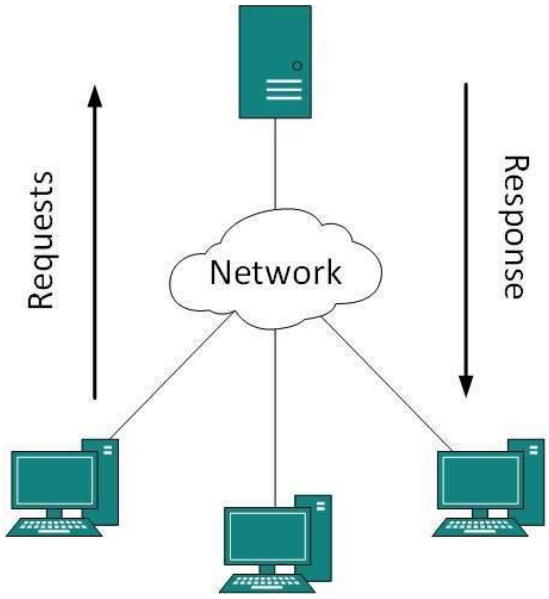
# Chapter 1
# Network Programming Fundamentals
# (2Q) (15 marks)

## 1. Define computer network and network programming.

### a. Computer Network

- A computer network is a system in which multiple computers are connected to each other to share their resources, data and applications.
- Those computers are connected by communication medium.
- A communication medium can be wired or wireless.
- Computer networks can be local, like a home or office network, or they can be global like the internet.
- They use different types of network topologies, such as bus, star, ring or mesh, to organize the connections between devices.



**(Asked 2 times)**

**b. Network Programming**

- Network programming is the process of writing software applications that can communicate with other devices or applications over a computer network.
- This involves designing and implementing protocols and algorithms to enable data exchange and resource sharing across the network.
- Network programming allows applications to interact with other computers or devices by sending and receiving data packets.
- It often involves using network protocols, such as TCP/IP (Transmission Control Protocol/Internet Protocol), UDP (User Datagram Protocol), HTTP (Hypertext Transfer Protocol), and more.
- It is crucial for building various networked applications, such as web servers, web clients (browsers), email clients, chat applications, online games, and many other distributed systems.
- It is used in everyday applications, such as web browsers (like Chrome or Firefox) that retrieve web pages from servers, or messaging apps that send texts and images between devices.
- One common approach in network programming is the client-server model (e.g., a web browser (client) requesting a web page from a web server).
- Network programming also deals with security concerns, as data needs to be protected from unauthorized access during transmission. Encryption and authentication are important aspects of secure network programming.
- Network programming can be applied to various types of networks, including local networks (like in your home or office) or global networks like the Internet.

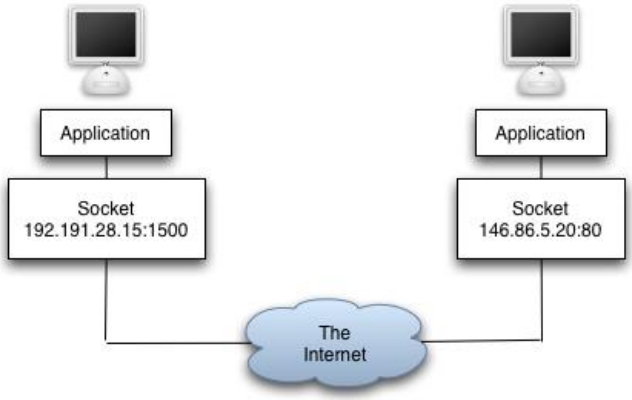



Fig: Network Programming

## 2. Describe the application of Network Programming.

The applications of network programming are:

1. **Communication Medium:**
   - Network programming serves as the backbone of various communication platforms, such as email services, messaging apps, and voice/video conferencing tools, enabling seamless communication between users worldwide.

2. **Web Development:**
   - Network programming is used in web development to create websites and web applications that can communicate with servers.
   - This allows users to view web pages, interact with forms, and submit data over the internet.

3. **E-commerce:**
   - In e-commerce, network programming enables online shopping platforms to process transactions securely, manage inventory, and provide real-time updates on product availability and order status.

4. **Financial Services:**
   - Network programming plays a crucial role in financial services, enabling secure communication between banks, financial institutions, and customers for online banking, money transfers, and stock trading.

5. **Online Education:**
   - In online education, network programming facilitates the delivery of educational content, live classes, and interactive sessions between students and teachers over the internet.

6. **Online Gaming:**
   - In online gaming, network programming enables players to connect, compete, and interact in real-time with others over the internet, creating a shared gaming experience.

7. **Data Storage:**

- Network programming is used to access and store data in remote servers or cloud platforms, allowing users to save and retrieve files from anywhere with an internet connection.

8. **File Sharing and Transfer:**

- Network programming is used in file-sharing applications, allowing users to send and receive files between devices or share them with others over networks.

***(Asked 4 times)***

## 3. Explain network programming model with help of suitable diagram.

## OR

## Compare peer to peer and client server-based model on the basis of communication, cost and security.

- Network programming models describe the way different computers or devices interact and communicate in a networked environment.
- Two popular network programming models are:
    1. Peer-To-Peer Model
    2. Client-Server Model

**1. Peer-To-Peer Model**

- In the Peer-to-Peer model, all devices (peers) are considered equal.
- They can communicate directly with each other without relying on a central server.
- Each peer can act both as a client or a server depending on whether the node is requesting or providing the services.
- Each peer can share resources (files, data, etc.) with others in the network.
- It is useful for small environments, usually up to 10 computers.
- It doesn't have a central server.

Advantages

i. No need for network administrators.
ii. Network is fast/inexpensive to setup and monitor.

Disadvantages

i. It doesn't scale well, their efficiency decreases rapidly as the number of computer increases.
ii. It has a security issue as the device is managed itself.

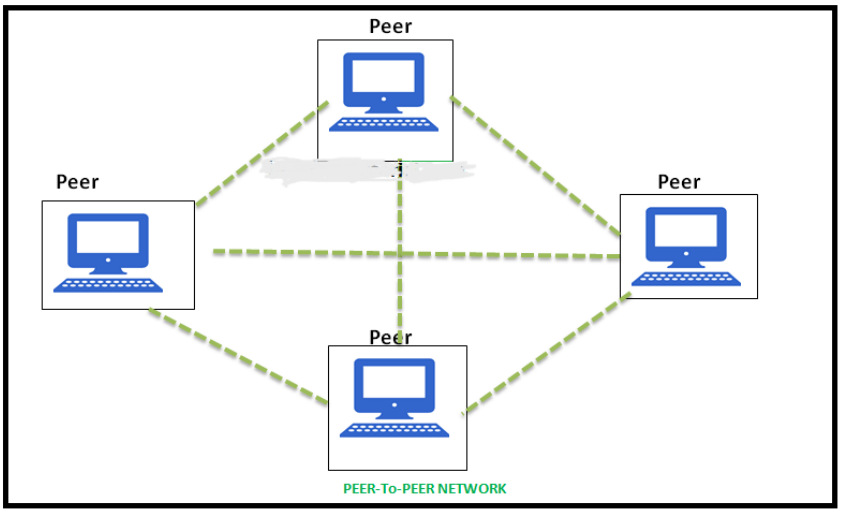



Fig: Peer to Peer model

**2. Client-Server Model**

- Client-Server describes a relationship between two computer programs in which one computer (i.e. client) makes a service request to another computer (i.e. server) which fulfills the request.
- The central controller is known as the server while all other computers in the network are called clients.
- Clients send requests to the server for specific services or data.
- The server processes the requests and sends back responses with the requested information.
- All the clients communicate with each other through a server.
- A server performs all major operations such as security and network management.
- Before a client can access the server resources the client must be identified and be authorized to use the resource.
- Example: Web browsing

Advantages

i. Provides better security.
ii. Easier to administrate because administration is centralized.
iii. All data can be backed up in one centralized location.

Disadvantages

i. It is expensive and requires a server with a large memory.
ii. Requires a professional administrator.

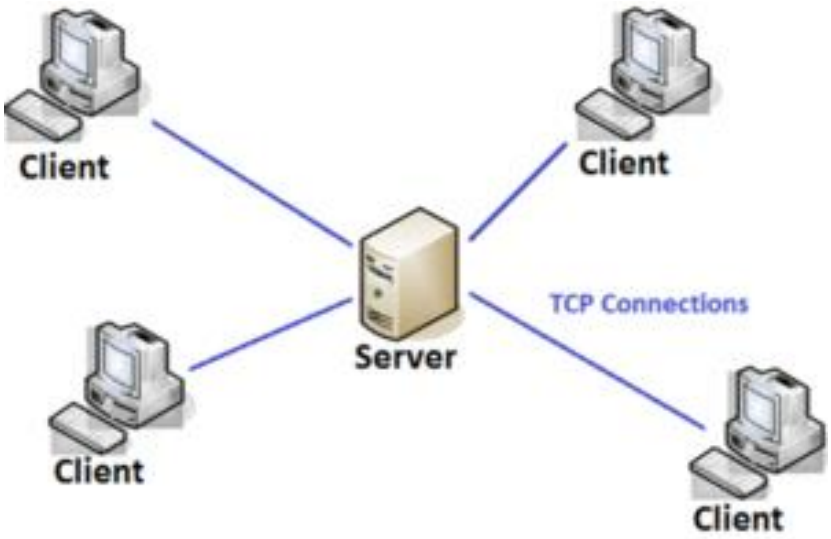



Fig: Client Server Model

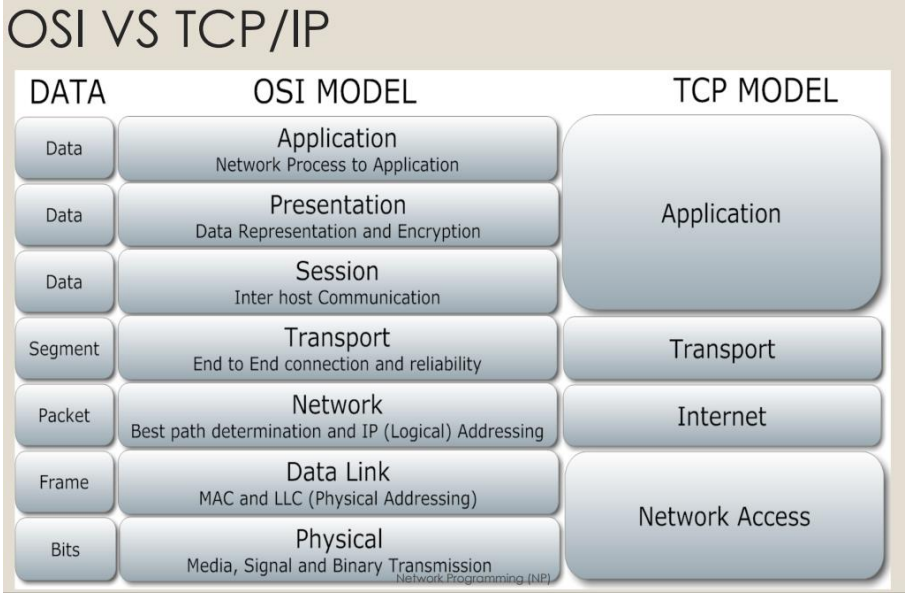
**(Asked 2 times)**

## 4. What do you mean by active network model.

- The active network model is a communication model where packets flowing through a network can dynamically change or modify the operation of the network.
- In this model, packets used are known as active packets.
- Unlike traditional data packets that carry data only, active packets contain executable code that can be executed within the network.
- One challenge with active networking is that it's more complicated to design and manage because now packets can run code.

(Not asked yet)

## 5. Explain in brief about OSI reference model and TCP/IP model.

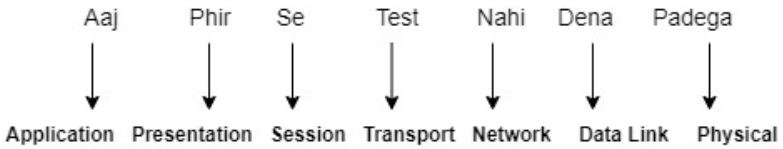



### A. OSI reference model

The seven layers of OSI reference model are:

1. Application Layer - this layer provides services to end users.
2. Presentation Layer – it is responsible for compression, encryption and decryption.
3. Session Layer - It is used to establish, manage, and terminate the sessions.
4. Transport Layer – It provides reliable message delivery from source to destination.
5. Network Layer – It is responsible for moving the packets from source to destination.
6. Data Link Layer - It is used for error free transfer of data frames.
7. Physical Layer – It provides a physical medium through which bits are transmitted.

**Trick to Remember**



***(Asked in Assessment and Assignment)***

## 6. Explain different protocols (at least 3) in each layer of OSI reference model.

1. **Application Layer**
    i. **FTP (File Transfer Protocol):** FTP is used for transferring files between a client and a server over a network, commonly used for file downloads and uploads.
    ii. **HTTP (Hypertext Transfer Protocol):** HTTP is the foundation of data communication on the World Wide Web. It enables web browsers (clients) to request and receive web pages from web servers.
    iii. **SMTP (Simple Mail Transfer Protocol):** SMTP is used for sending and relaying email messages between email clients and servers.

2. **Presentation Layer**
    i. **JPEG (Joint Photographic Experts Group):** JPEG is a widely used image compression format that reduces image file sizes while maintaining acceptable quality.
    ii. **MPEG (Moving Picture Experts Group):** MPEG is a family of video and audio compression standards used in digital video and television broadcasting.
    iii. **GIF (Graphics Interchange Format):** GIF is a popular image format that supports animations and is widely used for sharing images.

3. **Session Layer**
    i. **RPC (Remote Procedure Call):** RPC enables a program running on one computer to call a function or procedure on another remote computer as if it were a local function call, facilitating communication between distributed systems.
    ii. **NFS (Network File System):** NFS allows users to access files and directories on remote computers over a network, providing a convenient way to share files between machines.

4. **Transport Layer**
    i. **TCP (Transmission Control Protocol):** TCP is a reliable, connection-oriented protocol that ensures data delivery and sequencing between devices.

ii. **UDP (User Datagram Protocol):** UDP is a connectionless, lightweight protocol that provides faster data transfer but without the guarantee of reliable delivery.
iii. **SCTP (Stream Control Transmission Protocol):** SCTP is designed for message-oriented applications and provides features like multi-streaming, making it suitable for real-time communication.

5. **Network Layer**
   i. **ICMP (Internet Control Message Protocol):** ICMP is used for network diagnostics and reporting errors, such as ping requests and responses for network testing.
   ii. **Ping:** Ping is a utility that uses ICMP to check if a remote host is reachable over a network and to measure the round-trip time for data packets.
   iii. **IP (Internet Protocol):** IP is the foundational protocol of the internet, responsible for routing data packets between devices across different networks.

6. **Data Link Layer**
   i. **X.25**: X.25 is a packet-switching protocol which defines a way in which packet travels in wide area network.
   ii. **Frame Relay** – It is also a packet switching protocol which is higher performance than X.25.
   iii. **ATM (Asynchronous Transfer Mode):** It is also a packet switching protocol which provides faster packet switching than the previous two protocols.

7. **Physical Layer**
   i. **Ethernet Cabling (e.g., Cat5e, Cat6):** Different types of Ethernet cables are used in the physical layer to provide wired connections between devices in a local area network (LAN).
   ii. **Wi-Fi (IEEE 802.11**): Wi-Fi is a wireless networking technology used for communication between devices in a local area network (LAN).
   iii. **Bluetooth:** Bluetooth is a wireless technology commonly used for short-range communication between devices, such as connecting a smartphone to a wireless headset or a computer to a wireless mouse.

**Q. Describe different internet layer protocols in detail.**

It is Network layer of OSI reference model. Refer to above question.

## 7. Define protocol? “Protocol is needed for network communication”. Justify with client server as two communicating parties.

- A protocol is a standard set of rules that allow electronic devices to communicate or transfer data with each other.
- These rules include:

i. How computers connect to the network.
ii. What type of data may be transmitted.
iii. What commands are used to send and receive data.
iv. How data transfer is confirmed.

Protocols are essential for client-server communication for following reasons:

1. **Data Exchange format**
- Protocols define how data should be structured and formatted for effective communication between client and server.

2. **Standardization**
- Protocols provide a standardized way for different systems to communicate, ensuring compatibility and interoperability.

3. **Connection Management**
- Protocols define how connections are established, maintained, and terminated between client and server.

4. **Error Handling and Recovery**
- Protocols include mechanisms to detect and correct errors during data transmission, ensuring reliable communication.

5. **Security**
- Protocols can incorporate encryption and authentication to protect data during transmission.

6. **Flow Control**

- Protocols manage data flow between client and server, preventing overload and ensuring smooth communication.

**(Asked 6 times)**

## 8. Explain different communication protocols used in networking.

OR

Describe TCP, UDP and SCTP.

OR

Compare different communication protocols based on fragmentation, reliability, flow control and sequencing.

- Different communication protocols are explained below:

1. **TCP (Transmission Control Protocol):**

a. **Reliability**: TCP provides reliable, connection-oriented data transmission. It ensures data delivery and retransmission if packets are lost or corrupted.

b. **Flow Control:** TCP uses flow control mechanisms to manage the rate of data transmission between sender and receiver, preventing data overload and ensuring efficient data delivery.

c. **Sequencing**: TCP guarantees in-order delivery of data packets, ensuring that data is received in the same order it was sent by the sender.

d. **Fragmentation:** Fragmentation refers to the process of breaking larger data packets into smaller segments to fit within the Maximum Transmission Unit (MTU) of the underlying network.
TCP handles fragmentation automatically as part of its data segmentation process.

e. **Connection-oriented:** TCP establishes a connection between the computers before data transmission.

f. **Retransmission:** TCP employs automatic retransmission of lost or corrupted packets to ensure reliable data delivery.

g. **Suitability:** TCP is ideal for applications where data integrity and reliability are critical, such as email transfer, file transfer, and web browsing.

## 2. UDP (User Datagram Protocol):

a. **Reliability**: UDP is a connectionless and unreliable protocol. It does not guarantee data delivery or perform retransmissions, making it faster but less reliable than TCP.

b. **Flow Control:** UDP lacks built-in flow control mechanisms. Applications must implement their own flow control if needed.

c. **Sequencing:** UDP does not guarantee in-order delivery of packets. Packets may arrive out of order at the receiver's end.

d. **Fragmentation:** UDP does not handle fragmentation.

e. **Connectionless:** UDP does not establish a connection before data transmission. It allows data to be sent without prior communication setup, making it faster and simpler than TCP.

f. **Retransmission:** UDP does not provide built-in retransmission, making it a faster but unreliable protocol where data loss or corruption may occur without retransmission.

g. **Suitability:** UDP is suitable for real-time applications, such as video streaming, voice calls, and online gaming, where speed is more critical than data reliability.

### 3. SCTP (Stream Control Transmission Protocol):

- SCTP is a connection-oriented network protocol for transmitting multiple streams of data simultaneously between two endpoints that have established a connection in a computer network.
- SCTP combines the best features of UDP and TCP.

a. **Reliability:** SCTP is a reliable protocol like TCP, ensuring accurate data delivery and retransmission of lost or corrupted packets.

b. **Flow Control:** SCTP uses flow control mechanisms to manage data transmission between sender and receiver, preventing congestion and ensuring efficient data flow.

c. **Sequencing:** SCTP guarantees in-order delivery of messages within each stream, but it can process different streams independently, allowing for better parallelism.

d. **Fragmentation:** SCTP handles data segmentation and reassembly, like TCP, ensuring data can fit within the network's MTU.

e. **Connection-oriented:** SCTP establishes a connection before data transmission, providing a reliable and established communication channel between the client and server.

f. **Retransmission:** SCTP, like TCP, incorporates automatic retransmission of lost or unacknowledged packets to ensure reliable data delivery, making it suitable for real-time applications with reliability requirements.

g. **Suitability:** SCTP is suitable for real-time applications that require reliability, multiple streams, and efficient error recovery, such as voice and video communication, and certain telecommunication services.

**Q. Which transports layer protocols will you use to exchange control information (play, pause, rewind, forward, etc.) and real time audio video data between client and server in the movie streaming application. Justify your answer.**

Ans: We can use TCP, UDP and SCTP for this.

## 9. Justify TCP and IP is needed for exchanging message between process in Client/Server Paradigm.

In the client-server paradigm, where a client application communicates with a server application over a network, both TCP (Transmission Control Protocol) and IP (Internet Protocol) play essential roles in enabling message exchange between processes.

Their roles are:

1. **Internet Protocol (IP):**
a. **Device Identity:** Every device (including clients and servers) has an IP address to identify itself on the network.
b. **Client to Server:** When a client sends data to a server, it includes the server's IP address in the data packet so it reaches the correct destination.
c. **Server to Client**: When the server responds, it uses the client's IP address as the destination in the data packet to send the response back to the client.
d. **Data Routing:** IP handles the path and delivery of data packets between devices, making sure they reach the intended devices.

2. **Transmission Control Protocol (TCP):**
a. **Reliable Communication:** TCP ensures messages are reliably sent and received between client and server processes.
b. **Connection Setup:** Before exchanging messages, TCP sets up a connection between the client and server.
c. **Message Handling:** TCP breaks messages into smaller pieces, sends them, and reassembles them in the correct order at the destination.
d. **Acknowledgments:** TCP confirms the successful delivery of messages to avoid data loss or errors.
e. **Flow and Congestion Control:** TCP manages the speed of data exchange to prevent network congestion and ensure smooth communication.

***(Asked 3 times)***

## 10. Explain 3-way and 4-way handshake for connection establishment and connection termination mechanisms with suitable diagram.

## OR

## Illustrate the TCP 3-way and 4-way handshake mechanism with suitable state transition diagram.

- Handshaking is the process of one computer establishing a connection with another computer or device.
- There are two types of handshaking.
  a. 3-way handshaking – for connection establishment
  b. 4-way handshaking – for connection termination

1. 3-way handshaking

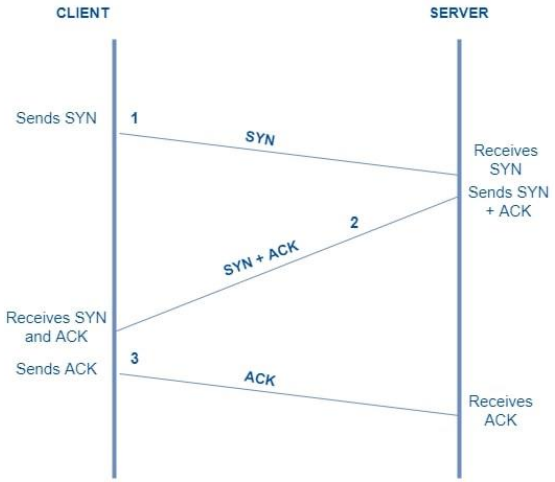


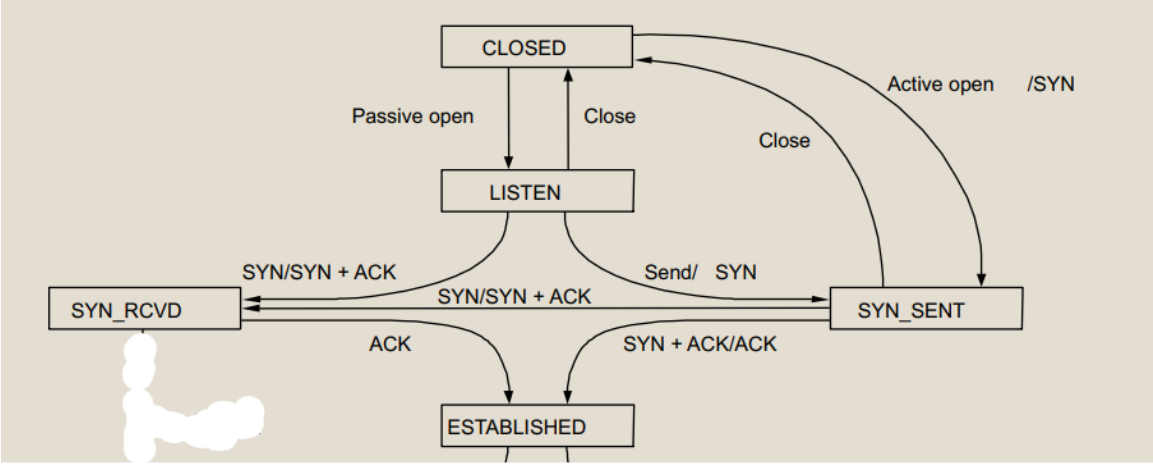

Fig: 3-way handshaking

**Step 1:** The client sends SYN (Synchronization Sequence Number) to the server, because the client wants to establish a connection with the server. The SYN segment contains the initial sequence number (ISN) generated by the client.

**Step 2:** In this step the server responds to the client request with SYN+ACK signal. ACK(Acknowledgement) signifies the response of the segment it received. SYN signifies with what sequence number it is likely to start the segments with.

**Step 3:** After receiving the SYN/ACK segment from the server, the client sends an acknowledgment back to the server. And they both establish a reliable connection with which they will start actual data transfer.

**State transition diagram of 3-way handshaking**

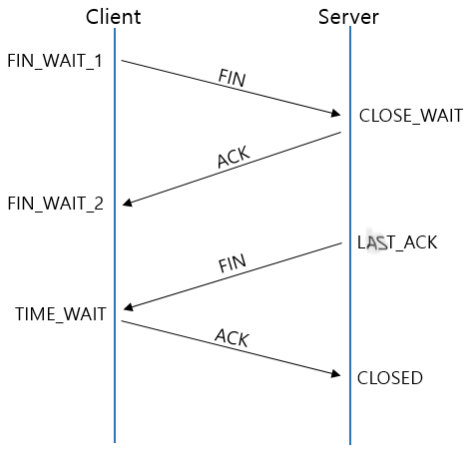



1. **CLOSED**: Initial state, where no connection exists.
2. **LISTEN**: Server is actively waiting for incoming connection requests (passive open)
3. **SYN-SENT:** Client initiates the connection by sending a SYN segment to the server.
4. **SYN-RECEIVED**: Server responds with a SYN/ACK segment, acknowledging the connection request and sending its own SYN.
5. **ESTABLISHED**: The connection is established, and data can be exchanged between client and server.

## Q. Briefly describe CLOSED, LISTEN, SYN_SENT, SYN_RCVD, ESTABLISHED based on socket implementation of client/server architecture for using TCP protocol.

2. **4-way handshaking**

- It is used for closing a TCP connection between two devices.




**4-way handshaking**

**Step 1 (FIN from client to server)**

- When a client decides it want to close the connection. The first FIN termination request is sent by the client to the server.
- It depicts the start of the termination process between the client and server.

**Step 2 (ACK from Server)**

- The server sends the ACK (Acknowledgement) segment to the client immediately after it receives the FIN request.
- It depicts that the server is ready to close and terminate the connection.

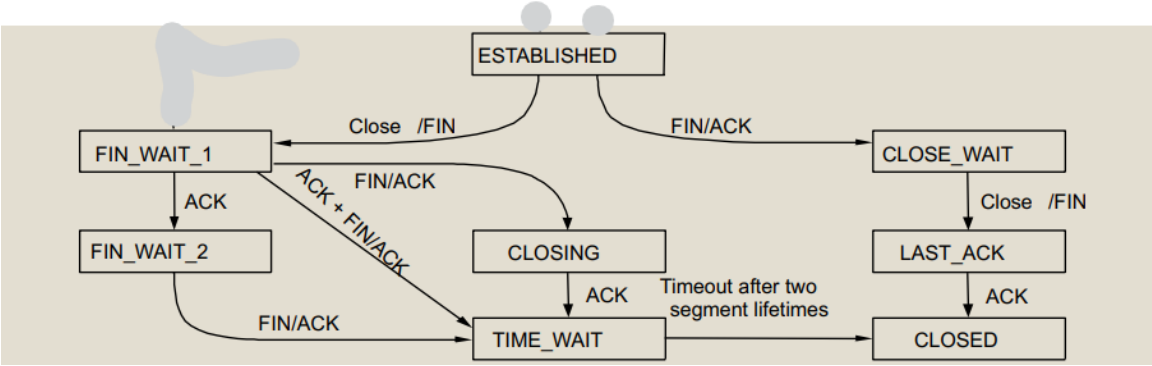
**Step 3 (FIN from Server)**

- The server sends the FIN segment to the client after some time of Step 2 (because of some closing process in the server)
- It is a confirmation signal. It depicts the successful approval for the termination.

**Step 4 (ACK from client)**

- The client now sends the ACK segment to the server that it has received the FIN signal.
- As soon as the server receives the ACK segment it terminates the connection.

**State transition diagram of 4-way handshaking**




1. **ESTABLISHED**: The initial state of an established TCP connection where data is actively exchanged.

2. **FIN-WAIT-1**: The active close initiated by the client. The client sends a FIN segment to the server to request connection termination.

3. **FIN-WAIT-2:** The client waits for an acknowledgment (ACK) of its FIN segment from the server.

4. **CLOSE-WAIT:** The server has received the client's FIN segment, indicating the client's intention to close the connection. It sends ACK to the client. However, the server may still have data to send to the client before closing its end of the connection. Therefore, it enters the CLOSE-WAIT state, waiting for the application to release the connection.

5. **LAST-ACK:** The server sends its own FIN segment to the client, indicating its intention to close the connection. It then waits for an acknowledgment (ACK) from the client to confirm the receipt of its FIN segment.

6. **TIME-WAIT:** The client has received the server's FIN segment, acknowledges it, and enters a TIME-WAIT state. It waits for a brief period before transitioning to the CLOSED state to ensure that all segments related to the connection are cleared from the network. This step prevents any delayed segments from interfering with future connections.

7. **CLOSING**: Both the client and server have initiated the process of closing the connection simultaneously. They have sent FIN segments to each other and are waiting for ACKs.

8. **CLOSED:** The server has received the ACK from the client and transitions to the CLOSED state, closing its end of the connection.

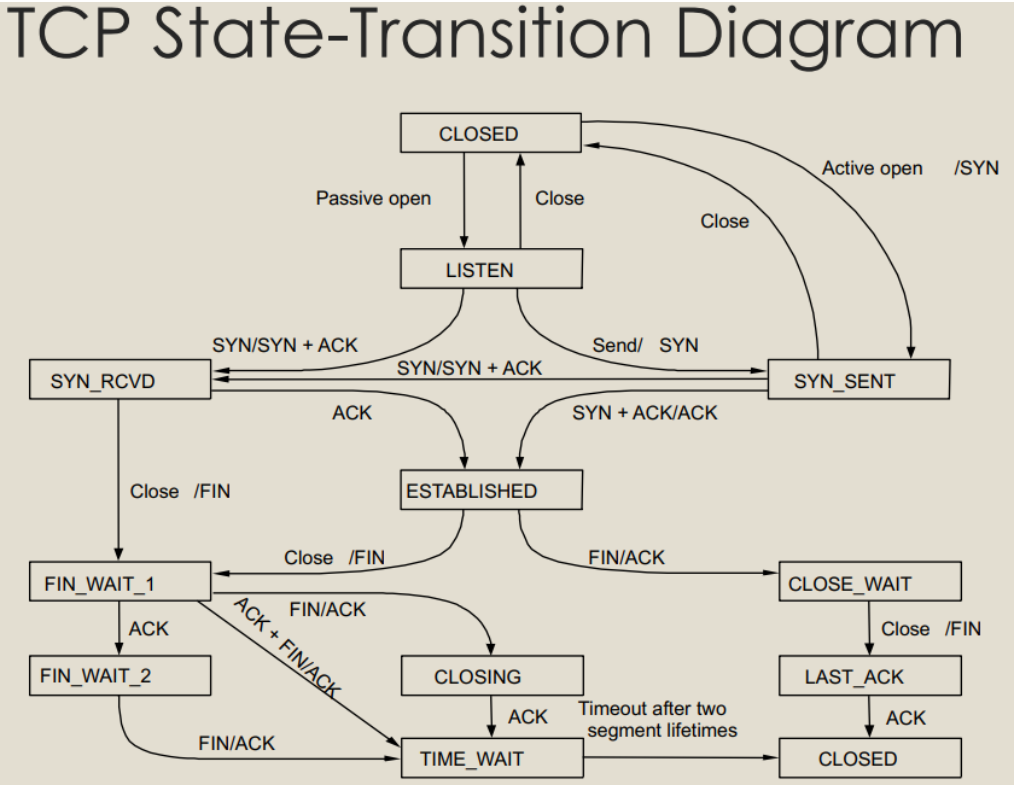
**Q) TCP client /server socket are in ESTABLISHED() state, what if client call the close() function. Explain the state of TCP client/server state after close call is issued.**

Ans: Explain TCP connection termination process. i.e. 4-way handshaking.

***(Asked 10 times)***

## 11. Explain the various states used in TCP state transition diagram with supporting figure. (Explain TCP state transition)

***Note: Study Que no 9 in detail because this is the combination of 3-way and 4-way handshaking***



1. **CLOSED**: The initial state. No connection exists.

2. **LISTEN**: The server is actively waiting for incoming connection requests (passive open) or has initiated a connection request (active open).

3. **SYN_SENT**: The client initiates the connection by sending a SYN segment to the server.

4. **SYN_RECEIVED**: The server has received a SYN segment from the client, acknowledging the connection request. The server responds with a SYN/ACK segment.

5. **ESTABLISHED**: The connection is established, and data can be exchanged between client and server.

6. **FIN-WAIT-1:** The active close initiated by the client. The client sends a FIN segment to the server to request connection termination.

7. **FIN-WAIT-2:** The client waits for an acknowledgment (ACK) of its FIN segment from the server.

8. **CLOSE-WAIT:** The server has received the client's FIN segment, indicating the client's intention to close the connection. It sends ACK to the client. However, the server may still have data to send to the client before closing its end of the connection. Therefore, it enters the CLOSE-WAIT state, waiting for the application to release the connection.

9. **LAST-ACK:** The server sends its own FIN segment to the client, indicating its intention to close the connection. It then waits for an acknowledgment (ACK) from the client to confirm the receipt of its FIN segment.

10. **TIME-WAIT:** The client has received the server's FIN segment, acknowledges it, and enters a TIME-WAIT state. It waits for a brief period before transitioning to the CLOSED state to ensure that all segments related to the connection are cleared from the network. This step prevents any delayed segments from interfering with future connections.

11. **CLOSING:** Both the client and server have initiated the process of closing the connection simultaneously. They have sent FIN segments to each other and are waiting for ACKs.

12. **CLOSED:** The server has received the ACK from the client and transitions to the CLOSED state, closing its end of the connection.

## 12. Which function is responsible for sending SYN segment during TCP connection establishment phase?

- In the context of TCP connection establishment, the responsibility for sending the SYN segment lies with the client-side TCP stack.
- When a client application wants to establish a connection to a server, it requests the operating system's TCP stack to initiate the connection.
- In C/C++, the socket() function creates a new socket, and the connect() function initiates the connection, which includes sending the SYN segment.

## 13. What is Maximum Segment Lifetime (MSL).

- Maximum Segment Lifetime (MSL) is a time limit set by TCP to make sure data doesn't stay in the network forever.
- It prevents outdated or lost data from causing problems.
- The MSL value is typically around 2 minutes, and after this time, if the data hasn’t been acknowledged, TCP considers the data lost and takes appropriate action.
- It's especially important during connection termination to clear any delayed/old data before fully closing the connection.

## 14. Discuss about the different ways for making Remote Procedure Call (RPC).

- RPC is a protocol used in distributed system to allow a program running on one computer to invoke a procedure on another computer over a network.
- The different ways of making RPC are:
  i. Remote Method Invocation (RMI) – It is a Java -specific implementation of RPC that allows java objects to be executed on remote servers.
  ii. Socked-based RPC.
  iii. XML RPC
  iv. JSON RPC

## 15. While establishing TCP connection, the initial sequence number should not start from 0. Why?

- Starting the initial sequence number in a TCP connection from 0 or a fixed value can make the connection vulnerable to security attacks.
- Attackers may predict future sequence numbers, leading to security risks like session hijacking and data manipulation.
- To enhance security, TCP connections use a random initial sequence number that is difficult to predict.
- Randomness in the initial sequence number makes it challenging for attackers to guess subsequent sequence numbers.
- Modern TCP implementations may also use cryptographic techniques or timestamps to further strengthen security against prediction attacks.
- By avoiding predictable sequence numbers, TCP connections become more secure and resistant to unauthorized access and data manipulation.

## Q. Explain the role of IP, Port and Protocol in network-based program.

**Ans: Chapter 1 Que no 7 and Chapter 2 Que no 1**

# Chapter 2
# UNIX Programming
# (6 Que) (45 marks)

## 1. Explain relationship between Socket, Port and Ip with help of outline code.

**IP:**

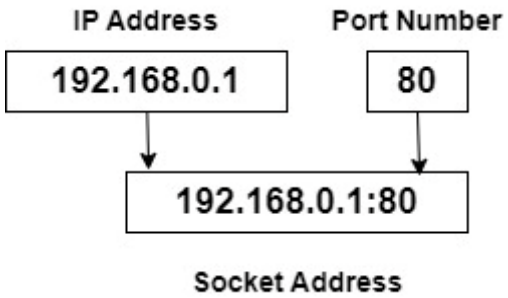
- IP stands for Internet protocol.
- IP address is a unique address that identifies a device on the internet or a local network. It helps to identify and locate devices so they can communicate with each other. Example: 192.168.1.1

**Port**:

- There are many processes running on the computer.
- Data which is sent / received must reach the right process.
- Port numbers are used to identify particular process executing in the device.
- Example: HTTPS – 80, FTP-21

**Socket:**

- Socket is an endpoint of a 2-way communication between two different processes on the same or different machines running on the network.
- A socket is a combination of IP address of a system and port number of a program/process within the system.




**Q. What are the different types of communication address used by socket?**

- When you use a socket to send or receive data, you specify both the IP address and the port number as part of the communication address.

**(Asked 3 times)**

## 2. What is socket and socket API?

**Socket(Internet Socket):**

- A socket is a 2-way communication endpoint that enables data exchange between two devices over a network.
- Think of a socket as an interface that allows programs (applications) to send and receive data to and from the network.
- Sockets are identified by an IP address and a port number, allowing communication between different devices on the network.
- To the kernel, a socket is an endpoint of communication.
- To an application, a socket is a file descriptor that lets the application read/write from/to the network.

**Socket API (Application Programming Interface):**

- The socket API is a set of programming functions and procedures provided by the operating system or networking library to interact with sockets.
- It offers a standardized way for applications to create, bind, connect, send, and receive data through sockets.
- Common socket API functions include socket(), bind(), connect(), listen(), accept(), send(), and receive().

**Normal Workflow of Socket**

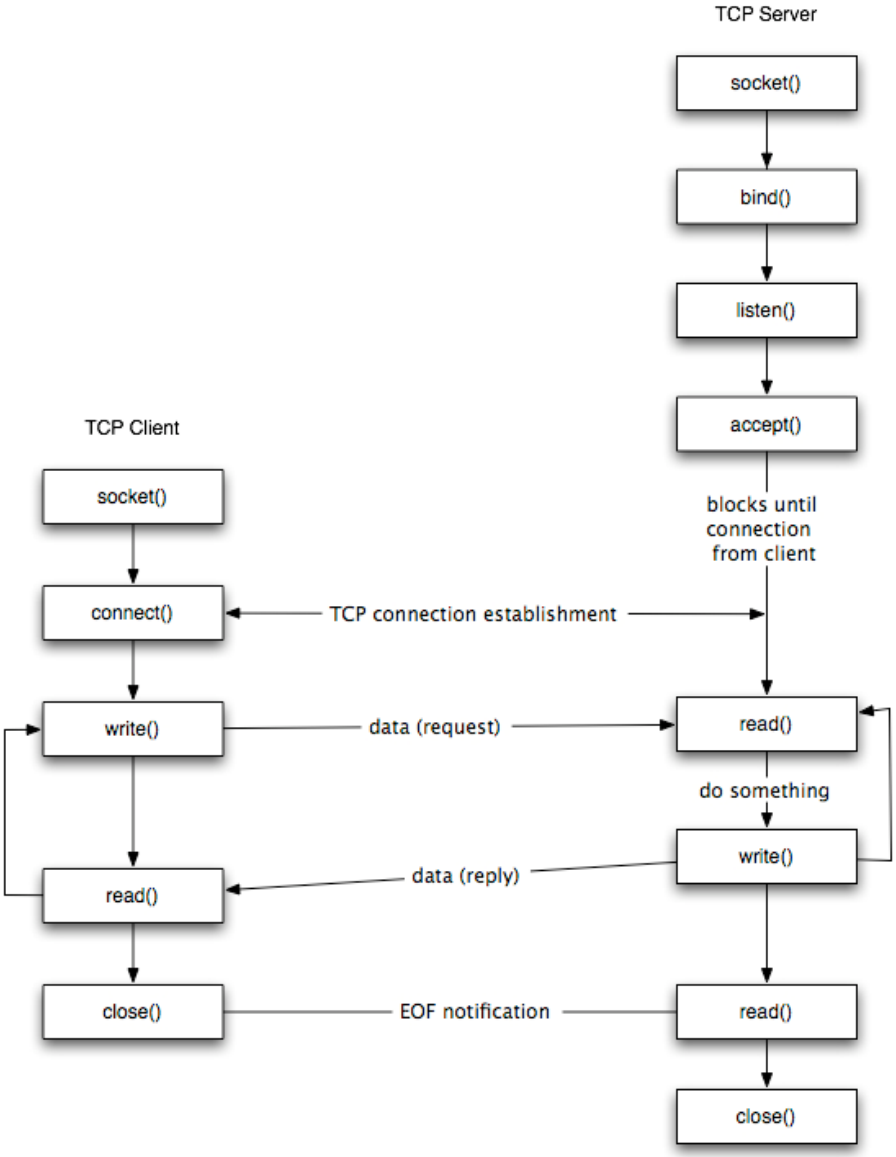
- When we desire a communication between two applications possibly running on different machines, we need sockets.
- Sockets are generally employed in the client-server applications.
- The server creates a socket, attaches it to a Ip address and port then waits for the client to connect it.

*The bind() function is used to associate a specific IP address and port number with a socket. By doing this, the socket becomes "bound" to that particular combination of IP address and port, and it can then send and receive data using that network address.*

- The client creates a socket and then attempts to connect to the server socket.
- When the connection is established, transfer of data takes place.

(Important in later topics)

## 3. Explain TCP client/server flow with suitable diagram.




The steps for establishing a TCP socket on the client side are:

i. Create a socket using the socket() function.

ii. Connect the socket to the address of the server using the connect() function.

iii. Send and receive data by the means of write() and read() functions.

