COMS3200 Assignment 1 2023S1

100 total marks, 25% overall course mark
Due: 15:00 19 April 2023

1 Preface

1.1 Notes

- This document is subject to change for the purposes of clarification. Changes made since the original release will be highlighted in red.
- Please post any questions on the course Ed stem page.
- It is strongly recommended that all programming tasks are done in Python 3.6.8 or C.

1.2 Revision History

- 20 March, 2023: Version 1.0 Released.
- 24 March, 2023: Version 1.1 Clarifications to startup behaviour, malformed commands and python version.
- 29 March, 2023: Version 1.2 Clarifications to commands and queue behaviour.
- 31 March, 2023: Version 1.3 Clarifications to commands

2 Part A: Problem Solving Questions

This section is worth 25% of the assignment.
The scope of this section is chapters 1-3.
Show full working for all questions. Marks may be withheld if working is not provided.
The question set is located on Blackboard under Assessment $\implies$ Assignment 1 $\implies$ Part A: Problem solving questions.
There is no time limit to submit these answers and multiple resubmissions are permitted. Only the last submitted attempt will be marked.

3 Part B: Wireshark Questions

This section is worth 20% of the assignment.
This section covers ICMP, HTTP, DNS and DHCP.
The question set is located on Blackboard under Assessment $\implies$ Assignment 1 $\implies$ Part B: Wireshark questions and the capture file is located under Assessment $\implies$ Assignment 1 $\implies$ Part B: Packet capture File.
There is no time limit to submit these answers and multiple resubmissions are permitted. Only the last submitted attempt will be marked.

4 Part C: Socket Programming

This section is worth 55% of the assignment.

4.1 Goals

You will implement a multi-channel chat application using socket programming in Python 3 or C.
You will create two programs; a chat client and a chat server.

4.2 Programs

You must use socket and multithreading libraries. The use of any other third-party libraries is not allowed. You must implement all chat logic yourself.
If you choose to use Python then you must name your programs `chatclient.py` and `chatserver.py`.
If you choose to use C then you must name your programs `chatclient` and `chatserver`.

4.3 Report

You must also submit a plaintext README file named `readme.txt`. This file must contain:

- A brief and high-level abstract detailing your overall approach.
- A list of functions you implemented with their respective descriptions.
- An overview to how you tested your code.
- IEEE references to any code you used and was not written by you. This includes generative AI.

4.4 Requirements

4.4.1 High-level Requirements

You must implement a server program which can support concurrent communication between multiple clients across multiple channels.
The server must be able to maintain at least three distinct chat channels simultaneously, each of which may contain at least five simultaneous connections. The exact operation of the server's channels will be specified in an external config file.
As clients connect to a channel within the server, they are to be placed in a waiting queue until there is room available for them to join the channel.
Once in a channel, clients are able to send messages to other clients within that same channel.
Clients are able to specify the username which will be used to identify them within the channel. Your client and server programs must rely on socket and multithreading libraries in either Python or C. If you wish to use another programming language, you must get written permission from the course coordinator.
You may use any transport layer protocol but it is highly recommended you use TCP.

4.4.2 Functional Requirements

The server must be able to be started by one of the two below commands:

```
$ ./chatserver configfile
$ python3 chatserver.py configfile
```

where `configfile` is the path to the server configuration file. The client must be able to be started by one of the two below commands:

```
$ ./chatclient port username
$ python3 chatclient.py port username
```

where `port` is the port of the desired channel and `username` is the name which should be used to identify the client in the chat channel.

All lines in the server configuration file must follow the below format:

```
channel <channel_name> <channel_port> <channel_capacity>
```

where `<channel_name>` is the name of the channel, `<channel_port>` is the port that channel should use and `<channel_capacity>` is the maximum number of users who can be active at once within a channel. The below requirements must be met:

- Each channel must operate on independent sockets.
- No two channels can have the same name or port.
- No channel should use the ephemeral port.
- Channel names cannot begin with a number.
- There must be at least three channels in the config file.
- Each channel's capacity must be at least five.

If the configuration file is missing, invalid or not specified, then the server process should exit immediately with status code 1. Similarly, if the client cannot connect to the server due to malformed arguments or an invalid port number, then the client process should also exit immediately with status code 1.
Once the server is running, it will accept incoming client connections. Before clients are permitted to enter their requested channel, they will be entered into a first-in-first-out waiting queue specific to that channel. Once there is a free position in the channel, the longest-waiting client may be removed from the waiting queue and entered into the channel.
Upon entering the waiting queue, the below should be displayed on the client's `stdout`:

```
[Server message (<time>)] Welcome to the <channel_name> channel, <username>.
```

Where `<time>` is the server's current time in 24 hour time and in `hh:mm:ss` format, `<channel_name>` is the name of the channel the client is waiting to join and `<username>` is the name of the client.

When the client joins the waiting queue, and whenever a client leaves the queue, the below should be displayed on the client's stdout:

```
[Server message (<time>)] You are in the waiting queue and there are <n> user(s) ahead of
    you.
```

where <n> is the number of users in the waiting queue who joined before this client.

When the client is finally able to join the channel, or if the queue is empty and there is already available space in the channel, the above message is not displayed. Instead, the below message is displayed to the stdout of every client in the channel:

```
[Server message (<time>)] <username> has joined the channel.
```

and the below is displayed to the server's stdout:

```
[Server message (<time>)] <username> has joined the <channel_name> channel.
```

Clients are able to send messages by typing in stdin. These messages are then sent to all other clients currently in the same channel. For example, if the client named Dan types Hello COMS3200 students! into his stdin, the below message should be displayed to the stdout of both the server and all clients within the channel:

```
[Dan (<time>)] Hello COMS3200 students!
```

4.4.3 Clientside Commands

Besides just broadcasting messages to each other, clients are also able to enter special commands into stdin:

- /whisper <username> <message>
  Only the client named <username> receives the message. It will be displayed as:

  ```
  [<sender> whispers to you: (<time>)] <message>
  ```

  Where <sender> is the username of the client which typed the whisper command. If <username> is not in the channel, the whisperer will receive this message to stdout:

  ```
  [Server message (<time>)] <username> is not here.
  ```

  Regardless of whether or not the target client is present in the channel, the below message will be displayed to the server's stdout:

  ```
  [Dan whispers to <username>: (<time>)] <message>
  ```

- /quit
  The client who uses this command exits the channel and closes their connection to the server. The below message is displayed to the stdout of both the server and all remaining users:

  ```
  [Server message (<time>)] <username> has left the channel.
  ```

  This command should be usable while the client is still within a waiting queue as well as within a channel. If a client in the queue uses this command then only the server should receive the above message.

- /list
  The list of all channels is displayed to the client's stdout in the order in which they are listed in the server configuration file. Each channel is listed on a separate line according to the below format:

  ```
  [Channel] <channel_name> <current>/<capacity>/<queue_length>.
  ```

  where <channel_name> is the name of the channel, <current> is the current number of clients currently in the channel, <capacity> is the total capacity of the channel and <queue_length> is the number of clients currently in the queue.

- `/switch <channel_name>`
  The client leaves their current channel or waiting queue and joins the waiting queue for the specified channel. If the specified channel exists, then the below message is displayed to the `stdout` of both the server and every client remaining in the channel:

  ```
  [Server message (<time>)] <username> has left the channel.
  ```

  The client who issued the switch command should receive the waiting queue messages (pg 3) for that channel. If the channel does not exist then the client will stay in the channel and the below message will be displayed to their `stdout`:

  ```
  [Server message (<time>)] <channel_name> does not exist.
  ```

  No two clients with the same username should be in the same channel at the same time. If a client attempts to switch into a channel where there is already a client with the same username (or in its respective waiting queue), the below should be displayed to that client's `stdout`:

  ```
  [Server message (<time>)] Cannot switch to the <channel_name> channel.
  ```

  The client should not leave the channel and no messages will be sent to any other clients or the server. This command should be usable within both channels and waiting queues.

- `/send <target> <file_path>`
  The file on the sender's machine located at `<file_path>` will be sent to the client named `<target>`. The below message will be displayed to the sender's `stdout`:

  ```
  [Server message (<time>)] You sent <file_path> to <username>.
  ```

  and the below message will be displayed to the server's `stdout`:

  ```
  [Server message (<time>)] <sender> sent <file_path> to <username>.
  ```

  If the target client is not within the same channel as the sending client, then neither of the above two messages are displayed and the file will not be sent. Instead, the below message will be displayed to the sender's `stdout` only:

  ```
  [Server message (<time>)] <target> is not here.
  ```

  If the file specified by `<file_path>` does not exist, then only the below is displayed to the sender's `stdout`:

  ```
  [Server message (<time>)] <file_path> does not exist.
  ```

  If the file can be sent, it should be saved to the same directory as the target's `chatclient` executable.
  If both the file does not exist and the target client is not within the same channel as the sender, then both error messages should be displayed. The invalid client message should be displayed first.
  N.B. For testing purposes, we will always be starting the server process from its directory.

- Anything starting with / which is not one of these commands should be treated as a chat message.

4.4.4 Serverside Commands

The server is also able to use these below commands by typing them into `stdin`:

- `/kick <channel_name>:<username>`
  If there is a client named `<username>`, in the channel named `<channel_name>`, they will be removed and their connection to the server will be closed. The below message is displayed to the `stdout` all remaining users within that channel:

  ```
  [Server message (<time>)] <username> has left the channel.
  ```

  and the below message is displayed to the `stdout` of the server:

  ```
  [Server message (<time>)] Kicked <username>.
  ```

  If there is no such client in the specified channel then nothing is displayed to the clients' `stdout` and the below is displayed on the server's stdout:

  ```
  [Server message (<time>)] <username> is not in <channel_name>.
  ```

  If the specified channel does not exist then nothing is displayed to the clients' `stdout` and the below should be displayed to the server's `stdout`:

  ```
  [Server message (<time>)] <channel_name> does not exist.
  ```

- `/mute <channel_name>:<username> <time>`
  Prevents the client named `<username>` in the channel named `<channel_name>` from sending any messages for `<time>` seconds.
  If this client exists then the below will be displayed to the server's `stdout`:

  ```
  [Server message (<time>)] Muted <username> for <time> seconds.
  ```

  The below is displayed to the muted client's `stdout`:

  ```
  [Server message (<time>)] You have been muted for <time> seconds.
  ```

  and the below is displayed to every other client's `stdout`:

  ```
  [Server message (<time>)] <username> has been muted for <time> seconds.
  ```

  If a client tries to send messages or use `/whisper` while muted, the below will be displayed to their `stdout`:

  ```
  [Server message (<time>)] You are still muted for <n> seconds.
  ```

  Where `<n>` is the number of seconds remaining before the client is unmuted. The message they attempted to send will not be sent to any other users.
  If the target client is not in the specified channel or the channel does not exist, the below will be displayed to the server's `stdout`:

  ```
  [Server message (<time>)] <username> is not here.
  ```

  `<time>` must be a positive integer. If anything other than a positive integer is specified, then the below message should be displayed to the server's `stdout`:

  ```
  [Server message (<time>)] Invalid mute time.
  ```

  All commands except `/whisper` should still be usable while muted.

- `/empty <channel_name>`
  Every client in the channel named `<channel_name>` will have their connection closed. The below will be displayed to the server's `stdout`:

  ```
  [Server message (<time>)] <channel_name> has been emptied.
  ```

  If the specified channel does not exist, then the below will be displayed to the server's `stdout` instead:

  ```
  [Server message (<time>)] <channel_name> does not exist.
  ```

  No clients in the emptied channel should receive any messages.

