EECS 3214 Assignment 2

Jun Lin Chen (Michael)
Student ID: 214533111
CSE Account: chen256

Overview

This project contains two programs: the client and the server. The client program allows you to log on to the server and exchange message with other people logged on to the server.

The server is also a directory server. The client can obtain a list of active clients from the server. And then the client can establish connection with another client, which is a peer-to-peer connection. And then those connected clients can exchange message each other without passing the message to the server.

In this article, "peer" is always referring to the "client program".

How It Works

Using the same ideas in the Assignment 1, I set up the directory server. The server will return the list of active clients in the following format:

```
LIST          ②          ③          ④
①[1] 127.0.0.1:42816 << 54895
 [2] 127.0.0.1:42814 << 5489
 [3] 127.0.0.1:42812 << 4567
⑤3 on-line user(s).
```

① ID of the clients.
② IP address of the clients.
③ The port to the directory server.
④ The port listening for other clients.
⑤ Number of active clients.

On the client side, I wrote a parser that can analysis the return values from the `LIST` command. After the client has information about the other clients, the user can use `CONNECT` command follows by the ID of the peer to connect to the other client.

```
>CONNECT 1
```

The client which accept the connection will behave like the server. The message can be exchanged without going through the directory server.

Design Decisions

➢ Peer to Peer Structure

Peer-to-peer structure is similar to server-client structure, but the peers should decide who needs to be the server. In my design, each client program has two modes.

1. Server mode
   `mc_peer_passive_response()`
   This thread is for the server mode. It accepts new connection from the other peers. It behaves like the server program in Assignment 1.

2. Client mode
   `mc_peer_active_response()`
   This thread is for the client mode. It can create new connection to the other peers. It behaves like the client program in Assignment 1.

If one of these two modes is active, it means the peer is occupied. The peer should only be able to accept any connection if it is not occupied.

➢ Directory Protocol

As for the client, it need to understand the LIST information sent from the server. I wrote a parser (`mc_update_peer_list()`). If a message has a prefix "LIST", the program will pass the message to that parser. The parser uses `scanf()` to pick up the useful information and store them into a one-way linked list.

And then you can use the `CONNECT` command to connect to other peers.

➢ Chatting Protocol

To avoid corrupting in messages and commands, I add the keyword "MSG" in front of every messages sent by the client. So, a message will be packed into a `MSG` command. The other peer will be required to unpack the information by removing the "MSG" prefix.

➢ Concurrent Design

Although the requirement for this assignment says that "The peers need not be concurrent", it does not mean we can write this program without concurrent design. In order to have a good user experience, I use

two threads in the client program. Therefore, the client can handle user inputs and handle the connection issue at the same time.

➢ **Single Connection**

Considering we can only use one connection per client, we cannot read and write at the same time. Instead of switching between reading and writing manually, I have implemented a heart-beat mechanism. The server will send a request to the client periodically.

In the client program, I use a Mutex structure from `pthread.h` (`pthread_mutex_t mc_holding_mutex`) to control the responses. And I have two separate threads in the client program.

1. `mc_client_response()` thread:
   This thread handles the user command and send the command to the server.
2. `mc_heartbeat_response()` thread
   This thread handles the message from the server and sends the default response back to the server. And it maintains the heartbeats between the server and the client.

Once the client has finished receiving the message. The server will be block on `revc()`. And the client will have a chance to send a message to the server. So I unlock the Mutex.

If the user has invoked any command, the `mc_client_response()` thread will lock the Mutex and send the message to the server.

Avoiding flooding the network, The `mc_heartbeat_response()` thread `usleep()` for a short time (`#define INTERVAL 50000`). Then this thread will try to lock the Mutex and send the default message (`IMOK` command) to the server. If the Mutex has been locked already, it means someone has already used this chance to send the message to the server. This thread will go back to `revc()` and wait for the next message from the server.

Similar in the peer-to-peer connection, the procedure `mc_peer_passive_response()` is like `mc_message_handler()` in the server program. `mc_peer_active_response()` is like the `mc_heartbeat_response()` in the client program. They are sending message periodically so that both peers have a chance to send message and receive message in one single connection.

Possible Improvements

- **Security**

This program is not protected by any kind of encryption. You can hack it easily. Next step, I think I should implement something like TLS.

- **Not a real-time response**

Because we only use one connection per client. The client and the server is switching between reading and writing. Although the interval is very short, it is not a real-time respond. For convenient, I should implement at least two connections between the client and the server.

- **No guarantee of delivery**

There is no any mechanism to guarantee the message has been sent from the server to the client.

Known Issues

- **IPv6 support**

This program will only work on IPv4 environment.

- **Subnet address problem**

A peer cannot connect to the peer hide by a subnet.

For example, if a peer connects the server using 127.0.0.1. If another peer out of the subnet wants to connect to the peer in the subnet, it will not be able to do that. Because it only knows the subnet address.

How to Use

In the server program, you can use these commands:

`LIST`
The program will print all the connected clients.

`CLOSE`
The program will issue a shutdown notification to all the connected clients. And it will exit in a few seconds.

`message`
this can be any message you want to send to all the clients.

In the client program, you can use these commands:

`JOIN`
The server will mark you as an active user. Other users will see your IP address and port if they use the `LIST` command. If you have joined successfully, a message will be printed.

`LEAVE`
The server will mark you as an inactive user. Other users will no longer see your IP address and port if they use the `LIST` command. If you have left successfully, a message will be printed.

`LIST`
The program will print all the active connected clients.

`CLOSE`
The program will close the connection and exit.

`BROADCAST` `message`
The message can be any message you want to send to all the other clients. If the message has been sent to the server, a message will be printed to the standard output.

`CONNECT` `id`
You can see the list of available peer by using the LIST command. The id of the peer is listed in the beginning of the list. Use this command to establish peer to peer connection.

`DISCONNECT`
The program will disconnect peer-to-peer connection if there is an established peer-to-peer connection.

`message`
this can be any message you want to send to the other peer.

Build and Deploy

This project is written in C++. And it contains 7 files:

- client.hpp
- client.cpp
- server.hpp
- server.cpp
- mc_util.hpp
- mc_util.cpp
- makefile

client.hpp and ***client.cpp*** are the source code files for the client program. ***server.hpp*** and ***server.cpp*** are the source code files for the server program. ***mc_util.hpp*** and ***mc_util.cpp*** contains the functions can be used in both the client program and the server program.

For avoiding the naming conflict, all the functions and large-scope-variables have the prefix ***mc_*** .

To compile the programs, you can use the ***makefile*** that provided in this solution by using the command:

```
make
```

These commands can also be used for compiling the programs:
Server:

```
gcc server.hpp server.cpp \
 mc_util.hpp mc_util.cpp \
-lstdc++ -lpthread -o server.out
```

Client:

```
gcc client.hpp client.cpp \
 mc_util.hpp mc_util.cpp \
-lstdc++ -lpthread -o client.out
```

Then you can run the server by execute it
Server:

```
./server.out
```

Client:

```
./client.out 127.0.0.1
```

The server program listens on `0.0.0.0:23111`. The client program can accept one parameter for the server's IP address. You may also enter the IP address after the client program is running without providing this parameter.

I am also hosting the server program on my VPS. Should you do not want to run the server, you can try the IP address of

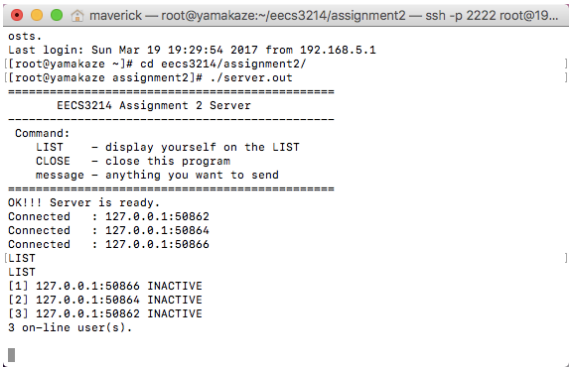
`www.masterchan.me` (106.185.43.242)

Screenshots & Test Cases

✧ Example 1

Server `LIST` command:

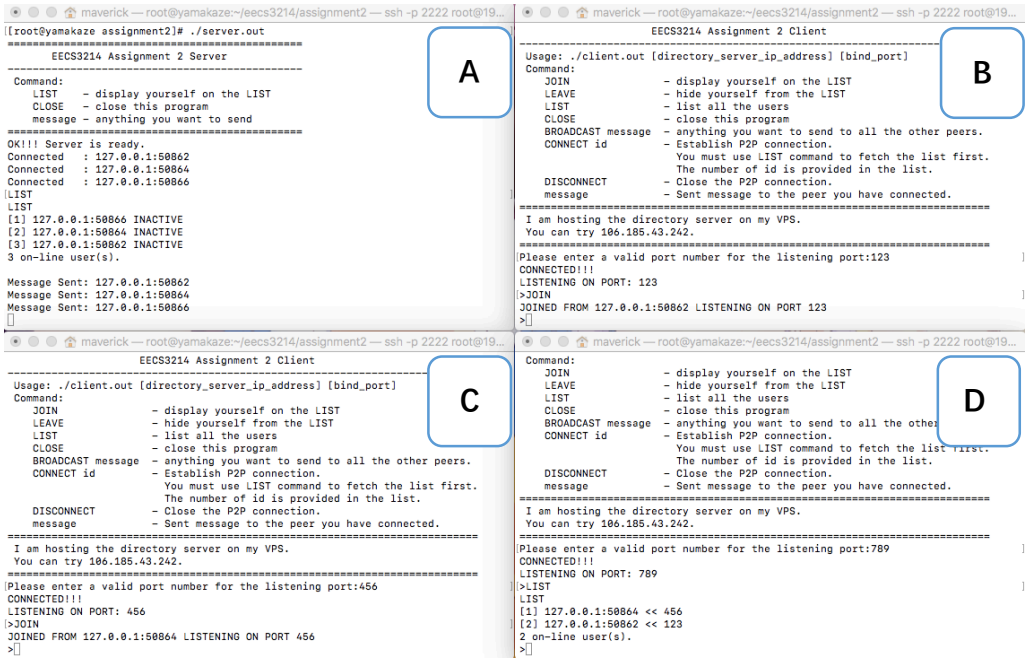
You can see all the connected client in the server even though they did not `JOIN` the network.



✧ Example 2

Client `JOIN` `LIST` command:

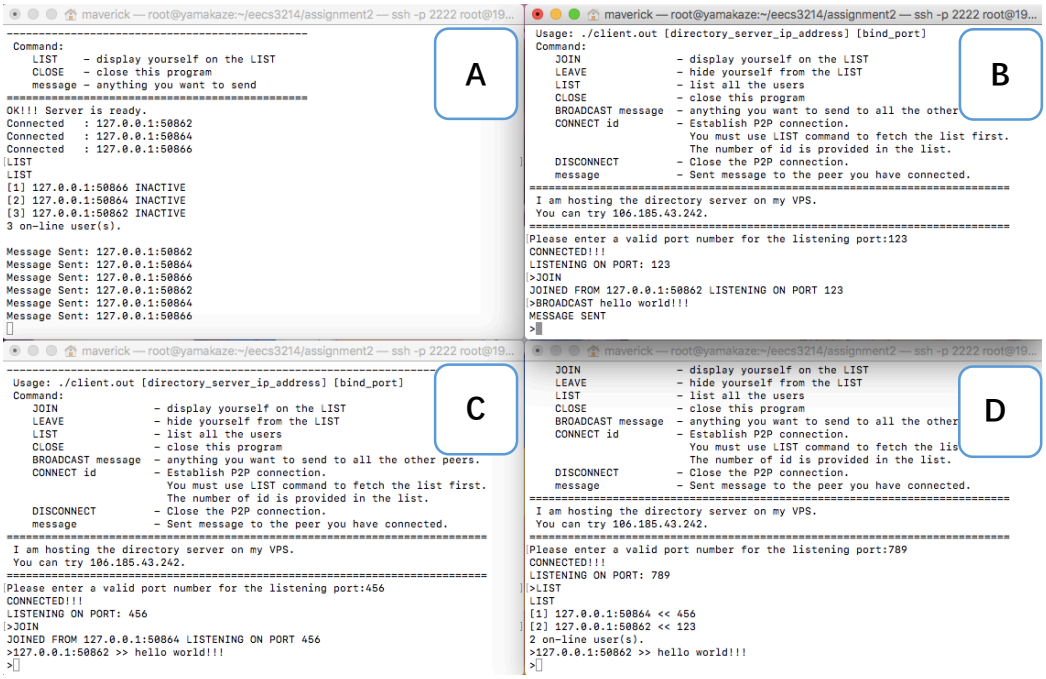
In this example, B, C and D are the client. A is the server. B and C has `JOIN` the network, therefore, in A you can see the port number for peer-to-peer connection. You can see they were listed in D as well.



