Programming TCP sockets in C

This lab is based on Transmission Control Protocol (TCP) Socket programming. Unlike UDP, TCP is a connection-oriented protocol. This means that before the client and server can start to send data to each other, they first need to handshake and establish a TCP connection. One end of the TCP connection is attached to the client socket and the other end is attached to a server socket. When creating the TCP connection, we associate with it the client socket address (IP address and port number) and the server socket address (IP address and port number). With the TCP connection established, when one side wants to send data to the other side, it just drops the data into the TCP connection via its socket. This is different from UDP, for which the server must attach a destination address to the packet before dropping it into the socket. TCP sockets are used for reliable communication between client and server. They have error control mechanism such that if error occur, protocol detects the error and compensate it by resending failed packets.

The following code has been taken from Silver Moon's article published on July 30, 2012 on web page with URL: https://www.binarytides.com/server-client-example-c-sockets-linux/. This code example will start a server on localhost (127.0.0.1) port 8888. Once it receives a connection, it will read input from the client and reply back with the same message.

Server

Let's build a very simple TCP server. There is almost a standard sequence of calls that are needed to build a TCP server program. The steps to make a TCP server are as follows:

1. Create socket
2. Bind to address and port
3. Put in listening mode
4. Accept connections and process thereafter.

```
/*
    C socket server example
*/
#include<stdio.h>
#include<string.h>
#include<sys/socket.h>
#include<arpa/inet.h>
#include<unistd.h>

int main(void)
{
    int socket_desc , client_sock , c , read_size;
    struct sockaddr_in server , client;
    char client_message[2000];
```

```
//Create socket
socket_desc = socket(AF_INET , SOCK_STREAM , 0);
if (socket_desc == -1)
{
    printf("Could not create socket");
}
puts("Socket created");

//Prepare the sockaddr_in structure
server.sin_family = AF_INET;
server.sin_addr.s_addr = INADDR_ANY;
server.sin_port = htons( 8888 );

//Bind
if( bind(socket_desc,(struct sockaddr *)&server , sizeof(server)) < 0)
{
    //print the error message
    perror("bind failed. Error");
    return 1;
}
puts("bind done");

//Listen
listen(socket_desc , 3);

//Accept and incoming connection
puts("Waiting for incoming connections...");
c = sizeof(struct sockaddr_in);

//accept connection from an incoming client
client_sock = accept(socket_desc, (struct sockaddr *)&client, (socklen_t*)&c);
if (client_sock < 0)
{
    perror("accept failed");
    return 1;
}
puts("Connection accepted");

//Receive a message from client
while((read_size = recv(client_sock , client_message , 2000 , 0)) > 0 )
{
    //Send the message back to client
    write(client_sock , client_message , strlen(client_message));
}
if(read_size == 0)
{
    puts("Client disconnected");
    fflush(stdout);
```

```
    }
    else if(read_size == -1)
    {
        perror("recv failed");
    }
    close(client_sock);
    return 0;
}
```

Client

Just like the building the TCP server, there is almost a standard set of calls and sequence to develop a TCP client.

1. Create socket
2. Connect to TCP server
3. send()/recv()
4. close socket

```
/*
    C ECHO client example using sockets
*/
#include<stdio.h>
#include<string.h>
#include<sys/socket.h>
#include<arpa/inet.h>
#include<unistd.h>

int main(void)
{
    int sock;
    struct sockaddr_in server;
    char message[1000] , server_reply[2000];

    //Create socket
    sock = socket(AF_INET , SOCK_STREAM , 0);
    if (sock == -1)
    {
        printf("Could not create socket");
    }
    puts("Socket created");

    server.sin_addr.s_addr = inet_addr("127.0.0.1");
    server.sin_family = AF_INET;
    server.sin_port = htons(8888);
```

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

    puts("Connected\n");


    //keep communicating with server
    while(1)
    {
        printf("Enter message : ");
        gets(message);

        //Send some data
        if(send(sock , message , strlen(message) , 0) < 0)
        {
            puts("Send failed");
            return 1;
        }

        //Receive a reply from the server
        if(recv(sock , server_reply , 2000 , 0) < 0)
        {
            puts("recv failed");
            break;
        }

        puts("Server reply :");
        puts(server_reply);
    }

    close(sock);
    return 0;
}
```

Server to handle multiple connections

The TCP server in the above example has a drawback. It can handle communication with only one client which is not very useful. One way to work around this is by using threads. A thread can be assigned for each connected client which will handle communication with the client.

```
/*
    C socket server example, handles multiple clients using threads
*/

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

//the thread function
void *connection_handler(void *);

int main(void)
{
    int socket_desc , client_sock , c , *new_sock;
    struct sockaddr_in server , client;

    //Create socket
    socket_desc = socket(AF_INET , SOCK_STREAM , 0);
    if (socket_desc == -1)
    {
        printf("Could not create socket");
    }
    puts("Socket created");

    //Prepare the sockaddr_in structure
    server.sin_family = AF_INET;
    server.sin_addr.s_addr = INADDR_ANY;
    server.sin_port = htons( 8888 );

    //Bind
    if( bind(socket_desc,(struct sockaddr *)&server , sizeof(server)) < 0)
    {
        //print the error message
        perror("bind failed. Error");
        return 1;
    }
    puts("bind done");

    //Listen
    listen(socket_desc , 3);

    //Accept and incoming connection
    puts("Waiting for incoming connections...");
```

```
    c = sizeof(struct sockaddr_in);
    while( (client_sock = accept(socket_desc, (struct sockaddr *)&client,
(socklen_t*)&c)) )
    {
        puts("Connection accepted");

        pthread_t sniffer_thread;
        new_sock = malloc(1);
        *new_sock = client_sock;

        if( pthread_create( &sniffer_thread , NULL ,  connection_handler ,
(void*) new_sock) < 0)
        {
            perror("could not create thread");
            return 1;
        }

        puts("Handler assigned");
    }

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

    return 0;
}

/*
 * This will handle connection for each client
 * */
void *connection_handler(void *socket_desc)
{
    //Get the socket descriptor
    int sock = *(int*)socket_desc;
    int read_size;
    char *message , client_message[2000];

    //Send some messages to the client
    message = "Greetings! I am your connection handler\n";
    write(sock , message , strlen(message));

    message = "Now type something and i shall repeat what you type \n";
    write(sock , message , strlen(message));

    //Receive a message from client
    while( (read_size = recv(sock , client_message , 2000 , 0)) > 0 )
    {
        //Send the message back to client
        write(sock , client_message , strlen(client_message));
    }
```

```
    if(read_size == 0)
    {
        puts("Client disconnected");
        fflush(stdout);
    }
    else if(read_size == -1)
    {
        perror("recv failed");
    }

    //Free the socket pointer
    close(sock);
    free(socket_desc);

    return 0;
}
```

Run the above server and connect from multiple clients, it will handle all of them.

In order to compile a C program with **pthread.h,** you have to use **-lpthread** option in compiling command.

```
$ gcc server.c -lpthread && ./a.out
Socket created
bind done
Waiting for incoming connections...
```

Why TCP server must be running before the client attempts to initiate contact? What happen when you try to run TCP client program in the absence of any server program running?

What do you think, why don't we use **`accept()`** system call with UDP sockets?

What is difference between **`recv()`** and **`recvfrom()`** system calls?

Briefly describe functionality of following system calls?
`listen(), accept(), connect().`

Conclusion: