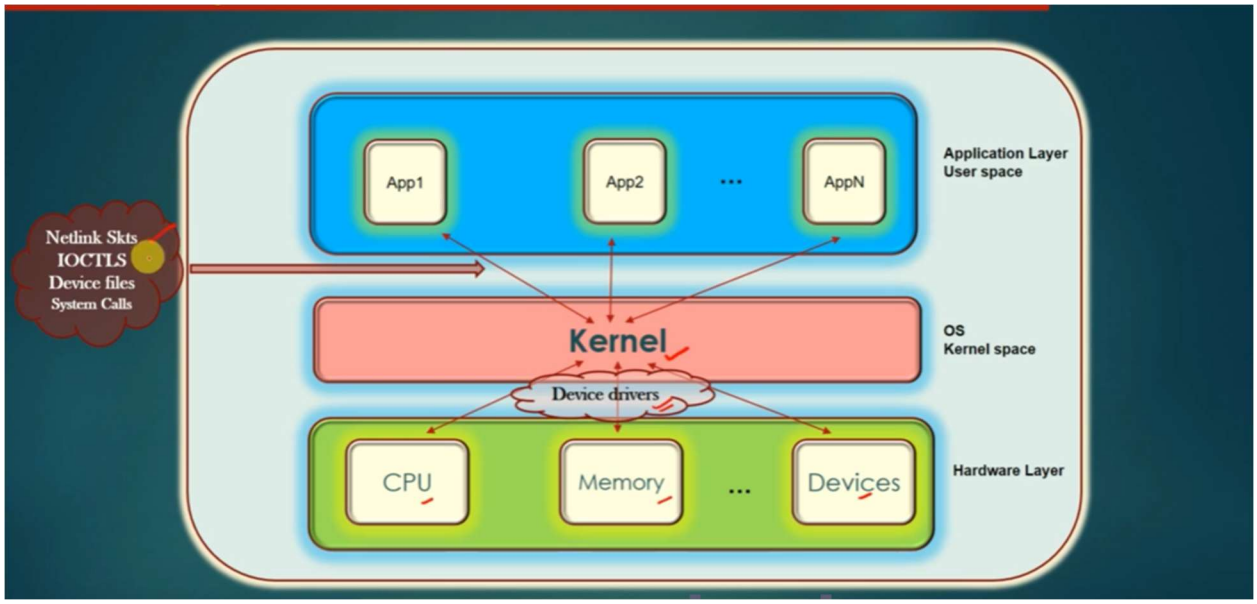
What is IPC?

IPC is a Mechanism using which two or more Process running on the same machine exchange their personal Data each other.

- Communication between processes running on different machines are not termed as IPC
- Process running on same machine often Need to exchange data with each other in order to implement some functionality.
- Linux OS provides several mechanisms using which user space processes can carry out communication with Other, each mechanism has its own pros and cons.

Computer System Architecture:



There are several ways to carry out IPC.

IPC Technique:

1. Using Unix Domain socket
2. Using Network Sockets(Not Covered in this course)
3. Message Queues
4. Shared Memory
5. Pipes(Not used in the industry, Not covered)
6. Signals

IPC Technique -Socket :

- Unix/Linux like OS provide Socket interface to carry out communication between various type of entities.
- The socket interface are a bunch of socket Programming related APIs
- We shall be using these APIs to implement the socket of various types
- In this course we shall learn how to implement Two System:
- **Unix Domain Socket**: IPC between process running the same System.
- **Network Socket** : Communication between process running on different machines over the network.

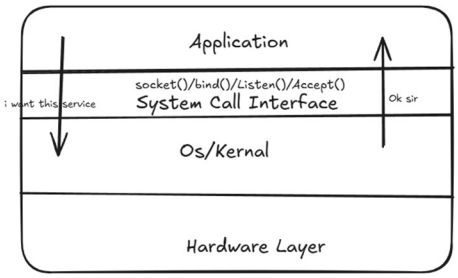
Socket System Call / Socket Interface

Socket Programming Step and related APIS

Steps:

1. Remove the Socket. If already exit.
2. Crete a Unix socket using socket()
3. Specify the socket name
4. Bind the socket bind()
5. Listen()
6. Accept()
7. Read the data received on socket using recvfrom()
8. Send back the result using sendto()
9. Close the data socket
10. Close the connection socket
11. Remove the socket
12. Exit

Computer Layer Architecture:



Socket Message Types:

Message exchanged between the client and the server process can be categorized into two types:

- This msg used by the client process to request the server process to establish a dedicated connection
- Only after the connection has been established.then only client can send Service request message to server.

Socket:

- A socket between two program.
- It helps two program send and receive data.
- Used in client-server communication.

Types:

Socket is Basically Two types :

1. Stream Socket
2. Datagram Socket

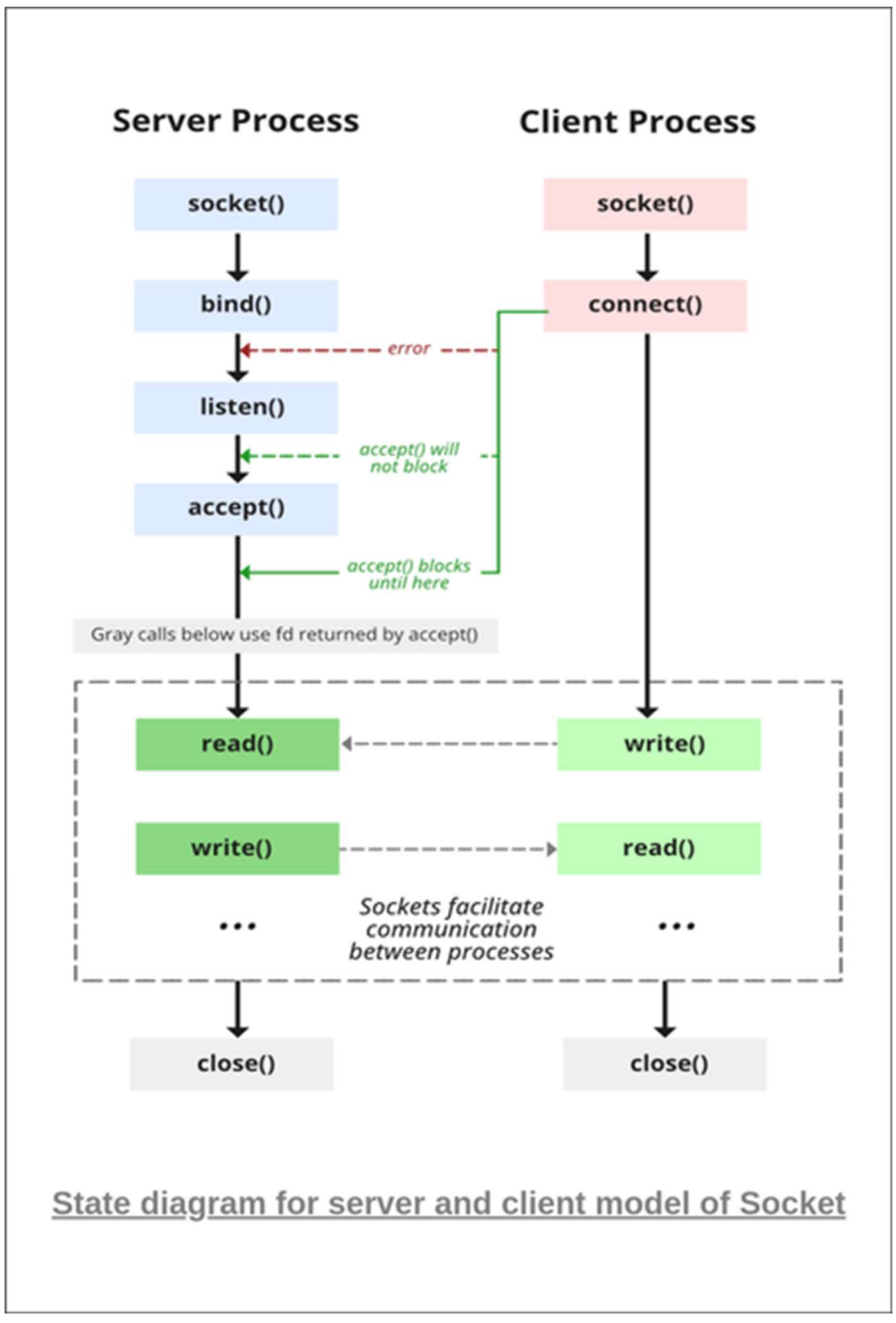
Stream Socket:

