A Secure Person2Person (P2P) Micropayment System w/ OpenSSL User Manual

IM 3010 Programming Assignment: Phase 03 Implementation

Brian Li-Hsuan Chen (B07705031)

January 18, 2022

Contents

Introduction

In **Phase 03**, we are asked to implement a secure transmission for both server-side and client-side programs in the Micropayment System.

There are five functions in the system, i.e. *registering*, *login*, *listing*, *transacting*, and *exiting*. Asymmetric encryption is asked to be used in *transaction* (*transacting*), implemented by OpenSSL with public key and private key encryption and decryption.

The implementation in this repository has adopted asymmetric encryption for every transmission between client and server. Peer-to-peer communication is also implemented using asymmetric encryption. Client program will generate a pair of certificate and private key during runtime.

Simply start the server by running `./server <SERVER_PORT> <CONCURRENT_USER_LIMIT> [--silent]`. `-s` or `--silent` is passed in to suppress the output of the encrypted message in server.

Simply start the client program by running `./client <SERVER_IP> <SERVER_PORT> [--verbose]`. `-v` or `--verbose` is passed in to print the encrypted message in client.

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 server and client program, the compilation, the mechanism for secure transmission, 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[1].

You may need to install OpenSSL additionally:

```
brew install openssl
```

By default, `openssl` is installed in the `/usr/local/opt/openssl` directory. If you use `brew install openssl`, you can use `openssl@3` instead of `openssl@1.1` (with deprecation warnings).

When you run your code, you' ll have to include the path manually as in `OPENSSLCPPFLAGS` and `LDFLAGS`:

```
INCLUDE = -I$(WORKDIR)/include/\
                    -I$(WORKDIR)/src/

OPENSSLCPPFLAGS = -I/usr/local/opt/openssl@1.1/include
LDFLAGS = -L/usr/local/opt/openssl@1.1/lib
CFLAGS=-lstdc++ -lpthread -lsqlite3 $(LDFLAGS) $(INCLUDE) $(
    OPENSSLCPPFLAGS) -lssl -lcrypto

```

[1] How to install SQLite on macOS

```
8  gcc -std=c++17 yourprogram.cpp $(CFLAGS) -o yourprogram
```

Same implementation can be seen in `Makefile`.

Ubuntu

For the given `client` and `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. Install `openssl`

   ```
   1  sudo apt-get install openssl
   2  sudo apt-get install libssl-dev
   ```

   Installing `openssl` alone is not enough. You also need to install `libssl-dev` to compile both files (as in my case).

3. 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 `client` and `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. **You may encounter strange behavior occasionally on Linux**, since the code is mostly tested on macOS.

Notice that the user interface requires Nerd Fonts to render.

Certificates and Private Keys

By default, you shall see a `certs/` directory listed in the root directory of the project. This directory contains the certificates and private keys used for the program so that you do not need to regenerate them on the first run.

Since client's certificate and private keys will be generated in runtime, you'll only have to generate server's certificate and private key. The server's certificate and private key will be generated in the `certs/` directory.

Use the following command if `server.crt` and `server.key` are missing in the `certs/` directory:

```
sh create_ca.sh cert server server # this will generate server.crt and server.key using the configuration file
sh create_ca.sh CA # this will generate a PEM file containing listed certificates
```

Usage

TL;DR (assuming you don't delete the `certs/` directory):

```
./server <SERVER_PORT> [CONCURRENT_USER_LIMIT] [--silent]
```

```
./client <SERVER_IP> <SERVER_PORT> [--verbose]
```

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:

```
./server 8888 3
```

By default, the server should print everything verbosely. That is, it should print the encrypted message (in binaries, not readable) received from client(s). To suppress the encrypted message, you can pass in `--silent` or `-s` at the end:

Here is a look of the above command:

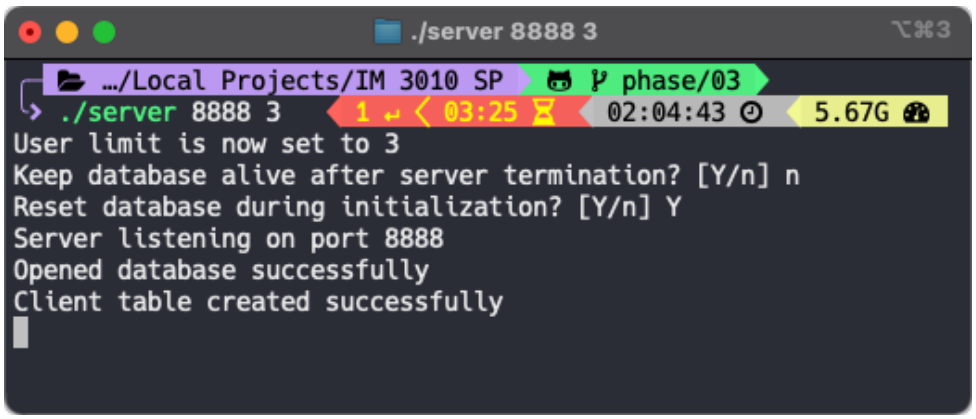


Figure 1: Without silent mode

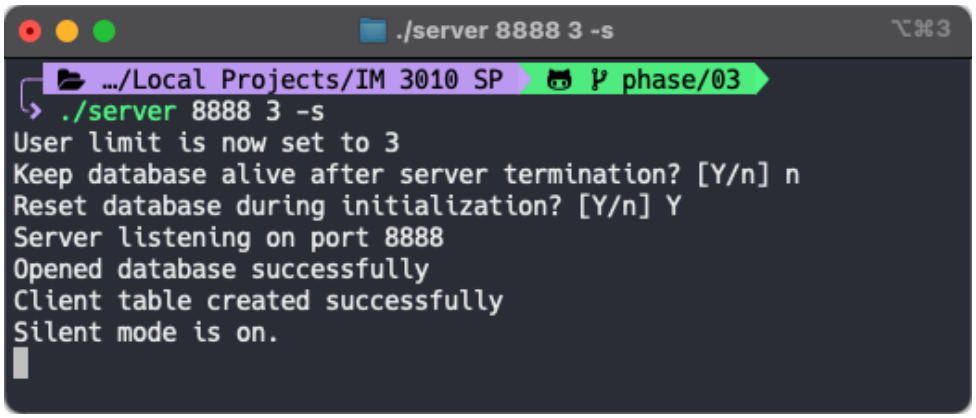


Figure 2: With silent mode on

The basic usage is: `./server <SERVER_PORT> [CONCURRENT_USER_LIMIT] [--silent]`.

And that is simply it. After running the server, you can choose to keep the database alive and whether it should reset the database with the prompt.

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:

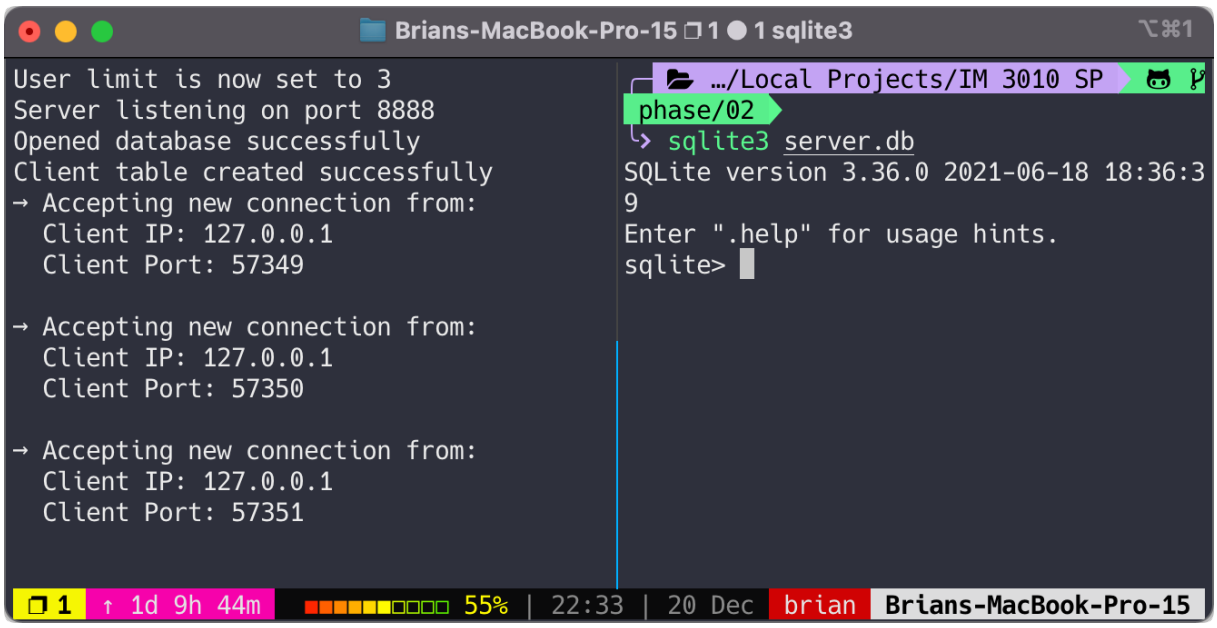


Figure 3: Same concept as Phase 02' s server

Working Server

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

If not in silent mode, the server will first show the encrypted message (not readble, just for clarification):

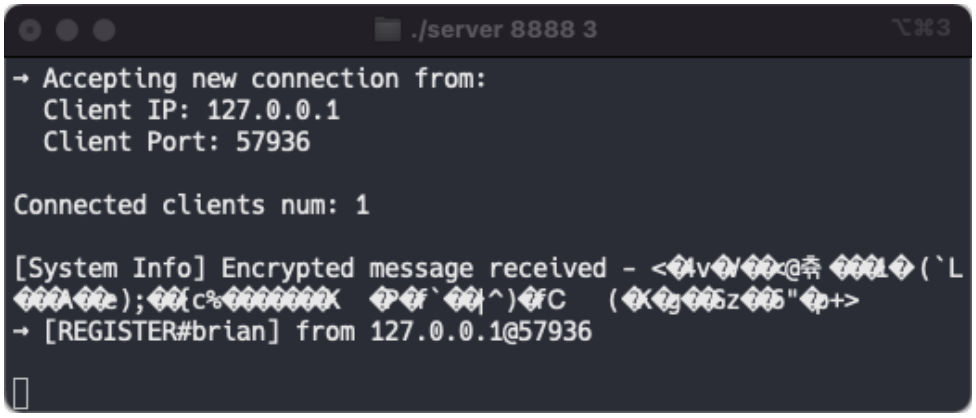
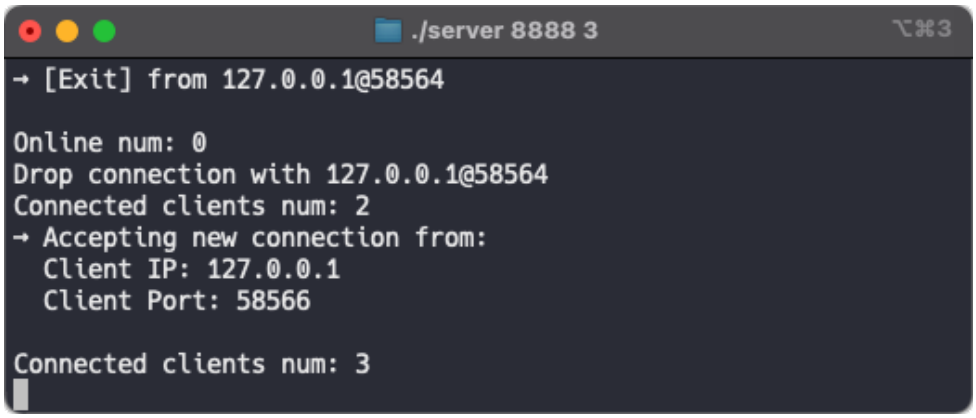


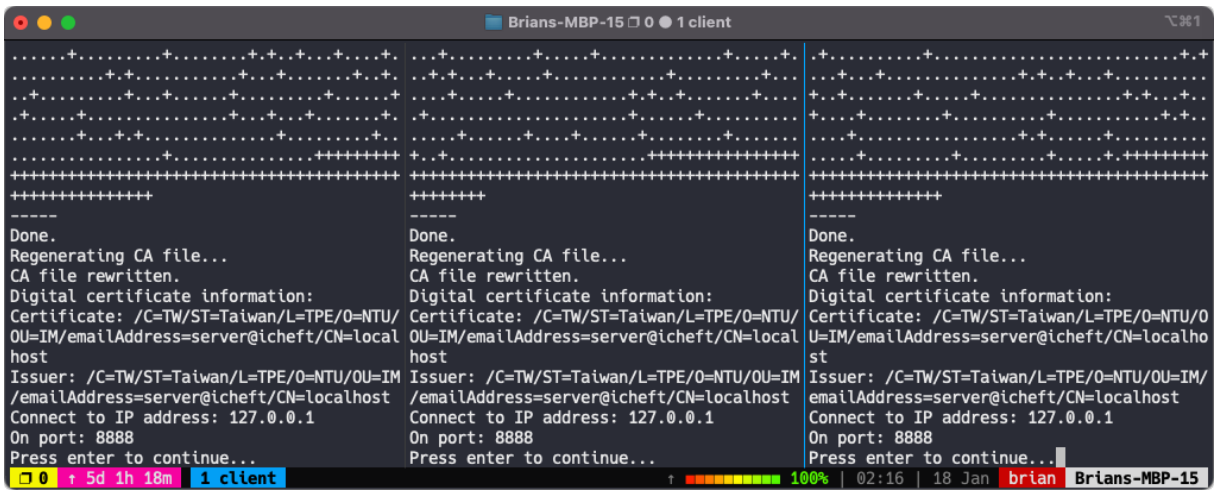
Figure 4: Server in default mode receiving [REGISTER#brian] from a client

You should see different encrypted message.

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 idea is pretty similar to Phase 02 implementation.

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

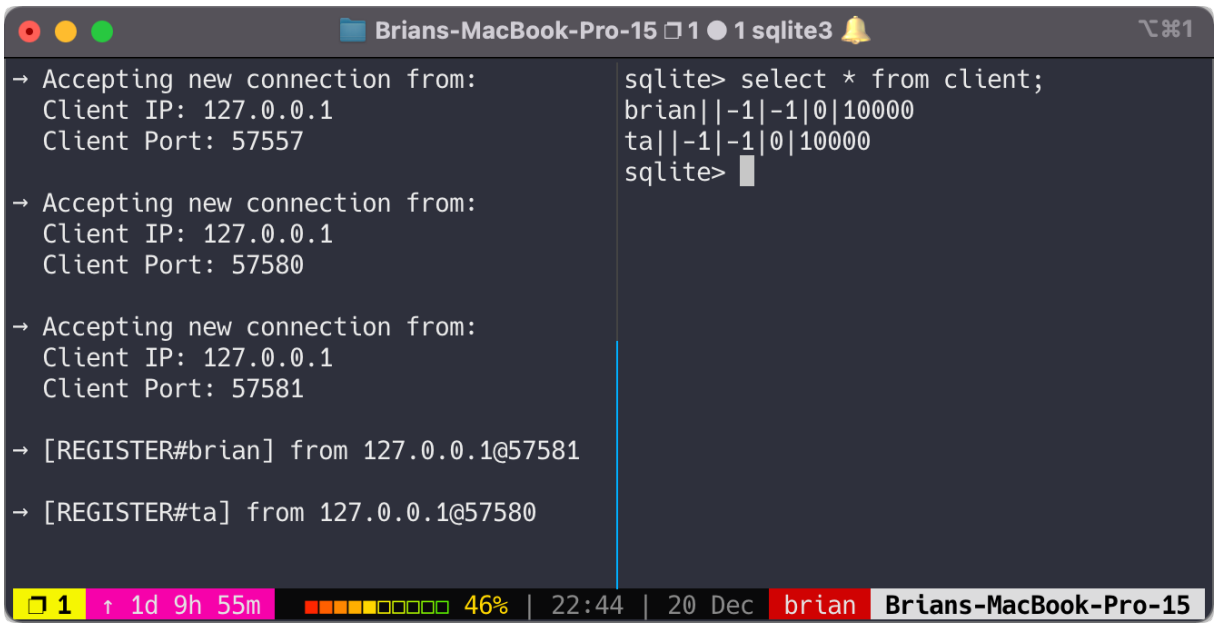


Figure 5: Same as the Phase 02 implementation

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]`).

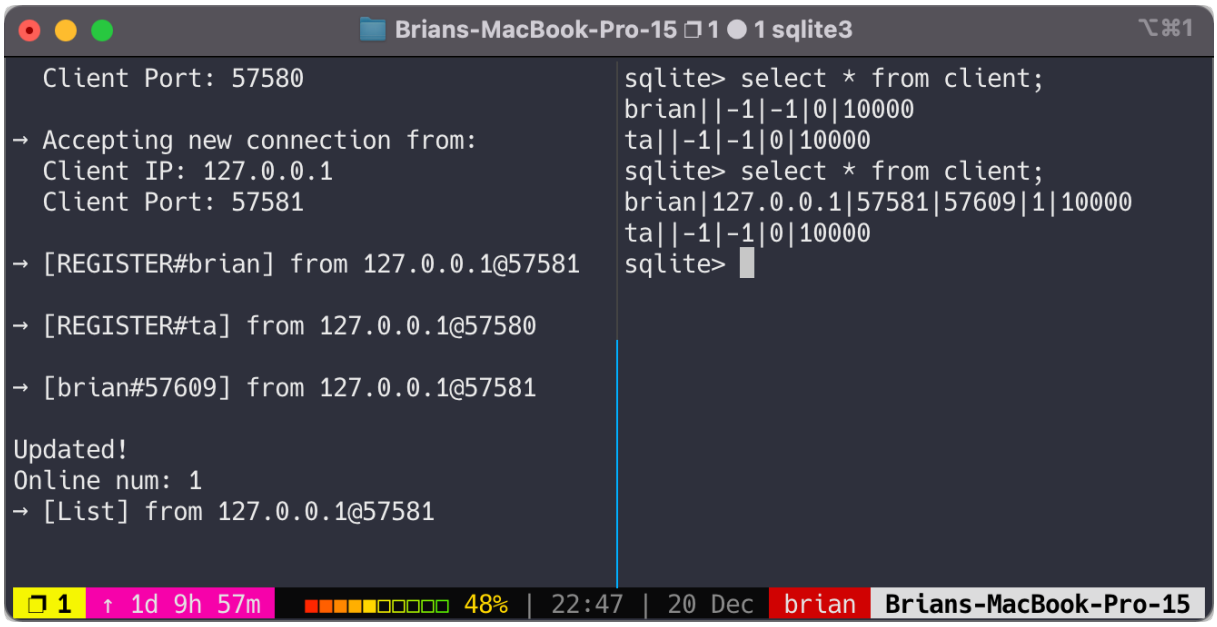


Figure 6: Same as the Phase 02 implementation

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`.

Before the transaction, the server will receive a `TRANSACTION` command from a client, this is due to the fact that the real transaction message (e.g. `[brian#2000#ta]`) can be implemented with higher form of encryption.

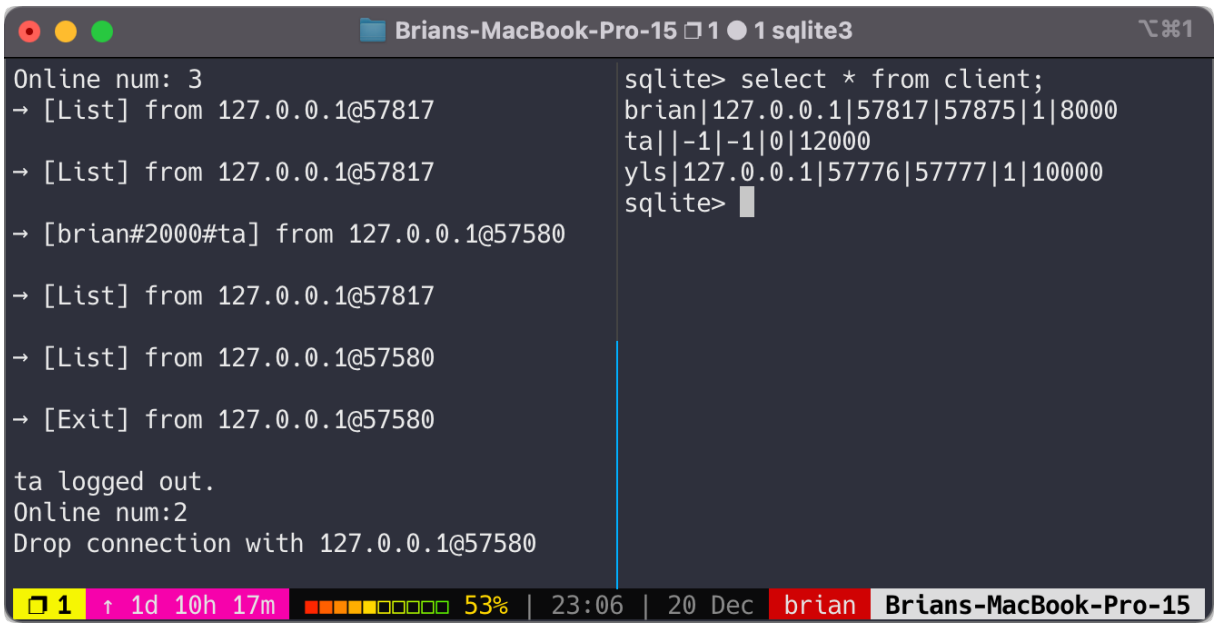
Simply put, the server will receive two messages from the same client when a P2P transaction occurs.

User Logout This is the same as shown in Phase 02 implementation.

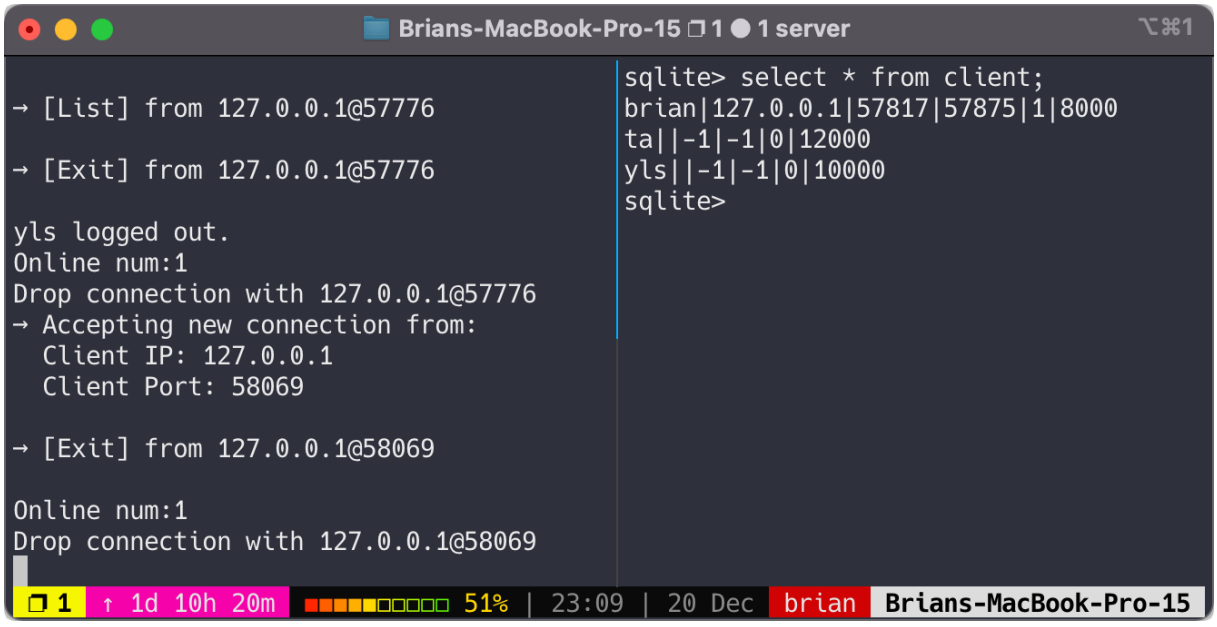
There might be three cases:

When a user logs out:

The server will prompt *who* logs out.

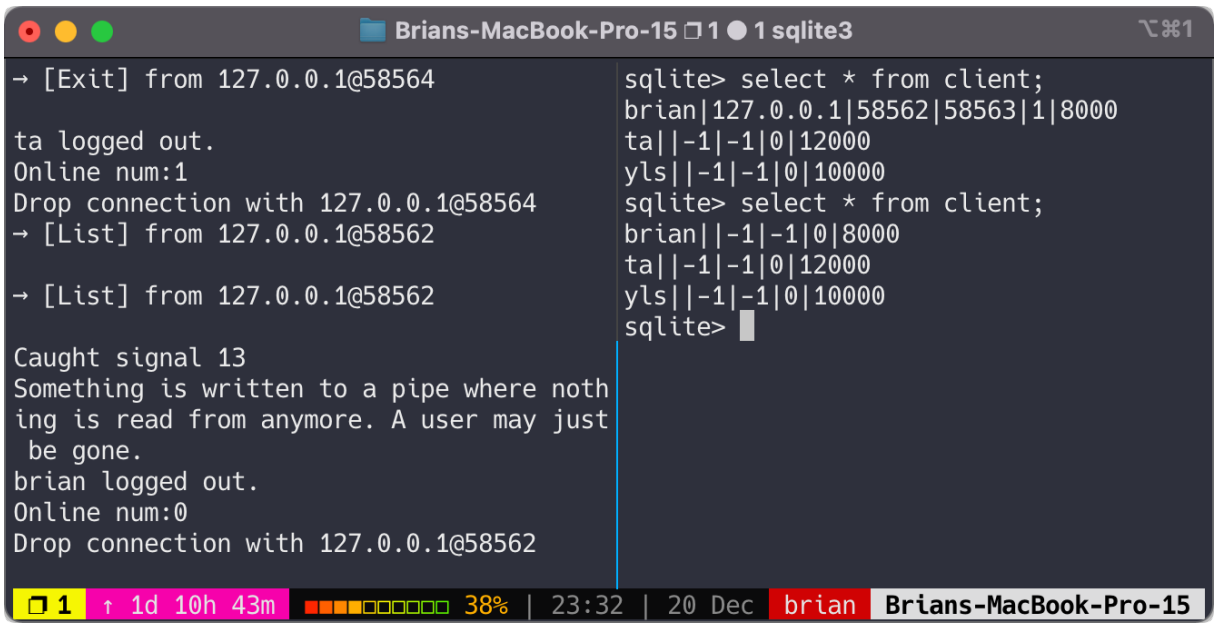


When a client exits properly:



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:



When Clients Exceed Thread Limit The server will notice and soon prompt that it has now detected more clients that it can handle. The client that triggers this action will have to wait until some other online users exit the server. The client will expect to see some "waiting".

This part has been properly shown in the demo session.

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` if you set the `Keep database alive` setting to `n` at the beginning of the process.

Running Client Program

For the client part, you can see most of 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 [-v]
```

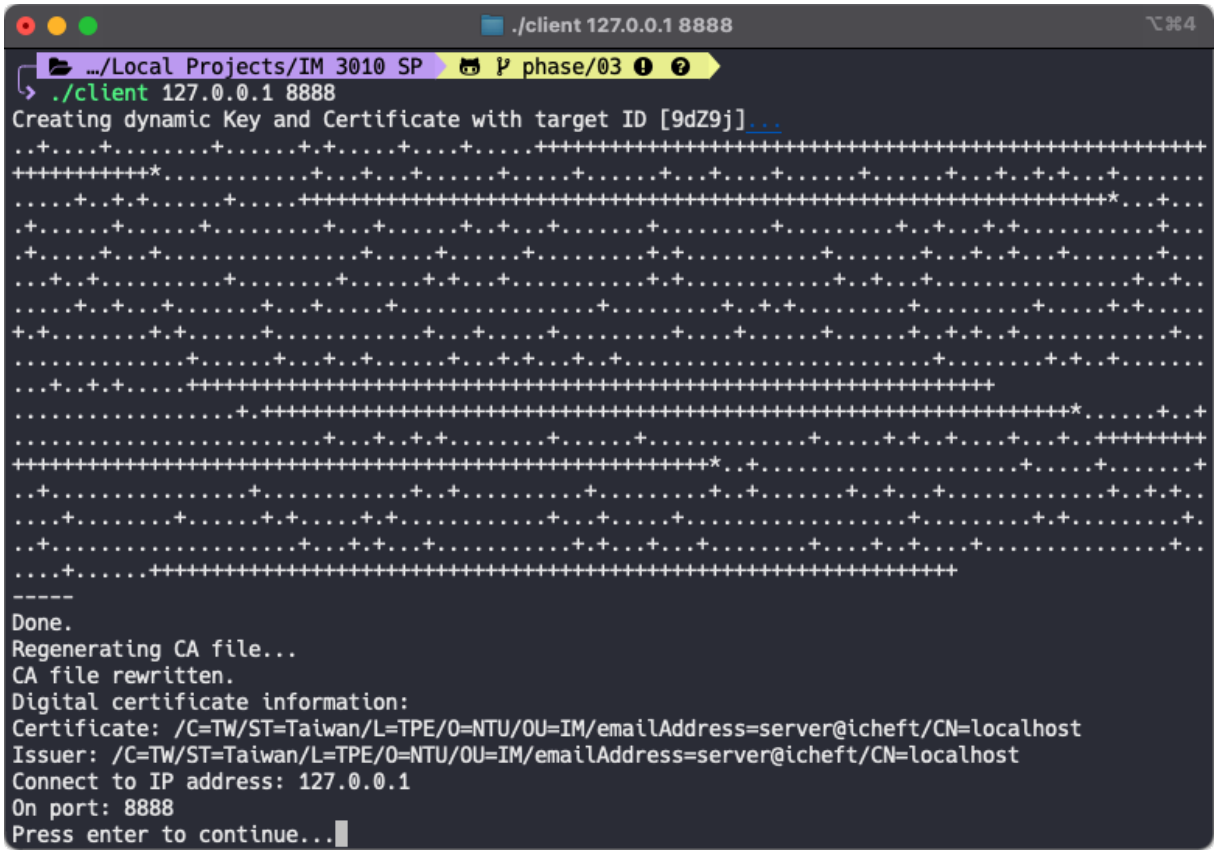
Otherwise, you will have to figure out the right server IP address and the port the server is listening on.

The basic usage is: `./client <SERVER_IP> <SERVER_PORT> [--verbose]`.

`-v` or `--verbose` will show the encrypted message. Since we do not want background information on the client side, by default, the client will not show the encrypted message.

Runtime Key Generation

When you start a client program, by default, it will generate a pair of certificate and private key at runtime.



It will generate a random key pair and store the certificate and key under that key.

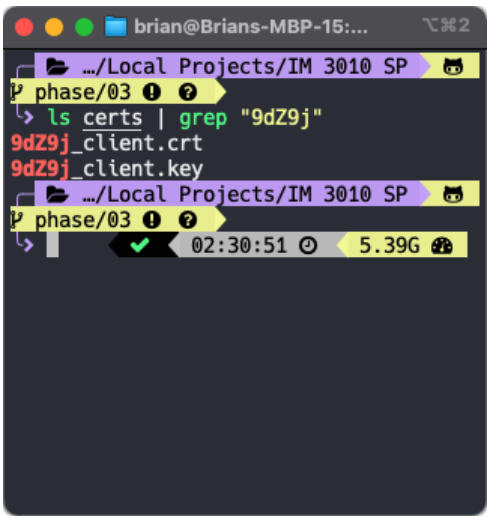


Figure 7: A pair of crt and key will be generated under `certs` directory

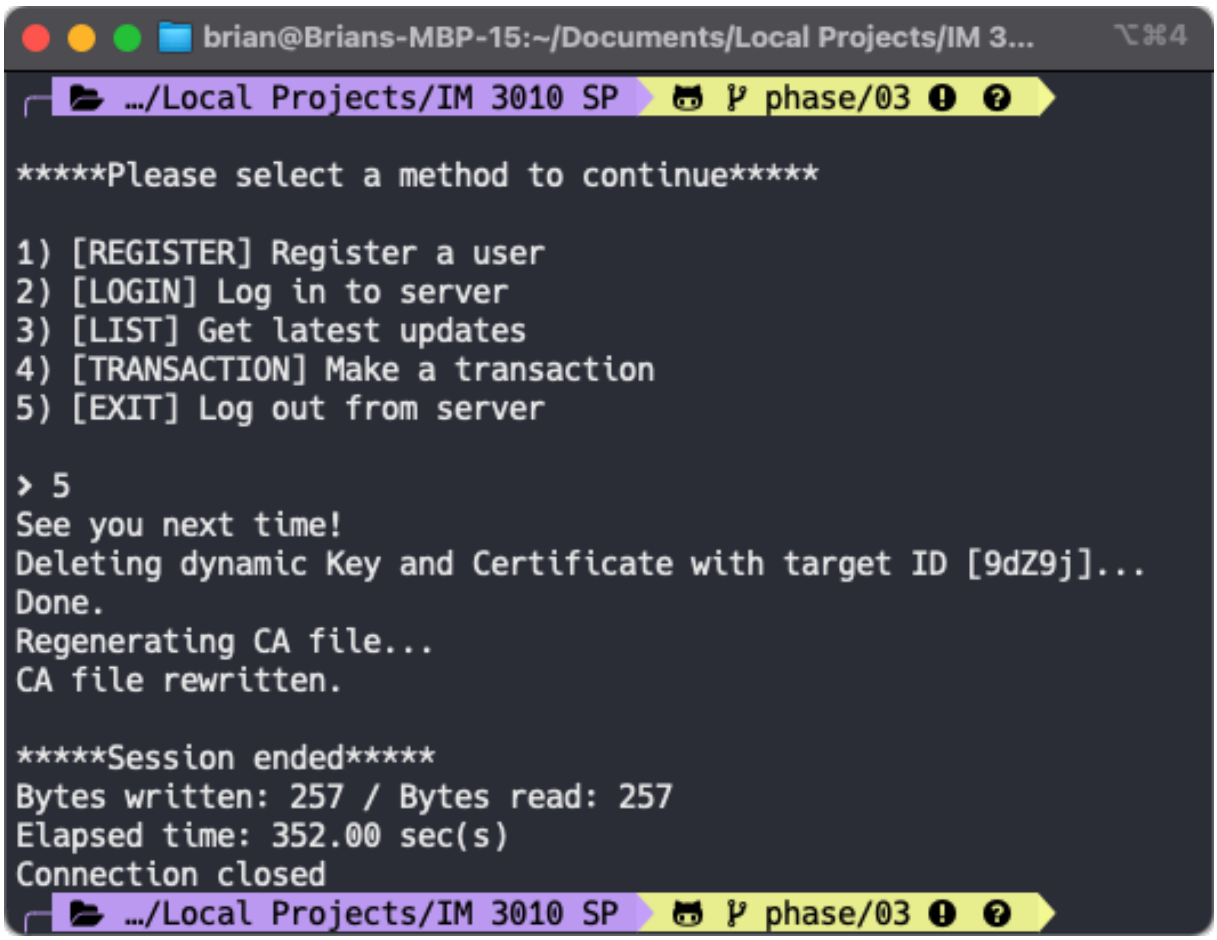
The `CA.pem` file will also be update using the example given from the OpenSSL official documentation.

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.

When you exit the client program, the generated key and certificate will the specific ID will be deleted. `CA.pem` file will also be updated.



How to Compile

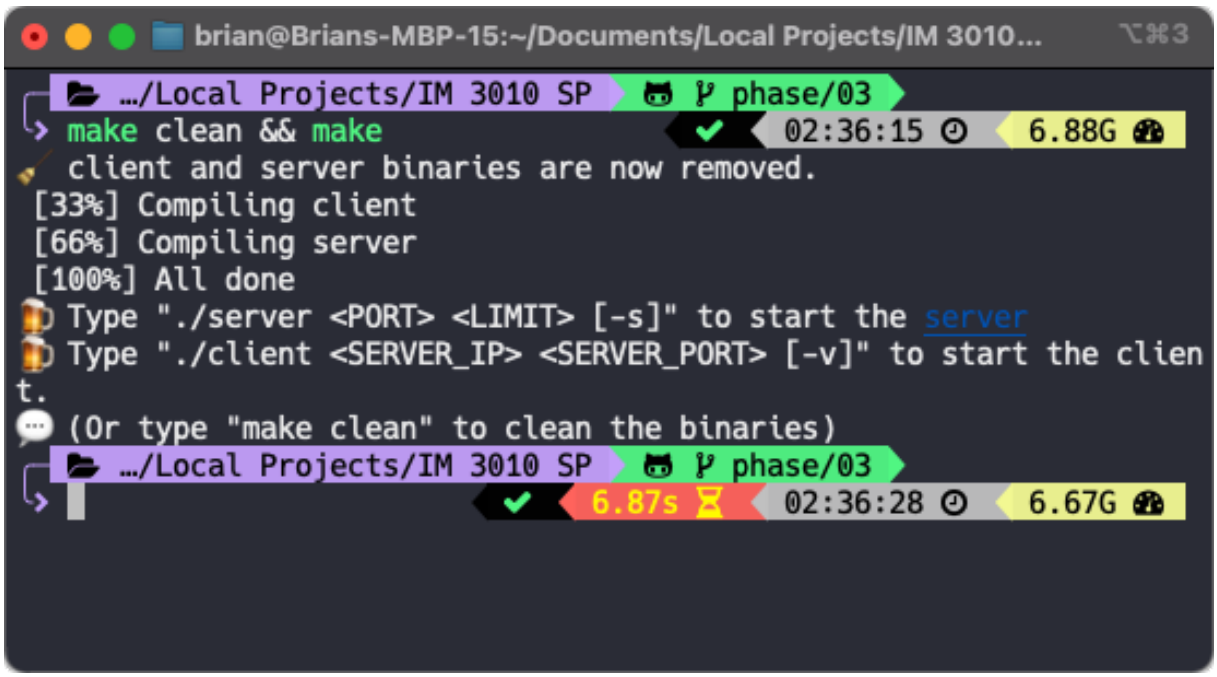
Notice that you will need to have the dependencies installed first if you are using Linux. Please click here if you miss the part.

Compiling Client and Server

You can simply start off by doing:

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

You shall see the following output on your terminal:



Debugging Mode If strange behaviors occur during runtime, you can use `make debug` to examine the program(s). Usually errors (encryption fails or decryption fails) will occur at client-side. Please remember the cert ID for the failed client, and set the `UID_TEST` macro to that ID.

Run the client program by changing the `./client` to `./client_d` and `./server` to `./server_d`.

Mechanism for Secure Transmission

Say `alice pays 2000 to bob`, the transaction can be divided into following steps:

1. alice generates transaction notification message `[alice#2000#bob]`
2. alice encrypts the message with her own private key and sends the 256-byte ciphertext to bob
3. bob receives the encrypted message, he will use alice's public key to decrypt the message
4. bob now has the original message, he will now notify the server an incoming transaction message by sending a `TRANSACTION` message encrypted with his private key (*can be implemented to a less strict encryption than asymmetric encryption*)
5. bob will then use his own private key to encrypt the original transaction message and sends another 256-byte ciphertext to the server
6. server receives the encrypted message, it then decrypted the message with bob's public key (server will not have to check that the message is from alice to bob; bob has already checked that by decrypting the message with alice's public key)
7. server will then return the transfer status message to alice encrypted with server's private key

8. when alice receives the encrypted message, it can decrypt with server’s pubilc key

In general, the entire system is designed in the same fashion.

References

OpenSSL

- SSL Socket:
  - OpenSSL Official Documentation https://www.openssl.org/docs/manmaster/man3/
  - SSL socket 小攻略
  - SSL 通道小攻略
  - https://www.ibm.com/docs/en/ztpf/2020?topic=apis-ssl-ctx-load-verify-locations
  - https://stackoverflow.com/questions/28366384/troubles-with-ssl-ctx-load-verify-locations → a client cannot use same configuration file for its key and cert pair
- Installation
  - 编译 Linux 内核时出现 fatal error: openssl/opensslv.h 解决的办法 https://zhuanlan.zhihu.com/p/61636004
- Key Generation
  - https://stackoverflow.com/questions/10175812/how-to-generate-a-self-signed-ssl-certificate-using-openssl
  - https://docs.oracle.com/cd/E65459_01/admin.1112/e65449/content/admin_csr.html
  - https://www.ibm.com/docs/en/ztpf/1.1.0.15?topic=gssccr-configuration-file-generating-self-signed-certificates-certificate-requests

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 → 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:

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

```
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