# Experiment 6: Socket Programming-I

**Aim:** To use TCP Sockets for Inter Process Communication

**Objective:** After carrying out this experiment, students will be able to:

* Apply TCP Socket programming technique to establish IPC between remote processes
* Analyse the difference between sockets and other enabling techniques for IPC such as Pipes and Message Queues

**Problem statement:** You are required to write programs to implement a TCP based echo server. The functionality of this server is that it should echo any data it receives from a client back to it.

**Analysis:** While analyzing your program, you are required to address the following points:

* How is socket programming different from other techniques for IPC such as Pipes and Message Queues?
* What happens if the number of incoming client requests exceeds the second argument of the listen() function in the server?

**MARKS DISTRIBUTION**

|  |  |  |
| --- | --- | --- |
| **Component** | **Maximum Marks** | **Marks Obtained** |
| Preparation of Document | 7 |  |
| Results | 7 |  |
| Viva | 6 |  |
| **Total** | **20** |  |

Submitted by: SATYAJIT GHANA

Register No: 17ETCS002159

1. Algorithm/Flowchart

Client code description:

* Include all the standard libraries and the libraries required for socket programming like <netb.h> which is responsible for reserving highest port internet number, <sys/socket.h>,
* Create a structure of socket

struct sockaddr\_in server

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

bzero( &servaddr, sizeof(servaddr))

servaddr.sin\_family = AF\_INET

servaddr.sin\_addr.s\_addr = htons(INADDR\_ANY)

servaddr.sin\_port = htons(8080)

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

listen(listen\_fd, 10)

* Echo the message back to the server using commands like accept(), with appropriate arguments.

Server code description:

* Include all the standard and necessary files required for socket programming.
* Create a structure for the server as well by using struct data-type.
* Pass the main string that needs to be echoed back by the client.

while(1)

bzero(sendline,100)

bzero(recvline,100)

fgets(sendline,100,stdin)

write(sockfd,sendline,strlen(sendline)+1)

read(sockfd,recvline,100)

printf("From Server: %s",recvline)

1. Program

// Client side C/C++ program to demonstrate Socket programming

#include <iostream>

#include <string>

#include <arpa/inet.h>

#include <stdio.h>

#include <string.h>

#include <sys/socket.h>

#include <unistd.h>

#define PORT 8080

#define IPADDR "127.0.0.1"

#define BUFSIZE 1024

/\*\*

 \* Client Side code, connects to the server,

 \* sends the input from stdin to server

 \* recieves from server and prints the data

\*/

int main(int argc, char const \*argv[]) {

    int sock = 0, valread;

    struct sockaddr\_in serv\_addr;

    char buffer[BUFSIZE] = {0};

    std::cout << "CLIENT" << std::endl;

    // create the socket

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

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

        return -1;

    }

    // create the structure member values

    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, IPADDR, &serv\_addr.sin\_addr) <= 0) {

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

        return -1;

    }

    // connect to the server

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

        printf("\nConnection Failed \n");

        return -1;

    }

    // infinite loop

    while (true) {

        std::string hello;

        std::cout << "Enter the Message to be sent to server : ";

        std::cin >> hello;

        send(sock, hello.c\_str(), hello.size(), 0);

        std::cout << "Message sent to Server !\n"

                  << std::endl;

        valread = read(sock, buffer, BUFSIZE);

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

        // zero the buffer

        bzero(buffer, sizeof(buffer));

    }

    return 0;

}

#include <netinet/in.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/socket.h>

#include <unistd.h>

#include <iostream>

#include <string>

#define PORT 8080

#define BUFSIZE 1024

/\*\*

 \* Server Side code, takes input from stdin and sends it to the client

 \* Recieves data from client and prints it.

\*/

int main(int argc, char const \*argv[]) {

    int server\_fd, new\_socket, valread;

    struct sockaddr\_in address;

    int opt = 1;

    int addrlen = sizeof(address);

    char buffer[BUFSIZE] = {0};

    std::cout << "SERVER" << std::endl;

    // Creating socket file descriptor

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

        perror("socket failed");

        exit(EXIT\_FAILURE);

    }

    // Forcefully attaching socket to the port 8080

    if (setsockopt(server\_fd, SOL\_SOCKET, SO\_REUSEADDR | SO\_REUSEPORT,

                   &opt, sizeof(opt))) {

        perror("setsockopt");

        exit(EXIT\_FAILURE);

    }

    // assign the structure members

    address.sin\_family = AF\_INET;

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

    address.sin\_port = htons(PORT);