A stream socket is a type of network socket that provides a connection-oriented, reliable, and bidirectional data flow between two communicating endpoints. It uses the Transmission Control Protocol (TCP) to ensure ordered, error-free, and connection-based communication.

Datagram Socket:

A Datagram Socket is a type of network socket that utilizes the User Datagram Protocol (UDP) for communication.

Socket Programming How Work



State diagram for server and client model of Socket

What is a System Call?

- A **system call** is a way for a program to **ask the operating system** to do something.
- Example: reading a file, creating a socket, or sending data over the network.
- It's like your program saying:
  👉 “Hey OS, please do this for me!”

📝 Socket Functions in Simple Words

1. **socket()**
   ➤ Creates a communication endpoint.
   ➤ Like buying a phone to talk to someone.
2. **bind()**
   ➤ Gives your socket an address (IP + port).
   ➤ Like assigning a phone number to your phone.
3. **listen()**
   ➤ Waits for incoming connections.
   ➤ Like turning on your phone and waiting for calls.
4. **accept()**
   ➤ Accepts a connection from a client.
   ➤ Like picking up the phone when someone calls.
5. **read() / recv()**
   ➤ Receives data from the client.
   ➤ Like listening to what the caller says.
6. **send() / write()**
   ➤ Sends data to the client.
   ➤ Like talking back to the caller.
7. **close()**
   ➤ Closes the connection.
   ➤ Like hanging up the phone.

Unix Socket :

- A Unix socket is socket that works inside the same computer.
- It does not Ip address or port No.
- It uses a file path like /tmp/mysocket.
- Faster than TCP/IP because it avoid network steps.

Header file:

```c
#include <stdio.h> //printing
#include <string.h> //string strcpy()
#include <stdlib.h> //exit()
#include <sys/socket.h> //socket all function
#include <sys/un.h> //unix socket
#include <unistd.h> //unlink() close()
```

Why use:

1. **<stdio.h>** – For input/output functions like printf() and scanf().
2. **<string.h>** – For string functions like strcpy(), strlen(), etc.
3. **<stdlib.h>** – For general functions like exit(), malloc(), etc.
4. **<sys/socket.h>** – For creating and using sockets (network communication).
5. **<sys/un.h>** – For UNIX domain sockets (local communication between programs).
6. **<unistd.h>** – For system calls like close(), unlink(), read(), etc.

