Computer Networks - Programming Assignment

Part 2: Server-side Program Operation Explanation Document

1. Source Code is located at `ARCHIVE_ROOT/src/`. The client source code and Makefile entries are still included.
2. 操作說明文件 PDF 檔 is the document you are reading right now
3. Binary 執行檔 is located at `ARCHIVE_ROOT/build/`
4. 用以編譯程式之 Makefile is located at `ARCHIVE_ROOT/Makefile`

1. compilation

Prerequisite:

`make`

Run the following command if `make` is not installed on your system.

```
sudo apt update && sudo apt install make
```

Install dependencies:

`gcc`, `build-essential`, `libgtkmm-3.0-dev`

Run the following command in the same directory as the `Makefile` (source code root) to install the required packages.
You may need to install the `libgtkmm-3.0-dev` package regardless of whether you are building the application yourself or are planning to run the precompiled binaries.

```
make install-deps
```

Compilation:

Run the following command in the same directory as the `Makefile` (source code root) to compile the server-side program to `build/server`.
The application is written in C++ for the Linux platform, using the `gtkmm` library.

```
make server
```

2. execution

Run headless (no GUI)

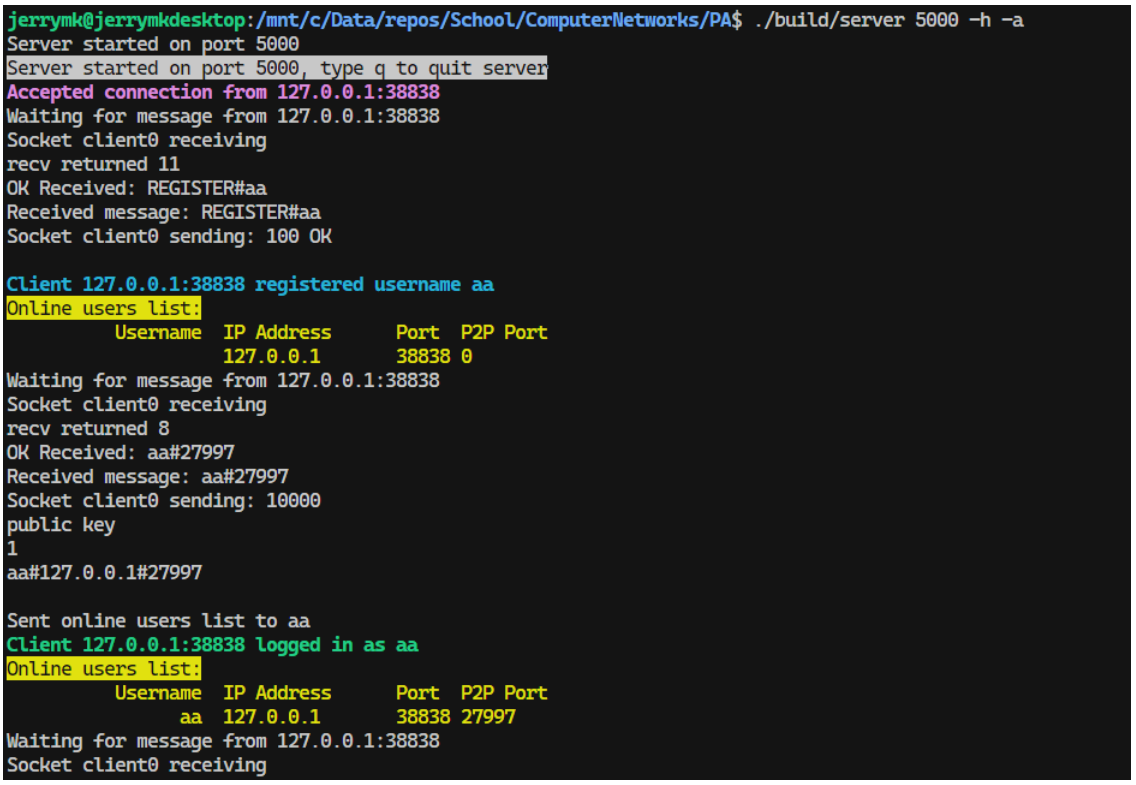
Execute the following command:

```
./build/server <server_port> -h <log_options>
```

Option flags are as follows:

```
<log_options>:
-d: show client register, login, and exit info in console only
-s: also show online list on login or exit
-a: also show TCP messages, without this tag, errors will still be shown
```

```
<other flags>:
-h: run headless, no gui
```



With or without GUI, you are able to type `q` and press enter in the terminal to stop the server and its sockets gracefully, to prevent issues like bound socket not released after program got interrupted.