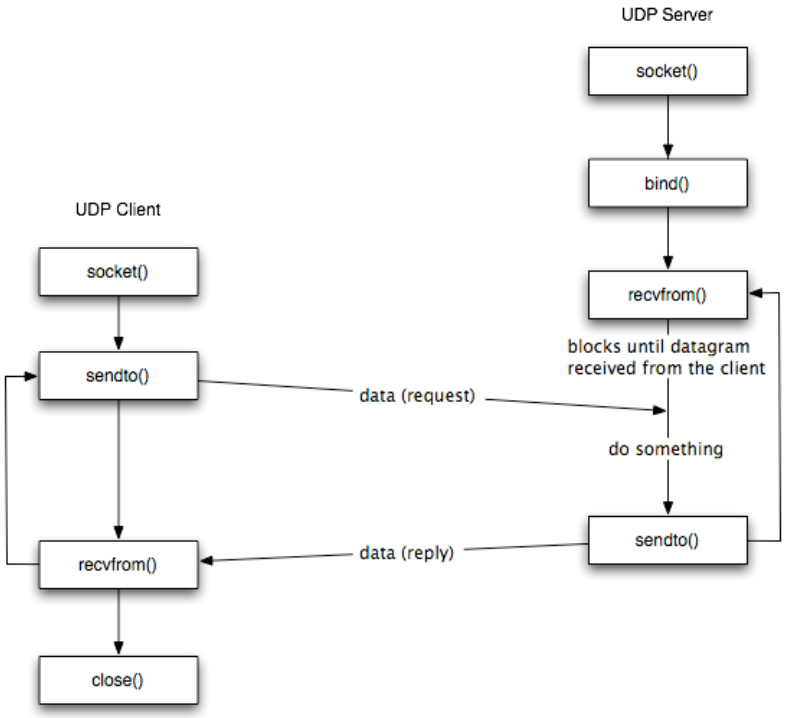
iv. Close the connection by the means of the close() function.

The steps involved in establishing a TCP socket on the server side are:

i. Create a socket with the socket() function.
ii. Bind the socket to an address using the bind() function.
iii. Listen for connection with the listen() function.
iv. Accept a connection with the accept() function. At this point, connection is established between client and server, and they are ready to transfer data.
v. Send and receive data by the means of read() and write() functions.
vi. Close the connection by means of the close() function.

(Important in later topics)

## 4. Explain UDP client/server socket flow with suitable diagram.




The steps involved in establishing a UDP socket communication on the client side are:

- Create a socket using the socket() function.
- Send and receive data by means of recvfrom() and sendto() functions.
- Close the connection by means of the close() function.

The steps involved in establishing a UDP socket communication on the server side are:

- Create a socket with the socket() function.
- Bind the socket to an address using the bind() function.
- Send and receive data by means of recvfrom() and sendto() functions.
- Close the connection by means of the close() function.

**(Asked 4 times)**

## 5. What do you mean by socket address structure. Discuss the different constituents of the UNIX socket address structure.

## OR

## Explain different socket address structures available in UNIX network programming.

Generic Socket address structure

1. 'struct sockaddr'
2. 'struct sockaddr_storage'

Unix domain Socket address structure

1. 'struct sockaddr_un'

Internet Socket address structure

1. 'struct sockaddr_in' (for IPv4)
2. 'struct sockaddr_in6' (for IPV6)

- Various structures are used in Unix Socket Programming to hold information about the address and port, and other information.
- A socket address structure is a data format used in computer networking to represent essential details about a network connection.
- It contains the necessary details like IP addresses and port numbers to establish connections and send/receive data.

Different socket address structures available in UNIX programming are:

1. **'struct sockaddr'**

- The 'struct sockaddr' is a generic socket address structure used for all types of sockets, including Unix domain sockets and Internet domain sockets (IPv4 and IPv6).
- It's used when the specific IP version is not known in advance.
- Think of it as a flexible address structure that can handle various types of network addresses.
- The 'struct sockaddr' has two main fields:
  i. sa_family (address family)
  ii. sa_data (the actual address data).

2. **'struct sockaddr_storage'**

- It's a larger version of 'struct sockaddr'.
- It can hold any type of address supported by the system, including IPv4, IPv6, etc.
- Think of it as a bigger container that can store any type of address, making it useful for handling different address families.

3. **'struct sockaddr_un'**

- This structure is used for Unix domain sockets, which are used for communication between processes on the same machine.
- It contains the following fields:
  i. sun_family (address family)
  ii. sun_path: (Unix domain socket path, which is a file path in the filesystem)

4. **'struct sockaddr_in':**

- The struct sockaddr_in is used for Internet domain sockets IPv4 addresses.
- It provides detail about the IPv6 address and port number.
- It contains three main fields:
  i. sin_family (address family),
  ii. sin_addr (IPv4 address)
  iii. sin_port (port number)

5. '**struct sockaddr_in6'**:

- The struct sockaddr_in6 is used for Internet domain sockets with IPv6 addresses.
- It provides detail about the IPv6 address and port number.
- It contains three main fields:
  i. sin6_family (address family)
  ii. sin6_addr (ipv6 address)
  iii. sin6_port (port number)

***(Asked 2 times)***

## Q. Explain Generic socket address structure.

## Q. What are the different socket address structures used in Unix system to make system calls such as connect and bind independent of IP versions

Ans: Explain generic socket address structure for both of them:

1. 'struct sockaddr'
2. 'struct sockaddr_storage'

**(Asked 2 times)**

## Q. Explain socket address structure for IPv4 and IPv6.

Ans: Explain Internet Socket address structure.

1. 'struct sockaddr_in' (for IPv4)
2. 'struct sockaddr_in6' (for IPV6)

## Q. Compare Unix domain socket address structure and internet domain socket address structure.

Ans: Compare between ***struct sockaddr_un*** and

***(struct sockaddr_in and struct sockaddr_in6***)

## 6. Explain the importance of Generic socket address structure.

They are listed below:

1. **Protocol Independence:** The struct sockaddr allows socket operations to work with various network protocols and address families. It can handle both IPv4 and IPv6 addresses, as well as other address families supported by the operating system, making the code adaptable to different network configurations.

2. **Flexibility:** By using a generic socket address structure, developers can write socket code that can be easily extended to support future address families or protocols without significant modifications.

3. **Portability:** When applications are written using the generic socket address structure, they can be easily ported across different platforms and operating systems with minimal changes. The code remains more compatible and reusable.

4. **Adapting to Network Changes**: With the generic socket address structure, applications can be more resilient to changes in the network environment. They can handle network reconfigurations or shifts between IPv4 and IPv6 addressing without requiring major code revisions.

5. **Future-Proofing**: As new address families or protocols emerge, applications using the generic socket address structure will be better prepared to support them, reducing the need for significant code updates.

***(Asked 3 times) Imp (Asked in Internal Exam)***

## 6. Explain the concept of value result argument in socket programming.

- In socket programming, the concept of "value-result" arguments refers to function parameters that serve a dual purpose.
- These parameters are used both as input values to provide information to the function and as output values to return additional information back to the caller.
- Simply, the function takes certain data as input, performs some operations, and then updates the same variables to return additional data after the operation.

**Initial Values:** Before calling the socket function, you provide some initial values for certain parameters, such as IP address and port number, using a socket address structure.

**Function Call:** You call the socket function, which performs the intended operation, like retrieving the socket's local address.

**Output Values:** After the function call, the socket function updates the provided parameters with new information. For example, it may return the local address of the socket.

**Example**: A common use is with getsockname(), which retrieves the local address of a socket.

- When a socket address structure is passed to any socket function, it is always passed by reference, meaning a pointer to the structure is passed.
- Additionally, the length of the structure is also passed as an argument, but the way in which the length is passed depends on the direction in which the structure is being passed:

  i. from the process to the kernel

  ii. from the kernel to the process.

### i. From process to kernel

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

**Arumgents to these functions:**

a. The pointer to the socket address structure
b. The integer size of the structure.

*struct sockaddr_in serv;*

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

### ii. Kernel to Process

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

**Arguments to these functions:**

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

*struct sockaddr_un cli;*

*socklen_t len; len = sizeof(cli);*

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

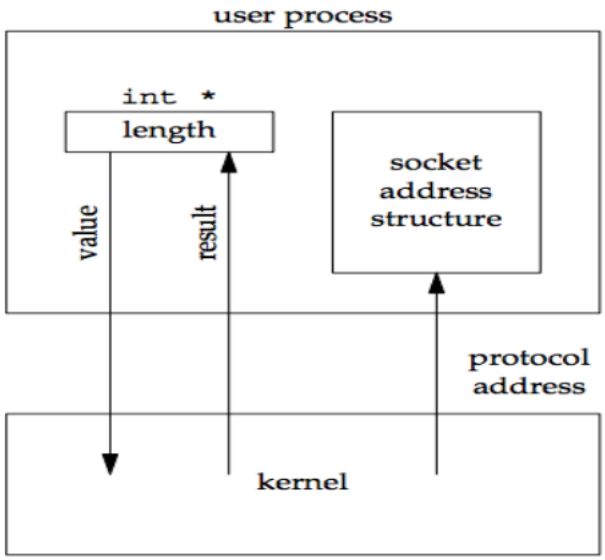



Fig: Value Result Argument

## Q. What are the ways to pass the length of socket structure for different socket API's argument? Explain them in detail with function prototype and argument detail.

In Unix-like systems, when passing socket structures as arguments to various socket-related functions, you often need to specify the length of the structure.

This is crucial for the function to properly understand the size of the structure and avoid memory access issues.

We pass length of socket structure in these two APIs: bind() and connect().

When we use these functions to set up network communication, we need to provide the length of the socket structure as one of the arguments. This length helps the functions know the size of the structure you're passing.

**1. bind() function**

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

In above code example, addrlen is the length of the addr structure in bytes.

**2. connect() function**

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

In above code example, addrlen is the length of the addr structure in bytes.

**(Asked 2 times)**

## 7. What is byte ordering? What is the use of byte ordering in network programming.

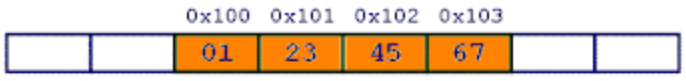
- Byte ordering refers to the way in which multi-byte data (such as integers or floating-point numbers) is represented and stored in memory.
- In network programming, byte ordering becomes essential when data is exchanged between different computer systems, which might have different native byte orders.
- There are two common byte orderings: Big Endian and Little Endian.
- They are two ways of storing multibyte data-types.

**i. Big endian**

- The most significant byte (MSB) is stored at the lowest memory address, and the least significant byte (LSB) is stored at the highest memory address.

**ii. Little endian**

- The least significant byte (LSB) is stored at the lowest memory address, and the most significant byte (MSB) is stored at the highest memory address.




Big Endian


Little Endian

The use of byte ordering in network programming are:

i. **Data Consistency**: Byte ordering in network programming ensures data consistency when exchanging information between computers with different ways of storing data.
ii. **Data Conversion**: Byte ordering allows data to be converted to a standardized "Network Byte Order," ensuring consistency across all systems.

iii. **Portability**: By handling byte ordering, network programs become portable and can work correctly on diverse systems without modification.
iv. **Network Reliability**: Proper byte ordering ensures data is accurately interpreted by both the sender and receiver, improving network communication reliability.

## Q. Differentiate little endian and big endian.

Ans: above answer and figure

## 8. Explain and differentiate network and host byte ordering with necessary functions.

Network Byte Ordering: Typically big-endian.

Host Byte Ordering: Can be little-endian or big-endian, depending on the computer architecture.

**i. Network byte ordering**

- Network byte ordering, also known as big-endian byte order, is a standardized way of representing multi-byte data when exchanging information over a network.
- Necessary Functions: In C language, the following functions are used to convert data to and from network byte order:
  i. **htonl():** Host to Network Long - Converts a 32-bit integer from host byte order to network byte order.
  ii. **htons():** Host to Network Short - Converts a 16-bit integer from host byte order to network byte order.

**ii. Host Byte Ordering:**

- The host byte order varies depending on the architecture and operating system of the computer.
- Necessary Functions: In C language, the following functions are used to convert data to and from host byte order:
  i. **ntohl():** Network to Host Long - Converts a 32-bit integer from network byte order to host byte order.
  ii. **ntohs():** Network to Host Short - Converts a 16-bit integer from network byte order to host byte order.

**(Asked 3 times)**

## 9. What do you mean by byte manipulation function? Describe any five byte manipulation functions with their uses.

OR

List out some of the ANSI C. (Byte manipulation function)

- Byte manipulation functions are programming functions that allow you to work with individual bytes of data.
- They allow you to perform operations at the byte level.
- Useful for working with raw binary data or binary data structures.
- Common tasks include copying, filling, or comparing data on a byte-by-byte basis.
- Often used in low-level programming tasks and network programming.

Some of the byte manipulation functions are:

1. **bzero:**
- Sets a block of memory to zero (nullifies) by writing zero bytes to the specified memory location.
- Use: Useful for initializing buffers or arrays with zero values.

2. **bcmp (also known as memcmp):**
- Compares two blocks of memory byte-by-byte and returns an integer value representing their relationship.
- Use: Used to compare memory areas, often used to check if two pieces of data are the same.

3. **bcopy (also known as memcpy):**
- Copies a specified number of bytes from one memory location to another.
- Use: Used to copy data between buffers or arrays, helpful for data manipulation and manipulation of data structures.

4. **htonl():** Host to Network Long - Converts a 32-bit integer from host byte order to network byte order.

5. **htons():** Host to Network Short - Converts a 16-bit integer from host byte order to network byte order.

6. **ntohl():** Network to Host Long - Converts a 32-bit integer from network byte order to host byte order.

7. **ntohs():** Network to Host Short - Converts a 16-bit integer from network byte order to host byte order.

***(Asked 3 times)***

## 10. Explain the statement fork() is called once but returns twice. Explain how fork() identifies child and parent process with suitable code.

- The system call fork() is used to create a new process.
- The new process is an exact copy of the parent process, including the same code, data, and resources.
- After forking, the parent process and child process execute independently.

- The statement "fork() is called once and it returns twice" refers to the behavior of the fork() system call in Unix-like operating systems.
- When the fork() function is called, it creates a new process (child process) that is an exact copy of the calling process (parent process) including the same code, data, and resources.
- After the fork() call, both the parent and the child processes continue executing from the point of the fork() call, but they return different values.

```c
#include <stdio.h>
#include <unistd.h>

int main() {
    pid_t pid = fork();

    if (pid < 0) {
        // Error handling for fork failure
        printf("Error: Failed to create a new process\n");
        return 1;
    } else if (pid == 0) {
        // Code executed by the child process
        printf("Child process (PID: %d)\n", getpid());
    } else {
        // Code executed by the parent process
        printf("Parent process (PID: %d), Child PID: %d\n", getpid(), pid);
    }

    return 0;
}
```

Output

Parent process (PID: 12345), Child PID: 12346

Child process (PID: 12346)

The return value of fork is:

i. -1 if the creation of the process failed.
ii. 0 if the current process is the child process.
iii. Positive value if the current process is the parent process.

## 11. fork() and exec()

**i. fork():**

- fork() is a system call in Unix-like operating systems.
- It creates a new process (child process) that is an exact copy of the calling process (parent process).
- After fork(), both the parent and the child processes continue executing from the point of the fork() call, but they return different values.
- In the parent process, fork() returns the process ID (PID) of the child process.
- In the child process, fork() returns 0 to indicate that it is the child process.

**ii. exec()**

- exec() is another system call in Unix-like operating systems.
- After a successful fork(), the parent and child processes are identical, which means they share the same program, code, data, and file descriptors.
- However, in many scenarios, you might want the child process to execute a different program rather than continuing with the same program as the parent.
- This is where exec() comes into play.
- The exec() system call replaces the current process image with a new process image, effectively changing the program being executed by the child process.
- If exec() is successful, it does not return to the calling process; instead, the new program starts executing immediately. If there is an error, it returns -1.

***(Asked 3 times)***

## 12. What are the major differences of wait() and waitpid()?

wait()

- wait() is a system call in Unix-like operating systems.
- It is used by the parent process to wait for the termination of its child process.
- When a parent process calls wait(), it suspends its execution and waits until any of its child processes terminates.
- If the parent has multiple child processes, wait() will return when any one of the child processes terminates, and it returns the termination status of that child.

  **Use Case:**
- A typical use case of wait() is in a parent process that has spawned multiple child processes. The parent needs to wait for the child processes to finish their tasks before it continues with its execution or takes further actions based on the child's termination status.

waitpid():

- waitpid() is a system call in Unix-like operating systems, similar to wait().
- It allows the parent process to wait for a specific child process, rather than waiting for any child like wait() does.
- With waitpid(), the parent can specify which child process it wants to wait for by providing the child's process ID (PID) as an argument.

  **Use Case:**
- waitpid() is useful when the parent process wants to wait for a particular child process to finish, among multiple child processes it has spawned.
- It is also helpful when the parent wants to check the termination status of a specific child process and take appropriate action based on that status, without waiting for all child processes to finish.

Summary:

- wait() is used by the parent process to wait for any of its child processes to terminate.
- waitpid() is used by the parent process to wait for a specific child process to terminate.

***(Asked 2 times)***

## 13. Explain the possible option values that you can supply in watpid() system call.

- The waitpid() system call in Unix allows a parent process to wait for the termination of a specific child process.
- It also provides some additional options to customize its behavior.
- Here are the possible option values that you can supply as the third argument to the waitpid() function:

1. **WNOHANG**:
    - With this flag, waitpid() returns immediately if none of the child processes specified by pid have terminated.
    - It does not block the parent's execution and returns 0 as the PID of the child.

2. **WUNTRACED**:
    - This flag causes waitpid() to return if a child process has stopped (but not terminated) due to a signal.
    - It allows the parent to monitor child processes even if they are stopped temporarily.

3. **WCONTINUED**:
    - This flag is used with WUNTRACED and causes waitpid() to return if a child process has been resumed from a stopped state due to a SIGCONT signal.

4. **WEXITED**:
    - This flag causes waitpid() to return if a child has terminated normally.

(Asked 6 times)

## 14. Concurrent Server in Unix.

### OR

### What is a concurrent Server. Explain the fork() function in developing a concurrent server. Provide a stepwise overview.

- In Unix, a concurrent server is a type of server architecture designed to handle multiple client connections simultaneously and independently.
- It allows multiple clients to interact with the server concurrently, without waiting for previous clients to finish their requests.
- The concurrent server achieves this by creating a new process or thread for each client connection, which allows the server to serve multiple clients concurrently.
- Key characteristics of a concurrent server in Unix:

1. **Multiple Client Connections**: A concurrent server can handle multiple client connections simultaneously. Each client connection is treated as a separate task, and the server does not block other clients while serving one.
2. **Independent Processing**: Each client connection is processed independently in its own child process or thread. This means that if one client takes longer to process its request, it does not affect the processing of other clients.
3. **Forking or Threading**: Two common approaches to implement a concurrent server in Unix are forking and threading. In the forking approach, the server forks a new process for each client connection, while in the threading approach, the server creates a new thread for each client connection.
4. **Resource Sharing**: Concurrent servers need to manage shared resources, such as shared data or file descriptors, to ensure correct and safe communication between different client connections.
5. **Scalability**: Concurrent servers are often preferred in situations where there are potentially many client connections to handle. They can scale better as they can leverage multiple CPU cores and system resources efficiently.

**Example use case of a concurrent server:**

- Imagine a web server receiving requests from multiple clients simultaneously.
- Instead of processing each request sequentially, a concurrent server can create a new process or thread for each client connection, allowing the server to respond to multiple clients concurrently.
- This enables the web server to serve multiple web pages or resources to different clients simultaneously, improving response times and overall performance.

- The fork() function is used to create a new process (child process) that is a duplicate of the calling process (parent process).
- In the context of a concurrent server, the fork() function is used to handle multiple client connections simultaneously by creating a new process for each client.
- Here's how the fork() function is used in developing a concurrent server: the stepwise overview is provided.

1. **Server Initialization:**
- Create and bind a socket to a specific port using socket() and bind() functions.
- Set the socket to the listening state using listen() with a specified backlog.

2. **Accepting Client Connections:**
- Enter a loop to continuously accept incoming client connections using accept().
- When a client connects, a new dedicated socket for that client is returned.

3. **Forking a Child Process:**
- After accepting a client connection, use fork() to create a child process.
- In the child process (fork returns 0), handle the connected client.

4. **Child Process Handling:**
- Close the original listening socket in the child process to avoid conflicts.
- The child process is now dedicated to serving the connected client.
- Perform the necessary operations to read client requests and send responses using the new client socket.

5. **Parent Process Handling:**
- Close the client socket returned by accept() in the parent process.
- The parent process can immediately go back to the beginning of the loop to continue listening for new client connections.

6. **Multiple Concurrent Clients:**
- The server handles multiple clients concurrently using child processes.
- Each client connection is handled independently by its own child process.

7. **Reaping Child Processes:**
- The operating system reaps child processes automatically to prevent zombie processes.

8. **Graceful Shutdown:**
- Implement a mechanism to handle server shutdown gracefully, such as handling specific signals to terminate the server and child processes.

## 15. What will happen when close () is called in TCP socket during the implementation of concurrent server?

- In a concurrent server, multiple client connections are handled simultaneously by creating a new process (or thread) for each client.
- When a client connection is established, a new process (child process) is created using fork() to handle that specific client.
- Each child process has its own set of file descriptors inherited from the parent process.
- When a child process finishes handling a specific client connection, it should close the corresponding socket using the close() function to release the associated resources.
- Closing the socket with close() in a child process is crucial for the following reasons:

1. **Resource Cleanup:** The close() function releases the resources associated with the socket, such as file descriptors and memory buffers. This ensures proper cleanup and prevents resource leaks.

2. **Avoiding Orphaned Connections**: If a child process doesn't close the socket after handling the client connection, the socket will remain open in the parent process. This could lead to orphaned connections and potentially cause resource exhaustion if many connections are left open.

3. **Signal Proper Termination:** Closing the socket in the child process signals to the operating system that the connection is terminated, allowing it to clean up the connection-related data structures and handle any pending signals or I/O operations associated with the socket.

- In summary, calling close() in the child process ensures proper cleanup of resources and allows the operating system to handle the termination of the connection accurately.
- This practice is essential for the correct and efficient implementation of a concurrent server that handles multiple client connections simultaneously.

***(Asked 3 times)***

## 16. Explain Unix Socket. Explain advantages of Unix domain protocol.

## OR

## What is Unix domain socket?

**Note: Unix socket / Unix domain socket / Unix domain protocol all are same.**

- A Unix socket, or IPC (Inter-Process Communication) socket, is a type of communication endpoint used for local communication between processes on the same Unix-like operating system.
- It enables processes on the same machine to communicate with each other.
- Unlike network sockets that use IP addresses and port numbers for communication over a network, Unix sockets use the file system as a namespace.
- They offer high-speed communication with minimal overhead by eliminating the overhead associated with network communication.
- Provides security through file permissions.
- They are a powerful tool for achieving efficient communication between processes running on the same Unix-like operating system.

**Creating and using Unix sockets involves the following steps:**

- Create a socket using the socket() system call.
- Bind the socket to a specific file system path using the bind() system call. This file system path will be used to locate the socket.
- Listen for incoming connections (if applicable) using the listen() system call.
- Accept incoming connections using the accept() system call (if applicable).
- Communicate using the established socket using send() and recv() functions or their variations.

The advantages of Unix socket protocol are:

1. **Fast and Efficient Communication:**
- Unix domain sockets enable quick and efficient communication between processes on the same machine.
- They share data directly in memory, avoiding complex network steps. This makes things happen faster.

2. **No Network Overhead:**
- Unlike regular network sockets, these sockets don't deal with the extra work of networks.

3. **Reduced Resource Usage:**
- Unix domain sockets uses less system resources compared to network sockets because they avoid the complexities of network-related operations.

4. **Security Through File Permissions:**
- Communication occurs through special files in the file system.
- File permissions can be used to control which processes can communicate through the socket, providing a degree of access control and security.

5. **Private Communication:**
- Communication remains within the same machine, making Unix domain sockets ideal for private and internal communication.

6. **Well-Suited for Inter-Process Communication:**
- Unix domain sockets are designed for inter-process communication (IPC) scenarios, where different parts of an application or different applications need to exchange data efficiently.

7. **No Need for IP Addresses or Port Numbers:**
- Unix domain sockets use file system paths for addressing, eliminating the need for managing IP addresses and port numbers.

8. **Compatible with Standard Tools:**

- Unix domain sockets can be used with standard tools like netstat to monitor socket status, making troubleshooting easier.

Note: There are two types of socket:

**i. Internet Socket (Network Socket)**

- Used for communication between processes on different machines over a network.
- Uses IP addresses and port numbers for addressing.
- Que no 2

**ii. Unix Domain Sockets (Local Sockets):**

- Used for communication between processes on the same machine.
- Communication is based on the file system, and sockets are represented as special files.
- Que no 10

## Q. Compare and contrast the Internet domain socket and Unix domain socket.

***(Asked 3 times)***

17. Describe different socket system calls used in TCP server with outline code.

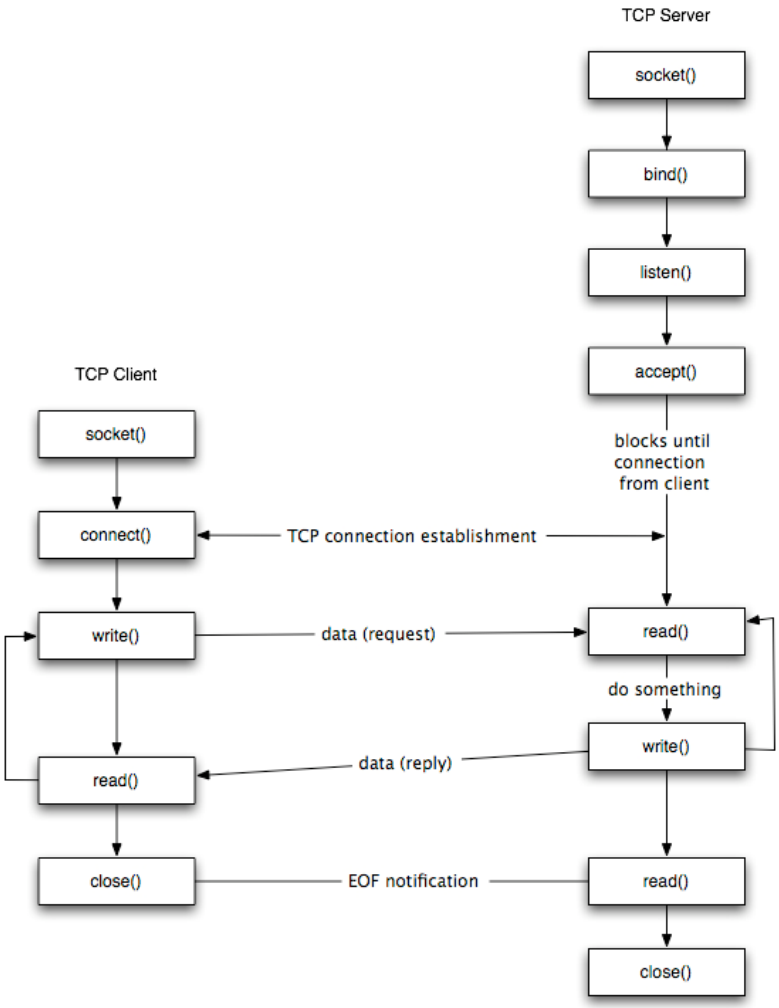
OR

Socket System Call.

OR

Explain different system call in specific order, required to create a TCP client and TCP server in the Unix System.

**Note: Refer Que no 3 once**



**TCP Server:**

1. socket()

- Create a socket, a communication endpoint for your server.
- This prepares the server to listen for incoming connections.
- System Call: socket(domain, type, protocol)
- Returns: non-negative descriptor if OK, -1 on error

2. bind()

- Choose an address (IP and port) for your server to listen on.
- System Call: bind(sockfd, addr, addrlen)
- It tells your server where to wait for incoming connections.
- Returns: 0 if OK,-1 on error

3. Listen()

- Set your socket to "listen" mode, ready to accept incoming connection requests.
- System Call: listen(sockfd, backlog)
- Explanation: Your server is waiting for someone to connect.

4. accept()

- When a client tries to connect, this function accepts the call and sets up a channel for communication.
- System Call: accept(sockfd, addr, addrlen)

5. read() and write()

- Use write() to send data to the client and read() to receive data from the client.
- System Calls: write(sockfd, buffer, length) and read(sockfd, buffer, length)
- Explanation: It's like writing and reading messages during a conversation.

6. close()

- End the communication and close the connection when the conversation is over.
- System Call: close(sockfd)

**TCP Client:**

1. socket()

- Create a socket to connect to the server.
- System Call: socket(domain, type, protocol)
- This prepares you to connect to the server.

2. connect()

- Use connect() to establish a connection to the server's address.
- System Call: connect(sockfd, serv_addr, addrlen)
- It's like dialing the phone number to reach the server.

3. read() and write()

- Exchange information with the server using write() and read() functions.
- System Calls: write(sockfd, buffer, length) and read(sockfd, buffer, length)

4. close()

- End the conversation and close the connection when done.
- System Call: close(sockfd)

***(Asked 2 times) (Asked in internal)***

## 12. What are the different arguments/parameters for socket() function call in Berkeley socket API.

- The socket() function call in the Berkeley socket API is used to create a new communication endpoint, which is a socket.
- It takes several arguments: the family (address family),type of communication, and the protocol.
- The function prototype for the socket() call is as follows:

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

**1. family (Address Family):**

- Specifies the protocol family or address family for the socket.
- Common values include:
  i. '**AF_INET'** for IPv4 (Internet Protocol version 4) addresses.
  ii. **'AF_INET6'** for IPv6 (Internet Protocol version 6) addresses.
  iii. '**AF_UNIX'** for Unix domain sockets (local communication on the same machine).

**2. type (Socket Type):**

- Specifies the type of socket.
- Common values include:
  i. '**SOCK_STREAM**' (TCP socket).
  ii. '**SOCK_DGRAM**' (UDP socket)
  iii. **'SOCK_RAW'** (raw socket)

3. **protocol**:

- Specifies the protocol to be used. Usually, you can use 0 to let the system choose the appropriate protocol based on the domain and type.
- For example, if you choose SOCK_STREAM (TCP), the protocol might be IPPROTO_TCP.
- Similarly, if you choose SOCK_DGRAM (UDP), the protocol might be IPPROTO_UDP.

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

int main() {
    int sockfd;

    // Create a TCP socket
    sockfd = socket(AF_INET, SOCK_STREAM, 0);

    if (sockfd == -1) {
        perror("socket");
        return 1;
    }

    printf("TCP socket created successfully.\n");

    return 0;
}
```

Fig: Example of creating a TCP socket using socket() function

**(Asked 2 times)**

## 13. What is the purpose of bind() function.

- The bind() function is used to associate a specific IP address and port number with a socket.
- By doing this, the socket becomes "bound" to that particular combination of IP address and port, and it can then send and receive data using that network address.
- This step is essential, especially for server applications, before the socket can be used for network communication.
- This allows the operating system to direct incoming data to the correct socket.
- Here's the syntax of the bind() function:

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

i. **sockfd**: The socket file descriptor obtained from the socket() function call.
ii. **addr**: A pointer to a struct sockaddr structure containing the local address information to which the socket should be bound.
iii. **addrlen**: The size (in bytes) of the struct sockaddr structure.

Here's a step-by-step explanation of how the bind() function works:

**1. Create a Socket:**

- First, create a socket using the socket() function.
- The returned socket file descriptor (sockfd) will be used in the bind() function call.

**2. Prepare Address Structure:**

- Create a struct sockaddr and populate it with the local address information to which you want to bind the socket.
- Set the address family, IP address, and port number in the address structure.

**3. Call the bind() Function:**

- Pass the socket file descriptor (sockfd), a pointer to the address structure (addr), and the size of the address structure (addrlen) to the bind() function.

**4. Error Handling:**

- Check the return value of the bind() function. If it returns -1, an error occurred.

## 14. What will be the outcome if we do not specify IP address, port, both or neither in bind() function.

- The outcome of calling the bind() function without specifying the IP address, port, both, or neither depends on the situation and the specific requirements of the program.
  The various situations are:

**1. Not Specifying IP Address or Port:**

- If we don't specify the IP address or port in the bind() function, the socket will be bound to the wildcard address INADDR_ANY and an available random port (selected by the operating system).
- INADDR_ANY is a special IP address that is used when you want to bind a socket to any available network interface on the host machine.

**2. Not Specifying IP Address but Specifying Port:**

- If we specify only the port in the bind() function, the socket will be bound to the wildcard address INADDR_ANY and the specified port.
- This is helpful when we want to listen on a specific port across all available network interfaces.

**3. Specifying IP Address but Not Port:**

- If we specify only the IP address in the bind() function and not the port, we'll encounter a compilation error because the struct sockaddr_in structure requires both the address and port fields to be filled.

**4. Specifying Both IP Address and Port:**

- This is the typical usage of the bind() function.
- We specify the local IP address and the port to bind the socket to a specific endpoint.

## 15. List out and explain TCP server socket listen() function along with its completed and pending queue with suitable code.

- The listen() function is used with a TCP server socket to indicate that the socket is ready to accept incoming client connections.
- It sets up the socket to listen for incoming connection requests from clients.

*int listen(int sockfd, int backlog);*

i. **sockfd**: The socket file descriptor obtained from the socket() function call.
ii. **backlog**: The maximum number of pending connection requests that can be queued up while waiting to be accepted. It represents the size of the "pending connection queue."

Steps:

- Create a socket using the socket() function.
- Use the bind() function to associate the socket with a local address.
- Call the listen() function to start listening for incoming connection requests.
- The server is now ready to accept incoming connections from clients.

**Completed and Pending Connection Queues:**

- The listen() function creates two queues associated with the socket: the "completed connection queue" and the "pending connection queue."
- The "**completed connection queue**" holds the established connections that have been accepted by the server but haven't been processed yet.
- The "**pending connection queue**" holds the incoming connection requests that are waiting to be accepted by the server.

## 16. What is a connection queue? What are the possible circumstances that might cause connection() function to return an error.

- A connection queue, also known as a "pending connection queue," is a buffer or storage area maintained by a server to hold incoming connection requests from clients that are waiting to be accepted.
- When a client tries to establish a connection with a server, the server may not immediately be able to process the connection.
- Instead, the connection request is placed in this queue until the server can accept and process it.

- Possible circumstances that might cause the connect() function to return an error in simple words:

i. **Server Unavailable**: If the server is not running or not reachable on the specified IP address and port, the connect() function might return an error.

ii. **Connection Refused**: If the server is reachable but actively refuses the connection request, the connect() function can return an error.

iii. **Network Issues**: If there are problems with the network, such as a loss of connectivity or an unstable connection, the connect() function may not be able to establish a connection and can return an error.

iv. **Firewalls or Security**: Firewalls or security measures on either the client or server side can block or restrict connections.

v. **Timeout**: If the connect() function takes too long to establish a connection (exceeds the timeout duration), it might return an error.

vi. **Invalid Address or Port**: If the provided IP address or port number is incorrect or invalid, the connect() function will not be able to find the server and can return an error

## 17. Explain accept() and close() function with its argument in simple words and explanation.

**1. accept() Function:**

- The accept() function is used in server-side programming to accept incoming client connections on a listening socket.
- It creates a new socket that represents the established connection with the client.
- This new socket is used to communicate with that specific client.

Syntax:

*int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);*

i. sockfd: The socket file descriptor returned by the socket() function and previously used with the bind() and listen() functions.
ii. addr: A pointer to a structure that will be filled with the client's address information.
iii. addrlen: A pointer to the length of the address structure.

Explanation:

- The accept() function is called on a listening socket.
- It waits until an incoming client connection request arrives.
- When a connection request is received, accept() creates a new socket specifically for that client connection.
- The original listening socket (sockfd) remains available for further connection requests.
- The new socket is returned by the accept() function. It's used to communicate with the specific client.
- The client's address information is stored in the addr structure, and the length of the structure is stored in the addrlen variable.

**2. close() Function:**

- The close() function is used to close a socket, terminating the communication on that socket.
- After closing, the socket is no longer available for sending or receiving data.

Syntax:

*int close(int sockfd);*

i. **sockfd**: The socket file descriptor to be closed.

**Explanation**:

- The close() function is called on a socket that you want to close.
- Once the function is executed, the socket is closed, and the resources associated with it are released.
- The file descriptor (sockfd) is no longer valid and should not be used for any further communication.
- Any data in the socket's send or receive buffers that has not been fully transmitted will be discarded.
- It's essential to close sockets properly to avoid resource leaks and ensure proper communication termination.

In simple words, accept() is used to accept incoming client connections, creating a new socket for communication, while close() is used to gracefully close a socket, ending communication and releasing associated resources.

**(Asked 3 times)**

## 18. What do you mean by socket descriptor?

- Socket descriptor is a unique number that the operating system assigns to each socket when it's created using the socket() function.
- This descriptor acts like an identifier or handle that your program uses to refer to the socket during its lifetime.
- In simple words, think of a socket descriptor as a special ID or label that helps your program keep track of the different communication channels (sockets) it opens.
- You use this ID to perform various operations on the socket, like sending and receiving data, closing the connection, and more.
- For example, when you create a socket like this:

*int sockfd = socket(AF_INET, SOCK_STREAM, 0);*

- The **sockfd** variable holds the socket descriptor for the newly created socket.
- You use this descriptor to interact with the socket throughout your program. It's how your program knows which socket it's talking about among potentially many sockets.

**(Asked 2 times)**

## 19. What do you mean by file descriptor? Write its importance.

- A file descriptor is a unique number assigned by the operating system to each open resource, including files, network sockets, and other I/O resources.
- This number is used by the program to interact with the resource.
- Simply, File descriptors are nonnegative integers that the kernel uses to identify the file being accessed by a particular process.
- Programs use file descriptors to perform operations on resources, like reading data from files or sending data over a network connection.
- Whenever the kernel opens an existing file or creates a new file it returns the file descriptor.

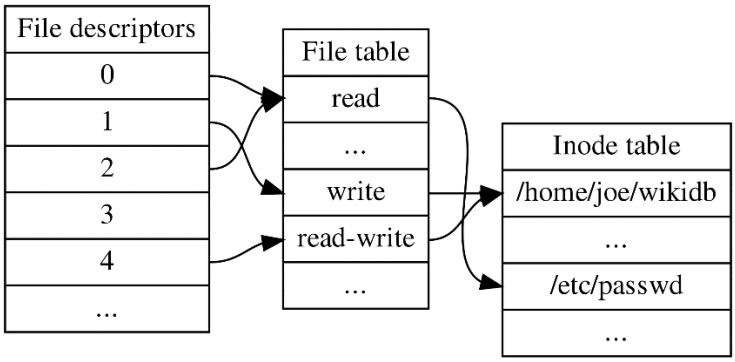



Fig: File Descriptor

**Importance of file descriptors:**

**1. Resource Management:**

- File descriptors help the operating system keep track of open resources within a process.
- This ensures that resources are properly managed, avoiding issues like resource leaks or conflicts.

**2. Input/Output**:

- They enable programs to read data from and write data to files and other input/output channels.

**3. Abstraction**:

- File descriptors abstract the underlying details of resource handling.
- Developers can work with files, network sockets, and other resources using a consistent interface, regardless of the underlying complexities.

**4. Standardization:**

- File descriptors provide a standardized way for programs to interact with different I/O resources.
- Regardless of whether the program is dealing with a file on disk or a network socket, it can use similar operations like reading, writing, and closing.

**5. Efficiency**:

- File descriptors allow multiple processes to share the same resource efficiently.
- Instead of duplicating data for each process, the operating system keeps track of open resources using descriptors, saving memory and improving performance.

**6. Portability**:

- File descriptors enable code to be written in a way that can work across different systems and environments.
- The same code that handles file operations can often be adapted to work with sockets or other resources without significant modifications.

**7. Error Handling:**

- Error Handling: They are used to report errors and exceptions related to resource operations.

**(Asked 4 times)**

## 20. What is file descriptor passing?

## OR

## Explain the mechanism of passing file descriptor in the UNIX system.

- File descriptor passing in network programming refers to the technique of transferring open file descriptors between processes over a network connection.
- This technique is used to enable one process to share a file descriptor with another process, allowing them to communicate using the same file or socket.
- It's particularly useful in scenarios where processes need to collaborate or share resources across a network.

File descriptor passing involves the following steps:

**1. Sending Process (Sender):**

- The sending process has an open file descriptor (e.g., a socket or a regular file).
- Instead of sending the actual file content, the sender sends the file descriptor over the network to the receiving process.
- The sending process uses the **sendmsg**() system call to send the file descriptor.

**2. Receiving Process (Receiver):**

- The receiving process accepts the file descriptor sent by the sender using the **recvmsg**() system call.
- Once received, the receiving process can use the shared file descriptor to perform operations on the same file or socket.

**(Asked 3 times)**

## 21. What is I/O model. List I/O models in UNIX system. Elaborate all of them.

- The I/O (Input/Output) model in Unix programming refers to how programs handle input and output operations, such as reading from and writing to files, sockets, and other communication channels.
- Different I/O models define how these operations are carried out and how they impact the program's behavior.
- There are 5 types of I/o models in UNIX programming.
- They are listed below:
    a. Blocking I/O
    b. Non-Blocking I/O
    c. I/O multiplexing (select/poll/epoll)
    d. Signal-driven I/O(Sigio)
    e. Asynchronous I/O

**a. Blocking I/O:**

- **What it does**: Blocks the program until the I/O operation is done.
- **How it works:**
    - When the program performs a read or write operation, it waits until the data transfer is finished.
    - During this time, the program is "blocked" and cannot continue executing other tasks.
- **When to use**: When you want to make sure data is read/written completely before moving on.
- **Example**: Imagine a server using blocking I/O to receive data from clients. When a client sends a message, the server reads from the socket and waits until the message is fully received before processing it.

**b. Non-blocking I/O:**

- **What it does**: Non-blocking I/O returns immediately, allowing the program to continue executing.

- **How it works:**
  - When the program performs a read or write operation, it returns immediately, whether the operation succeeds or not.
  - The program must actively check if the data is ready for reading or if the system is ready for writing.
- **When to use:** Use non-blocking I/O when you want the program to keep running even if the I/O operation isn't ready yet.
- **Example**: A client using non-blocking I/O tries to read data from a socket. If data is available, it's read immediately. If not, the program keeps running without waiting.

**c. I/O Multiplexing (select/poll/epoll):**

- **What it does**: Lets the program monitor multiple I/O sources simultaneously, waiting for any of them to be ready.
- **How it works:**
  - The program uses functions like select(), poll(), or epoll() to watch multiple file descriptors.
  - It waits until any of the descriptors are ready for I/O, and then it can perform the operation on the ready descriptor.
- **When to use**: When you need to manage multiple I/O operations without blocking.
- **Example**: A server using select() watches several sockets for incoming client data. When any of the sockets have data ready, the server reads from the respective socket.

**d. Signal-Driven I/O:**

- In this model, the program sets up a signal handler for I/O events. When an I/O operation completes, the operating system sends a signal to the program, allowing it to handle the I/O data.
- **What it does**: Combines non-blocking I/O with asynchronous notifications using signals.
- **How it works**:
  - The program sets up a signal handler to receive a notification (signal) when the I/O is ready.

    - It starts the I/O operation and then continues with other tasks. When the I/O is ready, a signal triggers the handler.
- **When to use**: When you want to perform other tasks while waiting for I/O, and you want to be notified asynchronously when I/O is ready.
- **Example**: A program sets up a signal handler to receive a notification when data arrives on a socket. It then continues with other tasks and gets notified when data is available to read.

**e. Asynchronous I/O:**

- This model allows programs to initiate I/O operations and continue executing other tasks without waiting for the operations to complete.
- When an operation finishes, the program is notified, and it can then handle the data.
- This is more advanced and complex compared to the other models.

- **What it does**: initiates operations and continues execution, getting notified when I/O is finished.
- **How it works:**
    - The program starts an I/O operation and continues working on other tasks.
    - When the I/O is done, the system notifies the program, which can then process the completed operation.
- **When to use**: Use asynchronous I/O when you want to do other tasks while waiting for I/O and receive notifications when the operation is done.
- **Example**: A program initiates a file read operation asynchronously. It continues executing other tasks and is notified by the system when the read is completed, allowing it to use the read data.

**(Asked 3 times)**

## 22. Explain blocking and non-blocking I/O model with diagram.

### a. Blocking I/O model

- In this model, when an I/O operation is requested, the program blocks (waits) until the operation is complete.
- I/O operations are handled one after the other, in the order they were requested.
- It's a straightforward model but can lead to inefficiencies.
- If an I/O operation takes a long time, the whole program stops and waits, leading to inefficiency.

- **What it does**: Blocks the program until the I/O operation is done.
- **How it works:**
  - When the program performs a read or write operation, it waits until the data transfer is finished.
  - During this time, the program is "blocked" and cannot continue executing other tasks.
- **When to use**: When you want to make sure data is read/written completely before moving on.
- **Example**: Imagine a server using blocking I/O to receive data from clients. When a client sends a message, the server reads from the socket and waits until the message is fully received before processing it.

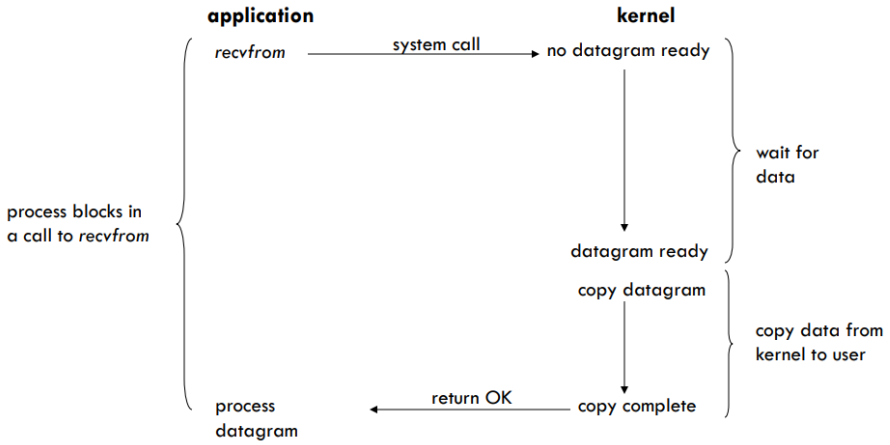



Fig: Blocking I/O model

**Details**:

- By default, all sockets operate in a blocking mode.
- When a process calls a function like recvfrom, it enters a waiting state. It won't continue until the requested data arrives or an error happens.
- During this wait, we say the process is "blocked." It's not doing anything else; it's just waiting for the data.
- The process remains blocked from the moment it calls recvfrom until it gets the data or encounters an error.
- Once recvfrom successfully returns, the process can process the received data.

**b. Non-blocking I/O model**

- Program initiates I/O and keeps doing other things while checking periodically for completion
- If the data is not ready for reading or the system isn't ready for writing, the operation doesn't wait and the program keeps running.
- The program needs to actively check if the I/O operation can be completed.
- Efficient for handling multiple I/O sources, but it keeps the program busy with constant checking.

- **What it does**: Non-blocking I/O returns immediately, allowing the program to continue executing.

- **How it works:**
  - When the program performs a read or write operation, it returns immediately, whether the operation succeeds or not.
  - The program must periodically check if the data is ready for reading or if the system is ready for writing.
- **When to use:** Use non-blocking I/O when you want the program to keep running even if the I/O operation isn't ready yet.
- **Example**: A client using non-blocking I/O tries to read data from a socket. If data is available, it's read immediately. If not, the program keeps running without waiting.

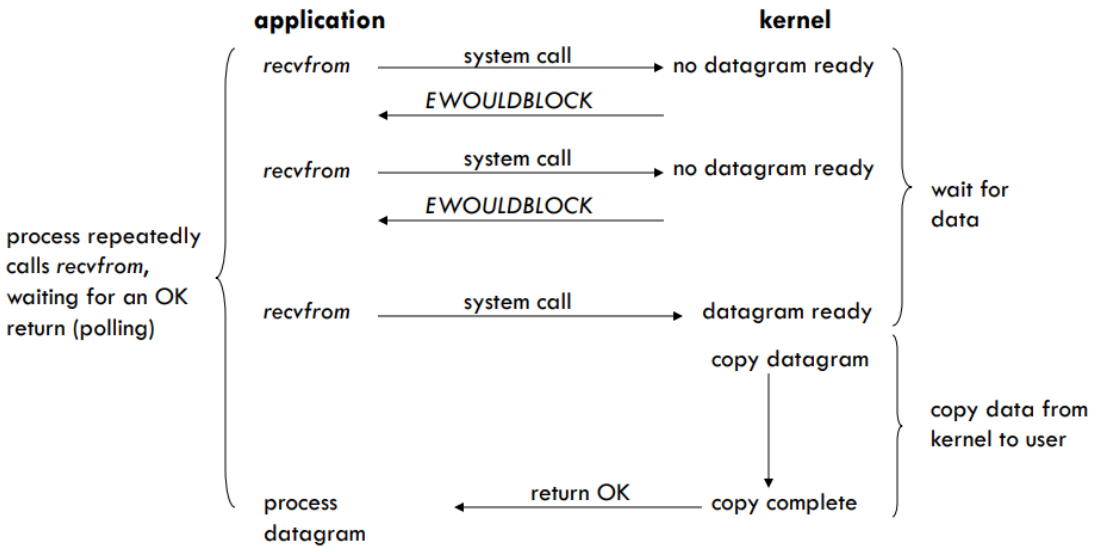



Fig: Non-blocking I/O model

***(Asked 3 times)***

## 23. Which function is used to convert sockets into non-blocking mode? Write syntax of the function.

## OR

## Explain with the help of code how fcntl() function sets blocking socket to non-blocking socket.

- The function used to set a socket into non-blocking mode is fcntl() (file control).
- We can use the fcntl() function with the F_SETFL command and the O_NONBLOCK flag.

Syntax:

```
#include <fcntl.h>

int flags = fcntl(sockfd, F_GETFL, 0);

flags |= O_NONBLOCK;
```

*fcntl(sockfd, F_SETFL, flags);*

In above syntax:

i. **sockfd**: Socket file descriptor for the socket you want to modify to non-blocking.
ii. The first call to fcntl() with F_GETFL retrieves the current settings(flags) of the socket referred by sockfd.
iii. The O_NONBLOCK flag is bitwise ORed with the existing flags to set the non-blocking mode.
iv. The second call to fcntl() with F_SETFL applies the modified flags to the socket, effectively enabling non-blocking mode.

Above steps in simple terms:

i. We're checking how the socket currently operates (blocking or non-blocking).
ii. Then we're making a change to the socket to set it into non-blocking mode.
iii. Finally, we're applying this change to the socket, so it starts working in non-blocking mode.

After this process, the socket will operate in non-blocking mode, allowing read and write operations to return immediately without waiting for data to be ready, making the program more responsive.

***(Asked 3 times)***

## 24. Does getaddrinfo() is blocking or non-blocking?

- The getaddrinfo() function is a blocking function.
- It is used to convert a human-readable host name or service name into an address that can be used by computer for communication.
- However, the blocking behavior of getaddrinfo() is generally short-lived and should not significantly impact the responsiveness of your application.

**Blocking Vs Non-Blocking**

- **Blocking functions** are those that pause the execution of the program until they complete their task. They can take some time to finish, causing the program to wait.
- **Non-blocking functions**, on the other hand, return immediately, even if the task is not complete. They allow the program to continue executing while the task is being performed in the background.

**Blocking behaviour of getaddrinfo()**

- When you call getaddrinfo() to resolve a host name or service name, it queries the DNS (Domain Name System) or other relevant services to obtain the corresponding IP address and other information.
- This query is a network operation and can involve some network communication and possibly DNS lookups.
- During this process, your program may pause briefly while getaddrinfo() performs the query and retrieves the necessary information.
- However, this blocking period is usually short and doesn't affect the overall performance of your application.
- Since it might talk to networks, the speed depends on the network response and the availability of cached information.

In summary, getaddrinfo() is a blocking function, but its blocking behavior is usually short-lived and does not cause significant delays in most applications.

***(Asked 4 times)***

## 25. List out the different system call used to create TCP and UDP client server on the basis of blocking and non-blocking in nature. Also explain them in very short.

**Blocking I/O with TCP Socket:**

**TCP Server:**

- The server creates a TCP socket using socket() to establish a reliable communication.
- It binds the socket to a specific address and port using bind().
- The server listens for incoming connections using listen().
- When a client connects, the server accepts the connection using accept() and gets a new socket descriptor (client_sockfd).
- **The server uses read() to blockingly receive data from the client.**
- It sends a response message using write() back to the client.
- The server closes the socket using close().

**TCP Client:**

- The client creates a TCP socket similarly to the server.
- It connects to the server using connect() and the server's address and port.
- The client uses write() to send a message to the server.
- **It then uses read() to blockingly receive a response message from the server.**
- The client closes its socket using close().

**Blocking I/O with UDP Socket:**

**UDP Server:**

- The server creates a UDP socket using socket() to establish datagram communication.
- It binds the socket to a specific address and port using bind().
- **The server uses recvfrom() to blockingly receive data from any client.**
- It identifies the client's address and port from the received message.
- The server responds using sendto() to the client's address and port.
- The server closes the socket using close().

**UDP Client:**

- The client creates a UDP socket similarly to the server.
- It uses sendto() to send a message to the server's address and port.
- **The client uses recvfrom() to blockingly receive a response from the server.**
- The client closes its socket using close().

**Non-Blocking System Calls: (Normal flow )**

**TCP Server:**

i. socket()
ii. bind()
iii. listen()
iv. accept()
v. read()
vi. write()
vii. close()

**TCP Client:**

i. socket()
ii. connect()
iii. write()
iv. read()
v. close()

**UDP Server:**

i. socket()
ii. bind()
iii. recvfrom()
iv. sendto()
v. close()

**UDP Client:**

i. socket()
ii. sendto()
iii. recvfrom()
iv. close()

- In both blocking and non-blocking cases, the essential system calls are quite similar.
- The difference lies in whether the calls block (wait for completion) or return immediately (without waiting).

## Q. How Blocking I/O model, TCP and UDP socket read and write message to and from the kernel buffer.

- Ans: Que no 25, blocking part.

***(Asked 2 times)***

## 26. What is I/O multiplexing. What are the functions used int this I/O model.

- **What it does**: Lets the program monitor multiple I/O sources simultaneously, waiting for any of them to be ready.
- **How it works:**
  - The program uses functions like select(), poll(), or epoll() to watch multiple file descriptors.
  - It waits until any of the descriptors are ready for I/O, and then it can perform the operation on the ready descriptor.
- **When to use**: When you need to manage multiple I/O operations without blocking.
- **Example**: A server using select() watches several sockets for incoming client data. When any of the sockets have data ready, the server reads from the respective socket.

  **Functions for Synchronous I/O Multiplexing in Berkeley Socket API:**

  i. **select():**This function monitors a set of sockets for readability, writability, or exceptional conditions. It blocks until any of the specified conditions occur on any of the monitored sockets.

  ii. **pselect():** Similar to select(), but with additional control over the signal mask to avoid race conditions.

  iii. **poll():** Similar to select(), poll() also waits for events on array of sockets, but it offers more flexibility and is often considered more efficient for managing large numbers of sockets.

  iv. **epoll():** Provides a more scalable and efficient interface for monitoring large numbers of sockets

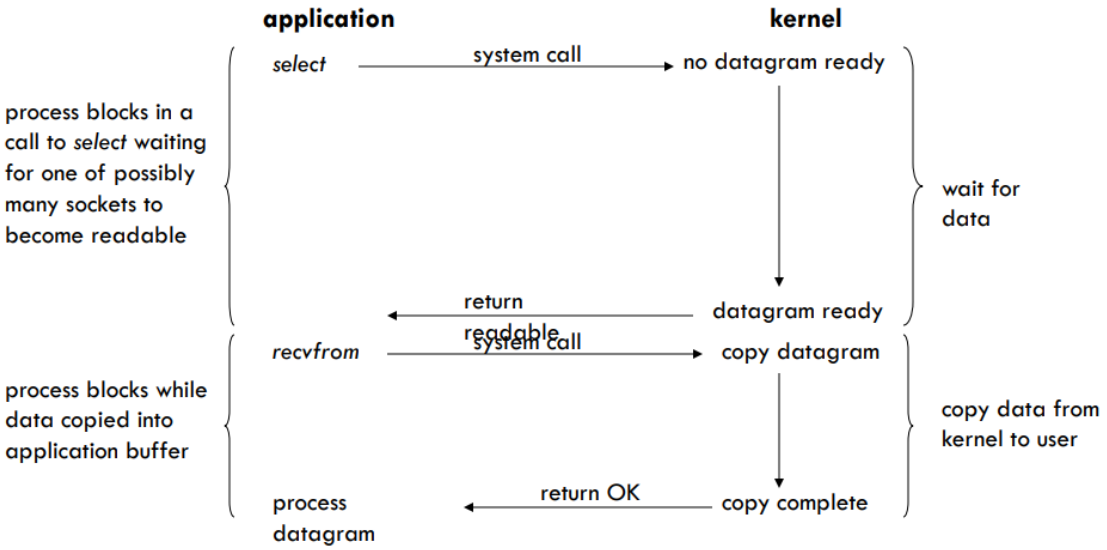



Fig: I/O multiplexing

***(Asked 3 times)***

## 27. Explain the use of select function in the context of I/O multiplexing in detail.

## OR

## How can a server handle multiple clients without using fork function? Write such a server application using select function (I/O multiplexing)

## OR

## How can a TCP server handle multiple clients without doing fork() system call.

- The **select()** function helps server to handle multiple clients without using **fork()**.
- It allows a single process to efficiently monitor and manage multiple client sockets.
- Simply, by using select() the server can handle multiple clients without creating separate processes (using fork()) for each client.

**Handling Multiple Clients with I/O Multiplexing:**

**1. Server Setup:**

- The server creates a socket and binds it to a specific address and port.
- The server maintains a set of sockets (server socket and client sockets) it wants to monitor.
- This set is divided into subsets, like sockets ready for reading, writing, or exceptional conditions.

**2. Using select():**

- The server uses the select() function to wait for activity on any of the sockets that it is monitoring.
- It specifies a timeout or waits indefinitely until activity occurs.

**3. Monitoring Sockets:**

- While waiting, the server doesn't actively check each client socket.
- It lets the operating system monitor and notify when data arrives or a new client connects.

**4. Activity Detected:**

- When data arrives on a client socket or a new client connects, the corresponding socket becomes "ready."
- The select() function returns, and the server identifies the ready socket using the subsets.

**5. Processing Ready Sockets:**

- The server processes the data on the ready sockets or handles new client connections.
- It can read data from clients, write responses, or perform other actions.

**6. Non-Blocking Behavior:**

- The server can make the sockets non-blocking, allowing it to perform other tasks while waiting for activity.

**7. Single Process, Multiple Clients:**

- With select(), the server manages multiple client sockets within a single process.
- It doesn't need to create a new process ( **fork()** ) for each client.

**8. Closing and Cleanup:**

- If a client disconnects, the server closes the client socket and removes it from the active sockets set.

In essence, the select() function acts as a traffic controller for sockets, allowing a single process to efficiently manage multiple clients by waiting for activity to occur on any of the monitored sockets.

This approach eliminates the need for excessive process creation (fork()) and ensures that the server can handle multiple clients with good resource utilization and responsiveness.

## Q. Compare synchronous I/O multiplexing with non blocking I/O mode. What are the different functions used to implement these I/O models in Berkeley socket API.

Ans: Que no 22 and 26

**Functions for Synchronous I/O Multiplexing in Berkeley Socket API:**

i. **select()**: Monitors a set of sockets for readiness to read, write, or exceptional conditions.
ii. **pselect()**: Similar to select(), but with additional control over the signal mask to avoid race conditions.
iii. **poll():** Monitors an array of sockets for events (read, write, error) and returns information about the ready sockets.
iv. **epoll():** Provides a more scalable and efficient interface for monitoring large numbers of sockets.

Functions for Non-blocking I/O in Berkeley Socket API:

i. socket(): Create a socket.
ii. bind(): Bind a socket to a specific address and port.
iii. listen(): Listen for incoming connections.
iv. accept(): Accept incoming connections (may block depending on the blocking behavior of the socket).
v. connect(): Initiate a connection to a remote host.
vi. fcntl(): Control the properties of a socket, including setting it to non-blocking mode.
vii. read(), recvfrom(): Read data from a socket (may return immediately if no data is available).
viii. write(), sendto(): Send data through a socket (may return immediately if the socket's buffer is full).

## 28. Explain signal driven I/O with diagram.

- In this model, the program sets up a signal handler for I/O events. When an I/O operation completes, the operating system sends a signal to the program, allowing it to handle the I/O data.
- **What it does**: Combines non-blocking I/O with asynchronous notifications using signals.
- **How it works**:
  - The program sets up a signal handler to receive a notification (signal) when the I/O is ready.
  - It starts the I/O operation and then continues with other tasks. When the I/O is ready, a signal triggers the handler.
- **When to use**: When you want to perform other tasks while waiting for I/O, and you want to be notified asynchronously when I/O is ready.
- **Example**: A program sets up a signal handler to receive a notification when data arrives on a socket. It then continues with other tasks and gets notified when data is available to read.

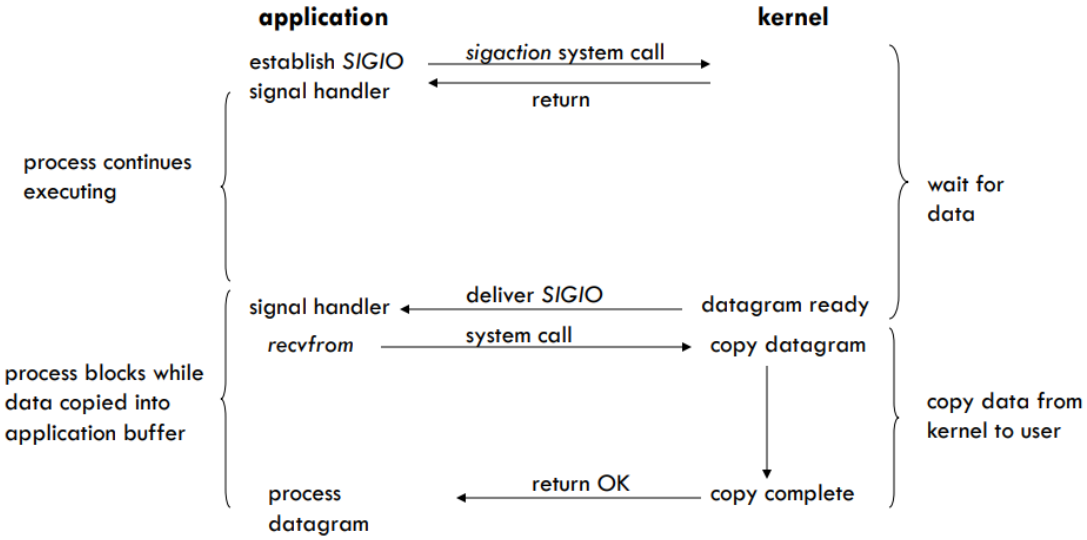



**Fig: Signal driven I/O**

## 29. Explain asynchronous model with diagram.

- This model allows programs to initiate I/O operations and continue executing other tasks without waiting for the operations to complete.
- When an operation finishes, the program is notified, and it can then handle the data.
- This is more advanced and complex compared to the other models.

- **What it does**: initiates operations and continues execution, getting notified when I/O is finished.
- **How it works:**
  - The program starts an I/O operation and continues working on other tasks.
  - When the I/O is done, the system notifies the program, which can then process the completed operation.
- **When to use**: Use asynchronous I/O when you want to do other tasks while waiting for I/O and receive notifications when the operation is done.
- **Example**: A program initiates a file read operation asynchronously. It continues executing other tasks and is notified by the system when the read is completed, allowing it to use the read data.

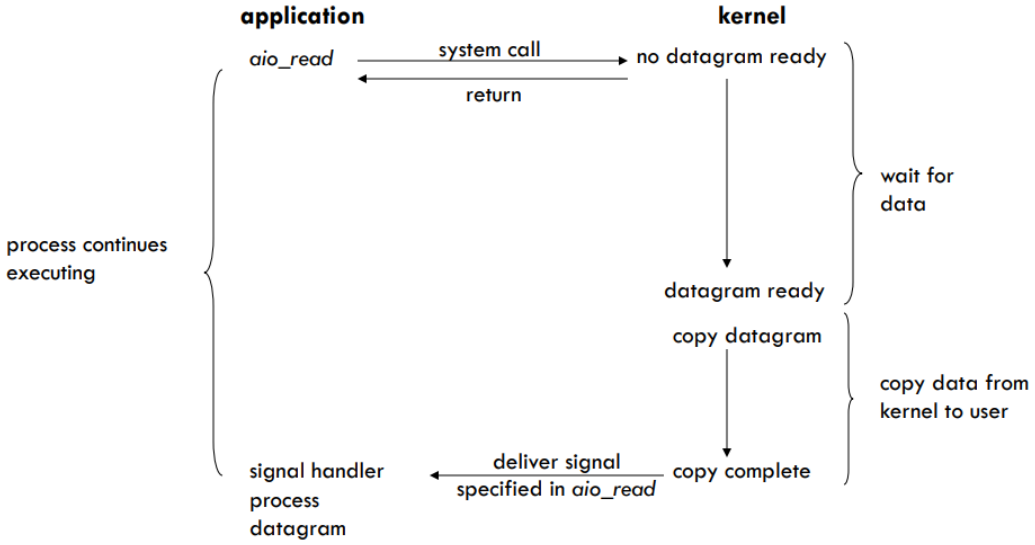



**Fig: Asynchronous I/O model**

## 30. Compare and contrast synchronous and asynchronous I/O models for UNIX programming.

**1. Blocking vs. Non-blocking:**

- Synchronous I/O blocks the program until the I/O operation is complete.
- Asynchronous I/O allows the program to continue executing without waiting for the I/O operation to complete.

**2. Program Flow:**

- Synchronous I/O has a straightforward, linear program flow.
- Asynchronous I/O has a more complex, potentially non-linear program flow due to callbacks and notifications.

**3. Concurrency**:

- Synchronous I/O doesn't handle concurrency well and can lead to inefficient use of resources.
- Asynchronous I/O enables efficient concurrency, allowing the program to perform other tasks while waiting for I/O.

**4. Complexity:**

- Synchronous I/O is simpler to implement and understand.
- Asynchronous I/O is more complex due to the need for managing callbacks, events, and synchronization.

**5. Efficiency and CPU Utilization:**

- Synchronous I/O may waste CPU time during I/O wait, leading to inefficiency in handling concurrent tasks.
- Asynchronous I/O utilizes CPU more efficiently, enabling concurrent handling of multiple I/O operations.

**6. Multitasking and Scalability:**

- Synchronous I/O is not suitable for multitasking environments and may have limited scalability for concurrent tasks.
- Asynchronous I/O is suitable for multitasking, highly scalable for handling numerous concurrent tasks.

**6. Use Cases:**

- Synchronous I/O is suited for simpler applications where performance and concurrency are not primary concerns.
- Asynchronous I/O is well-suited for applications requiring high responsiveness, concurrency, and efficient resource utilization.

## 31. Compare TCP and UDP sockets on the basis of the socket call and I/O function.

**Socket Call:**

**1. TCP Socket:**

- To create a TCP socket, you use the socket() function with the domain AF_INET (IPv4) or AF_INET6 (IPv6), and the type SOCK_STREAM to indicate a stream-oriented socket.

**2. UDP Socket:**

- To create a UDP socket, you also use the socket() function with the same domain (e.g., AF_INET) but use the type SOCK_DGRAM to indicate a datagram-oriented socket.

**I/O Functions:**

**1. TCP Socket:**

- The read() function in TCP sockets blocks until there is data available to read from the remote end.

- The write() function in TCP sockets does not necessarily block until the data is sent successfully.
- It can block if the send buffer is full.

**UDP Socket:**

- UDP sockets use recvfrom() and sendto() functions for reading from and writing to the socket.
- recvfrom() blocks until data is received, and sendto() sends data to a specific destination without establishing a connection.

**(Asked 2 times)**

## 32. What are the socket options? Which functions are used to set and get a value of socket options? Explain them in detail.

## OR

## Why are the getsockopt and setsockopt used?

- Socket options are settings that you can adjust for a socket to customize its behavior and characteristics.
- They allow you to control various aspects of socket communication, such as timeout values, buffer sizes, and socket behavior.
- Two functions are commonly used to set and get the values of socket options:
  i. **setsockopt():** This function is used to set the value of a socket option. It allows you to configure how the socket behaves.
  ii. **getsockopt():** This function is used to retrieve the value of a socket option. It lets you query the current settings of a socket option.

Here's a simple explanation of these functions:

**i. setsockopt() - Set Socket Option:**

- Suppose you want to control the timeout period for socket operations, like how long the socket should wait for data to arrive.
- You can use setsockopt() to set this timeout value.
- You pass the socket descriptor, the level at which the option resides (like SOL_SOCKET for general socket options), the specific option you want to set (like SO_RCVTIMEO for receive timeout), and the value you want to set.
- For example, to set the receive timeout to 5 seconds:

*int timeout_seconds = 5;*

*setsockopt(socket_descriptor, SOL_SOCKET, SO_RCVTIMEO, &timeout_seconds, sizeof(timeout_seconds));*

**ii. getsockopt() - Get Socket Option:**

- If you want to know the current value of a socket option, you use getsockopt().
- You pass the socket descriptor, the level of the option, the specific option you're interested in, and a pointer to a variable where the value will be stored.
- For example, to retrieve the receive timeout value:

int timeout_seconds;

socklen_t option_length = sizeof(timeout_seconds);

getsockopt(socket_descriptor, SOL_SOCKET, SO_RCVTIMEO, &timeout_seconds, &option_length);

- In simple words, socket options are like settings that determine how a socket behaves.
- You can use setsockopt() to adjust these settings to your needs and getsockopt() to check the current settings.

## 33. Give your reason why we required to use getsockname() and get peername() function.

**1. getsockname() Function:**

**i. Know Your Own Address:**

- When you create a socket and bind it to a local address, you might want to know which address and port your socket is associated with.
- getsockname() helps you retrieve your own socket's local address and port.

**ii. Server Applications:**

- In server applications, you need to communicate your address and port to clients so they can connect to you.
- getsockname() provides the information you need to share with clients.

**2. getpeername() Function:**

**i. Client-Server Interaction:**

- In a client-server scenario, the server might want to know the address and port of the client that's connected to it.
- getpeername() helps the server get the details of the connected client.

**ii. Security Checks:**

- Servers might perform security checks based on the identity of connected clients.
- getpeername() helps servers identify and validate the clients they're interacting with.

## 34. Explain the set and get keepalive on socket.

Setting and getting the "keep-alive" option on a socket is a way to manage and maintain the connection between two communicating parties, such as a client and a server, even if there's no actual data being exchanged.

**Setting Keep-Alive:**

- **Purpose**: Setting the "keep-alive" option on a socket enables the operating system to periodically send small packets (keep-alive probes) to check if the other end of the connection is still responsive.
- **Usage**: To enable keep-alive on a socket, you use the setsockopt() function with the SO_KEEPALIVE option.

**Getting Keep-Alive:**

- **Purpose**: Getting the "keep-alive" option lets you check whether the "keep-alive" feature is enabled on the socket.
- **Usage**: To check if keep-alive is enabled on a socket, you use the getsockopt() function with the SO_KEEPALIVE option.

Summarizing, setting the "keep-alive" option on a socket helps maintain the connection's liveliness by periodically sending small packets.

This is useful to ensure that the connection is still functional, even if no actual data is being transferred.

You can use the setsockopt() function with SO_KEEPALIVE to enable this feature and getsockopt() to check whether it's enabled on a socket.

***(Asked 2 times)***

## 35. What is the daemon process? How does the daemon process gets started?

- A daemon process is a type of computer program that operates in the background, without requiring a user to interact with it directly.
- Unlike regular programs that start and stop with user actions, daemons keep running as long as the system is running.
- It typically performs specific tasks related to network services, often running continuously to provide services to other programs or users.
- Daemons are often started automatically when the computer boots up.
- Examples of daemons include Apache (web server), OpenSSH (remote access), and cron (scheduled tasks)

**Starting Daemon Processes:**

i. **During System Startup**: When your computer starts up, certain programs called daemons can also start. These daemons are often managed by the system itself and have special permissions.

ii. **inetd Superserver**: Some daemons are started by a special program called inetd. It listens for network requests and launches the actual servers when needed.

iii. **Cron Daemon**: Another daemon called cron manages programs that need to run regularly. It launches these programs at specific times.

iv. **"at" Command**: There's a command called "at" that lets you schedule a program to run later. The cron daemon handles these programs too.

v. **User Terminals**: You can also start daemons manually from your own terminal. This is useful for testing or restarting.

## 36. Explain how to create a daemon process. (Demonize a process)

**Creating a daemon process**

The steps included are:

**1. Fork**:

- We start by creating a copy of the current process using fork().
- The original process (parent) terminates so that the child process can continue running independently.
- This allows the child process to run in the background without needing the parent.

**2. setsid:**

- The child process calls setsid(), which creates a new session for it.
- The child becomes the leader of this session and also the leader of a new process group.
- This helps the process to detach from its original terminal and become independent.

**3. Ignore SIGHUP and Fork Again:**

- The child process ignores the SIGHUP signal, which is sent when a terminal connection is lost.
- We call fork() again, creating a grandchild process.
- The original child process terminates, ensuring the grandchild isn't associated with a terminal.

**4. Change Working Directory:**

- The daemon process changes its working directory to the root directory.
- This helps prevent problems when working with files.

**5. Close Any Open Descriptors:**

- We close any open file descriptors inherited from the parent process.
- This avoids potential conflicts with resources that the daemon doesn't need.

**6. Redirect stdin, stdout, and stderr to /dev/null:**

- We redirect standard input, output, and error streams to /dev/null.
- This ensures that any input/output from the daemon is discarded.

**7. Use syslogd for Errors:**

- The daemon uses the syslogd daemon to log error messages.
- This helps in centralized error logging and management.

## 37. Why log management is important in programming?

- Importance is listed below:

1. **Troubleshooting**: Logs provide a historical record of events, helping developers identify and fix issues in applications.

2. **Monitoring**: Logs allow real-time monitoring of application behavior, performance, and security.

3. **Auditing**: Logs help track user actions, aiding in compliance with regulations and security policies.

4. **Security**: Logs assist in post-incident analysis, determining the cause of failures or security breaches.

5. **Performance Analysis**: Logs reveal application performance trends and bottlenecks.

6. **Maintenance**: Logs aid in maintaining, upgrading, and optimizing applications over time.

## 38. Explain how Unix provides log management facility to network based application.

**UNIX Log Management for Network Applications:**

i. **Syslog Facility:**
- UNIX provides the syslog facility, a standardized way to generate, store, and manage log messages.

ii. **Syslog Daemon (syslogd)**:
- UNIX systems have a background daemon (syslogd) that collects and manages log messages.

iii. **Priority Levels:**
- UNIX syslog categorizes logs based on severity levels (e.g., DEBUG, INFO, WARNING, ERROR, CRITICAL).

iv. **Log Files:**
- Logs are stored in files, making it easy to access historical data for analysis.

v. **Log Rotation:**
- UNIX systems can rotate log files to prevent them from becoming too large and consuming excessive disk space.

vi. **Configuration:**
- Administrators can configure syslog to route logs to different destinations, including local files and remote servers.

vii. **Log Analysis Tools:**
- UNIX offers tools to search, filter, and analyze log data, aiding in troubleshooting and monitoring.

viii. **Security and Privacy:**
- Logs are secured to prevent unauthorized access and tampering.

***(Asked 2 times)***

## 39. What is syslog function and syslogd?

**1. Syslog Function ("syslog"):**

- "syslog" is a programming function or API provided by Unix-like operating systems.
- It allows applications to generate log messages and send them to a central logging system (i.e syslogd) for further processing.
- Applications use this function to provide information about their status, events, errors, and other relevant information.
- Applications can specify a priority level and a message to be logged using the "syslog" function.

**2. Syslog Daemon ("syslogd"):**

- "syslogd" is a system daemon in Unix-like OS responsible for receiving, storing, and managing log messages from various applications.
- It listens for incoming log messages from different sources, including applications, services, and devices.
- "syslogd" categorizes log messages based on their priority levels and routes them to appropriate log files or destinations.
- It can also forward log messages to remote log servers for centralized monitoring and analysis.
- Administrators can configure how syslogd handles different types of messages. They specify where to store log messages and how to handle different priority levels.
- Log messages are invaluable for monitoring system health and diagnosing issues.
- syslogd helps administrators quickly identify problems and potential security breaches.

## 40. Why do we need syslog? List and explain various syslog priority levels.

- We need syslog for:

i. Centralized Logging
ii. Troubleshooting
iii. Security Monitoring
iv. Compliance and Auditing
v. Historical Analysis
vi. Debugging

- The various syslog priority levels are: (higher to lower)

**1. LOG_EMERG (Emergency):**

- Indicates a system-wide panic condition.
- Example: "The system is going down for maintenance."

**2. LOG_ALERT (Alert):**

- Requires immediate attention.
- Indicates a condition that should be corrected promptly.
- Example: "Filesystem full, corrupted system database"

**3. LOG_CRIT (Critical):**

- Indicates a critical condition.
- A condition that should be addressed as soon as possible.
- Example: "Critical hardware failure."

**4. LOG_ERR (Error):**

- Indicates errors that are not critical.
- An error condition that can be handled.
- Example: "File not found."

**5. LOG_WARNING (Warning):**

- Indicates a warning.
- An issue that might cause problems if ignored.
- Example: "Low disk space."

**6. LOG_NOTICE (Notice):**

- Indicates normal but significant condition.
- For informational messages that might require attention.
- Example: "Configuration changes applied."

**7. LOG_INFO (Informational):**

- Provides general information about system operations.
- Helpful for tracking the system's general state.
- Example: "Service started successfully."

**8. LOG_DEBUG (Debug):**

- Provides debug-level messages.
- Primarily used during development and debugging.
- Example: "Entering function foo() with parameters."

**(Asked 2 times)**

## 41. Compare ioctl() and fcntl() function along with their code used.

**Ioctl() function**

- It stands for I/O control.
- It is used for controlling various device-specific operations that cannot be performed using standard file I/O operations.
- It is often used for configuring and controlling various device drivers, like setting terminal parameters or manipulating hardware devices.
- Supports a wider variety of operations but may lack standardization across different devices.

**Fcntl() function**

- Stands for "File Control."
- Used for controlling file-related behaviors and properties.
- Often used for file-level operations such as setting file access modes, locking, and changing file descriptors.
- More focused on file management and manipulation.
- Typically used for tasks like setting non-blocking mode, duplicating file descriptors, and obtaining or modifying file status flags.
- Offers a standardized set of operations.

## 42. Compare close() function and shutdown() function with outline code.

**1. close() function:**

- Used to close a socket, which means releasing its resources and terminating the connection.
- Closes both the read and write ends of the socket.
- Once closed, the socket cannot be used for further communication.
- Frees up system resources associated with the socket.
- Typically used when you no longer need the socket, such as after completing data exchange.

**2. shutdown() function:**

- Used to partially or fully shut down communication on a socket.
- You can specify whether to shut down the sending, receiving, or both ends of the socket.
- Allows you to control the direction of communication shutdown.
- The socket remains open, and you can still perform other operations on it.
- Useful for gracefully ending communication or for signaling the other party about the intent to close.

In simple terms, close() fully closes a socket, while shutdown() allows you to control how communication is stopped on the socket.

## 43. What is the technique for logging messages from a daemon process? Explain with sample code.

- Logging messages from a daemon process involves using the syslog library to send log messages to the system's logging infrastructure managed by syslogd.
- Here's how you can do it using the C programming language as an example:

**1. Include Necessary Headers:**

*#include <syslog.h>*

**2. Initializing Logging:**

// Open the connection to the syslog system

*openlog("my_daemon", LOG_PID, LOG_DAEMON);*

**3. Logging Messages:**

Use the syslog functions to log messages at different priority levels.

*// Log an informational message*

*syslog(LOG_INFO, "Daemon process started.");*

*// Log a warning message*

*syslog(LOG_WARNING, "Memory usage is high.");*

*// Log an error message*

*syslog(LOG_ERR, "Failed to open file: %s", filename);*

**4. Closing Logging:**

Close the connection to the syslog system when your daemon process is about to exit.

// Close the connection to the syslog system

*closelog();*

**Here's a full sample code snippet demonstrating the process:**

