CS6111-COMPUTER NETWORKS LAB

Ex.NO.2 : SOCKET PROGRAMMING

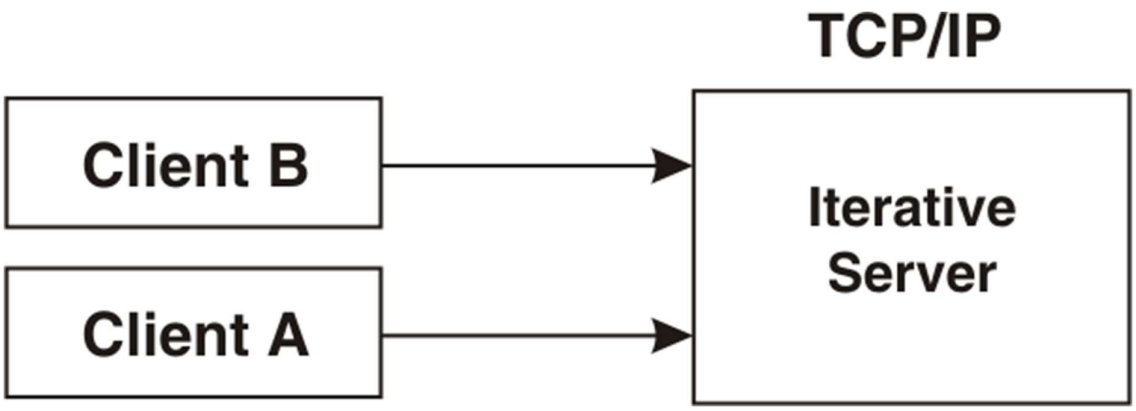
Client Server Programming to handle multiple clients

DATE: 9.08.2024

Part I:

Write an echo program with client and iterative server using TCP.

An ***iterative server*** handles both the connection request and the transaction involved in the call itself. Iterative servers are fairly simple and are suitable for transactions that do not last long.



TCP Echo Client

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>
#include <arpa/inet.h>

#define MAXLINE 4096 /*max text line length*/
#define SERV_PORT 3000 /*port*/

int
main(int argc, char **argv)
```

```
{
 int sockfd;
 struct sockaddr_in servaddr;
 char sendline[MAXLINE], recvline[MAXLINE];

 //basic check of the arguments
 //additional checks can be inserted
 if (argc !=2) {
  perror("Usage: TCPClient <IP address of the server");
  exit(1);
 }

 //Create a socket for the client
 //If sockfd<0 there was an error in the creation of the socket
 if ((sockfd = socket (AF_INET, SOCK_STREAM, 0)) <0) {
  perror("Problem in creating the socket");
  exit(2);
 }

 //Creation of the socket
 memset(&servaddr, 0, sizeof(servaddr));
 servaddr.sin_family = AF_INET;
 servaddr.sin_addr.s_addr= inet_addr(argv[1]);
 servaddr.sin_port =  htons(SERV_PORT); //convert to big-endian order

 //Connection of the client to the socket
 if (connect(sockfd, (struct sockaddr *) &servaddr, sizeof(servaddr))<0) {
  perror("Problem in connecting to the server");
  exit(3);
 }

 while (fgets(sendline, MAXLINE, stdin) != NULL) {

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

  if (recv(sockfd, recvline, MAXLINE,0) == 0){
   //error: server terminated prematurely
   perror("The server terminated prematurely");
   exit(4);
  }
  printf("%s", "String received from the server: ");
  fputs(recvline, stdout);
 }
```

```
 exit(0);
}
```

TCP Iterative Server

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

#define MAXLINE 4096 /*max text line length*/
#define SERV_PORT 3000 /*port*/
#define LISTENQ 8 /*maximum number of client connections */

int main (int argc, char **argv)
{
 int listenfd, connfd, n;
 socklen_t clilen;
 char buf[MAXLINE];
 struct sockaddr_in cliaddr, servaddr;

 //creation of the socket
 listenfd = socket (AF_INET, SOCK_STREAM, 0);

 //preparation of the socket address
 servaddr.sin_family = AF_INET;
 servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
 servaddr.sin_port = htons(SERV_PORT);

 bind (listenfd, (struct sockaddr *) &servaddr, sizeof(servaddr));

 listen (listenfd, LISTENQ);

 printf("%s\n","Server running...waiting for connections.");

 for ( ; ; ) {

  clilen = sizeof(cliaddr);
  connfd = accept (listenfd, (struct sockaddr *) &cliaddr, &clilen);
```

```
 printf("%s\n","Received request...");

 while ( (n = recv(connfd, buf, MAXLINE,0)) > 0)  {
  printf("%s","String received from and resent to the client:");
  puts(buf);
  send(connfd, buf, n, 0);
 }

if (n < 0) {
 perror("Read error");
 exit(1);
}
 close(connfd);

 }
 //close listening socket
 close (listenfd);
}
```

Part II :

2. Need for designing a concurrent server for handling clients using fork() call:

Fork() call *creates multiple child processes for concurrent clients and runs each call block in its own process control block (PCB).*

Through TCP basic server-client model, **one server attends only one client at a particular time**.

But, we are now trying to make our TCP server handle more than one client. Although, we can achieve this using **select() system call** but we can ease the whole process.

How is the fork() system call going to help in this?

Fork() creates a **new child process** that runs in sync with its **Parent process** and returns **0** if child process is created successfully.

- Whenever a new client will attempt to connect to the TCP server, we will create a new **Child Process** that is going to run in parallel with other clients' execution. In this way, we are going to design a concurrent server without using the **Select() system call.**

- A **pid_t (Process id) data type** will be used to hold the Child's process id. Example: **pid_t = fork( )**.

Difference from the other approaches:

This is the simplest technique for creating a concurrent server. Whenever a new client connects to the server, a fork() call is executed making a new child process for each new client.

- **Multi-Threading** achieves a concurrent server using a single processed program. Sharing of data/files with connections is usually slower with a **fork()** than with threads.
- **Select() system call** doesn't create multiple processes. Instead, it helps in **multiplexing** all the clients on a single program and doesn't need **non-blocking IO.**

Program to design a concurrent server for handling multiple clients using fork()

- Accepting a client makes a new child process that runs concurrently with other clients and the parent process
- C

```
// Accept connection request from client in cliAddr
// socket structure
clientSocket = accept(
   sockfd, (struct sockaddr*)&cliAddr, &addr_size);

// Make a child process by fork() and check if child
// process is created successfully
if ((childpid = fork()) == 0) {
   // Send a confirmation message to the client for
   // successful connection
   send(clientSocket, "hi client", strlen("hi client"),
      0);
}
```

- **Server Implementation:**
- C

```
// Server side program that sends
// a 'hi client' message
// to every client concurrently

#include <arpa/inet.h>
#include <netinet/in.h>
#include <stdio.h>
#include <stdlib.h>
```

```
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <unistd.h>

// PORT number
#define PORT 4444

int main()
{
   // Server socket id
   int sockfd, ret;

   // Server socket address structures
   struct sockaddr_in serverAddr;

   // Client socket id
   int clientSocket;

   // Client socket address structures
   struct sockaddr_in cliAddr;

   // Stores byte size of server socket address
   socklen_t addr_size;

   // Child process id
   pid_t childpid;

   // Creates a TCP socket id from IPV4 family
   sockfd = socket(AF_INET, SOCK_STREAM, 0);

   // Error handling if socket id is not valid
   if (sockfd < 0) {
      printf("Error in connection.\n");
      exit(1);
   }

   printf("Server Socket is created.\n");

   // Initializing address structure with NULL
   memset(&serverAddr, '\0',
        sizeof(serverAddr));

   // Assign port number and IP address
   // to the socket created
   serverAddr.sin_family = AF_INET;
```

```
serverAddr.sin_port = htons(PORT);

// 127.0.0.1 is a loopback address
serverAddr.sin_addr.s_addr
   = inet_addr("127.0.0.1");

// Binding the socket id with
// the socket structure
ret = bind(sockfd,
        (struct sockaddr*)&serverAddr,
        sizeof(serverAddr));

// Error handling
if (ret < 0) {
   printf("Error in binding.\n");
   exit(1);
}

// Listening for connections (upto 10)
if (listen(sockfd, 10) == 0) {
   printf("Listening...\n\n");
}

int cnt = 0;
while (1) {

   // Accept clients and
   // store their information in cliAddr
   clientSocket = accept(
      sockfd, (struct sockaddr*)&cliAddr,
      &addr_size);

   // Error handling
   if (clientSocket < 0) {
      exit(1);
   }

   // Displaying information of
   // connected client
   printf("Connection accepted from %s:%d\n",
        inet_ntoa(cliAddr.sin_addr),
        ntohs(cliAddr.sin_port));

   // Print number of clients
   // connected till now
   printf("Clients connected: %d\n\n",
```

```
        ++cnt);

    // Creates a child process
    if ((childpid = fork()) == 0) {

      // Closing the server socket id
      close(sockfd);

      // Send a confirmation message
      // to the client
      send(clientSocket, "hi client",
          strlen("hi client"), 0);
    }
  }

  // Close the client socket id
  close(clientSocket);
  return 0;
}
```

- **Client Implementation:**

- C

```
// Client Side program to test
// the TCP server that returns
// a 'hi client' message

#include <arpa/inet.h>
#include <netinet/in.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <unistd.h>

// PORT number
#define PORT 4444
```

```
int main()
{
   // Socket id
   int clientSocket, ret;

   // Client socket structure
   struct sockaddr_in cliAddr;

   // char array to store incoming message
   char buffer[1024];

   // Creating socket id
   clientSocket = socket(AF_INET,
                         SOCK_STREAM, 0);

   if (clientSocket < 0) {
      printf("Error in connection.\n");
      exit(1);
   }
   printf("Client Socket is created.\n");

   // Initializing socket structure with NULL
   memset(&cliAddr, '\0', sizeof(cliAddr));

   // Initializing buffer array with NULL
   memset(buffer, '\0', sizeof(buffer));

   // Assigning port number and IP address
   serverAddr.sin_family = AF_INET;
   serverAddr.sin_port = htons(PORT);

   // 127.0.0.1 is Loopback IP
   serverAddr.sin_addr.s_addr
      = inet_addr("127.0.0.1");

   // connect() to connect to the server
   ret = connect(clientSocket,
                 (struct sockaddr*)&serverAddr,
                 sizeof(serverAddr));

   if (ret < 0) {
      printf("Error in connection.\n");
```

```
        exit(1);
    }

    printf("Connected to Server.\n");

    while (1) {

        // recv() receives the message
        // from server and stores in buffer
        if (recv(clientSocket, buffer, 1024, 0)
            < 0) {
            printf("Error in receiving data.\n");
        }

        // Printing the message on screen
        else {
            printf("Server: %s\n", buffer);
            bzero(buffer, sizeof(buffer));
        }
    }

    return 0;
}
```

Compile script:

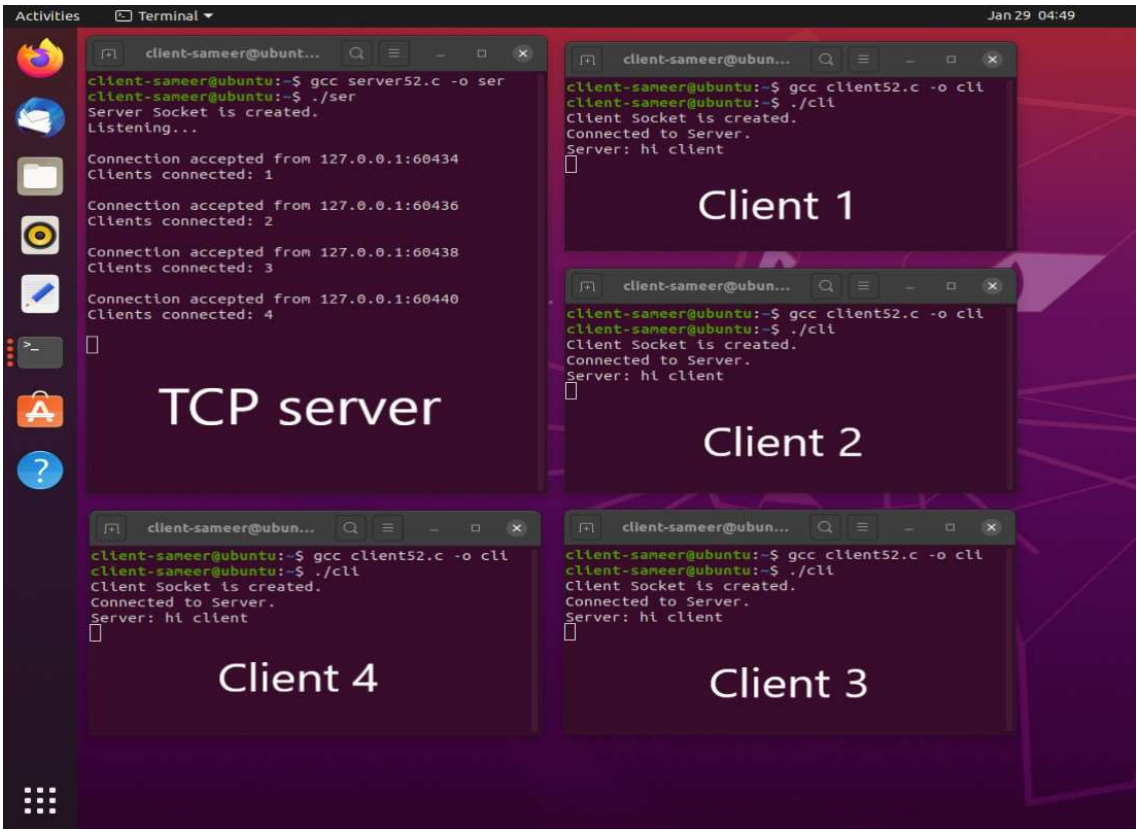
- Executing **server side** code

```
⇒gcc server.c -o ser
 ./ser
```

- Executing **client side** code

```
⇒gcc client.c -o cli
 ./cli
```

Output:



Advantages: The advantages of using this process are:

- Easy to implement in a program doing a far more complex task.
- Each child process (client) runs independently and is unable to read/write other clients' data.
- The server behaves as it has only one client connected to it. The child processes need not care about other incoming connections or the running of parallel child processes. Therefore, programming with fork() system call is transparent and takes less effort.

Disadvantages: The disadvantages are as mentioned here

- Fork is less efficient than multi-threading because it creates a large overhead by creating a new process but a thread is a lightweight process that shares the resources from the parent process itself.

- Operating system will need memory sharing or synchronization cost for achieving concurrency

Part III:Socket Programming in C/C++: Handling multiple clients on server without multi threading

In the basic model, server handles only one client at a time, which is a big assumption if you want to develop any scalable server model. The simple way to handle multiple clients would be to spawn new thread for every new client connected to the server. This method is strongly not recommended because of various disadvantages, namely:

- Threads are difficult to code, debug and sometimes they have unpredictable results.
- Overhead switching of context
- Not scalable for large number of clients
- Deadlocks can occur

Select()

A better way to handle multiple clients is by using

select()

linux command.

-
- Select command allows to monitor multiple file descriptors, waiting until one of the file descriptors become active.
- For example, if there is some data to be read on one of the sockets select will provide that information.
- **Select** works like an interrupt handler, which gets activated as soon as any file descriptor sends any data.

Data structure used for select:

fd_set It contains the list of file descriptors to monitor for some activity. There are four functions associated with fd_set:

```
fd_set readfds;

// Clear an fd_set
FD_ZERO(&readfds);

// Add a descriptor to an fd_set
FD_SET(master_sock, &readfds);
```

```
// Remove a descriptor from an fd_set
FD_CLR(master_sock, &readfds);

//If something happened on the master socket , then its an incoming connection
FD_ISSET(master_sock, &readfds);
```

Activating select:

Please read the man page for select to check all the arguments for select command.

```
activity = select( max_fd + 1 , &readfds , NULL , NULL , NULL);
```

Implementation:

C++C

#

consider adding thread if handling multiple client

simultaneously for sending and reciving data at the same time

```
/*
 structure to encapsulate data of client this make easy to
 passing the argument to new thread;
*/
struct clientDetails{
    int32_t clientfd;  // client file descriptor
    int32_t serverfd;  // server file descriptor
    std::vector<int> clientList; // for storing all the client fd
    clientDetails(void){ // initializing the variable
        this->clientfd=-1;
        this->serverfd=-1;
    }
};

const int port=4277;
const char ip[]="127.0.0.1"; // for local host
//const ip[]="0.0.0.0"; // for allowing all incomming connection from internet
const int backlog=5; // maximum number of connection allowed



int main() {
auto client= new clientDetails();

    client->serverfd= socket(AF_INET, SOCK_STREAM,0); // for tcp connection
```

```
    // error handling
    if (client->serverfd<=0){
        std::cerr<<"socket creation error\n";
      delete client;
        exit(1);
    }else{
        std::cout<<"socket created\n";
    }
     // setting serverFd to allow multiple connection
     int opt=1;
    if (setsockopt(client->serverfd,SOL_SOCKET,SO_REUSEADDR, (char*)&opt, sizeof
opt)<0){
        std::cerr<<"setSocketopt error\n";
     delete client;
        exit(2);
    }

     // setting the server address
     struct sockaddr_in serverAddr;
     serverAddr.sin_family=AF_INET;
     serverAddr.sin_port=htons(port);
     inet_pton(AF_INET, ip, &serverAddr.sin_addr);
     // binding the server address
     if (bind(client->serverfd, (struct sockaddr*)&serverAddr, sizeof(serverAddr))<0){
        std::cerr<<"bind error\n";
        delete client;
        exit(3);
     }else{
         std::cout<<"server binded\n";
     }
     // listening to the port
     if (listen(client->serverfd, backlog)<0){
        std::cerr<<"listen error\n";
        delete client;
        exit(4);
     }else{
          std::cout<<"server is listening\n";
     }

     fd_set readfds;
     size_t  valread;
     int maxfd;
     int sd=0;
     int activity;
     while (true){
       std::cout<<"waiting for activity\n";
       FD_ZERO(&readfds);
```

```
        FD_SET(client->serverfd, &readfds);
        maxfd=client->serverfd;
        // copying the client list to readfds
        // so that we can listen to all the client
        for(auto sd:client->clientList){
            FD_SET(sd, &readfds);
            if (sd>maxfd){
                maxfd=sd;
            }
        }
        //
        if (sd>maxfd){
            maxfd=sd;
        }
        /* using select for listen to multiple client
          select(int nfds, fd_set *restrict readfds, fd_set *restrict writefds,
          fd_set *restrict errorfds, struct timeval *restrict timeout);
        */

        // for more information about select type 'man select' in terminal
        activity=select(maxfd+1, &readfds, NULL, NULL, NULL);
        if (activity<0){
            std::cerr<<"select error\n";
            continue;
        }
        /*
         * if something happen on client->serverfd then it means its
         * new connection request
         */
        if (FD_ISSET(client->serverfd, &readfds)) {
            client->clientfd = accept(client->serverfd, (struct sockaddr *) NULL, NULL);
            if (client->clientfd < 0) {
                std::cerr << "accept error\n";
                continue;
            }
            // adding client to list
            client->clientList.push_back(client->clientfd);
            std::cout << "new client connected\n";
            std::cout << "new connection, socket fd is " << client->clientfd << ", ip is: "
                 << inet_ntoa(serverAddr.sin_addr) << ", port: " <<
ntohs(serverAddr.sin_port) << "\n";

            /*
             * std::thread t1(handleConnection, client);
             * t1.detach();
             *handle the new connection in new thread
             */
```

```
        }
        /*
         * else some io operation on some socket
         */

        // for storing the recive message
        char message[1024];
        for(int i=0;i<client->clientList.size();++i){
include <iostream>  // for cout/cerr
#include <arpa/inet.h>  // for ip inet_pton()
#include <netinet/in.h> // for address
#include <sys/select.h> // for io multiplexing (select)
#include <sys/socket.h> // for socket
#include <unistd.h>  // for close()
#include <vector> // for storing client
/*
            sd=client->clientList[i];
            if (FD_ISSET(sd, &readfds)){
                valread=read(sd, message, 1024);
                //check if client disconnected
                if (valread==0){
                    std::cout<<"client disconnected\n";

                    getpeername(sd, (struct sockaddr*)&serverAddr,
(socklen_t*)&serverAddr);
                    // getpeername name return the address of the client (sd)

                    std::cout<<"host disconnected, ip: "<<inet_ntoa(serverAddr.sin_addr)<<",
port: "<<ntohs(serverAddr.sin_port)<<"\n";
                    close(sd);
                    /* remove the client from the list */
                    client->clientList.erase(client->clientList.begin()+i);
                }else{
                    std::cout<<"message from client: "<<message<<"\n";
                    /*
                     * handle the message in new thread
                     * so that we can listen to other client
                     * in the main thread
                     * std::thread t1(handleMessage, client, message);
                     * // detach the thread so that it can run independently
                     * t1.detach();
                     */
                }
            }
        }
    }
     delete client;
```

```
    return 0;
}
```

Compile the file and run the server. Use telnet to connect the server as a client. Try running on different machines using following command:

telnet localhost 8888

Code Explanation:

- We have created a fd_set variable readfds, which will monitor all the active file descriptors of the clients plus that of the main server listening socket.
- Whenever a new client will connect, master_socket will be activated and a new fd will be open for that client. We will store its fd in our client_list and in the next iteration we will add it to the readfds to monitor for activity from this client.
- Similarly, if an old client sends some data, readfds will be activated and we will check from the list of existing client to see which client has send the data.
- **Alternatives:**
- There are other functions that can perform tasks similar to select. pselect , poll , ppoll

Part IV: Socket Programming in C/C++: Handling multiple clients on server without multi threading

In the basic model, server handles only one client at a time, which is a big assumption if you want to develop any scalable server model. The simple way to handle multiple clients would be to spawn new thread for every new client connected to the server. This method is strongly not recommended because of various disadvantages, namely:

- Threads are difficult to code, debug and sometimes they have unpredictable results.
- Overhead switching of context
- Not scalable for large number of clients
- Deadlocks can occur

Select()

A better way to handle multiple clients is by using

select()

linux command.

-

- Select command allows to monitor multiple file descriptors, waiting until one of the file descriptors become active.
- For example, if there is some data to be read on one of the sockets select will provide that information.
- **Select** works like an interrupt handler, which gets activated as soon as any file descriptor sends any data.

Data structure used for select:

fd_set It contains the list of file descriptors to monitor for some activity. There are four functions associated with fd_set:

```
fd_set readfds;

// Clear an fd_set
FD_ZERO(&readfds);

// Add a descriptor to an fd_set
FD_SET(master_sock, &readfds);

// Remove a descriptor from an fd_set
FD_CLR(master_sock, &readfds);

//If something happened on the master socket , then its an incoming connection
FD_ISSET(master_sock, &readfds);
```

Activating select:

Please read the man page for select to check all the arguments for select command.

```
activity = select( max_fd + 1 , &readfds , NULL , NULL , NULL);
```

Implementation:

C++C

```
#include <iostream>  // for cout/cerr

#include <arpa/inet.h>  // for ip inet_pton()

#include <netinet/in.h> // for address

#include <sys/select.h> // for io multiplexing (select)

#include <sys/socket.h> // for socket

#include <unistd.h>  // for close()

#include <vector> // for storing client

/*
```

```
 consider adding thread if handling multiple client
 simultaneously  for sending and reciving data at the same time

/*
 structure to encapsulate data of client this make easy to
 passing the argument to new thread;
*/
struct clientDetails{
    int32_t clientfd;  // client file descriptor
    int32_t serverfd;  // server file descriptor
    std::vector<int> clientList; // for storing all the client fd
    clientDetails(void){ // initializing the variable
        this->clientfd=-1;
        this->serverfd=-1;
    }
};

const int port=4277;
const char ip[]="127.0.0.1"; // for local host
//const ip[]="0.0.0.0"; // for allowing all incomming connection from internet
const int backlog=5; // maximum number of connection allowed



int main() {
auto client= new clientDetails();

    client->serverfd= socket(AF_INET, SOCK_STREAM,0); // for tcp connection
    // error handling
```

```
    if (client->serverfd<=0){
        std::cerr<<"socket creation error\n";
      delete client;
        exit(1);
    }else{
        std::cout<<"socket created\n";
    }
     // setting serverFd to allow multiple connection
     int opt=1;
    if (setsockopt(client->serverfd,SOL_SOCKET,SO_REUSEADDR, (char*)&opt, sizeof
opt)<0){
        std::cerr<<"setSocketopt error\n";
     delete client;
        exit(2);
    }

     // setting the server address
     struct sockaddr_in serverAddr;
     serverAddr.sin_family=AF_INET;
     serverAddr.sin_port=htons(port);
     inet_pton(AF_INET, ip, &serverAddr.sin_addr);
     // binding the server address
     if (bind(client->serverfd, (struct sockaddr*)&serverAddr, sizeof(serverAddr))<0){
        std::cerr<<"bind error\n";
        delete client;
        exit(3);
     }else{
         std::cout<<"server binded\n";
     }
     // listening to the port
```

```
if (listen(client->serverfd, backlog)<0){
   std::cerr<<"listen error\n";
   delete client;
   exit(4);
}else{
      std::cout<<"server is listening\n";
}


fd_set readfds;
size_t  valread;
int maxfd;
int sd=0;
int activity;
while (true){
  std::cout<<"waiting for activity\n";
  FD_ZERO(&readfds);
  FD_SET(client->serverfd, &readfds);
  maxfd=client->serverfd;
  // copying the client list to readfds
  // so that we can listen to all the client
  for(auto sd:client->clientList){
      FD_SET(sd, &readfds);
      if (sd>maxfd){
          maxfd=sd;
      }
  }
  //
  if (sd>maxfd){
      maxfd=sd;
```

```
        }
        /* using select for listen to multiple client
          select(int nfds, fd_set *restrict readfds, fd_set *restrict writefds,
          fd_set *restrict errorfds, struct timeval *restrict timeout);
        */

        // for more information about select type 'man select' in terminal
        activity=select(maxfd+1, &readfds, NULL, NULL, NULL);
        if (activity<0){
            std::cerr<<"select error\n";
            continue;
        }
        /*
         * if something happen on client->serverfd then it means its
         * new connection request
         */
        if (FD_ISSET(client->serverfd, &readfds)) {
            client->clientfd = accept(client->serverfd, (struct sockaddr *) NULL, NULL);
            if (client->clientfd < 0) {
                std::cerr << "accept error\n";
                continue;
            }
            // adding client to list
            client->clientList.push_back(client->clientfd);
            std::cout << "new client connected\n";
            std::cout << "new connection, socket fd is " << client->clientfd << ", ip is: "
                    << inet_ntoa(serverAddr.sin_addr) << ", port: " <<
ntohs(serverAddr.sin_port) << "\n";

            /*
```

```
            * std::thread t1(handleConnection, client);
            * t1.detach();
            *handle the new connection in new thread
            */
        }
        /*
         * else some io operation on some socket
         */

        // for storing the recive message
        char message[1024];
        for(int i=0;i<client->clientList.size();++i){
            sd=client->clientList[i];
            if (FD_ISSET(sd, &readfds)){
                valread=read(sd, message, 1024);
                //check if client disconnected
                if (valread==0){
                    std::cout<<"client disconnected\n";

                    getpeername(sd, (struct sockaddr*)&serverAddr,
(socklen_t*)&serverAddr);
                    // getpeername name return the address of the client (sd)

                    std::cout<<"host disconnected, ip: "<<inet_ntoa(serverAddr.sin_addr)<<",
port: "<<ntohs(serverAddr.sin_port)<<"\n";
                    close(sd);
                    /* remove the client from the list */
                    client->clientList.erase(client->clientList.begin()+i);
                }else{
                    std::cout<<"message from client: "<<message<<"\n";
```

```
                /*
                 * handle the message in new thread
                 * so that we can listen to other client
                 * in the main thread
                 * std::thread t1(handleMessage, client, message);
                 * // detach the thread so that it can run independently
                 * t1.detach();
                 */
            }
        }
    }
}
    delete client;
    return 0;
}
```

