mail, a system must have the client MTA, and to receive mail, a system must have a server MTA. The formal protocol that defines the MTA client and server in the Internet is called Simple Mail Transfer Protocol (SMTP). As we said before, two pairs of MTA client- server programs are used in the most common situation (fourth scenario).

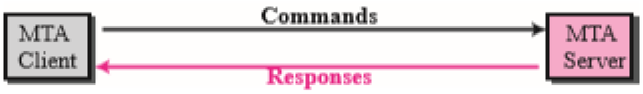
SMTP is used two times, between the sender and the sender’s mail server and between the two mail servers. As we will see shortly, another protocol is needed between the mail server and the receiver. SMTP simply defines how commands and responses must be sent back and forth. Each network is free to choose a software package for implementation.



Mail Agent

### Commands and Responses

SMTP uses commands and responses to transfer messages between an MTA client and an MTA server.

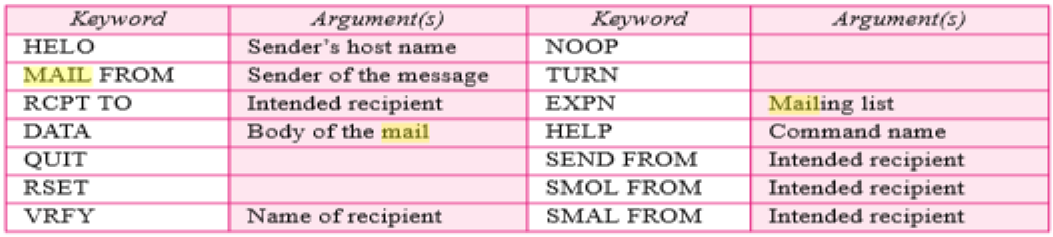


Each command or reply is terminated by a two-character (carriage return and line feed) end-of-line token. Figure

### Commands

Commands are sent from the client to the server. The format of a command is shown below:

It consists of a keyword followed by zero or more arguments. SMTP defines 14 commands listed in below table and described in more detail below.



| *Keyword* | *Argument(s)* | *Keyword* | *Argument(s)* |
|---|---|---|---|
| HELO | Sender's host name | NOOP | |
| MAIL FROM | Sender of the message | TURN | |
| RCPT TO | Intended recipient | EXPN | Mailing list |
| DATA | Body of the mail | HELP | Command name |
| QUIT | | SEND FROM | Intended recipient |
| RSET | | SMOL FROM | Intended recipient |
| VRFY | Name of recipient | SMAL FROM | Intended recipient |

**HELO:** This command is used by the client to identify itself. The argument is the domain name of the client host. The format is

HELO: challenger.atc.fhda.edu

**MAIL FROM:** This command is used by the client to identify the sender of the message. The argument is the e-mail address of the sender (local part plus the domain name). The format is

MAIL FROM: forouzan@challenger.atc.fhda.edu

**RCPT TO:** This command is used by the client to identify the intended recipient of the message. The argument is the e-mail address of the recipient. If there are multi- ple recipients, the command is repeated. The format is

RCPT TO: betsy@mcgraw-hill.com

**DATA:** This command is used to send the actual message. All lines that follow the DATA command are treated as the mail message. The message is terminated by a line containing just one period. The format is Keyword: argument(s)

DATA

This is the message to be sent to the McGraw-Hill Company. .

**QUIT:** This command terminates the message. The format is

QUIT

**RSET:**

This command aborts the current mail transaction. The stored information about the sender and recipient is deleted. The connection will be reset.

RSET

**VRFY:** This command is used to verify the address of the recipient, which is sent as the argument. The sender can ask the receiver to confirm that a name identifies a valid recipient. Its format is

VRFY: betsy@mcgraw-hill.com

**NOOP:** This command is used by the client to check the status of the recipient. It requires an answer from the recipient. Its format is

NOOP

**TURN:** This command lets the sender and the recipient switch positions, whereby the sender becomes the recipient and vice versa. However, most SMTP implemen- tations today do not support this feature. The format is

TURN

**EXPN:** This command asks the receiving host to expand the mailing list sent as the arguments and to return the mailbox addresses of the recipients that comprise the list. The format is

EXPN: x y z

**HELP:** This command asks the recipient to send information about the command sent as the argument. The format is

HELP: mail

**SEND FROM:** This command specifies that the mail is to be delivered to the terminal of the recipient, and not the mailbox. If the recipient is not logged in, the mail is bounced back. The argument is the address of the sender. The format is

SEND FROM: forouzan@fhda.atc.edu

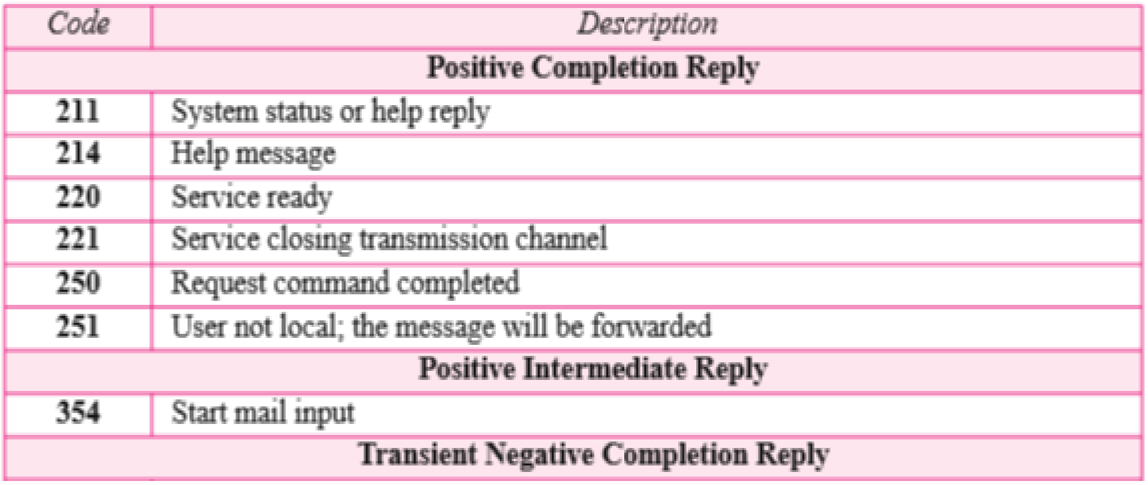
**SMOL FROM:** This command specifies that the mail is to be delivered to the termi- nal or the mailbox of the recipient. This means that if the recipient is logged in, the mail is delivered only to the terminal. If the recipient is not logged in, the mail is delivered to the mailbox. The argument is the address of the sender. The format is

SMOL FROM: forouzan@fhda.atc.edu

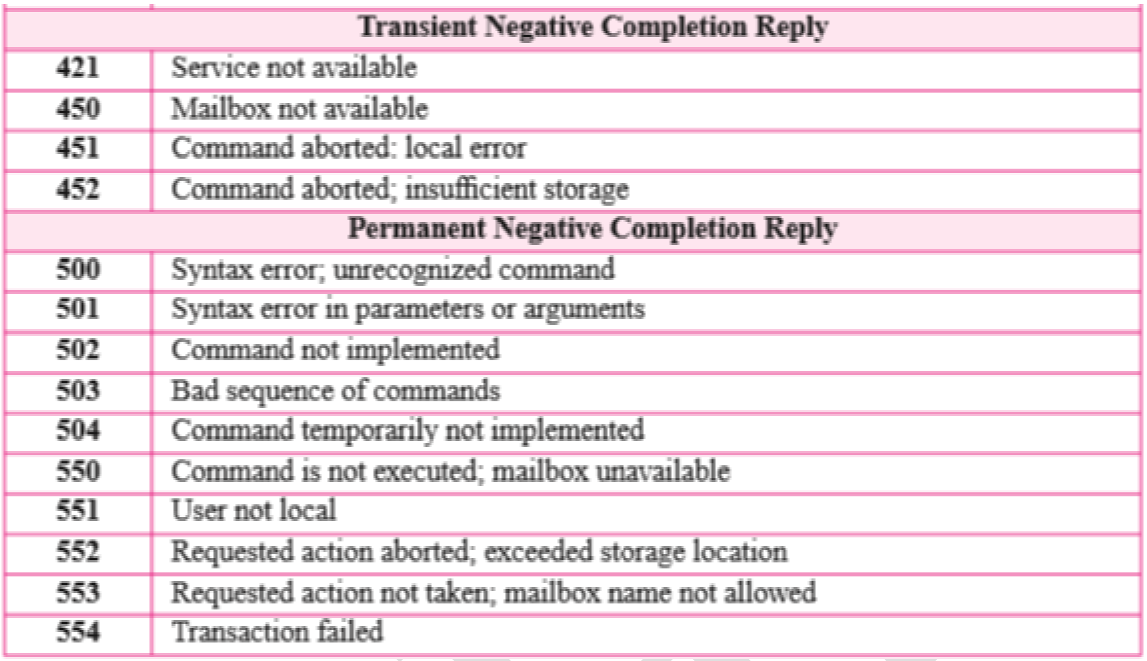
**SMAL FROM:** This command specifies that the mail is to be delivered to the terminal and the mailbox of the recipient. This means that if the recipient is logged in, the mail is delivered to the terminal and the mailbox. If the recipient is not logged in, the mail is delivered only to the mailbox. The argument is the address of the sender. The format is

SMAL FROM: forouzan@fhda.atc.edu

**Responses**: Responses are sent from the server to the client. A response is a three-digit code that may be followed by additional textual information.



| *Code* | *Description* |
|---|---|
| | **Positive Completion Reply** |
| **211** | System status or help reply |
| **214** | Help message |
| **220** | Service ready |
| **221** | Service closing transmission channel |
| **250** | Request command completed |
| **251** | User not local; the message will be forwarded |
| | **Positive Intermediate Reply** |
| **354** | Start mail input |
| | **Transient Negative Completion Reply** |



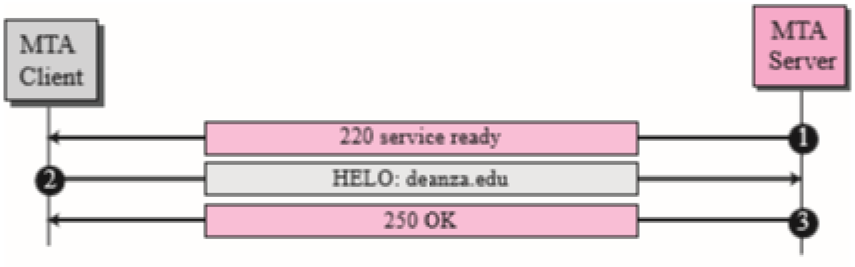
| | |
|---|---|
| | **Transient Negative Completion Reply** |
| **421** | Service not available |
| **450** | Mailbox not available |
| **451** | Command aborted: local error |
| **452** | Command aborted; insufficient storage |
| | **Permanent Negative Completion Reply** |
| **500** | Syntax error; unrecognized command |
| **501** | Syntax error in parameters or arguments |
| **502** | Command not implemented |
| **503** | Bad sequence of commands |
| **504** | Command temporarily not implemented |
| **550** | Command is not executed; mailbox unavailable |
| **551** | User not local |
| **552** | Requested action aborted; exceeded storage location |
| **553** | Requested action not taken; mailbox name not allowed |
| **554** | Transaction failed |

**Mail Transfer Phases**

The process of transferring a mail message occurs in three phases: connection establishment, mail transfer, and connection termination.

**Connection Establishment**

After a client has made a TCP connection to the well-known port 25, the SMTP server starts the connection phase. This phase involves the following three steps, which are illustrated in Figure



**Connection Establishment**

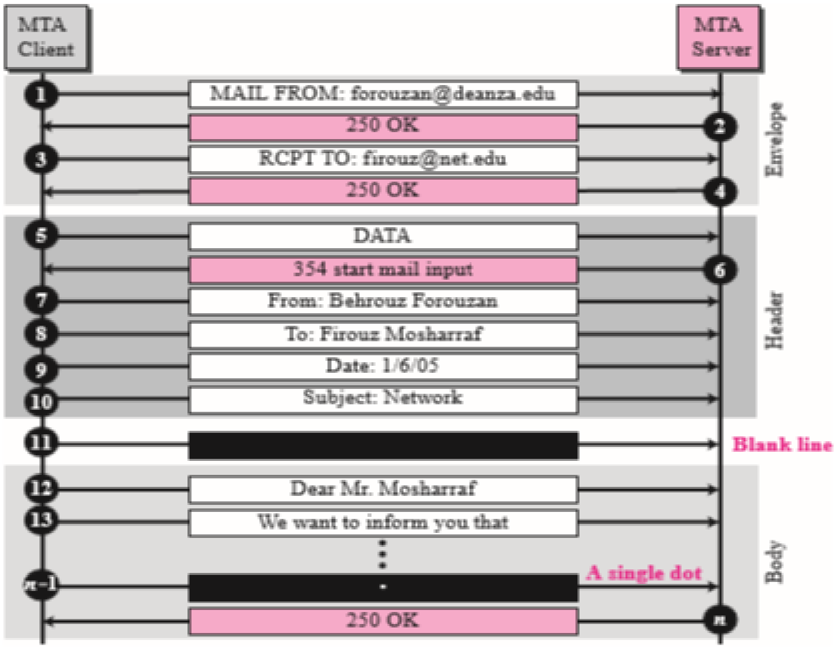
1. The server sends code 220 (service ready) to tell the client that it is ready to receive mail. If the server is not ready, it sends code 421 (service not available).
2. The client sends the HELO message to identify itself using its domain name address. This step is necessary to inform the server of the domain name of the client. Remember that during TCP connection establishment, the sender and receiver know each other through their IP addresses.

3. The server responds with code 250 (request command completed) or some other code depending on the situation.

**Message Transfer**

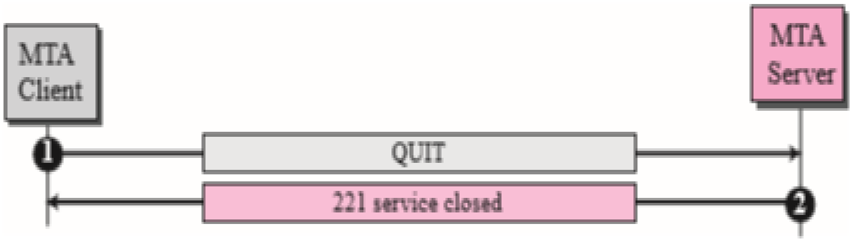
After connection has been established between the SMTP client and server, a single message between a sender and one or more recipients can be exchanged. This phase involves eight steps. Steps 3 and 4 are repeated if there is more than one recipient.

1. The client sends the MAIL FROM message to introduce the sender of the message. It includes the mail address of the sender (mailbox and the domain name). This step is needed to give the server the return mail address for returning errors and reporting messages.
2. The server responds with code 250 or some other appropriate code.
3. The client sends the RCPT TO (recipient) message, which includes the mail address of the recipient.
4. The server responds with code 250 or some other appropriate code.
5. The client sends the DATA message to initialize the message transfer.
6. The server responds with code 354 (start mail input) or some other appropriate message.
7. The client sends the contents of the message in consecutive lines. Each line is ter- minated by a two-character end-of-line token (carriage return and line feed). The message is terminated by a line containing just one period.
8. The server responds with code 250 (OK) or some other appropriate code.



### Connection Termination

After the message is transferred successfully, the client terminates the connection. This phase involves two steps.



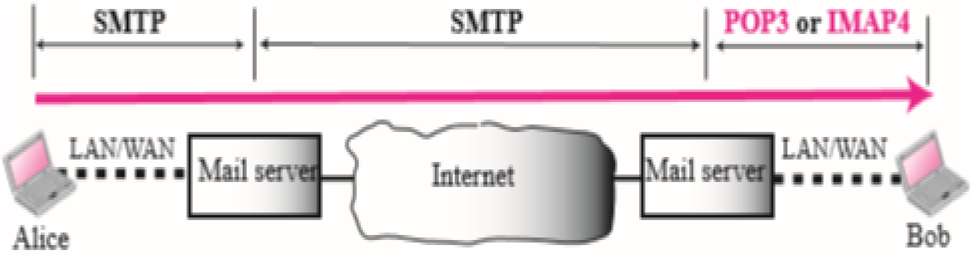
**Connection Termination**

1. The client sends the QUIT command.
2. The server responds with code 221 or some other appropriate code.

After the connection termination phase, the TCP connection must be closed.

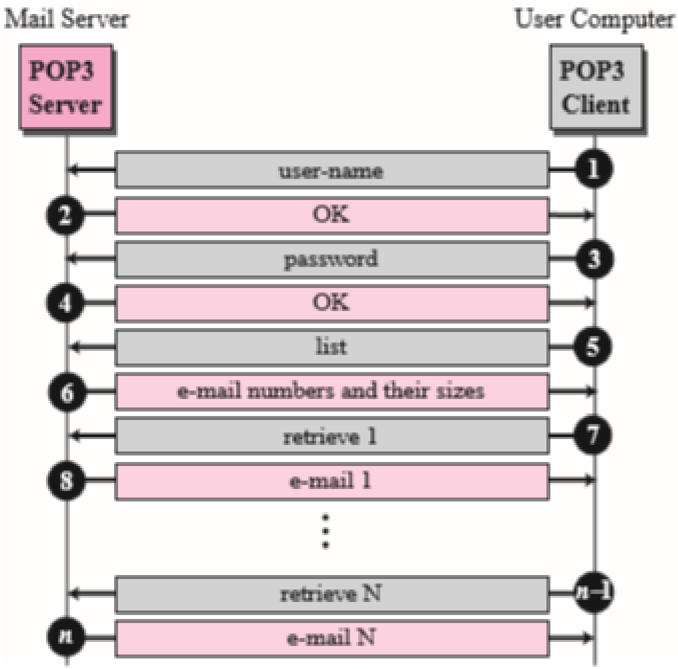
### Message Access Agent: POP & IMAP

The first and the 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. In other words, the direction of the bulk data (messages) is 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 are 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 and 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. The delete mode is normally used when the user is working at her permanent computer and can save and organize the received mail after reading or replying. The keep mode is normally used when the user accesses her mail away from her primary computer (e.g., a laptop). The mail is read but kept in the sys- tem for later retrieval and organizing.

**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. (Of course, the user can create folders on her own computer.) In addition, POP3 does not allow the user to par- tially 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 lim- ited and the e-mail contains multimedia with high bandwidth requirements.

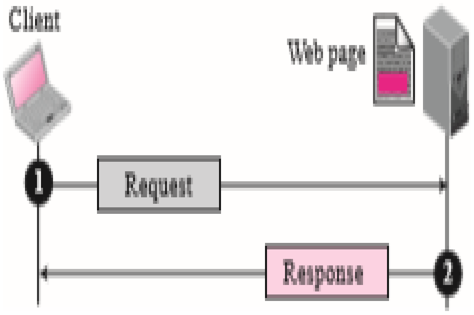
- A user can create, delete, or rename mailboxes on the mail server.
- A user can create a hierarchy of mailboxes in a folder for e-mail storage.

### 4. WWW

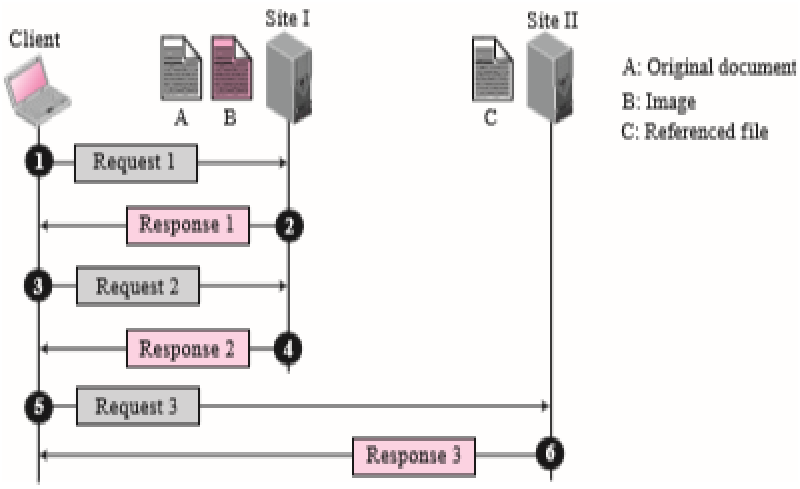
**Architecture**

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 documents, referred to as 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 link 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.

**Example 1:** Assume we need to retrieve a Web page that contains the biography of a famous character with some pictures, which are embedded in the page itself. Since the pictures are not stored as separate files, the whole document is a simple Web page. It can be retrieved using one single request/ response transaction.



**Example 2**: Now assume we need to retrieve a scientific document that contains one reference to another text file and one reference to a large image. Figure 22.2 shows the situation. The main document and the image are stored in two separate files in the same site (file A and file B); the referenced text file is stored in another site (file C). Since we are dealing with three different files, we need three transactions if we want to see the whole document. The first transaction (request/response) retrieves a copy of the main document (file A), which has a ref- erence (pointer) to the second and the third files. When a copy of the main document is retrieved and browsed, the user can click on the reference to the image to invoke the second transaction and retrieve a copy of the image (file B). If the user further needs to see the contents of the referenced text file, she can click on its reference (pointer) invoking the third transaction and retrieving a copy of the file C. Note that although files A and B both are stored in site I, they are independent files with different names and addresses. Two transactions are needed to retrieve them.

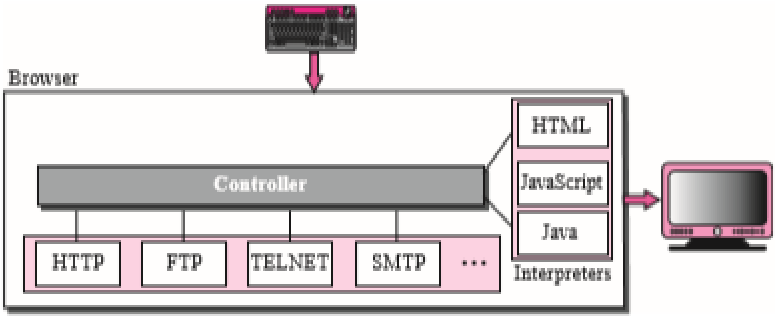


**Hypertext and Hypermedia**

The three previous examples show the idea of hypertext and hypermedia. Hypertext means creating documents that refer to other documents. In a hypertext document, a part of text can be defined as a link to another document. When a hypertext is viewed with a browser, the link can be clicked to retrieve the other document. Hypermedia is a term applied to document that contains links to other textual document or documents containing graphics, video, or audio.

**Web Client (Browser)**

A variety of vendors offer commercial browsers that interpret and display a Web document, and all of them use nearly the same architecture. Each browser usually consists of three parts: a controller, client protocol, and interpreters.



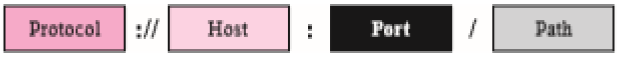
The controller receives input from the keyboard or the mouse and uses the client programs to access the document. After the document has been accessed, the controller uses one of the interpreters to display the document on the screen. The client protocol can be one of the protocols described previously such as FTP, or TELNET, or HTTP (as discussed later in the chapter). The interpreter can be HTML, Java, or JavaScript, depending on the type of document. We discuss the use of these interpreters based on the document type later in the chapter. Some commercial browsers include Internet Explorer, Netscape Navigator, and Firefox.

**Web Server**

The Web page is stored at the server. Each time a client request arrives, the correspond- ing 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 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 client that wants to access a Web page needs the file name and the address. To facilitate the access of documents distributed throughout the world, HTTP uses locators. The uniform resource locator (URL) is a standard locator for specifying any kind of information on the Internet. The URL defines four things: protocol, host computer, port, and path.



The protocol is the client-server application program used to retrieve the docu- ment. Many different protocols can retrieve a document; among them are Gopher, FTP, HTTP, News, and TELNET. The most common today is HTTP. The host is the domain name of the computer on which the information is located. Web pages are usually stored in computers, and computers are given domain name aliases that usually begin with the characters “www”. This is not mandatory, however, as the host can have any domain name. The URL can optionally contain the port number of the server. If the port is included, it is inserted between the host and the path, and it is separated from the host by a colon. Path is the pathname of the file where the information is located. Note that the path can itself contain slashes that, in the UNIX operating system, separate the directories from the subdirectories and files. In other words, the path defines the complete file name where the document is stored in the directory system.

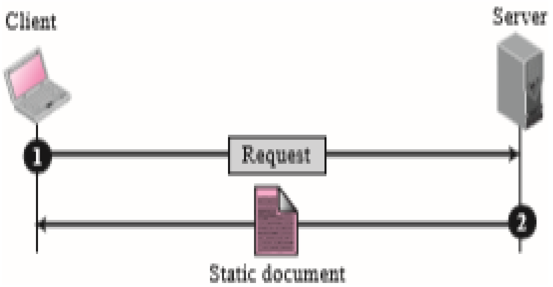
**Web Document**

The documents in the WWW can be grouped into three broad categories: static, dynamic, and active. The category is based on the time the contents of the document are determined.

**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. Of course, the contents in the server can be changed, but the user cannot change them.

When a client accesses the document, a copy of the document is sent. The user can then use a browsing program to display the document.

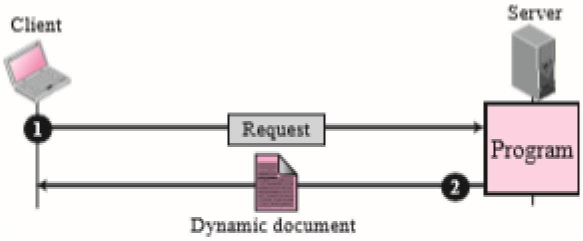


Static documents are prepared using one of the several languages: Hypertext Markup Language (HTML), Extensible Markup Language (XML), Extensible Style Language (XSL), and Extended Hypertext Markup Language (XHTML).

**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 output 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. Time and date are kinds of information that are dynamic in that they change from moment to moment. The client can ask the server to run a program such as the date program in UNIX and send the result of the program to the client.

Common Gateway Interface (CGI) The Common Gateway Interface (CGI) is a technology that creates and handles dynamic documents. CGI is a set of standards that defines how a dynamic document is written, how data are input to the program, and how the output result is used. CGI is not a new language; instead, it allows programmers to use any of several languages such as C, C++, Bourne Shell, Korn Shell, C Shell, Tcl, or Perl. The only thing that CGI defines is a set of rules and terms that the programmer must follow. The term common in CGI indicates that the standard defines a set of rules that is common to any language or platform. The term gateway here means that a CGI pro- gram can be used to access other resources such as databases, graphic packages, and so on. The term interface here means that there is a set of predefined terms, variables, calls, and so on that can be used in any CGI program. A CGI program in its simplest form is code written in one of the languages supporting CGI. Any programmer who can encode a sequence of thoughts in a program and knows the syntax of one of the above- mentioned languages can write a simple CGI program.

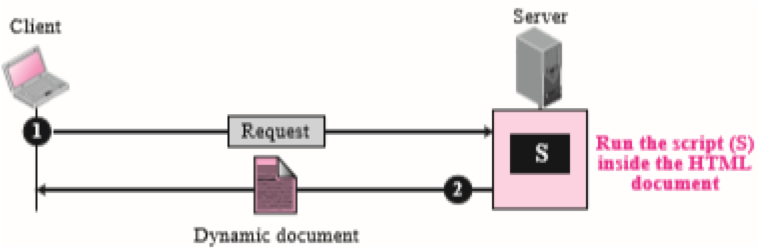


**Input:** In traditional programming, when a program is executed, parameters can be passed to the program. Parameter passing allows the programmer to write a generic program that can be used in different situations. For example, a generic copy program can be written to copy any file to another. A user can use the program to copy a file named x to another file named y by passing x and y as parameters. The input from a browser to a server is sent using a form. If the information in a form is small (such as a word), it can be appended to the URL after a question mark. For example, the following URL is carrying form information (23, a value):

When the server receives the URL, it uses the part of the URL before the question mark to access the program to be run, and it interprets the part after the question mark (23) as the input sent by the client. It stores this string in a variable. When the CGI pro- gram is executed, it can access this value. If the input from a browser is too long to fit in the query string, the browser can ask the server to send a form. The browser can then fill the form with the input data and send it to the server. The information in the form can be used as the input to the CGI program.

**Output:** Output The whole idea of CGI is to execute a CGI program at the server site and send the output to the client (browser). The output is usually plain text or a text with HTML structures; however, the output can be a variety of other things. It can be graphics or binary data, a status code, instructions to the browser to cache the result, or instructions to the server to send an existing document instead of the actual output. To let the client know about the type of document sent, a CGI program creates headers. As a matter of fact, the output of the CGI program always consists of two parts: a header and a body. The header is separated by a blank line from the body. This means any CGI program first creates the header, then a blank line, and then the body. Although the header and the blank line are not shown on the browser screen, the header is used by the browser to interpret the body.

**Scripting Technologies for Dynamic Documents**: The problem with CGI technology is the inefficiency that results if part of the dynamic document that is to be created is fixed and not changing from request to request. For example, assume that we need to retrieve a list of spare parts, their availability, and prices for a specific car brand. Although the availability and prices vary from time to time, the name, description, and the picture of the parts are fixed. If we use CGI, the program must create an entire document each time a request is made. The solution is to create a file containing the fixed part of the document using HTML and embed a script, a source code that can be run by the server to provide the varying availability and price section.

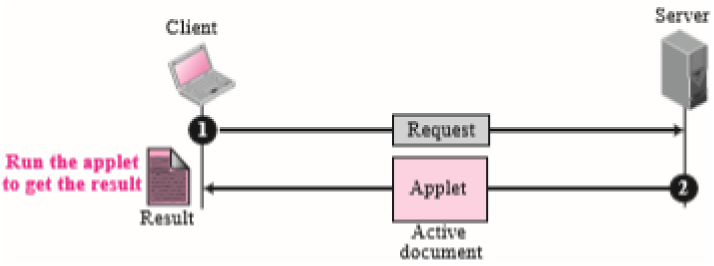


A few technologies have been involved in creating dynamic documents using scripts. Among the most common are Hypertext Preprocessor (PHP), which uses the Perl language; Java Server Pages (JSP), which uses the Java language for scripting; Active Server Pages (ASP), a Microsoft product, which uses Visual Basic language for scripting; and ColdFusion, which embeds SQL database queries in the HTML document.

**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. The program definitely needs to be run at the client site where the animation or inter- action takes place. When a browser requests an active document, the server sends a copy of the document or a script. The document is then run at the client (browser) site.

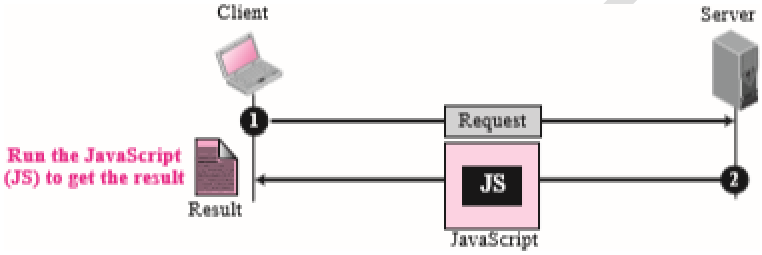
**Java Applets:** One way to create an active document is to use Java applets. Java is a combination of a high-level programming language, a run-time environment, and a class library that allows a programmer to write an active document (an applet) and a browser to run it. It can also be a stand-alone program that doesn't use a browser. An applet is a program written in Java on the server. It is compiled and ready to be run. The document is in bytecode (binary) format.



