**Socket in Linux System programming**

In Linux system programming, a socket is a communication endpoint that allows processes to communicate with each other over a network. It is a fundamental concept used for network programming and enables data exchange between processes running on different machines or on the same machine.

Sockets provide a programming interface for network communication using the socket system calls. In Linux, sockets are part of the Berkeley Sockets API, also known as the BSD Sockets API, which is widely used and supported across different operating systems.

There are two main types of sockets:

1. Stream Sockets (TCP): Stream sockets provide a reliable, connection-oriented communication channel. They use the Transmission Control Protocol (TCP) as the underlying protocol. TCP ensures that data sent through the socket arrives in the correct order and without errors. Stream sockets are commonly used for applications that require a reliable and ordered data stream, such as web browsing, file transfer, or email.
2. Datagram Sockets (UDP): Datagram sockets provide an unreliable, connectionless communication channel. They use the User Datagram Protocol (UDP) as the underlying protocol. UDP does not guarantee the order or reliability of data delivery, making it suitable for applications that require fast and lightweight communication, such as real-time multimedia streaming or online gaming.

To use sockets in Linux system programming, you typically follow these steps:

1. Create a socket: Use the **socket** system call to create a socket and specify the address family (such as AF\_INET for IPv4) and the socket type (such as SOCK\_STREAM for stream sockets or SOCK\_DGRAM for datagram sockets).
2. Bind (optional): For server applications, you can bind a socket to a specific address and port using the **bind** system call. This step is typically done for stream sockets to listen for incoming connections.
3. Listen (optional): For server applications using stream sockets, you can use the **listen** system call to put the socket into a passive listening mode, allowing it to accept incoming connections.
4. Connect (optional): For client applications using stream sockets, you can use the **connect** system call to establish a connection with a remote socket.
5. Send and Receive Data: Once the socket is created and connected (if applicable), you can use the **send**, **recv**, **sendto**, **recvfrom**, or other related system calls to send and receive data through the socket.
6. Close the socket: After you have finished using the socket, use the **close** system call to release the resources associated with it.

Sockets provide a powerful mechanism for inter-process communication over a network in Linux system programming. They allow processes to exchange data, establish connections, and build networked applications that can communicate with each other.

Certainly! Here are a few examples of socket programming in Linux using the C programming language:

1. TCP Server and Client:

TCP Server (server.c):

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <arpa/inet.h>

#define PORT 8080

#define BUFFER\_SIZE 1024

int main() {

 int server\_fd, new\_socket;

 struct sockaddr\_in address;

 int addrlen = sizeof(address);

 char buffer[BUFFER\_SIZE] = {0};

 char \*response = "Hello from server!";

 // Create socket

 if ((server\_fd = socket(AF\_INET, SOCK\_STREAM, 0)) == 0) {

 perror("socket failed");

 exit(EXIT\_FAILURE);

 }

 // Set up server address

 address.sin\_family = AF\_INET;

 address.sin\_addr.s\_addr = INADDR\_ANY;

 address.sin\_port = htons(PORT);

 // Bind socket to address and port

 if (bind(server\_fd, (struct sockaddr \*)&address, sizeof(address)) < 0) {

 perror("bind failed");

 exit(EXIT\_FAILURE);

 }

 // Listen for incoming connections

 if (listen(server\_fd, 3) < 0) {

 perror("listen failed");

 exit(EXIT\_FAILURE);

 }

 // Accept incoming connection

 if ((new\_socket = accept(server\_fd, (struct sockaddr \*)&address, (socklen\_t \*)&addrlen)) < 0) {

 perror("accept failed");

 exit(EXIT\_FAILURE);

 }

 // Read data from client

 if (read(new\_socket, buffer, BUFFER\_SIZE) < 0) {

 perror("read failed");

 exit(EXIT\_FAILURE);

 }

 printf("Client: %s\n", buffer);

 // Send response to client

 send(new\_socket, response, strlen(response), 0);

 printf("Response sent to client.\n");

 // Close sockets

 close(new\_socket);

 close(server\_fd);

 return 0;

}

TCP Client (client.c):

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <arpa/inet.h>

#define PORT 8080

#define BUFFER\_SIZE 1024

int main() {

 int sock = 0;

 struct sockaddr\_in serv\_addr;

 char \*message = "Hello from client!";

 char buffer[BUFFER\_SIZE] = {0};

 // Create socket

 if ((sock = socket(AF\_INET, SOCK\_STREAM, 0)) < 0) {

 printf("\n Socket creation error\n");

 return -1;

 }

 serv\_addr.sin\_family = AF\_INET;

 serv\_addr.sin\_port = htons(PORT);

 // Convert IPv4 and IPv6 addresses from text to binary form

 if (inet\_pton(AF\_INET, "127.0.0.1", &serv\_addr.sin\_addr) <= 0) {

 printf("\nInvalid address/ Address not supported\n");

 return -1;

 }

 // Connect to server

 if (connect(sock, (struct sockaddr \*)&serv\_addr, sizeof(serv\_addr)) < 0) {

 printf("\nConnection Failed\n");

 return -1;

 }

 // Send message to server

 send(sock, message, strlen(message), 0);

 printf("Message sent to server.\n");

 // Read response from server

 if (read(sock, buffer, BUFFER\_SIZE) < 0) {

 printf("\nRead failed\n");

 return -1;

 }

 printf("Server: %s\n", buffer);

 return 0;

}