**Lab Report No : 03**

**Lab Report Name : Socket Programming Implementation**

**Name: Md. Abul Basar**

**ID: IT-17026**

**Socket**

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

**Where is Socket Used?**

A Unix Socket is used ina client-server application framework. A server is a process that performs some functions on request from a client.Most of the application –level protocols like [FTP.SMTP,and](ftp://FTP.SMTP,and) POP3 make use of sockets to establish connection between client and server and then for exchanging data.

**Socket Types**

There are four types of sockets available to the users. The first two are most commonly used and the last two are rarely used.

Processes are presumed to communicate only between sockets of the same type but there is no restriction that prevents communication between sockets of different types.

**Three types of sockets are supported**:

* **Stream sockets -**  Delivery ina networked environment is guaranteed. If you send through the stream socket three items “A,B,C”, they will arrive in the same order – “A,B,C”
* **Datagram sockets** - allow processes to use UDP to communicate. A datagram socket supports bidirectional flow of messages. A process on a datagram socket can receive messages in a different order from the sending sequence and can receive duplicate messages. Record boundaries in the data are preserved. The socket type is SOCK\_DGRAM.
* **Raw sockets** - provide access to ICMP. These sockets are normally datagram oriented, although their exact characteristics are dependent on the interface provided by the protocol. Raw sockets are not for most applications. They are provided to support developing new communication protocols or for access to more esoteric facilities of an existing protocol. Only superuser processes can use raw sockets. The socket type is SOCK\_RAW.

**Java Socket Programming**

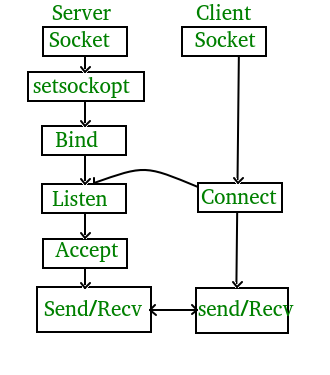
Java socket programming is used for communication between the applications running on different JRE. Java Socket programming can be connection –oriented or connection-less.

The client in socket programming must know two information :

1. IP Address of server and

2. Port number

In this application,client sends a message to the server server reads the message and prints it.Here,two classes are being used: Socket and ServerSocket.The Socket class is used to communicate client and server.Through this class,we can read and write message. The ServerSocket class is used at server-side.The accept() method of ServerSocket class blocks the console until the client is connected.

****

**A program to design a TCP Client –Server Which implements Echo protocol  
Server.c**

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

#include <unistd.h>

#include <stdio.h>

#include <sys/socket.h>

#include <stdlib.h>

#include <netinet/in.h>

#include <string.h>

#define PORT 8080

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[1024] = {0};

    char \*hello = "Hello from server";

    // 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);

    }

    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);

    }

    if (listen(server\_fd, 3) < 0)

    {

        perror("listen");

        exit(EXIT\_FAILURE);

    }

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

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

    {

        perror("accept");

        exit(EXIT\_FAILURE);

    }

    valread = read( new\_socket , buffer, 1024);

    printf("%s\n",buffer );

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

    printf("Hello message sent\n");

    return 0;

}

**Client.c**

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

#include <stdio.h>

#include <sys/socket.h>

#include <arpa/inet.h>

#include <unistd.h>

#include <string.h>

#define PORT 8080

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

{

    int sock = 0, valread;

    struct sockaddr\_in serv\_addr;

    char \*hello = "Hello from client";

    char buffer[1024] = {0};

    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;

    }

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

    {

        printf("\nConnection Failed \n");

        return -1;

    }

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

    printf("Hello message sent\n");

    valread = read( sock , buffer, 1024);

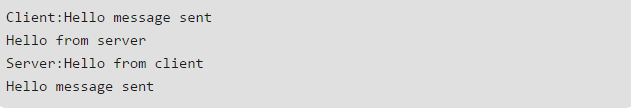
    printf("%s\n",buffer );

    return 0;

}

**Compiling:**  
gcc client.c -o client  
gcc server.c -o server

**Output :**



**Exercise 2.2.2**

# UDP Server-Client implementation in C

In UDP, the client does not form a connection with the server like in TCP and instead just sends a datagram. Similarly, the server need not accept a connection and just waits for datagrams to arrive. Datagrams upon arrival contain the address of sender which the server uses to send data to the correct client.

**Time Protocol**

When used via UDP the time service works as follows:

S: Listen on port 37 (45 octal).

U: Send an empty datagram to port 37.

S: Receive the empty datagram.

S: Send a datagram containing the time as a 32 bit binary number.