✧ **Example 3**

Client `BROADCAST` message:

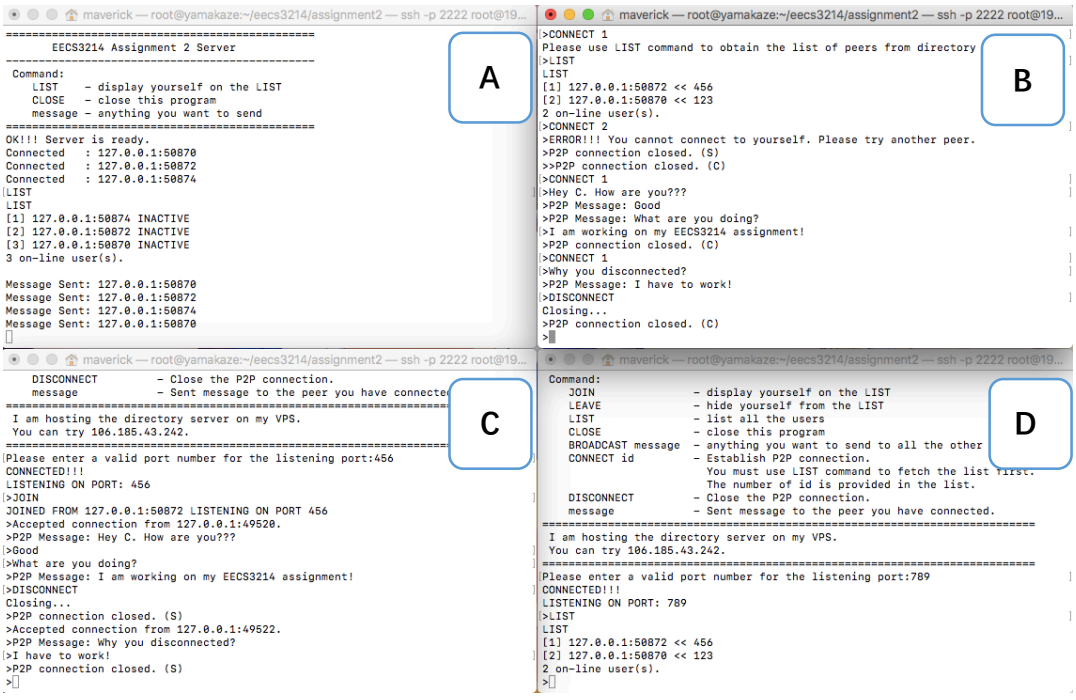
This is not a requirement for this assignment but I did that, a client can send message to all the other clients. In this case, B broadcast a message. C and D received that message.



✧ **Example 4**

Peer-to-peer message:

In this case B used the `LIST` command and `CONNECT` command connected to C. Then, B and C were able to exchange message. Either B or C can disconnect the connection. D cannot read the messages because they are peer-to-peer. And you can also see the messages did not went to A.



✧ **Example 5**

Directory server closing:

The server was closing. You can see the client is received notification from the server and closing the port properly.