

Computer Networks Lab

Spring 2024

Week 06

Socket Programming

Socket

A socket is one endpoint of a two way communication link between two programs running on the network. The socket mechanism provides a means of inter-process communication (IPC) by establishing named contact points between which the communication takes place. Like 'Pipe' is used to create pipes and sockets are created using 'socket' system calls. The socket provides a bidirectional FIFO Communication facility over the network. A socket connecting to the network is created at each end of the communication. Each socket has a specific address. This address is composed of an IP address and a port number.

Sockets are generally employed in client server applications. The server creates a socket, attaches it to a network port address then waits for the client to contact it. The client creates a socket and then attempts to connect to the server socket. When the connection is established, transfer of data takes place.

Sockets are primarily associated with the Transport layer. The Transport layer is responsible for end-to-end communication and ensures the reliable delivery of data between applications on different devices. Two key protocols within the Transport layer are Transmission Control Protocol (TCP) and User Datagram Protocol (UDP), each serving different communication needs.



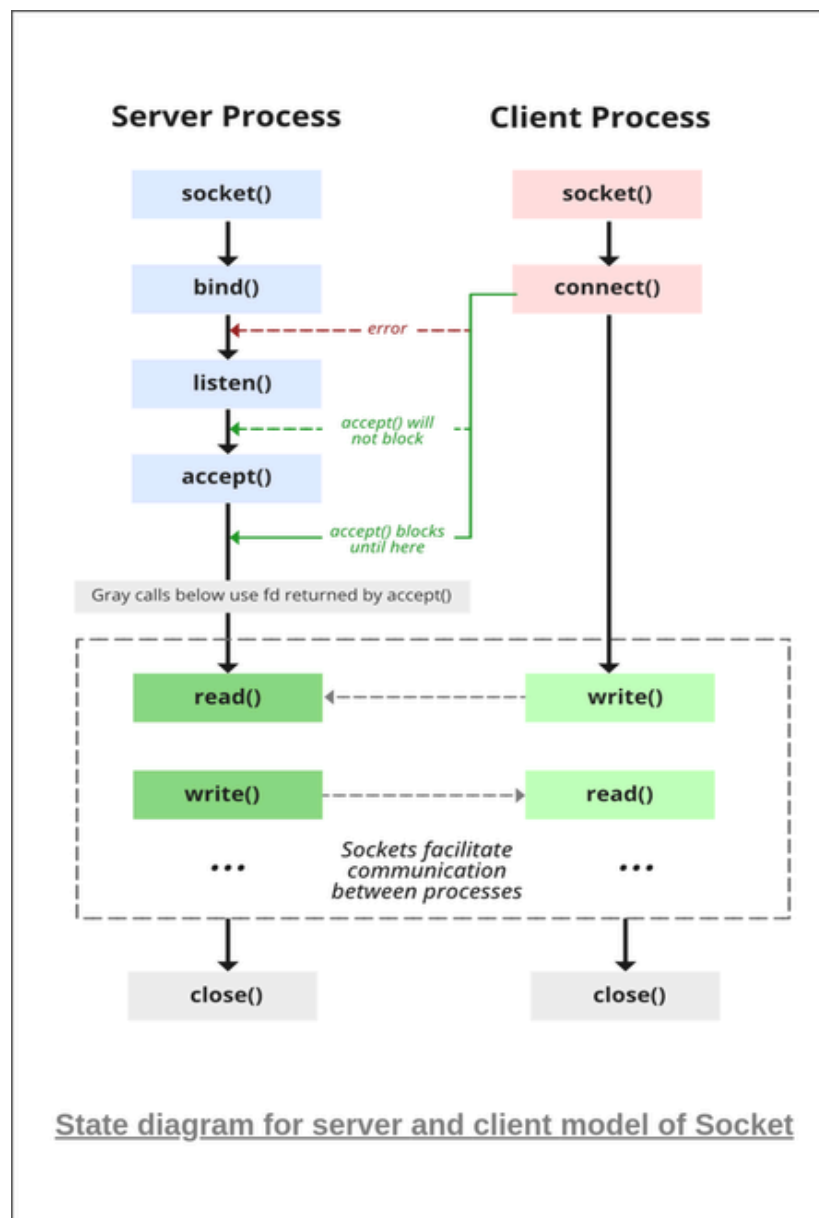
Socket Function Calls

1. **Socket():** To create a socket
2. **Bind():** It's a socket identification like a telephone number to contact

3. **Listen():** Ready to receive a connection
4. **Connect():** Ready to act as a sender
5. **Accept():** Confirmation, it is like accepting to receive a call from a sender
6. **Send():** To send data (write)
7. **Recv():** To receive data (read)
8. **Close():** To close a connection

Socket Programming

Socket programming is a way of connecting two nodes on a network to communicate with each other. One socket(node) listens on a particular port at an IP, while the other socket reaches out to the other to form a connection. The server forms the listener socket while the client reaches out to the server.



Stages for Server

The server is created using the following steps:

1. Socket Creation

`int sockfd = socket(domain, type, protocol)`

sockfd: socket descriptor, an integer (like a file handle)

domain: integer, specifies communication domain. The server address includes the address family (**AF_INET** for IPv4), IP address (**INADDR_ANY** for any local address), and port number.

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 that appears on the protocol field in the IP header of a packet.

```
// create the server socket
int server_socket;
server_socket = socket(AF_INET, SOCK_STREAM, 0);
```

2. Bind

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

After the creation of the socket, the bind function binds the socket to the address and port number specified in `addr`(custom data structure). In the example code, we bind the server to the localhost, hence we use `INADDR_ANY` to specify the IP address.

```
// define the server address
struct sockaddr_in server_address;
server_address.sin_family = AF_INET;
server_address.sin_port = htons(3001);
server_address.sin_addr.s_addr = INADDR_ANY;

// bind the socket to our specified IP and port
bind(server_socket, (struct sockaddr*) &server_address, sizeof(server_address));
```

3. 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. If a connection request arrives when the queue is full, the client may receive an error with an indication of `ECONNREFUSED`.

```
listen(server_socket, 5);
```

4. 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, the connection is established between client and server, and they are ready to transfer data.

```
int client_socket;  
client_socket = accept(server_socket, NULL, NULL);
```

Stages for Client

1. Socket connection

Exactly the same as that of server's socket creation

```
// create the socket  
int sock;  
sock = socket(AF_INET, SOCK_STREAM, 0);
```

2. Connect:

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

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

```
//setup an address  
struct sockaddr_in server_address;  
server_address.sin_family = AF_INET;  
server_address.sin_addr.s_addr = INADDR_ANY;  
server_address.sin_port = htons(3001);  
  
connect(sock, (struct sockaddr *) &server_address, sizeof(server_address));
```

Sample Code:

Server Side

```
#include <stdio.h> // basic C header
```

```
#include <stdlib.h>
```

```
#include <sys/types.h> // provides required data types
```

```
#include <sys/socket.h> // holds address family and socket functions
```

```
#include <unistd.h>
```

```
#include <netinet/in.h> // has the sockaddr_in structure
```

```
int main() {

    char server_message[256] = "Hi, Yes you have reached the server!";
    char buf[200];
    // create the server socket
    int server_socket;
    server_socket = socket(AF_INET, SOCK_STREAM, 0);

    // define the server address
    struct sockaddr_in server_address;
    server_address.sin_family = AF_INET;
    server_address.sin_port = htons(3001);
    server_address.sin_addr.s_addr = INADDR_ANY;

    // bind the socket to our specified IP and port
    bind(server_socket, (struct sockaddr*) &server_address, sizeof(server_address));
    listen(server_socket, 5);
    int client_socket;
    client_socket = accept(server_socket, NULL, NULL);

    recv(client_socket, &buf, sizeof(buf), 0);
    printf("\n %s \n", buf);
    send(client_socket, server_message, sizeof(server_message), 0);

    // close the socket
    close(server_socket);

    return 0;
}
```

Client Side

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <unistd.h>
#include <netinet/in.h>

int main() {

    char request[256] = "hello from the client side... !";
    char buf[200];

    // create the socket
    int sock;
    sock = socket(AF_INET, SOCK_STREAM, 0);

    //setup an address
    struct sockaddr_in server_address;
    server_address.sin_family = AF_INET;
    server_address.sin_addr.s_addr = INADDR_ANY;
    server_address.sin_port = htons(3001);

    connect(sock, (struct sockaddr *) &server_address, sizeof(server_address));

    send(sock, request, sizeof(request), 0);
    recv(sock, &buf, sizeof(buf), 0);
    printf("\n %s \n",buf);

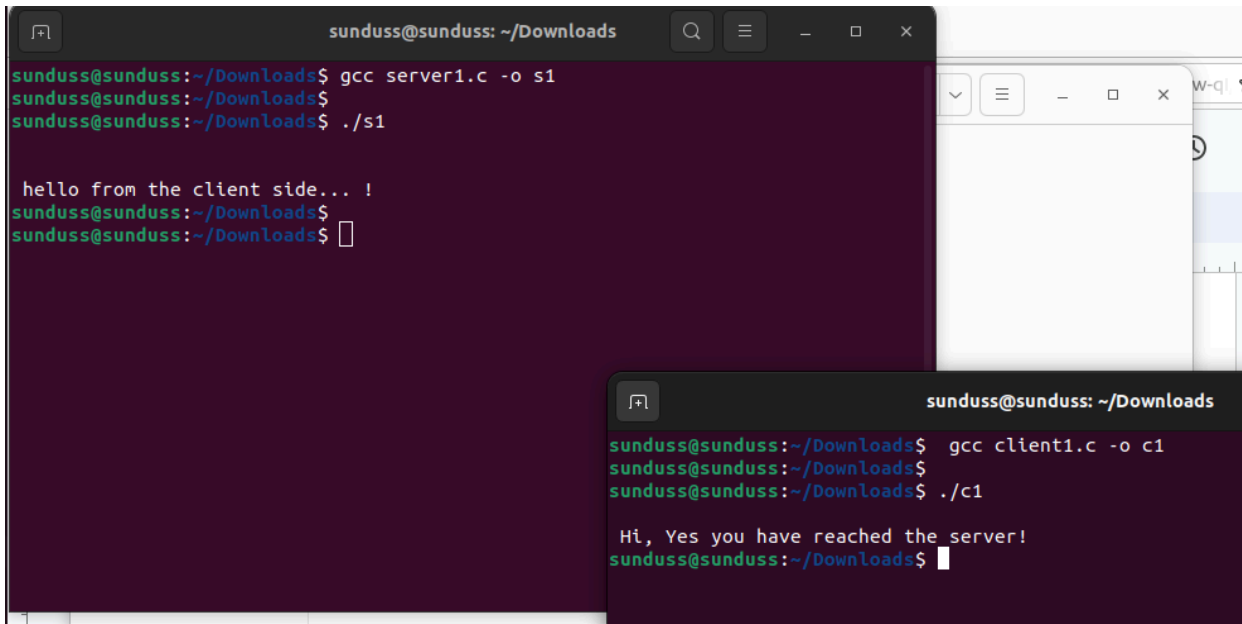
    close(sock);

    return 0;
}
```

How to Run the code?

Open two separate terminals, each accessing the folder you have your code in. Compile both the files using the command : ***gcc filename.c -o exefilenameTobeassigned***

Once compiled without errors, access the server exe file, and then the client exe file. For reference see the image attached below.



```
sunduss@sunduss: ~/Downloads
sunduss@sunduss:~/Downloads$ gcc server1.c -o s1
sunduss@sunduss:~/Downloads$
sunduss@sunduss:~/Downloads$ ./s1

hello from the client side... !
sunduss@sunduss:~/Downloads$
sunduss@sunduss:~/Downloads$

sunduss@sunduss: ~/Downloads
sunduss@sunduss:~/Downloads$ gcc client1.c -o c1
sunduss@sunduss:~/Downloads$
sunduss@sunduss:~/Downloads$ ./c1

Hi, Yes you have reached the server!
sunduss@sunduss:~/Downloads$
```

Practice tasks

Task 1:

Create a simple client-server interaction using socket programming in C. The server should listen for incoming connections and display messages received from the client. The client, on the other hand, should allow users to input messages to be sent to the server. Implement a mechanism for the client to signal the end of the conversation, and ensure that the server reacts accordingly by displaying the messages and terminating the connection. Provide both the client and server source code along with instructions for running the programs.

Submission Guidelines:

- Source code for both client and server
- Screenshot of the terminal with message transfer.



National University of Computer and Emerging Sciences (NUCES) Islamabad