HO CHI MINH UNIVERSITY OF TECHNOLOGY

EMBEDDED SYSTEM

REPORT LAB 2

Lecturer: PhD Pham Hoang Anh

Student:

Le Huy Hoang: 1652212

Contents

1. Introduction 1
2. Design 1
    2.1. UDP 1
        2.1.1. Server side 1
        2.1.2. Client side 3
    2.2. TCP 4
        2.2.1. Server side 4
    2.2.2. Client 5
3. Result 6
    3.1. UDP server – client 6
    3.2. TCP single server – client 6
    3.3. TCP multi server – client 7
4. Reference 7

Figures

Figure 1. Initiation code for UDP server 2
Figure 2. Main part of server side 2
Figure 3. Initiation of UDP echo client side 3
Figure 4. Send and receive from server 3
Figure 5. TCP Echo server's init section 4
Figure 6. Infinite section of TCP echo server 5
Figure 7. Initiation part of TCP echo client 5
Figure 8. Main part of TCP echo client 6
Figure 9. Screen capture for UDP echo server - client demo, the left terminal is of server, the right one of client 6
Figure 10. Screen capture for TCP echo server - client demo, left one is server, right one is client 7
Figure 11. Screen capture for TCP echo server - client demo, left one is server, two right ones are clients 7

Keynotes

Times New Roman, font size 12:	normal text
`Courier New`:	code

1. Introduction

The Intel Edison is a computer-on-module that was offered by Intel as a development system for wearable devices and Internet of Things devices. It runs a small Linux version.

Secure Shell (SSH) is a cryptographic network protocol for operating network services securely over an unsecured network. The best known example application is for remote login to computer systems by users. SSH provides a secure channel over an unsecured network in a client-server architecture, connecting an SSH client application with an SSH server. Common applications include remote command-line login and remote command execution, but any network service can be secured with SSH.

The Spanning Tree Protocol (STP) is a network protocol that builds a loop-free logical topology for Ethernet networks. The basic function of STP is to prevent bridge loops and the broadcast radiation that results from them. Spanning tree also allows a network design to include backup links to provide fault tolerance if an active link fails.

In this report, I use SSH to control Intel Edison and STP to transfer source code from my machine (Linux machine) to Intel Edison as well as enrich knowledge about UNIX network programming. The programs I make are simple version of echo server – client. The only difference between the two is one uses TCP and another uses UDP.

2. Design

In the jargon, the design of two program are similar in server side and client side. However, UDP-used program needs a set of built-in functions to implement rather than using directly system calls like TCP-used program.

2.1. UDP

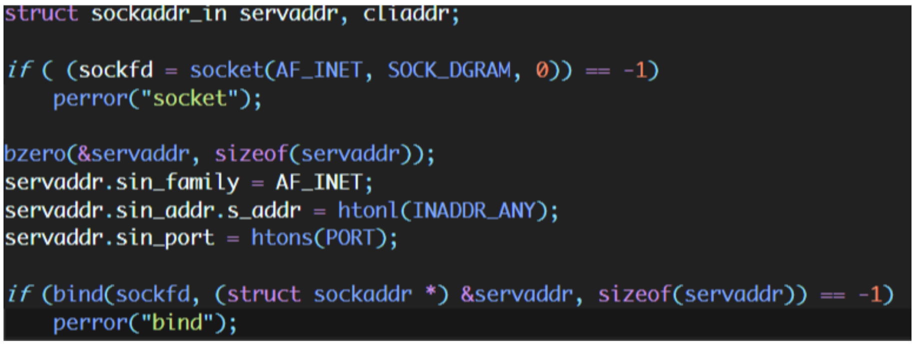
2.1.1. Server side

Server side of UDP program does not require to handle many incoming connections at the same time.

As other server side programs, mine includes 2 main part:

- Initiation: set up things for server such as socket address, socket port…
- Infinite loop: to handle clients