server.c

```c
// server.c
#include <stdio.h> //printing
#include <string.h> //string strcpy()
#include <stdlib.h> //exit()
#include <sys/socket.h> //socket all function
#include <sys/un.h> //unix socket
#include <unistd.h> //unlink() close()


#define SOCKET_PATH "unix_socket_file"  // The path for the Unix socket

int main() {
    int serverSocket, connectedClientSocket;
    struct sockaddr_un serverAddress;

    char messageBuffer[100];

    // 1. Create server socket
    serverSocket = socket(AF_UNIX, SOCK_STREAM, 0);
    if (serverSocket == -1) {
        perror("Server socket creation failed");
        exit(EXIT_FAILURE);
    }

    // 2. Clear and set address info
    memset(&serverAddress, 0, sizeof(struct sockaddr_un));
    serverAddress.sun_family = AF_UNIX;
    strcpy(serverAddress.sun_path, SOCKET_PATH);

    // 3. Remove any existing socket file
    unlink(SOCKET_PATH);

    // 4. Bind socket to address
    if (bind(serverSocket, (struct sockaddr *)&serverAddress, sizeof(struct sockaddr_un)) == -1) {
        perror("Binding failed");
        close(serverSocket);
        exit(EXIT_FAILURE);
    }

    // 5. Listen for client connections
    if (listen(serverSocket, 5) == -1) {
        perror("Listening failed");
        close(serverSocket);
        exit(EXIT_FAILURE);
    }
    printf("🕐 Server is waiting for a client connection...\n");

    // 6. Accept a client connection
    connectedClientSocket = accept(serverSocket, NULL, NULL);
    if (connectedClientSocket == -1) {
        perror("Accept failed");
        close(serverSocket);
        exit(EXIT_FAILURE);
    }
    printf("✅ Client connected!\n");

    // 7. Receive message from client
    memset(messageBuffer, 0, sizeof(messageBuffer));
    read(connectedClientSocket, messageBuffer, sizeof(messageBuffer));
    printf("📩 Message from client: %s\n", messageBuffer);

    // 8. Send response to client
    const char *serverReply = "👋 Hello from server!";
    write(connectedClientSocket, serverReply, strlen(serverReply) + 1);

    // 9. Close sockets
    close(connectedClientSocket);
    close(serverSocket);

    // 10. Remove socket file
    unlink(SOCKET_PATH);

    return 0;
}
```

client.c

```c
// client.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <unistd.h>

#define SOCKET_PATH "unix_socket_file"  // Must match server

int main() {
    int clientSocket;
    struct sockaddr_un serverAddress;
    const char *clientMessage = "👋 Hello from client!";
    char replyBuffer[100];

    // 1. Create client socket
    clientSocket = socket(AF_UNIX, SOCK_STREAM, 0);
    if (clientSocket == -1) {
        perror("Client socket creation failed");
        exit(EXIT_FAILURE);
    }

    // 2. Clear and set server address
    memset(&serverAddress, 0, sizeof(struct sockaddr_un));
    serverAddress.sun_family = AF_UNIX;
    strcpy(serverAddress.sun_path, SOCKET_PATH);

    // 3. Connect to server
    if (connect(clientSocket, (struct sockaddr *)&serverAddress, sizeof(struct sockaddr_un)) == -1) {
        perror("Connection to server failed");
        close(clientSocket);
        exit(EXIT_FAILURE);
    }

    // 4. Send message to server
    write(clientSocket, clientMessage, strlen(clientMessage) + 1);

    // 5. Receive server response
    memset(replyBuffer, 0, sizeof(replyBuffer));
    read(clientSocket, replyBuffer, sizeof(replyBuffer));
    printf("📩 Message from server: %s\n", replyBuffer);

    // 6. Close client socket
    close(clientSocket);

    return 0;
}
```

Message Queue:

A Message Queue is a type of inter-process-communication(IPC) mechanism that allow tow or more process

- communication with each other .using sender and receiver.
- It is asynchronous communication : means sender receiver does not active same time.
- Work like queue (FIFO -technique)
- each message have a message type -like (String)
- its also used Unix system. means one system.

Creation(Syntax):

*message queue creation using a function mq_open()

1. mqd_t mq_open(const char* name,int oflag);
2. mqd_t mq_open(const char* name,int oflag,mode_t mode,struct mq_attr *attr);

- name-> this is the queue name .
- oflag->How open queue like(Read Write)
- mode ->Who access the queue Ex:0660
- attr -> configaration means queue size

Enqueue():

- Adding item to the queue.
- Send message using -> mq_send()

Dequeue():

- Removing item to the Queue.
- Reciving message using -> mq.recive()

Example of code :

```c
#include <stdio.h>
#include <fcntl.h>      // For O_CREAT, O_RDWR
#include <sys/stat.h>   // For mode_t
#include <mqueue.h>     // For mq_open, mq_attr

int main() {
    mqd_t mq;                    // Message queue descriptor
    struct mq_attr attr;         // Queue attributes

    // Set queue attributes
    attr.mq_flags = 0;           // Blocking mode
    attr.mq_maxmsg = 10;         // Max 10 messages
    attr.mq_msgsize = 256;       // Max size per message
    attr.mq_curmsgs = 0;         // Current messages (read-only)

    mq = mq_open("/myqueue", O_CREAT | O_RDWR, 0660, &attr);
    if (mq == -1) {
        perror("mq_open failed");
        return 1;
    } else {
        printf("Message Queue created successfully\n");
    }

    mq_close(mq);
    return 0;
}
```

What is a TCP Socket?

A TCP socket is a programming interface that allows two computers to communicate over a network using the Transmission Control Protocol (TCP).

Why Use TCP?

TCP is used because it provides:

- **Reliable communication**: Ensures data is delivered without errors and in the correct order.
- **Connection-oriented**: Establishes a connection before data transfer.
- **Error checking**: Detects and corrects errors during transmission.
- **Flow control**: Manages data flow to prevent overwhelming the receiver.

Server.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>

#define PORT 8080

int main() {
    int server_socket, client_socket;
    struct sockaddr_in server_addr, client_addr;
    socklen_t addr_size;
    char messageBuffer[1024] = {0};

    // Create socket
    server_socket = socket(AF_INET, SOCK_STREAM, 0);
    if (server_socket == -1) {
        printf("Socket creation failed\n");
        return 1;
    }

    // Server address setup
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = INADDR_ANY;
    server_addr.sin_port = htons(PORT);

    // Bind
    if (bind(server_socket, (struct sockaddr*)&server_addr, sizeof(server_addr)) == -1) {
        printf("Bind failed\n");
        return 1;
    }

    // Listen
    if (listen(server_socket, 5) == -1) {
        printf("Listen failed\n");
        return 1;
    }

    printf("Server listening on port %d...\n", PORT);
```

```c
    // Accept
    addr_size = sizeof(client_addr);
    client_socket = accept(server_socket, (struct sockaddr*)&client_addr, &addr_size);
    if (client_socket == -1) {
        printf("Accept failed\n");
        return 1;
    }

    // Read message from client
    if (read(client_socket, messageBuffer, sizeof(messageBuffer)) == -1) {
        printf("Read failed\n");
        return 1;
    }
    printf("Client: %s\n", messageBuffer);

    // Send response
    strcpy(messageBuffer, "Hello from server");
    if (send(client_socket, messageBuffer, strlen(messageBuffer), 0) == -1) {
        printf("Send failed\n");
        return 1;
    }
    printf("Message sent to client.\n");

    // Close sockets
    close(client_socket);
    close(server_socket);

    return 0;
}
```

Explain:

Header File:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
```

✅ stdio.h → প্রিন্ট ও ইনপুট নেয়ার জন্য

✅ stdlib.h → প্রোগ্রাম বন্ধ ও utility কাজের জন্য

✅ string.h → স্ট্রিং কপি/তুলনার জন্য

✅ unistd.h → read/write/close করার জন্য

✅ arpa/inet.h → IP ও পোর্ট ম্যানেজ করার জন্য

```
int server_socket, client_socket;
struct sockaddr_in server_addr, client_addr;
socklen_t addr_size;
char messageBuffer[1024] = {0};
```

➤ **server_socket**

সার্ভারের মূল সোকেট, যেটা কানেকশন শোনে (listen করে)।

➤ **client_socket**

ক্লায়েন্ট যখন সংযুক্ত হয়, তখন তার সঙ্গে কথা বলার জন্য এই সোকেট তৈরি হয়।

➤ **server_addr**

সার্ভারের IP অ্যাড্রেস ও পোর্ট নম্বর ধরে রাখে।

➤ **client_addr**

ক্লায়েন্টের IP অ্যাড্রেস ও পোর্ট নম্বর ধরে রাখে।

➤ **addr_size**

ক্লায়েন্ট অ্যাড্রেসের সাইজ সংরক্ষণে লাগে, accept()-এ ব্যবহৃত হয়।

➤ **messageBuffer[1024]**

মেসেজ রাখার জন্য একটি বাফার, যাতে ক্লায়েন্টের পাঠানো বা সার্ভারের পাঠানো বার্তা থাকে।

```
// Create socket
server_socket = socket(AF_INET, SOCK_STREAM, 0);
if (server_socket == -1) {
    printf("Socket creation failed\n");
    return 1;
}
```

