1) **Socket Creation :**

used for creating a socket.
int sockfd = socket(domain, type, protocol)
sockfd: socket descriptor, an integer (like a file-handle)
domain: integer, communication domain e.g., AF_INET (IPv4 protocol) , AF_INET6 (IPv6 protocol)
type: communication type
SOCK_STREAM: TCP(reliable, connection oriented)
SOCK_DGRAM: UDP(unreliable, connectionless)
protocol: Protocol value for Internet Protocol(IP), which is 0. This is the same number which appears on protocol field in the IP header of a packet.(man protocols for more details)

2) **Setsockopt :**

int setsockopt(int sockfd, int level, int optname, const void *optval, socklen_t optlen)

This syscall helps in manipulating options for the socket referred to by the file descriptor sockfd. This syscall is completely optional, but it helps in the reuse of address and port. Prevents errors such as: “address already in use” .

3) **Bind :**

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

After creating the socket, the bind function binds the socket to the address and port number specified in addr(custom data structure). In the code attached above, I have bind the server to the localhost. Hence, I have used INADDR_ANY to specify the IP address.

4) **Listen :**

int listen(int sockfd, int backlog)

It puts the server socket in a passive mode, where it waits for the client to approach the server to make a connection. The backlog defines the maximum length to which the queue of pending connections for sockfd may grow. Suppose a connection request arrives when the queue is full. In that case, the client may receive an error with an indication of ECONNREFUSED.

5) **Accept :**

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

It extracts the first connection request on the queue of pending connections for the listening socket, sockfd, creates a new connected socket, and returns a new file descriptor referring to

that socket. At this point, a connection is established between client and server, and they are ready to transfer data.

6) **Connect :**

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

The connect() system call connects the socket referred to by the file descriptor sockfd to the address specified by addr. The server's address and port are specified in addr.

7) **Read**

read(sockfd, buff, sizeof(buff))
Read data from the file descriptor

8) **write**

write(sockfd, buff, sizeof(buff))
write data into the file descriptor

9) **send and recv**

num = send(s, addr_of_data, len_of_data, 0)
num = recv(s, addr_of_buffer, len_of_buffer, 0)
The send() and recv() calls specify:

- The socket s on which to communicate
- The address in storage of the buffer that contains, or will contain, the data (addr_of_data, addr_of_buffer)
- The size of this buffer (len_of_data, len_of_buffer)
- A flag that tells how the data is to be sent

Flag 0 tells TCP/IP to transfer the data normally. The server uses the socket that is returned from the accept() call.
These functions return the amount of data that was sent or received. Because stream sockets send and receive information in streams of data, it can take more than one send() or recv() to transfer all of the data. It is up to the client and the server to agree on some mechanism to signal that all of the data has been transferred.

10) **close**
close(s)

When the conversation is over, both the client and the server call close() to end the connection. Close() also deallocates the socket, freeing its space in the table of connections. To end a connection with a specific client, the server closes the socket returned by accept(). If the server

closes its original socket, it can no longer accept new connections, but it can still converse with the clients to which it is connected.

References :

1. https://www.tutorialspoint.com/unix_system_calls/socket.htm
2. https://www.geeksforgeeks.org/socket-programming-cc/