The University of Queensland
School of Electrical Engineering and Computer Science

COMS3200 – Semester 1, 2025
Assignment 1 (Version 1.2)

Marks: 100
Weighting: 35%
Due: 3:00pm, Friday, 9th May, 2025

Specification changes since version 1.0 are shown in red and are summarised at the end of the document.
Changes from v1.1 to v1.2 are shown in blue.

Introduction

This assignment is divided into Part A and Part B. Part A is a set of questions that examine your ability to use Wireshark. The question set can be found on the course Blackboard site. Part B contains programming tasks to demonstrate your understanding of socket programming, networking and multi-threaded programming. You have to create a server program and a client program so that multiple clients can chat in multiple channels under the management of the server program. In Part B of this assignment, you can use either C or Python as your coding language (you cannot use any other languages). Python 3.11 is the required version if you use Python for this assignment. This is the version available on `moss.labs.eait.uq.edu.au`.

Student Conduct

This is an individual assignment. You should feel free to discuss **general** aspects of C programming, Python programming and the assignment specification with fellow students, including on the discussion forum. In general, questions like "How should the program behave if ⟨this happens⟩?" would be safe, if they are seeking clarification on the specification.

You must not actively help (or seek help from) other students or other people with the actual design, structure and/or coding of your assignment solution. It is **cheating to look at another person's assignment code** and it is **cheating to allow your code to be seen or shared in printed or electronic form by others**. All submitted code will be subject to automated checks for plagiarism and collusion. If we detect plagiarism or collusion, formal misconduct actions will be initiated against you, and those you cheated with. That's right, if you share your code with a friend, even inadvertently, then **both of you are in trouble**. Do not post your code to a public place such as the course discussion forum or a public code repository. (Code in private posts to the discussion forum is permitted.) You must assume that some students in the course may have very long extensions so do not post your code to any public repository until at least three months after the result release date for the course (or check with the course coordinator if you wish to post it sooner). Do not allow others to access your computer – you must keep your code secure. Never leave your work unattended.

You must follow the following code usage and referencing rules:

Code Origin	Usage/Referencing
Code provided by teaching staff this semester Code provided to you in writing **this semester** by COMS3200 teaching staff (e.g., code hosted on Blackboard, found in `/local/courses/coms3200/resources` on `moss`, posted on the discussion forum by teaching staff, or shown in class).	**Permitted** May be used freely without reference. (You must be able to point to the source if queried about it – so you may find it easier to reference the code.)
Unpublished code you wrote earlier Code you have personally written in a previous enrolment in this course or in another UQ course or for other reasons and where that code has not been shared with any other person or published in any way.	**Conditions apply, references required** May be used provided you understand the code AND the source of the code is referenced in a comment adjacent to that code (in the required format – see the style guide). If such code is used without appropriate referencing then this will be considered misconduct.

Code Origin	Usage/Referencing
Code from `man` pages on `moss` Code examples found in `man` pages on `moss`. (This does not apply to code from `man` pages found on other systems or websites unless that code is also in the `moss` `man` page.)	**Conditions apply, references required** May be used provided you understand the code AND the source of the code is referenced in a comment adjacent to that code (in the required format – see the style guide). If such code is used without appropriate referencing then this will be considered misconduct.
Code and learning from AI tools Code written by, modified by, debugged by, explained by, obtained from, or based on the output of, an artificial intelligence tool or other code generation tool that you alone personally have interacted with, without the assistance of another person. This includes code you wrote yourself but then modified or debugged because of your interaction with such a tool. It also includes code you wrote where you learned about the concepts or library functions etc. because of your interaction with such a tool. It also includes where comments are written by such a tool – comments are part of your code.	**Conditions apply, references & documentation req'd** May be used provided you understand the code AND the source of the code or learning is referenced in a comment adjacent to that code (in the required format – see the style guide) AND an ASCII text file (named `toolHistory.txt`) is included in your submission that describes in detail how the tool was used. (All of your interactions with the tool must be captured.) If such code is used without appropriate referencing and without inclusion of the `toolHistory.txt` file then this will be considered misconduct. See the detailed AI tool use documentation requirements on Blackboard – this tells you what must be in the `toolHistory.txt` file.
Code copied from sources not mentioned above Code, in any programming language: • copied from any website or forum (including StackOverflow and CSDN); • copied from any public or private repositories; • copied from textbooks, publications, videos, apps; • copied from code provided by teaching staff only in a previous offering of this course (e.g. previous assignment one solution); • written by or partially written by someone else or written with the assistance of someone else (other than a teaching staff member); • written by an AI tool that you did not personally and solely interact with; • written by you and available to other students; or • from any other source besides those mentioned in earlier table rows above.	**Prohibited** May **not** be used. If the source of the code is referenced adjacent to the code then this will be considered code without academic merit (not misconduct) and will be removed from your assignment prior to marking (which may cause compilation to fail and zero marks to be awarded). Copied code without adjacent referencing will be considered misconduct and action will be taken. This prohibition includes code written in other programming languages that has been converted to C or Python.
Code that you have learned from Examples, websites, discussions, videos, code (in any programming language), etc. that you have learned from or that you have taken inspiration from or based any part of your code on but have not copied or just converted from another programming language. This includes learning about the existence of and behaviour of library functions and system calls that are not covered in class.	**Conditions apply, references required** May be used provided you do not directly copy code AND you understand the code AND the source of the code or inspiration or learning is referenced in a comment adjacent to that code (in the required format – see the style guide). If such code is used without appropriate referencing then this will be considered misconduct.

Uploading or otherwise providing the assignment specification or part of it to a third party including online tutorial and contract cheating websites is considered misconduct. The university is aware of many of these sites and many cooperate with us in misconduct investigations. You are permitted to post small extracts of this document to the course Ed Discussion forum for the purposes of seeking or providing clarification on this specification.

In short – **Don't risk it!** If you're having trouble, seek help early from a member of the teaching staff. Don't be tempted to copy another student's code or to use an online cheating service. Don't help another COMS3200 student with their code no matter how desperate they may be and no matter how close your relationship. You should read and understand the statements on student misconduct in the course profile and on the school website: `https://eecs.uq.edu.au/current-students/guidelines-and-policies-students/student-conduct`.

Part A

This part of the assignment is a Wireshark quiz which is worth 20% of this assignment. Remember to submit your quiz before the due date. The quiz is under the Assessment section on the Blackboard site. There is no time limit to submit these answers, and only one submission is allowed. Please make sure to complete your submission. Incomplete submissions will not be marked, and the mark for that part will be zero.

Part B

This part of the assignment is about TCP socket programming, which is worth 80% of this assignment. TCP must be used for all socket connections and communication. You will write two programs: `chatclient` and `chatserver`. `chatserver` hosts a channel or channels that support multiple `chatclient` processes (in the same channel) to chat with each other. Both `chatserver` and `chatclient` can execute specific commands, some of the commands require communication between the `chatserver` and `chatclient`. You will need to determine the communication protocol used between the client and the server and what state information is kept in each.

Specification - `chatclient`

The `chatclient` program provides a command line interface to chat with other client or clients (in the same channel) through a server (`chatserver`). Your client will need to listen for incoming message(s) on both `stdin` and the network connection.

Command Line Arguments

Your `chatclient` program is to accept command line arguments as follows:

`./chatclient` *`port_number client_username`* (if using C) or

`python3 chatclient.py` *`port_number client_username`* (if using Python)

Italics indicate a placeholder for user-supplied arguments. Arguments must always be in the given order. Some examples of how the program might be run include the following[1]:

```
./chatclient 1234 Bob
```

```
./chatclient 3456 Alice
```

The meaning of the arguments is as follows:

- *`port_number`* – this mandatory argument specifies which localhost port the server is listening on. The range of *`port_number`* is between 1024 and 65535 (inclusive). The server may be listening on multiple ports – each port corresponds to a *channel* that will have a specific name.
- *`client_username`* – this mandatory argument specifies the name of this client. This username will be sent to the server and will be used to identify the client during the chat. Thus, the username must be unique within the channel.

Before doing anything else, your program must check the command line arguments for validity. If the program receives an invalid command line then it must print the (single line) message:

```
Usage: chatclient port_number client_username
```

to standard error (with a following newline), and exit with an exit status of 3. (This will be the message whether you implement the client with Python or C.)

Invalid command lines include (but may not be limited to) any of the following:

- No arguments are present (i.e., there is no *`port_number`* argument).
- An unexpected argument is present (i.e., too many arguments).
- Any argument is an empty string.
- The *`client_username`* argument contains spaces.

Checking whether the *`port_number`* is a valid port is not part of the usage checking (other than checking that the value is not empty). The validity of the port is checked after command line validity as described next.

[1] This is not an exhaustive list and the examples assume that a `chatserver` server is running on the listed ports.

Client Port Checking

If the *port_number* argument is not an integer or `chatclient` is unable to create a socket and connect to the server on the specified port of host `localhost`, it shall print the following message (terminated by a newline) to `stderr` and exit with exit status 7:

`Error: Unable to connect to port N.`

where *N* should be replaced by the port number argument given on the command line.

Client Username Checking

If `chatclient` wishes to connect to a channel but there already exists a client with the same *client_username* in that channel or the queue for that channel, `chatclient` will be notified by the server and `chatclient` will print the following message (terminated by a newline) to `stdout` and exit with exit status 2:

`[Server Message] Channel "channel_name" already has user client_username.`

where *channel_name* should be replaced by the name of the channel that `chatclient` wishes to join, and *client_username* should be replaced by the client username given on the command line. The double quotes must be present.

Client Runtime Behaviour

Assuming that the checks above are passed and your `chatclient` can successfully connect to the channel, then your `chatclient` must print and flush the following message (terminated by a newline) to `stdout`:

`Welcome to chatclient, client_username.`

where *client_username* should be replaced by the username argument given on the command line.

If the channel has capacity for the client, then the client should be notified by `chatserver` and print the following message to `stdout` (terminated by a newline):

`[Server Message] You have joined the channel "channel_name".`

where *channel_name* is replaced by the name of the channel that the client has connected to. The double quotes must be present.

If the channel does not have capacity, then the client should be notified by `chatserver` and print the following message to `stdout` (terminated by a newline):

`[Server Message] You are in the waiting queue and there are X user(s) ahead of you.`

where *X* should be replaced by the number of users (clients) waiting in the queue ahead of this client.

The client must read lines of input from the user (via `stdin`) and respond to them as described below (e.g. usually by sending a message to the server), and will simultaneously read lines received from the server (via the connected socket) and respond to them as described below. Communication is asynchronous – either party can send a message at any time – so your client must be implemented to support this. You can do this by using two threads (one reading from `stdin` and one reading from the connected network socket).

Client – Handling Standard Input

If the client has connected to a channel, then the client can broadcast messages in this channel by entering messages in `stdin`. Messages are to be terminated by a single newline. These messages are to be sent to the server, and the server will send them to all other clients currently in the same channel. For example, if a client named `Bob` enters

`Hello COMS3200 students!`

into `stdin`, the below message (terminated by a newline) should be printed to the `stdout` of both the server and all clients (including the sender) within the channel (not the queue):

`[Bob] Hello COMS3200 students!`

The client can also enter the following commands on `stdin`. Commands should be started with `/` and terminated by a single newline. No leading or trailing spaces are to be accepted[2] but multiple spaces may appear between arguments. If an invalid command is entered (other than those that result in an error message printed to the user as described below), then the server will treat it as a broadcast message.

[2] Trailing whitespaces are permitted with the `/whisper` command since these are considered part of the message

If the command has an invalid number of arguments, then `chatclient` should print a usage message (terminated by a newline) to `chatclient`'s stdout to show the correct usage of this command. For example, if `chatclient` reads a line like this from stdin:

```
/send Bob
```

which is missing the *`file_path`* argument, then `chatclient` must print the following usage message (terminated by a newline) to its stdout:

```
[Server Message] Usage: /send target_client_username file_path
```

The content of this usage message depends on the command keyword. Valid commands (which define the messages to be printed) are listed below. In most cases, the use of a command will result in a message or messages being sent to the server. If the client is in a queue to join a channel then only the `/quit`, `/list` and `/switch` commands are valid. All other commands will be ignored.

- **`/send` *`target_client_username file_path`***
  This command indicates that the client wishes to send a file to another client within the same channel. *`file_path`* may include the path and basename of the file. If the *`target_client_username`* is the sender's own username then the following message will be printed to the sender's `stdout` (terminated by a newline):

  ```
  [Server Message] Cannot send file to yourself.
  ```

  If the target client is not within the same channel as the sending client, the file will not be sent, and the below message (terminated by a newline) will be printed to the sender's `stdout`:

  [Server Message] *`target_client_username`* is not in the channel.

  where *`target_client_username`* should be replaced by the target client username given in the `/send` command.

  If the file does not exist or can not be opened for reading, then only this message (terminated by a newline) is printed to the sender's `stdout`:

  [Server Message] "*`file_path`*" does not exist.

  where *`file_path`* should be replaced by the path of the file to send given in the `/send` command. The double quotes must be present.

  If both the file does not exist and the target client is not within the channel, then both error messages should be displayed. The check of the *`target_client_username`*'s existence shall be the first.

  If the file was sent successfully, it should be saved in the current working directory of the target's client. The following message (terminated by a newline) will then be printed to the sender's `stdout`:

  [Server Message] Sent "*`file_path`*" to *`target_client_username`*.

  where *`file_path`* should be replaced by the path of the file you wish to send given in the `/send` command, and *`target_client_username`* should be replaced by the target client username given in the `/send` command. The double quotes must be present.

  If the file was sent successfully (i.e. the receiving client successfully saved the file), the following message (terminated by a newline) will be printed to `chatserver` 's `stdout` and the receiving client's `stdout` (terminated by a newline):

  [Server Message] *`sender_client_username`* sent "*`file_path`*" to *`target_client_username`*.

  where *`sender_client_username`* should be replaced by the sender client's username given in the `/send` command, *`file_path`* should be replaced by the basename of the file sent, and *`target_client_username`* should be replaced by the target client username given in the `/send` command. The double quotes must be present.

  File sizes up to $2^{32} - 1 = 4294967295$ bytes are to be supported. File names may not contain spaces. Existing files are to be overwritten.

  If the receiving client was unable to successfully save the file, then the following message (terminated by a newline) will be printed to the sender's `stdout`:

  [Server Message] Failed to send "*`file_path`*" to *`sender_client_username`*

  where *`file_path`* should be replaced by the path of the file the sender wished to send as given in the `/send` command, and *`sender_client_username`* should be replaced by the sender client's username.

- **`/quit`**
  This command indicates that the client exits the channel and closes the connection to the server. The below message is printed (terminated by a newline) to the `stdout` of both the `chatserver` and all remaining users in the channel:

  `[Server Message]` *`client_username`* `has left the channel.`

  where *`client_username`* should be replaced by the username of the client quitting.

  The client can use this command when in a waiting queue or a channel. If a client in the queue uses this command, then only the server should print the above message. The client who wishes to quit should not print any message and exit with exit status 0.

  EOF on the client's `stdin`, should be treated the same as if the `/quit` command was entered. Termination by signal will be treated similarly (other than the client exit status – there is no need for the client to catch `SIGINT`, `SIGQUIT`, etc.).

- **`/list`**
  This command indicates that the client wishes to receive a list of all channels that the `chatserver` is hosting. This list will be in the order of server's *`config_file`* and it will be printed to the client's stdout. Each channel is listed in a separate line according to the following format:

  `[Channel]` *`channel_name channel_port`* `Capacity:` *`current`*`/`*`channel_capacity`*`, Queue:` *`in_queue`*

  where *`channel_name`* should be replaced by the name of the channel, *`channel_port`* should be replaced by the port where the channel listens on, *`current`* should be replaced by the current number of clients in this channel, *`channel_capacity`* should be replaced by the maximum number of clients this channel could host, and *`in_queue`* should be replaced by the current number of clients waiting in the queue of this channel.

- **`/whisper`** ***`receiver_client_username chat_message`***
  This command sends a chat message to a specific client in the channel. If the receiver is not in the channel, the whisperer (the client who sends out the message) will print this message (terminated by a newline) to its `stdout`:

  `[Server Message]` *`client_username`* `is not in the channel.`

  where *`client_username`* should be replaced by the name of the receiving client.

  The receiver client will print this command (terminated by a newline) to its `stdout`:

  `[sender_client_username` `whispers to you]` *`chat_message`*

  where *`sender_client_username`* should be replaced by the sender's client username, and *`chat_message`* should be replaced by the content of the message (all characters after the space(s) that follow the *`client_username`* in the `/whisper` command line).
  When one client successfully whispers to another client, a message (terminated by a newline) will be printed to the `chatserver`'s and `sender client`'s `stdout`:

  `[sender_client_username` `whispers to` *`receiver_client_username]`* *`chat_message`*

  where *`receiver_client_username`* should be replaced by the receiver's client username, and *`sender_client_username`* and *`chat_message`* are replaced as above.

  It is permissible for a client to `/whisper` to itself.

- **`/switch`** ***`channel_name`***
  This command indicates that the client wishes to switch to a channel based on the given channel name. If there does not exist a channel matching the given *`channel_name`*, the client that initiates the switch move will stay in the current channel and print the following message (terminated by a newline) to its `stdout`:

  `[Server Message] Channel "`*`channel_name`*`" does not exist.`

  where *`channel_name`* should be replaced by the channel name provided in the `/switch` command line. The double quotes must be present.
  If a client in the target channel has the same username as the client initiating the switch, the client that initiates the switch will stay in the current channel and the below message (terminated by a newline) will be printed to its `stdout`:

`[Server Message] Channel "`*`channel_name`*`" already has user `*`client_username`*`.`

where *`channel_name`* should be replaced by the channel name provided in the `/switch` command line and *`client_username`* should be replaced by the username of the client that initiates the switch command. The double quotes must be present.

If the specified channel exists and no user in that channel has the same name, then the client who initiated the switch operation will first leave the original channel (i.e. disconnect from the server). If the client wasn't in the queue, then the remaining client(s) in the original channel (excluding those in the queue) will be notified and print the following message (terminated by a newline) to `stdout`:

`[Server Message] `*`client_username`*` has left the channel.`

The server will also print this message to its `stdout` – including if the departing client was in a queue. Then the client will attempt to join the specified channel. The new channel and all client(s) in the new channel will treat this client as a new client trying to join the channel and print messages accordingly.

If the target channel is at maximum capacity, then the client will be placed into the queue of the target channel. In this case, the client will follow the expected behaviour when dealing with the queue of a channel.

Client – Handling Messages from Server

The client may receive messages from the server at any time – see the description of the server behaviour below. The client must print the messages specified as part of the server behaviour below. It is up to you whether the text of these messages form part of the communication protocol between the server and the client or whether the client constructs the messages from information received from the server. All messages must be flushed to `stdout` upon receipt[3].

Other Client Requirements

- If your client detects that the connection to the server has been closed then it should print the following message (terminated by a newline) to `stderr`:

  `Error: server connection closed.`

  and exit with status 8. This relates to unexpected disconnections (e.g. server crashes).
- There is no requirement that `chatclient` free all memory before exiting.
- Your client must not exit abnormally due to `SIGPIPE`. A `SIGPIPE` when writing to the server will be treated as the connection to the server being closed – see line 249.
- For any aspects of behaviour not described in this specification, your program must behave in the same manner as `demo-chatclient` on `moss`. If you're unsure about some aspect, then you can run the demo program to check the expected behaviour.

Specification – `chatserver`

Your `chatserver` program creates a channel or channels on specified port(s) based on a given configuration file. The channel or channels are chat rooms for client(s) to chat with other client(s). Each of the channel(s) has a defined maximum capacity and a queue to handle waiting client(s) when the maximum capacity is met. Your `chatserver` program is to accept command line arguments as follows:

`./chatserver [`*`afk_time`*`] `*`config_file`* (if using C) or

`python3 chatserver.py [`*`afk_time`*`] `*`config_file`* (if using Python)

Italics indicate a placeholder for a user-supplied argument. [ ] indicates an optional argument. `afk_time` indicates the number of seconds a client in a channel (not in queue) can be idle (**a**way **f**rom **k**eyboard) before being disconnected (see below). This must be an integer between 1 and 1000 inclusive. By default this will be 100 seconds. The program `chatserver` reads the content inside the configuration file and creates channels based on the contents. The format for each line inside `config_file` is:

`channel `*`channel_name channel_port channel_capacity`*

[3] `chatclient` can not assume that `stdout` is line buffered.

Lines will be terminated by a single newline character[4]. Fields can be separated by one or more spaces[5]. Some examples of lines in the configuration file might be [6]:

```
channel channel_1 4455 3
channel channel_2 4466 5
channel channel_abc 4477 7
```

The meaning of the arguments is as follows:

- `channel_name` – this argument specifies the name of the channel. Channel names must be unique and only contain letters (either cases), numbers and underscores.
- `channel_port` – this argument specifies the port number for this server to use. An integer value is expected and the port number must be in the range 1024 to 65535 inclusive.
- `channel_capacity` – this argument specifies the maximum number of clients that the channel can handle at the same time. An integer value that is in the range of 1 to 8 (inclusive) is expected.

Prior to doing anything else, your program must check the command line arguments for validity. Invalid command lines include (but may not be limited to):

- Invalid value for `afk_time`.
- No *`config_file`* argument given.
- Too many arguments given.
- The *`config_file`* argument is an empty string.

If the program receives an invalid command line, then it must print the following message to `stderr` (terminated by a newline):

```
Usage: chatserver [afk_time] config_file
```

and exit with an exit status of 4. (The message is the same whether your server is implemented in C or Python.)

After checking the command line arguments, your program must then check the validity of the configuration file. If the configuration file can't be opened, or contains an invalid command or commands, then it must print this (single-line) message:

```
Error: Invalid configuration file.
```

to standard error (with a following newline), and exit with an exit status of 5.

Invalid configuration file lines include (but may not be limited to) any of the following:

- No arguments are present after the `channel` keyword (e.g. there is no `channel_name` argument)
- An argument is present but is not a valid value (e.g. the `channel_port` argument is not an integer or it is an integer not in the range 1024 to 65535 inclusive.)
- A duplicate channel name is present.
- A duplicate port number is present.
- An incorrect number of arguments is present.
- An unexpected argument is present.
- ~~The *`config_file`* argument is an empty string.~~

Server Port Checking

If `chatserver` is unable to create a socket and listen to the specified port, it shall print the following message (terminated by a newline) to `stderr` and exit with exit status 6:

```
Error: unable to listen on port N.
```

where *`N`* should be replaced by the argument given in the configuration file. `chatserver` shall print this line *`X`* number of times if there are *`X`* number of ports that `chatserver` cannot listen to. For example, if there are 5 channels specified in the `config_file`, and 3 out of 5 channels have ports that `chatserver` cannot create a socket and listen on, then `chatserver` will print this message 3 times (terminated by a new line for each time) with each port number correspondingly. The order of the messages shall match the order in which the ports are mentioned in the `config_file`.

[4] Your program can accept files with lines terminated by \r\n if you wish. We will only test \n line endings
[5] We will only test configuration files with single spaces between fields.
[6] This is not an exhaustive list.

Server Runtime Behaviour

Assuming that the checks above are passed, then for each of the channel(s) that your `chatserver` successfully creates, print and flush a message (terminated by a newline) to `chatserver`'s `stdout`:

`Channel "`*`channel_name`*`" is created on port `*`channel_port`*`, with a capacity of `*`channel_capacity`*`.`

where *`channel_name`* should be replaced by the channel name given from a single line in the configuration file, *`channel_port`* should be replaced by the port number from that line in the configuration file, and *`channel_capacity`* should be replaced by the channel capacity from that same line in configuration file. The double quotes must be present.

The order of print of these channel creation message(s) can vary in each run due to the nature of multi-threading.

After printing these message(s), your `chatserver` must print and flush the following message (terminated by a newline) to `chatserver`'s `stdout`:

`Welcome to chatserver.`

Once this message is printed, the server will start to accept connection(s) from client(s), read messages or commands received from client(s) (via connected sockets) and handle them accordingly. Communication is asynchronous. A client could connect to any channel's port at any time and any connected party can send a message at any time. Your server can monitor the status of muted client(s) and AFK timer for all client(s), while accepting commands from stdin. Your server must be implemented to support this. You can do this by using multiple threads, e.g. one per listening socket, one per connected client and one to read from `stdin`. You can use other threads if you think it is necessary.

If a client connects to a channel, `chatserver` prints the following message (terminated by a newline) to `chatserver`'s `stdout`:

`[Server Message] `*`client_username`*` has joined the channel "`*`channel_name`*`".`

where *`client_username`* should be replaced by the name of the newly connected client, *`channel_name`* should be replaced by the name of the channel that this client connects to. The double quotes must be present.

If a channel is operating at the maximum capacity (the number of clients is equal to the capacity), a new client will enter a first-in-first-out waiting queue for that channel. Once there is a free position in the channel, the longest-waiting client will be removed from the waiting queue and enter the channel. Whenever a client joins the waiting queue, or whenever another client ahead of it leaves the queue, then the below message (terminated by a newline) should be printed to the client's `stdout`:

`[Server Message] You are in the waiting queue and there are `*`X`*` user(s) ahead of you.`

where *`X`* should be replaced by the number of users (clients) waiting in the queue ahead of this client. Note that if a channel is emptied (i.e. multiple clients are removed from the channel simultaneously – see `/empty` command below) then this message will only be printed once for those clients still in the queue after the `/empty` operation.

Clients waiting in the queue are not able to broadcast or receive chat messages. Client commands `/send` and `/whisper` will be ignored if the client is in a queue. The rest of the client commands can still be used when the client is in a queue.

If a client in a channel (not in the queue) does not send any message or use any command for the away-from-keyboard time period specified on the command line (100 seconds if not specified), then it will be removed from the channel and its connection will be closed. The message below (terminated by a newline) will be printed to the `stdout` of both `chatserver` and all clients in the channel (including the client about to be disconnected):

`[Server Message] `*`client_username`*` went AFK in channel "`*`channel_name`*`".`

where *`client_username`* should be replaced by the username of the client who went AFK, where *`channel_name`* should be replaced by the name of the channel that the AFK client stayed in.

Server – Handling Standard Input

While the `chatserver` supports concurrent communication between multiple clients across multiple channels (i.e. broadcasts non-command messages from one client to all other clients (if any) in the same channel, it can also accept commands from its `stdin`. The format of the commands is the same as for the `chatclient` – they will start with a `/` and be terminated by a single newline. No leading or trailing spaces are to be accepted but arguments may be separated by multiple spaces.

`chatserver` detects commands by the keyword after the `/`. If the keyword after the `/` is not one of the valid commands (including a following space), `chatserver` should ignore this invalid command. If the keyword after

the `/` is one of the valid commands (including the following space) but the command has an invalid number of arguments, then `chatserver` should print a usage message (terminated by a newline) to `chatserver`'s `stdout` to show the correct usage of this command.

For example, if `chatserver` reads a line like this from `stdin`:

```
/kick channel_abc
```

(which is missing the *`client_username`* argument), then `chatserver` must print the following usage message (terminated by a newline) to its `stdout`:

```
Usage: /kick channel_name client_username
```

The content of this usage message depends on the command keyword. Valid commands (which define the messages to be printed) are listed below.

- **`/kick` *`channel_name client_username`***
  This command indicates that `chatserver` wishes to kick a client out of a certain channel, where *`channel_name`* is the name of the channel that the server wishes to perform the kick operation on, and *`client_username`* is the name of the client that the server tries to kick out.

  If there does not exist a channel with the given *`channel_name`*, then a message will be printed to `chatserver`'s stdout:

  `[Server Message] Channel "`*`channel_name`*`" does not exist.`

  where *`channel_name`* should be replaced by the channel name given in the `/kick` command. The double quotes must be present.

  If there does not exist a client named *`client_username`* in the channel (i.e. active in the channel – the queue is not looked at), then a message (terminated by a newline) will be printed to `chatserver`'s `stdout`:

  `[Server Message]` *`client_username`* `is not in the channel.`

  where *`client_username`* should be replaced by the client name given in the `/kick` command.

  `chatserver` checks the validity of the `/kick` command based on the order given above. For example, if both *`channel_name`* and *`client_username`* do not exist, the server will only print this message (terminated by a newline) to `stdout`:

  `[Server Message] Channel "`*`channel_name`*`" does not exist.`

  If the `/kick` command is valid, `chatserver` proceeds with the kick operation. During the kick operation, the client is removed from the channel. The client who is kicked prints a message (terminated by a newline) to its `stdout`:

  `[Server Message] You are removed from the channel.`

  After this message, the connection between the client and the server is closed. The client will exit and will not print a message about the connection being closed. `chatserver` prints a message (terminated by a newline) to its `stdout`:

  `[Server Message] Kicked` *`client_username`*`.`

  where *`client_username`* should be replaced by the client name given in the `/kick` command.

  Finally, `chatserver` notifies every other client in the channel about this kick operation. Each client prints a message (terminated by a newline) to its `stdout`:

  `[Server Message]` *`client_username`* `has left the channel.`

  where *`client_username`* should be replaced by the client name given in the `/kick` command. This same message is transmitted to remaining channel members anytime a client leaves a channel. Note that this does not apply when a client leaves a queue and does not apply when a channel is emptied in response to an `/empty` command – all clients are considered to leave the channel simultaneously.

- **`/shutdown`**
  This command indicates that the entire `chatserver` will shut down. All connection(s) for every client in all channel(s) and queue(s) will closed without any prior notification to the client or clients.

  `chatserver` prints the following message (terminated by a newline) to its `stdout`:

  `[Server Message] Server shuts down.`

Then, the `chatserver` process will exit normally with exit status 0. Clients will output that the server connection has closed as per line 249.

EOF on the server's `stdin` should be treated the same as if the `/shutdown` command was entered.

- **`/mute` *`channel_name client_username duration`***
  This command indicates that the server wishes to mute a client in a certain channel for a while. *`channel_name`* should be replaced by the name of the channel to perform this mute operation on, *`client_username`* should be replaced by the name of the client that the server wishes to mute, and *`duration`* should be replaced by the duration in seconds that the client will be muted for, which must be a positive integer. If there does not exist a channel with the given *`channel_name`*, then a message will be printed to `chatserver`'s stdout:

  `[Server Message] Channel "`*`channel_name`*`" does not exist.`

  where *`channel_name`* should be replaced by the channel name given in the `/mute` command. The double quotes must be present.

  If there does exist a client named *`client_username`* in the channel, then a message (terminated by a newline) will be printed to `chatserver`'s `stdout`:

  `[Server Message]` *`client_username`* `is not in the channel.`

  where *`client_username`* should be replaced by the client username given in the `/mute` command.

  If the *`duration`* is invalid (not a positive integer), then a message (terminated by a newline) will be printed to `chatserver`'s `stdout`:

  `[Server Message] Invalid mute duration.`

  `chatserver` checks the validity of the `/mute` command based on the order given above. For example, if both *`client_username`* and *`duration`* are not valid, the server will only print this message (terminated by a newline) to `stdout`:

  `[Server Message]` *`client_username`* `is not in the channel.`

  where *`client_username`* should be replaced by the client username given in the `/mute` command.

  If the `/mute` command is valid, then `chatserver` proceeds with the mute operation. During the mute operation, `chatserver` prevents the client named *`client_username`* in channel *`channel_name`* from sending any messages to any other client or clients for *`duration`* seconds. The muted client will still be able to send commands (except command `/whisper` or `/send`) to `chatserver` and receive the response or responses from `chatserver` normally.

  After the client is muted, `chatserver` prints a message to its stdout:

  `[Server Message] Muted` *`client_username`* `for` *`duration`* `seconds.`

  where *`client_username`* should be replaced by the client username given in the `/mute` command, and *`duration`* should be replaced by the mute duration given in the `/mute` command.

  The client who is muted will be notified by `chatserver` and prints a message (terminated by a newline) to `stdout`:

  `[Server Message] You have been muted for` *`duration`* `seconds.`

  where *`duration`* should be replaced by the mute duration given in the `/mute` command.

  All other clients in the same channel will be notified by `chatserver` and will print a message (terminated by a newline) to their `stdout`:

  `[Server Message]` *`client_username`* `has been muted for` *`duration`* `seconds.`

  where *`client_username`* should be replaced by the client username given in the `/mute` command.

  If the muted client tries to broadcast a message or `/whisper` or `/send`, then a message (terminated by a newline) will be printed to the client's `stdout`:

  `[Server Message] You are still in mute for` *`duration`* `seconds.`

  where *`duration`* should be replaced by the mute duration given in the `/mute` command (i.e. the original duration given in the `/mute` command line).

If the client leaves the channel then the mute is cancelled, even if they immediately rejoin the channel. If a mute operation is performed on a client whilst they are muted, then the mute period is restarted with the new duration and messages are sent as above.

Except broadcasting message, `/whisper` and `/send`, all other client commands are usable during the mute.

- `/empty` *`channel_name`*
  This command indicates that the server wishes to remove all the clients in a specific channel. *`channel_name`* is the name of the channel to perform this empty operation on. If there does not exist a channel with the given *`channel_name`*, then a message (terminated by a newline) will be printed to `chatserver`'s `stdout`:

  `[Server Message] Channel "`*`channel_name`*`" does not exist.`

  where *`channel_name`* should be replaced by the channel name given in the `/empty` command. The double quotes must be present.

  If there exists a channel called *`channel_name`*, `chatserver` will notify all the clients in the channel. Every client in the channel will print a message (terminated by a newline) to its `stdout`:

  `[Server Message] You are removed from the channel.`

  Then all connection(s) in this channel will be closed. Note: this `/empty` command does not close the connection(s) for clients waiting in the queue. Hence, the client or clients waiting in the queue of this channel will be moved into the channel in the FIFO order until it reaches the channel's capacity.

  After emptying the channel and before other clients are added, `chatserver` prints the following message (terminated by a newline) to its `stdout`:

  `[Server Message] "`*`channel_name`*`" has been emptied.`

  where *`channel_name`* should be replaced by the name of the channel to empty given in the `/empty` command. The double quotes must be present.

Client Disconnection

If a client disconnects or there is a communication error on the socket (e.g. a `read()` or equivalent from the client returns `EOF`, or a `write()` or equivalent fails) then `chatserver` is to close the connection to that client. Communication with other clients and the `chatserver` program itself must continue uninterrupted. The `chatserver` program should not terminate under normal circumstances and must not exit in response to a `SIGPIPE`.

Other Requirements

- If a client is muted by the `chatserver`, the timer for AFK should freeze until the client becomes unmuted. If a client attempts to chat or use a command while muted, the timer should not be reset.
- We will not test for unexpected system call or library failures in an otherwise correctly-implemented program (e.g. resource allocation failure). Your program can behave however it likes in these cases, including crashing.
- Only one `chatserver` process should run at any time.
- For any aspects of behaviour not described in this specification, your program must behave in the same manner as `demo-chatserver` on `moss`. If you're unsure about some aspect, then you can run the demo program to check the expected behaviour.

Testing

You are responsible for ensuring that your program operates according to the specification. You are encouraged to test your program on a variety of scenarios. A variety of programs will be provided to help you in testing:

- Two demonstration programs (called `demo-chatserver` and `demo-chatclient`) that implement the correct behaviour will be made available on `moss`. You can use them to check the expected behaviour of the programs if some part of this specification is unclear.

- Remember that you can use netcat (`nc`) to do testing also – you can use netcat as a client to communicate with your server, or as a server that your client can communicate with. This will allow you to simulate and capture requests and responses between the server and client.
- A test script will be provided on `moss` that will test your programs against a subset of the functionality requirements – approximately 50% of the available functionality marks. The script will be made available about 7 to 10 days before the assignment deadline and can be used to give you some confidence that you're on the right track. The "public tests" in this test script will not test all functionality and you should be sure to conduct your own tests based on this specification. The "public tests" will be used in marking, along with a set of "private tests" that you will not see.
- The Gradescope submission site will also be made available about 7 to 10 days prior to the assignment deadline. Gradescope will run the test suite immediately after you submit. When this is complete[7] you will be able to see the results of the "public tests". You should check these test results to make sure your program is working as expected. Behaviour differences between `moss` and Gradescope may be due to memory initialisation assumptions in your code, so you should allow enough time to check (and possibly fix) any issues after submission.

Style

Your submission must comply with the *Documentation required for code referencing and the use of AI tools* if applicable.

Hints

1. Review the lecture sample code related to network clients, threads and synchronisation (semaphores), and multi-threaded network servers. This assignment builds on all of these concepts.
2. Remember to flush output to `stdout` and network sockets if using high level output primitives (e.g. `fprintf()` in C). Output may not be newline buffered.
3. You will need to use appropriate mutual exclusion in your server to avoid race conditions when accessing common data structures and shared resources.

Submission

You must submit a zip file to Gradescope that contains your code at the top level of the zip file. Only files matching the following names will be extracted and considered in marking:

- *.c
- *.h
- *.py
- Makefile (or makefile)
- toolHistory.txt

Other files and any subfolders in your zip file will be ignored for marking purposes. We will then build/check you code for marking as follows:

- If a Makefile (or makefile) exists then the marking script will run 'make' (with no target specified).
- If make was run and an executable file named '`chatclient`' was created then we will run your client as '`./chatclient`'.
- If a `chatclient` executable does not exist but '`chatclient.py`' exists then we will run your client as '`python3 ./chatclient.py`'.
- If neither '`chatclient`' nor '`chatclient.py`' exists then we will not test your client and you will receive zero marks for client functionality.

[7] Gradescope marking may take only a few minutes or more than 30 minutes depending on the functionality and performance of your code.

- If make was run and an executable file named '`chatserver`' was created then we will run your server as '`./chatserver`'.
- If a `chatserver` executable does not exist but '`chatserver.py`' exists then we will run your server as '`python3 ./chatserver.py`'.
- If neither '`chatserver`' nor '`chatserver.py`' exists then we will not test your server and you will receive zero marks for server functionality.

Marks

Marks will be awarded for functionality achieved. Marks may be reduced if you attend an interview about your assignment and you are unable to adequately respond to questions – see the COMS3200 Student Interviews section below.

Functionality

Provided your executables can be checked/generated as above, then you will earn functionality marks based on the number of features your program correctly implements, as outlined below. Not all features are of equal difficulty.

Partial marks will be awarded for partially meeting the functionality requirements. A number of tests will be run for each marking category listed below, testing a variety of scenarios. Your mark in each category will be proportional (or approximately proportional) to the number of tests passed in that category.

If your program does not allow a feature to be tested then you will receive zero marks for that feature, even if you claim to have implemented it. For example, if your client can never create a connection to a server then we can not determine whether it can send the correct requests or not.

Your tests must run in a reasonable time frame, which could be as short as a few seconds for usage checking to many tens of seconds in some cases. If your program takes too long to respond, then it will be terminated and you will earn no marks for the functionality associated with that test.

Exact text matching of output (`stdout` and `stderr`) is used for functionality marking. Strict adherence to the formats in this specification is critical to earn functionality marks.

The markers will make no alterations to your code (other than to remove code without academic merit).

Marking Scheme (80 marks)

Marks will be assigned in the following categories. Note that, other than for the the first few categories, your client and your server will be tested together and both the client and server need to be working correctly together to earn marks. Note that some functionality will be tested in multiple categories, e.g. correct queuing functionality is required when testing responses to the server `/empty` command.

1. `chatclient` correctly handles invalid command lines. (2 marks)
2. `chatserver` correctly handles invalid command lines. (2 marks)
3. `chatclient` correctly handles inability to establish a connection to server. (1 mark)
4. `chatserver` correctly handles invalid configuration file. (3 marks)
5. `chatserver` correctly handles inability to listen on port(s). (2 marks)
6. `chatserver` listens on all the ports required and multiple clients can establish connections to the server. `chatclient` and `chatserver` print appropriate messages. (No queueing or channel messages.) (4 marks)
7. `chatclient` and `chatserver` correctly handle messages sent to a channel (including testing with multiple channels and clients). (5 marks)
8. `chatclient` and `chatserver` correctly handle queuing behaviour when channels reach capacity. (5 marks)
9. `chatclient` and `chatserver` correctly handle client `/send` command. (8 marks)
10. `chatclient` and `chatserver` correctly handle client `/quit` command. (3 marks)
11. `chatclient` and `chatserver` correctly handle client `/list` command. (4 marks)
12. `chatclient` and `chatserver` correctly handle client `/whisper` command. (5 marks)
13. `chatclient` and `chatserver` correctly handle client `/switch` command. (5 marks)

14. `chatclient` and `chatserver` correctly handle server `/kick` command. (5 marks)
15. `chatclient` and `chatserver` correctly handle server `/shutdown` command. (3 marks)
16. `chatclient` and `chatserver` correctly handle server `/mute` command. (10 marks)
17. `chatclient` and `chatserver` correctly handle server `/empty` command. (4 marks)
18. `chatclient` and `chatserver` correctly handle AFK client(s). (7 marks)
19. `chatclient` and `chatserver` correctly handle communication failure (unexpected disconnection and SIGPIPE when writing). (Note that the impact of client termination will be tested in many tests above.) (2 marks)

COMS3200 Student Interviews

The teaching staff will conduct interviews with a subset of COMS3200 students about their submissions, for the purposes of establishing genuine authorship. If you write your own code, you have nothing to fear from this process. If you legitimately use code from other sources (following the usage/referencing requirements outlined in this assignment, the style guide, and the AI tool use documentation requirements) then you are expected to understand that code. If you are not able to adequately explain the design of your solution and/or adequately explain your submitted code (and/or earlier versions in your repository) and/or be able to make simple modifications to it as requested at the interview, then your assignment mark will be scaled down based on the level of understanding you are able to demonstrate and/or your submission may be subject to a misconduct investigation where your interview responses form part of the evidence. Failure to attend a scheduled interview will result in zero marks for the assignment unless there are documented exceptional circumstances that prevent you from attending.

Students will be selected for interview based on a number of factors that may include (but are not limited to):

- Feedback from course staff based on observations in class, on the discussion forum, and during marking;
- Use of unusual or uncommon code structure/functions etc.;
- Referencing, or lack of referencing, present in code;
- Use of, or suspicion of undocumented use of, artificial intelligence or other code generation tools; and
- Reports from students or others about student work.

Specification Updates

Any errors or omissions discovered in the assignment specification will be added here, and new versions released with adequate time for students to respond prior to due date. Potential specification errors or omissions can be discussed on the discussion forum.

Version 1.1

- Removed italics from usage messages (lines 70 and 292) – changes not shown in red.
- Clarified that client usernames are unique across the queue also (line 88).
- Clarified that the `/whisper` command can have trailing spaces since these are part of the message (footnote to line 123).
- Clarified that an empty string configuration file is a usage error (line ~~307~~ 289).
- Clarified that fields in a configuration file line can be separated by one or more spaces – but we will only test single spaces (line 272).
- Clarified that EOF on the server's `stdin` should be treated the same as the `/shutdown` command (line 422).
- Clarified that EOF on the client's `stdin` should be treated the same as the `/quit` command (line 185) – as should termination by signal (other than the exit status).
- Clarified that `/kick` does not apply to clients in the queue (line 391).
- Clarified that the `/empty` command will not result in multiple "ahead of you" messages as clients move up the queue (line 350) and does not result in clients in the channel receiving "has left the channel" messages (line 412). Clarified also that clients in a channel are not informed of a client leaving a queue.

- Added that `/send` should support files up to $2^{32} - 1$ bytes in size and clarified that filenames may not contain spaces and existing files are to be overwritten (line 169).
- Added message to be printed if `/send` operation fails (line 171).
- Clarified client `SIGPIPE` behaviour (line 254).
- Clarified that client usernames can not contain spaces – this is a usage error (line 77).
- Clarified that it is permissible for a client to `/whisper` to itself (line 215).
- Clarified messages output by the client after `/kick` (line 403) and server `/shutdown` (line 420).
- Clarified `/switch` behaviour for clients in queues (lines 232 to 236.)

Version 1.2

- Moved clarification about empty string configuration file to list of usage errors as described above for v1.1 (line 289).
- Added that a sender cannot send a file to itself – an error message should be printed (line 139)
- Added a note that the impact of client termination will be tested in many marking tests (line 609)