AF_INET

➤ এর মানে হলো — আমরা IPv4 ব্যবহার করছি।

👉 যেমন: 127.0.0.1 বা 192.168.0.1

- AF = Address Family
- INET = Internet (IPv4)

SOCK_STREAM

➤ আমরা TCP প্রোটোকল ব্যবহার করবো — এটা নির্ভরযোগ্য (reliable) ডেটা ট্রান্সমিশন করে।

- TCP মানে — ডেটা হারায় না, সঠিকভাবে গন্তব্যে পৌঁছায়।
- SOCK_STREAM = ধারাবাহিক data stream

0

➤ ডিফল্ট প্রোটোকল ব্যবহার করবো।

TCP ব্যবহারের জন্য IPPROTO_TCP লাগতো, কিন্তু এখানে 0 দিলেই সঠিক প্রোটোকল সিলেক্ট হয়।

```
// Server address setup
server_addr.sin_family = AF_INET;
server_addr.sin_addr.s_addr = INADDR_ANY;
server_addr.sin_port = htons(PORT);
```

🧩 server_addr.sin_family = AF_INET;

➤ এর মানে:

আমরা IPv4 address system ব্যবহার করছি।

- sin_family ফিল্ডে AF_INET মান দিয়ে বলি — এটা IPv4 টাইপের socket

🧩 server_addr.sin_addr.s_addr = INADDR_ANY;

➤ এর মানে:

সার্ভার যেকোনো IP address থেকে কানেকশন গ্রহণ করতে প্রস্তুত।

- INADDR_ANY মানে — সার্ভার সব নেটওয়ার্ক ইন্টারফেস থেকে কানেকশন নিতে পারবে

(যেমন: localhost, wifi, ethernet যাই হোক)।

✅ এটা দিলে তুমি পরে অন্য কোনো IP address দিতে না হলেও চলবে।

🧩 server_addr.sin_port = htons(PORT);

➤ এর মানে:

আমরা যে পোর্টে সার্ভার চালাবো (যেমন: 8080), সেটা সঠিক ফরম্যাটে (network order) রূপান্তর করে দিচ্ছি।

- htons() = host to network short
- TCP/IP communication-এর জন্য পোর্ট নম্বরকে network format এ রূপান্তর করতে হয়।

Client.c :

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>

#define PORT 8080

int main() {
    int client_socket;
    struct sockaddr_in server_addr;
    char messageBuffer[1024] = {0};

    // Create socket
    client_socket = socket(AF_INET, SOCK_STREAM, 0);
    if (client_socket == -1) {
        printf("Socket creation failed\n");
        return 1;
    }

    // Server address setup
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(PORT);
    server_addr.sin_addr.s_addr = inet_addr("127.0.0.1");

    // Connect
    if (connect(client_socket, (struct sockaddr*)&server_addr, sizeof(server_addr)) == -1) {
        printf("Connection failed\n");
        return 1;
    }

    // Send message
    strcpy(messageBuffer, "Hello from client");
    if (send(client_socket, messageBuffer, strlen(messageBuffer), 0) == -1) {
        printf("Send failed\n");
        return 1;
    }
    printf("Message sent to server.\n");

    // Read server response
    memset(messageBuffer, 0, sizeof(messageBuffer));
    if (read(client_socket, messageBuffer, sizeof(messageBuffer)) == -1) {
        printf("Read failed\n");
        return 1;
    }
    printf("Server: %s\n", messageBuffer);