Run with GUI

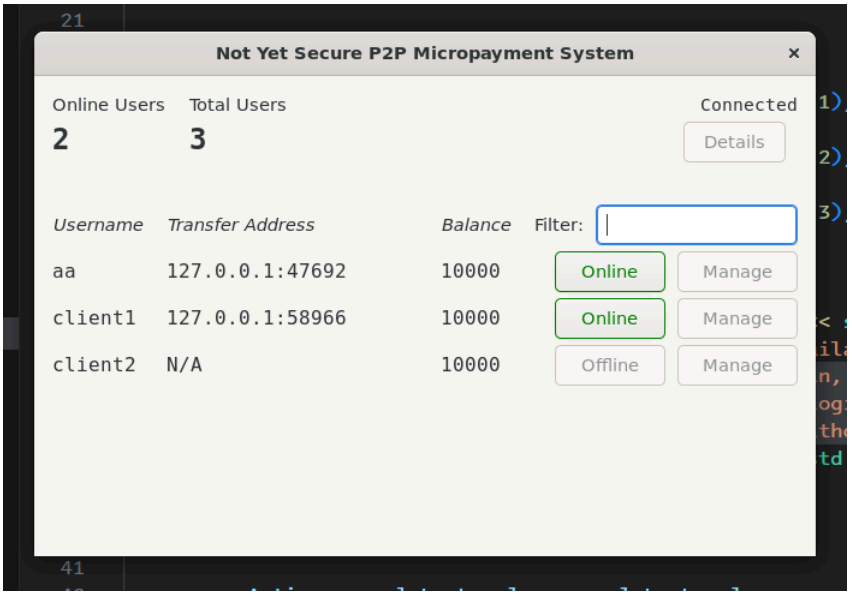
Execute the following command:

```
./build/server <server_port> <log_options>
```

or

```
make server-run
```

to run the server GUI application on port `5000` with `-s` logging level.



A look on how the console looks when executed with the `-s` option.

```
jerrymk@jerrymkdesktop:/mnt/c/Data/repos/School/ComputerNetworks/PA$ make server-run
./build/server 5000 -s
Server started on port 5000, type q to quit server
Accepted connection from 127.0.0.1:47692
Client 127.0.0.1:47692 registered username aa
Online users list:
          Username  IP Address      Port  P2P Port
                    127.0.0.1       47692 0
Client 127.0.0.1:47692 logged in as aa
Online users list:
          Username  IP Address      Port  P2P Port
                aa  127.0.0.1       47692 12778
Accepted connection from 127.0.0.1:58966
Client 127.0.0.1:58966 registered username client1
Online users list:
          Username  IP Address      Port  P2P Port
                aa  127.0.0.1       47692 12778
                    127.0.0.1       58966 0
Client 127.0.0.1:58966 logged in as client1
Online users list:
          Username  IP Address      Port  P2P Port
                aa  127.0.0.1       47692 12778
           client1  127.0.0.1       58966 48267
Accepted connection from 127.0.0.1:53240
Client 127.0.0.1:53240 registered username client2
Online users list:
          Username  IP Address      Port  P2P Port
                aa  127.0.0.1       47692 12778
           client1  127.0.0.1       58966 48267
                    127.0.0.1       53240 0
Client 127.0.0.1:53240 logged in as client2
Online users list:
          Username  IP Address      Port  P2P Port
                aa  127.0.0.1       47692 12778
           client1  127.0.0.1       58966 48267
           client2  127.0.0.1       53240 16654
Client client2 logged out
Online users list:
          Username  IP Address      Port  P2P Port
                aa  127.0.0.1       47692 12778
           client1  127.0.0.1       58966 48267
           client2  127.0.0.1       53240 16654
Client client1 logged out
Online users list:
          Username  IP Address      Port  P2P Port
                aa  127.0.0.1       47692 12778
           client1  127.0.0.1       58966 48267
Server stopped successfully
jerrymk@jerrymkdesktop:/mnt/c/Data/repos/School/ComputerNetworks/PA$
```

3. program execution environment

This program currently only runs on Linux with GUI. The `libgtkmm-3.0-dev` package may be needed during runtime. There is an option to run the application headless, but `gtkmm` libraries are still required to build the program. There

is no option to build without it.

4. references and sources

- The socket programming part of this application (included in the `src/mySocket.h` file) is heavily inspired by the awesome book `Beej's Guide to Network Programming - Using Internet Sockets` by Brian "Beej Jorgensen" Hall, completely free to read at **A4 direct link** *<https://beej.us/guide/bgnet/pdf/bgnet_a4_c_2.pdf>* or **the home page** *<https://beej.us/guide/bgnet/>*.
- Some code may be AI generated by `GitHub Copilot`, but all the code are understood and reviewed by me.

5. project structure

You may skip reading the remainder if you'd like. Below briefly describes how the program is written, as well as the graphical user interface screenshots and error handling demonstrations, which is not mentioned in the submission requirements to be included in the operation explanation document.

All source files are included in the `src` folder.

The entry point `main` for the server program is located at `server.cpp`. This file contains the code to run the `serverMainWindow` or headless.

The custom class for basic socket interfacing is located in `mySocket.h`. This file uses the Unix/Linux Socket Programming libraries, and include functions to interface with a socket in a simpler manner. All functions include extensive error handling, debug information, and timeouts for connecting and receiving.

```
// checks if a string port number is valid, and converts it to an integer. isServer = true
allows ports less than 1024
int MySocket::checkPort(const std::string &port, bool isServer = false);
// connect to the given hostname/IP address and port using TCP
bool MySocket::connect(const std::string &hostname, const std::string &serverPort, int timeout
= 5);
// bind the socket to the given port on the system
bool MySocket::bindSocket(const std::string &clientPort);
// listen for incoming TCP connections
bool MySocket::listen(int timeout_sec = 5);
// accept the incoming connection with the given socket
void MySocket::accept(MySocket &newSock);
// randomly find an available port on the system
int MySocket::findAvailablePort();
// send a message with the socket
bool MySocket::send(const std::string &message);
// receive a message from the socket
std::string MySocket::recv(int timeout_sec = 5);
// close the active TCP connection. This is also called when the MySocket object is destroyed
void MySocket::closeConnection();
```

All the logic required to run the server is located in `serverAction.h`. A global object `ServerAction serverAction` manages the state for the server program, and provides interfaces for performing server actions.

```
// this stores all the information of an account
struct ServerAction::UserAccount {
    std::string username;
    int balance;
};

// this stores all the information of an online connection
struct ServerAction::OnlineEntry {
    MySocket *clientSocket;
    std::string username;
```

```
    std::string ipAddr;
    int clientPort;
    int p2pPort;
};

std::vector<UserAccount> ServerAction::userAccounts;
std::vector<OnlineEntry> ServerAction::onlineUsers;

// start the server on the given port
bool ServerAction::startServer(const std::string &port);
// start the server port listening thread to accept client connections
void ServerAction::startListening();
// a successful connection from a client will create a new socket and a new thread will be
created running this function until client exits
void ServerAction::clientInstance(const std::pair<std::string, std::string> &clientAddr);
// the client socket thread keeps listening for incoming messages, and when a message is
received, this function is called to parse the message and perform the corresponding actions
bool ServerAction::handleIncomingMessage(std::vector<ServerAction::OnlineEntry>::iterator
&clientEntry);
// stop the listening thread and close the socket gracefully
void ServerAction::stopListening()

// print the online users to the console window
void ServerAction::printOnlineList()
// a function to create a new UserAccount entry to the list
bool ServerAction::registerUser(MySocket &client, const std::string &username)
// send the online users to a given client socket, when the user requests using the List
command, or when logging in
bool ServerAction::sendOnlineUsers(MySocket &client, const UserAccount &record)

// a helper function to find an OnlineEntry from the list
std::vector<OnlineEntry>::iterator ServerAction::findOnlineUser(const std::pair<std::string,
std::string> &ipAndPort);
// a helper function to find an UserAccount from the list
std::vector<UserAccount>::iterator ServerAction::findUserAccount(const std::string &username);
```

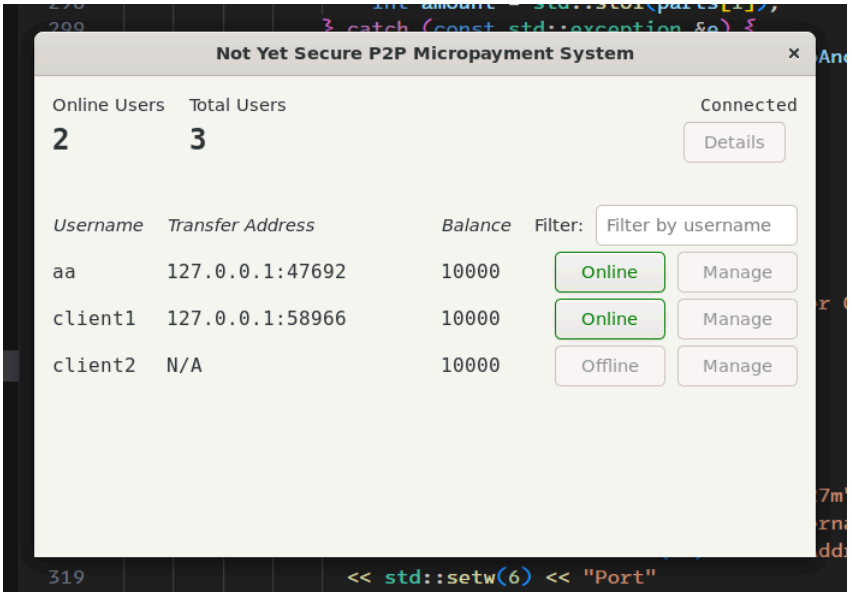
Another thread is created to listen for the terminal input, to achieve "type `q` to quit server" functionality while enabling console output.

So to recap, here are the threads used in this multi-threaded server implementation:

- main thread - runs the GUI code or idling waiting for server termination for headless
- key press thread - wait for `q` to be typed to quit the server
- server port listening thread - listens for incoming connection requests
- client instance thread **for each client** - when there is a connection request, a new socket is created to accept it, and a new thread is created running the `clientInstance` function to interact with the client

All the GTK window classes for the server GUI is in `server_windows/*.h`.

A brief look at the server main window:



Clicking the Online button shows the IP and port of the client.