```
#include <stdio.h>
#include <stdlib.h>
#include <syslog.h>
#include <unistd.h>

int main() {
    // Open the connection to the syslog system
    openlog("my_daemon", LOG_PID, LOG_DAEMON);

    // Log some messages
    syslog(LOG_INFO, "Daemon process started.");
    syslog(LOG_WARNING, "Memory usage is high.");
    syslog(LOG_ERR, "Failed to open file: %s", "example.txt");

    // Simulate daemon activities
    sleep(5);

    // Close the connection to the syslog system
    closelog();

    return 0;
}
```

- In this example, the daemon process uses the syslog library to send log messages.
- The openlog function initializes the logging system, and then you can use syslog to log messages of different priority levels.
- Finally, the closelog function is used to close the connection to the syslog system.

# Chapter 3
# Winsock Programming
# (4Q) (30 marks)

**(Asked 7 times)**

## 1. Explain the Winsock Architecture with suitable diagram. (Windows Socket Architecture)

## OR

## Explain windows socket library along with suitable diagram.

- Winsock (Windows Sockets) is an API (Application Programming Interface) that allows Windows-based applications to communicate over a network using standard network protocols such as TCP/IP and UDP.
- Winsock provides a standardized way for applications to perform tasks like creating sockets, establishing connections, sending and receiving data, and managing network-related events .
- It supports various protocols, including IPv4 and IPv6, and offers both blocking and non-blocking I/O models for efficient network communication.
- The Winsock architecture consists of various components and layers that facilitate network communication.
- The various elements of Winsock architecture are:

**1. Winsock API (WS2_32.DLL):**

- At the core of the Winsock architecture is the Winsock API, which provides a set of functions(eg - socket(),WSAStartup(),getaddrinfo() etc) and data structures(struct sockaddr, struct addrinfo etc) for network programming.
- This API is exposed through the WS2_32.DLL dynamic-link library, and applications can access it by including the appropriate header files and linking against the library.

**2. Application Layer:**

- The application layer is where your networked application resides. Example: Online Games, Web Browsers, etc.
- Applications interact with the Winsock API through function calls to initiate network operations.
- It utilizes the Winsock API to create sockets, establish connections, send and receive data, and handle network-related events.

**3. Socket:**

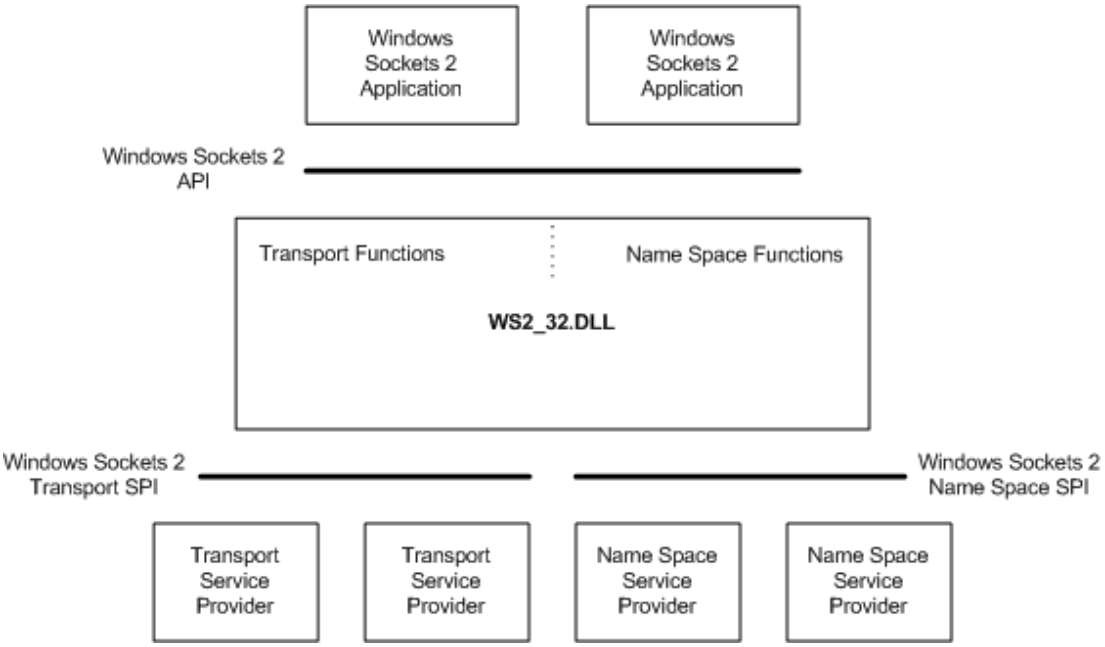
- Sockets are the endpoints of communication in the Winsock architecture.
- Applications create sockets using the socket() function, and each socket is associated with an address family, socket type and protocol.

**4. Winsock Service Provider Interface (SPI):**

- The Winsock architecture supports multiple network protocol stacks, such as TCP/IP, UDP, and others.
- The SPI is like a set of instructions that explain how the Winsock API functions should be used for a specific protocol stack.

**5. Transport Layer:**

- The transport layer is where the actual data transmission takes place.
- For example, when using TCP, the transport layer handles reliable and ordered data delivery.




**(Asked 4 times)**

## 2. Explain the types of DLL file in windows.

## OR

## Explain different DLLS used in Winsock program.

## OR

## Explain the helper DLLs and their interfaces.

- A DLL (Dynamic Link Library) file in Windows is a binary file that contains code and resources that multiple programs can use simultaneously.
- DLLs are designed to be reusable components that provide functions, classes, or resources.
- DLLs can contain executable code, data, and resources that can be called upon by various programs to perform specific tasks.
- When an application needs to use the functions or resources within a DLL, it can dynamically link to the DLL at runtime, allowing the application to access the features provided by the DLL without embedding the entire code within itself.
- The various types of DLLs are:

**1. Ws2_32.dll (Windows Sockets 2):**

- This is the core DLL for Winsock programming.
- It provides functions and services for creating and managing network sockets, sending and receiving data, and handling various network-related tasks.
- It is a central component for networking communication in Windows environments.

**2. Wship6.dll (IPv6 Helper DLL):**

- This DLL is responsible for handling IPv6-specific functionality.
- It handles tasks like translating human-readable names to numerical IP addresses (DNS resolution) and helping programs interact seamlessly with IPv6 networks.

**3. Wshtcpip.dll (TCP/IP Helper DLL):**

- This DLL provides various helper functions and utilities related to TCP/IP networking.
- It assists in tasks such as configuring network interfaces, managing network routes, and resolving host names to IP addresses.

**4. Ws2help.dll (Winsock Helper DLL):**

- This DLL provides additional helper functions and utilities for Winsock operations.
- It includes functions for advanced socket manipulation, handling socket events, and managing asynchronous operations.

**5. Mswsock.dll (Microsoft Windows Sockets Helper DLL):**

- This DLL is responsible for providing advanced networking features and services.
- It includes functions related to high-performance socket operations, advanced socket options, and handling socket-related events.

**(Asked 2 times)**

## 3. Point out the necessity of the Winsock DLL.

## OR

## What is the use of DLL in Winsock?

The necessity are listed below:

**1. Abstraction of Network Complexity:**

- Networking can be complex.
- The Winsock DLL handles these tricky parts behind the scenes, so programmers don't need to worry about them.

**2. Standardization:**

- Different network protocols (such as TCP/IP, UDP/IP) have varying implementations.
- The Winsock DLL provides a standardized interface that applications can use regardless of the underlying protocol or service.

**3. Ease of Development:**

- Network programming involves dealing with various aspects like socket creation, data transmission, and error handling.
- The Winsock DLL encapsulates these tasks in user-friendly functions, reducing the complexity of writing networking code.

**4. Simplified Access:**

- Without the Winsock DLL, developers would need to interact directly with the operating system's networking stack, which could be complex and vary across different versions of the OS.
- The Winsock DLL provides a standardized layer for accessing networking features.

**5. Efficient Management:**

- The Winsock DLL manages the networking resources and communication for applications, optimizing data transmission and ensuring that applications don't interfere with each other's network activities.

**6. Support for Third-Party Libraries:**

- Many network-related libraries and frameworks are built to work with the Winsock API, making it a common interface for integrating various networking functionalities into applications.

**7. Socket Management:**

- It simplifies the creation, management, and interaction with sockets, enabling seamless data exchange between applications.

**8. Cross-Compatibility:**

- Applications using Winsock DLL can be developed on Windows systems and run on different versions of Windows without major modifications.

**(Asked 3 times)**

## 4. Compare static and dynamic link library in the case of windows.

**Static Link Library**: Code is inside your program; it's self-contained and easy to move.

**Dynamic Link Library (DLL):** Code is separate; it's smaller and can be shared among programs.

**Static Link Library:**

i. **Code Inclusion**: All required code is bundled directly into your program during compilation.

ii. **Size**: The compiled program can become larger as it contains all the library code it needs.

iii. **Portability**: The program can be easily moved to other systems without worrying about missing library files. Sharing the program means sharing all its code.

iv. **Dependencies**: There are no external dependencies; the program is self-contained.

v. **Versioning**: Any updates to the library require recompiling the entire program.

vi. **Isolation**: Since each program has its own copy of the library, changes in one program don't affect other programs using the same library.

vii. **Larger Load Time**: Static libraries are linked into the program during compilation, which can increase the program's load time.

viii. **Example**: Used when you want to create a standalone executable with all dependencies included. For instance, creating a small utility tool. **(.lib files)**

**Dynamic Link Library (DLL):**

i. **Code Separation**: DLLs are separate files containing reusable code that can be shared among multiple applications. Applications only reference the DLL, rather than including the entire code.

ii. **Size**: The program size remains smaller as it doesn't include library code.

iii. **Portability**: DLLs must be distributed with the program, ensuring they're available on other systems.

iv. **Dependencies**: The program depends on external DLLs; if a DLL is missing, the program won't work.

v. **Versioning**: Updates can be made to DLLs without recompiling the program, allowing for easier maintenance.

vi. **Sharing:** Multiple programs can use the same DLL, saving memory and keeping updates consistent.

vii. **Smaller load time**: Dynamic libraries are loaded only when needed, potentially reducing load times.

viii. **Example**: Suitable for situations where multiple programs can benefit from shared code. For example, system-wide DLLs for handling graphics, networking, or database interactions across various applications. **(.dll files)**

**(Asked 5 times)**

## 5. Differences between Unix socket and windows socket.

## OR

## What are the major differences between Berkely socket API and Winsock API.

- The difference are:

**1. Platform:**

- **Unix Sockets**: Developed for Unix-like operating systems such as Linux, macOS, and various Unix distributions.
- **Windows Sockets (Winsock)**: Specifically designed for Windows operating systems, including various versions like Windows 7, Windows 10, etc.

**2. API Names:**

- **Unix Sockets**: Utilizes the traditional Berkeley Socket API.
- **Windows Sockets (Winsock):** Deploys the Windows-specific API known as Winsock.

**3. API Function Names:**

- **Unix Sockets**: Functions like socket(), bind(), listen(), accept(), and connect() adhere to Unix naming conventions.
- **Windows Sockets (Winsock):** Similar functions like socket(), bind(), listen(), accept(), and connect() are available as part of the Winsock API.

**4. Code Portability:**

- **Unix Sockets**: Code written using the Berkeley Socket API might require modifications to function on Windows.
- **Windows Sockets (Winsock):** Code developed with the Winsock API might need adjustments to work on Unix-like systems.

**5. Addressing and Data Types:**

- **Unix Sockets:** Use Unix-specific data types, like sockaddr_un, for Unix domain sockets.
- **Windows Sockets (Winsock):** Employ Windows-specific data types, such as SOCKADDR_IN, for Internet domain sockets.

**6. Socket Initialization and Cleanup:**

- **Unix Sockets:** It does not require explicit initialization or cleanup for the socket library.
- **Windows Sockets (Winsock):** It requires explicit initialization using the WSAStartup() function and cleanup using the WSACleanup() function.

**7. Error Handling:**

- **Unix Sockets**: Frequently use standard Unix error codes and error-handling mechanisms like errno.
- **Windows Sockets (Winsock):** It relies on the WSAGetLastError() function to retrieve the error code associated with the last operation that failed.

**8. Socket Creation:**

- **Unix Sockets**: It also uses the socket() function to create sockets, and the socket descriptor is represented using an integer file descriptor (usually int).
- **Windows Sockets (Winsock):** It uses the socket() function to create sockets, and the socket descriptor is represented using the SOCKET data type.

***(Berkeley/ BSD/Unix)***

**(Asked 3 times)**

## 6. In which approach of communication, windows socket provides better functionalities than Unix socket.

## OR

## What are the advancement that is done by WINSOCK over BSD socket.

- Winsock (Windows Sockets) was developed as an extension of the BSD socket API to provide networking capabilities on Windows operating systems.
- It introduced several advancements and improvements over the original BSD socket API.
- Here are some key advancements provided by Winsock:

**1. Integration with Windows Environment:**

- Winsock is designed to seamlessly integrate with the Windows operating system, making it a natural choice for networking on Windows platforms.

**2. Windows-Friendly Function Names:**

- Winsock functions have names that align with Windows naming conventions, making it easier for developers familiar with Windows development to work with networking.

**3. Security and Authentication:**

- Integration with Windows Authentication
- Leveraging Windows security mechanisms for enhanced network security.

**4. Asynchronous I/O Support:**

- Winsock introduced robust support for asynchronous I/O operations, enabling efficient handling of multiple socket connections without blocking the main application thread.
- WSAAsyncSelect() function is used for it.

**5. Event-Driven Programming:**

- Winsock provides event-driven programming capabilities, allowing developers to create applications that respond to various network events efficiently.
- WSAEventSelect() function is used for it.

**6. Windows-Specific Error Handling:**

- Winsock offers error codes that are specific to Windows, enabling developers to handle network-related errors in a way that is consistent with Windows programming practices.
- WSAGetLastError() function is used to get latest error.

**7. Compatibility with Windows GUI Frameworks:**

- Winsock integrates seamlessly with Windows GUI frameworks like WinForms and WPF, enabling developers to build network-aware graphical applications.

**8. Scalability and Performance Enhancements:**

- Winsock includes optimizations that leverage Windows' underlying network stack, enhancing the scalability and performance of network applications.

**9. Support for Additional Protocols:**

- While BSD sockets primarily focus on TCP and UDP, Winsock provides support for other Windows-specific protocols and extensions, such as Windows Raw Sockets for low-level packet manipulation.

***(Asked 3 times)***

## 7. How do you implement stream communication in Winsock? Describe each step with the help of relevant APIs.

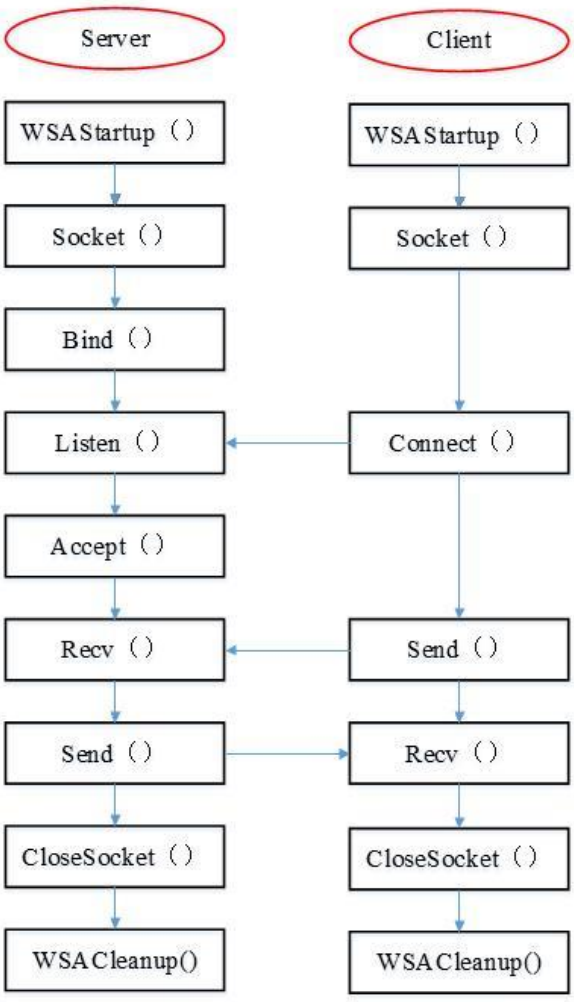



Fig: Client – Server Communication in Winsock

- Implementing stream communication using Winsock involves setting up a server and client to establish a TCP connection.

**Server Side:**

**1. Initialize Winsock:**

- Initialize the Winsock library using the WSAStartup() function.

**2. Create a Socket:**

- Create a socket using the socket() function.

**3. Bind the Socket:**

- Bind the socket to a specific IP address and port using the bind() function.

**4. Listen for Incoming Connections:**

- Put the socket in a listening state using the listen() function.

**5. Accept Incoming Connections:**

- Accept incoming connections using the accept() function, which creates a new socket for the established connection.

**6. Receive and Send Data:**

- Use the recv() function to receive data from the client and the send() function to send data to the client.

**7. Close Sockets:**

- Close the connected socket and the listening socket using the closesocket() function.

**8. Clean Up Winsock:**

- Call the WSACleanup() function to release resources used by the Winsock library.

**Client Side:**

**1. Initialize Winsock:**

- Initialize the Winsock library using the WSAStartup() function.

**2. Create a Socket:**

- Create a socket using the socket() function.

**3. Connect to Server:**

- Connect to the server's IP address and port using the connect() function.

**4. Send and Receive Data:**

- Use the send() function to send data to the server and the recv() function to receive data from the server.

**5. Close Socket:**

- Close the socket using the closesocket() function.

**6. Clean Up Winsock:**

- Call the WSACleanup() function to release resources used by the Winsock library.

***(Asked 2 times)***

## 8. Provide a code snippet for sending and receiving data over connection using Winsock Programming.

## OR

## Write simple TCP windows server and client to send and receive message.

```c
#include <Winsock2.h>

int main() {
    WSADATA wsaData;
    SOCKET serverSocket, clientSocket;

    // Initialize Winsock
    WSAStartup(MAKEWORD(2, 2), &wsaData);

    // Create socket
    serverSocket = socket(AF_INET, SOCK_STREAM, 0);

    // Bind socket
    // ...

    // Listen
    listen(serverSocket, SOMAXCONN);

    // Accept incoming connection
    clientSocket = accept(serverSocket, NULL, NULL);

    // Receive and send data
    // ...

    // Close sockets
    closesocket(clientSocket);
    closesocket(serverSocket);

    // Clean up Winsock
    WSACleanup();

    return 0;
}
```

Server Side

```c
#include <Winsock2.h>

int main() {
    WSADATA wsaData;
    SOCKET clientSocket;

    // Initialize Winsock
    WSAStartup(MAKEWORD(2, 2), &wsaData);

    // Create socket
    clientSocket = socket(AF_INET, SOCK_STREAM, 0);

    // Connect to server
    // ...

    // Send and receive data
    // ...

    // Close socket
    closesocket(clientSocket);

    // Clean up Winsock
    WSACleanup();

    return 0;
}
```

Client Side

(Asked 1 time, only code was asked)

## Q. Outline the simple UDP windows client program which can send and receive data without establishing the connection with the server.

- We can implement UDP communication using Winsock with the relevant APIs:

**Server Side:**

**1. Initialize Winsock:**

- Initialize the Winsock library using the WSAStartup() function.

**2. Create a Socket:**

- Create a socket using the socket() function

**3. Bind the Socket:**

- Bind the socket to a specific IP address and port using the bind() function.

**4. Receive Data:**

- Use the recvfrom() function to receive data from clients. This function also provides the source address and port.

**5. Send Data:**

- Use the sendto() function to send data to clients. Specify the destination address and port.

**6. Close Socket:**

- Close the socket using the closesocket() function.

**7. Clean Up Winsock:**

- Call the WSACleanup() function to release resources used by the Winsock library.

**Client Side:**

**1. Initialize Winsock:**

- Initialize the Winsock library using the WSAStartup() function.

**2. Create a Socket:**

- Create a socket using the socket() function.

**3. Send Data:**

- Use the sendto() function to send data to the server. Specify the destination address and port.

**4. Receive Data:**

- Use the recvfrom() function to receive data from the server. This function also provides the source address and port.

**5. Close Socket:**

- Close the socket using the closesocket() function.

**6. Clean Up Winsock:**

- Call the WSACleanup() function to release resources used by the Winsock library.

```
//server

#include <Winsock2.h>
#include <iostream>

int main() {
    WSADATA wsaData;
    SOCKET serverSocket;
    sockaddr_in serverAddr, clientAddr;
    int clientAddrSize = sizeof(clientAddr);

    // Initialize Winsock
    WSAStartup(MAKEWORD(2, 2), &wsaData);

    // Create socket
    serverSocket = socket(AF_INET, SOCK_DGRAM, 0);

    // Server address setup


    // Bind socket
    bind(serverSocket, (sockaddr*)&serverAddr, sizeof(serverAddr));

    // Receive and send data


    // Close socket
    closesocket(serverSocket);

    // Clean up Winsock
    WSACleanup();

    return 0;
}
```

```
//client

#include <Winsock2.h>
#include <iostream>

int main() {
    WSADATA wsaData;
    SOCKET clientSocket;
    sockaddr_in serverAddr;

    // Initialize Winsock
    WSAStartup(MAKEWORD(2, 2), &wsaData);

    // Create socket
    clientSocket = socket(AF_INET, SOCK_DGRAM, 0);

    // Server address setup


    // Send and receive data


    // Close socket
    closesocket(clientSocket);

    // Clean up Winsock
    WSACleanup();

    return 0;
}
```

**(Asked 3 times)**

## 9. Explain WSAStartup() and WSACleanup() function in window with suitable outline code.

## OR

## Explain the significance of setup and cleanup functions in windows socket with function prototype and required structure definition.

***Setup and cleanup are informal terms used to indicate the process of calling WSAStartup() and WSACleanup() functions respectively.***

- WSAStartup() is used to initialize the Winsock library, preparing it for networking operations by allocating resources and checking compatibility.
- After you're done using Winsock, WSACleanup() should be called to release resources and perform necessary cleanup tasks.

**1. WSAStartup()**

- WSAStartup is the first function you should call before using any other Winsock functions.
- It initializes the Winsock library and prepares it for networking operations.
- When you call WSAStartup(), Winsock performs various internal setup tasks, like allocating memory and preparing data structures for networking operations.
- It checks whether the requested version of Winsock is available and compatible.

**Syntax**:

*int WSAStartup(WORD wVersionRequested, LPWSADATA lpWSAData);*

**Parameters:**

i. **wVersionRequested:**
   - This parameter specifies the version of Winsock that our application wants to use.

- Typically, we pass MAKEWORD(2, 2) for Winsock version 2.2.

ii. **lpWSAData:** A pointer to a WSADATA structure that will hold information about the initialized Winsock library.

**Return Value**:

- The function returns zero if successful. If an error occurs, it returns a value other than zero.
- We can use the **WSAGetLastError** function to get the error code.

**2. WSACleanup()**

- WSACleanup() is the counterpart to WSAStartup().
- It's used to clean up and release the resources associated with the Winsock library when you're done using it.
- When you call WSACleanup, Winsock releases resources allocated during initialization, like memory and data structures.
- It's essential to call this function after you've finished using Winsock to avoid memory leaks and ensure proper shutdown.

**Syntax:**

*int WSACleanup();*

**Return Value:**

The function returns zero if successful. If an error occurs, it returns a value other than zero.

## Q. Write short notes on WSAStartup

## 10. Explain WSAData, WSACleanup, WSAstartup, and closesocket with help of suitable code.

**1. WSAData()**

- The WSAData structure is used to hold information about the Winsock library after it has been initialized using WSAStartup.
- It provides details about the version of Winsock, the high and low version numbers supported, and various other attributes.

*WSADATA wsaData;*

**2. WSAStartup Function:**

- WSAStartup initializes the Winsock library, and it's the first function you call before using other Winsock functions.
- It takes the requested version of Winsock and a pointer to a WSAData structure that will be filled with library information.

*WSAStartup(MAKEWORD(2, 2), &wsaData);*

**3. closesocket Function:**

- closesocket is used to close a socket that was previously created using the socket function.

*closesocket(clientSocket);*

**4. WSACleanup Function:**

- WSACleanup is called when you're done using the Winsock library to clean up resources.
- It should be called after all networking operations are complete.

*WSACleanup();*

Note: Better give code example of client from Que no 8

**(Asked 2 times)**

## 11. Why WSAGetLastError() function is required in Winsock programming.

## OR
## Explain WSAGetLastError and WSASetLastError.

**1. WSAGetLastError**

- WSAGetLastError retrieves the error code of the last Winsock function that encountered an error.
- It doesn't require any parameters.

*int WSAGetLastError(void);*

**2. WSASetLastError:**

- WSASetLastError lets us manually set and error code, which can be used with WSAGetLastError later.
- It is not typically used for regular error handling since the error code is automatically set by failed Winsock functions.
- It may be helpful in certain situations.

*void WSASetLastError(int iError);*

- **We need WSAGetLastError for following reasons:**

**1. Error Indication:**

- When a Winsock function returns a value indicating an error (usually a value other than 0), WSAGetLastError() is used to retrieve an error code that identifies the specific error that occurred.

**2. Diagnostic Information:**

- Winsock errors can be caused by various factors, such as network issues, incorrect usage of functions, or resource limitations.
- WSAGetLastError() helps you understand the nature of the error and what went wrong.

**3. Differentiation of Errors:**

- Winsock can encounter a wide range of errors, from simple connection issues to complex protocol-specific errors.
- WSAGetLastError() provides a distinct error code for each type of error, allowing you to tailor your error-handling logic.

**4. Custom Error Messages:**

- By using WSAGetLastError() in combination with error code information, you can provide customized error messages to users or log specific details for debugging purposes.
- This makes error handling more user-friendly and informative.

**5. Guidance for Troubleshooting:**

- With the error code obtained from WSAGetLastError(), you can refer to official documentation or resources provided by Microsoft to understand the error's cause and potential solutions.
- This helps in troubleshooting and resolving issues efficiently.

## 12. Compare how error handling facility is implemented in Berkeley socket API and Windows socket API.

- Error handling facilities in both the Berkeley Sockets API (BSD Sockets) and the Windows Sockets API (Winsock) serve similar purposes: to provide information about errors that occur during network operations.
- However, there are differences in how these APIs implement error handling.

**1. Berkeley Sockets API (BSD Sockets):**

**a. errno Global Variable:**

- BSD Sockets use the global variable errno to store the error code of the last function call that encountered an error.
- Developers can access the error code using errno after an error occurs. This code is specific to the last function called.

**b. Consistency Across Calls:**

- Since errno is a global variable, it retains the error code until another function call modifies it.
- This allows you to retrieve and analyze error codes from different parts of your code without worrying about them being reset automatically.

**c. Per-Thread Error Handling:**

- errno is thread-specific, meaning each thread has its own copy of the errno variable. This prevents interference between threads' error codes.

**2. Windows Sockets API (Winsock):**

**a. WSAGetLastError() Function:**

- In Winsock, error codes are obtained using the WSAGetLastError() function.
- Each Winsock function call that fails sets the error code, and you use WSAGetLastError() immediately to retrieve that error code.

**b. Thread-Specific Error Handling:**

- Similar to errno, WSAGetLastError() is also thread-specific. Each thread keeps track of its own error code.
- This is important in multithreaded applications, where threads may be performing different network operations concurrently.

**c. Immediate Error Retrieval:**

- Unlike errno, which can hold the error code across multiple function calls, WSAGetLastError() should be used immediately after the failed function call to capture the specific error code.

## 13. Explain different I/O handling modes in Windows Socket API. Which functions from Winsock API are used to provide each of these I/O handling modes? What parameters do they expect.

- In the Windows Sockets (Winsock) API, there are several I/O handling modes that allow you to manage how input and output operations are performed.
- These modes provide flexibility in handling data transmission and reception.
- Here are the main I/O handling modes along with the relevant functions from the Winsock API:

**1. Blocking Mode:**

- In blocking mode, socket operations (such as send and recv) block the calling thread until the operation is completed or an error occurs.
- Default mode for sockets.
- Functions: send, recv, accept, connect, etc.

**2. Non-blocking Mode:**

- In non-blocking mode, socket operations return immediately, even if the operation cannot be completed immediately.
- This mode is suitable for applications where responsiveness is crucial.
- Functions: send, recv, accept, connect, etc., with additional functions like WSAGetLastError, select, and asynchronous functions like WSAAsyncSelect.

**3. Overlapped Mode (Asynchronous I/O with overlapped operations):**

- In overlapped mode, socket operations are performed asynchronously, and the calling thread doesn't block while the operation is in progress.
- This mode is ideal for handling a large number of concurrent operations without blocking the main thread.
- Suitable for high-performance I/O operations.

Relevant Functions:

- WSASocket or socket to create a socket.
- WSAEventSelect to associate sockets with Windows events for asynchronous notification.
- WSAAsyncSelect to associate sockets with asynchronous message notifications.
- WSARecv, WSASend for receiving data and sending data asynchronously.

***(Asked 3 times)***

## 14. Explain different Winsock functions that supports synchronous and asynchronous I/O.

**Synchronous I/O Functions:**

- These functions perform I/O operations synchronously, meaning they block the calling thread until the operation is completed or an error occurs.

i. **recv**: Receive data from a socket.
ii. **send**: Send data to a socket.
iii. **accept**: Accept a new incoming connection.
iv. **connect**: Initiate a connection to a remote endpoint.
v. **listen**: Set a socket in a listening state to accept incoming connections.
vi. **gethostbyname**: Resolve a host name to an IP address (older method).

**Asynchronous I/O Functions:**

- These functions perform I/O operations asynchronously, meaning they allow the calling thread to continue executing while the operation is in progress.
- Completion notifications are typically delivered using callbacks or events.

i. WSAConnect: Asynchronously initiate a connection to a remote endpoint.
ii. WSAAccept: Asynchronously accept an incoming connection.
iii. WSARecv: Asynchronously receive data from a socket.
iv. WSASend: Asynchronously send data to a socket.

v. WSAEventSelect: Associate a socket with a Windows event object for asynchronous notification.
vi. WSAAsyncSelect: Associate a socket with asynchronous message notifications.

## Q. Explain WSAEventSeelct and WSAAsyncSelect function.

**1. WSAEventSelect Function**:

- **Event Association**: Links a socket to specific network events.
- **Event Objects**: Uses event objects for event notifications.
- **Non-Blocking**: Works non-blocking, reducing CPU usage.
- **Customized Monitoring**: Choose which events to monitor per socket.
- **Scalable:** Efficiently handles multiple sockets and events.

**2. WSAAsyncSelect Function:**

- **Event Subscription**: Subscribes to socket events with event codes.
- **Window Procedure**: Relies on the application's window procedure.
- **Message-Based**: Generates messages for event occurrences.
- **GUI Integration**: Well-suited for GUI-based applications.
- **Event Flexibility**: Supports a variety of event codes for selection.

***(Asked 3 times)***

## 15. What are overlapped I/O. Explain its advantages in windows programming.

- Overlapped I/O is a technique in computer programming where multiple input/output (I/O) operations can happen at the same time, without waiting for each one to finish before starting the next.
- Overlapped I/O is a specific implementation of asynchronous I/O.

**Advantages of Overlapped I/O:**

i. **Concurrency**:
- Overlapped I/O allows your program to perform multiple tasks at the same time.
- For example, you can read data from one network connection while writing data to another without waiting for each operation to complete.

ii. **Responsiveness**:
- Your program can stay responsive even when waiting for I/O operations to finish.
- This is important for applications that need to handle user interactions while performing I/O.

iii. **Efficiency**:
- Overlapped I/O lets your program use your computer's resources more effectively. Instead of waiting around, it can keep doing useful work.

iv. **Faster Execution**:
- By overlapping I/O operations, your program can reduce the total time needed to complete tasks, making it more efficient and faster.

v. **Scalability**:
- Overlapped I/O is great for handling many connections or tasks simultaneously, making it suitable for network servers and applications dealing with multiple clients.

vi. **Complex Tasks:**
- It simplifies handling complex tasks involving multiple I/O operations, like managing data transfers and user interactions simultaneously.

vii. **Asynchronous Behavior:**

- Overlapped I/O helps you achieve asynchronous behavior, where your program can initiate actions and continue without waiting for the actions to finish.

## Q. WSAGetOverlappedSocket

- WSAGetOverlappedResult is a function provided by the Windows Sockets API (Winsock) that is used to retrieve the results of an overlapped operation on a socket.
- Overlapped operations allow asynchronous communication, meaning that a function call doesn't block the program's execution, and the result is obtained later.

## Q. Compare overlapped socket system call with blocking socket system call along with the help of outline code.

## 16. Explain how WSASocket, WSAAccpet, WSAConnect, WSASend and WSARecv can be used to implement asynchronous I/O.

- All the above mentioned functions are part of the Windows Sockets (Winsock) API and are used for asynchronous and overlapped I/O operations in network programming.
- These functions allow you to perform non-blocking and concurrent I/O operations.

**1. WSASocket:**

- Create a socket for communication that supports overlapped I/O operations.
- Set the socket to non-blocking mode so it doesn't block the program while waiting for data.

**2. WSAAccept:**

- Listen for incoming connections on a socket.
- Associate the socket with an event object, so your program can do other tasks while waiting for a connection.

**3. WSAConnect:**

- Establish a connection to a remote socket asynchronously.
- Use asynchronous methods like event-driven mechanisms to keep your program responsive during connection.

**4. WSASend:**

- WSASend is used for sending data asynchronously over a socket.
- It allows you to initiate a send operation and continue executing other tasks without waiting for the send to complete.

**5. WSARecv:**

- WSARecv is used for receiving data asynchronously from a socket.
- Similar to WSASend, it lets you initiate a receive operation and continue other tasks while waiting for the data to arrive.

***(Asked 2 times)***

## 17. In which situation is asynchronous I/O is preferred than synchronous I/O.

- Asynchronous I/O is preferred over synchronous I/O in situations where:

i. **Responsiveness**:
- Asynchronous I/O allows the program to continue performing other tasks while waiting for I/O operations to complete.
- This is crucial for applications where responsiveness and multitasking are essential.

ii. **Scalability**:
- Asynchronous I/O enables handling multiple I/O operations concurrently without blocking the entire program.
- This is beneficial for applications dealing with many clients or heavy I/O workloads.

**iii. Efficiency:**
- Asynchronous I/O can optimize resource usage by avoiding unnecessary waits.
- It helps prevent blocking, which may occur in synchronous I/O when waiting for data or responses from external sources.

**iv. Latency-sensitive Applications:**
- Applications requiring low latency, like real-time communication or multimedia streaming, benefit from asynchronous I/O.
- It reduces the delay introduced by waiting for I/O operations to complete.

**v. Non-blocking Behavior:**
- Asynchronous I/O allows the program to keep executing even when I/O operations are not immediately ready.
- This is useful for handling dynamic or unpredictable data availability.

vi. **Handling Many Clients:**

- Servers that need to handle multiple clients simultaneously benefit from asynchronous I/O, as they can efficiently manage communication with numerous clients without creating a new thread or process for each.

vii. **Resource Utilization:**

- Asynchronous I/O reduces resource wastage by enabling more efficient usage of CPU and memory, as the program can perform tasks during I/O waits.

## 18. What is the difference between synchronous I/O and Asynchronous I/O?

**1. Blocking vs. Non-blocking:**

- Synchronous I/O blocks the program until the I/O operation is complete.
- Asynchronous I/O allows the program to continue executing without waiting for the I/O operation to complete.

**2. Program Flow:**

- Synchronous I/O has a straightforward, linear program flow.
- Asynchronous I/O has a more complex, potentially non-linear program flow due to callbacks and notifications.

**3. Concurrency**:

- Synchronous I/O doesn't handle concurrency well and can lead to inefficient use of resources.
- Asynchronous I/O enables efficient concurrency, allowing the program to perform other tasks while waiting for I/O.

**4. Complexity:**

- Synchronous I/O is simpler to implement and understand.
- Asynchronous I/O is more complex due to the need for managing callbacks, events, and synchronization.

**5. Efficiency and CPU Utilization:**

- Synchronous I/O may waste CPU time during I/O wait, leading to inefficiency in handling concurrent tasks.
- Asynchronous I/O utilizes CPU more efficiently, enabling concurrent handling of multiple I/O operations.

**6. Multitasking and Scalability:**

- Synchronous I/O is not suitable for multitasking environments and may have limited scalability for concurrent tasks.
- Asynchronous I/O is suitable for multitasking, highly scalable for handling numerous concurrent tasks.

**6. Use Cases:**

- Synchronous I/O is suited for simpler applications where performance and concurrency are not primary concerns.
- Asynchronous I/O is well-suited for applications requiring high responsiveness, concurrency, and efficient resource utilization.

***(Asked 3 times)***

## 19. Differentiate between recv() and WSAarecv() based on their uses, input/output arguments and return values.

## OR

## Differentiate between recv() and WSArecv() . Explain with their syntax and parameter.

- recv() is used for basic data receiving, while WSARecv() is for more advanced asynchronous operations and overlapped I/O, making it more suitable for applications that require non-blocking behavior, responsiveness, and efficient multitasking.

**1. recv():**

- recv() is used in both Unix sockets (BSD Sockets) and Windows Sockets (Winsock) for receiving data from sockets.

**Return Value:**

- Returns the number of bytes received.
- Returns 0 if the remote side closed the connection.
- Returns SOCKET_ERROR on error.

**2. WSARecv():**

- WSARecv() is specific to Windows Sockets (Winsock) and is used for overlapped and asynchronous I/O.

**Return Value:**

- Returns 0 on success.
- Returns SOCKET_ERROR on error.

**Use Case:**

- recv() is for basic data receiving from sockets.
- WSARecv() is used for advanced asynchronous and overlapped I/O.

**Advantages of WSARecv():**

- Supports overlapped I/O: Allows concurrent operations.
- Enables non-blocking behavior: Keeps program responsive.
- More suitable for complex, asynchronous tasks.
- Can improve efficiency and scalability in network applications.

## 20. Describe the use of connect function with non-blocking socket.

- In Windows, a non-blocking socket is one that doesn't halt the program when it waits for an event (like connecting to a server).
- With a non-blocking socket, the connect function initiates a connection but doesn't wait for it to complete immediately.
- Instead, the function returns immediately, allowing your program to continue doing other tasks while the connection attempt is ongoing.
- You can use other functions like select or asynchronous I/O methods to check the status of the connection in the background and take appropriate action.

***(Asked 5 times)***

## 21. Explain select() function in conjunction with accept() call in winsock.

## OR

## Explain with the help of pseudo-code the use of accept with select such that the accept function doesn't block.

## OR

## Explain select function with suitable piece of outline code.

## OR

## Explain with help of code how select can be used conjunction with accept call.

**1. Select Function:**

- The select function is a powerful mechanism in the Winsock library that allows you to monitor multiple sockets for events like data availability, connection requests, or errors.
- It helps you manage non-blocking sockets efficiently by allowing your program to wait for events on multiple sockets without blocking the entire execution.

**2. Accept Function:**

- The accept function is used to accept incoming connection requests on a listening socket.
- It blocks the program until a connection request arrives and a new socket is created to handle the actual communication with the client.

**3. Using Select with Accept:**

- In scenarios where you want to handle multiple sockets, including incoming connection requests, without blocking, you can combine the select function and the accept function.

- Here's how it works:

i. You create a set of sockets that you want to monitor using FD_SET.
ii. You call select to wait for events on the sockets in the set.
iii. When select indicates that one of the sockets has an event (such as a new connection request), you can use FD_ISSET to check which socket is ready.
iv. If the socket that's ready happens to be your listening socket (the one waiting for new connections), that means there's a new connection request waiting to be accepted.
v. This is like someone knocking on your door. You use the accept function to "open the door" and create a new socket for the incoming connection.

- In a nutshell, you're using select to keep an eye on a group of sockets, and when select says, "Hey, something's up," you use FD_ISSET to see which socket it is.
- If it's your listening socket, you know there's a new connection to be accepted, and you use accept to make it happen.
- All of this is done without making your program wait idly.

# Chapter 4
# Network Utilities and Application
# (2SQ) (10 marks)

## 1. Write short notes on:

**(Asked 8 times)**

### a. Remote login

- Remote login is the capability to access and control a computer or network device from a different location over a network.
- Simply, remote login facility permits a user who is using one computer to interact with a program on another computer.
- Remote login allows users to perform tasks, run programs, access files, and manage the remote system as if they had direct physical access to it.
- It is typically accomplished using protocols, such as
    i. **Secure Shell (SSH)**
    ii. **TELNET.**
- Some of the popular applications used for remote login are Team Viewer, Any Desk, Microsoft Remote Desktop, Chrome Remote Desktop, etc.
- It is widely used in various contexts, including IT support, server management, work-from-home setups, cloud infrastructure management, and more.
- It offers convenience, flexibility, and efficiency by enabling users to access and control remote resources without the need for physical presence at the remote location.

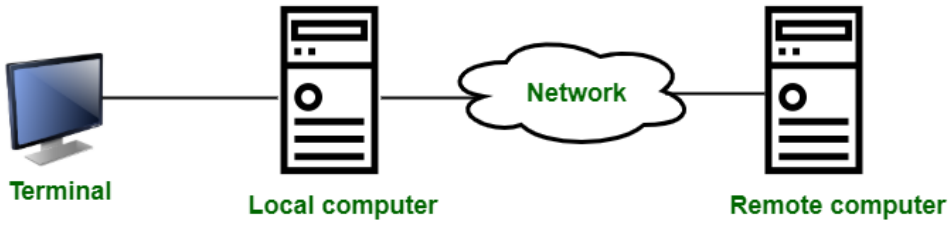



Fig: Remote Login

**(Asked 4 times)**

## b. Telnet

- TELNET (terminal network) is a TCP/IP standard for establishing a connection to a remote system.
- Telnet is a network protocol that allows you to remotely log in to and interact with a computer or network device over a network, such as the internet or a local network.
- Simply, it is used for remote login and command-line interaction with the remote system.
- Telnet is also the name of the application that implements the Telnet protocol on client and server sides.
- Telnet uses plain text communication. When you type commands or text, it's sent to the remote computer as plain, readable text, making it easy to understand but not secure.
- Telnet is not secure because it lacks encryption. That means any data, including usernames and passwords, is sent in plain text, making it vulnerable to interception by hackers.
- Due to its lack of security, using Telnet over the internet is generally not recommended.
- However, it may still be used on local networks where security is less of a concern and when dealing with legacy systems that do not support SSH.
- With TELNET, an application program on the user's machine becomes the client.

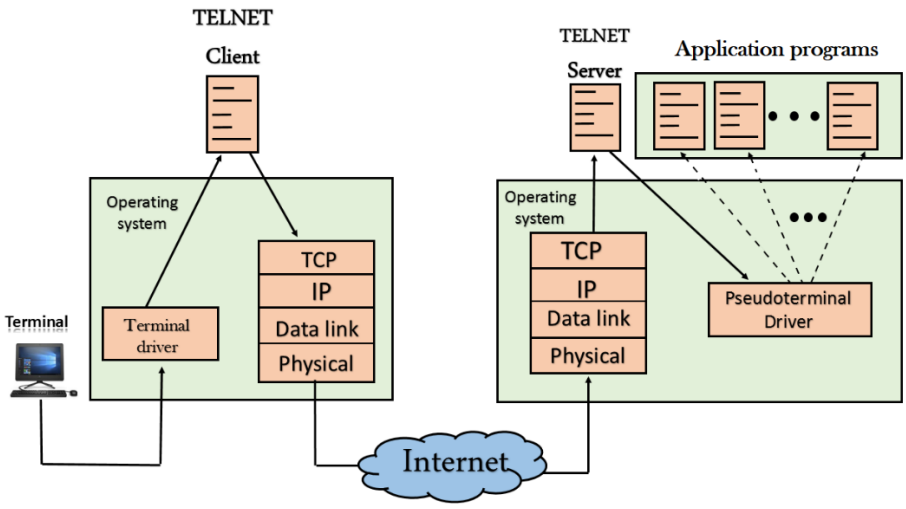



Fig: TELNET

**(Asked 5 times)**

## c. Netstat

- Netstat stands for "Network Statistics."
- It is a command-line tool used on computers to display information about network connections.
- Netstat helps you see all the active network connections your computer has with other devices on the internet or local network.
- It shows you which programs or applications are using these connections, the specific IP addresses they are connected to, and the ports they are using for communication.
- It indicates the status of each connection, such as "established" for connections that are currently active, or "listening" for connections that are waiting to receive data.
- It is a handy tool for troubleshooting network issues. It helps identify if certain programs are connecting to the internet or find any unusual network activity on your computer.
- To use netstat, you typically open a command prompt or terminal, type "netstat," and press enter. It then shows you a list of active connections and relevant details.
- While netstat is helpful for local network troubleshooting, avoid using it on public networks or unfamiliar systems, as it might display sensitive information about your computer's connections.

Some of the commands are:

i. netstat: The basic command without any options displays a list of all active network connections.
ii. netstat -r: displays the routing table i.e. destination, gateway, etc.
iii. netstat -p: shows the process id of the program associated with each network connection.
iv. netstat -t: Shows only the active TCP connections.
v. netstat -u: Shows only the active UDP connections.
vi. netstat -s: Provides statistics of various network protocols, such as TCP, UDP, ICMP, etc.

Q. Compare telnet and netstat.

**(Asked 5 times)**

## d. ipconfig/ifconfig

### a. **ipconfig** – command used in windows

- "ipconfig" stands for "IP configuration."
- It's a command used in the Windows command prompt to show information about your computer's network connections.
- When you run "ipconfig," you can see your computer's IP address, subnet mask, default gateway, and other network-related information.
- It helps you troubleshoot network issues, check if your computer is connected to the network correctly and identify any problems with the network configuration.

**Commands**

i. ipconfig: Displays basic IP configuration information for all active network interfaces.
ii. ipconfig /all: Shows detailed IP configuration for all network interfaces.
iii. ipconfig /renew: Renews the IP address for a network interface, requesting a new address from the DHCP server.

### b. **Ifconfig** – command used in linux

- It's a command used in Unix, Linux, and macOS terminals to display information about your computer's network interfaces (like Wi-Fi or Ethernet connections) and their configurations.
- When you run "ifconfig," you can see details about each network interface, such as its IP address, subnet mask, broadcast address, and status.
- Like "ipconfig," "ifconfig" helps you troubleshoot network problems, set up network interfaces, and check their settings.

**Commands**

i. ifconfig: Displays basic information for all active network interfaces.
ii. ifconfig -a: Shows details of all network interfaces, including inactive ones

Summarizing, "ipconfig" is used on Windows systems to show network information, while "ifconfig" is used on Unix-based systems (Linux and macOS) for the same purpose. Both commands are valuable tools for understanding and managing your computer's network connections.

**(Asked 5 times)**

## e. Ping

## OR

## Ping (icmp request and reply)

- "Ping" is a command-line utility used on computers and devices to check the availability and response time of other computers or devices over a network, such as the internet or a local network.
- When you run the "ping" command, your computer sends a small data packet, called an ICMP (Internet Control Message Protocol) echo request, to the target computer or device.
- The target computer receives the ICMP echo request and immediately sends back an ICMP echo reply if it is reachable and functioning correctly.
- The time it takes for the ICMP echo request to travel to the target and for the ICMP echo reply to return is measured, and this is known as the "ping time" or "round-trip time" (RTT). It is usually displayed in milliseconds(ms).
- By default, "ping" sends four ICMP echo requests (pings) to the target and displays the response time for each.
- If the target responds to all four pings, it means the connection is working, and the target is reachable. The response time gives an indication of how quickly data travels between your computer and the target.
- If the target does not respond to any of the pings, it might be unreachable, or there could be network connectivity issues between two devices.
- "Ping" is a valuable tool for network troubleshooting. If a website, server, or device doesn't respond to "ping," it may indicate network problems, server issues, or that the target is deliberately configured not to respond to ICMP requests.
- Some network administrators may disable ICMP echo responses (ping responses) for security reasons, as it could potentially be exploited for certain types of attacks.

**Example Command**

*ping www.example.com*

- This command is used to ping the server with the domain name www.example.com

***(Asked 6 times)***

## f. TFTP

- Trivial File Transfer Protocol (TFTP) is a basic file transfer protocol used to move files between computers on a network.
- Data transfer through TFTP is usually initiated through port 69.
- It is implemented on the top of the UDP protocol.
- It is simple to use and has minimal features, making it lightweight and easy to implement.
- TFTP only reads/writes files from/to a remote server. It cannot list, delete, or rename files or directories.
- TFTP is commonly used for tasks like booting devices over the network, transferring configuration files, or updating firmware on network devices.
- Unlike other file transfer protocols, TFTP does not provide security features like authentication or encryption, which makes it suitable for internal network environments but less secure for the internet.
- Due to its simplicity and small code footprint, TFTP is often used in embedded systems or scenarios where a lightweight file transfer solution is needed.

- The steps involved in TFTP are:
  i. **Client Request:** The client requests to read or write a file from/to the server.
  ii. **Server Response:** The server responds to the client's request.
  iii. **Data Transfer:** Data packets are exchanged between the client and server for file read/write.
  iv. **Acknowledgment:** Both sides acknowledge the successful receipt of data packets.
  v. **Finalization:** After the transfer is complete, the session is terminated.

## g. Network Diagnostic Tools

- A Network Diagnostic Tool is a software or utility designed to identify, troubleshoot, and resolve network-related issues.
- These tools help network administrators and users diagnose problems within computer networks, both local and remote, and ensure that the network is running smoothly and efficiently.
- They provide insights into network performance, connectivity, and configuration, making it easier to detect and fix network problems.

- Some of the common network diagnostic tools are:

1. ping
2. netstat
3. ipconfig
4. ifconfig

(Explain them)