Compile the file and run the server. Use telnet to connect the server as a client. Try running on different machines using following command:

telnet localhost 8888

Code Explanation:

- We have created a fd_set variable readfds, which will monitor all the active file descriptors of the clients plus that of the main server listening socket.
- Whenever a new client will connect, master_socket will be activated and a new fd will be open for that client. We will store its fd in our client_list and in the next iteration we will add it to the readfds to monitor for activity from this client.
- Similarly, if an old client sends some data, readfds will be activated and we will check from the list of existing client to see which client has send the data.
- **Alternatives:**
- There are other functions that can perform tasks similar to select. pselect , poll , ppoll

Part IV : Handling Multiple Clients using **Multi Threading:**

The primary intention of writing this article is to give you an overview of how we can entertain multiple client requests to a server in parallel. For example, you are going to create a TCP/IP server which can receive multiple client requests at the same time and entertain each client request in parallel so that no client will have to wait for server time. Normally, you will get lots of examples of TCP/IP servers and client examples online which are not capable of processing multiple client requests in parallel.

In the first example, the TCP/IP server has been designed with multi-threading for parallel processing and in the second example, I have implemented the server with multi-processing to accomplish the same goal.

Below is the server application (in C) with a function called socketThread, which is a thread function. Whenever a request comes to the server, the server's main thread will create a thread and pass the client request to that thread with its ID. The thread will start processing with the client request, generate the report, and send it back to the client. As this is just an example, you need to put your own business logic in the thread function. I have just put a sleep statement here and am sending a hard-coded reply from the server to the client. This is just a very simple example and not a professional based TCP/IP server.

The second program is a sample client to test this server. Both of these programs are for Unix/Linux environments only.

socket_server.c

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<string.h>
#include <arpa/inet.h>
#include <fcntl.h> // for open
#include <unistd.h> // for close
#include<pthread.h>

char client_message[2000];
```

```
char buffer[1024];
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

void * socketThread(void *arg)
{
  int newSocket = *((int *)arg);
  recv(newSocket , client_message , 2000 , 0);

  // Send message to the client socket
  pthread_mutex_lock(&lock);
  char *message = malloc(sizeof(client_message)+20);
  strcpy(message,"Hello Client : ");
  strcat(message,client_message);
  strcat(message,"\n");
  strcpy(buffer,message);
  free(message);
  pthread_mutex_unlock(&lock);
  sleep(1);
  send(newSocket,buffer,13,0);
  printf("Exit socketThread \n");
  close(newSocket);
  pthread_exit(NULL);
}

int main(){
  int serverSocket, newSocket;
  struct sockaddr_in serverAddr;
  struct sockaddr_storage serverStorage;
  socklen_t addr_size;
```

```
//Create the socket.
serverSocket = socket(PF_INET, SOCK_STREAM, 0);

// Configure settings of the server address struct
// Address family = Internet
serverAddr.sin_family = AF_INET;

//Set port number, using htons function to use proper byte order
serverAddr.sin_port = htons(7799);

//Set IP address to localhost
serverAddr.sin_addr.s_addr = inet_addr("127.0.0.1");



//Set all bits of the padding field to 0
memset(serverAddr.sin_zero, '\0', sizeof serverAddr.sin_zero);

//Bind the address struct to the socket
bind(serverSocket, (struct sockaddr *) &serverAddr, sizeof(serverAddr));

//Listen on the socket, with 40 max connection requests queued
if(listen(serverSocket,50)==0)
 printf("Listening\n");
else
 printf("Error\n");
 pthread_t tid[60];
 int i = 0;
 while(1)
```

```
  {
     //Accept call creates a new socket for the incoming connection
     addr_size = sizeof serverStorage;
     newSocket = accept(serverSocket, (struct sockaddr *) &serverStorage, &addr_size);

     //for each client request creates a thread and assign the client request to it to
process
    //so the main thread can entertain next request
     if( pthread_create(&tid[i++], NULL, socketThread, &newSocket) != 0 )
       printf("Failed to create thread\n");

     if( i >= 50)
     {
      i = 0;
      while(i < 50)
      {
       pthread_join(tid[i++],NULL);
      }
      i = 0;
     }
  }
 return 0;
}
```

Socket Client.c

This is an example of a simple multithreaded client for testing with 50 parallel requests to the server. If you want to test the client from a different machine, change the localhost to the actual server host and port number.

```
#include <stdio.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>
#include <arpa/inet.h>
#include <stdlib.h>
#include <fcntl.h> // for open
#include <unistd.h> // for close
#include<pthread.h>

void * cientThread(void *arg)
{
  printf("In thread\n");
  char message[1000];
  char buffer[1024];
  int clientSocket;
  struct sockaddr_in serverAddr;
  socklen_t addr_size;

  // Create the socket.
  clientSocket = socket(PF_INET, SOCK_STREAM, 0);

  //Configure settings of the server address
 // Address family is Internet
  serverAddr.sin_family = AF_INET;

  //Set port number, using htons function
  serverAddr.sin_port = htons(7799);
```

```
 //Set IP address to localhost
  serverAddr.sin_addr.s_addr = inet_addr("localhost");
  memset(serverAddr.sin_zero, '\0', sizeof serverAddr.sin_zero);

    //Connect the socket to the server using the address
    addr_size = sizeof serverAddr;
    connect(clientSocket, (struct sockaddr *) &serverAddr, addr_size);
    strcpy(message,"Hello");

   if( send(clientSocket , message , strlen(message) , 0) < 0)
    {
          printf("Send failed\n");
    }

    //Read the message from the server into the buffer
    if(recv(clientSocket, buffer, 1024, 0) < 0)
    {
      printf("Receive failed\n");
    }
    //Print the received message
    printf("Data received: %s\n",buffer);
    close(clientSocket);
    pthread_exit(NULL);
}
int main(){
  int i = 0;
  pthread_t tid[51];
  while(i< 50)
  {
```

```
  if( pthread_create(&tid[i], NULL, cientThread, NULL) != 0 )
       printf("Failed to create thread\n");
  i++;
 }
 sleep(20);
 i = 0;
 while(i< 50)
 {
   pthread_join(tid[i++],NULL);
   printf("%d:\n",i);
 }
 return 0;
}
```

Compile both the client and the server in Linux or in Unix like below:

- cc socket_client.c -o client -lsocket -lnsl
- cc socket_server.c -o server -lsocket –lnsl

First, run the server and then run the client from a different terminal (better to run both from different machines). When you run the client from a different Linux/Unix server, please consider the firewall issues.

----------X_----------