2.1.1.1. Initiation



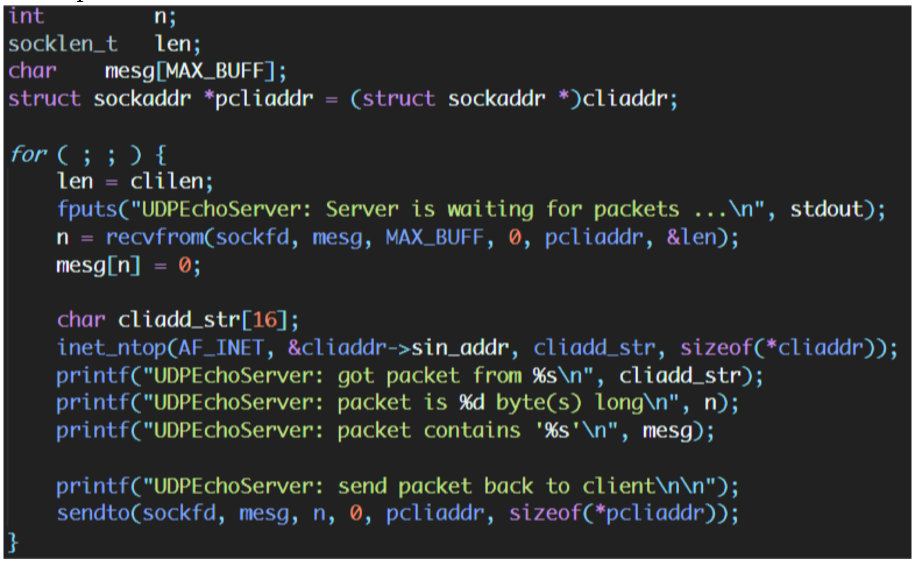
```c
struct sockaddr_in servaddr, cliaddr;

if ( (sockfd = socket(AF_INET, SOCK_DGRAM, 0)) == -1)
    perror("socket");

bzero(&servaddr, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
servaddr.sin_port = htons(PORT);

if (bind(sockfd, (struct sockaddr *) &servaddr, sizeof(servaddr)) == -1)
    perror("bind");
```

Figure 1. Initiation code for UDP server

Detail of the given snippet code is as follows:

`SOCK_DGRAM`: use UDP protocol
`sockaddr_in`: standard struct of socket. Not only server but also client do use this struct for socket.
`AF_INET`: use IPv4
`htonl`: convert number in host's order byte to network's order byte. Network order byte is little endian while for some host, that can be big endian.
`INADDR_ANY (= 0)`: means computer can take any arbitrary address for server since this is localhost server.
`htons`: their function is the same as that of htonl but with smaller convert number range
`PORT`: this is an given port set by programmer.

2.1.1.2. Infinite loop



```c
int         n;
socklen_t   len;
char        mesg[MAX_BUFF];
struct sockaddr *pcliaddr = (struct sockaddr *)cliaddr;

for ( ; ; ) {
    len = clilen;
    fputs("UDPEchoServer: Server is waiting for packets ...\n", stdout);
    n = recvfrom(sockfd, mesg, MAX_BUFF, 0, pcliaddr, &len);
    mesg[n] = 0;

    char cliadd_str[16];
    inet_ntop(AF_INET, &cliaddr->sin_addr, cliadd_str, sizeof(*cliaddr));
    printf("UDPEchoServer: got packet from %s\n", cliadd_str);
    printf("UDPEchoServer: packet is %d byte(s) long\n", n);
    printf("UDPEchoServer: packet contains '%s'\n", mesg);

    printf("UDPEchoServer: send packet back to client\n\n");
    sendto(sockfd, mesg, n, 0, pcliaddr, sizeof(*pcliaddr));
}
```

Figure 2. Main part of server side

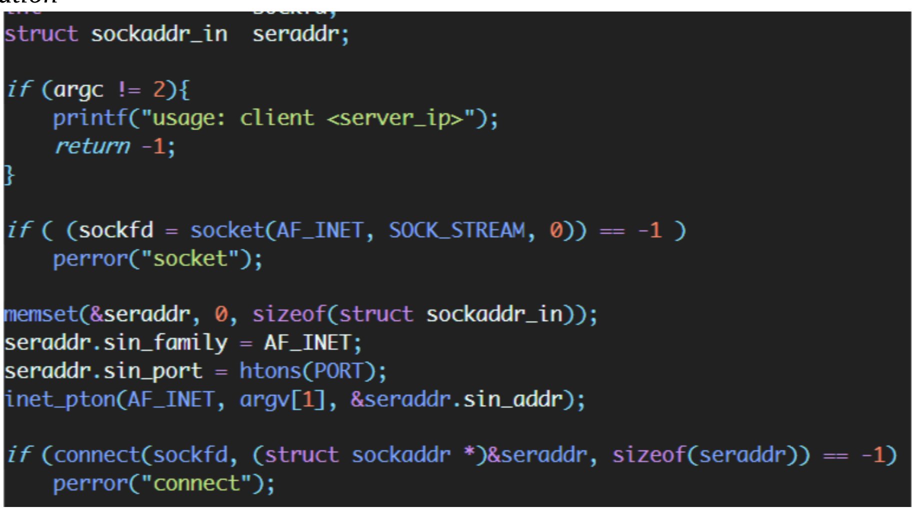
As I mentioned before, UDP server code and TCP server code are different in some. The first one is UDP server has no **accept** function.

`inet_ntop()` function is used to convert IP from numeric format which is saved in `struct cliaddr` to presentation format which is later stored in string `cliadd_str`. This string is declared 16 chars long for the reason that the maximum length of presentation IPv4 IP address is: 255.255.255.255, plus **'\0'** character will be exactly 16 characters long. The declaration is to ensure this issue.

The second difference between UDP server and TCP server is UDP server code uses `recvfrom()` and `sendto()` functions are dedicated for UDP protocol.

2.1.2. Client side

2.1.2.1. Initiation

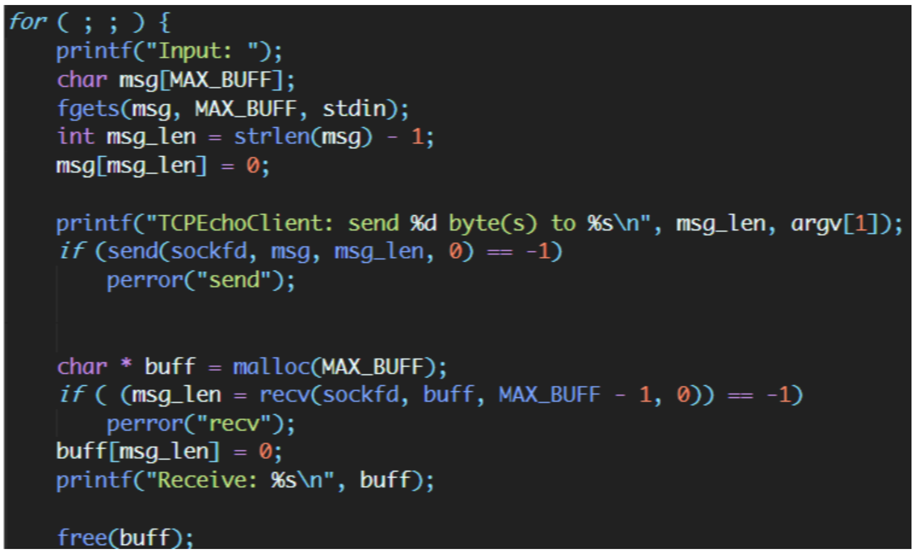


```
struct sockaddr_in  seraddr;

if (argc != 2){
    printf("usage: client <server_ip>");
    return -1;
}

if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1 )
    perror("socket");

memset(&seraddr, 0, sizeof(struct sockaddr_in));
seraddr.sin_family = AF_INET;
seraddr.sin_port = htons(PORT);
inet_pton(AF_INET, argv[1], &seraddr.sin_addr);

if (connect(sockfd, (struct sockaddr *)&seraddr, sizeof(seraddr)) == -1)
    perror("connect");
```

Figure 3. Initiation of UDP echo client side

Basically, the skeleton of client side is not much different with that of the server.

2.1.2.2. Sending to and receiving from server



```
for ( ; ; ) {
    printf("Input: ");
    char msg[MAX_BUFF];
    fgets(msg, MAX_BUFF, stdin);
    int msg_len = strlen(msg) - 1;
    msg[msg_len] = 0;

    printf("TCPEchoClient: send %d byte(s) to %s\n", msg_len, argv[1]);
    if (send(sockfd, msg, msg_len, 0) == -1)
        perror("send");


    char * buff = malloc(MAX_BUFF);
    if ( (msg_len = recv(sockfd, buff, MAX_BUFF - 1, 0)) == -1)
        perror("recv");
    buff[msg_len] = 0;
    printf("Receive: %s\n", buff);

    free(buff);
```

Figure 4. Send and receive from server

Although there exists an infinite for loop, this loop and the loop of server is totally irrelevant in terms of core function. In particular, the loop in client serves the constant read-send-receive-show cycles. Client first reads from **STDIN** (`fgets()` function, this function reads until reaching **'\n'** character (Enter button in keyboard)), send to Echo Server via `send()`. It then receives message by `recv()`. The client only reads no more than `MAX_BUFF` bytes and later shows onto **STDOUT** (in this case is the screen).

For loop of server, on the other hand, is to serve constantly request sent by clients.

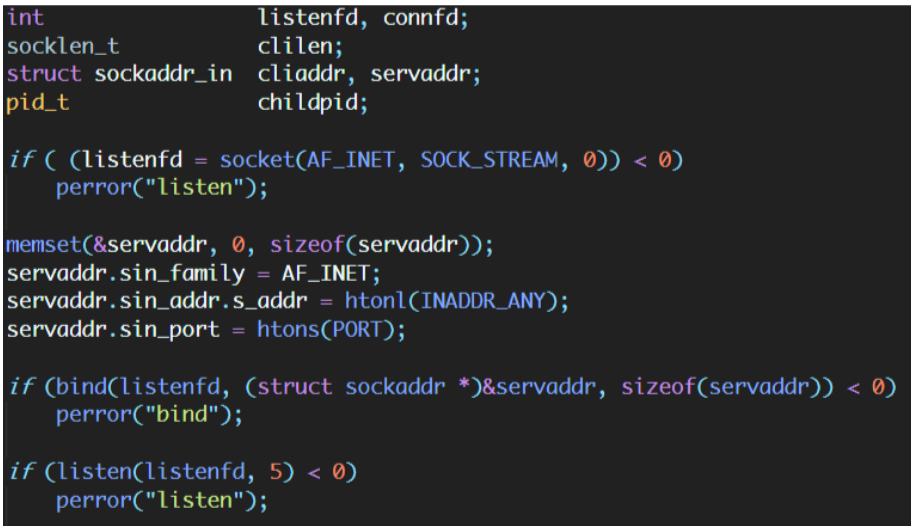
2.2. TCP

Normally, TCP send-receive functions can utilize system calls (`write()` and `read()` respectively), but in this program, I do use built-in functions.

Besides, I will only present the design of the server version that permits many clients connect at the same time. In the summit code, however, I do attach single-request server version.

2.2.1. Server side

2.2.1.1. Initiation



```
int                 listenfd, connfd;
socklen_t           clilen;
struct sockaddr_in  cliaddr, servaddr;
pid_t               childpid;

if ( (listenfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    perror("listen");

memset(&servaddr, 0, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
servaddr.sin_port = htons(PORT);

if (bind(listenfd, (struct sockaddr *)&servaddr, sizeof(servaddr)) < 0)
    perror("bind");

if (listen(listenfd, 5) < 0)
    perror("listen");
```

Figure 5. TCP Echo server's init section

It is somewhat similar with UDP server.

`SOCK_STREAM` in `socket()` implies that this server uses TCP protocol.

However, TCP server does have `listen()` function. The parameter `5` implies the backlog for this socket is 5.

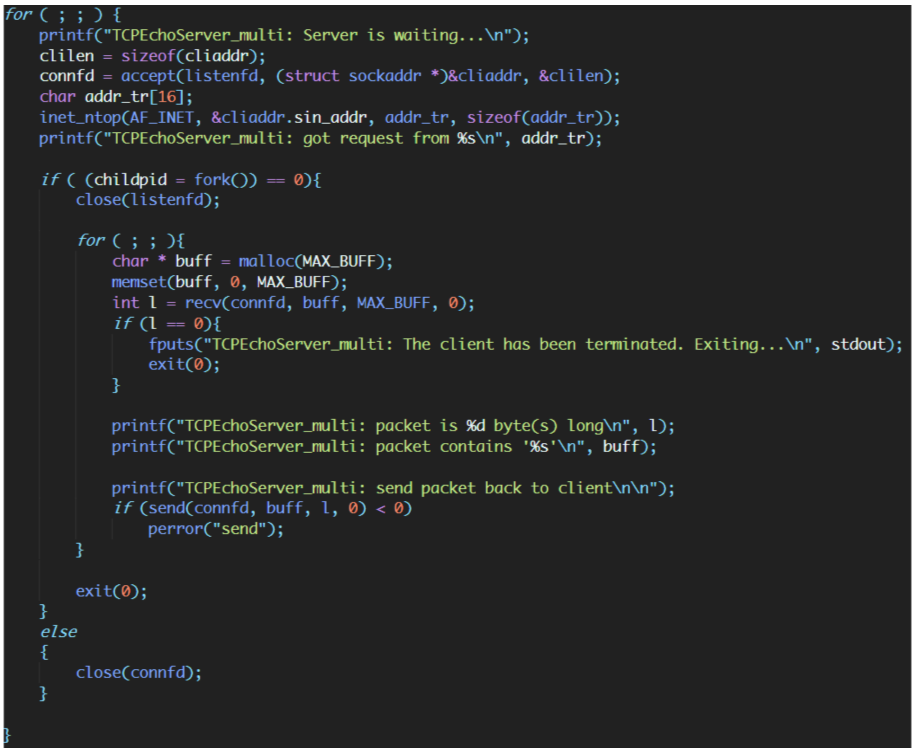
2.2.1.2. Infinite loop

Fig. 6 shows the code of TCP multi-client server. The server, in fact, has 2 different parts serving 2 separated jobs:

1. Accepting incoming connection
2. Serving connection

The first one consists of six first lines of the code. This section is responsible for receiving request from new connection.

The second one is embedded in `fork()` sub-section. After the above section has received the request, it then creates a new process by `fork()` and let the newly created process to deal with it. The first section later waits for new request.

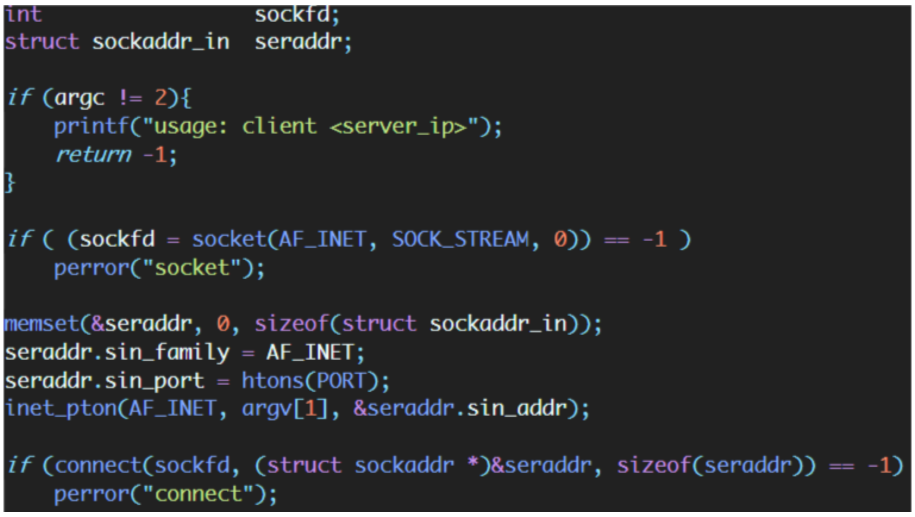


```
for ( ; ; ) {
    printf("TCPEchoServer_multi: Server is waiting...\n");
    clilen = sizeof(cliaddr);
    connfd = accept(listenfd, (struct sockaddr *)&cliaddr, &clilen);
    char addr_tr[16];
    inet_ntop(AF_INET, &cliaddr.sin_addr, addr_tr, sizeof(addr_tr));
    printf("TCPEchoServer_multi: got request from %s\n", addr_tr);

    if ( (childpid = fork()) == 0){
        close(listenfd);

        for ( ; ; ){
            char * buff = malloc(MAX_BUFF);
            memset(buff, 0, MAX_BUFF);
            int l = recv(connfd, buff, MAX_BUFF, 0);
            if (l == 0){
                fputs("TCPEchoServer_multi: The client has been terminated. Exiting...\n", stdout);
                exit(0);
            }

            printf("TCPEchoServer_multi: packet is %d byte(s) long\n", l);
            printf("TCPEchoServer_multi: packet contains '%s'\n", buff);

            printf("TCPEchoServer_multi: send packet back to client\n\n");
            if (send(connfd, buff, l, 0) < 0)
                perror("send");
        }

        exit(0);
    }
    else
    {
        close(connfd);
    }

}
```

Figure 6. Infinite section of TCP echo server

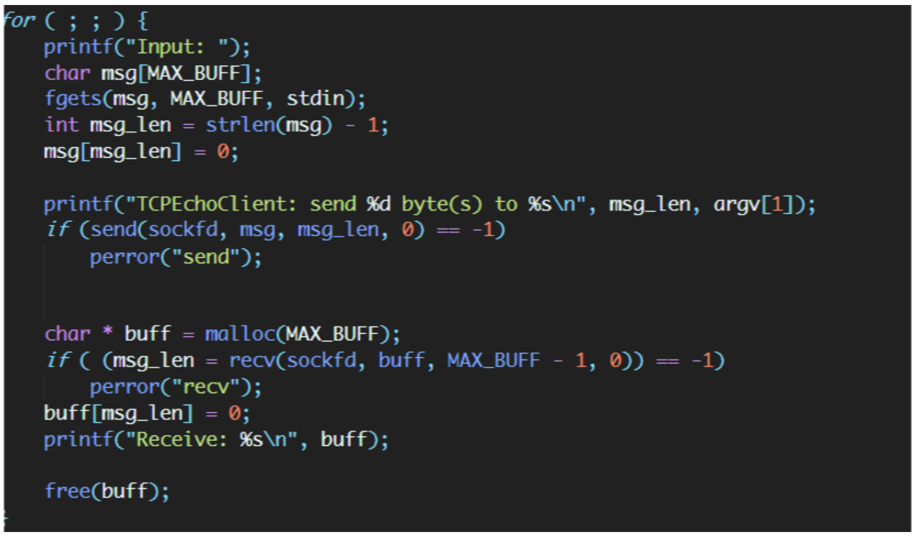
2.2.2. Client

TCP client is similar with UDP client.



```
int                 sockfd;
struct sockaddr_in  seraddr;

if (argc != 2){
    printf("usage: client <server_ip>");
    return -1;
}

if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1 )
    perror("socket");

memset(&seraddr, 0, sizeof(struct sockaddr_in));
seraddr.sin_family = AF_INET;
seraddr.sin_port = htons(PORT);
inet_pton(AF_INET, argv[1], &seraddr.sin_addr);

if (connect(sockfd, (struct sockaddr *)&seraddr, sizeof(seraddr)) == -1)
    perror("connect");
```

Figure 7. Initiation part of TCP echo client



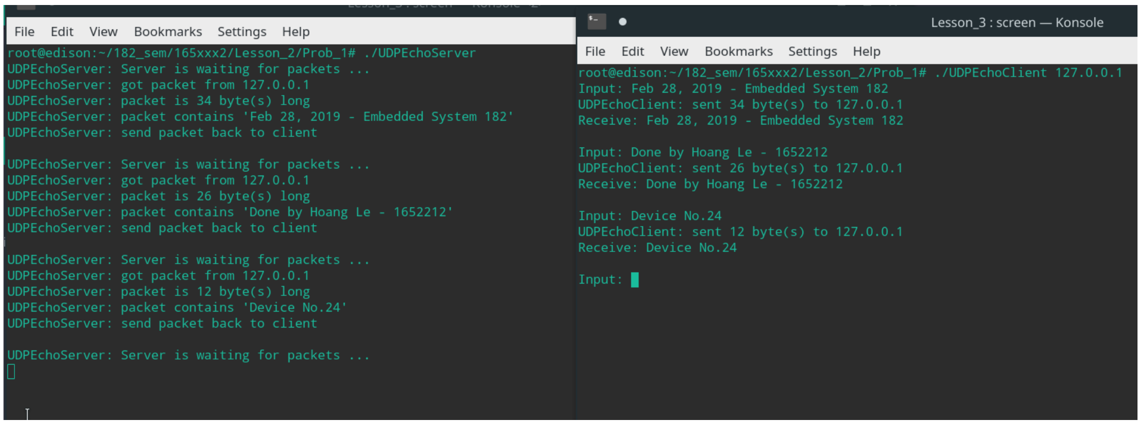
```c
for ( ; ; ) {
    printf("Input: ");
    char msg[MAX_BUFF];
    fgets(msg, MAX_BUFF, stdin);
    int msg_len = strlen(msg) - 1;
    msg[msg_len] = 0;

    printf("TCPEchoClient: send %d byte(s) to %s\n", msg_len, argv[1]);
    if (send(sockfd, msg, msg_len, 0) == -1)
        perror("send");


    char * buff = malloc(MAX_BUFF);
    if ( (msg_len = recv(sockfd, buff, MAX_BUFF - 1, 0)) == -1)
        perror("recv");
    buff[msg_len] = 0;
    printf("Receive: %s\n", buff);

    free(buff);
```

Figure 8. Main part of TCP echo client

3. Result

3.1. UDP server – client

I created two SSH to Intel Edison, one for server, another for client. Client connects to 127.0.0.1.


File Edit View Bookmarks Settings Help
Lesson_3 : screen — Konsole
File Edit View Bookmarks Settings Help

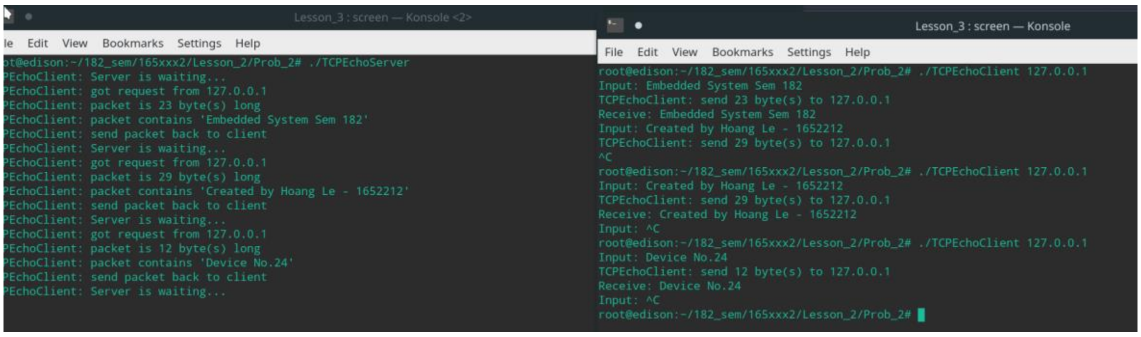
```
root@edison:~/182_sem/165xxx2/Lesson_2/Prob_1# ./UDPEchoServer
UDPEchoServer: Server is waiting for packets ...
UDPEchoServer: got packet from 127.0.0.1
UDPEchoServer: packet is 34 byte(s) long
UDPEchoServer: packet contains 'Feb 28, 2019 - Embedded System 182'
UDPEchoServer: send packet back to client

UDPEchoServer: Server is waiting for packets ...
UDPEchoServer: got packet from 127.0.0.1
UDPEchoServer: packet is 26 byte(s) long
UDPEchoServer: packet contains 'Done by Hoang Le - 1652212'
UDPEchoServer: send packet back to client

UDPEchoServer: Server is waiting for packets ...
UDPEchoServer: got packet from 127.0.0.1
UDPEchoServer: packet is 12 byte(s) long
UDPEchoServer: packet contains 'Device No.24'
UDPEchoServer: send packet back to client

UDPEchoServer: Server is waiting for packets ...
```

```
root@edison:~/182_sem/165xxx2/Lesson_2/Prob_1# ./UDPEchoClient 127.0.0.1
Input: Feb 28, 2019 - Embedded System 182
UDPEchoClient: sent 34 byte(s) to 127.0.0.1
Receive: Feb 28, 2019 - Embedded System 182

Input: Done by Hoang Le - 1652212
UDPEchoClient: sent 26 byte(s) to 127.0.0.1
Receive: Done by Hoang Le - 1652212

Input: Device No.24
UDPEchoClient: sent 12 byte(s) to 127.0.0.1
Receive: Device No.24

Input:
```

Figure 9. Screen capture for UDP echo server - client demo, the left terminal is of server, the right one of client

3.2. TCP single server – client

I did similar with UDP.