The client process (browser) creates an instance of this applet and runs it. A Java applet can be run by the browser in two ways. In the first method, the browser can directly request the Java applet program in the

URL and receive the applet in binary form. In the second method, the browser can retrieve and run an HTML file that has embedded the address of the applet as a tag.

**JavaScript**: The idea of scripts in dynamic documents can also be used for active documents. If the active part of the document is small, it can be written in a scripting language; then it can be interpreted and run by the client at the same time. The script is in source code (text) and not in binary form. The scripting technology used in this case is usually JavaScript. JavaScript, which bears a small resemblance to Java, is a very high level scripting language developed for this purpose.



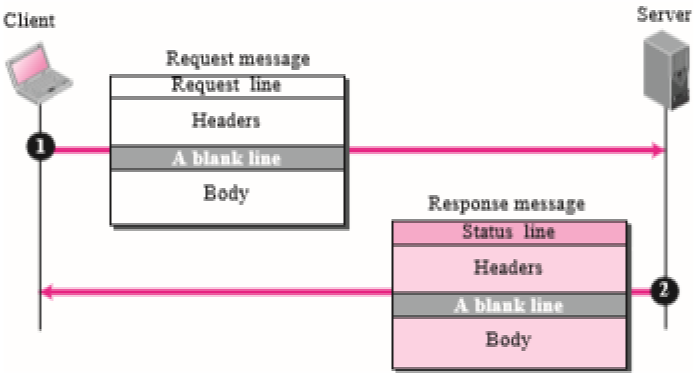
### 6. HTTP

The Hypertext Transfer Protocol (HTTP) is a protocol used mainly to access data on the World Wide Web. HTTP functions like a combination of FTP and SMTP. It is similar to FTP because it transfers files and uses the services of TCP. However, it is much simpler than FTP because it uses only one TCP connection. There is no separate control connection; only data are transferred between the client and the server. HTTP is like SMTP because the data transferred between the client and the server look like SMTP messages. In addition, the format of the messages is controlled by MIME-like headers. Unlike SMTP, the HTTP messages are not destined to be read by humans; they are read and interpreted by the HTTP server and HTTP client (browser). SMTP messages are stored and forwarded, but HTTP messages are delivered immediately. The commands from the client to the server are embedded in a request message. The contents of the requested file or other information are embedded in a response message. HTTP uses the services of TCP on well-known port 80.

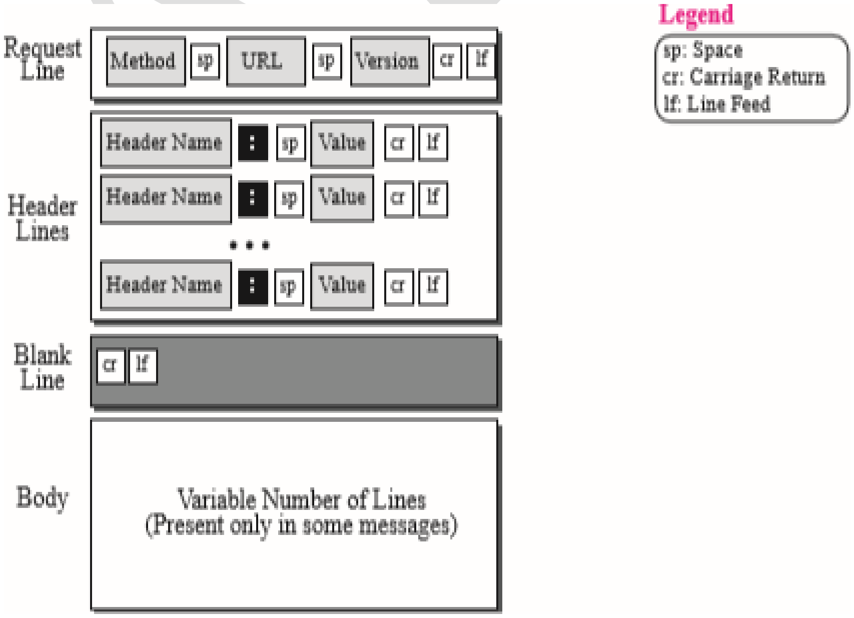
HTTP It is similar to FTP because it transfers files and uses the services of TCP. However, it is much simpler than FTP because it uses only one TCP connection. There is no separate control connection; only data are transferred between the client and the server. HTTP is like SMTP because the data transferred between the client and the server look like SMTP messages. In addition, the format of the messages is controlled by MIME-like headers. Unlike SMTP, the HTTP messages are not destined to be read by humans; they are read and interpreted by the HTTP server and HTTP client (browser). SMTP messages are stored and forwarded, but HTTP messages are delivered immedi- ately. The commands from the client to the server are embedded in a request message. The contents of the requested file or other information are embedded in a response message. HTTP uses the services of TCP on well-known port 80.

**HTTP Transaction**

Figure below, illustrates the HTTP transaction between the client and server. Although HTTP uses the services of TCP, HTTP itself is a stateless protocol, which means that the server does not keep information about the client. The client initializes the transaction by sending a request. The server replies by sending a response.

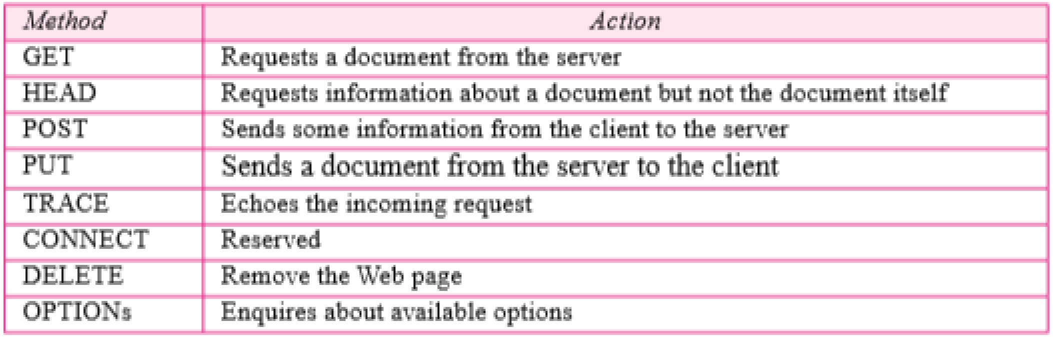


**Request Message**: The format of the request is shown in Figure below. A request message consists of a request line, a header, and sometimes a body.



**Request Line**: The first line in a request message is called a request line. There are three fields in this line separated by some character delimiter as shown in above Figure. The fields are called methods, URL, and

Version. These three should be separated by a space character. At the end two characters, a carriage return followed by a line feed, terminate the line. The method field defines the request type. In version 1.1 of HTTP, several methods are defined, as shown in table below.



| *Method* | *Action* |
|---|---|
| GET | Requests a document from the server |
| HEAD | Requests information about a document but not the document itself |
| POST | Sends some information from the client to the server |
| PUT | Sends a document from the server to the client |
| TRACE | Echoes the incoming request |
| CONNECT | Reserved |
| DELETE | Remove the Web page |
| OPTIONs | Enquires about available options |

The second field, URL, was discussed earlier in the chapter. It defines the address and name of corresponding Web page. The third field, version, gives the version of the protocol; the most current version of HTTP is 1.1.

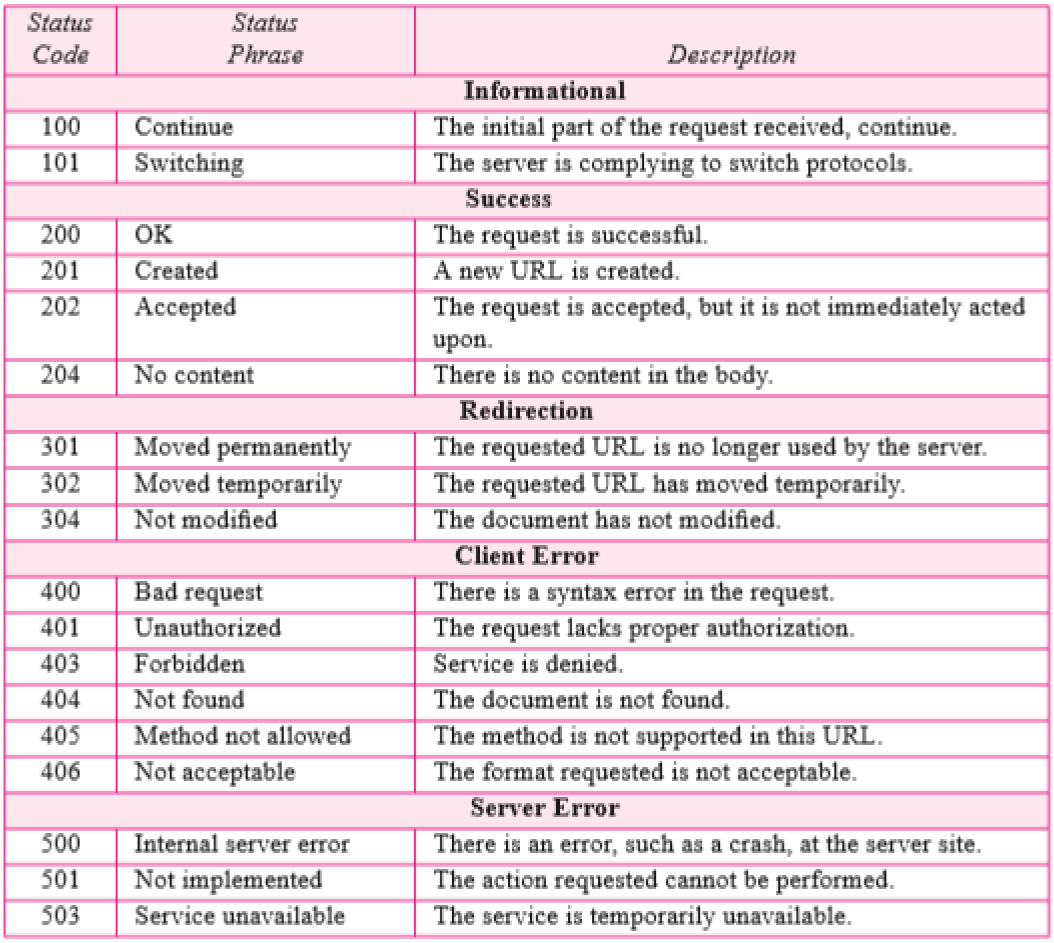
**Body**: In Request Message The body can be present in a request message. Usually, it contains the comment to be sent.

**Response Message**: A response message con- sists of a status line, header lines, a blank line and sometimes a body.

**Status Line:** 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. The possible values for the status code and status phrase are shown in Table below.

**Header Lines**: In Response Message 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. For example, the sender can send extra information about the document.

Each header line has a header name, a colon, a space, and a header value. We will show some header lines in the examples at the end of this chapter.



| Status Code | Status Phrase | Description |
|---|---|---|
| | | **Informational** |
| 100 | Continue | The initial part of the request received, continue. |
| 101 | Switching | The server is complying to switch protocols. |
| | | **Success** |
| 200 | OK | The request is successful. |
| 201 | Created | A new URL is created. |
| 202 | Accepted | The request is accepted, but it is not immediately acted upon. |
| 204 | No content | There is no content in the body. |
| | | **Redirection** |
| 301 | Moved permanently | The requested URL is no longer used by the server. |
| 302 | Moved temporarily | The requested URL has moved temporarily. |
| 304 | Not modified | The document has not modified. |
| | | **Client Error** |
| 400 | Bad request | There is a syntax error in the request. |
| 401 | Unauthorized | The request lacks proper authorization. |
| 403 | Forbidden | Service is denied. |
| 404 | Not found | The document is not found. |
| 405 | Method not allowed | The method is not supported in this URL. |
| 406 | Not acceptable | The format requested is not acceptable. |
| | | **Server Error** |
| 500 | Internal server error | There is an error, such as a crash, at the server site. |
| 501 | Not implemented | The action requested cannot be performed. |
| 503 | Service unavailable | The service is temporarily unavailable. |

**Body**: The body contains the document to be sent from the server to the client. The body is present unless the response is an error message.

**6. Cookies**

The World Wide Web was originally designed as a stateless entity. A client sends a request; a server responds. Their relationship is over. The original design of WWW, retrieving publicly available documents, exactly fits this purpose. Today the Web has other functions; some are listed below:

- Websites are being used as electronic stores that allow users to browse through the store, select wanted items, put them in an electronic cart, and pay at the end with a credit card.
- Some websites need to allow access to registered clients only.
- Some websites are used as portals: The user selects the Web pages he wants to see.
- Some websites are just advertising. For these purposes, the cookie mechanism was devised.

**Creating and Storing Cookies**: The creation and storing of cookies depend on the implementation; however, the principle is the same.

1. When a server receives a request from a client, it stores information about the client in a file or a string. The information may include the domain name of the client, the contents of the cookie (information the server has gathered about the client such as name, registration number, and so on), a timestamp, and other information depending on the implementation.

2. The server includes the cookie in the response that it sends to the client. 3. When the client receives the response, the browser stores the cookie in the cookie directory, which is sorted by the domain server name.

**Using Cookies**:

- When a client sends a request to a server, the browser looks in the cookie directory to see if it can find a cookie sent by that server. If found, the cookie is included in the request. When the server receives the request, it knows that this is an old client, not a new one. Note that the contents of the cookie are never read by the browser or disclosed to the user. It is a cookie made by the server and eaten by the server. Now let us see how a cookie is used for the four previously mentioned purposes:
- An electronic store (e-commerce) can use a cookie for its client shoppers. When a client selects an item and inserts it into a cart, a cookie that contains information about the item, such as its number and unit price, is sent to the browser. If the client selects a second item, the cookie is updated with the new selection information. And so on. When the client finishes shopping and wants to check out, the last cookie is retrieved and the total charge is calculated.
- The site that restricts access to registered clients only sends a cookie to the client when the client registers for the first time. For any repeated access, only those clients that send the appropriate cookie are allowed. A Web portal uses the cookie in a similar way. When a user selects her favorite pages, a cookie is made and sent. If the site is accessed again, the cookie is sent to the server to show what the client is looking for.
- A cookie is also used by advertising agencies. An advertising agency can place banner ads on some main website that is often visited by users. The advertising agency supplies only a URL that gives the banner address instead of the banner itself. When a user visits the main website and clicks the icon of an advertised corporation, a request is sent to the advertising agency. The advertising agency sends the banner, a GIF file for example, but it also includes a cookie with the ID of the user. Any future use of the banners adds to the database that profiles the Web behavior of the user. The advertising agency has compiled the interests of the user and can sell this

information to other parties. This use of cookies has made them very controversial. Hopefully, some new regulations will be devised to preserve the privacy of users

**Web Caching: Proxy Server:** 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. Note that the proxy server acts both as a server and client. When it receives a request from a client for which it has a response, it acts as a server and sends the response to the client. When it receives a request from a client for which it does not have a response, it first acts as a server and sends a request to the target server. When the response has been received, it acts again as a server and sends the response to the client.

**Proxy Server Location**: The proxy servers are normally located at the client site. This means that we can have a hierarchy of proxy servers as shown below:

1. A client computer can also be used as a proxy server in a small capacity that stores responses to requests often invoked by the client.
2. In a company, a proxy server may be installed on the computer LAN to reduce the load going out of and coming into the LAN.
3. An ISP with many customers can install a proxy server to reduce the load going out of and coming into the ISP network.

**Cache Update**: A very important question is how long a response should remain in the proxy server before being deleted and replaced. Several different strategies are used for this purpose. One solution is to store the list of sites whose information remains the same for a while. For example, a news agency may change its news page every morning. This means that a proxy server can get the news early in the morning and keep it until the next day. Another recommendation is to add some headers to show the last modification time of the information. The proxy server can then use the information in this header to guess how long the information would be valid. There are more recommendations for Web caching, but we leave them to more specific books on this subject.

## Possible Questions

### 2 Mark Questions

1. What is WWW?
2. What is the use of Http?
3. What is the use of TELNET?
4. Who is an email user Agent?
5. List the scenario of Email.

### 6 Mark Questions

1. What is the use of TELNET? Explain the working methodology of TELNET.
2. What is the role of user agent in email?
3. Discuss about SMTP in detail
4. What is the purpose of Email? Explain the various scenario of Email.
5. Discuss about WWW architecture in detail
6. Write a detail note on Web Document.
7. What is the use of Dynamic Document? Explain about it.
8. Discuss about HTTP in detail.
9. Write a detail note on POP.
10. Discuss about email in general.

**KARPAGAM ACADEMY OF HIGHER EDUCATION**
**(Deemed University Established Under Section 3 of UGC Act, 1956)**
**Coimbatore – 641021, INDIA**
**Department of Computer Science, Applications & Information Technology**

## Unit - IV

| S.No | Questions | Opt1 | opt2 | opt3 | opt4 | Answer |
|---|---|---|---|---|---|---|
| 1 | ______ was designed at a time when most operating systems, such as UNIX, were operating in a time-sharing environment | TELNET | SSH | PUTTY | Email | TELNET |
| 2 | When a user logs into a local time-sharing system of TELNET , it is called _______ | Local Login | Remote Login | Login | Logout | Local Login |
| 3 | When a user wants to access an application program or utility located on a remote machine, he performs _______ | Local Login | Remote Login | Login | Logout | Remote Login |
| 4 | NVT refers to ____ | Network Visual Terminal | Network Virtual Task | Network Virtual Terminal | Network Visual Task | Network Virtual Terminal |
| 5 | How many Character set dose NVT has? | 3 | 2 | 5 | 4 | 2 |
| 6 | EOF in NVT Character Set refers to ____ | End of Filtering | Exit on File | End of File | Exit on Filtering | End of File |
| 7 | NOP in NVT Character Set refers to ____ | No open Terminal | No Operations | No Open Host | Non Operation | No Operations |
| 8 | _____ is a command used to display the file in Unix | Show | Type | ls | cat | cat |
| 9 | The ___ option of TELNET allows the receiver to interpret every 8-bit character received,except IAC, as binary data. | Echo | Binary | Supress | Status | Binary |
| 10 | Which of the following is not an negotiation option of TELNET? | WILL | WONT | DO | DOES | DOES |
| 11 | _____ command is used to enable options in TELNET | WILL | WONT | DO | DOES | WILL |
| 12 | _____ is the reply for WILL command in TELNET Option. | WONT | DO | DOES | NOT | DO |
| 13 | _____ is the reply for WILL command in TELNET Option. | WONT | DONT | DOES | NOT | DONT |
| 14 | Which of the following is an suboption negotiation of TELNET? | SE | SD | SR | SW | SE |
| 15 | Which of the following Characters not used to control a program running on remote server? | IP | AO | AYT | SX | SX |
| 16 | What is the use of EL character that is used to control a program running on server? | Enter Line | Enter Logical Value | Erase Logical Value | Erase Line | Erase Line |
| 17 | IP character of TELNET refers to ____ | Internet Process | Intranet Process | Interrupt program | Interrupt Process | Interrupt Process |
| 18 | AO character of TELNET refers to ____ | Abort OFF | Abort ON | Abort Out | Abort Output | Abort Output |
| 19 | To make control characters effective in special situations, TELNET uses _____ signaling. | In-bound | Out of Band | Special Band | Multiband | Out of Band |
| 20 | SSH refers to ______ | Security Shell | Shell Script Hello | Secure Shell | Security Script | Secure Shell |
| 21 | SSh Works in ____ layer | Application | Network | Datalink | Physical | Application |
| 22 | Telnet Runs in ____ layer | Application | Network | Datalink | Physical | Application |
| 23 | _____ is a program to prepare the message and store it in receivers mail box | User Agent | Mail Agent | Communication Agent | compose agent | User Agent |
| 24 | MTA in mailing refers to _____ | Message Tansist Agent | Mail Transist Agent | Mail Transfer Agent | Message Transfer Agent | Mail Transfer Agent |
| 25 | UA in mailing refers to ____ | Unit Agent | Universal Agent | User Agent | Unicast Agent | User Agent |
| 26 | MAA in mailing refers to_____ | Mail Application Agent | Mail Access Agent | Mail Application Access | Mail Access Applicant | Mail Access Agent |
| 27 | Email address consists of ___ parts | 3 | 4 | 5 | 2 | 2 |
| 28 | ____part of email refers to the providers name | Domain name | Local | local agent | Provider Agent | Domain name |
| 29 | SMTP refers to ____ | Simple Mail Transfer Protocol | Simplex Mail Transfer Protocol | Single Mail Transfer Protocol | Switched Mail Transfer Protocol | Simple Mail Transfer Protocol |
| 30 | ____ is a command used by the client in SMTP to check the status of the recipient. | TURN | NOOP | RSET | EXPN | NOOP |
| 31 | ____ is a SMTP command that lets the sender and the recipient switch positions, | TURN | NOOP | RSET | EXPN | TURN |
| 32 | ____ is the SMTP command that asks the recipient to send information about the command sent as the argument. | TURN | NOOP | HELP | EXPN | HELP |
| 33 | _____is the SMTP command that aborts the current mail transaction | TURN | NOOP | RSET | EXPN | RSET |

## UNIT-V

## SYLLABUS

**LAN administration:** Linux and TCP/IP networking: Network Management and Debugging.

### 1. TCP/IP (Transmission Control Protocol / Internet Protocol) Networking:

### 1.1 Introduction:

TCP/IP is the networking protocol suite most commonly used with Linux/UNIX, Mac OS, Windows, and most other operating systems. It is also the native language of the Internet. Devices that speak the TCP/IP protocol can exchange data (“interoperate”) despite their many differences. IP, the suite’s underlying delivery protocol, is the workhorse of the Internet.

TCP is a connection-oriented protocol that facilitates a conversation between two programs. The analogy of TCP/IP is a Telephone Call. TCP is a polite protocol that forces competing users to share bandwidth and generally behave in ways that are good for the productivity of the overall network.

As the Internet becomes more popular and more crowded, we need the traffic to be mostly TCP to avoid congestion and effectively share the available bandwidth. Today, TCP accounts for the vast majority of Internet traffic, with UDP and ICMP checking in at a distant second and third, respectively.

### 1.2 TCP & Internet

TCP/IP and the Internet share a history that goes back several decades. The technical success of the Internet is due largely to the elegant and flexible design of TCP/IP and to the fact that TCP/IP is an open and nonproprietary protocol suite.

How the Internet is managed today? The Internet is the driving force in the world economy, several sectors worry that it seems to be in the hands of a bunch of computer geeks, with perhaps a little direction from the U.S. government.

Several organizations are involved in management of Internet:

• ICANN, the Internet Corporation for Assigned Names and Numbers: if anyone can be said to be in charge of the Internet, this group is it. (www.icann.org)

• ISOC, the Internet Society: ISOC is a membership organization that represents Internet users. (www.isoc.org)

• IETF, the Internet Engineering Task Force: this group oversees the development and standardization of the technical aspects of the Internet. It is an open forum in which anyone can participate. (www.ietf.org) of these groups, ICANN has the toughest job: establishing itself as the authority in charge of the Internet, undoing the mistakes of the past, and foreseeing the future.

### 1.3 Network Standard & Documentation

The technical activities of the Internet community are summarized in documents known as RFCs; an RFC is a Request for Comments. Protocol standards, proposed changes, and informational bulletins all usually end up as RFCs. RFCs are numbered sequentially; currently, there are about 4,000. RFCs also have descriptive titles (e.g., Algorithms for Synchronizing Network Clocks), but to forestall ambiguity they are usually cited by number.

Not all RFCs are dry and full of boring technical details. Some of our favorites on the lighter side (often written on April 1st) are RFCs 1118, 1149, 1925, 2324, and 2795: • RFC1118 – The Hitchhiker's Guide to the Internet • RFC1149 – A Standard for the Transmission of IP Datagrams on Avian Carriers1 • RFC1925 – The Twelve Networking Truths • RFC2324 – Hyper Text Coffee Pot Control Protocol (HTCPCP/1.0) • RFC2795 – The Infinite Monkey Protocol Suite (IMPS)

### 1.4 Networking Road Map

TCP/IP is a "protocol suite," a set of network protocols designed to work smoothly together. It includes several components, each defined by a standards-track RFC or series of RFCs:

- IP, the Internet Protocol, which routes data packets from one machine to another (RFC791)
- ICMP, the Internet Control Message Protocol, which provides several kinds of low-level support for IP, including error messages, routing assistance, and debugging help (RFC792)
- ARP, the Address Resolution Protocol, which translates IP addresses to hardware addresses (RFC823)2
- UDP, the User Datagram Protocol, and TCP, the Transmission Control Protocol, which deliver data to specific applications on the destination machine. UDP provides unverified, "best effort" transport for individual messages, whereas TCP guarantees a reliable, full duplex, flow-controlled, error-corrected conversation between processes on two hosts. (RFCs 768 and 793)

TCP/IP is designed to work around the five layers namely Application Layer, Transport Layer, Network Layer, Data Link Layer and Physical Layer. After TCP/IP had been implemented and deployed, the International Organization for Standardization came up with its own seven-layer protocol suite called OSI where Presentation and Session layers where added.

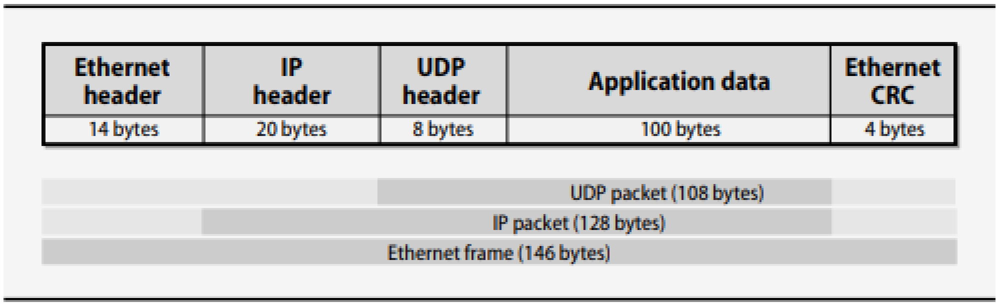
### 1.5 Packets and Encapsulation

Data travels on a network in the form of packets, bursts of data with a maximum length imposed by the link layer. Each packet consists of a header and a payload. The header tells where the packet came from and where it's going. It can also include checksums, protocol-specific information, or other handling instructions. The payload is the data to be transferred.

The name of the primitive data unit depends on the layer of the protocol. At the link layer it is called a frame, at the IP layer a packet, and at the TCP layer a segment. Here, we use "packet" as a generic term that encompasses all these cases.

As a packet travels down the protocol stack (from TCP or UDP transport to IP to Ethernet to the physical wire) in preparation for being sent, each protocol adds its own header information. Each protocol's finished packet becomes the payload part of the packet generated by the next protocol. This nesting is known as encapsulation. On the receiving machine, the encapsulation is reversed as the packet travels back up the protocol stack.

A typical network packet



**1.5.1 The link layer**:

The gap between the lowest layers of the networking software and the network hardware itself is bridged are

***Ethernet framing standards***: One of the main chores of the link layer is to add headers to packets and to put separators between them. The headers contain the packets' link-layer addressing information and checksums, and the separators ensure that receivers can tell where one packet stops and the next one begin. The process of adding these extra bits is known generically as framing. The framing that a machine uses is determined both by its interface card and by the interface card's driver.

***Ethernet cabling and signaling standards***: The cabling options for the various Ethernet speeds (10 Mb/s, 100 Mb/s, 1 Gb/s, and now 10 Gb/s) are usually specified as part of the IEEE's standardization efforts. Often, a single type of cable with short distance limits will be approved as a new technology emerges

***Wireless networking***: The IEEE 802.11 standard attempts to define framing and signaling standards for wireless links. One interoperability issue you may need to pay attention to is that of "translation" vs. "encapsulation."

Translation converts a packet from one format to another; Encapsulation wraps the packet with the desired format.

***Maximum transfer unit:*** The size of packets on a network may be limited both by hardware specifications and by protocol conventions. For example, the payload of a standard Ethernet frame can be no longer than 1,500 bytes. The size limit is associated with the link-layer protocol and is called the maximum transfer unit or MTU. The TCP protocol can determine the smallest MTU along the path to the destination and use that size from the outset. In the TCP/IP suite, the IP layer splits packets to conform to the MTU of a particular network link.

If a packet is routed through several networks, one of the intermediate networks may have a smaller MTU than the network of origin. In this case, the router that forwards the packet onto the small-MTU network further subdivides the packet in a process called fragmentation.

***Packet addressing:*** Like letters or email messages, network packets must be properly addressed in order to reach their destinations. Several addressing schemes are used in combination:

- MAC (medium access control) addresses for hardware
- IP addresses for software
- Hostnames for people

A host's network interface usually has a link-layer MAC address that distinguishes it from other machines on the physical network, an IP address that identifies it on the global Internet, and a hostname that's used by humans. A 6-byte Ethernet address is divided into two parts: the first three bytes identify the manufacturer of the hardware, and the last three bytes are a unique serial number that the manufacturer assigns.

Sysadmins can often identify at least the brand of machine that is trashing the network by looking up the 3-byte identifier in a table of vendor IDs. A current vendor table is available from www.iana.org/assignments/ethernet-numbers. The mapping between IP addresses and hardware addresses is implemented at the link layer of the TCP/IP model.

***Ports: TCP*** and UDP extend IP addresses with a concept known as a "port." A port is 16-bit number that supplements an IP address to specify a particular communication channel. Standard services such as email, FTP, and the web all associate themselves with "well known" ports defined in /etc/services.

***Address types***: Both the IP layer and the link layer define several different types of addresses: • Unicast – addresses that refer to a single host (network interface, really) • Multicast – addresses that identify a group of hosts • Broadcast – addresses that include all hosts on the local network

### 1.6 IP Addressing

The success of TCP/IP as the network protocol of the Internet is largely because of its ability to connect together networks of different sizes and systems of different types. These networks are arbitrarily defined into three main classes (along with a few others) that have predefined sizes, each of which can be divided into smaller sub-networks by system administrators.

A subnet mask is used to divide an IP address into two parts. One part identifies the host (computer), the other part identifies the network to which it belongs. To better understand how IP addresses and subnet masks work, look at an IP (Internet Protocol) address and see how it is organized.

**IP addresses: Networks and hosts**

An IP address is a 32-bit number that uniquely identifies a host (computer or other device, such as a printer or router) on a TCP/IP network. IP addresses are normally expressed in dotted-decimal format, with four numbers separated by periods, such as 192.168.123.132. To understand how subnet masks are used to distinguish between hosts, networks, and sub- networks, examine an IP address in binary notation.

For example, the dotted-decimal IP address 192.168.123.132 is (in binary notation) the 32 bit number 110000000101000111101110000100. This number may be hard to make sense of, so divide it into four parts of eight binary digits. These eight bit sections are known as octets. The example IP address, then, becomes 11000000.10101000.01111011.10000100. This number only makes a little more sense, so for most uses, convert the binary address into dotted-decimal format (192.168.123.132). The decimal numbers separated by periods are the octets converted from binary to decimal notation.

For a TCP/IP wide area network (WAN) to work efficiently as a collection of networks, the routers that pass packets of data between networks do not know the exact location of a host for which a packet of information is destined. Routers only know what network the host is a member of and use information stored in their route table to determine how to get the packet to the destination host's network. After the packet is delivered to the destination's network, the packet is delivered to the appropriate host.

For this process to work, an IP address has two parts. The first part of an IP address is used as a network address, the last part as a host address. If you take the example 192.168.123.132 and divide it into these two parts you get the following:

192.168.123. Network .132 Host

-or-

192.168.123.0 - network address. 0.0.0.132 - host address.

**Subnet mask**

The second item, which is required for TCP/IP to work, is the subnet mask. The subnet mask is used by the TCP/IP protocol to determine whether a host is on the local subnet or on a remote network. In TCP/IP, the parts of the IP address that are used as the network and host addresses are not fixed, so the network and host addresses above cannot be determined unless you have more information. This information is supplied in another 32-bit number called a subnet mask.

In this example, the subnet mask is 255.255.255.0. It is not obvious what this number means unless you know that 255 in binary notation equals 11111111; so, the subnet mask is:

11111111.11111111.11111111.0000000

Lining up the IP address and the subnet mask together, the network and host portions of the address can be separated:

11000000.10101000.01111011.10000100 -- IP address (192.168.123.132)

11111111.11111111.11111111.00000000 -- Subnet mask (255.255.255.0)

The first 24 bits (the number of ones in the subnet mask) are identified as the network address, with the last 8 bits (the number of remaining zeros in the subnet mask) identified as the host address. This gives you the following:

11000000.10101000.01111011.00000000 -- Network address (192.168.123.0)

00000000.00000000.00000000.10000100 -- Host address (000.000.000.132)

So now you know, for this example using a 255.255.255.0 subnet mask, that the network ID is 192.168.123.0, and the host address is 0.0.0.132. When a packet arrives on the 192.168.123.0 subnet (from the local subnet or a remote network), and it has a destination address of 192.168.123.132, your computer will receive it from the network and process it.

Almost all decimal subnet masks convert to binary numbers that are all ones on the left and all zeros on the right. Some other common subnet masks are:

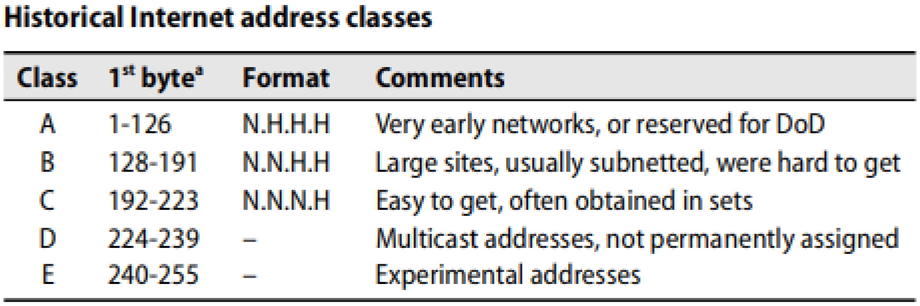
| Decimal | Binary |
|---|---|
| 255.255.255.192 | 1111111.11111111.1111111.11000000 |
| 255.255.255.224 | 1111111.11111111.1111111.11100000 |

**Network classes**

Internet addresses are allocated by the InterNIC (http://www.internic.net), the organization that administers the Internet. These IP addresses are divided into classes. The most common of these are classes A, B, and C. Classes D and E exist, but are not generally used by end users. Each of the address classes has a different default subnet mask. You can identify the class of an IP address by looking at its first octet. Following are the ranges of Class A, B, and C Internet addresses, each with an example address:

- Class A networks use a default subnet mask of 255.0.0.0 and have 0-127 as their first octet. The address 10.52.36.11 is a class A address. Its first octet is 10, which is between 1 and 126, inclusive.
- Class B networks use a default subnet mask of 255.255.0.0 and have 128-191 as their first octet. The address 172.16.52.63 is a class B address. Its first octet is 172, which is between 128 and 191, inclusive.
- Class C networks use a default subnet mask of 255.255.255.0 and have 192-223 as their first octet. The address 192.168.123.132 is a class C address. Its first octet is 192, which is between 192 and 223, inclusive.

In some scenarios, the default subnet mask values do not fit the needs of the organization, because of the physical topology of the network, or because the numbers of networks (or hosts) do not fit within the default subnet mask restrictions. The next section explains how networks can be divided using subnet masks.



**Historical Internet address classes**

| Class | 1st byte[a] | Format | Comments |
|---|---|---|---|
| A | 1-126 | N.H.H.H | Very early networks, or reserved for DoD |
| B | 128-191 | N.N.H.H | Large sites, usually subnetted, were hard to get |
| C | 192-223 | N.N.N.H | Easy to get, often obtained in sets |
| D | 224-239 | – | Multicast addresses, not permanently assigned |
| E | 240-255 | – | Experimental addresses |

**Subnetting:**

A Class A, B, or C TCP/IP network can be further divided, or subnetted, by a system administrator. This becomes necessary as you reconcile the logical address scheme of the Internet (the abstract world of IP addresses and subnets) with the physical networks in use by the real world.

A system administrator who is allocated a block of IP addresses may be administering networks that are not organized in a way that easily fits these addresses. For example, you have a wide area network with 150 hosts on three networks (in different cities) that are connected by a TCP/IP router. Each of these three networks has 50 hosts. You are allocated the class C network 192.168.123.0. (For illustration, this address is actually from a range that is not allocated on the Internet.) This means that you can use the addresses 192.168.123.1 to 192.168.123.254 for your 150 hosts.

Two addresses that cannot be used in your example are 192.168.123.0 and 192.168.123.255 because binary addresses with a host portion of all ones and all zeros are invalid. The zero address is invalid because it is used to specify a network without specifying a host. The 255 address (in binary notation, a host address of all ones) is used to broadcast a message to every host on a network. Just remember that the first and last address in any network or subnet cannot be assigned to any individual host.

You should now be able to give IP addresses to 254 hosts. This works fine if all 150 computers are on a single network. However, your 150 computers are on three separate physical networks. Instead of requesting more address blocks for each network, you divide your network into subnets that enable you to use one block of addresses on multiple physical networks.

In this case, you divide your network into four subnets by using a subnet mask that makes the network address larger and the possible range of host addresses smaller. In other words, you are 'borrowing' some of the bits usually used for the host address, and using them for the network portion of the address. The subnet mask 255.255.255.192 gives you four networks of 62 hosts each. This works because in binary notation, 255.255.255.192 is the same as 1111111.11111111.1111111.11000000. The first two digits of the last octet become network addresses, so you get the additional networks 00000000 (0), 01000000 (64), 10000000 (128) and 11000000 (192). (Some administrators will only use two of the subnetworks using 255.255.255.192 as a subnet mask. For more information on this topic, see RFC 1878.) In these four networks, the last 6 binary digits can be used for host addresses.

Using a subnet mask of 255.255.255.192, your 192.168.123.0 network then becomes the four networks 192.168.123.0, 192.168.123.64, 192.168.123.128 and 192.168.123.192. These four networks would have as valid host addresses:

192.168.123.1-62
192.168.123.65-126
192.168.123.129-190
192.168.123.193-254

Remember, again, that binary host addresses with all ones or all zeros are invalid, so you cannot use addresses with the last octet of 0, 63, 64, 127, 128, 191, 192, or 255.
You can see how this works by looking at two host addresses, 192.168.123.71 and 192.168.123.133. If you used the default Class C subnet mask of 255.255.255.0, both addresses are on the 192.168.123.0 network. However, if you use the subnet mask of 255.255.255.192, they are on different networks; 192.168.123.71 is on the 192.168.123.64 network, 192.168.123.133 is on the 192.168.123.128 network.

**Default gateways**

If a TCP/IP computer needs to communicate with a host on another network, it will usually communicate through a device called a router. In TCP/IP terms, a router that is specified on a host, which links the host's subnet to other networks, is called a default gateway. This section explains how TCP/IP determines whether or not to send packets to its default gateway to reach another computer or device on the network. When a host attempts to communicate with another device using TCP/IP, it performs a comparison process using the defined subnet mask and the destination IP address versus the subnet mask and its own IP address. The result of this comparison tells the computer whether the destination is a local host or a remote host.

If the result of this process determines the destination to be a local host, then the computer will simply send the packet on the local subnet. If the result of the comparison determines the destination to be a remote host, then the computer will forward the packet to the default gateway defined in its TCP/IP properties. It is then the responsibility of the router to forward the packet to the correct subnet.

### Reserved Private Ranges

There are also some portions of the IPv4 space that are reserved for specific uses. One of the most useful reserved ranges is the loopback range specified by addresses from 127.0.0.0 to 127.255.255.255. This range is used by each host to test networking to itself. Typically, this is expressed by the first address in this range: 127.0.0.1. Each of the normal classes also have a range within them that is used to designate private network addresses. For instance, for class A addresses, the addresses from 10.0.0.0 to 10.255.255.255 are reserved for private network assignment. For class B, this range is 172.16.0.0 to 172.31.255.255. For class C, the range of 192.168.0.0 to 192.168.255.255 is reserved for private usage.

Any computer that is not hooked up to the internet directly (any computer that goes through a router or other NAT system) can use these addresses at will.

### *Net-masks and Subnets*

The process of dividing a network into smaller network sections is called **sub-netting**. This can be useful for many different purposes and helps isolate groups of hosts together and deal with them easily.

As we discussed above, each address space is divided into a network portion and a host portion. The amount the address that each of these take up is dependent on the class that the address belongs to. For instance, for class C addresses, the first 3 octets are used to describe the network. For the address 192.168.0.15, the 192.168.0 portion describes the network and the 15 describes the host.

By default, each network has only one subnet, which contains the entire host addresses defined within. A net-mask is basically a specification of the amount of address bits that are used for the network portion. A subnet mask is another net-mask within used to further divide the network.

Each bit of the address that is considered significant for describing the network should be represented as a "1" in the netmask.

For instance, the address we discussed above, 192.168.0.15 can be expressed like this, in binary:

1100 0000 - 1010 1000 - 0000 0000 - 0000 1111

As we described above, the network portion for class C addresses is the first 3 octets, or the first 24 bits. Since these are the significant bits that we want to preserve, the net-mask would be:

1111 1111 - 1111 1111 - 1111 1111 - 0000 0000

This can be written in the normal IPv4 format as 255.255.255.0. Any bit that is a "0" in the binary representation of the netmask is considered part of the host portion of the address and can be variable. The bits that are "1" are static, however, for the network or sub-network that is being discussed.

We determine the network portion of the address by applying a bitwise AND operation to between the address and the net-mask. A bitwise AND operation will basically save the networking portion of the address and discard the host portion. The result of this on our above example that represents our network is:

1100 0000 - 1010 1000 - 0000 0000 - 0000 0000

This can be expressed as 192.168.0.0. The host specification is then the difference between these original value and the host portion. In our case, the host is "0000 1111" or 15.

The idea of sub-netting is to take a portion of the host space of an address, and use it as an additional networking specification to divide the address space again.

For instance, a net-mask of 255.255.255.0 as we saw above leaves us with 254 hosts in the network (you cannot end in 0 or 255 because these are reserved). If we wanted to divide this into two subnetworks, we could use one bit of the conventional host portion of the address as the subnet mask.

So, continuing with our example, the networking portion is:

1100 0000 - 1010 1000 - 0000 0000

The host portion is:

0000 1111

We can use the first bit of our host to designate a sub-network. We can do this by adjusting the subnet mask from this:

1111 1111 - 1111 1111 - 1111 1111 - 0000 0000

To this:

1111 1111 - 1111 1111 - 1111 1111 - 1000 0000

In traditional IPv4 notation, this would be expressed as 192.168.0.128. What we have done here is to designate the first bit of the last octet as significant in addressing the network. This effectively produces two sub-networks. The first sub-network is from 192.168.0.1 to 192.168.0.127. The second sub-network contains the hosts 192.168.0.129 to 192.168.0.255. Traditionally, the subnet itself must not be used as an address.

If we use more bits out of the host space for networking, we can get more and more sub-networks.

***CIDR Notation***

A system called **Classless Inter-Domain Routing**, or CIDR, was developed as an alternative to traditional sub-netting. The idea is that you can add a specification in the IP address itself as to the number of significant bits that make up the routing or networking portion.

For example, we could express the idea that the IP address 192.168.0.15 is associated with the net-mask 255.255.255.0 by using the CIDR notation of 192.168.0.15/24. This means that the first 24 bits of the IP address given are considered significant for the network routing.

This allows us some interesting possibilities. We can use these to reference "super-nets". In this case, we mean a more inclusive address range that is not possible with a traditional subnet mask. For instance, in a class C network, like above, we could not combine the addresses from the networks 192.168.0.0 and 192.168.1.0 because the net-mask for class C addresses is 255.255.255.0.

However, using CIDR notation, we can combine these blocks by referencing this chunk as 192.168.0.0/23. This specifies that there are 23 bits used for the network portion that we are referring to.

So the first network (192.168.0.0) could be represented like this in binary:

```
1100 0000 - 1010 1000 - 0000 0000 - 0000 0000
```

While the second network (192.168.1.0) would be like this:

```
1100 0000 - 1010 1000 - 0000 0001 - 0000 0000
```

The CIDR address we specified indicates that the first 23 bits are used for the network block we are referencing. This is equivalent to a net-mask of 255.255.254.0, or:

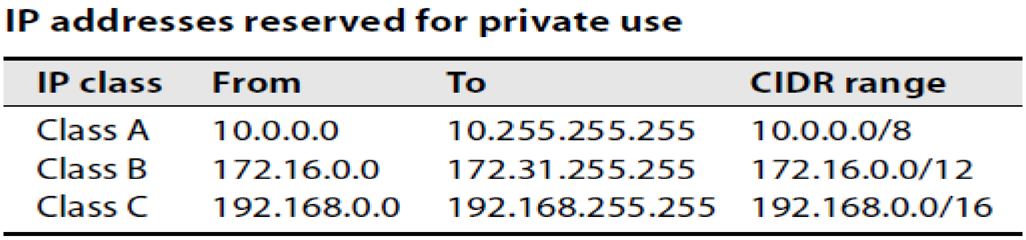
```
1111 1111 - 1111 1111 - 1111 1110 - 0000 0000
```

As you can see, with this block the 24th bit can be either 0 or 1 and it will still match, because the network block only cares about the first 23 digits.

Basically, CIDR allows us more control over addressing continuous blocks of IP addresses. This is much more useful than the sub-netting we talked about originally.

**Private addresses and NAT**

Another temporary solution to address space depletion is the use of private IP address spaces, described in RFC1918 (February 1996). In the CIDR era, sites normally obtain their IP addresses from their Internet service provider. If a site wants to change ISPs, it may be held for ransom by the cost of renumbering its networks. One alternative to using ISP-assigned addresses is to use private addresses that are never shown to your ISP. RFC1918 sets aside 1 class A network, 16 class B networks, and 256 class C networks that will never be globally allocated and can be used internally by any site. The catch is that packets bearing those addresses must never be allowed to sneak out onto the Internet.



IP addresses reserved for private use

| IP class | From | To | CIDR range |
|---|---|---|---|
| Class A | 10.0.0.0 | 10.255.255.255 | 10.0.0.0/8 |
| Class B | 172.16.0.0 | 172.31.255.255 | 172.16.0.0/12 |
| Class C | 192.168.0.0 | 192.168.255.255 | 192.168.0.0/16 |

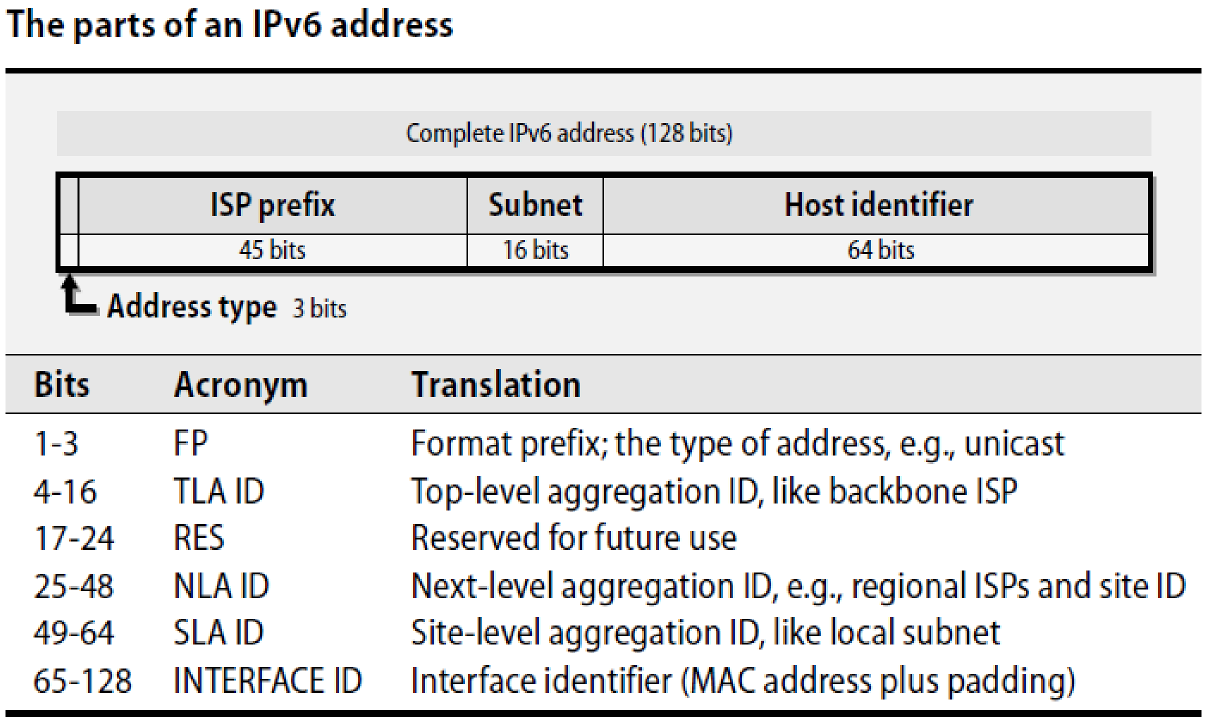
To allow hosts that use these private addresses to talk to the Internet, the site's border router runs a system called NAT (Network Address Translation). NAT intercepts packets addressed with these internal-only addresses and rewrites their source addresses, using a real external IP address and perhaps a different source port number. It also maintains a table of the mappings it has made between internal and external address/source-port pairs so that the translation can be performed in reverse when answering packets arrive from the Internet. NAT's use of port number mapping allows several conversations to be multiplexed  onto the same IP address so that a single external address can be shared by many internal hosts. In some cases, a site can get by with only one "real" IP address. A site that uses NAT must still request address space from its ISP, but most of the addresses thus obtained are used for NAT mappings and are not assigned to individual hosts. If the site later wants to choose another ISP, only the border router and its NAT configuration need to change, not the configurations of the individual hosts.

An incorrect NAT configuration can let private-address-space packets escape onto the Internet. The packets will get to their destinations, but answering packets won't be able to get back. CAIDA, 13 an organization that measures everything in sight about the backbone networks, finds that 0.1% to 0.2% of the packets on the backbone have either private addresses or bad checksums. One disadvantage of NAT (or perhaps an advantage) is that an arbitrary host on the Internet cannot connect directly to your site's internal machines. Some implementations (e.g., Linux and Cisco PIX) let you configure "tunnels" that support direct connections for particular hosts.

**IPv6 addressing**

An IPv6 address is 128 bits long. These long addresses were originally intended to solve the problem of IP address exhaustion. Now that they're here, however, they are being exploited to help with issues of routing, mobility, and locality of reference. IP addresses have never been geographically clustered in the way that phone numbers or zip codes are. Now, with the proposed segmentation of the

IPv6 address space, they will at least cluster to ISPs. The boundary between the network portion and the host portion of an IPv6 address is fixed at /64; the boundary between public topology and a site's local topology is fixed at /48. Table 12.8 shows the various parts of an IPv6 address.



The parts of an IPv6 address

Complete IPv6 address (128 bits)
ISP prefix
45 bits
Subnet
16 bits
Host identifier
64 bits
Address type
3 bits

| Bits | Acronym | Translation |
|---|---|---|
| 1-3 | FP | Format prefix; the type of address, e.g., unicast |
| 4-16 | TLA ID | Top-level aggregation ID, like backbone ISP |
| 17-24 | RES | Reserved for future use |
| 25-48 | NLA ID | Next-level aggregation ID, e.g., regional ISPs and site ID |
| 49-64 | SLA ID | Site-level aggregation ID, like local subnet |
| 65-128 | INTERFACE ID | Interface identifier (MAC address plus padding) |

In IPv6, the MAC address is seen at the IP layer, a situation with both good and bad implications. The brand and model of interface card are encoded in the first half of the MAC address, so hackers with code for a particular architecture will be helped along. The visibility of this information has also worried some privacy advocates. The IPv6 folks have responded by pointing out that site are not actually required to use MAC addresses; they're free to use whatever they want for the host address.

ARIN generally allocates IPv6 space only to large ISPs or to local Internet registries that plan to dole out large chunks of address space in the near future. These organizations can then allocate subspaces to their downstream customers. Here are some useful sources of IPv6 information:

- www.ipv6tf.net – An IPv6 information portal
- www.ipv6.org – FAQs and technical information
- www.ipv6forum.com – marketing folks and IPv6 propaganda

**1.6 Routing:**

Routing is the process of directing a packet through the maze of networks that stand between its source and its destination. In the TCP/IP system, it is similar to asking for directions in an unfamiliar country.

Routing information is stored in a table in the kernel. Each table entry has several parameters, including a netmask for each listed network (once optional but now required if the default netmask is not correct). To route a packet to a particular address, the kernel picks the most specific of the matching routes (that is, the one with the longest netmask). If the kernel finds no relevant route and no default route, then it returns a "network unreachable" ICMP error to the sender.

The word "routing" is commonly used to mean two distinct things:

- Looking up a network address in the routing table to forward a packet toward its destination
- Building the routing table in the first place

### Routing Table

You can examine a machine's routing table with **netstat -r**. Use **netstat -rn** to avoid DNS lookups and to present all the information numerically. Routing tables can be configured statically, dynamically, or with a combination of the two approaches. A static route is one that you enter explicitly with the **route** command. Static routes should stay in the routing table as long as the system is up; they are often set up at boot time from one of the system start-up scripts. In a stable local network, static routing is an efficient solution. It is easy to manage and reliable. However, it requires that the system administrator know the topology of the network accurately at boot time and that the topology not change often. Most machines on a local area network have only one way to get out to the rest of the network, so the routing problem is easy. A default route added at boot time suffices to point toward the way out. For more complicated network topologies, dynamic routing is required. Dynamic routing is typically performed by a daemon process that maintains and modifies the routing table. Routing daemons on different hosts communicate to discover the topology of the network and to figure out how to reach distant destinations. Several routing daemons are available.

### 1.7 ARP (Address Resolution Protocol)

ARP, the Address Resolution Protocol, discovers the hardware address associated with a particular IP address. It can be used on any kind of network that supports broadcasting but is most commonly described in terms of Ethernet. If host A wants to send a packet to host B on the same Ethernet, it uses ARP to discover B's hardware address. If B is not on the same network as A, host A uses the routing system to determine the next-hop router along the route to B and then uses ARP to find that router's hardware address. Since ARP uses broadcast packets, which cannot cross networks, it can only be used to find the hardware addresses of machines directly connected to the sending host's local network. Every machine maintains a table in memory called the ARP cache, which contains the results of recent ARP queries. Under normal circumstances, many of the addresses a host needs are discovered soon after booting, so ARP does not account for a lot of network traffic.

The **arp** command examines and manipulates the kernel's ARP cache, adds or deletes entries, and flushes or shows the table. The command **arp -a** displays the contentsof the ARP cache. The **arp** command is generally useful only for debugging and for situations that involve special hardware. Some devices are not smart enough to speak ARP. To support such devices, you might need to configure another machine as a proxy ARP server for your crippled hardware. That's normally done with the **arp** command as well

**ADDITION OF A MACHINE TO A NETWORK**

Only a few steps are involved in adding a new machine to an existing local area network, but some vendors hide the files you must modify and generally make the chore difficult. Others provide a setup script that prompts for the networking parameters that are needed, which is fine until you need to undo something or move a machine. Before bringing up a new machine on a network that is connected to the Internet, you should secure it so that you are not inadvertently inviting hackers onto your local network.

The basic steps to add a new machine to a local network are as follows:

- Assign a unique IP address and hostname.
- Set up the new host to configure its network interfaces at boot time.
- Set up a default route and perhaps fancier routing.
- Point to a DNS name server, to allow access to the rest of the Internet.

Of course, you could add a debugging step to this sequence as well. After any change that might affect booting, you should always reboot to verify that the machine comes up correctly. If your network uses DHCP, the Dynamic Host Configuration Protocol, the DHCP server will do these chores for you.

**ifconfig: configure network interfaces**

**ifconfig** enables or disables a network interface, sets its IP address and subnet mask, and sets various other options and parameters. It is usually run at boot time but it can also make changes on the fly. Be careful if you are making **ifconfig** changes and are logged in remotely; many a sysadmin has been locked out this way and had to drive in to fix things. An **ifconfig** command most commonly has the form

**ifconfig** *interface address options* …

for example:

```
ifconfig eth0 192.168.1.13 netmask 255.255.255.0 up
```

**ifconfig** *interface* displays the current settings for *interface* without changing them. Many systems understand **-a to** mean "all interfaces," and **ifconfig -a** can therefore be used to find out what interfaces are present on the system.

**route: configure static routes**

The **route** command defines static routes, explicit routing table entries that never change (you hope), even if you run a routing daemon. When you add a new machine to a local area network, you usually need to specify only a default route;

**Default routes**

A default route causes all packets whose destination network is not found in the kernel's routing table to be sent to the indicated gateway. To set a default route, simply add the following line to your startup files:

```
route add default gw gateway-IP-address
```

Rather than hard coding an explicit IP address into the startup files, most vendors have their systems get the gateway IP address from a configuration file. The way that local routing information is integrated into the startup sequence is unfortunately different for each of our Linux systems

**DNS configuration**

To configure a machine as a DNS client, you need to edit only one or two files: all systems require **/etc/resolv.conf** to be modified, and some require you to modify a "service switch" file as well. The **/etc/resolv.conf** file lists the DNS domains that should be searched to resolve names that are incomplete (that is, not fully qualified, such as anchor instead of anchor. cs.colorado.edu) and the IP addresses of the name servers to contact for name lookups.

**Security Issues**

**IP forwarding**

A UNIX or Linux system that has IP forwarding enabled can act as a router. That is, it can accept third-party packets on one network interface, match them to a gateway or destination host on another interface, and retransmit the packets. Unless your system has multiple network interfaces and is actually supposed to function as a router, it's advisable to turn this feature off. Hosts that forward packets can sometimes be coerced into compromising security by making external packets appear to have come from inside your network. This subterfuge can help an intruder's packets evade network scanners and packet filters. It is perfectly acceptable for a host to use multiple network interfaces for its own traffic without forwarding third-party traffic.

**ICMP redirects**

ICMP redirects can maliciously reroute traffic and tamper with your routing tables. Most operating systems listen to ICMP redirects and follow their instructions by default. It would be bad if all your traffic were rerouted to a competitor's network for a few hours, especially while backups were running. In such case configure your routers (and hosts acting as routers) to ignore and perhaps log ICMP redirect attempts.

**Source routing**: It was part of the original IP specification; it was intended primarily to facilitate testing. It can create security problems because packets are often filtered according to their origin. If someone can cleverly route a packet to make it appear to have originated within your network instead of the Internet, it might slip through your firewall. We recommend that you neither accept nor forward source-routed packets.

**Broadcast pings and other directed broadcasts**

Ping packets addressed to a network's broadcast address (instead of to a particular host address) are typically delivered to every host on the network. Such packets have been used in denial of service attacks; for example, the so-called Smurf attacks. (The "Smurf attacks" Wikipedia article has details.) Broadcast pings are a form of "directed broadcast" in that they are packets sent to the broadcast address of a distant network. The default handling of such packets has been gradually changing.

**IP spoofing**

The source address on an IP packet is normally filled in by the kernel's TCP/IP implementation and is the IP address of the host from which the packet was sent. However, if the software creating the packet uses a raw socket, it can fill in any source address it likes. This is called IP spoofing and is usually associated with some kind of malicious network behaviour. The machine identified by the spoofed source IP address (if it is a real address at all) is often the victim in the scheme. Error and return packets can disrupt or flood the victim's network connections. You should deny IP spoofing at your border router by blocking outgoing packets whose source address is not within your address space. This precaution is especially important if your site is a university where students like to experiment and may be tempted to carry out digital vendettas.

**Host-based firewalls**

Traditionally, a network packet filter or firewall connects your local network to the outside world and controls traffic according to a site-wide policy. The last few Windows releases all come with their own personal firewalls, and they complain bitterly if you try to turn the firewall off. Our example systems all include packet filtering software, but you should not infer from this that every UNIX or Linux machine needs its own firewall. It does not. The packet filtering features are there to allow these machines to serve as network gateways.

**Virtual private networks** Many organizations that have offices in several locations would like to have all those locations connected to one big private network. Such organizations can use the Internet as if it were a private network by establishing a series of secure, encrypted "tunnels" among their various locations. A network that includes such tunnels is known as a virtual private network or VPN. VPN facilities are also needed when employees must connect to your private network from their homes or from the field. A VPN system doesn't eliminate every possible security issue relating to such ad hoc connections, but it's secure enough for many purposes.

**PPP: The Point to Point Protocol**

PPP represents an underlying communication channel as a virtual network interface. However, since the underlying channel need not have any of the features of an actual network, communication is restricted to the two hosts at the ends of the link—a virtual network of two. PPP has the distinction of being used on both the slowest and the fastest IP links, but for different reasons. In its asynchronous form, PPP is best known as the protocol used to provide dialup Internet service over phone lines and serial links. These channels are not inherently packet oriented, so the PPP device driver encodes network packets into a unified data stream and adds link-level headers and markers to separate packets. In its synchronous form, PPP is the encapsulation protocol used on high-speed circuits that have routers at either end. It's also commonly used as part of the implementation of DSL and cable modems for broadband service. In these latter situations, PPP not only converts the underlying network system (often ATM in the case of DSL) to an IP-friendly form, but it also provides authentication and access control for the link itself. In addition to specifying how the link is established, maintained, and torn down, PPP implements error checking, authentication, encryption, and compression. These features make it adaptable to a variety of situations.

**2. Network Management & Debugging**

**2.1 Introduction**:

Network management is the art and science of keeping a network healthy. It generally includes the following tasks:

- Fault detection for networks, gateways, and critical servers
- Schemes for notifying an administrator of problems
- General monitoring, to balance load and plan expansion
- Documentation and visualization of the network
- Administration of network devices from a central site

As your network grows, management procedures should become more automated. On a network consisting of several different subnets joined with switches or routers, you may want to start automating management tasks with shell scripts and simple programs. If you have a WAN or a complex local network, consider installing a dedicated network management station.

### 2.2 Network Troubleshooting

Network issues can also stem from problems with higher-level protocols such as DNS, NFS, and HTTP. Troubleshooting requires strong commands like **ping**, **traceroute**, **netstat**, **tcpdump**, and Wireshark.

Before you attack your network, consider these principles:

- Make one change at a time, and test each change to make sure that it had the effect you intended. Back out any changes that have an undesired effect.
- Document the situation as it was before you got involved, and document every change you make along the way.
- Start at one "end" of a system or network and work through the system's critical components until you reach the problem. For example, you might start by looking at the network configuration on a client, work your way up to the physical connections, investigate the network hardware, and finally, check the server's physical connections and software configuration.
- Communicate regularly. Most network problems involve or affect lots of different people: users, ISPs, system administrators, telco engineers, network administrators, etc. Clear, consistent communication prevents you from hindering one another's efforts to solve the problem.
- Work as a team. Years of experience show that people make fewer stupid mistakes if they have a peer helping out.
- Use the layers of the network to negotiate the problem. Start at the "top" or "bottom" and work your way through the protocol stack.

**PING: Check to see if Host is Alive**

The **ping** command is embarrassingly simple, but in many situations it is all you need. It sends an ICMP ECHO_REQUEST packet to a target host and waits to see if the host answers back. Despite its simplicity, **ping** is one of the workhorses of network debugging. You can use **ping** to check the status of individual hosts and to test segments of the network. Routing tables, physical networks, and gateways are all involved in processing a ping, so the network must be more or less working for **ping** to succeed. If **ping** doesn't work, you can be pretty sure that nothing more sophisticated will work either.

**ping** runs in an infinite loop unless you supply a packet count argument. Once you've had your fill of pinging, type the interrupt character (usually <Control-C>) to get out. Here's an example:

```
$ ping beast
PING beast (10.1.1.46): 56 bytes of data.
        64 bytes from beast (10.1.1.46): icmp_seq=0 ttl=54 time=48.3ms
        64 bytes from beast (10.1.1.46): icmp_seq=1 ttl=54 time=46.4ms
```

```
64 bytes from beast (10.1.1.46): icmp_seq=2 ttl=54 time=88.7ms
^C
--- beast ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2026ms
rtt min/avg/max/mdev = 46.490/61.202/88.731/19.481 ms
```

The output for beast shows the host's IP address, the ICMP sequence number of each response packet, and the round trip travel time. The most obvious thing that the output above tells you is that the server beast is alive and connected to the network. On a healthy network, **ping** can allow you to determine if a host is down. Conversely, when a remote host is known to be up and in good working order, **ping** can give you useful information about the health of the network. Ping packets are routed by the usual IP mechanisms, and a successful round trip means that all networks and gateways lying between the source and destination are working correctly, at least to a first approximation.

To track down the cause of disappearing packets, first run **traceroute** to discover the route that packets are taking to the target host. Then ping the intermediate gateways in sequence to discover which link is dropping packets. To pin down the problem, you need to send a statistically significant number of packets.

The round trip time reported by **ping** gives you insight into the overall performance of a path through a network. Moderate variations in round trip time do not usually indicate problems. Packets may occasionally be delayed by tens or hundreds of milliseconds for no apparent reason; that's just the way IP works.

The **ping** program can send echo request packets of any size, so by using a packet larger than the MTU of the network (1,500 bytes for Ethernet), you can force fragmentation.

```
$ ping -s 1500 cuinfo.cornell.edu
```

Use the **ping** command with the following caveats in mind.

- First, it is hard to distinguish the failure of a network from the failure of a server with only the **ping** command. In an environment where ping tests normally work, a failed ping just tells you that *something* is wrong. (Network firewalls sometimes intentionally block ICMP packets.)
- Second, a successful ping does not guarantee much about the target machine's state. Echo request packets are handled within the IP protocol stack and do not require a server process to be running on the probed host. A response guarantees only that a machine is powered on and has not experienced a kernel panic.

**traceroute: Trace IP Packets:**

traceroute, originally written by Van Jacobson, uncovers the sequence of gateways through which an IP packet travels to reach its destination. All modern operating systems come with some version of traceroute. The syntax is simply

```
traceroute hostname
```

There are a variety of options, most of which are not important in daily use. As usual, the hostname can be specified with either a DNS name or an IP address. The output is simply a list of hosts, starting with the first gateway and ending at the destination. For example, a traceroute from the host jaguar to the host nubark produces the following output:

```
$ traceroute nubark
```

traceroute to nubark (192.168.2.10), 30 hops max, 38 byte packets 1 lab-gw (172.16.8.254) 0.840 ms 0.693 ms 0.671 ms 2 dmz-gw (192.168.1.254) 4.642 ms 4.582 ms 4.674 ms 3 nubark (192.168.2.10) 7.959 ms 5.949 ms 5.908 ms From this output we can tell that jaguar is exactly three hops away from nubark, and we can see which gateways are involved in the connection. The round trip time for each gateway is also shown—three samples for each hop are measured and displayed. A typical traceroute between Internet hosts often includes more than 15 hops. traceroute works by setting the time-to-live field (TTL, actually "hop count to live") of an outbound packet to an artificially low number. As packets arrive at a gateway, their TTL is decreased.

When a gateway decreases the TTL to 0, it discards the packet and sends an ICMP "time exceeded" message back to the originating host. The first three **traceroute** packets have their TTL set to 1. The first gateway to see such a packet (lab-gw in this case) determines that the TTL has been exceeded and notifies jaguar of the dropped packet by sending back an ICMP message. The sender's IP address in the header of the error packet identifies the gateway; **traceroute** looks up this address in DNS to find the gateway's hostname. To identify the second-hop gateway, **traceroute** sends out a second round of packets with TTL fields set to 2.

The first gateway routes the packets and decreases their TTL by 1. At the second gateway, the packets are then dropped and ICMP error messages are generated as before. This process continues until the TTL is equal to the number of hops to the destination host and the packets reach their destination successfully. Since **traceroute** sends three packets for each value of the TTL field, you may sometimes observe an interesting artifact. If an intervening gateway multiplexes traffic across several routes, the packets might be returned by different hosts; in this case, **traceroute** simply prints them all.

## NETSTAT: GET NETWORK STATISTICS

**netstat** collects a wealth of information about the state of your computer's networking software, including interface statistics, routing information, and connection tables. There isn't really a unifying theme to the different sets of output, except that they all relate to the network.

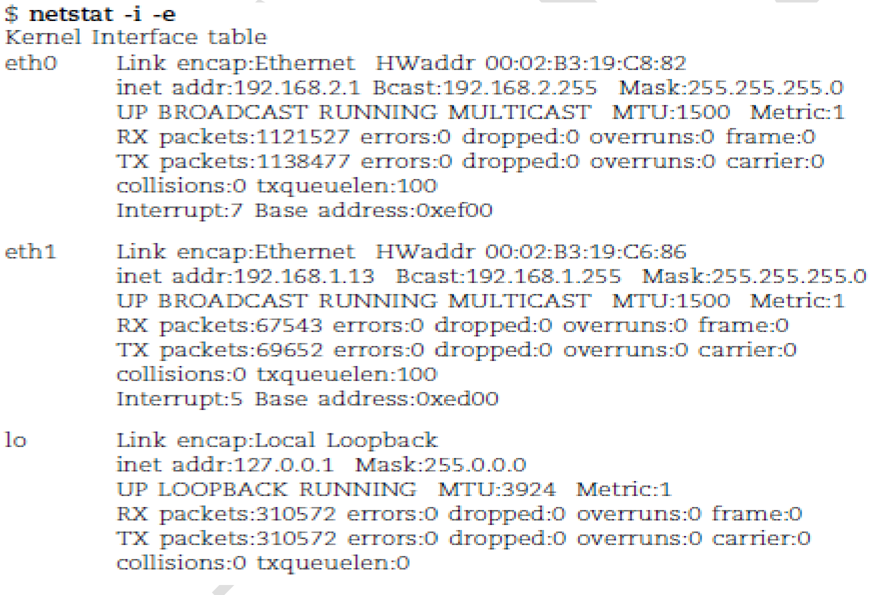
The five most common uses of **netstat are** :

- Inspecting interface configuration information
- Monitoring the status of network connections
- Identifying listening network services
- Examining the routing table
- Viewing operational statistics for various network protocols

***Inspecting interface configuration information***

**netstat -i** displays information about the configuration and state of each of the host's network interfaces. You can run **netstat -i** as a good way to familiarize yourself with a new machine's network setup. Add the **-e** option for additional details.

For example:

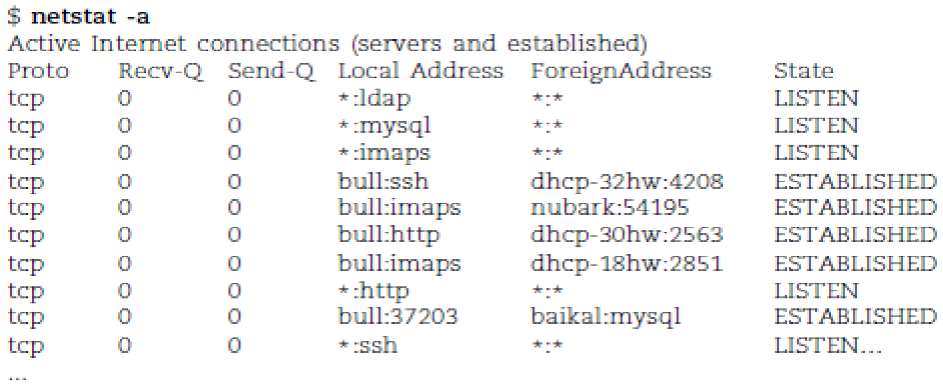


```
$ netstat -i -e
Kernel Interface table
eth0      Link encap:Ethernet  HWaddr 00:02:B3:19:C8:82
          inet addr:192.168.2.1 Bcast:192.168.2.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:1121527 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1138477 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:100
          Interrupt:7 Base address:0xef00

eth1      Link encap:Ethernet  HWaddr 00:02:B3:19:C6:86
          inet addr:192.168.1.13  Bcast:192.168.1.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:67543 errors:0 dropped:0 overruns:0 frame:0
          TX packets:69652 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:100
          Interrupt:5 Base address:0xed00

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:3924  Metric:1
          RX packets:310572 errors:0 dropped:0 overruns:0 frame:0
          TX packets:310572 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
```

This host has two network interfaces: one for regular traffic plus a second connection for system management named eth1. RX packets and TX packets report the number of packets that have been received and transmitted on each interface since the machine was booted. Many different types of errors are counted in the error buckets, and it is normal for a few to show up. Errors should be less than 1% of the associated packets. If your error rate is high, compare the rates of several neighboring machines. A large number of errors on a single machine suggest a problem with that machine's interface or connection. A high error rate everywhere most likely indicates a media or network problem. One of the most common causes of a high error rate is an Ethernet speed or duplex mismatch caused by a failure of autosensing or auto negotiation.

**Monitoring the status of network connections**

With no arguments, **netstat** displays the status of active TCP and UDP ports. Inactive ("listening") servers waiting for connections aren't normally shown; they can be seen with **netstat -a**. The output looks like this:



```
$ netstat -a
Active Internet connections (servers and established)
Proto   Recv-Q  Send-Q  Local Address   ForeignAddress      State
tcp     0       0       *:ldap          *:*                 LISTEN
tcp     0       0       *:mysql         *:*                 LISTEN
tcp     0       0       *:imaps         *:*                 LISTEN
tcp     0       0       bull:ssh        dhcp-32hw:4208      ESTABLISHED
tcp     0       0       bull:imaps      nubark:54195        ESTABLISHED
tcp     0       0       bull:http       dhcp-30hw:2563      ESTABLISHED
tcp     0       0       bull:imaps      dhcp-18hw:2851      ESTABLISHED
tcp     0       0       *:http          *:*                 LISTEN
tcp     0       0       bull:37203      baikal:mysql        ESTABLISHED
tcp     0       0       *:ssh           *:*                 LISTEN...
...
```

This example is from the host otter, and it has been severely pruned; for example, UDP and UNIX socket connections are not displayed. The output above shows an inbound SSH connection, two inbound IMAPS connections, one inbound HTTP connection, an outbound MySQL connection, and a bunch of ports listening for other connections.

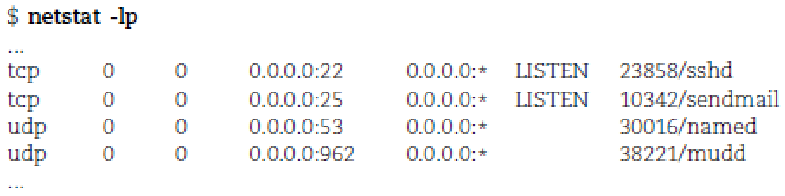
Addresses are shown as *hostname.service*, where the *service* is a port number. For well-known services, **netstat** shows the port symbolically, using the mapping defined in the **/etc/services** file. You can obtain numeric addresses and ports with the **-n** option. As with most network debugging tools, if your DNS is broken, **netstat** is painful to use without the **-n** flag.

Send-Q and Recv-Q show the sizes of the send and receive queues for the connection on the local host; the queue sizes on the other end of a TCP connection might be different. They should tend toward 0 and at least not be consistently nonzero. Of course, if you are running **netstat** over a network terminal, the send queue for your connection may never be 0.

**Identifying listening network services**

One common question in this security-conscious era is "What processes on this machine are listening on the network for incoming connections?" **netstat -a** shows all the ports that are actively listening (any TCP port in state LISTEN, and potentially any UDP port), but on a busy machine those lines can get lost in the noise of established TCP connections. Use **netstat -l** to see only the listening ports. The output format is the same as for **netstat -a**.

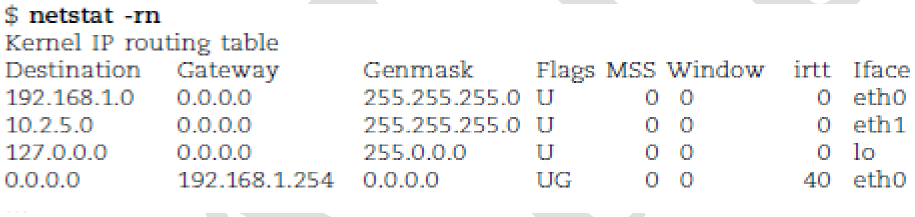
You can add the **-p** flag to make **netstat** identify the specific process associated with each listening port. The sample output below shows three common services (**sshd**, **sendmail**, and **named**), followed by an unusual one:



```
$ netstat -lp
...
tcp     0     0     0.0.0.0:22      0.0.0.0:*   LISTEN   23858/sshd
tcp     0     0     0.0.0.0:25      0.0.0.0:*   LISTEN   10342/sendmail
udp     0     0     0.0.0.0:53      0.0.0.0:*            30016/named
udp     0     0     0.0.0.0:962     0.0.0.0:*            38221/mudd
...
```

**Examining the routing table**

**netstat -r** displays the kernel's routing table. The following sample is from a Red Hat machine with two network interfaces.



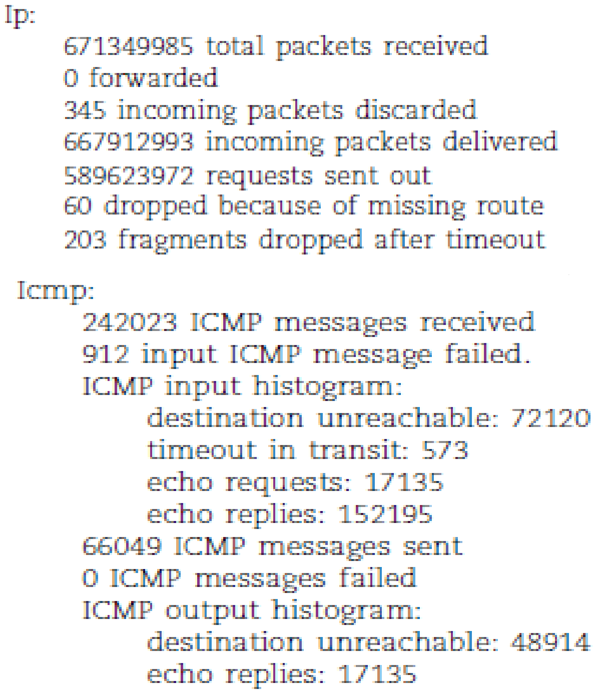
```
$ netstat -rn
Kernel IP routing table
Destination    Gateway        Genmask          Flags MSS Window   irtt  Iface
192.168.1.0    0.0.0.0        255.255.255.0    U       0  0          0  eth0
10.2.5.0       0.0.0.0        255.255.255.0    U       0  0          0  eth1
127.0.0.0      0.0.0.0        255.0.0.0        U       0  0          0  lo
0.0.0.0        192.168.1.254  0.0.0.0          UG      0  0         40  eth0
...
```

Destinations and gateways can be displayed either as hostnames or as IP addresses; the **-n** flag requests numeric output. The Flags characterize the route: U means up (active), G is a gateway, and H is a host route. U, G, and H together indicate a host route that passes through an intermediate gateway. The D flag (not shown) indicates a route resulting from an ICMP redirect.

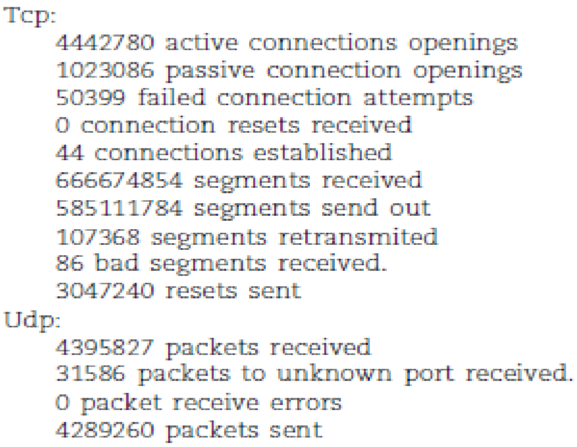
The remaining fields give statistics on the route: the current number of TCP connections using the route, the number of packets sent, and the interface used. Use this form of **netstat** to check the health of your system's routing table.

**Viewing operational statistics for network protocols**

**netstat -s** dumps the contents of counters that are scattered throughout the network code. The output has separate sections for IP, ICMP, TCP, and UDP. Below are pieces of **netstat -s** output from a typical server;



```
Ip:
     671349985 total packets received
     0 forwarded
     345 incoming packets discarded
     667912993 incoming packets delivered
     589623972 requests sent out
     60 dropped because of missing route
     203 fragments dropped after timeout

Icmp:
     242023 ICMP messages received
     912 input ICMP message failed.
     ICMP input histogram:
          destination unreachable: 72120
          timeout in transit: 573
          echo requests: 17135
          echo replies: 152195
     66049 ICMP messages sent
     0 ICMP messages failed
     ICMP output histogram:
          destination unreachable: 48914
          echo replies: 17135
```



```
Tcp:
     4442780 active connections openings
     1023086 passive connection openings
     50399 failed connection attempts
     0 connection resets received
     44 connections established
     666674854 segments received
     585111784 segments send out
     107368 segments retransmited
     86 bad segments received.
     3047240 resets sent
Udp:
     4395827 packets received
     31586 packets to unknown port received.
     0 packet receive errors
     4289260 packets sent
```

**Packet Sniffer:**

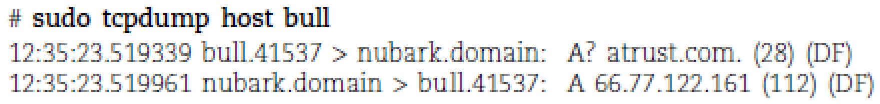
**tcpdump** and Wireshark belong to a class of tools known as packet sniffers. They listen to the traffic on a network and record or print packets that meet certain criteria specified by the user. For example, all packets sent to or from a particular host or TCP packets related to one particular network connection could be inspected.

Packet sniffers are useful both for solving problems you know about and for discovering entirely new problems. It's a good idea to take an occasional sniff of your network to make sure the traffic is in order. Packet sniffers need to be able to intercept traffic that the local machine would not normally receive (or at least, pay attention to), so the underlying network hardware must allow access to every packet.

Broadcast technologies such as Ethernet work fine, as do most other modern local area networks. Since packet sniffers need to see as much of the raw network traffic as possible, they can be thwarted by network switches, which by design try to limit the propagation of “unnecessary” packets. However, it can still be informative to try out a sniffer on a switched network. Packet sniffers understand many of the packet formats used by standard network services, and they can often print these packets in a human-readable form. This capability makes it easier to track the flow of a conversation between two programs. Some sniffers print the ASCII contents of a packet in addition to the packet header and so are useful for investigating high-layer protocols. Since some of these protocols send information (and even passwords) across the network as clear-text, you must take care not to invade the privacy of your users.

**tcpdump: king of sniffers**

**tcpdump**, yet another amazing network tool by Van Jacobson, is included in most Linux distributions. **tcpdump** has long been the industry-standard sniffer; most other network analysis tools read and write trace files in “**tcpdump** format.” By default, **tcpdump** tunes in on the first network interface it comes across. If it chooses the wrong interface, you can force an interface with the **-i** flag. If DNS is broken or you just don’t want **tcpdump** doing name lookups, use the **-n** option. For example, the following truncated output comes from the machine named nubark. The filter specification **host bull** limits the display of packets to those that directly involve the machine bull, either as source or as destination.



```
# sudo tcpdump host bull
12:35:23.519339 bull.41537 > nubark.domain:  A? atrust.com. (28) (DF)
12:35:23.519961 nubark.domain > bull.41537:  A 66.77.122.161 (112) (DF)
```

The first packet shows the host bull sending a DNS lookup request about atrust.com to nubark. The response is the IP address of the machine associated with that name, which is 66.77.122.161. Note the time stamp on the left and **tcpdump**’s understanding of the application-layer protocol (in this case, DNS). The port number on bull is arbitrary and is shown numerically (41537), but since the server port number (53) is well known, **tcpdump** shows its symbolic name (“domain”) instead.

**Wireshark: visual sniffer**

If you’re more inclined to use a point-and-click program for packet sniffing, then Wireshark may be for you. Available under the GNU General Public License from www.wireshark.org, Wireshark is a GTK+ (GIMP tool kit)-based GUI packet sniffer that has more functionality than most commercial sniffing products. You can run Wireshark on your Linux desktop, or if your laptop is still painfully suffering in the dark ages of Windows, you can download binaries for that too.

In addition to sniffing packets, Wireshark has a couple of features that make it extra handy. One nice feature is that Wireshark can read and write a large number of other packet trace file formats, including (but not limited to):

- TCPDUMP
- NAI's Sniffer
- Sniffer Pro
- NetXray
- Snoop
- Shomiti Surveyor
- Microsoft's Network Monitor
- Novell's LANalyzer
- Cisco Secure IDS iplog

The second extra-handy feature is that you can click on one packet in a TCP stream and ask Wireshark to "reassemble" (splice together) the payload data of all the packets in the stream. This feature is useful if you want to quickly examine the data transferred during a complete TCP conversation, such as a connection carrying an email message across the network.5 Wireshark has capture filters, which function identically to **tcpdump**'s. Watch out, though—one important gotcha with Wireshark is the added feature of "display filters," which affect what you see rather than what's actually captured by the sniffer. Oddly, display filters use an entirely different syntax from capture filters. Wireshark is an incredibly powerful analysis tool and is included in almost every networking expert's tool kit. Moreover, it's also an invaluable learning aid for those just beginning to explore packet networking. Wireshark's help menu provides many great examples to get you started.

**NETWORK MANAGEMENT PROTOCOLS**

Network management protocols standardize the method of probing a device to discover its configuration, health, and network connections. In addition, they allow some of this information to be modified so that network management can be standardized across different kinds of machinery and performed from a central location. The most common management protocol used with TCP/IP is the Simple Network Management Protocol, SNMP. Despite its name, SNMP is actually quite complex. It defines a hierarchical namespace of management data and a way to read and write the data at each node. It also defines a way for managed servers and devices ("agents") to send event notification messages (“traps”) to management stations. The SNMP protocol itself is simple; most of SNMP’s complexity lies above the protocol layer in the conventions for constructing the namespace and in the unnecessarily baroque vocabulary that surrounds SNMP like a protective shell.

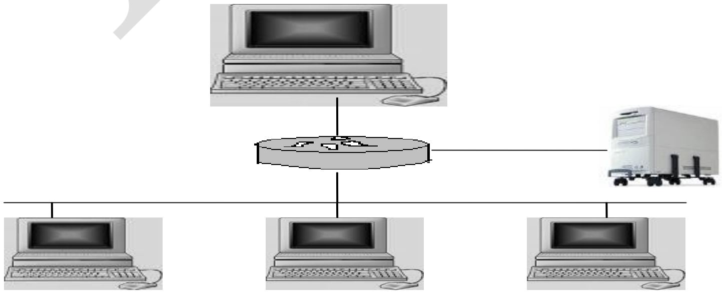
Several other standards are floating around out there. Many of them originate from the Distributed Management Task Force (DMTF), which is responsible for concepts such as WBEM (Web-Based Enterprise Management), DMI (Desktop Management Interface), and the CIM (Conceptual Interface Model). Some of these concepts, particularly DMI, have been embraced by several major vendors and may become a useful complement to (or even a replacement for) SNMP.

A major advantage of management-by-protocol is that it promotes all kinds of network hardware onto a level playing field. Linux systems are all basically similar, but routers, switches, and other low-level components are not. With SNMP, they all speak a common language and can be probed, reset, and configured from a central location. It’s nice to have one consistent interface to the entire network’s hardware.

### Simple Network Management Protocol

### Background

The Simple Network Management protocol (SNMP) is an application layer protocol that facilitates the exchange of the management information between network devices. It is part of the Transmission Control Protocol / Internet Protocol (TCP/IP) protocol suite. SNMP enables network administrators to manage network performance, find and solve network problems, and plan for network growth.Two versions of SNMP exist: SNMP version 1 (SNMPv1) and SNMP version 2 (SNMPv2). Both versions have a number of features in common. but SNMPv2 offers enhancements , such as additional protocol operations. Standardization of yet another version of SNMP - SNMP version 3 (SNMPv3) – is pending. This chapter provides descriptions of the SNMPv1 and SNMPv2 protocol operations. Figure 56-1 illustrates a basic network managed by SNMP.

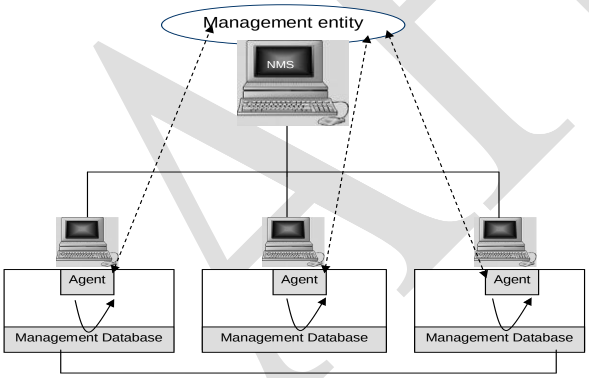


## SNMP Basic Components

An SNMP - managed network consists of three key components: managed devices, agents, and network – management systems (NMSs).

A managed device is a network node that contains an SNMP agent and that resides on a managed network. Managed devices collect and store management information and make this information available to NMSs using SNMP. Managed devices, sometimes called network elements, can be routers and access servers, switches and bridges, hubs, computer hosts, or printers.

An agent is a network management software module that resides in a managed device. An agent has local knowledge of management information and translates that information into a form compatible with SNMP. An NMS executes applications that monitor and control managed devices. NMSs  provide the bulk of the processing and memory resources required for network management. One or more NMSs must exist on any managed network.



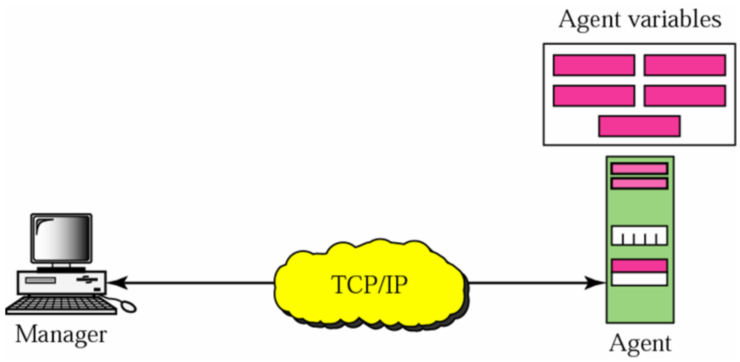
## SNMP Commands

Managed devices are monitored and controlled using four basic SNMP commands: read, write, trap, and traversal operations. The read command is used by an NMS to monitor managed devices. The NMS examines the different variables that are maintained by the managed devices. The write command is used by an NMS to control managed devices. The NMS changes the values of the variables stored within managed devices. The trap command is used by the managed devices to asynchronously report events to the NMS. When certain types of events occur, a managed device sends a trap to the NMS. Traversal operations are used by the NMS to determine which variables a managed device supports and to sequentially gather information in variable tables, such as a routing table.

**Network Management Architecture**

Network management system contains two primary elements. A manager and agents. The manager is the console through which the network administrator performs network management functions. Agents are the entities that interface to the actual device being managed. Bridges, hubs, routers or network servers are examples of managed devices that contain managed objects.

These managed objects might be hardware, configuration parameters, performance statistics, and so on, that directly relate to the current operation of the device in question. These objects are arranged in what is known as a virtual information database, called a management information base, also called MIB. SNMP allows managers and agents to communicate for the purpose of accessing these objects. The model of network management architecture looks like this:



Typical agent usually:

1 Implements full SNMP protocol.

2 Stores and retrieves management data as defined by the MIB.

3 Can asynchronously signal an event to the manager.

4 Can be a proxy for some non-SNMP manageable network node. Click here to see typical proxy architecture.

Atypical manager usually:

1 Implemented as a Network Management Station (the NMS)

2 Implements full SNMP protocol

3 Able to send Query

4 Get responses from agents

5 Set variables in agents

6 Acknowledge asynchronous events from agents

Some prominent vendors offer network management platforms which implement the role of the manager (listed in alphabetic order):

1 Dec PolyCenter Network Manager

2 Hewlett – Packard Open View

3 IBM AIX NetView/6000

4 SunConnect SunNet Manager

**Management Information Base**

Management Information Bases (MIBs) are a collection of definitions, which define the properties of the managed object within the device to be managed. Every managed device keeps a database of values for each of the definitions written in the MIB. It is not the actual database itself – it is the implementation dependent.

Definition of the MIB conforms to the SMI given in RFC 1155. Latest Internet MIB is given in RFC 1213 sometimes called the MIB-II. Click here to see MIB architecture. You can think of a MIB as an information warehouse.

Criteria and Philosophy for standardized MIB

1 Objects have to be uniquely named

2 Objects have to be essential

3 Abstract structure of the MIB needed to be universal

4 For the standard MIB maintain only a small number of objects

5 Allow for private extensions

6 Object must be general and not too device dependent

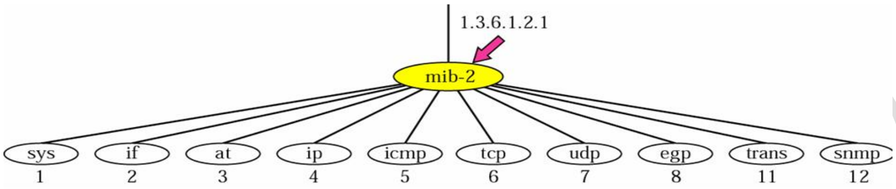
7 Objects cannot be easily derivable from their objects

**8** If agent is to be SNMP manageable then it is mandatory to implement the Internet MIB (currently given as MIB-II in RFC 1157)

**Naming an object**

1. Universal unambiguous identification of arbitrary objects
2. Can be achieved by using an hierarchical tree
3. Based on the Object Identification Scheme defined by OSI

**The Registered Tree**



**Identifiers**

1 Object name is given by its name in the tree.

2 All child nodes are given by the unique integer values within the new sub-tree.

3 Children can be parents of further child sub-tree (ie: they have subordinates) where the numbering scheme is recursively applied.

4 The Object Identifier (or name) of an object is the sequence of non-negative Integer values traversing the tree to the node required.

5 Allocation of an integer value for a node in the tree is an act of registration by whoever has delegated authority for that sub tree.

6 This process can go to an arbitrary depth.

7 If a node ha children then it is an aggregate node.

8 Children of the same parent cannot have the same integer value.

**Object and Object Identifiers**

1 Object is named or identified by the sequence of integers in traversing the tree to the object type required

2 This does not identify an instance of the object

3 The Object Identifier(OID) is shown in afew ways with a.b.c.d.e being the preferred

4 OIDs can name many types of objects:

### The Internet Sub – tree

- Directory sub-tree if for future directory services
- Experimental sub-tree is for experimental MIB work – still
- Has to be registered with the authority (IESG)
- MIB sub-tree is the actual mandatory Internet MIB for all
- Agents to implement (currently MIB-II RFC 1156- this is the Only sub-tree for management)
- Enterprise sub-tree (of private) are MIBs of proprietary objects And are of course not mandatory (sub-tree registered with Internet assigned numbers authority) for example: CISCO
- Router OID: 1.3.6.1.4.1.9.1.1
- SNMP management nearly always Internet in MIB and specific enterprises MIBs.

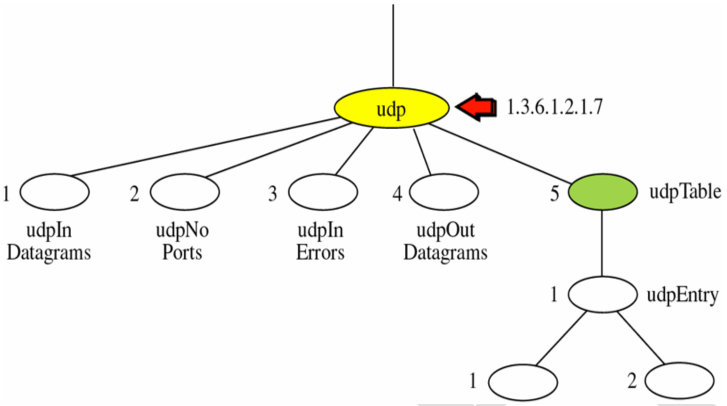
### MIB-II Standard Internet MIB

1. Definition follows structure given in SMI
2. MIB-II (RFC 1213) is current standard definition of the virtual file store for SNMP manageable objects
    - Has 10 basic groups
    - System
    - Interfaces
    - AT
    - IP
    - ICMP
    - TCP
    - UDP
    - EGP
    - Transmission
    - SNMP

If agent implements any group then is has to implement all of the managed objects within the group. An agent does not have to implement all groups. Note: MIB –I and MIB-II have some OID (position in the Internet sub-tree)

**MIB-II**

**The MIB sub-tree**



Note: there is an object cm OT (9) under the MIB but it has become almost superfluous and for all intense and purposes is not one of the SNMP manageable groups within MIB.

**SNMP Protocol**

SNMP is based on the managers/ agent model. SNMP is referred to as "simple" because the agent requires minimal software. Most of the processing power and the data storage reside on the management system, while a complementary subset of those functions resides in the managed system.

To achieve its goal of being simple, SNMP includes a limited set of management commands and responses. The management system issues Get, GetNext and Set messages to retrieve single or multiple object variables or to establish the value of a single variable. The managed agent sends a response message to complete the Get, GetNext or Set. The managed agents send an event notification, called a trap to the management system to identify the occurrence of conditions such as threshold that exceeds a predetermined value. In short there are only five primitive operations:

1 Get(retrieve operation)
2 Getnext( traversal operation)

3 Getresponse(indicative operation)

4 Set(alter operation)

5 Trap(asynchronous trap operation)

**SNMP Message Construct**

Each SNMP message has the format:

1 Version number

2 Community name – kind of a password

3 One or more SNMP PDUs – assuming trivial authentication

Each SNMP PDU except trap has the following format:

1 Request id – request sequence number

2 Error status – zero if no error otherwise one of a small set

3 Error index – if non zero indicates which of the OIDs in the PDU caused the error 2

4 List of OIDs and values - values are null for get and getnext

Trap PDUs have the following format:

1 Enterprise – identifies the type of object causing the trap

2 Agent address – IP address of agent which sent a the trap

3 Generic trap id – the common standard traps

4 Specific trap id – proprietary or enterprise trap

5 Time stamp – when trap occurred in time ticks

6 List of OIDs and values – OIDs that may be relevant to Send to the NMS

### Possible Questions

### 2 Mark Questions

1. List the different classes of Classful Addressing?
2. What is Masking?
3. What is the use of ARP?
4. What is the role of DHCP?
5. What is a Router?

### 6 Mark Questions

1. Discuss about Classful addressing.
2. Write a note on ping, tracetoute and netstat
3. Explain about Classless addressing with Masking.
4. What is ARP? Discuss about it in detail.
5. What is RARP? Explain about it.
6. Explain DHCP working Principle.
7. Discuss the Security issues of the Networking.
8. What is the use of SNMP? Discuss in detail about it.
9. Discuss about BOOTP in general.
10. Discuss about classless Addressing.

**Reg.No ---------------------**

**[**16ITU502B**]**

**KARPAGAM ACADEMY OF HIGHER EDUCATION**

(Deemed to be University)

(Established Under Section 3 of UGC Act 1956)

COIMBATORE – 641 021

**B.Sc. DEGREE EXAMINATION**

**Fifth Semester - I Internal Examination**

**Information Technology**

**Network Programming**

**Time: 2 Hrs** **Max.Marks: 50**

**Date :** **Class: III B.Sc (IT) - 'A' & 'B'**

---

**Part - A** **(10 X 1 = 20)**

1. TCP is a ______type of Protocol

   a) Connection Oriented b) Connection Less c) Reference d) intermediate

2. Process to Process Communication is performed in ______ layer.

   a) Network b) Data Link Layer c) Physical d) Transport

3. Which of the following Protocol is not the Transport layer Protocol?

   a) TCP b) UDP c) IP d) SCTP

4. The TCP Connection establishment is called as ______Way handshake

   a) Two b) Three c) Four d) One

5. The Packets of UDP is called as _______

   a) Datagram b) Segment c) Packet d) Header

6. The Packets of TCP is called as _______

   a) Datagram b) Segment c) Packet d) Header

7. Systems in network is called as ______

   a) Node b) Tree c) System d) All

8. _______ is a combination of IP-address and Port Number

   a) Server b) Client c) Server Socket d) Socket

9. ______ is the unique address that identifies a machine in a network.

   a) IP Address b) Port Number c) Socket d) All

10. ______ is used for process to process communication.

    a) IP Address b) Port Number c) Socket d) All

11. Port numbers are classified in to _____types.

    a) 4 b) 3 c) 2 d) 5

12. The well Known port numbers are used by _____

    a) Client b) Server c) Client & Server d) Routers

13. The System that request for Service is called _____

    a) Sever b) Client c) Router d) Hub

14. The System that Provide Service is called _____

a) Sever b) Client c) Router d) Hub

15. Dynamic ports are also called as _______port

a) Private b) Public c) Server d) Reserved

16. MTU refers to _______

a) Minimum Transfer Unit b) Maximum Transport Unit c) Maximum Transfer Unit d) Minimum Transport Unit

17. DF refers to _____

a) Do fragment b) Don’t Fragment c) Double Fragment d) None

18. The number of layers in TCP/IP reference model is ____

a) 7 b) 5 c) 4 d) 6

19. The number of layers in OSI reference model is ____

a) 7 b) 5 c) 4 d) 6

20. The header size of TCP packet is ______ bytes.

a) 20 b) 21 c) 25 d) 18

**Part - B** **(3 X 2 = 6)**

21. What is a Network?

22. What is connection oriented System?

23. What is a Port Number?

**Part - C** **(3 X 8 = 24)**

24. a) What is reference model? Explain about OSI reference Model.

(or)

b) Discuss in detail about UDP.

25. a) What is Connection Oriented Service? Discuss about TCP in detail.

(or)

b) Differentiate Connection Oriented and Connection Less protocols.

26. a) What is a socket? Discuss in general about the socket.

(or)

b) Write a simple java socket program which transfer data between client and Server.

**KARPAGAM ACADEMY OF HIGHER EDUCATION**
**(Deemed University Established Under Section 3 of UGC Act, 1956)**
**Coimbatore – 641021, INDIA**
**Department of Computer Science, Applications & Information Technology**

| | **Unit - V** | | | | | |
|---|---|---|---|---|---|---|
| **S.No** | **Questions** | **Opt1** | **Opt2** | **Opt3** | **Opt4** | **Answer** |
| 1 | The progenitor of the modern Internet was a network called _____ | ARPANET | CNET | OCTNET | TELNET | ARPANET |
| 2 | ARPANET was Established in ______ | 1972 | 1973 | 1969 | 1968 | 1969 |
| 3 | ISP Refers to ____ | Internet Scheme Providers | Internet Service Provider | Intranet Service Providers | Intranet Scheme Providers | Internet Service Provider |
| 4 | DNS refers to _____ | Domain Number Service | Domain Name Service | Desktop Number Service | Desktop Name Service | Domain Name Service |
| 5 | RFC stands for ______ | Request for Comment | Request for Connection | Respect to Comment | Regulation for Connection | Request for Comment |
| 6 | TCP/IP Reference Model consists of ____ layers | 5 | 6 | 8 | 7 | 5 |
| 7 | ICMP lies in ____layer of TCP/IP reference Model | Application | Network | DataLink | Transport | Network |
| 8 | CRC refers to _____ | Circular Redundancy Check | Cyclic Redundancy Check | Common Redundancy Check | Client Redundancy Check | Cyclic Redundancy Check |
| 9 | MTU refers to _____ | Minimum Transfer Unit | Maximum Transport Unit | Minimum Transport Unit | Maximum Transfer Unit | Maximum Transfer Unit |
| 10 | The MTU of Ethernet is _____bytes | 1500 | 2000 | 2500 | 3000 | 1500 |
| 11 | The MTU of FDDI is ____bytes | 1500 | 4000 | 4470 | 5000 | 4470 |
| 12 | The MTU of PPP modem link is _____ | 512 | 654 | 858 | 1024 | 512 |
| 13 | The MTU of P2P WAN link is _____ | 1500 | 600 | 5000 | 6000 | 1500 |
| 14 | _____ is a command used to display the IP configuration in UNIX | ipconfog | ifconfig | ipconfiguation | ifconfigurations | ifconfig |
| 15 | MAC refers to_____ | Medium Address Control | Maximum Address Control | Medium Access Control | Maximum Access Control | Medium Access Control |
| 16 | _____ is also called Physical Address | IP Address | Port Number | Process Number | MAC | MAC |
| 17 | ____ is also called Logical Number | IP Address | Port Number | Process Number | MAC | IP Address |
| 18 | Address which refers to Single host in destination Address is ____ | unicast | multicast | broadcast | doublecast | unicast |
| 19 | When Destination refers to group of host it is called ______ | unicast | multicast | broadcast | doublecast | multicast |
| 20 | When Destination refers to all the host in network it is called ______ | unicast | multicast | broadcast | doublecast | broadcast |
| 21 | IGMP refers to _____ | Internet Group Management Protocol | Intranet Group Management Protocol | Internet Group Message Protocol | Internet Group Message Protocol | Internet Group Management Protocol |
| 22 | Which of the following is not a Correct IP Address? | 125.16.25.1 | 172.16.8.200 | 172.16.25.1 | 172.16.256.10 | 172.16.256.10 |
| 23 | What is the size of the IPv4 IP Address? | 32 bits | 64 bits | 128 bits | 16 bits | 32 bits |
| 24 | IP Classful addressing consists of ____classes | 6 | 4 | 5 | 7 | 5 |
| 25 | Class A of Classful address can contain ____number of Host | 65025 | 255 | 255 X 255 | 1024 | 255 |
| 26 | Class B of Classful address can contain ____number of Host | 65025 | 255 | 255 X 25 | 1024 | 65025 |
| 27 | Which of the following is not a Correct IP Address? | 11111111 11111111 11111111 11111111 | 00000000 00000000 00000000 00000000 | 11110000 11110000 11110000 00001111 | 1111 1111 1111 1111 | 1111 1111 1111 1111 |
| 28 | If a network is sub divided it is called ______ | Supernetting | Subnetting | Masking | Polling | Subnetting |
| 29 | If networks are grouped to form a single network it is called _____ | Supernetting | Subnetting | Masking | Polling | Supernetting |
| 30 | What is the masking value of the given IP 125.16.23.56/12? | 126 | 16 | 23 | 12 | 12 |
| 31 | Which representation denotes class A? | N.N.N.N | N.N.H.H | N.H.H.H | N.N.N.H | N.N.N.H |
| 32 | What does N represent in the N.N.N.H ? | Network Address | Host Address | Next Network | Node | Network Address |
| 33 | What does H represent in the N.N.N.H ? | Network Address | Host Address | Next Network | Node | Host Address |
| 34 | ____ is a device used to connect different Network | Switch | bridge | Router | Hub | Router |
| 35 | CIDR refers to ____ | Classless Intra Domain Routing | Classful Intra Domain Routing | Classless Interdomain Routing | Classful InterDomain Routing | Classless Interdomain Routing |
| 36 | NIC stands for ____ | Network Interface Console | Network Interface Component | Network Interface Card | Network Interference Card | Network Interface Card |
| 37 | NAT refers to ____ | Network Address transalation | Neighbour Address transalation | Network Addressing Translator | Network Interdependent Translation | Network Address transalation |
| 38 | IPV6 adress is ___bit long | 156 | 64 | 32 | 128 | 128 |
| 39 | _____ is the process of directing a packet through the maze of networks that stand between its source and its destination. | Hacking | Switching | Translating | Routing | Routing |
| 40 | Routing is handled in ____ layer | Application | Datalink | Network | Transport | Network |
| 41 | _____ is the table used for Routing process in Router | Routing | Symbol | Grasping | Piping | Routing |
| 42 | ICMP is used to ____ | Report Error | Find Error | Remove Error | All | Report Error |
| 43 | DHCP refers to _____ | Digital Host Configuration Protocol | Dynamic host Control Protocol | Dynamic Host Configuration Protocol | Digital Host Control protocol | Dynamic Host Configuration Protocol |
| 44 | ARP refers to _____ | Address Reservation Protocol | Address Rendring Protocol | Adding Resolution Protocols | Address Resolution Protocol | Address Resolution Protocol |
| 45 | The ______ option of ifconfig sets the subnet mask for the interface and is required if the network is not subnetted according to its address class | Submask | Supermask | Masking | netmask | netmask |
| 46 | The ____ command defines static routes, explicit routing table entries that never change, even if you run a routing daemon. | netstat | route | mask | ipconfig | route |
| 47 | ______removes a specific entry from the routing table when used with route command | del | delete | remove | rmv | del |
| 48 | To inspect the Existing route we use ____command | route | netstat -nr | ifconfig | routing | netstat -nr |
| 49 | A _____ router causes all packets whose destination network is not found in the kernel's routing table to be sent to the indicated gateway. | Final | Alter | default | finally | default |
| 50 | ____ command is used to check the existance of a host | route | ifconfig | ping | traceout | ping |
| 51 | _____ is acommand that uncovers the sequence of gateways through which an IP packet travels to reach its destination. | route | ifconfig | ping | traceout | traceout |
| 52 | Network Statistic  can be known through ____ command | traceout | ping | ifcongig | netstat | netstat |
| 53 | _____ is a command used for inspecting live interface Activity | SAP | SAD | SAR | SRA | SAR |
| 54 | ____ is a tool used for packet sniffer | SAP | DAD | tcpdump | tcpclear | tcpdump |
| 55 | ____ is a tool used for packet sniffer | SAP | DAD | Wireshark | tcpclear | Wireshark |
| 56 | ______ is a visual Packet Sniffer | SAP | DAD | Wireshark | tcpclear | Wireshark |
| 57 | SNMP refers to ______ | Simple Network Mail Protocol | Simple Network Management Process | Simplex Network Management Process | Simple Network Management Protocol | Simple Network Management Protocol |
| 58 | _____ is a function used to create a child Process in UNIX | trap | frok | fork | down | fork |
| 59 | _____command is used to traverses a MIB starting at a particular OID | snmpget | snmprun | snmptable | snmpwalk | snmpwalk |
| 60 | Physical address is used in _____layer | application | Datalink | network | Presentation | Datalink |

**Reg.No ---------------------**

**[**16ITU502B**]**

**KARPAGAM ACADEMY OF HIGHER EDUCATION**

(Deemed to be University)

(Established Under Section 3 of UGC Act 1956)

COIMBATORE – 641 021

**B.Sc. DEGREE EXAMINATION**

**Fifth Semester - I Internal Examination**

**Information Technology**

**Network Programming**

**Time: 2 Hrs** **Max.Marks: 50**

**Date :** **Class: III B.Sc (IT) - 'A' & 'B'**

---

**Part - A** **(10 X 1 = 20)**

1. TCP is a ______type of Protocol

   a) **Connection Oriented** b) Connection Less c) Reference d) intermediate

2. Process to Process Communication is performed in ______ layer.

   a) Network b) Data Link Layer c) Physical d) **Transport**

3. Which of the following Protocol is not the Transport layer Protocol?

   a) TCP b) UDP c) **IP** d) SCTP

4. The TCP Connection establishment is called as ______Way handshake

   a) Two b) **Three** c) Four d) One

5. The Packets of UDP is called as _______

   a) **Datagram** b) Segment c) Packet d) Header

6. The Packets of TCP is called as _______

   a) Datagram b) **Segment** c) Packet d) Header

7. Systems in network is called as ______

   a) **Node** b) Tree c) System d) All

8. _______ is a combination of IP-address and Port Number

   a) Server b) Client c) Server Socket d) **Socket**

9. ______ is the unique address that identifies a machine in a network.

   a) **IP Address** b) Port Number c) Socket d) All

10. ______ is used for process to process communication.

    a) IP Address b) **Port Number** c) Socket d) All

11. Port numbers are classified in to _____types.

    a) 4 b) **3** c) 2 d) 5

12. The well Known port numbers are used by _____

    a) Client b) **Server** c) Client & Server d) Routers

13. The System that request for Service is called _____

    a) Sever b) **Client** c) Router d) Hub

14. The System that Provide Service is called _____

a) **Sever** b) Client c) Router d) Hub

15. Dynamic ports are also called as _______port

a) **Private** b) Public c) Server d) Reserved

16. MTU refers to _______

a) Minimum Transfer Unit b) Maximum Transport Unit c) **Maximum Transfer Unit**

d) Minimum Transport Unit

17. DF refers to _____

a) Do fragment b) **Don't Fragment** c) Double Fragment d) None

18. The number of layers in TCP/IP reference model is ____

a) 7 b) **5** c) 4 d) 6

19. The number of layers in OSI reference model is ____

a) **7** b) 5 c) 4 d) 6

20. The header size of TCP packet is ______ bytes.

a) **20** b) 21 c) 25 d) 18

**Part - B** **(3 X 2 = 6)**

21. What is a Network?

**Ans**: The Collection of systems that are inter-connected to share the resource is called Network.

22. What is connection oriented System?

**Ans**: The process to process connection which provides reliability, uses single communication channel, manages error control and flow control is called Connection Oriented Systems.

23. What is a Port Number?

**Ans**: It is a 16 bit number which identifies the process in a machine which is having communication with others machine.

**Part - C** **(3 X 8 = 24)**

24. a) What is reference model? Explain about OSI reference Model.

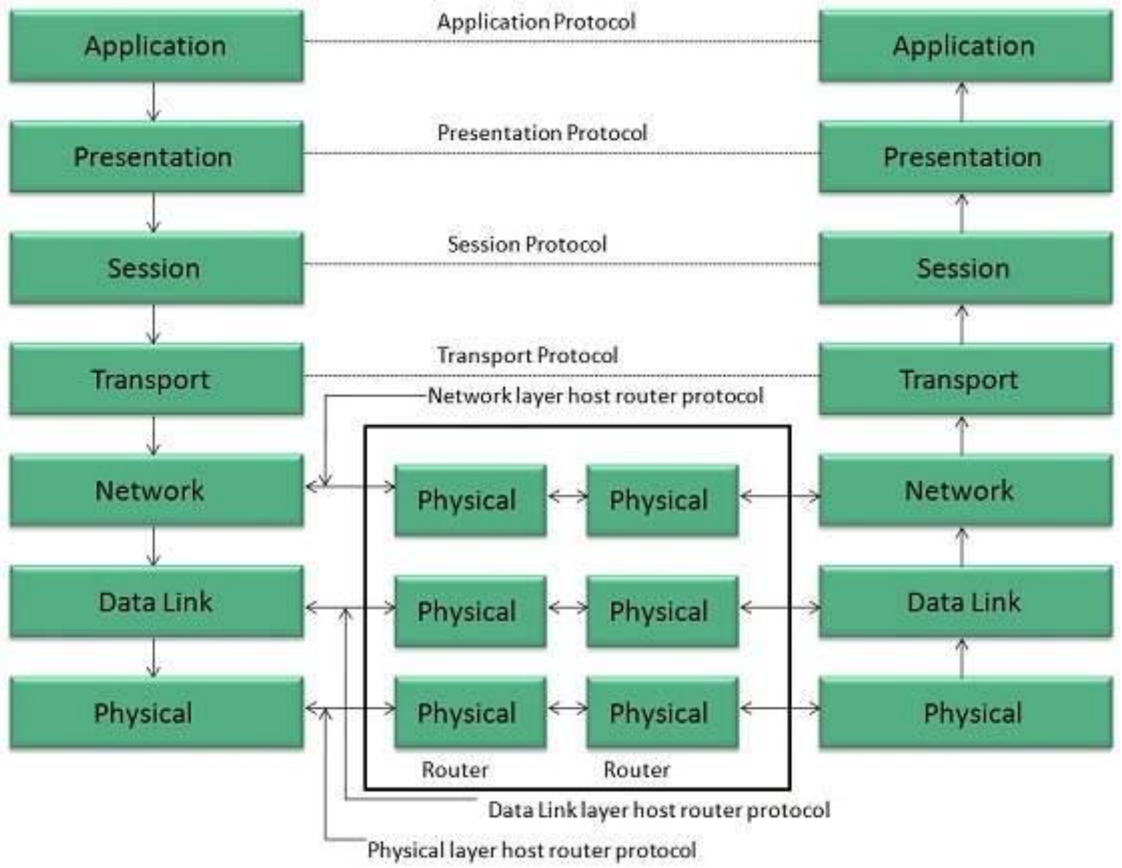
**Ans: OSI** is acronym of **Open System Interface**. This model is developed by the **International organization of Standardization (ISO)** and therefore also referred as **ISO-OSI** Model.The OSI model consists of seven layers as shown in the following diagram. Each layer has a specific function, however each layer provide services to the layer above.

**Physical Layer**

The Physical layer is responsible for the following activities:

- Activating, maintaining and deactivating the physical connection.
- Defining voltages and data rates needed for transmission.
- Converting digital bits into electrical signal.

- Deciding whether the connection is simplex, half duplex or full duplex.



**Data Link Layer**

The data link layer performs the following functions:

- Performs synchronization and error control for the information which is to be transmitted over the physical link.
- Enables error detection, and adds error detection bits to the data which are to be transmitted.

**Network Layer**

Following are the functions of Network Layer:

- To route the signals through various channels to the other end.
- To act as the network controller by deciding which route data should take.
- To divide the outgoing messages into packets and to assemble incoming packets into messages for higher levels.

**Transport Layer**

The Transport layer performs the following functions:

- It decides if the data transmission should take place on parallel paths or single path.
- It performs multiplexing, splitting on the data.
- It breaks the data groups into smaller units so that they are handled more efficiently by the network layer.

**Session Layer**

The Session layer performs the following functions:

- Manages the messages and synchronizes conversations between two different applications.
- It controls logging on and off, user identification, billing and session management.

**Presentation Layer**

The Presentation layer performs the following functions:

- This layer makes it sure that the information is delivered in such a form that the receiving system will understand and use it.

**Application Layer**

The Application layer performs the following functions:

- It provides different services such as manipulation of information in several ways, retransferring the files of information, distributing the results etc.
- The functions such as LOGIN or password checking are also performed by the application layer.

(or)

b) Discuss in detail about UDP.

**Ans: USER DATAGRAM PROTOCOL (UDP)**

The User Datagram Protocol (UDP) is called a connectionless, unreliable transport protocol. It does not add anything to the services of IP except to provide process-to process communication instead of host-to-host communication. Also, it performs very limited error checking.

UDP is a very simple protocol using a minimum of overhead. If a process wants to send a small message and does not care much about reliability, it can use UDP. Sending a small message by using UDP takes much less interaction between the sender and receiver than using TCP or SCTP.

**Well-Known Ports for UDP**

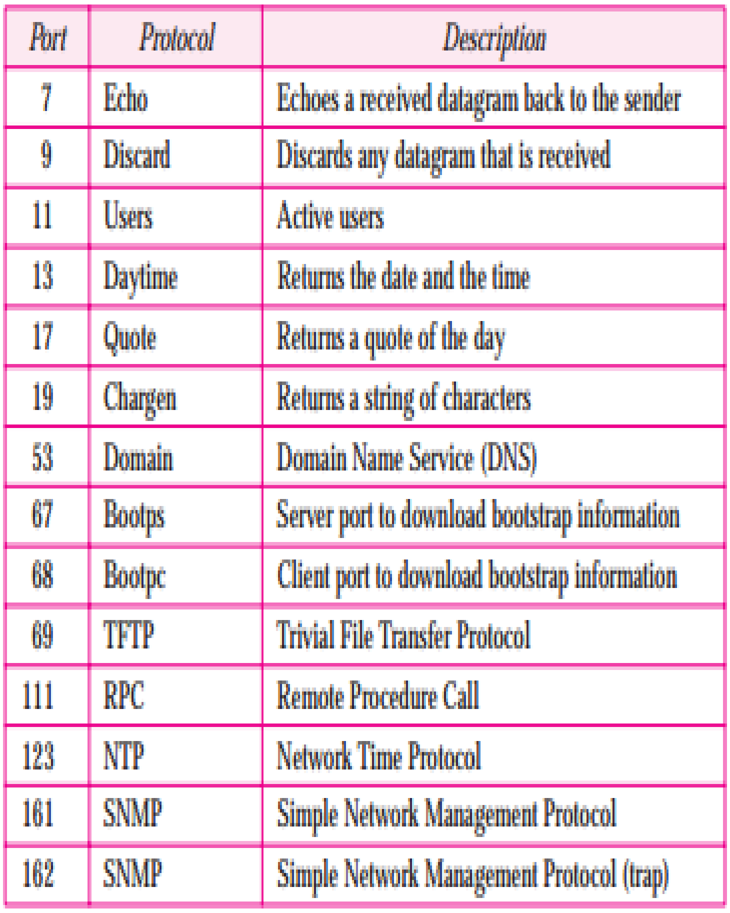
Table shows some well-known port numbers used by UDP. Some port numbers can be used by both UDP and TCP.

**User Datagram**

UDP packets, called user datagrams, have a fixed-size header of 8 bytes. Figure 8 shows the format of a user datagram. The fields are as follows:

***Source port number:*** This is the port number used by the process running on the source host. It is 16 bits long, which means that the port number can range from 0 to 65,535. If the source host is the client (a client sending a request), the port number, in most cases, is an ephemeral port number requested by the process and chosen by the UDP software running on the source host. If the source host is the server (a server sending a response), the port number, in most cases, is a well-known port number

**Table :** ***Well-known ports used with UDP***



| *Port* | *Protocol* | *Description* |
|---|---|---|
| 7 | Echo | Echoes a received datagram back to the sender |
| 9 | Discard | Discards any datagram that is received |
| 11 | Users | Active users |
| 13 | Daytime | Returns the date and the time |
| 17 | Quote | Returns a quote of the day |
| 19 | Chargen | Returns a string of characters |
| 53 | Domain | Domain Name Service (DNS) |
| 67 | Bootps | Server port to download bootstrap information |
| 68 | Bootpc | Client port to download bootstrap information |
| 69 | TFTP | Trivial File Transfer Protocol |
| 111 | RPC | Remote Procedure Call |
| 123 | NTP | Network Time Protocol |
| 161 | SNMP | Simple Network Management Protocol |
| 162 | SNMP | Simple Network Management Protocol (trap) |

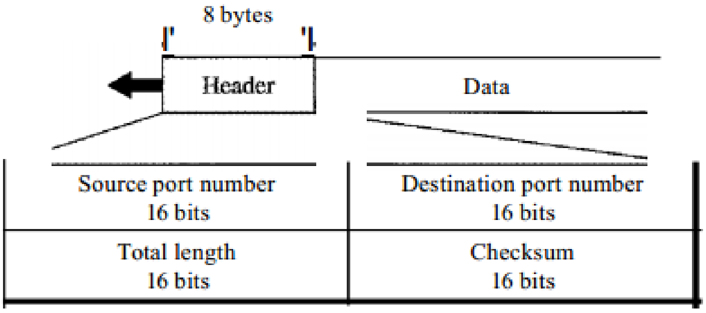


**Fig.8: User datagram format**

***Destination port number:*** This is the port number used by the process running on the destination host. It is also 16 bits long. If the destination host is the server (a client sending a request), the port number, in most cases, is a well-known port number. If the destination host is the client (a server sending a response), the port number, in most cases, is an ephemeral port number. In this case, the server copies the ephemeral port number it has received in the request packet.

***Length:*** This is a 16-bit field that defines the total length of the user datagram, header plus data. The 16 bits can define a total length of 0 to 65,535 bytes. However, the total length needs to be much less because a UDP user datagram is stored in an IP datagram with a total length of 65,535 bytes.

**UDP Operation**

UDP uses concepts common to the transport layer. These concepts will be discussed here briefly, and then expanded in the next section on the TCP protocol.

***Connectionless Services***

As mentioned previously, UDP provides a connectionless service. This means that each user datagram sent by UDP is an independent datagram. There is no relationship between the different user datagrams even if they are coming from the same source process and going to the same destination program. The user datagrams are not numbered. Also, there is no connection establishment and no connection termination, as is the case for TCP. This means that each user datagram can travel on a different path.  One of the ramifications of being connectionless is that the process that uses UDP cannot send a stream of data to UDP and expect UDP to chop them into different related user datagrams. Instead each request must be small enough to fit into one user datagram. Only those processes sending short messages should use UDP.

***Flow and Error Control***

UDP is a very simple, unreliable transport protocol. There is no flow control and hence no window mechanism. The receiver may overflow with incoming messages. There is no error control mechanism in UDP except for the checksum. This means that the sender does not know if a message has been lost or duplicated. When the receiver detects an error through the checksum, the user datagram is silently discarded. The lack of flow control and error control means that the process using UDP should provide these mechanisms.

***Encapsulation and Decapsulation***

To send a message from one process to another, the UDP protocol encapsulates and decapsulates messages in an IP datagram.

***Queuing***

We have talked about ports without discussing the actual implementation of them. In UDP, queues are associated with ports (see Figure 11).

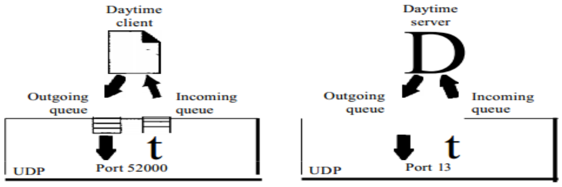


**Fig.11 Queues in UDP**

At the client site, when a process starts, it requests a port number from the operating system. Some implementations create both an incoming and an outgoing queue associated with each process. Other implementations create only an incoming queue associated with each process. Note that even if a process wants to communicate with multiple processes, it obtains only one port number and eventually one outgoing and one incoming queue. The queues opened by the client are, in most cases, identified by ephemeral port numbers.

The queues function as long as the process is running. When the process terminates, the queues are destroyed. The client process can send messages to the outgoing queue by using the source port number specified in the request. UDP removes the messages one by one and, after adding the UDP header, delivers them to IP. An outgoing queue can overflow. If this happens, the operating system can ask the client process to wait before sending any more messages. When a message arrives for a client, UDP checks to see if an incoming queue has been created for the port number specified in the destination port number field of the user datagram. If there is such a queue, UDP sends the received user datagram to the end of the queue. If there is no such queue, UDP discards the user datagram and asks the ICMP protocol to send a *port unreachable* message to the server. All the incoming messages for one particular client program, whether coming from the same or a different server, are sent to the same queue. An incoming queue can overflow. If this happens, UDP drops the user datagram and asks for a port unreachable message to be sent to the server. At the server site, the mechanism of creating queues is different. In its simplest form, a server asks for incoming and outgoing queues, using its well-known port, when it starts running. The queues remain open as long as the server is running.

When a message arrives for a server, UDP checks to see if an incoming queue has been created for the port number specified in the destination port number field of the user datagram. If there is such a queue, UDP sends the received user datagram to the end of the queue. If there is no such queue, UDP discards the user datagram and asks the ICMP protocol to send a port unreachable message to the client. All the incoming messages for one particular server, whether coming from the same or a different client, are sent to the same queue. An incoming queue can overflow. If this happens, UDP drops the user datagram and asks for a port unreachable message to be sent to the client. When a server wants to respond to a client, it sends messages to the outgoing queue, using the source port number specified in the request. UDP removes the messages one by one and, after adding the UDP header, delivers them to IP. An outgoing queue can overflow. If this happens, the operating system asks the server to wait before sending any more messages.

**Use of UDP**

The following lists some uses of the UDP protocol:

- UDP is suitable for a process that requires simple request-response communication with little concern for flow and error control. It is not usually used for a process such as FrP that needs to send bulk data.

- UDP is suitable for a process with internal flow and error control mechanisms. For example, the Trivial File Transfer Protocol (TFTP) process includes flow and error control. It can easily use UDP.
- UDP is a suitable transport protocol for multicasting. Multicasting capability is embedded in the UDP software but not in the TCP software.
- UDP is used for management processes such as SNMP .
- UDP is used for some route updating protocols such as Routing Information Protocol (RIP).

25. a) What is Connection Oriented Service? Discuss about TCP in detail.

**Ans: TCP**

The second transport layer protocol we are going to discuss is called **Transmission Control Protocol (TCP**). TCP, like UDP, is a process-to-process (program-to-program) protocol. TCP, therefore, like UDP, uses port numbers.

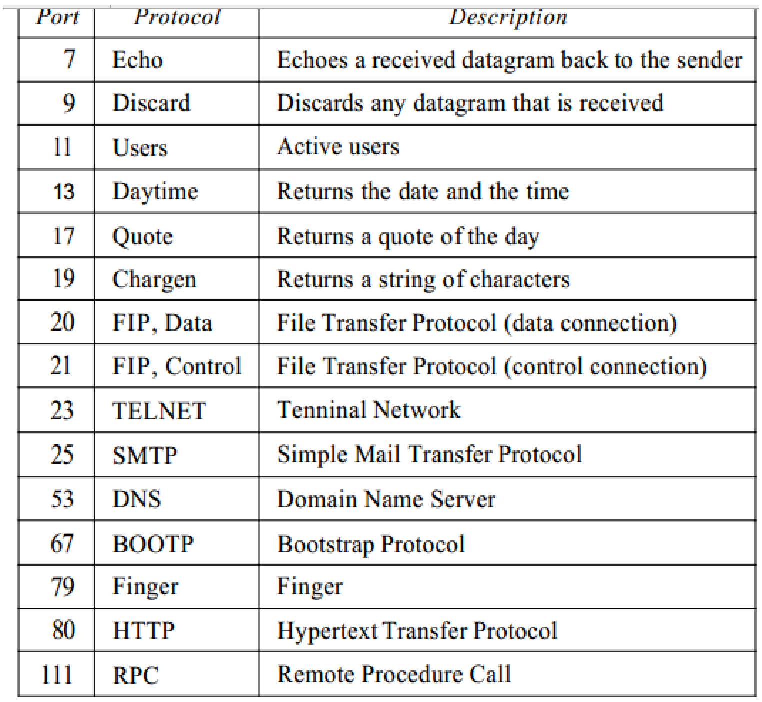
Unlike UDP, TCP is a connection oriented protocol; it creates a virtual connection between two TCPs to send data. In addition, TCP uses flow and error control mechanisms at the transport level. In brief, TCP is called a *connection-oriented, reliable* transport protocol. It adds connection-oriented and reliability features to the services of IP.

**TCP Services**

Before we discuss TCP in detail, let us explain the services offered by TCP to the processes at the application layer.

***Process-to-Process Communication*** TCP provides process-to-process communication using port numbers. Table 1 lists some well-known port numbers used by TCP.

**Table 1: Port numbers used by TCP**



| Port | Protocol | Description |
|---|---|---|
| 7 | Echo | Echoes a received datagram back to the sender |
| 9 | Discard | Discards any datagram that is received |
| 11 | Users | Active users |
| 13 | Daytime | Returns the date and the time |
| 17 | Quote | Returns a quote of the day |
| 19 | Chargen | Returns a string of characters |
| 20 | FIP, Data | File Transfer Protocol (data connection) |
| 21 | FIP, Control | File Transfer Protocol (control connection) |
| 23 | TELNET | Tenninal Network |
| 25 | SMTP | Simple Mail Transfer Protocol |
| 53 | DNS | Domain Name Server |
| 67 | BOOTP | Bootstrap Protocol |
| 79 | Finger | Finger |
| 80 | HTTP | Hypertext Transfer Protocol |
| 111 | RPC | Remote Procedure Call |

***Stream Delivery Service***

TCP is a stream-oriented protocol. In UDP, a process (an application program) sends messages, with predefined boundaries, to UDP for delivery. UDP adds its own header to each of these messages and delivers them to IP for transmission. Each message from the process is calIed a user datagram and becomes, eventually, one IP datagram. Neither IP nor UDP recognizes any relationship between the datagrams.

TCP, on the other hand, allows the sending process to deliver data as a stream of bytes and allows the receiving process to obtain data as a stream of bytes. TCP creates an environment in which the two processes seem to be connected by an imaginary "tube" that carries their data across the Internet.

This imaginary environment is depicted in Figure 1. The sending process produces (writes to) the stream of bytes, and the receiving process consumes (reads from) them.

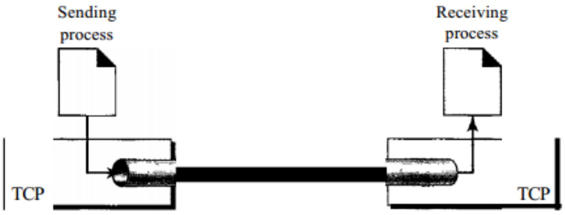


**Fig.1: Stream delivery**

**Sending and Receiving Buffers**

Because the sending and the receiving processes may not write or read data at the same speed, TCP needs buffers for storage. There are two buffers, the sending buffer and the receiving buffer, one for each direction. One way to implement a buffer is to use a circular array of I-byte locations as shown in Figure 2. For simplicity, we have shown two buffers of 20 bytes each; normally the buffers are hundreds or thousands of bytes, depending on the implementation. We also show the buffers as the same size, which is not always the case.

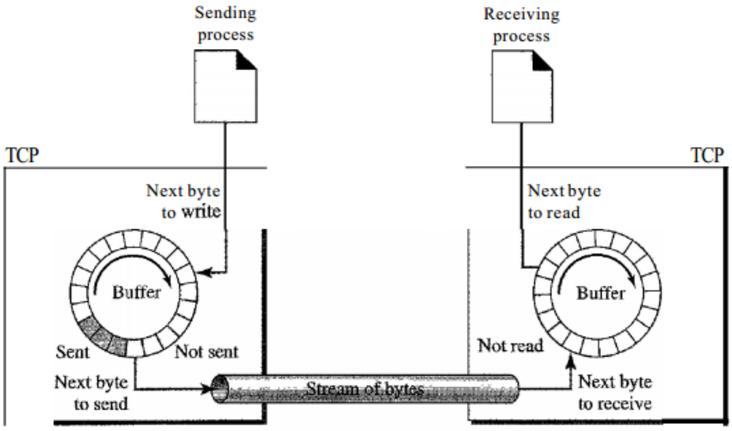


**Fig.2: Sending and receiving buffers**

Figure 2 shows the movement of the data in one direction. At the sending site, the buffer has three types of chambers. The white section contains empty chambers that can be filled by the sending process (producer). The gray area holds bytes that have been sent but not yet acknowledged. TCP keeps these bytes in the buffer until it receives an acknowledgment. The colored area contains bytes to be sent by the sending TCP. However, as we will see later in this chapter, TCP may be able to send only part of this colored section. This could be due to the slowness of the receiving process or perhaps to congestion in the network. Also note that after the bytes in the gray chambers are acknowledged, the chambers are recycled and available for use by the sending process. This is why we show a circular buffer.

The operation of the buffer at the receiver site is simpler. The circular buffer is divided into two areas (shown as white and colored). The white area contains empty chambers to be filled by bytes received from the network. The colored sections contain received bytes that can be read by the receiving process. When a byte is read by the receiving process, the chamber is recycled and added to the pool of empty chambers.

**Segments:** Although buffering handles the disparity between the speed of the producing and consuming processes, we need one more step before we can send data. The IP layer, as a service provider for TCP, needs to send data in packets, not as a stream of bytes. At the transport layer, TCP groups a number of bytes together into a packet called a segment. TCP adds a header to each segment (for control purposes) and delivers the segment to the IP layer for transmission. The segments are encapsulated in IP datagrams and transmitted. This entire operation is transparent to the receiving process. Later we will see that segments may be received out of order, lost, or corrupted and resent. All these are handled by TCP with the receiving process unaware of any activities. Figure 3 shows how segments are created from the bytes in the buffers.

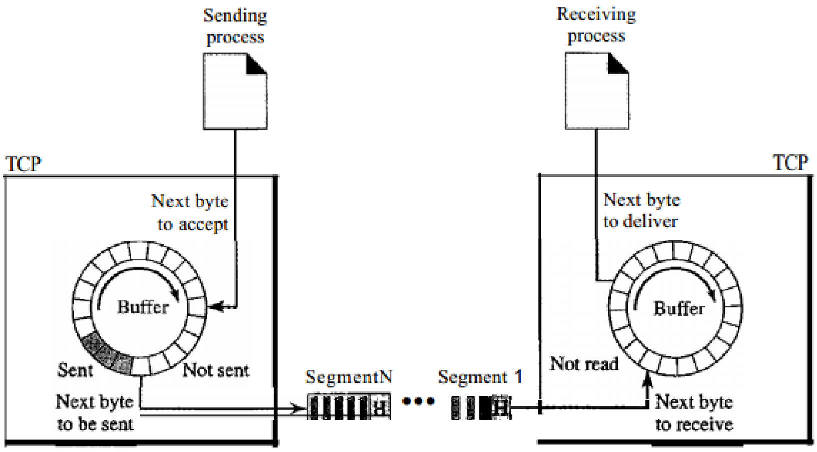


**Fig.3: TCP Segments**

Note that the segments are not necessarily the same size. In Figure 3, for simplicity, we show one segment carrying 3 bytes and the other carrying 5 bytes. In reality, segments carry hundreds, if not thousands, of bytes.

***Full-Duplex Communication***

TCP offers full-duplex service, in which data can flow in both directions at the same time. Each TCP then has a sending and receiving buffer, and segments move in both directions.

***Connection-Oriented Service***

TCP, unlike UDP, is a connection-oriented protocol. When a process at site A wants to send and receive data from another process at site B, the following occurs:

1. The two TCPs establish a connection between them.
2. Data are exchanged in both directions.
3. The connection is terminated.

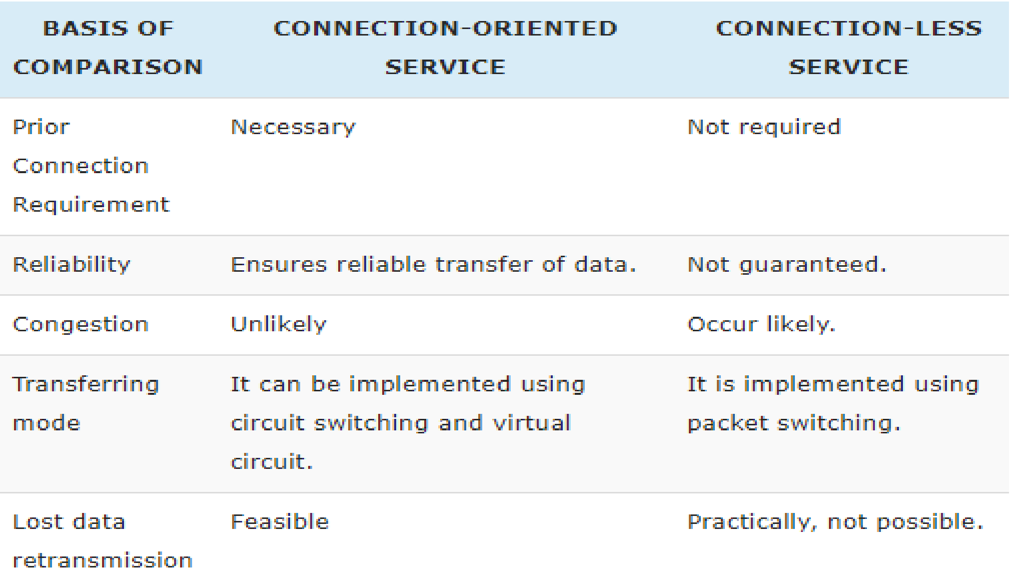
Note that this is a virtual connection, not a physical connection. The TCP segment is encapsulated in an IP datagram and can be sent out of order, or lost, or corrupted, and then resent. Each may use a different path to reach the destination. There is no physical connection. TCP creates a stream-oriented environment in which it accepts the responsibility of delivering the bytes in order to the other site. The situation is similar to creating a bridge that spans multiple islands and passing all the bytes from one island to another in one single connection.

***Reliable Service*** TCP is a reliable transport protocol. It uses an acknowledgment mechanism to check the safe and sound arrival of data. We will discuss this feature further in the section on error control.

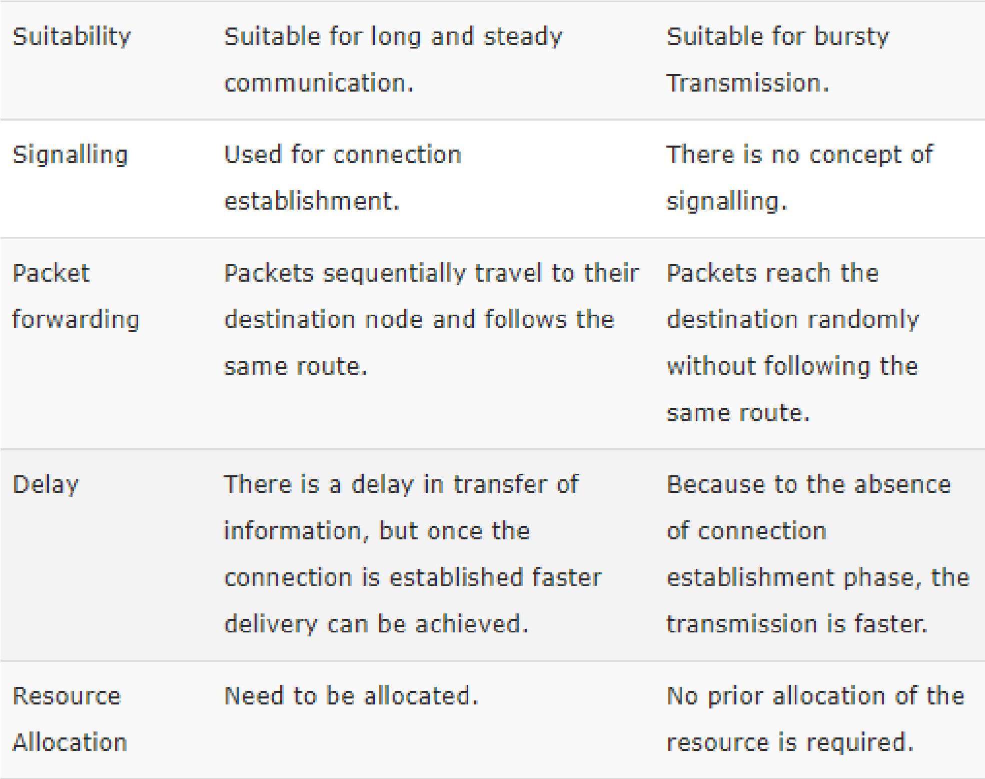
(or)

b) Differentiate Connection Oriented and Connection Less protocols.

**Ans:**



| BASIS OF COMPARISON | CONNECTION-ORIENTED SERVICE | CONNECTION-LESS SERVICE |
|---|---|---|
| Prior Connection Requirement | Necessary | Not required |
| Reliability | Ensures reliable transfer of data. | Not guaranteed. |
| Congestion | Unlikely | Occur likely. |
| Transferring mode | It can be implemented using circuit switching and virtual circuit. | It is implemented using packet switching. |
| Lost data retransmission | Feasible | Practically, not possible. |



| | | |
|---|---|---|
| Suitability | Suitable for long and steady communication. | Suitable for bursty Transmission. |
| Signalling | Used for connection establishment. | There is no concept of signalling. |
| Packet forwarding | Packets sequentially travel to their destination node and follows the same route. | Packets reach the destination randomly without following the same route. |
| Delay | There is a delay in transfer of information, but once the connection is established faster delivery can be achieved. | Because to the absence of connection establishment phase, the transmission is faster. |
| Resource Allocation | Need to be allocated. | No prior allocation of the resource is required. |

26. a) What is a socket? Discuss in general about the socket.

**Ans: Socket Programming**

Sockets allow communication between two different processes on the same or different machines.

**Where is Socket Used?**

A Unix Socket is used in a client-server application framework. A server is a process that performs some functions on request from a client. Most of the application-level protocols like FTP, SMTP, and POP3 make use of sockets to establish connection between client and server and then for exchanging data.

**Socket Types**

There are four types of sockets available to the users. The first two are most commonly used and the last two are rarely used. Processes are presumed to communicate only between sockets of the same type but there is no restriction that prevents communication between sockets of different types.

- **Stream Sockets** − Delivery in a networked environment is guaranteed. If you send through the stream socket three items "A, B, C", they will arrive in the same order − "A, B, C". These sockets use TCP (Transmission Control Protocol) for data transmission. If delivery is impossible, the sender receives an error indicator. Data records do not have any boundaries.
- **Datagram Sockets** − Delivery in a networked environment is not guaranteed. They're connectionless because you don't need to have an open connection as in Stream Sockets −

you build a packet with the destination information and send it out. They use UDP (User Datagram Protocol).

- **Raw Sockets** − These provide users access to the underlying communication protocols, which support socket abstractions. These sockets are normally datagram oriented, though their exact characteristics are dependent on the interface provided by the protocol. Raw sockets are not intended for the general user; they have been provided mainly for those interested in developing new communication protocols, or for gaining access to some of the more cryptic facilities of an existing protocol.
- Sequenced Packet Sockets − They are similar to a stream socket, with the exception that record boundaries are preserved. This interface is provided only as a part of the Network Systems (NS) socket abstraction, and is very important in most serious NS applications. Sequenced-packet sockets allow the user to manipulate the Sequence Packet Protocol (SPP) or Internet Datagram Protocol (IDP) headers on a packet or a group of packets, either by writing a prototype header along with whatever data is to be sent, or by specifying a default header to be used with all outgoing data, and allows the user to receive the headers on incoming packets.

(or)

b) Write a simple java socket program which transfer data between client and Server.

**Ans:**

```
//TCP Client
import java.io.*;
import java.net.*;
import java.util.*;
class TcpClient
{
            public static void main(String str[]) throws IOException, Exception
            {
                  Socket sock = new Socket("localHost",1999);
                  int i,n;
                  Scanner s = new Scanner(System.in);
                  DataOutputStream out = new
DataOutputStream(sock.getOutputStream());
                  DataInputStream in = new DataInputStream(sock.getInputStream());
                  System.out.println("Enter the Number of Frames");
                  n = s.nextInt();
                  for(i=1;i<=n;i++)
                  {
```

```
                                        out.writeInt(i);
                                        System.out.println("Enter the " + i + " string");
                                        String msg = s.next();
                                        out.writeUTF(msg);
                                        Thread.sleep(1000);
                                        System.out.println(in.readUTF());
                                }
                                out.writeInt(0);
                                sock.close();
                        }
        }
//TCP Server
import java.io.*;
import java.net.*;
class TcpServer
{
        public static void main(String str[]) throws IOException, Exception
        {
                ServerSocket server = new ServerSocket(1999);
                Socket sock = server.accept();
                int seqno=-1;
                String msg;
                DataInputStream br = new DataInputStream(sock.getInputStream());
                DataOutputStream out = new DataOutputStream(sock.getOutputStream());
                while(seqno!=0)
                {
                        seqno=br.readInt();
                        msg=br.readUTF();
                        System.out.println("Frame " + seqno + " is Read");
                        System.out.println(msg);
                        out.writeUTF("Looking for " + (seqno+1) + "Frame");
                }
                sock.close();
                server.close();
        }
}
```

**Reg.No ---------------------**

[16ITU502B]

**KARPAGAM ACADEMY OF HIGHER EDUCATION**

(Deemed to be University)

(Established Under Section 3 of UGC Act 1956)

COIMBATORE – 641 021

**B.Sc. DEGREE EXAMINATION**

**Fifth Semester - II Internal Examination**

**Information Technology**

**Network Programming**

**Time: 2 Hrs** **Max.Marks: 50**

**Date :** **Class: III B.Sc (IT) - 'A' & 'B'**

---

**Part - A** **(10 X 1 = 20)**

**Answer all the Questions**

1. htons refres to _____

   a) host-network address b) host-network short c) host-network long d) all

2. _____ is a 16 bit number used in socket.

   a) Port b) IP number c) Switch d) Logical Address

3. AF_INET refers to ____

   a) Family name b) Socket Type c) Connection Type d) None

4. fork functions is used for ____

   a) Creating a copy of running process b) Creating a child c) Both a & b d) None

5. Which of the following functions denotes the maximum users of a socket to kernel?

   a) bind b) accept c) socket d) listen

6. ______ is used for process to process communication.

   IP Address b) Port Number c) Socket d) All

7. The ____functions of the server makes the server to accept the client request

   a) listen b) accept c) bind d) close

8. The ____ function combines the socket with the server configurations.

   a) listen b) accept c) bind d) close

9. Which of the following does the sockfd parameter of read refers to ?

   a) Socketfd b) acceptid c) connectid d) All

10. ______ is the unique address that identifies a machine in a network.

    a) IP Address b) Port Number c) Socket d) All

11. _______ is a combination of IP-address and Port Number

    a) Server b) Client c) Server Socket d) Socket

12. AF_NET refers to ____Family.

    a) IPV6 b) IPv5 c) IPv4 d) ICMP

13. AF_NET6 refers to ____Family.

a) IPV6 b) IPv5 c) IPv4 d) ICMP

14. Which of the following function is used to clear the values of the structure variable of type sockaddr_in?

a) bzero b) bcopy c) null d) all

15. To represent the default protocol of the machine to socket the constant ____is used.

a) 2 b) 3 c) 4 d) 0

16. Which of the following socket type refers to connection oriented?

a) SOCK_DGRAM b) SOCK_SEQPACKET c) SOCK_STREAM d) None

17. The Syntax of getsockname takes ____parameters.

a) 3 b) 4 c) 5 d) 6

18. Which of the following socket type refers to connectionless socket?

a) SOCK_DGRAM b) SOCK_SEQPACKET c) SOCK_STREAM d) None

19. _____ signal is generated when child process terminate.

a) SIGDFL b) SIGIGN c) SIGCHILD d) SIGPIPE

20. ___ is the function used to terminate the three way handshake with TIME_WAIT.

a) close b) terminate c) closecon d) termcon

**Part - B** **(3 X 2 = 6)**

**Answer all the Questions**

21. What is the return values of fork command?

22. Write the syntax of socket function.

23. Write the structure of sockaddr structure.

**Part - C** **(3 X 8 = 24)**

**Answer all the Questions**

24. a) Discuss about the functions that support Passive Open in a Server.

(or)

b) Write a note on Concurrent Servers.

25. a) Explain the functions that support Active Open in a Client.

(or)

b) Write a note on fork and exec functions

26. a) Write a c program that simulates the TCP/IP Echo server.

(or)

b) Write a detail note on a) getsockname b) getpeername c) signal handling

**Reg.No ---------------------**

**[**16ITU502B**]**

**KARPAGAM ACADEMY OF HIGHER EDUCATION**

(Deemed to be University)

(Established Under Section 3 of UGC Act 1956)

COIMBATORE – 641 021

**B.Sc. DEGREE EXAMINATION**

**Fifth Semester - II Internal Examination**

**Information Technology**

**Network Programming**

**Time: 2 Hrs** **Max.Marks: 50**

**Date :** **Class: III B.Sc (IT) - 'A' & 'B'**

---

## Part - A (10 X 1 = 20)

**Answer all the Questions**

1. htons refres to _____

   a) host-network address b) **host-network short** c) host-network long d) all

2. _____ is a 16 bit number used in socket.

   a) **Port** b) IP number c) Switch d) Logical Address

3. AF_INET refers to ____

   a) **Family name** b) Socket Type c) Connection Type d) None

4. fork functions is used for ____

   a) Creating a copy of running process b) Creating a child c) **Both a & b** d) None

5. Which of the following functions denotes the maximum users of a socket to kernel?

   a) bind b) accept c) socket d) **listen**

6. ______ is used for process to process communication.

   a) IP Address b) **Port Number** c) Socket d) All

7. The ____functions of the server makes the server to accept the client request

   a) **listen** b) accept c) bind d) close

8. The ____ function combines the socket with the server configurations.

   a) listen b) accept c) **bind** d) close

9. Which of the following does the sockfd parameter of read refers to?

   a) Socketfd b) **acceptid** c) connectid d) All

10. ______ is the unique address that identifies a machine in a network.

    a) **IP Address** b) Port Number c) Socket d) All

11. _______ is a combination of IP-address and Port Number

    a) Server b) Client c) Server Socket d) **Socket**

12. AF_NET refers to ____Family.

    a) IPV6 b) IPv5 c) **IPv4** d) ICMP

13. AF_NET6 refers to ____Family.

    a) **IPV6** b) IPv5 c) IPv4 d) ICMP

14. Which of the following function is used to clear the values of the structure variable of type sockaddr_in?

    a) **bzero** b) bcopy c) null d) all

15. To represent the default protocol of the machine to socket the constant ____is used.

    a) 2 b) 3 c) 4 d) **0**

16. Which of the following socket type refers to connection oriented?

    a) SOCK_DGRAM b) SOCK_SEQPACKET c) **SOCK_STREAM** d) None

17. The Syntax of getsockname takes ____parameters.

    a) **3** b) 4 c) 5 d) 6

18. Which of the following socket type refers to connectionless socket?

    a) **SOCK_DGRAM** b) SOCK_SEQPACKET c) SOCK_STREAM d) None

19. _____ signal is generated when child process terminate.

    a) SIGDFL b) SIGIGN c) **SIGCHILD** d) SIGPIPE

20. ___ is the function used to terminate the three way handshake with TIME_WAIT.

    a) **close** b) terminate c) closecon d) termcon

**Part - B** **(3 X 2 = 6)**

**Answer all the Questions**

21. What is the return value of fork command?

    **Ans:** This function is called once and it returns two values (One from parent and other from child).

    Parent returns process-id of Child and Child returns ‘0’ on successful creation.

22. Write the syntax of socket function.

    **Ans:** socket(Family,Connection_Stream,Protocol)

23. Write the structure of sockaddr structure.

    **Ans:** struct sockaddr

```
{
        uint8_t  sa_len;
        sa_family_t  sa_family;    /* address family: AF_xxx value */
        char sa_data[14];  /* protocol-specific address */
};
```

**Part - C** **(3 X 8 = 24)**

**Answer all the Questions**

24. a) Discuss about the functions that support Passive Open in a Server.

**Ans:**

**i) socket Function**

To perform network I/O, the first thing a process must do is call the socket function, specifying the type of communication protocol desired (TCP using IPv4, UDP using IPv6, Unix domain stream protocol, etc.).

```
int socket (int family, int type, int protocol);
/* Returns: non-negative descriptor if OK, -1 on error */
```

**Arguments:**

- *family* specifies the protocol family and is one of the constants in the table below. This argument is often referred to as *domain* instead of *family*.
- *Family* = AF_INET, AF_INET6, AF_LOCAL, AF_ROUTE, AF_KEY
- The socket *type* = SOCK_STREAM, SOCK_DGRAM,SOCK_SEQPACKET,SOCK_RAW
- The *protocol* argument to 0 to select the system's default for the given combination of *family* and *type* or to IPPROTO_TCP, IPPROTO_UDP, IPPROTO_SCTP

  Not all combinations of socket *family* and *type* are valid.

**bind Function**

The bind function assigns a local protocol address to a socket. The protocol address is the combination of either a 32-bit IPv4 address or a 128-bit IPv6 address, along with a 16-bit TCP or UDP port number.

```
#include <sys/socket.h>
int bind(int sockfd, const struct sockaddr * myaddr, socklen_t addrlen);
Returns 0 if OK else -1 on error
```

The first argument sockfd refers to fd of socket function

The second argument *myaddr* is a pointer to a protocol-specific addres

The third argument *addrlen* is the size of this address structure.

With TCP, calling bind lets us specify a port number, an IP address, both, or neither.

- **Servers bind their well-known port when they start.** If a TCP client or server does not do this, the kernel chooses an ephemeral port for the socket when either connects or listen is called.
- It is normal for a TCP client to let the kernel choose an ephemeral port, unless the application requires a reserved port.

- However, it is rare for a TCP server to let the kernel choose an ephemeral port, since servers are known by their well-known port.
- Exceptions to this rule are Remote Procedure Call (RPC) servers. They normally let the kernel choose an ephemeral port for their listening socket since this port is then registered with the RPC port mapper. Clients have to contact the port mapper to obtain the ephemeral port before they can connect to the server. This also applies to RPC servers using UDP.
- **A process can bind a specific IP address to its socket.** The IP address must belong to an interface on the host.
- For a TCP client, this assigns the source IP address that will be used for IP datagrams sent on the socket. Normally, a TCP client does not bind an IP address to its socket. The kernel chooses the source IP address when the socket is connected, based on the outgoing interface that is used, which in turn is based on the route required to reach the server.
- For a TCP server, this restricts the socket to receive incoming client connections destined only to that IP address. If a TCP server does not bind an IP address to its socket, the kernel uses the destination IP address of the client's SYN as the server's source IP address.

**listen function**

The listen function is called only by a TCP server and it performs two actions:

- The listen function converts an unconnected socket into a passive socket, indicating that the kernel should accept incoming connection requests directed to this socket. In terms of the TCP state transition diagram ,the call to listen moves the socket from the CLOSED state to the LISTEN state.
- When a socket is created by the socket function (and before calling listen), it is assumed to be an active socket, that is, a client socket that will issue a connect.
- The second argument *backlog* to this function specifies the maximum number of connections the kernel should queue for this socket.
- This function is normally called after both the socket and bind functions and must be called before calling the accept function.

**Connection queues ***

To understand the *backlog* argument, we must realize that for a given listening socket, the kernel maintains two queues:

- An **incomplete connection queue**, which contains an entry for each SYN that has arrived from a client for which the server is awaiting completion of the TCP three-way handshake. These sockets are in theSYN_RCVD state.

- A **completed connection queue**, which contains an entry for each client with whom the TCP three-way handshake has completed. These sockets are in the ESTABLISHED state.

**accept Function**

accept is called by a TCP server to return the next completed connection from the front of the completed connection queue. If the completed connection queue is empty, the process is put to sleep (assuming the default of a blocking socket).

```
#include <sys/socket.h>
int accept (int sockfd, struct sockaddr *cliaddr, socklen_t *addrlen);
/* Returns: non-negative descriptor if OK, -1 on error */
```

The *cliaddr* and *addrlen* arguments are used to return the protocol address of the connected peer process (the client). *addrlen* is a value-result argument :

- Before the call, we set the integer value referenced by **addrlen* to the size of the socket address structure pointed to by *cliaddr*;
- On return, this integer value contains the actual number of bytes stored by the kernel in the socket address structure.

If successful, accept returns a new descriptor automatically created by the kernel. This new descriptor refers to the TCP connection with the client.

- The **listening socket** is the first argument (*sockfd*) to accept (the descriptor created by socket and used as the first argument to both bind and listen).
- The **connected socket** is the return value from accept the connected socket.

It is important to differentiate between these two sockets:

- A given server normally creates only one listening socket, which then exists for the lifetime of the server.
- The kernel creates one connected socket for each client connection that is accepted (for which the TCP three-way handshake completes).
- When the server is finished serving a given client, the connected socket is closed.

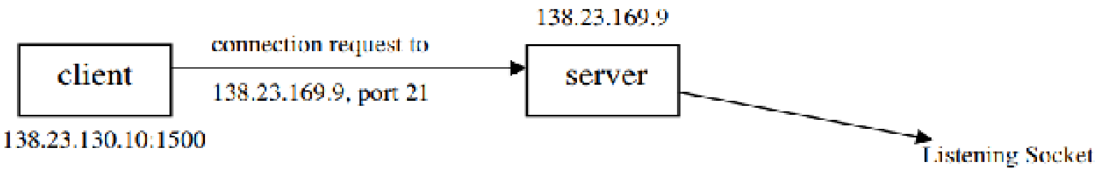
This function returns up to three values:

- An integer return code that is either a new socket descriptor or an error indication,
- The protocol address of the client process (through the *cliaddr* pointer),
- The size of this address (through the *addrlen* pointer).

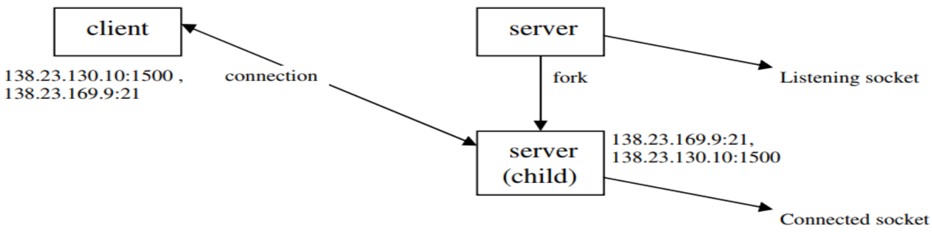
(or)

b) Write a note on Concurrent Servers.

**Ans:** The server can be iterative, i.e. it iterates through each client and serves one request at a time. Alternatively, a server can handle multiple clients at the same time in parallel, and this type of a server is called a concurrent server.



We have already seen an iterative connection-oriented (TCP-implemented) server in the echo client, so now we will focus our attention on the concurrent connection-oriented server. There are several ways we can implement this server: The simplest technique is to call the Unix fork() function. Other techniques are to use threads or to pre-fork a fixed number of children when the function starts. When the server receives and accepts the client's connection, it forks a copy of itself and lets the child handle the client as shown in the figure below:



The listening socket must be distinguished from the connected socket on the server host. Although both sockets use the same local port on the server machine, they are indicated by distinct socket file descriptors, returned server's call of functions socket () and accept () respectively for listening and connected sockets. Notice from figure 3 above that TCP must look at all four segments in the socket pair to determine which endpoint receives an arriving segment. In this figure, there are 3 sockets with the same local port 21 on the server. If a segment arrives from 138.23.130.10 port 1500 destined for 138.23.169.9 port 21, it is delivered to the first child. If a segment arrives from 138.23.130.11 port 1501 destined for 138.23.169.9 port 21, it is delivered to the second child. All other TCP segments destined for port 21 are delivered to the original listening socket. The listening socket must be distinguished from the connected socket on the server host. Although both sockets use the same local port on the server machine, they are indicated by distinct socket file descriptors, returned server's call of functions socket () and accept () respectively for listening and connected sockets.

**Function Description: fork ()** The fork command creates a new separate process for each client. The fork () command splits the current process into two processes: a parent and a child. The new process (child process) is an almost exact copy of the process that calls it (the parent process). The fork() command returns 0 when called in the child process, returns the process ID of the newly created (child) process when called in the parent process, and –1 on error. Therefore, the return value of the function call to fork() tells the process whether it is the parent or the child. For a parent to keep track of its children, it should record the return values from call to fork ().If it is desired to get the process ID of the parent, the child can obtain it by calling getppid command.)

25. a) Explain the functions that support Active Open in a Client.

**Ans:**

**i) socket Function**

To perform network I/O, the first thing a process must do is call the socket function, specifying the type of communication protocol desired (TCP using IPv4, UDP using IPv6, Unix domain stream protocol, etc.).

**int socket (int family, int type, int protocol);**

**/* Returns: non-negative descriptor if OK, -1 on error */**

**Arguments:**

- *family* specifies the protocol family and is one of the constants in the table below. This argument is often referred to as *domain* instead of *family*.
- *Family* = AF_INET, AF_INET6, AF_LOCAL, AF_ROUTE, AF_KEY
- The socket *type* = SOCK_STREAM, SOCK_DGRAM,SOCK_SEQPACKET,SOCK_RAW
- The *protocol* argument to 0 to select the system's default for the given combination of *family* and *type* or to IPPROTO_TCP, IPPROTO_UDP, IPPROTO_SCTP

Not all combinations of socket *family* and *type* are valid.

**connect Function**

The connect function is used by a TCP client to establish a connection with a TCP server.

**#include <sys/socket.h>**

**int connect(int sockfd, const struct sockaddr *servaddr, socklen_t addrlen);**

**/* Returns: 0 if OK, -1 on error */**

- *sockfd* is a socket descriptor returned by the socket function.
- The *servaddr* and *addrlen* arguments are a pointer to a socket address structure (which contains the IP address and port number of the server) and its size.

The client does not have to call bind before calling connect: the kernel will choose both an ephemeral port and the source IP address if necessary.

In the case of a TCP socket, the connect function initiates TCP's three-way handshake. The function returns only when the connection is established or an error occurs. There are several different error returns possible:

1. If the client TCP receives no response to its SYN segment, ETIMEDOUT is returned.
   - For example, in 4.4BSD, the client sends one SYN when connect is called, sends another SYN 6 seconds later, and sends another SYN 24 seconds later. If no response is received after a total of 75 seconds, the error is returned.

    - Some systems provide administrative control over this timeout.

2. If the server's response to the client's SYN is a reset (RST), this indicates that no process is waiting for connections on the server host at the port specified (the server process is probably not running). This is a **hard error** and the error ECONNREFUSED is returned to the client as soon as the RST is received. An RST is a type of TCP segment that is sent by TCP when something is wrong. Three conditions that generate an RST are:
    - When a SYN arrives for a port that has no listening server.
    - When TCP wants to abort an existing connection.
    - When TCP receives a segment for a connection that does not exist.
3. If the client's SYN elicits an ICMP "destination unreachable" from some intermediate router, this is considered a **soft error**. The client kernel saves the message but keeps sending SYNs with the same time between each SYN as in the first scenario. If no response is received after some fixed amount of time (75 seconds for 4.4BSD), the saved ICMP error is returned to the process as either EHOSTUNREACHor ENETUNREACH. It is also possible that the remote system is not reachable by any route in the local system's forwarding table, or that the connect call returns without waiting at all. Note that network unreachables are considered obsolete, and applications should just treat ENETUNREACH andEHOSTUNREACH as the same error.

***Example: nonexistent host on the local subnet ****

We run the client daytimetcpcli and specify an IP address that is on the local subnet (192.168.1/24) but the host ID (100) is nonexistent. When the client host sends out ARP requests (asking for that host to respond with its hardware address), it will never receive an ARP reply.

```
solaris % daytimetcpcli 192.168.1.100
connect error: Connection timed out
```

We only get the error after the connect times out. Notice that our err_sys function prints the human-readable string associated with the ETIMEDOUT error.

***Example: no server process running ****

We specify a host (a local router) that is not running a daytime server:

```
solaris % daytimetcpcli 192.168.1.5
connect error: Connection refused
```

The server responds immediately with an RST.

***Example: destination not reachable on the Internet ****

Our final example specifies an IP address that is not reachable on the Internet. If we watch the packets withtcpdump, we see that a router six hops away returns an ICMP host unreachable error.

```
solaris % daytimetcpcli 192.3.4.5
connect error: No route to host
```

As with the ETIMEDOUT error, connect returns the EHOSTUNREACH error only after waiting its specified amount of time.

In terms of the TCP state transition diagram:

- connect moves from the CLOSED state (the state in which a socket begins when it is created by thesocket function) to the SYN_SENT state, and then, on success, to the ESTABLISHED state.
- If connect fails, the socket is no longer usable and must be closed. We cannot call connect again on the socket.

(or)

b) Write a note on fork and exec functions

**Ans:** fork() is a function that is called once and it returns two integer value (Successful execution) one from the running process and other from the child newly created from the running process. The following is the syntax of fork ()

```
#include <unistd.h>
pid_t fork(void);
```

**Returns: 0 in child, process ID of child in parent, -1 on error**

The reason fork returns 0 in the child, instead of the parent's process ID, is because a child has only one parent and it can always obtain the parent's process ID by calling getppid. A parent, on the other hand, can have any number of children, and there is no way to obtain the process IDs of its children. If a parent wants to keep track of the process IDs of all its children, it must record the return values from fork.

All descriptors open in the parent before the call to fork are shared with the child after fork returns. We will see this feature used by network servers: The parent calls accept and then calls fork. The connected socket is then shared between the parent and child. Normally, the child then reads and writes the connected socket and the parent closes the connected socket.

There are two typical uses of fork:

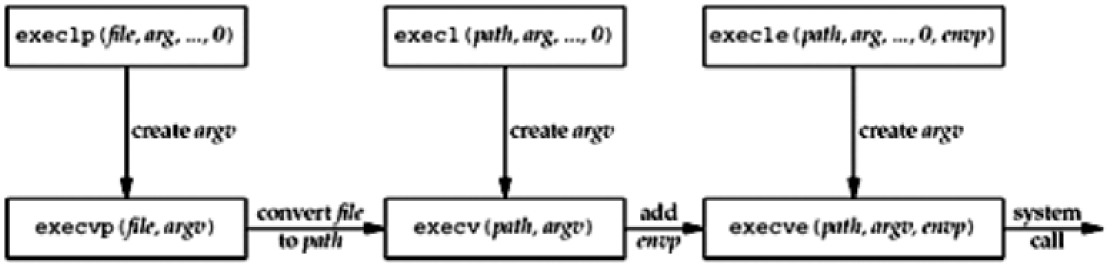
1. A process makes a copy of itself so that one copy can handle one operation while the other copy does another task. This is typical for network servers. We will see many examples of this later in the text.

2. A process wants to execute another program. Since the only way to create a new process is by calling fork, the process first calls fork to make a copy of itself, and then one of the copies (typically the child process) calls exec (described next) to replace itself with the new program. This is typical for programs such as shells.

The only way in which an executable program file on disk can be executed by Unix is for an existing process to call one of the six exec functions. (We will often refer generically to "the exec function" when it does not matter which of the six is called.) exec replaces the current process image with the new program file, and this new program normally starts at the main function. The process ID does not change. We refer to the process that calls exec as the calling process and the newly executed program as the new program.

| |
|---|
| #include <unistd.h> |
| int execl (const char *pathname, const char *arg0, ... /* (char *) 0 */ ); |
| int execv (const char *pathname, char *const argv[]); |
| int execle (const char *pathname, const char *arg0, ... |
| /* (char *) 0, char *const envp[] */ ); |
| int execve (const char *pathname, char *const argv[], char *const envp[]); |
| int execlp (const char *filename, const char *arg0, ... /* (char *) 0 */ ); |
| int execvp (const char *filename, char *const argv[]); |
| All six return: -1 on error, no return on success |

These functions return to the caller only if an error occurs. Otherwise, control passes to the start of the new program, normally the main function.



Note the following differences among these six functions:

1. The three functions in the top row specify each argument string as a separate argument to the exec function, with a null pointer terminating the variable number of arguments. The three functions in the second row have an argv array, containing

pointers to the argument strings. This argv array must contain a null pointer to specify its end, since a count is not specified.

2. The two functions in the left column specify a filename argument. This is converted into a pathname using the current PATH environment variable. If the filename argument to execlp or execvp contains a slash (/) anywhere in the string, the PATH variable is not used. The four functions in the right two columns specify a fully qualified pathname argument.
3. The four functions in the left two columns do not specify an explicit environment pointer. Instead, the current value of the external variable environ is used for building an environment list that is passed to the new program. The two functions in the right column specify an explicit environment list. The envp array of pointers must be terminated by a null pointer.

26. a) Write a c program that simulates the TCP/IP Echo server.

**Ans:**

**Server Program:**

```
#include<netinet/in.h>
#include<sys/socket.h>
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
main()
{
        int sockfd, acceptid;
        char buffer[50]="";
        struct sockaddr_in saddr;
        bzero(&saddr,sizeof(saddr));
        saddr.sin_family=AF_INET;
        saddr.sin_addr.s_addr=htons(INADDR_ANY);
        saddr.sin_port=htons(1500);
        sockfd=socket(AF_INET,SOCK_STREAM,0);
        bind(sockfd,(struct sockaddr*)&saddr,sizeof(saddr));
        listen(sockfd,5);
        acceptid=accept(sockfd, (struct sockaddr*) NULL, NULL);
        while(1)
        {
                read(acceptid,buffer,sizeof(buffer));
                printf("Read %s\n",buffer);
```

```
		write(acceptid,buffer,sizeof(buffer));
	}
}
```

**Client Program:**

```
#include<netinet/in.h>
#include<sys/socket.h>
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
main()
{
	int sockfd, acceptid;
	char buffer[50]="";
	struct sockaddr_in caddr;
	int i;
	bzero(&caddr,sizeof(caddr));
	caddr.sin_family=AF_INET;
	caddr.sin_addr.s_addr=htons(INADDR_ANY);
	caddr.sin_port=htons(1500);
	sockfd=socket(AF_INET,SOCK_STREAM,0);
	connect(sockfd,(struct sockaddr*)&caddr,sizeof(caddr));
	for(i=0;i<=4;i++)
	{
		puts("Enter the string to echo");
		gets(buffer);
		write(sockfd,buffer,sizeof(buffer));
		read(sockfd,buffer,sizeof(buffer));
		printf("Echoing Back %s\n",buffer);
	}
}
```

(or)

b) Write a detail note on a) getsockname b) getpeername c) signal handling

**Ans:**

**getsockname and getpeername Functions**

- getsockname returns the local protocol address associated with a socket.
- getpeername returns the foreign protocol address associated with a socket.

```
#include <sys/socket.h>
int getsockname(int sockfd, struct sockaddr *localaddr, socklen_t *addrlen);
int getpeername(int sockfd, struct sockaddr *peeraddr, socklen_t *addrlen);
/* Both return: 0 if OK, -1 on error */
```

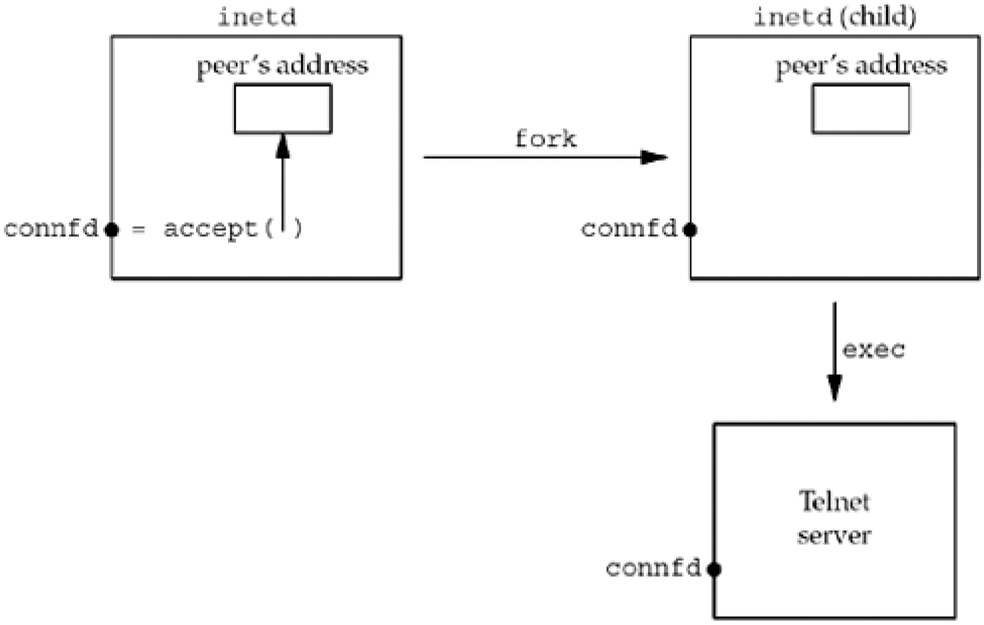
The *addrlen* argument for both functions is value-result argument: both functions fill in the socket address structure pointed to by localaddr or peeraddr.

The term "name" in the function name is misleading. These two functions return the protocol address associated with one of the two ends of a network connection, which for IPV4 and IPV6 is the combination of an IP address and port number. These functions have nothing to do with domain names.

These two functions are required for the following reasons:

- After connect successfully returns in a TCP client that does not call bind, getsockname returns the local IP address and local port number assigned to the connection by the kernel.
- After calling bind with a port number of 0 (telling the kernel to choose the local port number),getsockname returns the local port number that was assigned.
- getsockname can be called to obtain the address family of a socket.
- In a TCP server that binds the wildcard IP address, once a connection is established with a client (accept returns successfully), the server can call getsockname to obtain the local IP address assigned to the connection. The socket descriptor argument to getsocknamemust be that of the connected socket, and not the listening socket.
- When a server is execed by the process that calls accept, the only way the server can obtain the identity of the client is to call getpeername. For example, inetd forks and execs a TCP server (follwing figure):
    - inetd calls accept, which return two values: the connected socket descriptor (connfd, return value of the function) and the "peer's address" (an Internet socket address structure) that contains the IP address and port number of the client.
    - fork is called and a child of inetd is created, with a copy of the parent's memory image, so the socket address structure is available to the child, as is the connected socket descriptor (since the descriptors are shared between the parent and child).
    - When the child execs the real server (e.g. Telnet server that we show), the memory image of the child is replaced with the new program file for the Telnet server (the socket address structure containing the peer's

address is lost), and the connected socket descriptor remains open across the exec. One of the first function calls performed by the Telnet server is getpeername to obtain the IP address and port number of the client.



In this example, the Telnet server must know the value of connfd when it starts. There are two common ways to do this.

1. The process calling exec pass it as a command-line argument to the newly execed program.
2. A convention can be established that a certain descriptor is always set to the connected socket before calling exec.

   The second one is what inetd does, always setting descriptors 0, 1, and 2 to be the connected socket.

**Reg.No ---------------------**

**[**16ITU502B**]**

**KARPAGAM ACADEMY OF HIGHER EDUCATION**

(Deemed to be University)

(Established Under Section 3 of UGC Act 1956)

COIMBATORE – 641 021

**B.Sc. DEGREE EXAMINATION**

**Fifth Semester - III Internal Examination**

**Information Technology**

**Network Programming**

**Time: 2 Hrs** **Max.Marks: 50**

**Date :** **Class: III B.Sc (IT) - 'A' & 'B'**

---

**Part - A** **(10 X 1 = 20)**

**Answer all the Questions**

1. Telnet is used for _____ environment.

   a) Time Sharing b) Distributed c) Cloud d) all

2. Telnet refers to _____

   a) TeleNetworking b) TelephoneNetworking c) TerminalNetwork d) None

3. NVT refers to ______

   a) Network Virtualization Terminal b) Network Virtual Terminal

   c) Network Visual Terminal d) Network Virtual Task

4. UA refers to ______

   a) User Agent b) User Assistance c) Unique Agent d) Unique Assistance

5. Email address consists of ___parts

   a) 2 b) 3 c) 4 d) 5

6. SMTP is used with ______

   a) Management b) Mail Transfer c) Message Transfer d) Both a & b

7. NOP refers to ____

   a) No Operand b) No Operations c) No Operator d) Network Operation

8. Which of the following protocols is supported in mail transferring?

   a) POP3 b) IMAP4 c) SMTP d) all

9. There are ____type of web Documents.

   a) 2 b) 3 c) 4 d) 5

10. ______ is a type of Active Document.

    a) Java Applet b)Web Page c) Html Document d) None

11. Which of the following refers to HTTP Port Number?

    a) 80 b) 81 c) 8080 d) 8181

12. Which of the following command is used for configuring Static route in LINUX?

    b) routing b) ifconfig c) route d)mii-tool

13. Which of the following is alternate of stateless entity?

a) Cookies b) Providing DB c) Setting Environment Variable d) Both a & C

14. Https uses ____as its Port number

a) 80 b) 81 c) 13 d) 15

15. There are ___types of classes in Classful addressing

a) 6 b) 3 c) 4 d) 5

16. The ____protocol is used for Static Host Configuration.

a) BOOTP b) DHCP c) SMTP d)SNMP

17. Process to Process communication is done using ____.

a) Socket b) Port Number c) IP number d) Socket FD

18. EC option in TELNET refers to______

a) Erase Character b) Enrich Character c) Erase Colour d)Enrich Colour

19. The ____protocol is used for Dynamic Host Configuration.

a) BOOTP b) DHCP c) SMTP d)SNMP

20. ____is the command used to check if the host is alive or not?

a) traceroute b) ping c) communicate d) tcpdump

**Part - B** **(3 X 2 = 6)**

**Answer all the Questions**

21. List the types of Web Documents.
22. What is RARP used for?
23. What is the use of ping command?

**Part - C** **(3 X 8 = 24)**

**Answer all the Questions**

24. a) Write a detail note on TELNET.

(or)

b) Write a java program that simulates day-time server operation.

25. a) Discuss about Email scenario in detail.

(or)

b) Write a java program that simulates File Copy operation between Machines using socket.

26. a) Elaborate Address Resolution Protocol.

(or)

b) Discuss about DHCP in general.

**Reg.No ---------------------**

**[**16ITU502B**]**

**KARPAGAM ACADEMY OF HIGHER EDUCATION**

(Deemed to be University)

(Established Under Section 3 of UGC Act 1956)

COIMBATORE – 641 021

**B.Sc. DEGREE EXAMINATION**

**Fifth Semester - III Internal Examination**

**Information Technology**

**Network Programming**

**Time: 2 Hrs** **Max.Marks: 50**

**Date : 08/10/18** **Class: III B.Sc (IT) - 'A' & 'B'**

---

**Part - A** **(10 X 1 = 20)**

**Answer all the Questions**

1. Telnet is used for _____ environment.

   a) **Time Sharing** b) Distributed c) Cloud d) all

2. Telnet refers to _____

   a) TeleNetworking b) TelephoneNetworking c) **TerminalNetwork** d) None

3. NVT refers to ______

   a) Network Virtualization Terminal b) **Network Virtual Terminal**

   c) Network Visual Terminal d) Network Virtual Task

4. UA refers to ______

   a) **User Agent** b) User Assistance c) Unique Agent d) Unique Assistance

5. Email address consists of ___parts

   a) 2 b) **3** c) 4 d) 5

6. SMTP is used with ______

   a) Management b) **Mail Transfer** c) Message Transfer d) Both a & b

7. NOP refers to ____

   a) No Operand b) **No Operations** c) No Operator d) Network Operation

8. Which of the following protocols is supported in mail transferring?

   **a)** POP3 b) IMAP4 c) SMTP d) **all**

9. There are ____type of web Documents.

   a) 2 b) **3** c) 4 d) 5

10. ______ is a type of Active Document.

    a) **Java Applet** b)Web Page c) Html Document d) None

11. Which of the following refers to HTTP Port Number?

    a) **80** b) 81 c) 8080 d) 8181

12. Which of the following command is used for configuring Static route in LINUX?

    b) routing b) **ifconfig** c) route d)mii-tool

13. Which of the following is alternate of stateless entity?

    a) Cookies b) Providing DB c) Setting Environment Variable d) **Both a & C**

14. Https uses ____as its Port number

    a) 80 b) **81** c) 13 d) 15

15. There are ___types of classes in Classful addressing

    a) 6 b) 3 c) 4 d) **5**

16. The ____protocol is used for Static Host Configuration.

    a) **BOOTP** b) DHCP c) SMTP d)SNMP

17. Process to Process communication is done using ____.

    a) Socket b) **Port Number** c) IP number d) Socket FD

18. EC option in TELNET refers to______

    a) **Erase Character** b) Enrich Character c) Erase Colour d)Enrich Colour

19. The ____protocol is used for Dynamic Host Configuration.

    a) BOOTP b) **DHCP** c) SMTP d)SNMP

20. ____is the command used to check if the host is alive or not?

    a) traceroute b) **ping** c) communicate d) tcpdump

**Part - B** **(3 X 2 = 6)**

**Answer all the Questions**

21. List the types of Web Documents.

    **Ans:** Static, Dynamic and Active Documents

22. What is RARP used for?

    **Ans:** This is the protocol used to find the IP-Address by providing the physical Address

23. What is the use of ping command?

    **Ans:** The command used to check if the remote host is in live or alive state.

**Part - C** **(3 X 8 = 24)**

**Answer all the Questions**

24. a) Write a detail note on TELNET.

    **Ans:** TELNET is an abbreviation for TErminaL NETwork. It is the standard TCP/IP protocol for virtual terminal service as proposed by ISO. TELNET enables the establishment of a connection to a remote system in such a way that the local terminal appears to be a terminal at the remote system.

    **Time-Sharing Environment**

    TELNET was designed at a time when most operating systems, such as UNIX, were operating in a time-sharing environment. In such an environment, a large computer supports multiple users. The interaction between a user and the computer occurs through a terminal, which is usually a combination of keyboard, monitor, and mouse. Even a microcomputer can simulate a terminal with a terminal emulator. In a time-sharing environment, all of the processing must be done by the central computer. When a user types a character on the keyboard, the character is usually sent to the computer and echoed to the monitor.

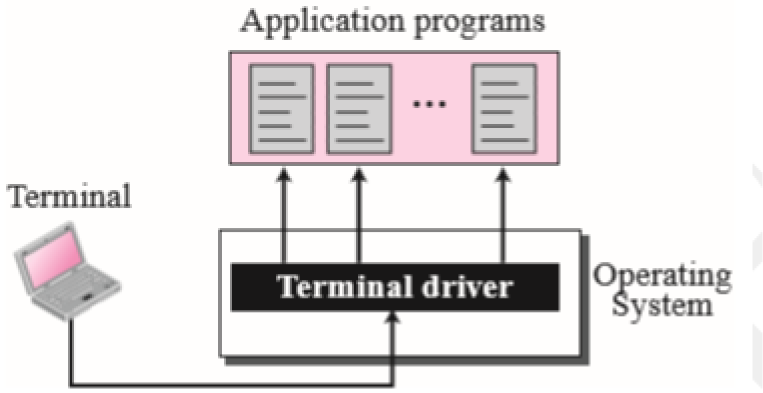
**Login**

In a time-sharing environment, users are part of the system with some right to access resources. Each authorized user has an identification and probably a password. The user identification defines the user as part of the system. To access the system, the user logs into the system with a user id or login name. The system also includes password checking to prevent an unauthorized user from accessing the resources.

**Local Login**

When a user logs into a local time-sharing system, it is called local login. 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.

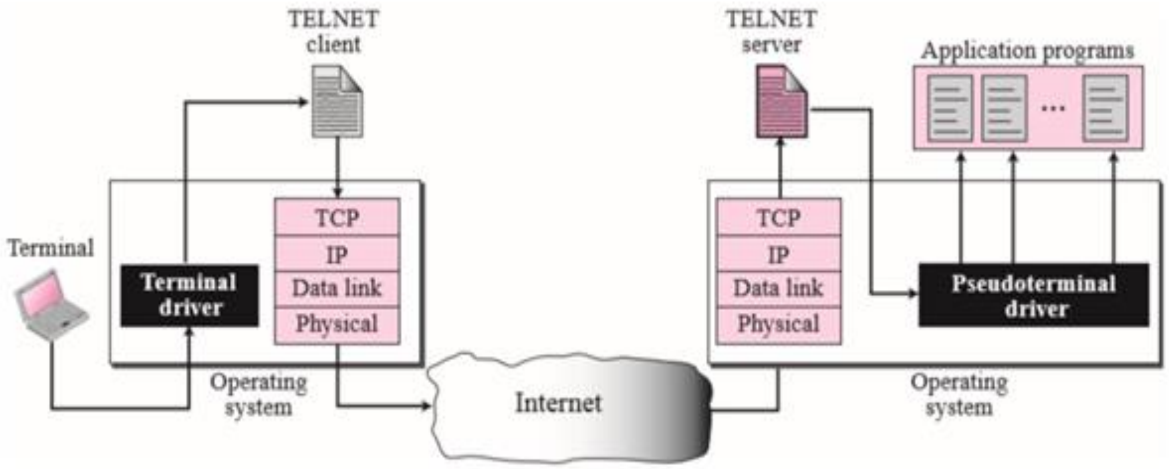
The mechanism, however, is not as simple as it seems because the operating system may assign special meanings to special characters. For example, in UNIX some combinations of characters have special meanings, such as the combination of the control character with the character z, which means suspend; the combination of the control character with the character c, which means abort; and so on.



