**Government College of Engineering, Jalgaon**

**Department of Computer Engineering**

**(CO309-Computer Network Technology Lab)**

|  |
| --- |
| Name: Pallavi Govardhan Patil. PRN: 2241046. Class: TY.    Practicale No: 1. Session: V Subject Teacher: Ms.prajakta Swale.  Date of Performance: 02/09/2024. Date Of completation:09/09/2024. |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Write Up** | **Correctness of Program** | **Documentation of Program** | **Viva** | **Attendance for Practical** | **Timely**  **Completion** | **Total** | **Dated sign of Subject Teacher** |
| **4** | **2** | **2** | **5** | **2** | **5** | **20** |  |
|  |  |  |  |  |  |  |

**Practical No.1**

* **AIM :**-

1. UNIX Sockets: WAP program in C/C++ /Python/Java sockets API.

a. TCP sockets.

b. UDP sockets.

* **SOFTWARE REQUIRED :-**

Operating System: - Ubuntu JAVA .

* **THEROY:-**
  + - **What is Unix Socket**

A **Unix socket** (or **Unix domain socket**, **UDS**) is an inter-process communication (IPC) mechanism that allows data exchange between processes running on the same host (computer). Unlike network sockets, which communicate over a network (e.g., TCP/IP), Unix sockets operate only within the local machine, providing a faster and more efficient means of communication between processes.

Here are some key characteristics of Unix sockets:

1. **Local-only Communication**: They are limited to processes on the same system, making them different from network sockets, which can communicate across different systems.
2. **Types**:
   * **Stream Sockets** (SOCK\_STREAM): Provide reliable, connection-based communication similar to TCP. E.g **Like a Telephone Line**: Once a connection is established, it acts like a continuous phone call. Data flows back and forth smoothly and reliably
   * **Datagram Sockets** (SOCK\_DGRAM): Provide connectionless, message-based communication similar to UDP. Eg.**Like Sending Letters**: You send individual messages (like letters) without making a permanent connection. Each message is sent independently and might arrive out of order or not at all.
3. **File System Representation**: Unix sockets are represented as files in the file system. These files allow processes to connect to the socket by referencing the file's path.
4. **File System Representation**: Unix sockets are represented as files in the file system. These files allow processes to connect to the socket by referencing the file's path.
5. **Performance**: Since they bypass the network stack, Unix sockets tend to have lower latency and higher throughput compared to network sockets.
6. **Common Use Cases**:
   * Communication between system services (like a database server and client).
   * Communication within containerized environments (e.g., Docker containers on the same host).
   * APIs like **gRPC**, which can use Unix sockets for communication between microservices on the same machine

* **TCP Sockets (Stream Sockets)**

Definition: TCP is a connection-oriented protocol that ensures reliable and ordered data transfer between two endpoints. When using TCP sockets, a persistent connection is established, and data is transmitted after a successful connection handshake.

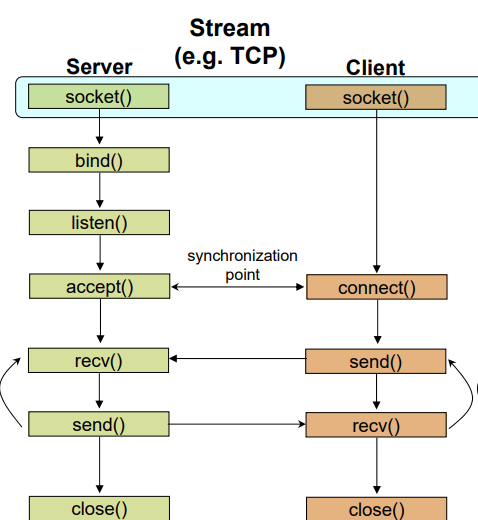
* **TCP communication involves a three-way handshake**:

1. SYN: The client sends a synchronization (SYN) message to the server.
2. SYN-ACK: The server responds with an acknowledgment (SYN-ACK).
3. ACK: The client sends a final acknowledgment (ACK), establishing the connection.

* **Key Features of TCP**:

1. Reliable Transmission: TCP ensures that data is delivered to the destination. If packets are lost or corrupted, TCP automatically retransmits them.
2. Connection-Oriented: A connection must be established between the client and server before data transmission can occur.
3. Ordered Data Delivery: TCP ensures that packets are received in the order they were sent, making it ideal for applications requiring sequential data.
4. Error Checking: TCP provides mechanisms to check for errors in transmitted data using checksums.

* Diagram :- (Client - Server Communication – Unix)



* **TCP Client:**

A **TCP client** is a program or system that initiates a connection to a **TCP server** to send and receive data using the **Transmission Control Protocol (TCP)**. TCP provides reliable, ordered, and error-checked delivery of data between applications running on different hosts on a network.

Here’s a step-by-step breakdown of the typical operations of a TCP client:

1. **socket()**:
   * **Purpose**: The client first creates a **socket**. A socket is an endpoint for sending or receiving data over a network.
   * **How it works**: The socket() function is called, which specifies the address family (e.g., IPv4/IPv6), the type of communication (TCP for reliable streaming), and the protocol to be used.
   * **Example**: socket(AF\_INET, SOCK\_STREAM, 0) creates a TCP socket using the IPv4 address family.
2. **connect()**:
   * **Purpose**: The client establishes a connection to the **server** by specifying the server's IP address and port number.
   * **How it works**: The connect() function initiates a connection to the server, where the server must be listening for connections on the specified port.
   * **Example**: connect(socket\_fd, (struct sockaddr\*)&server\_address, sizeof(server\_address)) establishes the connection.
3. **send()**:
   * **Purpose**: After the connection is established, the client can start **sending data** to the server.
   * **How it works**: The send() function is used to transmit data to the server. TCP ensures that the data is delivered in the correct order and without errors.
   * **Example**: send(socket\_fd, "message", strlen("message"), 0) sends the data over the established connection.
4. **recv()**:
   * **Purpose**: The client waits to **receive a response** from the server.
   * **How it works**: The recv() function is used to read incoming data from the server. TCP will ensure the data is received in the correct order.
   * **Example**: recv(socket\_fd, buffer, sizeof(buffer), 0) receives data from the server into a buffer.
5. **close()**:
   * **Purpose**: Once the client has finished communicating with the server, it **closes** the connection.
   * **How it works**: The close() function is called to close the socket and terminate the connection.
   * **Example**: close(socket\_fd) terminates the connection and frees up the socket.

* **TCP Server:**

A **TCP server** is a program or system that listens for incoming connection requests from **TCP clients** and engages in reliable, two-way communication using the **Transmission Control Protocol (TCP)**. Unlike a TCP client, the server waits for clients to connect and manages multiple client connections as needed.

Here’s a step-by-step breakdown of the operations of a TCP server:**Steps for a TCP Server:**

1. **socket()**:
   * **Purpose**: The server creates a **socket** to establish a communication endpoint.
   * **How it works**: The socket() function is used to create a socket for network communication. Just like the client, the server specifies the address family (IPv4/IPv6), type (TCP for reliable streaming), and protocol.
   * **Example**: socket(AF\_INET, SOCK\_STREAM, 0) creates a TCP socket.
2. **bind()**:
   * **Purpose**: The server assigns (binds) the socket to a specific **IP address** and **port number**.
   * **How it works**: The bind() function associates the socket with a local IP address and a specific port, so the server knows where to listen for incoming connections.
   * **Example**: bind(socket\_fd, (struct sockaddr\*)&server\_address, sizeof(server\_address)) binds the socket to a port (e.g., port 8080).
3. **listen()**:
   * **Purpose**: The server **listens** for incoming connection requests from clients.
   * **How it works**: The listen() function tells the server to start listening for client requests. It also specifies the maximum number of queued connection requests.
   * **Example**: listen(socket\_fd, 5) allows the server to accept up to 5 queued connection requests.
4. **accept()**:
   * **Purpose**: The server **accepts** an incoming connection from a client.
   * **How it works**: The accept() function blocks until a client attempts to connect. Once a connection is made, the server creates a new socket dedicated to this specific client while continuing to listen for more connections on the original socket.
   * **Example**: client\_socket\_fd = accept(socket\_fd, (struct sockaddr\*)&client\_address, &client\_len) accepts a client connection and returns a new socket for communicating with that client.
5. **recv()** and **send()**:
   * **Purpose**: Once the connection is established, the server and client can exchange data using recv() to receive and send() to transmit.
   * **How it works**: The server can receive data from the client using recv() and send data back using send(). These functions ensure that data is reliably transmitted in the correct order.
   * **Example**:
     + recv(client\_socket\_fd, buffer, sizeof(buffer), 0) to receive data from the client.
     + send(client\_socket\_fd, "message", strlen("message"), 0) to send data to the client.
6. **close()**:
   * **Purpose**: Once the server is finished communicating with the client, it closes the dedicated client socket, freeing up resources.
   * **How it works**: The close() function closes the specific client connection, and the server is ready to accept new connections.
   * **Example**: close(client\_socket\_fd) closes the connection with a particular client.

* TCP Advantages

1. **Reliability**: Ensures accurate and complete data delivery with error detection and retransmissions.
2. **Order Preservation**: Maintains the sequence of data packets.
3. **Flow Control**: Prevents overwhelming the receiver.

### **TCP Disadvantages**

1. **Overhead**: Adds extra data and processing due to its control features.
2. **Latency**: Introduces delays due to connection setup and error recovery.
3. **Resource Intensive**: Consumes more system resources for connection management.
4. **Not Ideal for Real-Time**: Can cause delays in real-time applications like video streaming.
5. **Congestion Control**: Adjusts data transmission based on network congestion.

* **UDP Sockets (Datagram Sockets)**

**Definition:**

UDP is a connectionless protocol that sends data in discrete packets, without establishing a connection or ensuring delivery. This makes UDP faster but less reliable than TCP.

### **Basic Principles of UDP Communication**

1. **Connectionless Nature:**

UDP is connectionless, meaning each datagram is sent independently without establishing a connection between sender and receiver. There’s no handshake or session setup.

1. **Datagram-Based:**

Data is sent in discrete packets called datagrams. Each datagram is self-contained and includes both header and payload.

1. **Unreliable Delivery:**

UDP does not guarantee delivery, order, or error-free communication. Packets may be lost, duplicated, or received out of order. The protocol does not perform any retransmissions.

1. **Low Overhead:**

Due to minimal protocol overhead (no connection management, flow control, or error recovery), UDP is faster and requires fewer resources compared to TCP.

1. **Application Layer Responsibilities:**

The application layer is responsible for handling tasks such as error correction, packet sequencing, and data recovery if needed.

* **Key Features of UDP**:

1. **No Connection Overhead**: UDP does not require a connection, making it lightweight and faster than TCP.
2. **Unreliable Transmission**: There are no guarantees that data will arrive, nor that it will be received in the correct order.
3. **Broadcast and Multicast Support**: UDP allows sending data to multiple recipients simultaneously.

* Diagram :- (Client - Server Communication – Unix)