- `/shutdown`
  The entire server will shut down. Connections to every client in every channel and waiting queue will be closed and the server process will exit. No clients should receive any messages.

- Anything entered into the server's `stdin` which is not one of these commands should be ignored.

4.4.5 Miscellaneous Requirements

- If you choose to use C, then you must also submit a makefile. Both your client and server program must be able to be compiled with the `$ make` command.
  If your code does not compile, you might not receive any marks.

- Both client and server processes should be runnable on Moss.

- The use of `fork()` and `select()` in C and Python are expressly forbidden - This includes the Python's select library functions such as devpoll, epoll, kqueue, poll, kevent and select.

- Only one server process should run at any time.

- When a client's connection with the server ends, the client process should exit.

- No two clients with the same name should be within the same channel at the same time. If client attempts to join a channel already containing another client of the same name when first starting from the command line, the below message should be displayed to the client's `stdout`:

  ```
  [Server message (<time>)] Cannot connect to the <channel_name> channel.
  ```

  and the client process should immediately exit.

- If a client in a channel does not send any message or use any command for 100 seconds, then it should be removed from the channel and their connection should be closed. The below message should be displayed to the `stdout` of both the server and all remaining clients in the channel:

  ```
  [Server message (<time>)] <username> went AFK.
  ```

  If a client is muted by the server, the timeout counter should freeze until the client becomes unmuted. If a client attempts to chat or issues a command while muted, their timeout counter should not reset.

- Everything taken from `stdin` should have whitespace stripped from both sides before being evaluated as a chat message or a command.

- Commands may have multiple spaces between arguments.

4.5 Marking Breakdown

Marks will be awarded according to the following breakdown:

- Client and server processes start correctly or exit on invalid startup parameters - 1 mark
- Chat messages and announcements can be sent/received in at least one channel - 4 marks
- Chat messages and announcements can be sent/received in at least three channels. Chat messages and announcements sent to one channel do not appear in other any other channels - 5 marks
- Clients are held in the waiting queue until space in the correct channel is available - 5 marks
- Clients are correctly removed from chat after timing out - 5 marks
- `/whisper` and `/quit` work - 5 marks
- `/kick` and `/empty` work - 5 marks
- `/shutdown` works - 5 marks
- `/list` and `/switch` work - 5 marks
- `/mute` works - 5 marks
- `/send` works - 10 marks

Total: 55 marks.
N.B.: Although the README itself is not assigned marks, it is still mandatory for integrity purposes. Failure to include the README file will result in a $\geq$50% penalty.

5 Miscellaneous Notes

- "Clients within a channel" refer only to those in the channel and not those in the related waiting queue.
- Your code will not be tested for memory leaks but having large memory leaks may render your code untestable. If you make any critical programming mistakes which impact our ability to test your code, your marks may be severely limited.
- For marking purposes, you should ignore commands which have too many or too few arguments.

6 Integrity

All code submissions will be checked for similarity and plagiarism. All code in your submission which was not written by you (or was written by you for a previous assignment) must be correctly referenced in IEEE format in both in your code and your README file. This includes code written by generative AI.

7 Submission

You must submit all source code files and your README file together in a zip file. This zip file must be named `A1_sxxxxxxx.zip`, where the `sxxxxxxx` is your student ID. This zip file must be submitted to the course Blackboard page under Assessment $\implies$ Assignment 1 $\implies$ Part C: Code Submission.

8 Resources

These resources may be of use to you:

- Joyce, P. (2022). Sockets. In: C and Python Applications. Apress, Berkeley, CA. https://doi-org.ezproxy.library.uq.edu.au/10.1007/978-1-4842-7774-4_6
- Socket Programming HOWTO, available at: https://docs.python.org/3/howto/sockets.html
- Socket Programming with Multi-threading in Python, https://www.tutorialspoint.com/socket-programming-with-multi-threading-in-python
- The Linux Manual Pages