EECS 3214 Assignment 1

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

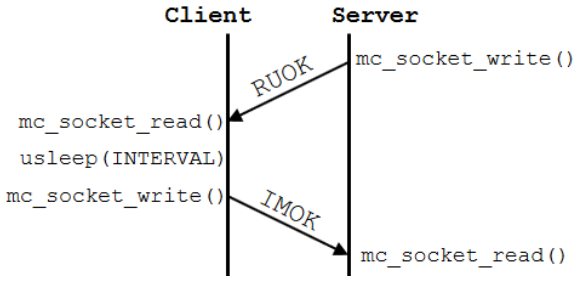
Overview

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.

How It Works

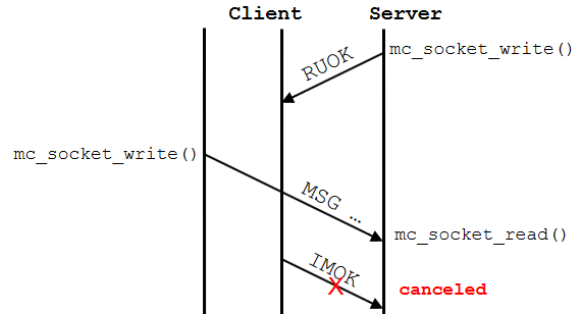
User should know the IP address where the server is listening, and provide it to the client program.

After the client connected to the server. The server will ask the client if the client is okay (`RUOK` command). If the client have not invoked any command, the client will sleep for a while then respond to the server saying that he is okay (`IMOK` command). And the server will ask the client `RUOK` again.



During this process, if the server has any message for the client, instead of sending the `RUOK` command, the server will send the that message to the client. And the client will print out that message.

On the other hand, if the client has any commands issued by the user, instead of sending the `IMOK` command, the client will send that command to the server. And the server will handle that command.



With these periodic signal, the client and the server can switch between reading and writing. Therefore, the message can be deliver in both ways using only one connection.

Design Decisions

➢ Concurrent

Although this is an assignment for a network course, the concurrent and multithread programming skills are very important in this project. All the design decisions in this project follow the ideas from multithread and concurrent.

When the server program finished initialization. It will create a thread for listening (`mc_listener()`). When a new connection was established, the listening thread will also create a new thread for the client (`mc_message_handler()`). Therefore, each client will have a dedicated thread to handle the messages.

➢ 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.

➢ **User Manage**

All the connected clients are managed by a two-way linked list in the server program. The `LIST` command in the server program will show all the users. But the `LIST` command in the client program will only show the user who has joined the list using the `JOIN` command.

The user list structure is managed by the Mutex (`mc_user_list_mutex`). Any operations related to the user list must acquire a lock on this Mutex.

Message can be spread out easily by using this structure. Because each node has two pointers to its previous node and its next node. The iteration process takes $O(n)$, which n is the number of clients. And adding or removing a user will only take a constant time.

➢ **Message Buffer**

Each client will have message buffer on the server (`client_structure.message_buffer`). The access to this data shall be strictly controlled by the Mutex for this buffer (`client_structure.message_mutex`).

The user thread shall not send message to client by direct invoking the `mc_socket_write()` function. But It should append the message to the corresponding buffer. The buffer will be evacuated and delivered to the client by the `mc_message_handler()` thread periodically.

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.

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)

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.

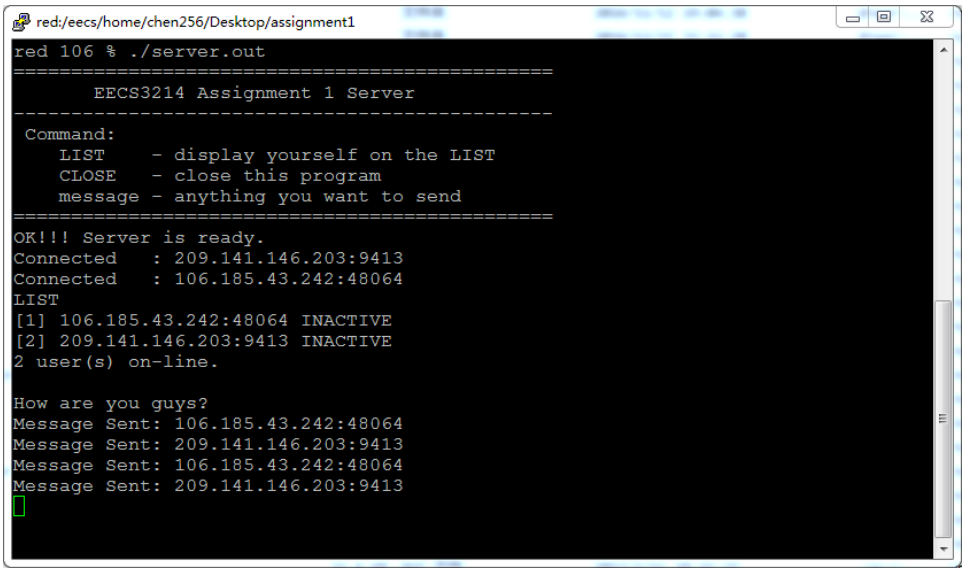
`LIST`
The program will print all the active connected clients.