U: Receive the time datagram.

**The Time**

The time is the number of seconds since 00:00 (midnight) 1 January 1900

GMT, such that the time 1 is 12:00:01 am on 1 January 1900 GMT; this

base will serve until the year 2036.

For example:

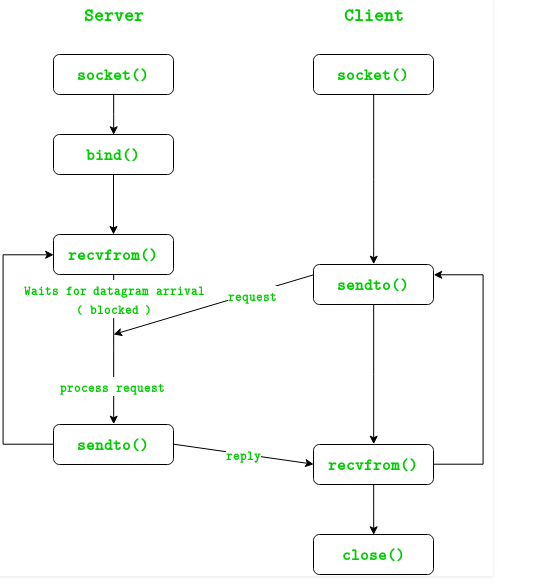
the time 2,208,988,800 corresponds to 00:00 1 Jan 1970 GMT,

2,398,291,200 corresponds to 00:00 1 Jan 1976 GMT,

2,524,521,600 corresponds to 00:00 1 Jan 1980 GMT,

2,629,584,000 corresponds to 00:00 1 May 1983 GMT,

**State diagram for server and client model of UDP Protocol**



The entire process can be broken down into following steps :  
**UDP Server :**

1. Create UDP socket.
2. Bind the socket to server address.
3. Wait until datagram packet arrives from client.
4. Process the datagram packet and send a reply to client.
5. Go back to Step 3.

**UDP Client :**

1. Create UDP socket.
2. Send message to server.
3. Wait until response from server is recieved.
4. Process reply and go back to step 2, if necessary.
5. Close socket descriptor and exit.

**Code:**

**UDPServer.c**// Server side implementation of UDP client-server model

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <string.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <arpa/inet.h>

#include <netinet/in.h>

#define PORT     8080

#define MAXLINE 1024

// Driver code

int main() {

    int sockfd;

    char buffer[MAXLINE];

    char \*hello = "Hello from server";

    struct sockaddr\_in servaddr, cliaddr;

    // Creating socket file descriptor

    if ( (sockfd = socket(AF\_INET, SOCK\_DGRAM, 0)) < 0 ) {

        perror("socket creation failed");

        exit(EXIT\_FAILURE);

    }

    memset (&servaddr, 0, sizeof(servaddr));

    memset(&cliaddr, 0, sizeof(cliaddr));

    // Filling server information

    servaddr.sin\_family    = AF\_INET; // IPv4

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

    servaddr.sin\_port = htons(PORT);

    // Bind the socket with the server address

    if ( bind(sockfd, (const struct sockaddr \*)&servaddr,

            sizeof(servaddr)) < 0 )

    {

        perror("bind failed");

        exit(EXIT\_FAILURE);

    }

    int len, n;

    len = sizeof(cliaddr);  //len is value/resuslt

    n = recvfrom(sockfd, (char \*)buffer, MAXLINE,

                MSG\_WAITALL, ( struct sockaddr \*) &cliaddr,

                &len);

    buffer[n] = '\0';

    printf("Client : %s\n", buffer);

    sendto(sockfd, (const char \*)hello, strlen(hello),

        MSG\_CONFIRM, (const struct sockaddr \*) &cliaddr,

            len);

    printf("Hello message sent.\n");

    return 0;

}

**UDPClient.C**// Client side implementation of UDP client-server model

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <string.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <arpa/inet.h>

#include <netinet/in.h>

#define PORT     8080

#define MAXLINE 1024

// Driver code

int main() {

    int sockfd;

    char buffer[MAXLINE];

    char \*hello = "Hello from client";

    struct sockaddr\_in     servaddr;

    // Creating socket file descriptor

    if ( (sockfd = socket(AF\_INET, SOCK\_DGRAM, 0)) < 0 ) {

        perror("socket creation failed");

        exit(EXIT\_FAILURE);

    }

    memset(&servaddr, 0, sizeof(servaddr));

    // Filling server information

    servaddr.sin\_family = AF\_INET;

    servaddr.sin\_port = htons(PORT);

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

    int n, len;

    sendto(sockfd, (const char \*)hello, strlen(hello),

        MSG\_CONFIRM, (const struct sockaddr \*) &servaddr,

            sizeof(servaddr));

    printf("Hello message sent.\n");

    n = recvfrom(sockfd, (char \*)buffer, MAXLINE,

                MSG\_WAITALL, (struct sockaddr \*) &servaddr,

                &len);

    buffer[n] = '\0';

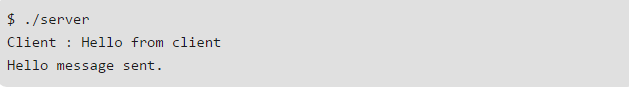
    printf("Server : %s\n", buffer);

    close(sockfd);

    return 0;

}

**OutPut:**



**Exercise 2.3.1**

**TFTP :**

**Run TFTP Server**  
$./tftp\_s  
**Run TFTP Client**  
$./tftp\_c GET/PUT server\_address file\_name  
Tftp Client.c

|  |
| --- |
| /\*\* |
|  | \* tftp\_c.c - tftp client |
|  | \*/ |
|  | #include <stdio.h> |
|  | #include <stdlib.h> |
|  | #include <unistd.h> |
|  | #include <errno.h> |
|  | #include <string.h> |
|  | #include <sys/types.h> |
|  | #include <sys/socket.h> |
|  | #include <sys/time.h> |
|  | #include <netinet/in.h> |
|  | #include <arpa/inet.h> |
|  | #include <netdb.h> |
|  | #include "utility.h" |
|  |  |
|  | void \*get\_in\_addr(struct sockaddr \*sa) |
|  | { |
|  | if (sa->sa\_family == AF\_INET) { |
|  | return &(((struct sockaddr\_in\*)sa)->sin\_addr); |
|  | } |
|  | return &(((struct sockaddr\_in6\*)sa)->sin6\_addr); |
|  | } |
|  |  |
|  | //CHECKS FOR TIMEOUT |
|  | int check\_timeout(int sockfd, char \*buf, struct sockaddr\_storage their\_addr, socklen\_t addr\_len){ |
|  | fd\_set fds; |
|  | int n; |
|  | struct timeval tv; |
|  |  |
|  | // set up the file descriptor set |
|  | FD\_ZERO(&fds); |
|  | FD\_SET(sockfd, &fds); |
|  |  |
|  | // set up the struct timeval for the timeout |
|  | tv.tv\_sec = TIME\_OUT; |
|  | tv.tv\_usec = 0; |
|  |  |
|  | // wait until timeout or data received |
|  | n = select(sockfd+1, &fds, NULL, NULL, &tv); |
|  | if (n == 0){ |
|  | printf("timeout\n"); |
|  | return -2; // timeout! |
|  | } else if (n == -1){ |
|  | printf("error\n"); |
|  | return -1; // error |
|  | } |
|  |  |
|  | return recvfrom(sockfd, buf, MAXBUFLEN-1 , 0, (struct sockaddr \*)&their\_addr, &addr\_len); |
|  | } |
|  |  |
|  | int main(int argc, char\* argv[]){ |
|  | int sockfd; |
|  | struct addrinfo hints, \*servinfo, \*p; |
|  | int rv; |
|  | int numbytes; |
|  | char buf[MAXBUFLEN]; |
|  | char s[INET6\_ADDRSTRLEN]; |
|  | struct sockaddr\_storage their\_addr; |
|  | socklen\_t addr\_len; |
|  |  |
|  | if(argc != 4){// CHECKS IF args ARE VALID |
|  | fprintf(stderr,"USAGE: tftp\_c GET/PUT server filename\n"); |
|  | exit(1); |
|  | } |
|  |  |
|  | char \*server = argv[2];// server address |
|  | char \*file = argv[3]; // file name on which operation has to be done |
|  |  |
|  | //===========CONFIGURATION OF CLIENT - STARTS=========== |
|  | memset(&hints, 0, sizeof hints); |
|  | hints.ai\_family = AF\_UNSPEC; |
|  | hints.ai\_socktype = SOCK\_DGRAM; |
|  | if((rv = getaddrinfo(server, SERVERPORT, &hints, &servinfo)) != 0){ |
|  | fprintf(stderr, "CLIENT: getaddrinfo: %s\n", gai\_strerror(rv)); |
|  | return 1; |
|  | } |
|  |  |
|  | // loop through all the results and make a socket |
|  | for(p = servinfo; p != NULL; p = p->ai\_next) { |
|  | if ((sockfd = socket(p->ai\_family, p->ai\_socktype, p->ai\_protocol)) == -1){ |
|  | perror("CLIENT: socket"); |
|  | continue; |
|  | } |
|  | break; |
|  | } |
|  | if(p == NULL){ |
|  | fprintf(stderr, "CLIENT: failed to bind socket\n"); |
|  | return 2; |
|  | } |

**Tftp Server.c**

|  |
| --- |
| /\*\* |
|  | \* tftp\_s.c - tftp server |
|  | \*/ |
|  | #include <stdio.h> |
|  | #include <stdlib.h> |
|  | #include <unistd.h> |
|  | #include <errno.h> |
|  | #include <string.h> |
|  | #include <sys/types.h> |
|  | #include <sys/socket.h> |
|  | #include <netinet/in.h> |
|  | #include <arpa/inet.h> |
|  | #include <netdb.h> |
|  | #include "utility.h" |
|  |  |
|  | void \*get\_in\_addr(struct sockaddr \*sa) |
|  | { |
|  | if (sa->sa\_family == AF\_INET) { |
|  | return &(((struct sockaddr\_in\*)sa)->sin\_addr); |
|  | } |
|  | return &(((struct sockaddr\_in6\*)sa)->sin6\_addr); |
|  | } |
|  |  |
|  | //CHECKS FOR TIMEOUT |
|  | int check\_timeout(int sockfd, char \*buf, struct sockaddr\_storage their\_addr, socklen\_t addr\_len){ |
|  | fd\_set fds; |
|  | int n; |
|  | struct timeval tv; |
|  |  |
|  | // set up the file descriptor set |
|  | FD\_ZERO(&fds); |
|  | FD\_SET(sockfd, &fds); |
|  |  |
|  | // set up the struct timeval for the timeout |
|  | tv.tv\_sec = TIME\_OUT; |
|  | tv.tv\_usec = 0; |
|  |  |
|  | // wait until timeout or data received |
|  | n = select(sockfd+1, &fds, NULL, NULL, &tv); |
|  | if (n == 0){ |
|  | printf("timeout\n"); |
|  | return -2; // timeout! |
|  | } else if (n == -1){ |
|  | printf("error\n"); |
|  | return -1; // error |
|  | } |
|  |  |
|  | return recvfrom(sockfd, buf, MAXBUFLEN-1 , 0, (struct sockaddr \*)&their\_addr, &addr\_len); |
|  | } |
|  |  |
|  | int main(void){ |
|  | int sockfd; |
|  | struct addrinfo hints, \*servinfo, \*p; |
|  | int rv; |
|  | int numbytes; |
|  | struct sockaddr\_storage their\_addr; |
|  | char buf[MAXBUFLEN]; |
|  | socklen\_t addr\_len; |
|  | char s[INET6\_ADDRSTRLEN]; |
|  |  |
|  | //===========CONFIGURATION OF SERVER - START=========== |
|  | memset(&hints, 0, sizeof hints); |
|  | hints.ai\_family = AF\_UNSPEC; // set to AF\_INET to force IPv4 |
|  | hints.ai\_socktype = SOCK\_DGRAM; |
|  | hints.ai\_flags = AI\_PASSIVE; // use my IP |
|  |  |
|  | if ((rv = getaddrinfo(NULL, MYPORT, &hints, &servinfo)) != 0) { |
|  | fprintf(stderr, "SERVER: getaddrinfo: %s\n", gai\_strerror(rv)); |
|  | return 1; |
|  | } |
|  |  |
|  | // loop through all the results and bind to the first we can |
|  | for(p = servinfo; p != NULL; p = p->ai\_next) { |
|  | if ((sockfd = socket(p->ai\_family, p->ai\_socktype, p->ai\_protocol)) == -1) { |
|  | perror("SERVER: socket"); |
|  | continue; |
|  | } |
|  | if (bind(sockfd, p->ai\_addr, p->ai\_addrlen) == -1) { |
|  | close(sockfd); |
|  | perror("SERVER: bind"); |
|  | continue; |
|  | } |
|  | break; |
|  | } |
|  | if (p == NULL) { |
|  | fprintf(stderr, "SERVER: failed to bind socket\n"); |
|  | return 2; |
|  | } |
|  | freeaddrinfo(servinfo); |
|  |  |
|  | printf("SERVER: waiting to recvfrom...\n")  **Summary**  TFTP is a very simple protocol used to transfer files. It is from  this that its name comes, Trivial File Transfer Protocol or TFTP.  Each nonterminal packet is acknowledged separately. This document  describes the protocol and its types of packets. The document also  explains the reasons behind some of the design decisions. |