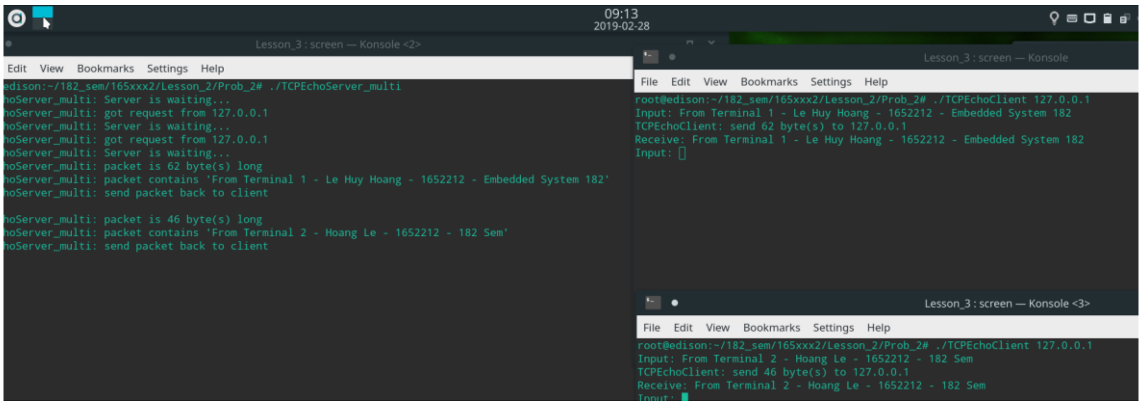

Lesson_3 : screen — Konsole <2>
le Edit View Bookmarks Settings Help
Lesson_3 : screen — Konsole
File Edit View Bookmarks Settings Help

```
ot@edison:~/182_sem/165xxx2/Lesson_2/Prob_2# ./TCPEchoServer
PEchoClient: Server is waiting...
PEchoClient: got request from 127.0.0.1
PEchoClient: packet is 23 byte(s) long
PEchoClient: packet contains 'Embedded System Sem 182'
PEchoClient: send packet back to client
PEchoClient: Server is waiting...
PEchoClient: got request from 127.0.0.1
PEchoClient: packet is 29 byte(s) long
PEchoClient: packet contains 'Created by Hoang Le - 1652212'
PEchoClient: send packet back to client
PEchoClient: Server is waiting...
PEchoClient: got request from 127.0.0.1
PEchoClient: packet is 12 byte(s) long
PEchoClient: packet contains 'Device No.24'
PEchoClient: send packet back to client
PEchoClient: Server is waiting...
```

```
root@edison:~/182_sem/165xxx2/Lesson_2/Prob_2# ./TCPEchoClient 127.0.0.1
Input: Embedded System Sem 182
TCPEchoClient: send 23 byte(s) to 127.0.0.1
Receive: Embedded System Sem 182
Input: Created by Hoang Le - 1652212
TCPEchoClient: send 29 byte(s) to 127.0.0.1
^C
root@edison:~/182_sem/165xxx2/Lesson_2/Prob_2# ./TCPEchoClient 127.0.0.1
Input: Created by Hoang Le - 1652212
TCPEchoClient: send 29 byte(s) to 127.0.0.1
Receive: Created by Hoang Le - 1652212
Input: ^C
root@edison:~/182_sem/165xxx2/Lesson_2/Prob_2# ./TCPEchoClient 127.0.0.1
Input: Device No.24
TCPEchoClient: send 12 byte(s) to 127.0.0.1
Receive: Device No.24
Input: ^C
root@edison:~/182_sem/165xxx2/Lesson_2/Prob_2#
```

Figure 10. Screen capture for TCP echo server - client demo, left one is server, right one is client

3.3. TCP multi server – client


09:13
2019-02-28
Lesson_3 : screen — Konsole <2>
Edit View Bookmarks Settings Help
Lesson_3 : screen — Konsole
File Edit View Bookmarks Settings Help
Lesson_3 : screen — Konsole <3>
File Edit View Bookmarks Settings Help

```
edison:~/182_sem/165xxx2/Lesson_2/Prob_2# ./TCPEchoServer_multi
hoServer_multi: Server is waiting...
hoServer_multi: got request from 127.0.0.1
hoServer_multi: Server is waiting...
hoServer_multi: got request from 127.0.0.1
hoServer_multi: Server is waiting...
hoServer_multi: packet is 62 byte(s) long
hoServer_multi: packet contains 'From Terminal 1 - Le Huy Hoang - 1652212 - Embedded System 182'
hoServer_multi: send packet back to client

hoServer_multi: packet is 46 byte(s) long
hoServer_multi: packet contains 'From Terminal 2 - Hoang Le - 1652212 - 182 Sem'
hoServer_multi: send packet back to client
```

```
root@edison:~/182_sem/165xxx2/Lesson_2/Prob_2# ./TCPEchoClient 127.0.0.1
Input: From Terminal 1 - Le Huy Hoang - 1652212 - Embedded System 182
TCPEchoClient: send 62 byte(s) to 127.0.0.1
Receive: From Terminal 1 - Le Huy Hoang - 1652212 - Embedded System 182
Input:
```

```
root@edison:~/182_sem/165xxx2/Lesson_2/Prob_2# ./TCPEchoClient 127.0.0.1
Input: From Terminal 2 - Hoang Le - 1652212 - 182 Sem
TCPEchoClient: send 46 byte(s) to 127.0.0.1
Receive: From Terminal 2 - Hoang Le - 1652212 - 182 Sem
Input:
```

Figure 11. Screen capture for TCP echo server - client demo, left one is server, two right ones are clients

4. Reference

Steven - UNIX Network Programming, volume 1