Name : Gourav Kumar Shaw

Enrollment Id. : 2020CSB010

Section: Gx

Subject : Computer Network Lab (CS 3272)

Assignment 3: Application development using TCP Socket

(a) Develop a simple TCP Server Client application where the Client sends text messages to the Server in a user-defined know port (of your choice). On reception of that message, the Server forwards the same message to the Client. Both Server and Client print the message.

Answer:

Code:

server.c

```c
// 2020CSB010 GOURAV KUMAR SHAW

#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <pthread.h>

#define PORT 12345
#define BUFFER_SIZE 1024

void *connection_handler(void *);
```

```c
int main(int argc, char *argv[])
{
    int socket_desc, client_sock, c, *new_sock;
    struct sockaddr_in server, client;

    socket_desc = socket(AF_INET, SOCK_STREAM, 0);// The first argument, AF_INET, specifies that the socket should use the IPv4 protocol. The second argument, SOCK_STREAM, specifies that the socket should use the TCP protocol, which provides a reliable, stream-oriented connection. The third argument, 0, specifies a default protocol to be used
    if (socket_desc == -1)
    {
        perror("Could not create socket");
        return 1;
    }

    server.sin_family = AF_INET;
    server.sin_addr.s_addr = INADDR_ANY;
    server.sin_port = htons(PORT); // unsigned short int htons(unsigned short inthostshort);

    if (bind(socket_desc, (struct sockaddr *)&server, sizeof(server)) < 0)
    {
        perror("Bind failed");
        return 1;
    }

    listen(socket_desc, 3); // int listen(int sockfd, int backlog); here 3 is backlog means The backlog, defines the maximum length to which the queue of pending connections for sockfd may grow.

    printf("Waiting for incoming connections...\n");
    c = sizeof(struct sockaddr_in);

    while ((client_sock = accept(socket_desc, (struct sockaddr *)&client, (socklen_t *)&c)))
    {
        printf("Connection accepted\n");

        pthread_t sniffer_thread;
        new_sock = malloc(1);
        *new_sock = client_sock;
        // 1.When a client connects, the accept function returns a new socket descriptor for communication with the client. The client structure is filled with the client's address information.
        // 2.For each new connection, a new thread is created using pthread_create. The thread function connection_handler is passed the new_sock
```

```
parameter, which is a pointer to the newly created socket descriptor for
communication with the client.
        // 3.The malloc function is used to allocate memory for the new_sock
variable. This memory is then passed to the thread function as a parameter.
        // 4.The main thread continues to wait for new connections and repeat
the process of creating a new thread for each connection.
        if (pthread_create(&sniffer_thread, NULL, connection_handler, (void
*)new_sock) < 0)
        {
            perror("Could not create thread");
            return 1;
        }

        printf("Handler assigned\n");
    }

    if (client_sock < 0)
    {
        perror("Accept failed");
        return 1;
    }

    return 0;
}

void *connection_handler(void *socket_desc)
{
    int sock = *(int *)socket_desc;
    int read_size;
    char message[BUFFER_SIZE];

    while ((read_size = recv(sock, message, BUFFER_SIZE, 0)) > 0)
    {
        printf("Server received message: %s\n", message);

        if (send(sock, message, strlen(message), 0) < 0)
        {
            perror("Send failed");
            return 0;
        }
    }

    if (read_size == 0)
    {
        printf("Client disconnected\n");
        fflush(stdout);
    }
    else if (read_size == -1)
```

```
    {
        perror("Recv failed");
    }

    free(socket_desc);

    return 0;
}
```

client.c

```
// 2020CSB010 GOURAV KUMAR SHAW

#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>

#define PORT 12345
#define BUFFER_SIZE 1024

int main(int argc, char *argv[]) {
    int sock;
    struct sockaddr_in server;
    char message[BUFFER_SIZE];

    sock = socket(AF_INET, SOCK_STREAM, 0);// sock_stream means TCP , 0 is IP
(int sockfd = socket(domain, type, protocol))
    if (sock == -1) {
        perror("Could not create socket");
        return 1;
    }

    server.sin_addr.s_addr = inet_addr("127.0.0.1");// local address
    server.sin_family = AF_INET;// AF_INET FOR IPv4
    server.sin_port = htons(PORT);// unsigned short int htons(unsigned short
int hostshort);//host byte order to network byte order

    if (connect(sock, (struct sockaddr *)&server, sizeof(server)) < 0) {
        perror("Connect failed");
        return 1;
    }
```

```c
    printf("Enter message: ");
    scanf("%s", message);

    if (send(sock, message, strlen(message), 0) < 0) {
        perror("Send failed");
        return 1;
    }



    if (recv(sock, message, BUFFER_SIZE, 0) < 0) {
        perror("Recv failed");
        return 1;
    }

    printf("Client received message: %s\n", message);

    close(sock);

    return 0;
}
```

Output:

```
gourav  LAPTOP-868QQ3N0  ../Assignment_3_networking  master   ./server_1
Waiting for incoming connections...
Connection accepted
Handler assigned
Server received message: hello
Client disconnected
```

```
gourav  LAPTOP-868QQ3N0  ../Assignment_3_networking  master   ./client_1
Enter message: hello
Client received message: hello
```

(b) Develop a simple TCP Server and Client application to perform chat.

Answer:

Code:

server.c

```c
// 2020CSB010 GOURAV KUMAR SHAW

#include <stdio.h>
#include <netdb.h>
#include <netinet/in.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <pthread.h>

#define MAX 80 // message size
#define PORT 8080
#define SA struct sockaddr

void *receive_thread_function(void *arg)
{
    int sockfd = *(int *)arg;

    char buff[MAX];
    int n;
    while (1)
    {
        bzero(buff, MAX);

        read(sockfd, buff, sizeof(buff)); // buff is storing the message
        printf("From client: %s\n", buff);
        if (strncmp("exit", buff, 4) == 0)
        {
            printf("Server Exit...\n");
            break;
        }
    }
}

void *send_thread_function(void *arg)
{
    int sockfd = *(int *)arg;
```

```
    char buff[MAX];
    int n;
    while (1)
    {
        bzero(buff, MAX);
        n = 0;
        while ((buff[n++] = getchar()) != '\n')
            ;

        write(sockfd, buff, sizeof(buff));
        if (strncmp("exit", buff, 4) == 0)
        {
            printf("Server Exit...\n");
            break;
        }
    }
}

int main()
{
    int sockfd, connfd, len;
    struct sockaddr_in servaddr, cli;
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd == -1)
    {
        printf("socket creation failed...\n");
        exit(0);
    }
    else
        printf("Socket successfully created..\n");
    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(PORT);
    /*
    This line of code binds the created socket to the specified IP address and
port number using the bind function. sockfd is the socket file descriptor
returned by the socket function. SA is a type defined as a struct sockaddr,
which is used for socket addresses. servaddr is an instance of the sockaddr_in
structure, which contains the IP address and port number that the server is
binding to. The sizeof operator is used to calculate the size of the servaddr
structure.

    The bind function returns 0 if the binding is successful, and -1 if an
error occurs. If the binding fails, the program will print an error message
and exit.
    */
```

```
    if ((bind(sockfd, (SA *)&servaddr, sizeof(servaddr))) != 0)
    {
        printf("socket bind failed...\n");
        exit(0);
    }
    else
        printf("Socket successfully bound..\n");
        /*
        This line of code starts the listening process on the server socket
created earlier, using the listen function. The listen function puts the
server socket in a passive mode, waiting for client connections.

        sockfd is the socket file descriptor returned by the socket function.
The second argument, 5, specifies the maximum length to which the queue of
pending connections for this socket may grow. This means that the server will
be able to handle up to 5 client connections simultaneously.

        The listen function returns 0 if successful, and -1 if an error
occurs. If an error occurs, the program will print an error message and exit.
        */
    if ((listen(sockfd, 5)) != 0)
    {
        printf("Listen failed...\n");
        exit(0);
    }
    else
        printf("Server listening..\n");
        /*
        This line of code sets the size of the cli structure, which is a
sockaddr_in structure containing the client's address and port number.

        The sizeof(cli) returns the size of the cli structure in bytes, which
is passed to the accept function as the size of the cli structure. The accept
function then populates the cli structure with the client's address and port
number information.

        It is important to note that the len variable is passed to the accept
function as a pointer, so that the function can update its value with the
actual size of the cli structure. This is necessary because the size of the
cli structure may vary depending on the type of the address family being used
(IPv4 or IPv6).
                */
    len = sizeof(cli);
    /*
    code accepts a connection from a client and creates a new socket for
communication with the client. The accept function blocks the server process
until a client connects to the server.
```

```
    sockfd is the server socket file descriptor returned by the socket
function. cli is a pointer to a sockaddr structure that will hold the client's
address and port number. len is a pointer to a variable that holds the size of
the sockaddr structure.

    The accept function returns a new file descriptor representing the client
socket. This file descriptor is used for subsequent communication with the
client. If an error occurs, the function returns -1.
        */
    connfd = accept(sockfd, (SA *)&cli, &len);
    if (connfd < 0)
    {
        printf("server acccept failed...\n");
        exit(0);
    }
    else
        printf("server acccept the client...\n");
    pthread_t receive_tid, send_tid;
    /*
    This line of code declares two variables of type pthread_t: receive_tid
and send_tid. pthread_t is a data type in the pthreads library that represents
a thread ID.

    The variables are used to store the thread IDs returned by the
pthread_create function when it creates two new threads. The first thread is
responsible for receiving messages from the client, and the second thread is
responsible for sending messages to the client. By storing the thread IDs in
variables, the main thread can later use these IDs to join the threads and
wait for their completion.
        */
    int *p = malloc(sizeof(int));
    *p = connfd;
    pthread_create(&receive_tid, NULL, receive_thread_function, p);
    pthread_create(&send_tid, NULL, send_thread_function, p);
    pthread_join(receive_tid, NULL);
    pthread_join(send_tid, NULL);
    /*
    The code above is creating two threads, receive_thread_function and
send_thread_function, passing the socket file descriptor connfd as an
argument, and then waiting for the threads to finish using pthread_join().

    malloc() is used to allocate memory dynamically to store the integer value
of connfd, which is then passed as a void pointer to the threads.

    pthread_create() is used to create a new thread. It takes four arguments:
a pointer to a pthread_t object to store the ID of the newly created thread, a
pthread_attr_t object that specifies various attributes of the thread (e.g.,
its scheduling policy), a pointer to the function that the thread should run,
```

```
and a void pointer to any arguments that should be passed to the thread
function.

    After both threads have been created, pthread_join() is used to wait for
them to finish. pthread_join() takes two arguments: the pthread_t object for
the thread to wait for, and a pointer to a location where the exit status of
the thread should be stored (if the exit status is not needed, this argument
can be NULL).
    */
    close(sockfd);
}
```

