

---

# **A Secure Person2Person (P2P) Micropayment System - Server User Manual**

IM 3010 Programming Assignment: Phase 02 Implementation

Brian Li-Hsuan Chen (B07705031)

December 20, 2021

## Contents

Introduction . . . . .	2
Environment . . . . .	2
macOS . . . . .	2
Ubuntu . . . . .	3
Usage . . . . .	4
Running Server Program . . . . .	4
Working Server . . . . .	5
Terminating Server Program . . . . .	10
Running Client Program . . . . .	11
Exiting Client Program . . . . .	11
How to Compile . . . . .	11
Compiling Server Only . . . . .	11
Compiling Client and Server . . . . .	12
References . . . . .	13
Server-side Implementation . . . . .	13
Client-side Implementation . . . . .	14
User Manual . . . . .	14

## Introduction

In **Phase 02**, we are asked to implement a server-side program to handle requests sent by clients in the Micropayment System. Functions for a server-side program include *registering*, *login*, *listing*, *transacting*, and *exiting*. Simply start running the program by `./server <SERVER_PORT> <CONCURRENT_USER_LIMIT>` after compilation (which can be done by `make server`).

`CONCURRENT_USER_LIMIT` is completely optional. The maximum is set to the number of concurrent threads supported by the implementation.

The user manual will cover the running environment used when developing the program, the environment that this code could be used in, the usage of the server-side program, the compilation, and the references when doing this assignment.

## Environment

### macOS

The environment used to develop this project is:

Operating System: macOS 12.0.1  
CPP Standard: C++17

It means that **you can run this program in a macOS environment** if the program is also compiled in the exact environment.

C++17 is used to serve the standard library header `filesystem` used when creating/deleting a database file.

I am using `sqlite3` to handle user profiles on *the backend*. By default, `sqlite3` is pre-installed in all versions of macOS<sup>1</sup>.

## Ubuntu

For the given `server` binary, you can run it on:

Operating System: Ubuntu 20.04  
CPP Standard: C++17

To compile, **you may need to install some extra dependencies/packages** on your system:

1. Install `sqlite3` (you can see why in the References section)

```
1 sudo apt install sqlite3
2 sudo apt-get install libsqlite3-dev
```

2. Make sure your GCC version is up-to-date (GCC 9-ish) to support C++17 (**please ignore this if you did not mess up with your environment**)

```
1 sudo add-apt-repository ppa:ubuntu-toolchain-r/test
2 sudo apt update
3 sudo apt install gcc-9 g++-9
4 sudo update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-9
   60 --slave /usr/bin/g++ g++ /usr/bin/g++-9 # to make sure gcc
   is using the latest version of GCC
```

I am testing out the Linux-formatted `server` binary on Karton from my MacBook. Karton is developed based on Docker containers. The code is also tested on Ubuntu 20.04 virtual box.

**Notice that the user interface requires Nerd Fonts to render.**

---

<sup>1</sup>How to install SQLite on macOS

## Usage

### TL;DR:

```
1 ./server <SERVER_PORT> <[OPTIONAL] CONCURRENT_USER_LIMIT>
```


```
1 ./client <SERVER_IP> <SERVER_PORT>
```

## Running Server Program

Before running the `client` program, you have to make sure that the `server` is running. You can start the server on port 8888 and limited to a maximum of three concurrent connected clients by running:

```
1 ./server 8888 3
```

Here is a look of the above command:

A screenshot of a macOS terminal window titled './server 8888 3'. The terminal shows the command './server 8888 3' being executed. The output is: 'User limit is now set to 3', 'Server listening on port 8888', 'Opened database successfully', and 'Client table created successfully'. The terminal has a dark background with light-colored text. At the top, there's a status bar with various icons and a progress indicator showing '5.11s' and '22:10:09'.

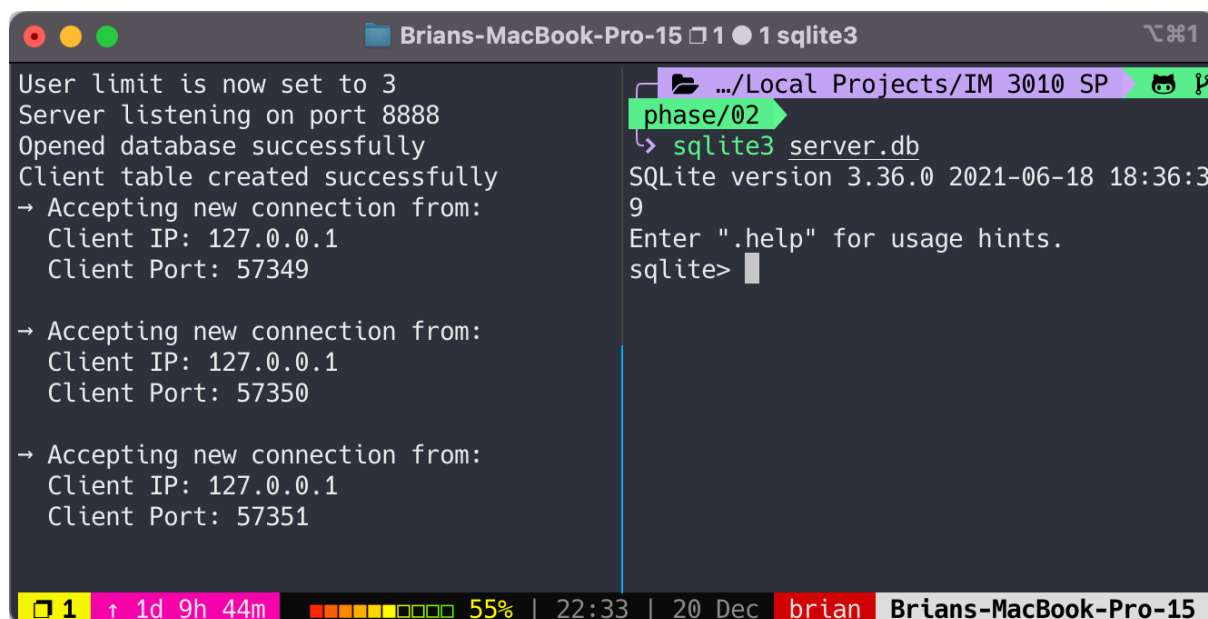
The basic usage is: `./server <SERVER_PORT> <CONCURRENT_USER_LIMIT>`.

And that is simply it. You can have a glimpse of what is going on in the server by peeking into the `server.db` file via `sqlite3` simply by typing in:

```
1 sqlite3 server.db
```

and you shall get access to the database with a table named `client` that stores all user data, including users' connection states.

So together with `sqlite3`, you will be able to gain access to the database and keep the server running at the same time:



```

Brians-MacBook-Pro-15 1 ● 1 sqlite3
User limit is now set to 3
Server listening on port 8888
Opened database successfully
Client table created successfully
→ Accepting new connection from:
  Client IP: 127.0.0.1
  Client Port: 57349

→ Accepting new connection from:
  Client IP: 127.0.0.1
  Client Port: 57350

→ Accepting new connection from:
  Client IP: 127.0.0.1
  Client Port: 57351

~/Local Projects/IM 3010 SP
phase/02
> sqlite3 server.db
SQLite version 3.36.0 2021-06-18 18:36:39
Enter ".help" for usage hints.
sqlite>
1 ↑ 1d 9h 44m 55% | 22:33 | 20 Dec brian Brians-MacBook-Pro-15
```

## Working Server

The server will always throw out `[<CLIENT_MSG>] from <CLIENT_PORT>@<CLIENT_PUBLIC_PORT>` when a message is received from a client.

**When Clients Connect to Server** When a client connects to the server, it will show IP address and port of that client.

The following screenshots show how three clients connect to the server listening on port 8888:



The image shows two terminal windows. The top window, titled `./server 8888 3`, displays the server's startup sequence and three incoming connections from `127.0.0.1`. The bottom window, titled `Brians-MacBook-Pro-15 0 ● 1 client`, shows three parallel client terminals, each running `./client 127.0.0.1 8888` and waiting for a connection.

```
./server 8888 3
User limit is now set to 3
Server listening on port 8888
Opened database successfully
Client table created successfully
→ Accepting new connection from:
  Client IP: 127.0.0.1
  Client Port: 57148

→ Accepting new connection from:
  Client IP: 127.0.0.1
  Client Port: 57149

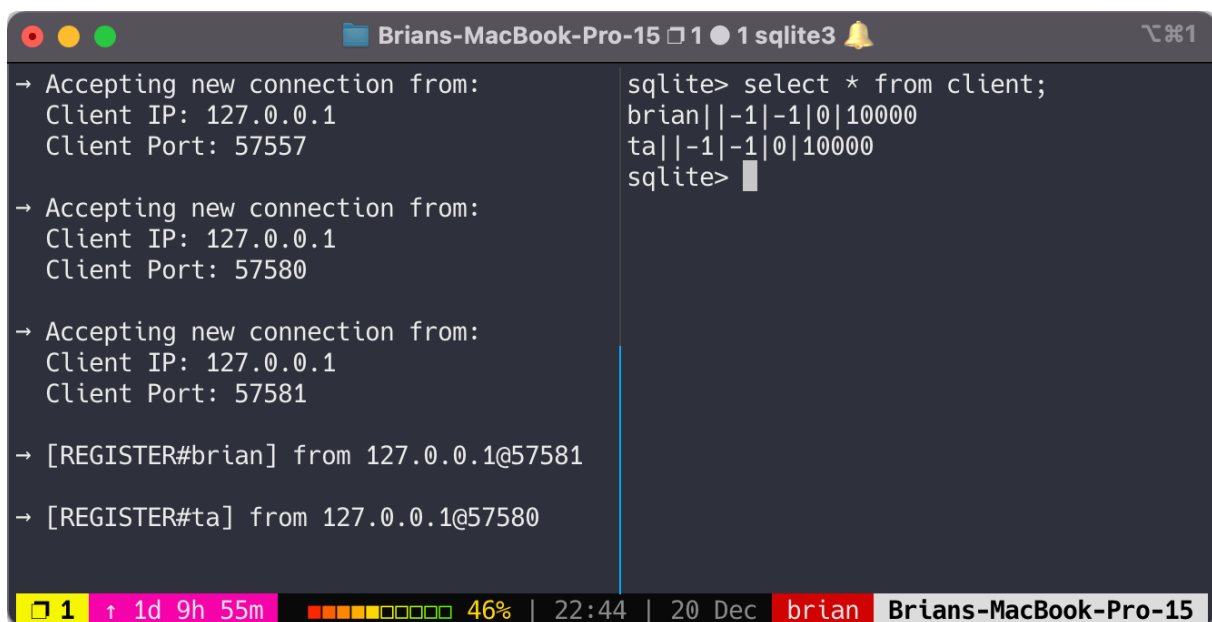
→ Accepting new connection from:
  Client IP: 127.0.0.1
  Client Port: 57151

phase/02
./client 127.0.0.1 8888
Connect to IP address: 127.0.0.1
On port: 8888
Press enter to continue...

phase/02
./client 127.0.0.1 8888
Connect to IP address: 127.0.0.1
On port: 8888
Press enter to continue...

phase/02
./client 127.0.0.1 8888
Connect to IP address: 127.0.0.1
On port: 8888
Press enter to continue...
```

**User Registration** When a user registers, you shall also see an update directly within the database.



```

Brians-MacBook-Pro-15 1 ● 1 sqlite3

→ Accepting new connection from:
Client IP: 127.0.0.1
Client Port: 57557

→ Accepting new connection from:
Client IP: 127.0.0.1
Client Port: 57580

→ Accepting new connection from:
Client IP: 127.0.0.1
Client Port: 57581

→ [REGISTER#brian] from 127.0.0.1@57581

→ [REGISTER#ta] from 127.0.0.1@57580

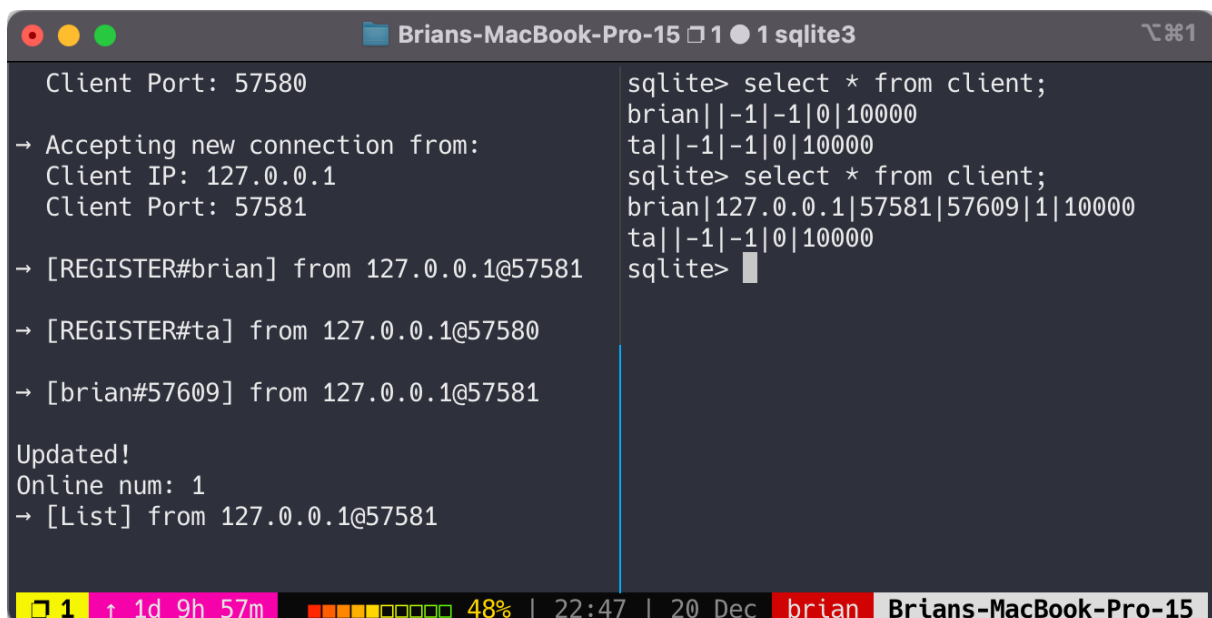
sqlite> select * from client;
brian|-1|-1|0|10000
ta|-1|-1|0|10000
sqlite>

```

1 ↑ 1d 9h 55m 46% | 22:44 | 20 Dec brian Brians-MacBook-Pro-15

If a user already registered, the server will return 210 `FAIL` to the client.

**User Login** You see that the server recorded IP address, public port, private port (the port specified by the logged in client as shown in `[brian#57609]`).



```

Brians-MacBook-Pro-15 1 ● 1 sqlite3

Client Port: 57580

→ Accepting new connection from:
Client IP: 127.0.0.1
Client Port: 57581

→ [REGISTER#brian] from 127.0.0.1@57581

→ [REGISTER#ta] from 127.0.0.1@57580

→ [brian#57609] from 127.0.0.1@57581

Updated!
Online num: 1
→ [List] from 127.0.0.1@57581

sqlite> select * from client;
brian|-1|-1|0|10000
ta|-1|-1|0|10000
sqlite> select * from client;
brian|127.0.0.1|57581|57609|1|10000
ta|-1|-1|0|10000
sqlite>

```

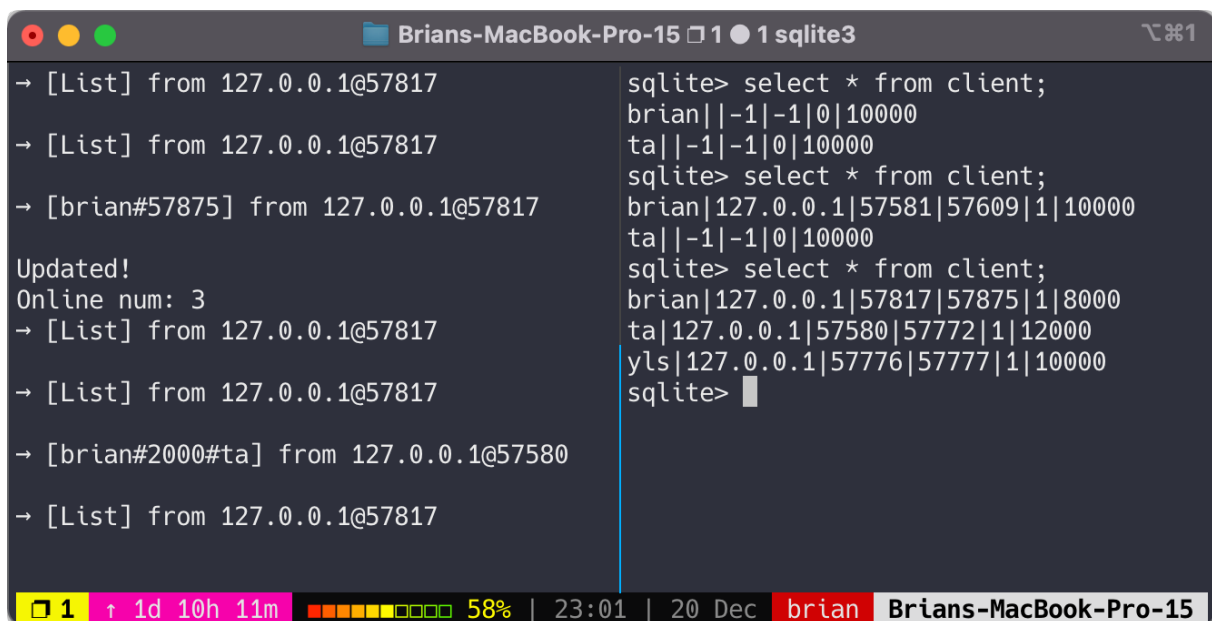
1 ↑ 1d 9h 57m 48% | 22:47 | 20 Dec brian Brians-MacBook-Pro-15

I will log in with three users on three clients in the rest of the demonstration.

**Information Listing** A user must log in before requesting for system information. A message `Please login first` will be sent to the client if the session is not logged in.

Otherwise, it will show the system information to the user as specified in the assignment specification.

**P2P Transaction** As can be seen below, the `[brian#2000#ta]` message indicates that user `brian` transfers 2000 dollars to `ta`.



```

Brians-MacBook-Pro-15 1 ● 1 sqlite3
→ [List] from 127.0.0.1@57817
→ [List] from 127.0.0.1@57817
→ [brian#57875] from 127.0.0.1@57817
Updated!
Online num: 3
→ [List] from 127.0.0.1@57817
→ [List] from 127.0.0.1@57817
→ [brian#2000#ta] from 127.0.0.1@57580
→ [List] from 127.0.0.1@57817

sqlite> select * from client;
brian|-1|-1|0|10000
ta|-1|-1|0|10000
sqlite> select * from client;
brian|127.0.0.1|57581|57609|1|10000
ta|-1|-1|0|10000
sqlite> select * from client;
brian|127.0.0.1|57817|57875|1|8000
ta|127.0.0.1|57580|57772|1|12000
yls|127.0.0.1|57776|57777|1|10000
sqlite>

```

1 ↑ 1d 10h 11m 58% | 23:01 | 20 Dec brian Brians-MacBook-Pro-15

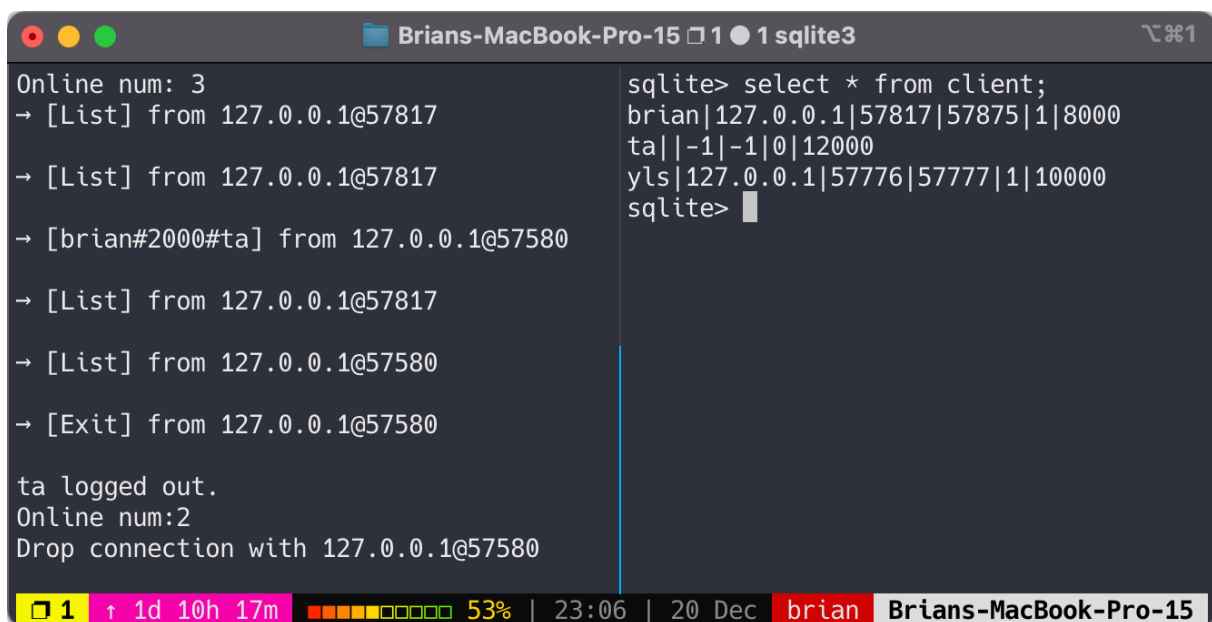
You can also see that the database is updated accordingly.

**User Logout** There might be three cases:

**When a user logs out:**

The server will prompt *who* logs out.





```

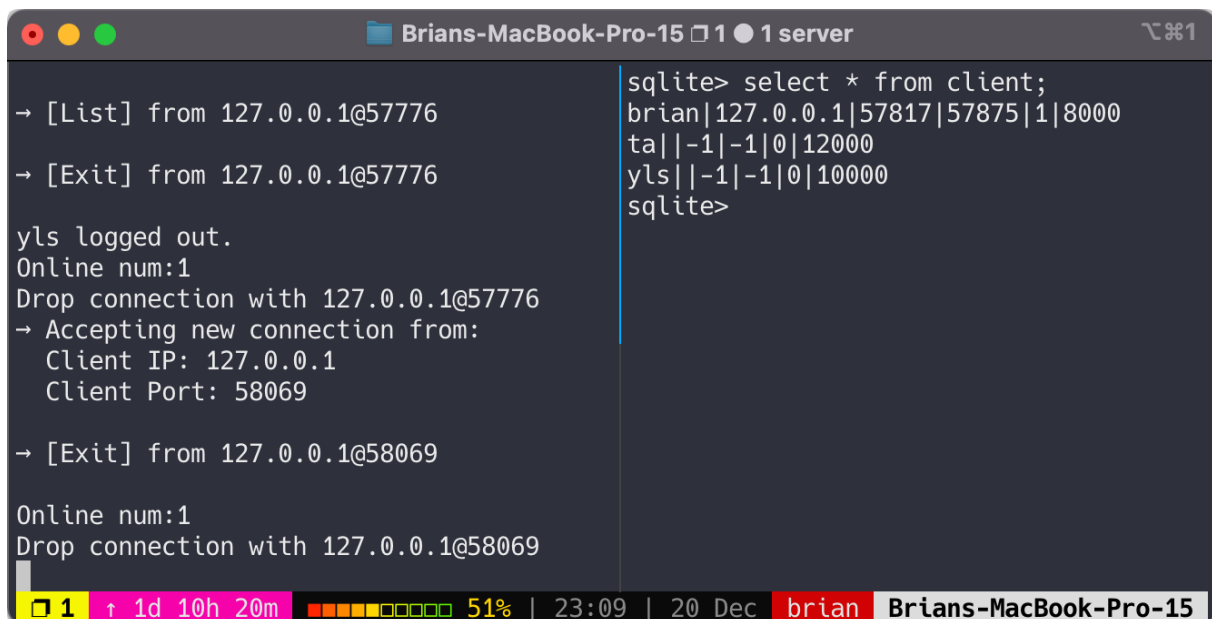
Online num: 3
→ [List] from 127.0.0.1@57817
→ [List] from 127.0.0.1@57817
→ [brian#2000#ta] from 127.0.0.1@57580
→ [List] from 127.0.0.1@57817
→ [List] from 127.0.0.1@57580
→ [Exit] from 127.0.0.1@57580

ta logged out.
Online num:2
Drop connection with 127.0.0.1@57580

sqlite> select * from client;
brian|127.0.0.1|57817|57875|1|8000
ta||-1|-1|0|12000
yls|127.0.0.1|57776|57777|1|10000
sqlite>

```

**When a client exits properly:**



```

→ [List] from 127.0.0.1@57776
→ [Exit] from 127.0.0.1@57776

yls logged out.
Online num:1
Drop connection with 127.0.0.1@57776
→ Accepting new connection from:
  Client IP: 127.0.0.1
  Client Port: 58069

→ [Exit] from 127.0.0.1@58069

Online num:1
Drop connection with 127.0.0.1@58069

sqlite> select * from client;
brian|127.0.0.1|57817|57875|1|8000
ta||-1|-1|0|12000
yls||-1|-1|0|10000
sqlite>

```

**When a client terminates suddenly (and returns a `SIGPIPE`):**

This is not really possible since the client is well implemented. But if it does, server will handle:

```

Brians-MacBook-Pro-15 1 ● 1 sqlite3
→ [Exit] from 127.0.0.1@58564
ta logged out.
Online num:1
Drop connection with 127.0.0.1@58564
→ [List] from 127.0.0.1@58562
→ [List] from 127.0.0.1@58562

Caught signal 13
Something is written to a pipe where nothing is read from anymore. A user may just be gone.
brian logged out.
Online num:0
Drop connection with 127.0.0.1@58562

sqlite> select * from client;
brian|127.0.0.1|58562|58563|1|8000
ta||-1|-1|0|12000
yls||-1|-1|0|10000
sqlite> select * from client;
brian||-1|-1|0|8000
ta||-1|-1|0|12000
yls||-1|-1|0|10000
sqlite>

```

1 ↑ 1d 10h 43m 38% | 23:32 | 20 Dec brian Brians-MacBook-Pro-15

### Terminating Server Program

To terminate the server, you will have to **CTRL + C** while the server is running. Signal handling is implemented so that you can do it safely.

The termination of the server will also cause the deletion of the db file `server.db`. The settings for not deleting the database can only be modified from the source code at `src/Database.cpp`.

```

Brians-MacBook-Pro-15 1 ● 1 zsh
→ [List] from 127.0.0.1@58190
Drop connection with 127.0.0.1@58190
Caught signal 13
Something is written to a pipe where nothing is read from anymore. A user may just be gone.
→ [Exit] from 127.0.0.1@57817

brian logged out.
Online num:1
Drop connection with 127.0.0.1@57817
^C
Caught signal 2
Database server.db deleted successfully.

phase/02

```

1 ↑ 1d 10h 30m 44% | 23:19 | 20 Dec brian Brians-MacBook-Pro-15

## Running Client Program

For the client part, I won't show the screenshot in this documentation again; you can see the demos/screenshots in *Phase 01 User Manual*.

If you are testing out the client program on your localhost:

```
1 ./client 127.0.0.1 8888
```

Otherwise, you will have to figure out the right server IP address and the port the server is listening on.

The basic usage is: `./client <SERVER_IP> <SERVER_PORT>`.

## Exiting Client Program

You can exit a client using two methods:

1. Typing in 5 to *properly tell the server you are to exit* and quit the session peacefully, or
2. Hitting `CTRL + C` to forcefully terminate the session; `server` will handle the error accordingly.

## How to Compile

**Notice that you will need to have the dependencies installed first if you are using Linux.** Please [click here](#) if you miss the part.

### Compiling Server Only

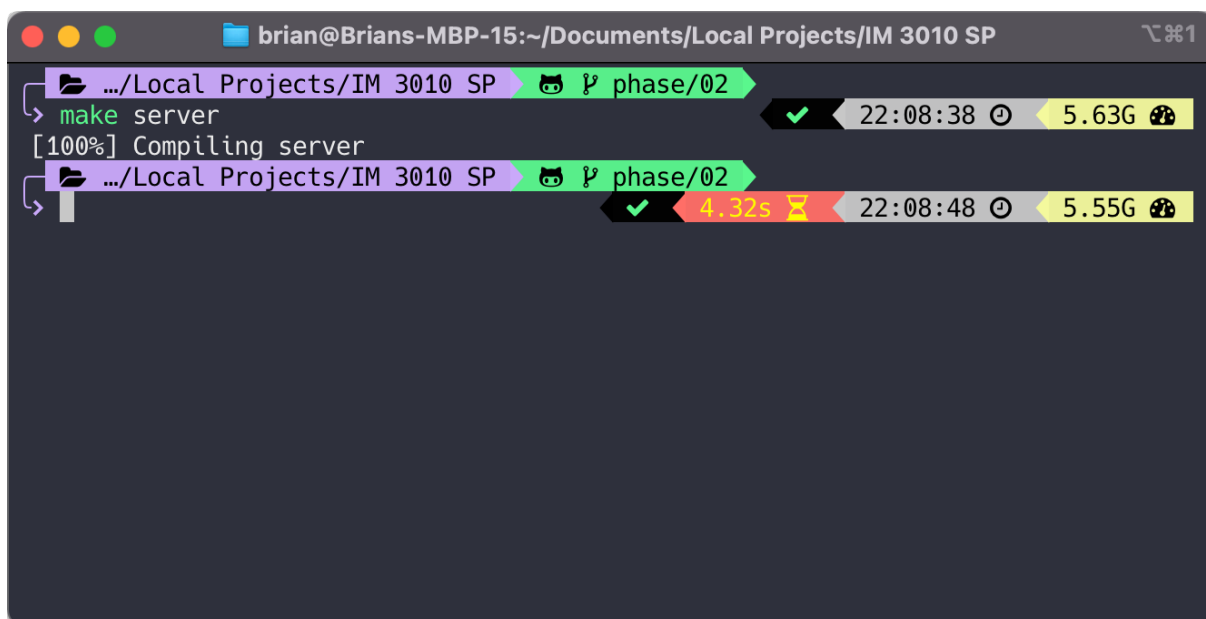
You can start the program already by typing `./server <SERVER_PORT> <CONCURRENT_USER_LIMIT>` into your terminal if you are on a Linux distribution (*Ubuntu* is used for testing) or on a macOS.

To rebuild the program, on either **macOS** or a **Linux** system, make sure to install the dependencies first before you run `make server` in the terminal app.

It will take a bit longer to compile `server` as compared to `client` since it is linking much more powerful and bigger libraries (`-sqlite3` for using database).

If `server` binary already exists, you may want to run `make clean` first to remove the file.

If done properly, you should see the following output:

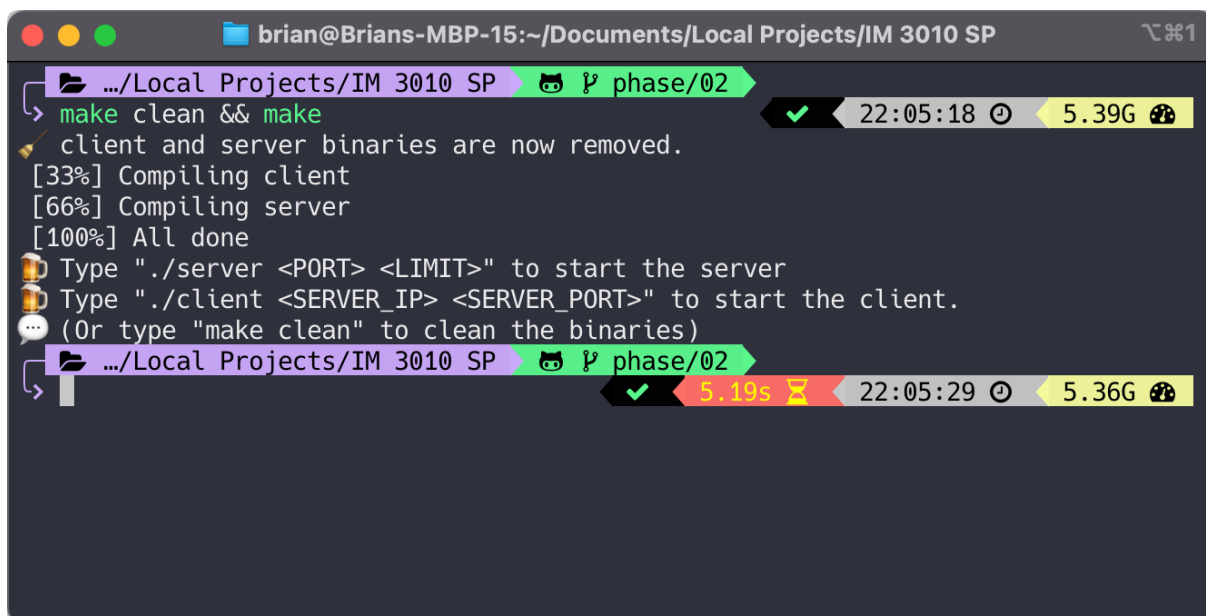


## Compiling Client and Server

You can simply start off by doing:

```
1 make clean && make
```

If no errors occur, you shall see the following output on your terminal:



## References

### Server-side Implementation

- ☒ Creation of a **database** for handling multiple input and querying

Background: As far as I know, handling simultaneous reads and writes can be a hassle when implemented manually. I consulted to my friends studying CSIE, and they also believe using a database could be a more practical and reasonable way to implement such querying function.

I looked it up and find out that `sqlite` integrates so well with C/C++. `sqlite` can drive a db of up to 140TB, allows multiple simultaneous reads and, like other databases stores data in files on disk.

Though for our use case, the need for a database is unnecessary due to the fact that we are only opening to **few users** (3) at a time, I still feel this urge to learn how to implement one for this project.

- ☒ Deletion of the database (`*.db`)

- Filesystem Library in C++17 at <https://stackoverflow.com/a/59424074/10871988>

- ☒ More on `sqlite` C++

- [https://github.com/fnc12/sqlite\\_orm](https://github.com/fnc12/sqlite_orm) → I am using this
  - <https://www.runoob.com/sqlite/sqlite-c-cpp.html>

- ☒ Thread and Worker Pool

- <https://ncona.com/2019/05/using-thread-pools-in-cpp/> - a very good article explaining how to use thread pools
  - <https://stackoverflow.com/questions/15752659/thread-pooling-in-c11>
  - <https://stackoverflow.com/questions/48943929/killing-thread-from-another-thread-c>
- ☒ <https://github.com/vit-vit/ctpl> → I am using this

- ☒ Handling `SIGINT`

- <https://stackoverflow.com/questions/1641182/how-can-i-catch-a-ctrl-c-event>

- ☒ `TIME_WAIT`

Background: `server` could not close connection after the socket is closed:

```
1 sudo netstat -tanl | grep 8888
```

```
1 tcp4      0      0      127.0.0.1.64480      127.0.0.1.8888
    TIME_WAIT
```

<https://stackoverflow.com/questions/23915304/how-to-avoid-time-wait-for-server-sockets>

☒ Catch SIGPIPE from sudden death of a client

- <https://stackoverflow.com/questions/61688091/catching-client-exit-from-server-on-socket-programing>
- <https://stackoverflow.com/questions/26752649/so-nosigpipe-was-not-declared>
- <https://stackoverflow.com/questions/18935446/program-received-signal-sigpipe-broken-pipe/18963142>

## Client-side Implementation

(from phase01)

- Repositories
  - Learn Network Protocol and Programming Using C at <https://github.com/apsrcreatix/Socket-Programming-With-C>
  - Peer to peer program in C at <https://github.com/um4ng-tiw/Peer-to-Peer-Socket-C>
  - C Multithreaded Client-Server at <https://github.com/RedAndBlueEraser/c-multithreaded-client-server>
  - Socket programming examples in C++ at <https://github.com/zappala/socket-programming-examples-c>
- Others
  - Parse (split) a string in C++ using string delimiter (standard C++) at <https://stackoverflow.com/a/14266139/10871988>
  - Finding Unused Port in C++ at <https://stackoverflow.com/a/1107242/10871988>
  - Unix Specification (link to `bind()`) at <https://pubs.opengroup.org/onlinepubs/007908799/xns/bind.html>, but of course many more functions are looked up
  - Port Forwarding for a Docker Container at <https://docs.docker.com/config/containers/container-networking/>
  - Karton for not running on virtual machine at <https://karton.github.io>

## User Manual

(from phase01)

- Eisvogel at <https://github.com/Wandmalfarbe/pandoc-latex-template>