    // Close socket
    close(client_socket);

    return 0;
}
```

UDP socket:

UDP socket routines enable simple IP communication using the user datagram protocol (UDP)

Key Characteristics of UDP Sockets:

- Connectionless
- Datagram-based
- Unreliable
- Fast
- Checksums

How UDP Sockets Work:

How UDP Sockets Work:

1. Creation:

A UDP socket is created using a system call (e.g., `socket()` in POSIX systems).

2. Binding:

The socket is optionally bound to a specific IP address and port number.

3. Sending:

Data is encapsulated into a UDP datagram, which includes the destination IP address and port number, and sent using a send function (e.g., `sendto()` or `sendmsg()`).

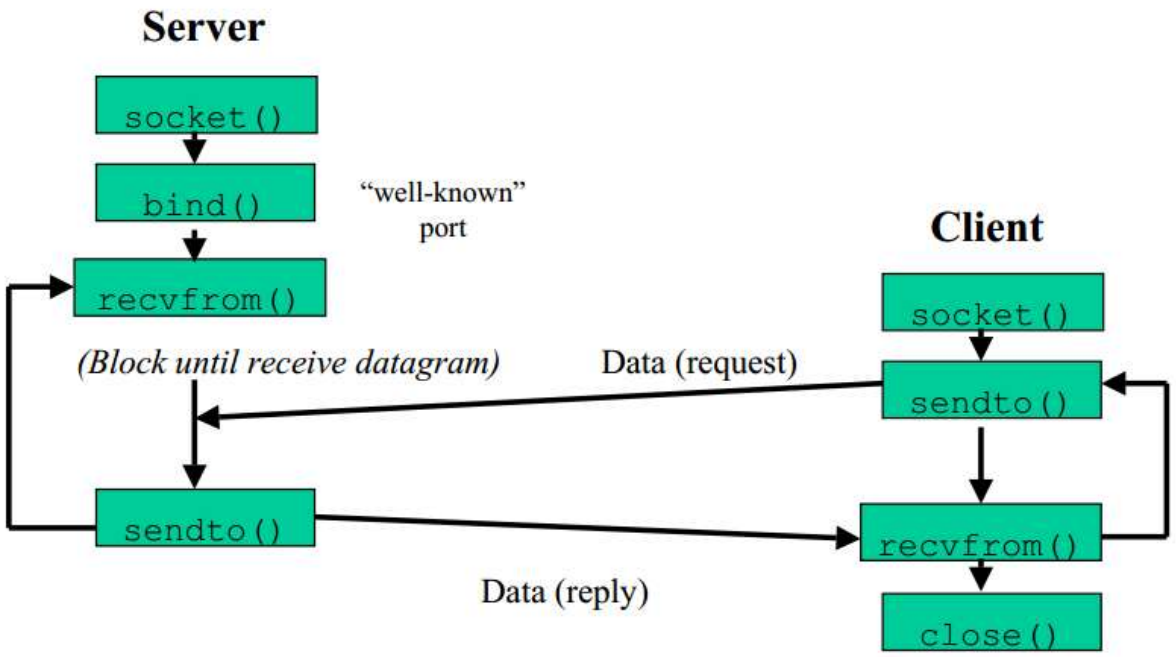
4. Receiving:

The application listens for incoming datagrams on the socket and receives them using a receive function (e.g., `recvfrom()` or `recvmsg()`).

5. No Connection State:

Unlike TCP, UDP doesn't maintain a persistent connection state. Each datagram is treated independently.

UDP Client-Server



TCP VS UDP :

TCP	UDP
Connected	Connectionless
State Memory	Stateless
Byte Stream	Packet/Datagram
Ordered Data Delivery	No Sequence Guarantee
Reliable	Lossy
Error Free	Error Packets Discarded
Handshake	No Handshake
Flow Control	No Flow Control
Relatively Slow	Relatively Fast
Point to Point	Supports Multicast
Security: SSL/TLS	Security: DTLS