client.c

```
// 2020CSB010 GOURAV KUMAR SHAW

#include <stdio.h>
#include <netdb.h>
#include <netinet/in.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <pthread.h>

#define MAX 80
#define PORT 8080
#define SA struct sockaddr

void *receive_thread_function(void *arg)
{
    int sockfd = *(int *)arg;

    char buff[MAX];
    int n;
    while (1)
    {
        bzero(buff, MAX);

        read(sockfd, buff, sizeof(buff));
        printf("From server: %s\n", buff);
        if (strncmp("exit", buff, 4) == 0)
        {
            printf("Client Exit...\n");
            break;
        }
```

```
    }
}

void *send_thread_function(void *arg)
{
    int sockfd = *(int *)arg;

    char buff[MAX];
    int n;
    while (1)
    {
        bzero(buff, MAX);
        n = 0;
        while ((buff[n++] = getchar()) != '\n')
            ;

        write(sockfd, buff, sizeof(buff));
        if (strncmp("exit", buff, 4) == 0)
        {
            printf("Client Exit...\n");
            break;
        }
    }
}

int main()
{
    int sockfd, connfd;
    struct sockaddr_in servaddr, cli;

    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd == -1)
    {
        printf("socket creation failed...\n");
        exit(0);
    }
    else
        printf("Socket successfully created..\n");

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = inet_addr("127.0.0.1");
    servaddr.sin_port = htons(PORT);

    if (connect(sockfd, (SA *)&servaddr, sizeof(servaddr)) != 0)
    {
        printf("connection with the server failed...\n");
        exit(0);
```

```
    }
    else
        printf("connected to the server..\n");

    pthread_t receive_tid, send_tid;
    int *p = malloc(sizeof(int));
    *p = sockfd;
    pthread_create(&receive_tid, NULL, receive_thread_function, p);
    pthread_create(&send_tid, NULL, send_thread_function, p);
    pthread_join(receive_tid, NULL);
    pthread_join(send_tid, NULL);
    close(sockfd);
}
```

Output:

```
gourav  LAPTOP-868QQ3N0  ../Assignment_3_networking  master   ./server_2
Socket successfully created..
Socket successfully bound..
Server listening..
server acccept the client...
From client: Hello

From client: how

From client: are you

From client: Gourav

I am fine
Thank you!!
From client: Today is New Year

wow!! that's great !!
From client: okay than bye we will meet at the party.

okay bye!
```

```
gourav  LAPTOP-868QQ3N0  ../Assignment_3_networking  master   ./client_2
Socket successfully created..
connected to the server..
Hello
how
are you
Gourav
From server: I am fine

From server: Thank you!!

Today is New Year
From server: wow!! that's great !!

okay than bye we will meet at the party.
From server: okay bye!
```

(c) Develop a TCP chat Server application where multiple Clients participate. For example, if there are four clients, Client #1 has conversations with Client #2, and Client #3 has conversations with Client #4. The Server does the job of forwarding the message to the intended Client.

Answer:

code:

server.cpp

```
// 2020CSB010 GOURAV KUMAR SHAW

#include <stdlib.h>
#include <signal.h>
#include <pthread.h>
#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <map>
#include <cstring>
#include <iostream>
#include <unistd.h>

using namespace std;

#define PORT 8080
#define MAX_CLIENTS 5

map<int, char *> m;                 // socket file descriptor and name
map<int, pthread_t> client_threads; // client socket file descriptor and
thread
int server_socket;
/*
    m: a map that associates a client socket file descriptor with the name of
the client.
    client_threads: a map that associates a client socket file descriptor with
the thread that will be used to handle messages to/from that client.
    server_socket: the socket file descriptor that the server will use to
listen for connections from clients.
*/

void *sendingMessage(void *arg)
{
```

```
    int client_socket = *(int *)arg;
    /*
    Assuming arg was passed as a void pointer to a function that is creating a
new thread, this code is extracting the client socket file descriptor from the
arg pointer, which was passed to the thread function as an argument. The
client socket file descriptor is used to communicate with the client over the
network.
    */
    char sender[1024];
    strcpy(sender, m[client_socket]);
    char destination[1024];
    char message[1024];
    while (1)
    {
        bool flag = false;
        /*
        The first argument, client_socket, is a file descriptor for a socket
that has been created and connected to a remote host. This socket is used for
receiving data from the remote host.

        The second argument, destination, is a pointer to a buffer that will
hold the received data. This buffer must be large enough to hold the maximum
amount of data that may be received in a single call to recv(). In this case,
the buffer size is set to 1024 bytes.

        The third argument, 1024, is the maximum amount of data that recv()
will attempt to receive in a single call. This value should not exceed the
size of the buffer pointed to by the destination argument.

        The fourth argument, 0, specifies the flags to be used when receiving
the data. In this case, no special flags are set.
                */
        int destSize = recv(client_socket, destination, 1024, 0);
        if (destSize > 0)
        {
            int messageSize = recv(client_socket, message, 1024, 0); //
message size
            for (auto it : m)
            {
                if (messageSize > 0)
                {
                    if (!strcmp(message, "exit"))
                    {
                        strcpy(m[client_socket], "\0");
                        if (close(client_socket) == 0)
                            cout << "Client Socket Closed!"
                                 << endl;
                    }
```

```
                    else if (!strcmp(it.second, destination))
                    {
                        flag = true;
                        if (messageSize > 0)
                        {
                            strcat(message, " from ");
                            strcat(message, m[client_socket]);
                            send(it.first, message, 1024, 0);
                        }
                        break;
                    }
                }
            }
            if (!flag)
            {
                send(client_socket, "Offline!", 1024, 0);
            }
        }
    }
    return NULL;
}
void *close(void *arg)
{
    int server_fd = *(int *)arg;
    char closemsg[1024];
    while (1)
    {
        cout << "Enter (close) to close the server" << endl;
        cin >> closemsg;
        if (!strcmp(closemsg, "close"))
        {
            for (auto i : m)
            {
                write(i.first, "server down", 12);
                if (close(i.first) == 0) // "close(i.first) == 0" is a
comparison that evaluates to true if the "close" system call on the file
descriptor represented by "i.first" returns 0, indicating that the file
descriptor was successfully closed.
                {
                    cout << "Client Socket of  " << i.second << " closed" <<
endl;
                }
            }
            close(server_fd);
            exit(0);
        }
    }
```

```
    return NULL;
}

int main()
{
    // Here Creating Socket
    server_socket = socket(AF_INET, SOCK_STREAM, 0);
    if (server_socket == -1)
    {
        cout << "Failed to Create Socket!" << endl;
        exit(EXIT_FAILURE);
    }

    // initializing sockaddr_in structure
    struct sockaddr_in server;
    server.sin_family = AF_INET;
    server.sin_port = htons(PORT);
    server.sin_addr.s_addr = INADDR_ANY;

    // bind
    if (bind(server_socket, (struct sockaddr *)&server, sizeof(server)) == -1)
    {
        cout << "Failed to Bind!" << endl;
        close(server_socket);
        exit(EXIT_FAILURE);
    }

    // listening
    if (listen(server_socket, MAX_CLIENTS) == -1)
    {
        cout << "Failed to Listen!" << endl;
        close(server_socket);
        exit(EXIT_FAILURE);
    }
    else
    {
        cout << "Server is Listening on Port " << PORT << endl;
    }

    pthread_t exit_thread;
    pthread_create(&exit_thread, NULL, close, &server_socket);

    while (1)
    {
        struct sockaddr_in client;
        socklen_t len = sizeof(client);
```

```
        int client_socket = accept(server_socket, (struct sockaddr *)&client,
&len);
        if (client_socket == -1)
        {
            cout << "Client Failed to Connect!!" << endl;
            close(server_socket);
            exit(EXIT_FAILURE);
        }
        char name[1024];
        int nameSize = recv(client_socket, name, 1024, 0);
        if (nameSize > 0)
        {
            m[client_socket] = (char *)malloc(strlen(name) * sizeof(char));
            strcpy(m[client_socket], name);
            cout << "Connection Successful with Client : " << m[client_socket]
<< " !" << endl;

            pthread_create(&client_threads[client_socket], NULL,
&sendingMessage, &client_socket);
        }
        /*
        The "if (nameSize > 0)" condition checks if the size of the client
name (in bytes) is greater than zero. If so, it proceeds with creating a new
thread to handle the client connection.

        The line "m[client_socket] = (char *)malloc(strlen(name) *
sizeof(char));" allocates memory for a new character array in the "m" array at
the index "client_socket". The size of the array is determined by the length
of the "name" string (excluding the null terminator) multiplied by the size of
a single character (in bytes). The "(char *)" typecast is used to convert the
void pointer returned by "malloc" into a char pointer that can be assigned to
the "m" array.

        The line "strcpy(m[client_socket], name);" copies the contents of the
"name" string (including the null terminator) into the newly allocated
character array at the index "client_socket" of the "m" array. This allows the
server to keep track of the name of the client associated with each socket.

        The line "cout << "Connection Successful with Client : " <<
m[client_socket] << " !" << endl;" outputs a message to the console indicating
that a connection has been established with the client associated with the
current socket.

        The line "pthread_create(&client_threads[client_socket], NULL,
&sendingMessage, &client_socket);" creates a new thread using the
"pthread_create" function. The thread is created with the "sendingMessage"
function as the entry point and the current socket index as the argument. This
allows the new thread to handle messages sent by the client on this socket
```

```
while the main thread continues to listen for new connections on other
sockets.
        */
    }
    return 0;
}
```

client.cpp

```
// 2020CSB010 GOURAV KUMAR SHAW

#include <pthread.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <iostream>
#include <cstring>
#include <signal.h>
#include <unistd.h>


#define PORT 8080
using namespace std;

int client_socket;
void *SendingMessage(void *arg)
{
    pthread_t thread2 = *(pthread_t *)arg;
    while (1)
    {
        char message1[1024];
        cin >> message1;
        if (sizeof(message1) / sizeof(message1[0]) > 0)
        {
            if (send(client_socket, message1, sizeof(message1) /
sizeof(message1[0]), 0) == -1)
            {
                cout << "Failed to Send!" << endl;
            }
        }
        if (!strcmp(message1, "exit"))
        {
            if (close(client_socket) == 0)
                cout << "Client Socket Closed!"
                     << endl;
            if (pthread_cancel(thread2) == 0)
                cout << "Thread 2 Cancelled!" << endl;
```

```
            pthread_exit(NULL);
        }
    }
    return NULL;
}

void *receivingMessage(void *arg)
{
    pthread_t thread1 = *(pthread_t *)arg;
    while (1)
    {
        char message2[1024];
        int msgSize = recv(client_socket, message2, 1024, 0);
        if (msgSize > 0)
        {
            if (!strcmp(message2, "server down"))
            {
                if (close(client_socket) == 0)
                    cout << "Client Socket Closed!"
                         << endl;
                if (pthread_cancel(thread1) == 0)
                    cout << "Thread 1 Cancelled!" << endl;
                pthread_exit(NULL);
            }
            cout << message2 << endl;
        }
    }
    return NULL;
}

int main()
{
    // create socket
    client_socket = socket(AF_INET, SOCK_STREAM, 0);
    if (client_socket == -1)
    {
        cout << "Failed to Create Socket!" << endl;
        exit(EXIT_FAILURE);
    }

    // initialize  sockaddr_in structure
    struct sockaddr_in server_address;
    server_address.sin_family = AF_INET;
    server_address.sin_port = htons(PORT);
    server_address.sin_addr.s_addr = INADDR_ANY;

    // connect to server
```

```
    if (connect(client_socket, (struct sockaddr *)&server_address,
sizeof(server_address)) == -1)
    {
        cout << "Error in Connecting to Server!" << endl;
        close(client_socket);
        exit(EXIT_FAILURE);
    }
    else
    {
        cout << "Client Successfully Connected to Server!" << endl;
    }
    pthread_t thread1, thread2;
    pthread_create(&thread1, NULL, SendingMessage, &thread2);
    pthread_create(&thread2, NULL, receivingMessage, &thread1);
    pthread_join(thread1, NULL);
    cout << "Thread 1 Closed!" << endl;
    pthread_join(thread2, NULL);
    cout << "Thread 2 Closed!" << endl;
    return 0;
}
```

Output: