

1. **socket() System Call:** Creates an end point for communication and returns a

descriptor:

**int socket (int AddressFamily, int Type, int Protocol);**

s = socket(domain, type, protocol);

address family or domain - AF\_APPLETALK, AF\_INET, AF\_PUP, AF\_UNIX

Type: SOCK\_STREAM, SOCK\_DGRAM, or SOCK\_RAW--are supported by AF\_INET and AF\_UNIX

**listen\_fd** = socket(AF\_INET, SOCK\_STREAM, 0);

**listen\_fd** = socket(AF\_UNIX, SOCK\_DGRAM, 0);

SOCK\_STREAM: Socket Type (TCP)

SOCK\_STREAM specifies the socket type to be created as a stream socket. This indicates that the socket will use the TCP (Transmission Control Protocol) protocol for communication. TCP provides reliable, connection-oriented, and stream-oriented communication.

SOCK\_DGRAM: Socket Type (UDP)

SOCK\_DGRAM specifies the socket type to be created as a datagram socket. This indicates that the socket will use the UDP (User Datagram Protocol) for communication. UDP provides connectionless and datagram-oriented communication. Unlike TCP, UDP does not provide guarantees for reliable and ordered delivery of data.

0 (Protocol)

The protocol parameter allows you to specify a particular protocol to use for the socket. In this case, a value of 0 indicates that the appropriate protocol for the chosen socket type and address family will be automatically selected. For SOCK\_STREAM, this typically means using the TCP protocol.

1. **Bind( ) System call:** Binds a name to a socket. The bind subroutine assigns a Name

parameter to an unnamed socket. It assigns a local protocol address to a socket.

**Syntax : int bind (int sockfd, struct sockaddr \*myaddr, int addrlen);**

**bind(listen\_fd, (struct sockaddr \*)&servaddr, sizeof(servaddr));**

The bind() function is used in socket programming to associate a socket with a specific local address and port. It's typically used on the server side to "bind" the socket to a specific network interface and port number, allowing the server to listen for incoming connections on that interface and port.

**int sockfd:** Socket Descriptor

This is the socket descriptor that was obtained when you created the socket using the **socket()** function. It represents the socket you want to bind to a specific address and port.

**struct sockaddr \*myaddr:** Local Address Structure

This parameter points to a structure of type **struct sockaddr** (or its derivatives like **struct sockaddr\_in** for IPv4 or **struct sockaddr\_in6** for IPv6). This structure specifies the local address (IP address and port) to which you want to bind the socket.

**int addrlen:** Address Length

This parameter specifies the size (in bytes) of the address structure pointed to by **myaddr.**

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

**#include <netinet/in.h>**

**int main() {**

**int sockfd;**

**struct sockaddr\_in server\_address;**

**sockfd = socket(AF\_INET, SOCK\_STREAM, 0);**

**server\_address.sin\_family = AF\_INET;**

**server\_address.sin\_addr.s\_addr = INADDR\_ANY; // Listen on all available network interfaces**

**server\_address.sin\_port = htons(8080); // Port number**

**if (bind(sockfd, (struct sockaddr \*)&server\_address, sizeof(server\_address)) == -1) {**

**perror("Bind failed");**

**exit(EXIT\_FAILURE);**

**}**

**// Rest of the server logic (listen, accept, etc.)**

**}**

1. **listen() System call:** This system call is used by a connection-oriented server to

indicate that it is willing to receive connections.

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

listen(listen\_fd, 10);

**int backlog:** Maximum Pending Connections

The **backlog** parameter specifies the maximum number of pending connections that the socket can queue up before they are accepted. This value gives an upper limit to the number of clients waiting to connect. Note that this parameter does not limit the total number of connections that can be established; it only affects the number of connections waiting to be accepted.

Example: #include <sys/socket.h>

int main() {

int sockfd;

// Create and bind the socket

if (listen(sockfd, 5) == -1) {

perror("Listen failed");

exit(EXIT\_FAILURE);

}

// Rest of the server logic (accept incoming connections, etc.)

}

1. **accept() System call:** The actual connection from some client process is waited for

by having the server execute the accept system call.

**Syntax: int accept (int sockfd, struct sockaddr \*cliaddr, int \*addrlen);**

comm\_fd = accept(listen\_fd, (struct sockaddr \*)NULL, NULL);

**struct sockaddr \*cliaddr:** Client Address Structure (Output)

This parameter is a pointer to a structure (of type **struct sockaddr** or its derivatives like struct **sockaddr\_in** for IPv4) where the details of the client's address will be stored. It's an output parameter because the function fills it with the client's information upon successful connection.

**int \*addrlen:** Address Length (Input and Output)

The **addrlen** parameter is a pointer to an integer that specifies the size of the **cliaddr** structure in bytes. Before calling **accept(),** you should initialize this integer to the size of the allocated address structure. After a connection is accepted, the function updates **addrlen** with the actual size of the client's address.

Example: #include <sys/socket.h>

int main() {

int listen\_sockfd, comm\_sockfd;

struct sockaddr\_in client\_address;

socklen\_t client\_address\_length;

// Create and bind the listening socket

listen(listen\_sockfd, 5); // Start listening

while (1) {

client\_address\_length = sizeof(client\_address);

// Accept an incoming connection

comm\_sockfd = accept(listen\_sockfd, (struct sockaddr \*)&client\_address, &client\_address\_length);

if (comm\_sockfd == -1) {

perror("Accept failed");

exit(EXIT\_FAILURE);

}

// Handle communication with the client using comm\_sockfd

close(comm\_sockfd); // Close the communication socket

}

// Rest of the server logic

}

1. **send( )**, **sendto( )**, **recv( )** and **recvfrom( ) System calls:** These system calls are

similar to the standard read and write functions.

**Syntax: recv(s, buf, sizeof buf, flags);**

recv(comm\_fd, str, 100, 0);

**comm\_fd:** Communication Socket Descriptor

comm\_fd is the socket descriptor of the communication socket over which you want to receive data. This socket should represent the active communication channel with a specific client.

**str:** Receive Buffer

str is a pointer to a buffer (an array of characters) where the received data will be stored. This buffer is where the received bytes will be placed after the **recv()** call.

**100:** Maximum Number of Bytes to Receive

This parameter specifies the maximum number of bytes that the function should receive and store in the str buffer. In this case, it's set to 100, meaning the function will try to receive up to 100 bytes of data.

**0:** Flags

The flags parameter allows you to provide optional flags that modify the behavior of the **recv()** function. A value of **0** typically means no special flags are used

Example: #include <sys/socket.h>

int main() {

int comm\_fd;

char buffer[1024];

// Create and set up the communication socket (comm\_fd)

while (1) {

// Receive data from the client

int bytes\_received = recv(comm\_fd, buffer, sizeof(buffer) - 1, 0);

if (bytes\_received == -1) {

perror("Receive failed");

exit(EXIT\_FAILURE);

} else if (bytes\_received == 0) {

// Connection closed by the client

break;

}

// Null-terminate the received data

buffer[bytes\_received] = '\0';

// Process the received data

printf("Received: %s", buffer);

}

// Rest of the communication logic

}

send(comm\_fd, str, strlen(str), 0);

**comm\_fd:** Communication Socket Descriptor

**comm\_fd** is the socket descriptor of the communication socket over which you want to send data. This socket should represent the active communication channel with a specific client.

**str:** Data to Send

str is a pointer to the data you want to send. It's assumed to be a null-terminated C-style string in this case.

**strlen(str):** Length of Data to Send

The **strlen()** function is used here to determine the length of the data in the str buffer. This value indicates how many bytes of data will be sent.

**0:** Flags

The flags parameter allows you to provide optional flags that modify the behavior of the **send()** function. A value of **0** typically means no special flags are used.

1. **close( ) system call**: The normal Unix close function is also used to close a socket

and terminate a TCP connection.

**Syntax : int close (int sockfd);**

**//server.c**

#include <sys/types.h>

#include <sys/socket.h>

#include <netdb.h>

#include <stdio.h>

#include <string.h>

int main() {

char str[100];

int listen\_fd, comm\_fd;

struct sockaddr\_in servaddr;

**// Create a socket**

listen\_fd = socket(AF\_INET, SOCK\_STREAM, 0);

**// Initialize the server address structure**

bzero(&servaddr, sizeof(servaddr));

servaddr.sin\_family = AF\_INET;

servaddr.sin\_addr.s\_addr = htonl(INADDR\_ANY);

servaddr.sin\_port = htons(22000);

**// Bind the socket to the server address**

bind(listen\_fd, (struct sockaddr \*)&servaddr, sizeof(servaddr));

**// Start listening for incoming connections**

listen(listen\_fd, 10);

**// Enter the main loop to accept and process connections**

while (1) {

**// Accept a client connection**

comm\_fd = accept(listen\_fd, (struct sockaddr \*)NULL, NULL);

**// Clear the str buffer**

bzero(str, 100);

**// Receive data from the client**

recv(comm\_fd, str, 100, 0);

printf("Echoing back - %s", str);

**// Send the received data back to the client**

send(comm\_fd, str, strlen(str), 0);

// Close the client connection

close(comm\_fd);

}

}

Include Required Headers:

In your code, you need to include the necessary header files for socket programming. These headers provide functions, data structures, and constants required for socket operations. Common headers include <sys/types.h>, <sys/socket.h>, <netinet/in.h>, and <netdb.h>.