**Local Login**

**Remote Login**

When a user wants to access an application program or utility located on a remote machine, he or she performs remote login. 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 to a universal character set called Network Virtual Terminal (NVT) characters and delivers them to the local TCP/IP stack

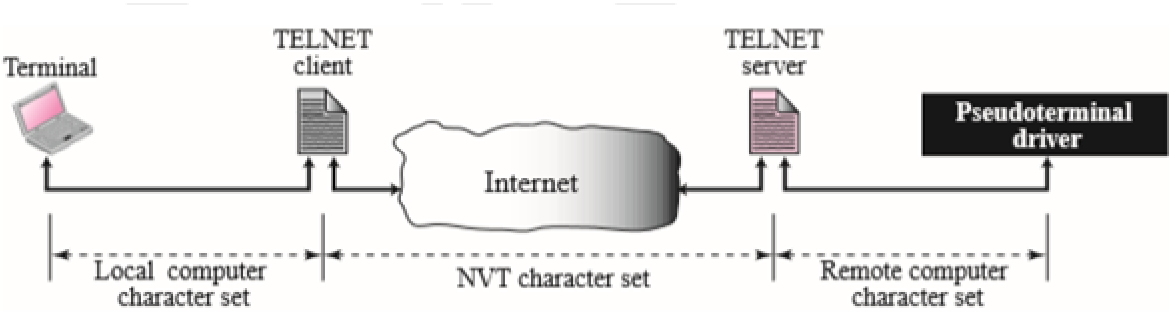


**Remote Login**

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 pseudo-terminal driver, which pretends that the characters are coming from a terminal. The operating system then passes the characters to the appropriate application program.

**Network Virtual Terminal (NVT)**

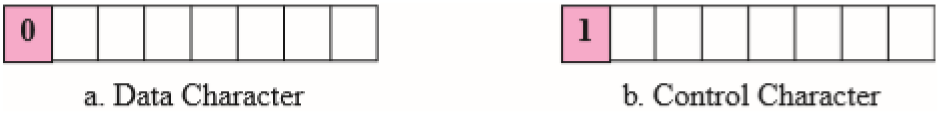
The mechanism to access a remote computer is complex. This is because every computer and its operating system accepts a special combination of characters as tokens. For example, the end-of-file token in a computer running the DOS operating system is Ctrl+z, while the UNIX operating system recognizes Ctrl+d. We are dealing with heterogeneous systems. If we want to access any remote computer in the world, we must first know what type of computer we will be connected to, and we must also install the specific terminal emulator used by that computer. TELNET solves this problem by defining a universal interface called the Network Virtual Terminal (NVT) character set. Via this interface, the client TELNET translates characters (data or commands) that come from the local terminal into NVT form and delivers them to the network. The server TELNET, on the other hand, translates data and commands from NVT form into the form acceptable by the remote computer.



**Concept of NVT**

### NVT Character Set

NVT uses two sets of characters, one for data and one for control. Both are 8-bit bytes.

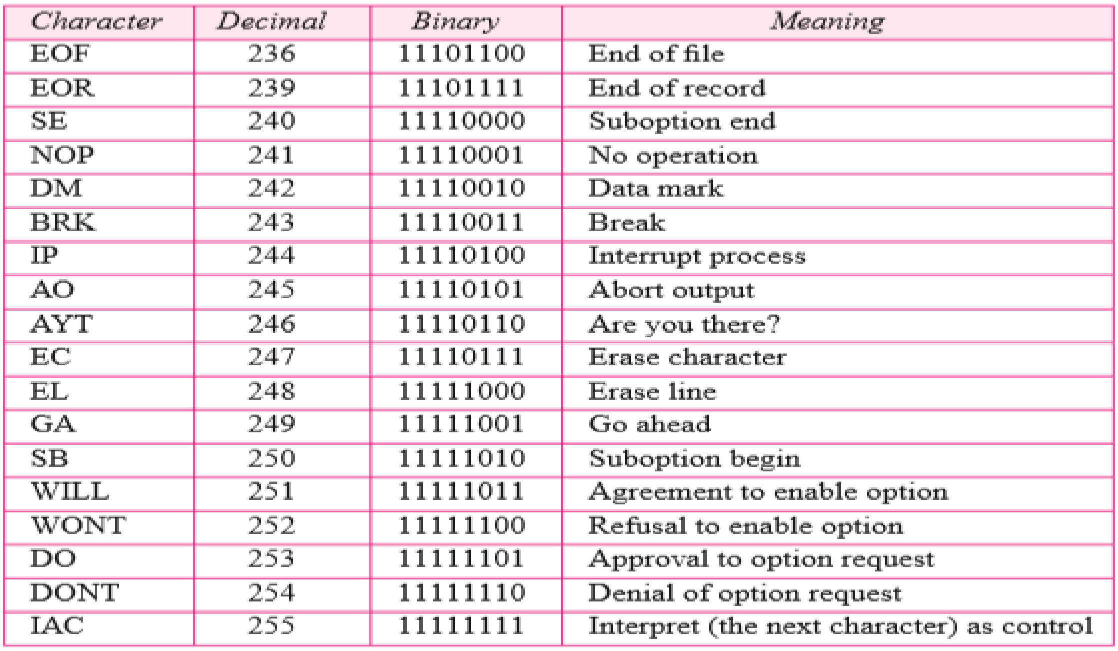


### Data Characters

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. Although it is possible to send an 8-bit ASCII (with the highest order bit set to be 0 or 1), this must first be agreed upon between the client and the server using option negotiation.

### Control Characters

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. The below lists some of the control characters and their meanings.

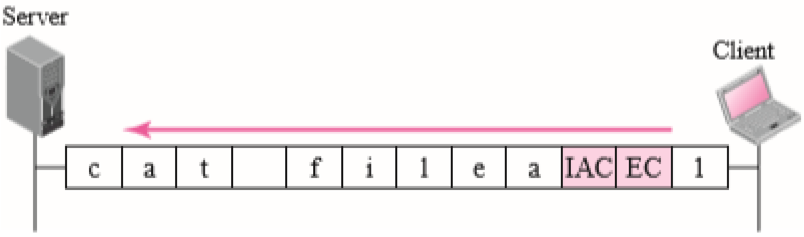


| *Character* | *Decimal* | *Binary* | *Meaning* |
|---|---|---|---|
| EOF | 236 | 11101100 | End of file |
| EOR | 239 | 11101111 | End of record |
| SE | 240 | 11110000 | Suboption end |
| NOP | 241 | 11110001 | No operation |
| DM | 242 | 11110010 | Data mark |
| BRK | 243 | 11110011 | Break |
| IP | 244 | 11110100 | Interrupt process |
| AO | 245 | 11110101 | Abort output |
| AYT | 246 | 11110110 | Are you there? |
| EC | 247 | 11110111 | Erase character |
| EL | 248 | 11111000 | Erase line |
| GA | 249 | 11111001 | Go ahead |
| SB | 250 | 11111010 | Suboption begin |
| WILL | 251 | 11111011 | Agreement to enable option |
| WONT | 252 | 11111100 | Refusal to enable option |
| DO | 253 | 11111101 | Approval to option request |
| DONT | 254 | 11111110 | Denial of option request |
| IAC | 255 | 11111111 | Interpret (the next character) as control |

**Embedding**

TELNET uses only one TCP connection. The server uses the well-known port 23 and the client uses an ephemeral port. The same connection is used for sending both data and control characters. TELNET accomplishes this by embedding the control characters in the data stream. However, to distinguish data from control characters, each sequence of control characters is preceded by a special control character called interpret as control (IAC).

For example, imagine a user wants a server to display a file (file1) on a remote server he uses “cat file1” command. However, the name of the file has been mistyped (filea instead of file1). The user uses the backspace key to correct this situation.” cat filea<backspace>1”. However, in the default implementation of TELNET, the user cannot edit locally; the editing is done at the remote server. The backspace character is translated into two remote characters (IAC EC), which is embedded in the data and sent to the remote server. What is sent to the server is shown in Figure.



**Embedding**

(or)

b) Write a java program that simulates day-time server operation.

**Ans:**

**Server**

```
import java.io.*;
import java.net.*;
import java.util.Date;
class DayTimeServer
{
    public static void main(String str[]) throws IOException
    {
            ServerSocket sSock = new ServerSocket(1999);
            Socket sock = sSock.accept();
            DataInputStream in = new DataInputStream(sock.getInputStream());
            DataOutputStream out = new DataOutputStream(sock.getOutputStream());
            Date d= new Date();
            out.writeUTF(d.toString());
```

```
                sock.close();
                sSock.close();
        }
}
```

**Client**

```
import java.io.*;
import java.net.*;
import java.util.Date;
class DayTimeClient
{
        public static void main(String str[]) throws IOException
        {
                Socket sock = new Socket("localhost",1999);
                DataInputStream in = new DataInputStream(sock.getInputStream());
                String s = in.readUTF();
                System.out.println(s);
                sock.close();
        }
}
```

25. a) Discuss about Email scenario in detail.

**Ans: Email**

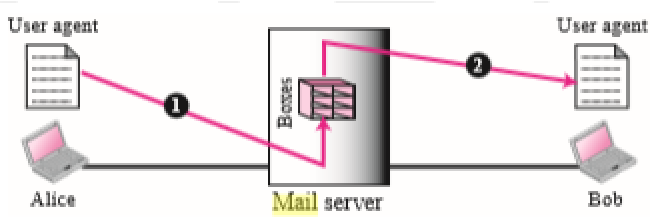
One of the most popular Internet services is electronic mail (e-mail). At the beginning of the Internet era, the messages sent by electronic mail were short and consisted of text only; they let people exchange quick memos. Today, electronic mail is much more complex. It allows a message to include text, audio, and video. It also allows one message to be sent to one or more recipients.

**Architecture**

To explain the architecture of e-mail, we give four scenarios. We begin with the simplest situation and add complexity as we proceed. The fourth scenario is the most common in the exchange of e-mail.

**First Scenario**

In the first scenario, the sender and the receiver of the e-mail are users (or application programs) on the same mail server; they are directly connected to a shared mail server. The administrator has created one mailbox for each user where the received messages are stored. A mailbox is part of a local hard drive, a special file with permission restrictions. Only the owner of the mailbox has access to it. When Alice needs to send a message to Bob, she runs a user agent (UA) program to prepare the message and store it in Bob's mailbox. The message has the sender and recipient mailbox addresses (names of files). Bob can retrieve and read the contents of his mailbox at his convenience using a user agent. The following figure shows the scenario.

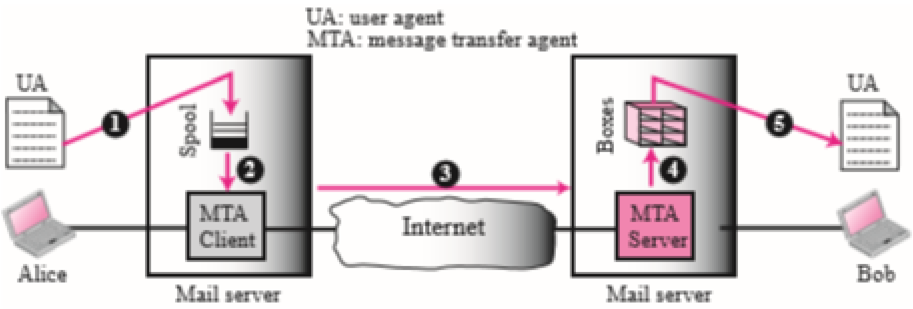


First Scenario

**Second Scenario**

In the second scenario, the sender and the receiver of the e-mail are users (or application programs) on two different mail servers. The message needs to be sent over the Internet. Here we need user agents (UAs) and message transfer agents (MTAs) as shown in Figure.

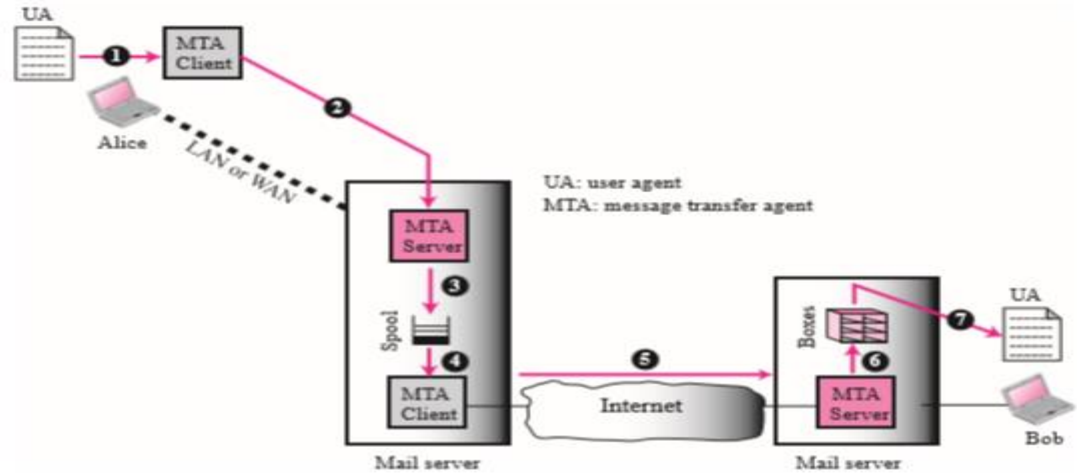
Alice needs to use a user agent program to send her message to the mail server at her own site. The mail server at her site uses a queue (spool) to store messages waiting to be sent. Bob also needs a user agent program to retrieve messages stored in the mail- box of the system at his site. The message, however, needs to be sent through the Inter- net from Alice's site to Bob's site. 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 of 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.



Second Scenario

**Third Scenario**

The Following Figure shows the third scenario. Bob, as in the second scenario, is directly connected to his mail server. Alice, however, is separated from her mail server. Alice is either connected to the mail server via a point-to-point WAN—such as a dial-up modem, a DSL, or a cable modem—or she is connected to a LAN in an organization that uses one mail server for handling e-mails; all users need to send their messages to this mail server.

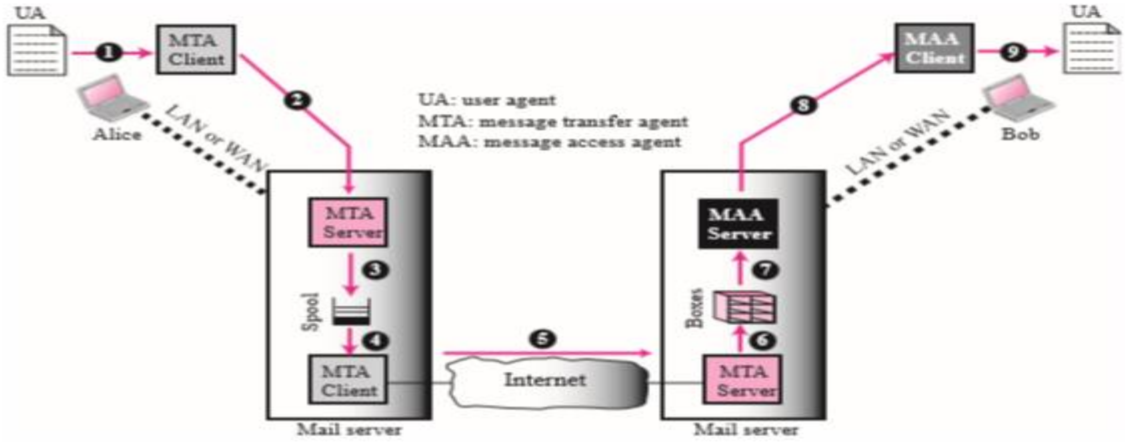


### Third Scenario

Alice still needs a user agent to prepare her message. She then needs to send the message through the LAN or WAN. This can be done through a pair of message transfer agents (client and server). Whenever Alice has a message to send, she calls the user agent which, in turn, calls the MTA client. The MTA client establishes a connection with the MTA server on the system, which is running all the time. The system at Alice's site queues all messages received. It then uses an MTA client to send the messages to the system at Bob's site; the system receives the message and stores it in Bob's mailbox.

### Fourth Scenario

In the fourth and most common scenario, Bob is also connected to his mail server by a WAN or a LAN. After the message has arrived at Bob's mail server, Bob needs to retrieve it. Here, we need another set of client-server agents, which we call message access agents (MAAs). Bob uses an MAA client to retrieve his messages. The client sends a request to the MAA server, which is running all the time, and requests the transfer of the messages. The situation is shown in Figure 23.4. There are two important points we need to emphasize here.



First, Bob cannot bypass the mail server and use the MTA server directly. To use the MTA server directly, Bob would need to run the MTA server all the time because he does not know when a message will arrive. This implies that Bob must keep his computer on all the time if he is connected to his system through a LAN. If he is connected through a WAN, he must keep the connection up all the time. Neither of these situations is feasible today. Second, note that Bob needs another pair of client-server programs: message access programs. This is because an MTA client-server program is a push program: the client pushes the message to the server. Bob needs a pull program. The client needs to pull the message from the server.

(or)

b) Write a java program that simulates File Copy operation between Machines using socket.

**Ans**

**Server**

```
import java.io.*;
import java.net.*;
class FileServer
{
        public static void main(String[] args)throws Exception
        {
                ServerSocket sSock = new ServerSocket(1999);
                Socket sock = sSock.accept();
                File file = new File("C:\\Users\\Home User\\Desktop\\FileServer.java");
                BufferedReader br = new BufferedReader(new FileReader(file));
                DataOutputStream out = new DataOutputStream(sock.getOutputStream());
                String st;
                while ((st = br.readLine()) != null)
                {
                        out.writeUTF(st);
                }
                out.writeUTF("exit");
                sock.close();
                sSock.close();
         }
}
```

**Client**

```
import java.io.*;
import java.net.*;
class FileClient
{
        public static void main(String[] args)throws Exception
```

```
    {
        Socket sock = new Socket("localhost",1999);
        DataInputStream in = new DataInputStream(sock.getInputStream());
        String st="";
        while(!st.equals("exit"))
        {
            st = in.readUTF();
            System.out.println(st);
        }
        sock.close();
    }
}
```

26. a) Elaborate Address Resolution Protocol.

**Ans: ARP (Address Resolution Protocol)**

ARP, the Address Resolution Protocol, discovers the hardware address associated with a particular IP address. It can be used on any kind of network that supports broadcasting but is most commonly described in terms of Ethernet. If host A wants to send a packet to host B on the same Ethernet, it uses ARP to discover B’s hardware address. If B is not on the same network as A, host A uses the routing system to determine the next-hop router along the route to B and then uses ARP to find that router’s hardware address. Since ARP uses broadcast packets, which cannot cross networks, it can only be used to find the hardware addresses of machines directly connected to the sending host’s local network. Every machine maintains a table in memory called the ARP cache, which contains the results of recent ARP queries. Under normal circumstances, many of the addresses a host needs are discovered soon after booting, so ARP does not account for a lot of network traffic.

The **arp** command examines and manipulates the kernel’s ARP cache, adds or deletes entries, and flushes or shows the table. The command **arp -a** displays the contentsof the ARP cache. The **arp** command is generally useful only for debugging and for situations that involve special hardware. Some devices are not smart enough to speak ARP. To support such devices, you might need to configure another machine as a proxy ARP server for your crippled hardware. That’s normally done with the **arp** command as well

**ADDITION OF A MACHINE TO A NETWORK**

Only a few steps are involved in adding a new machine to an existing local area network, but some vendors hide the files you must modify and generally make the chore difficult. Others provide a setup script that prompts for the networking parameters that are needed, which is fine until you need to undo something or move a machine. Before bringing up a new machine on a network that is connected to the Internet, you should secure it so that you are not inadvertently inviting hackers onto your local network.

The basic steps to add a new machine to a local network are as follows:

- Assign a unique IP address and hostname.
- Set up the new host to configure its network interfaces at boot time.
- Set up a default route and perhaps fancier routing.
- Point to a DNS name server, to allow access to the rest of the Internet.

Of course, you could add a debugging step to this sequence as well. After any change that might affect booting, you should always reboot to verify that the machine comes up correctly. If your network uses DHCP, the Dynamic Host Configuration Protocol, the DHCP server will do these chores for you.

**ifconfig: configure network interfaces**

**ifconfig** enables or disables a network interface, sets its IP address and subnet mask, and sets various other options and parameters. It is usually run at boot time but it can also make changes on the fly. Be careful if you are making **ifconfig** changes and are logged in remotely; many a sysadmin has been locked out this way and had to drive in to fix things. An **ifconfig** command most commonly has the form

**ifconfig** *interface address options …*

for example:

**ifconfig eth0 192.168.1.13 netmask 255.255.255.0 up**

**ifconfig** *interface* displays the current settings for *interface* without changing them. Many systems understand **-a to** mean "all interfaces," and **ifconfig -a** can therefore be used to find out what interfaces are present on the system.

**route: configure static routes**

The **route** command defines static routes, explicit routing table entries that never change (you hope), even if you run a routing daemon. When you add a new machine to a local area network, you usually need to specify only a default route;

**Default routes**

A default route causes all packets whose destination network is not found in the kernel's routing table to be sent to the indicated gateway. To set a default route, simply add the following line to your startup files:

**route add default gw** *gateway-IP-address*

Rather than hard coding an explicit IP address into the startup files, most vendors have their systems get the gateway IP address from a configuration file. The way that local routing information is integrated into the startup sequence is unfortunately different for each of our Linux systems

**DNS configuration**

To configure a machine as a DNS client, you need to edit only one or two files: all systems require **/etc/resolv.conf** to be modified, and some require you to modify a "service switch" file as well. The **/etc/resolv.conf** file lists the DNS domains that should be searched to resolve names that are incomplete (that is, not fully qualified, such as anchor instead of anchor. cs.colorado.edu) and the IP addresses of the name servers to contact for name lookups.

**Security Issues**

**IP forwarding**

A UNIX or Linux system that has IP forwarding enabled can act as a router. That is, it can accept third-party packets on one network interface, match them to a gateway or destination host on another interface, and retransmit the packets. Unless your system has multiple network interfaces and is actually supposed to function as a router, it's advisable to turn this feature off. Hosts that forward packets can sometimes be coerced into compromising security by making external packets appear to have come from inside your network. This subterfuge can help an intruder's packets evade network scanners and packet filters. It is perfectly acceptable for a host to use multiple network interfaces for its own traffic without forwarding third-party traffic.

**ICMP redirects**

ICMP redirects can maliciously reroute traffic and tamper with your routing tables. Most operating systems listen to ICMP redirects and follow their instructions by default. It would be bad if all your traffic were rerouted to a competitor's network for a few hours, especially while backups were running. In such case configure your routers (and hosts acting as routers) to ignore and perhaps log ICMP redirect attempts.

**Source routing**: It was part of the original IP specification; it was intended primarily to facilitate testing. It can create security problems because packets are often filtered according to their origin. If someone can cleverly route a packet to make it appear to have originated within your network instead of the Internet, it might slip through your firewall. We recommend that you neither accept nor forward source-routed packets.

**Broadcast pings and other directed broadcasts**

Ping packets addressed to a network's broadcast address (instead of to a particular host address) are typically delivered to every host on the network. Such packets have been used in denial of service attacks; for example, the so-called Smurf attacks. (The "Smurf attacks" Wikipedia article has details.) Broadcast pings are a form of "directed broadcast" in that they are packets sent to the broadcast address of a distant network. The default handling of such packets has been gradually changing.

**IP spoofing**

The source address on an IP packet is normally filled in by the kernel's TCP/IP implementation and is the IP address of the host from which the packet was sent. However, if the software creating the packet uses a raw socket, it can fill in any source address it likes. This is called IP spoofing and is usually associated with some kind of malicious network behaviour. The machine identified by the spoofed source IP address (if it is a real address at all) is often the victim in the scheme. Error and return packets can disrupt or flood the victim's network connections. You should deny IP spoofing at your border router by blocking outgoing packets whose source address is not within your address space. This precaution is especially important if your site is a university where students like to experiment and may be tempted to carry out digital vendettas.

**Host-based firewalls**

Traditionally, a network packet filter or firewall connects your local network to the outside world and controls traffic according to a site-wide policy. The last few Windows releases all come with their own personal firewalls, and they complain bitterly if you try to turn the firewall off. Our example systems all include packet filtering software, but you should not infer from this that every UNIX or Linux machine needs its own firewall. It does not. The packet filtering features are there to allow these machines to serve as network gateways.

**Virtual private networks** Many organizations that have offices in several locations would like to have all those locations connected to one big private network. Such organizations can use the Internet as if it were a private network by establishing a series of secure, encrypted "tunnels" among their various locations. A network that includes such tunnels is known as a virtual private network or VPN. VPN facilities are also needed when employees must connect to your private network from their homes or from the field. A VPN system doesn't eliminate every possible security issue relating to such ad hoc connections, but it's secure enough for many purposes.

**PPP: The Point to Point Protocol**

PPP represents an underlying communication channel as a virtual network interface. However, since the underlying channel need not have any of the features of an actual network, communication is restricted to the two hosts at the ends of the link—a virtual network of two. PPP has the distinction of being used on both the slowest and the fastest IP links, but for different reasons. In its asynchronous form, PPP is best known as the protocol used to provide dialup Internet service over phone lines and serial links. These channels are not inherently packet oriented, so the PPP device driver encodes network packets into a unified data stream and adds link-level headers and markers to separate packets. In its synchronous form, PPP is the encapsulation protocol used on high-speed circuits that have routers at either end. It's also commonly used as part of the implementation of DSL and cable modems for broadband service. In these latter situations, PPP not only converts the underlying network system (often ATM in the case of DSL) to an IP-friendly form, but it also provides authentication and access control for the link itself. In addition to specifying how the link is established, maintained, and torn down, PPP implements error checking, authentication, encryption, and compression. These features make it adaptable to a variety of situations.

(or)

b) Discuss about DHCP in general.

**Ans: ADDITION OF A MACHINE TO A NETWORK**

Only a few steps are involved in adding a new machine to an existing local area network, but some vendors hide the files you must modify and generally make the chore difficult. Others provide a setup script that prompts for the networking parameters that are needed, which is fine until you need to undo something or move a machine. Before bringing up a new machine on a network that is connected to the Internet, you should secure it so that you are not inadvertently inviting hackers onto your local network.

The basic steps to add a new machine to a local network are as follows:

- Assign a unique IP address and hostname.
- Set up the new host to configure its network interfaces at boot time.
- Set up a default route and perhaps fancier routing.
- Point to a DNS name server, to allow access to the rest of the Internet.

Of course, you could add a debugging step to this sequence as well. After any change that might affect booting, you should always reboot to verify that the machine comes up correctly. If your network uses DHCP, the Dynamic Host Configuration Protocol, the DHCP server will do these chores for you.

**ifconfig: configure network interfaces**

**ifconfig** enables or disables a network interface, sets its IP address and subnet mask, and sets various other options and parameters. It is usually run at boot time but it can also make changes on the fly. Be careful if you are making **ifconfig** changes and are logged in remotely; many a sysadmin has been locked out this way and had to drive in to fix things. An **ifconfig** command most commonly has the form

**ifconfig** *interface address options …*

for example:

**ifconfig eth0 192.168.1.13 netmask 255.255.255.0 up**

**ifconfig** *interface* displays the current settings for *interface* without changing them. Many systems understand **-a to** mean “all interfaces,” and **ifconfig -a** can therefore be used to find out what interfaces are present on the system.

**route: configure static routes**

The **route** command defines static routes, explicit routing table entries that never change (you hope), even if you run a routing daemon. When you add a new machine to a local area network, you usually need to specify only a default route;

**Default routes**

A default route causes all packets whose destination network is not found in the kernel's routing table to be sent to the indicated gateway. To set a default route, simply add the following line to your startup files:

**route add default gw** *gateway-IP-address*

Rather than hard coding an explicit IP address into the startup files, most vendors have their systems get the gateway IP address from a configuration file. The way that local routing information is integrated into the startup sequence is unfortunately different for each of our Linux systems

**DNS configuration**

To configure a machine as a DNS client, you need to edit only one or two files: all systems require **/etc/resolv.conf** to be modified, and some require you to modify a "service switch" file as well. The **/etc/resolv.conf** file lists the DNS domains that should be searched to resolve names that are incomplete (that is, not fully qualified, such as anchor instead of anchor. cs.colorado.edu) and the IP addresses of the name servers to contact for name lookups.

**Security Issues**

**IP forwarding**

A UNIX or Linux system that has IP forwarding enabled can act as a router. That is, it can accept third-party packets on one network interface, match them to a gateway or destination host on another interface, and retransmit the packets. Unless your system has multiple network interfaces and is actually supposed to function as a router, it's advisable to turn this feature off. Hosts that forward packets can sometimes be coerced into compromising security by making external packets appear to have come from inside your network. This subterfuge can help an intruder's packets evade network scanners and packet filters. It is perfectly acceptable for a host to use multiple network interfaces for its own traffic without forwarding third-party traffic.

**ICMP redirects**

ICMP redirects can maliciously reroute traffic and tamper with your routing tables. Most operating systems listen to ICMP redirects and follow their instructions by default. It would be bad if all your traffic were rerouted to a competitor's network for a few hours, especially while backups were running. In such case configure your routers (and hosts acting as routers) to ignore and perhaps log ICMP redirect attempts.

**Source routing**: It was part of the original IP specification; it was intended primarily to facilitate testing. It can create security problems because packets are often filtered according to their origin. If someone can cleverly route a packet to make it appear to have originated within your network instead of the Internet, it might slip through your firewall. We recommend that you neither accept nor forward source-routed packets.

**Broadcast pings and other directed broadcasts**

Ping packets addressed to a network's broadcast address (instead of to a particular host address) are typically delivered to every host on the network. Such packets have been used in denial of service attacks; for example, the so-called Smurf attacks. (The "Smurf attacks" Wikipedia article has details.) Broadcast pings are a form of "directed broadcast" in that they are packets sent to the broadcast address of a distant network. The default handling of such packets has been gradually changing.

**IP spoofing**

The source address on an IP packet is normally filled in by the kernel's TCP/IP implementation and is the IP address of the host from which the packet was sent. However, if the software creating the packet uses a raw socket, it can fill in any source address it likes. This is called IP spoofing and is usually associated with some kind of malicious network behaviour. The machine identified by the spoofed source IP address (if it is a real address at all) is often the victim in the scheme. Error and return packets can disrupt or flood the victim's network connections. You should deny IP spoofing at your border router by blocking outgoing packets whose source address is not within your address space. This precaution is especially important if your site is a university where students like to experiment and may be tempted to carry out digital vendettas.

**Host-based firewalls**

Traditionally, a network packet filter or firewall connects your local network to the outside world and controls traffic according to a site-wide policy. The last few Windows releases all come with their own personal firewalls, and they complain bitterly if you try to turn the firewall off. Our example systems all include packet filtering software, but you should not infer from this that every UNIX or Linux machine needs its own firewall. It does not. The packet filtering features are there to allow these machines to serve as network gateways.

**Virtual private networks** Many organizations that have offices in several locations would like to have all those locations connected to one big private network. Such organizations can use the Internet as if it were a private network by establishing a series of secure, encrypted "tunnels" among their various locations. A network that includes such tunnels is known as a virtual private network or VPN. VPN facilities are also needed when employees must connect to your private network from their homes or from the field. A VPN system doesn't eliminate every possible security issue relating to such ad hoc connections, but it's secure enough for many purposes.

## PPP: The Point to Point Protocol

PPP represents an underlying communication channel as a virtual network interface. However, since the underlying channel need not have any of the features of an actual network, communication is restricted to the two hosts at the ends of the link—a virtual network of two. PPP has the distinction of being used on both the slowest and the fastest IP links, but for different reasons. In its asynchronous form, PPP is best known as the protocol used to provide dialup Internet service over phone lines and serial links. These channels are not inherently packet oriented, so the PPP device driver encodes network packets into a unified data stream and adds link-level headers and markers to separate packets. In its synchronous form, PPP is the encapsulation protocol used on high-speed circuits that have routers at either end. It's also commonly used as part of the implementation of DSL and cable modems for broadband service. In these latter situations, PPP not only converts the underlying network system (often ATM in the case of DSL) to an IP-friendly form, but it also provides authentication and access control for the link itself.  In addition to specifying how the link is established, maintained, and torn down, PPP implements error checking, authentication, encryption, and compression. These features make it adaptable to a variety of situations.