    // Forcefully attaching socket to the port 8080

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

        perror("bind failed");

        exit(EXIT\_FAILURE);

    }

    // listen to connections, max 3

    if (listen(server\_fd, 3) < 0) {

        perror("listen");

        exit(EXIT\_FAILURE);

    }

    // accept connection

    if ((new\_socket = accept(server\_fd, (struct sockaddr \*)&address,

                             (socklen\_t \*)&addrlen)) < 0) {

        perror("accept");

        exit(EXIT\_FAILURE);

    }

    // keep running until the server or the client is closed

    while (true) {

        valread = read(new\_socket, buffer, BUFSIZE);

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

        // zero the buffer

        bzero(buffer, sizeof(buffer));

        std::string hello;

        std::cout << "Enter the message to be sent to client : ";

        std::cin >> hello;

        send(new\_socket, hello.c\_str(), hello.size(), 0);

        printf("Message sent to Client !\n\n");

    }

    return 0;

}

1. Results



Figure ‑ Server Output



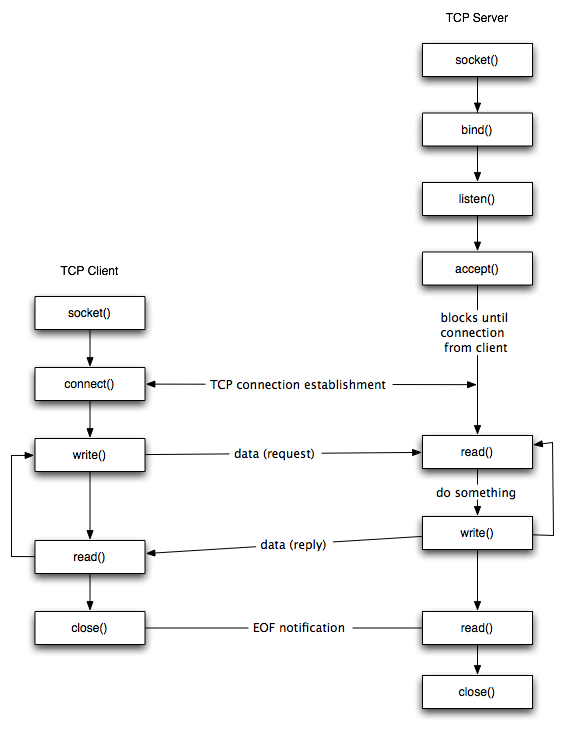
Figure ‑ Client Output

1. Analysis and Discussions

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 other socket reaches out to the other to form a connection. Server forms the listener socket while client reaches out to the server.

**Pipes** are used as a medium to simply exchange information between sender and receiver. The messages that are exchanged are not analyzed or processed. Pipes are used in inter-process communication.

**Message queues** – In this, a queue is used to store the messages which are being exchanged between the sender and the receiver. A process inputs a message in the queue which allows another process to read it. An interface is provided to the processes and these processes use this interface to access the message queue to either put one message in the queue for other multiple processes to read it or read one message from the queue.



* **Socket creation:**

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)

* **Setsockopt:**

int setsockopt(int sockfd, int level, int optname, const void \*optval, socklen\_t optlen);

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

* **Bind:**

int bind(int sockfd, const struct sockaddr \*addr, socklen\_t addrlen);

After creation of the socket, 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.

* **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.

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

**Stages for Client**

* **Socket connection:** Exactly same as that of server’s socket creation
* **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. Server’s address and port is specified in addr.

1. Conclusions

Sockets allow communication between two different processes on the same or different machines. To be more precise, it's a way to talk to other computers using standard Unix file descriptors. In Unix, every I/O action is done by writing or reading a file descriptor. A file descriptor is just an integer associated with an open file and it can be a network connection, a text file, a terminal, or something else.

To a programmer, a socket looks and behaves much like a low-level file descriptor. This is because commands such as read() and write() work with sockets in the same way they do with files and pipes.

Sockets were first introduced in 2.1BSD and subsequently refined into their current form with 4.2BSD. The sockets feature is now available with most current UNIX system releases.

1. Comments
   1. Limitations of the experiment

The experiment deals with only transmitting strings over the network, since they are simpler to transmit, transmitting different data structures and retaining the type of data is difficult, but justifies the complex real-world network transfer.

* 1. Limitations of the results obtained

The connection here is limited to only one, to test the experiment properly, a few more connections can be made.

* 1. Learning

We learnt how to setup simple socket connection over the network and transmit strings.

* 1. Recommendations

Something like Google’s protobuf can be used to transmit different data structures.