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

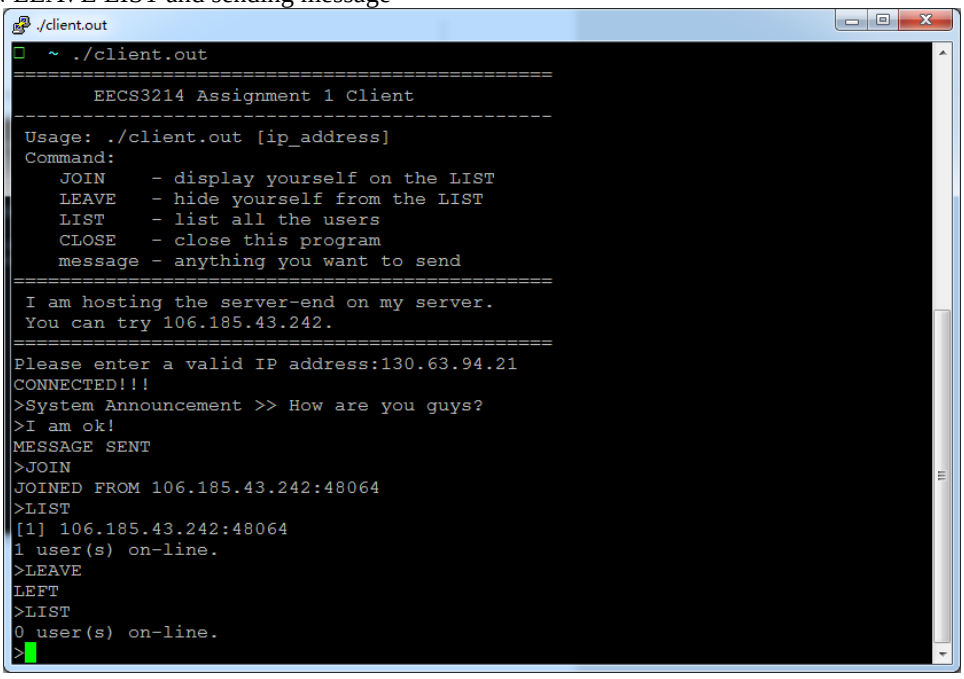
`message`
this can be any message you want to send to all the other clients. If the message has been sent to the server, a message will be printed to the standard output.

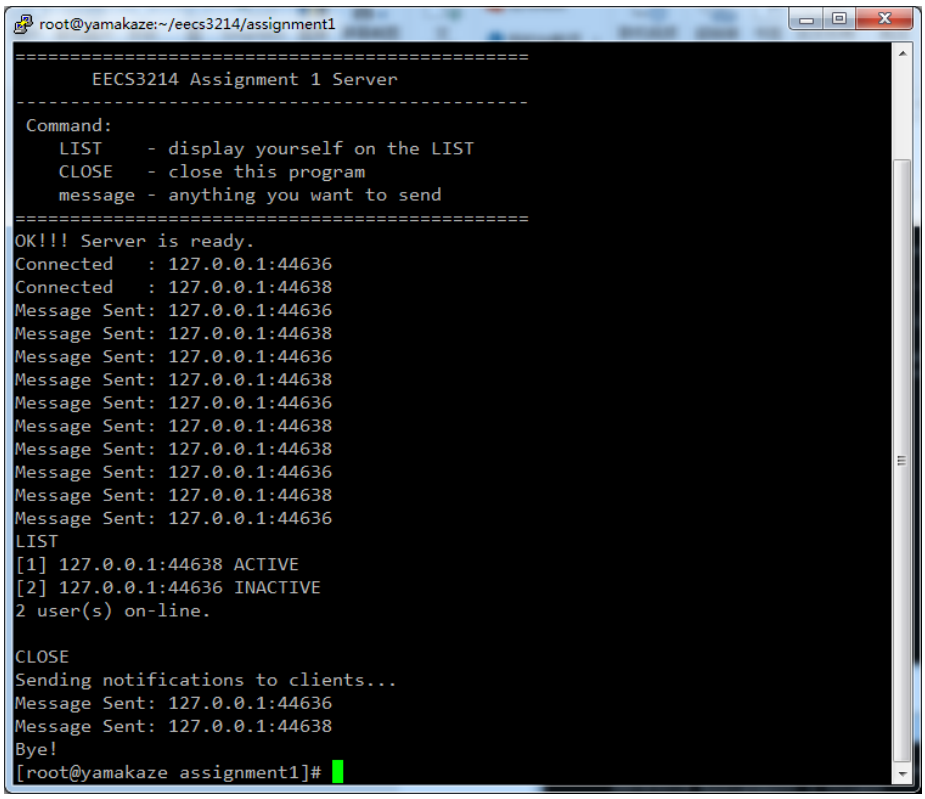
Screenshots

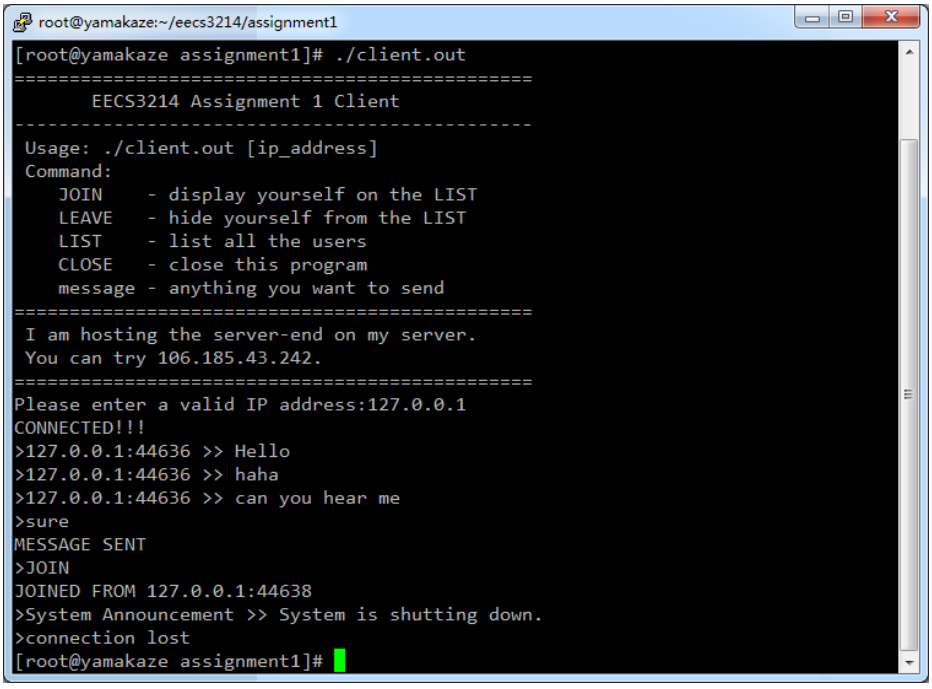
Server LIST

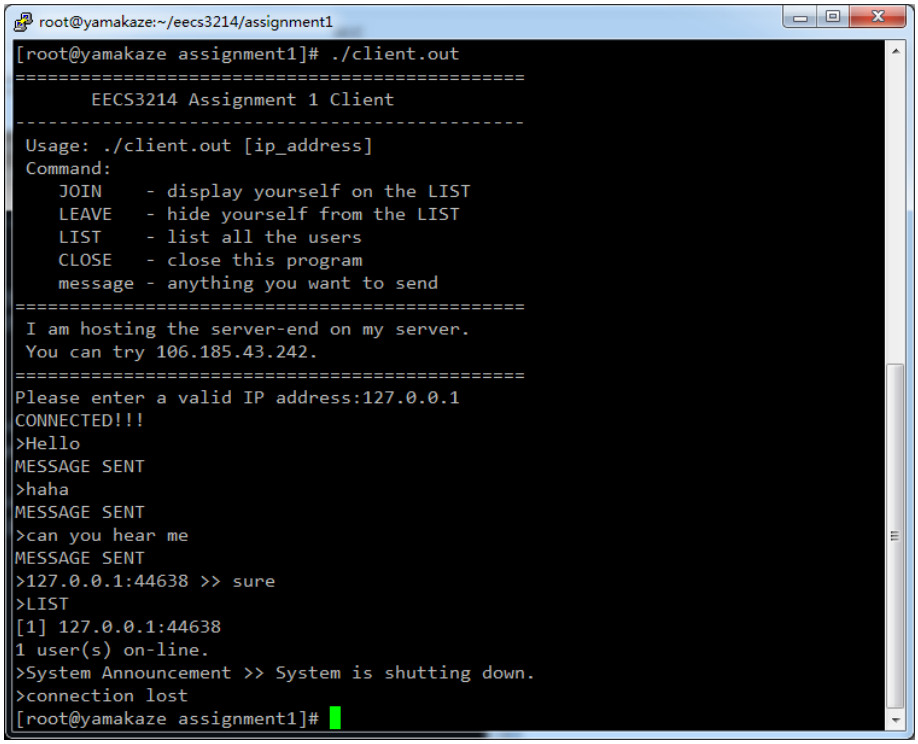


Client JOIN LEAVE LIST and sending message




root@yamakaze:~/eecs3214/assignment1
EECS3214 Assignment 1 Server
Command:
LIST - display yourself on the LIST
CLOSE - close this program
message - anything you want to send
OK!!! Server is ready.
Connected : 127.0.0.1:44636
Connected : 127.0.0.1:44638
Message Sent: 127.0.0.1:44636
Message Sent: 127.0.0.1:44638
Message Sent: 127.0.0.1:44636
Message Sent: 127.0.0.1:44638
Message Sent: 127.0.0.1:44636
Message Sent: 127.0.0.1:44638
Message Sent: 127.0.0.1:44636
Message Sent: 127.0.0.1:44638
Message Sent: 127.0.0.1:44636
LIST
[1] 127.0.0.1:44638 ACTIVE
[2] 127.0.0.1:44636 INACTIVE
2 user(s) on-line.
CLOSE
Sending notifications to clients...
Message Sent: 127.0.0.1:44636
Message Sent: 127.0.0.1:44638
Bye!
[root@yamakaze assignment1]#


root@yamakaze:~/eecs3214/assignment1
[root@yamakaze assignment1]# ./client.out
EECS3214 Assignment 1 Client
Usage: ./client.out [ip_address]
Command:
JOIN - display yourself on the LIST
LEAVE - hide yourself from the LIST
LIST - list all the users
CLOSE - close this program
message - anything you want to send
I am hosting the server-end on my server.
You can try 106.185.43.242.
Please enter a valid IP address:127.0.0.1
CONNECTED!!!
>127.0.0.1:44636 >> Hello
>127.0.0.1:44636 >> haha
>127.0.0.1:44636 >> can you hear me
>sure
MESSAGE SENT
>JOIN
JOINED FROM 127.0.0.1:44638
>System Announcement >> System is shutting down.
>connection lost
[root@yamakaze assignment1]#


root@yamakaze:~/eecs3214/assignment1
[root@yamakaze assignment1]# ./client.out
==
EECS3214 Assignment 1 Client

Usage: ./client.out [ip_address]
Command:
JOIN - display yourself on the LIST
LEAVE - hide yourself from the LIST
LIST - list all the users
CLOSE - close this program
message - anything you want to send
==
I am hosting the server-end on my server.
You can try 106.185.43.242.
==
Please enter a valid IP address:127.0.0.1
CONNECTED!!!
>Hello
MESSAGE SENT
>haha
MESSAGE SENT
>can you hear me
MESSAGE SENT
>127.0.0.1:44638 >> sure
>LIST
[1] 127.0.0.1:44638
1 user(s) on-line.
>System Announcement >> System is shutting down.
>connection lost
[root@yamakaze assignment1]#