Create a Socket:

You create a socket using the socket() function. This function takes parameters specifying the address family (IPv4 or IPv6), socket type (stream or datagram), and protocol. For example, socket(AF\_INET, SOCK\_STREAM, 0) creates a TCP socket.

Initialize Server/Client Address Structures:

If you're writing a server, you need to initialize a **struct sockaddr\_in** or **struct sockaddr\_in6** structure with the server's IP address, port number, and address family. For a client, you might need the server's address to connect.

Binding (Server-Side):

In server-side programming, after creating the socket, you bind it to a specific IP address and port using the bind() function. This step is essential for the server to listen for incoming connections on a specific interface and port.

Listening (Server-Side):

After binding, the server socket enters the listening state using the listen() function. This prepares the server to accept incoming connections from clients. The function takes the maximum number of pending connections as an argument.

Accepting Connections (Server-Side):

The accept() function is used by the server to accept incoming connections from clients. It creates a new communication socket for the accepted connection. The server then communicates with the client using this new socket.

Connecting (Client-Side):

On the client side, the connect() function is used to establish a connection to the server. It takes the client socket descriptor, the server's address structure, and the size of the address structure as arguments.

Sending and Receiving Data:

After the connection is established, both the client and server can use the send() and recv() functions to send and receive data. These functions handle data transmission over the established socket connection. For datagram sockets (UDP), you can use sendto() and recvfrom().

Closing Sockets:

Properly closing sockets is essential to release resources and ensure that the connection is terminated gracefully. The close() function is used to close sockets. Both the client and server should close their sockets when they are done with communication.

Error Handling:

Throughout the process, proper error handling is crucial. Check the return values of functions for errors and use functions like perror() or strerror() to get human-readable error messages.

1. The server creates a socket using **socket()** and initializes the server address structure with the desired IP address and port.
2. The **bind()** function associates the socket with the server address so that it can listen for incoming connections on that address and port.
3. The **listen()** function starts listening for incoming connections with a maximum queue length of 10 pending connections.
4. The server enters a loop using **while(1)** to continuously accept and process incoming connections.
5. **accept()** waits for an incoming connection and returns a new socket **(comm\_fd)** for communication with the connected client.
6. The **recv()** function receives data from the client and stores it in the str buffer.
7. The server prints the received data and then sends it back to the client using the **send()** function.
8. The **close()** function closes the communication socket after the data is sent back to the client.

**//client.c**

#include <sys/types.h>

#include <sys/socket.h>

#include <netdb.h>

#include <stdio.h>

#include <string.h>

int main(int argc, char \*\*argv) {

int sockfd, n;

char sendline[100];

char recvline[100];

struct sockaddr\_in servaddr;

**// Create a socket**

sockfd = socket(AF\_INET, SOCK\_STREAM, 0);

**// Initialize the server address structure**

bzero(&servaddr, sizeof(servaddr));

servaddr.sin\_family = AF\_INET;

servaddr.sin\_port = htons(22000);

servaddr.sin\_addr.s\_addr = inet\_addr("127.0.0.1");

**// Establish a connection to the server**

connect(sockfd, (struct sockaddr \*)&servaddr, sizeof(servaddr));

**// Enter a loop for communication with the server**

while (1) {

bzero(sendline, 100);

**////////////////////\***

**sendline: Pointer to Memory**

**sendline is a pointer to the start of the memory block (buffer) that you want to set to zero.**

**100: Number of Bytes**

**The second parameter specifies the number of bytes to set to zero, starting from the memory location pointed to by sendline. In this case, it's set to 100, indicating that the first 100 bytes of memory will be cleared.**

**Both bzero() and memset() set a range of bytes to zero, but memset() is more widely recognized and available across different platforms. \*///////////////////**

bzero(recvline, 100);

**// Read a line of text from the user (stdin)**

fgets(sendline, 100, stdin);

///////\*

**The function call fgets(sendline, 100, stdin); is used to read a line of text from the standard input (usually the keyboard) and store it in a buffer.**

**The fgets() function reads characters from the standard input until one of the following conditions is met:**

**The maximum number of characters specified (100 in this case) is read.**

**A newline character ('\n') is encountered.**

**The end-of-file (EOF) is reached.**

\*////////

**// Send the user's input to the server**

send(sockfd, sendline, strlen(sendline), 0);

**// Receive the response from the server**

recv(sockfd, recvline, 100, 0);

**// Print the server's response**

printf("%s", recvline);

}

}