COMP3331/9331 Computer Networks and Applications

Assignment for Term 1, 2021

Version 1.5

Due: ~~11:59am (noon) Friday, 23 April 2021 (Week 10)~~
11:59am (noon) Sunday, 25 April 2021 (Week 11)

Updates to the assignment, including any corrections and clarifications, will be posted on the subject website. Please make sure that you check the subject website regularly for updates.

1. Change Log

Version 1.0 released on 03/03/2021. Version 1.1 updated on 09/03/2021. Version 1.2 updated on 11/03/2021. Version 1.3 updated on 17/03/2021. Version 1.4 updated on 24/03/2021. Version 1.5 updated on 17/04/2021.

2. Goal and learning objectives

Zoom and Microsoft Team are widely used as a means for large groups of people to hold virtual meetings. A good example is the on-line Zoom lectures used for this course. In this assignment, you will have the opportunity to implement your own version of an online videoconferencing and messaging application. Your application is based on a client server model consisting of one server and multiple clients communicating concurrently. The text messages should be communicated using TCP for the reason of reliability, while the video (we will use video files instead of capturing the live video streams from cameras and microphones) should be communicated using UDP for the reason of low latency. Your application will support a range of functions that are typically found on videoconferencing including authentication, posting text message to all participants or one particular participant, uploading video streams (i.e., files in this assignment). You will be designing custom application protocols based on TCP and UDP.

2.1 Learning Objectives

On completing this assignment, you will gain sufficient expertise in the following skills:

1. Detailed understanding of how client-server and client-client interactions work.
2. Expertise in socket programming.
3. Insights into implementing an application layer protocol.

3. Assignment Specification

The base specification of the assignment is worth **20 marks**. The specification is structured in two parts. The first part covers the basic interactions between the clients and server and includes functionality for clients to communicate with the server. The second part asks you implement additional functionality whereby two clients can upload/download video files to each other directly in a peer-to-peer fashion via UDP. This first part is self-contained (Sections 3.2 – 3.3) and is worth

15 marks. Implementing video file uploading/downloading over UDP (Section 3.4) is worth **5 marks**. **CSE** students are expected to implement both functionalities. **Non-CSE** students are only required to implement the first part (i.e. no video file uploading/downloading over UDP). The marking guidelines are thus different for the two groups and are indicated in Section 7.

The assignment includes 2 major modules, the server program and the client program. The server program will be run first followed by multiple instances of the client program (Each instance supports one client). They will be run from the terminals on the same and/or different hosts.

Non-CSE Student: The rationale for this option is that students enrolled in a program that does not include a computer science component have had very limited exposure to programming and in particular working on complex programming assignments. A Non-CSE Student is a student who is not enrolled in a CSE program (single or double degree). Examples would include students enrolled exclusively in a single degree program such as Mechatronics or Aerospace or Actuarial Studies or Law. Students enrolled in dual degree programs that include a CSE program as one of the degrees do not qualify. Any student who meets this criterion and wishes to avail of this option MUST email cs3331@cse.unsw.edu.au to seek approval before **5pm**, **18 March (Friday, Week 5)**. We will assume by default that all students are attempting the CSE version of the assignment unless they have sought explicit permission. **No exceptions**.

3.1. Assignment Specification

In this programming assignment, you will implement the client and server programs of a video conference application, similar in many ways to the Zoom application that we use for this course. The difference being that your application won't capture and display live videos; instead, it will transmit and receive video files. The text messages must communicate over TCP to the server, while the clients communicate video files in UDP themselves. Your application will support a range of operations including authenticating a user, post a message to the server, edit or delete messages, read messages from server, read active users' information, and upload video files from one user to another user (**CSE Students only**). You will implement the application protocol to implement these functions. The server will listen on a port specified as the command line argument and will wait for a client to connect. The client program will initiate a TCP connection with the server. Upon connection establishment, the user will initiate the authentication process. The client will interact with the user through the command line interface. Following successful authentication, the user will initiate one of the available commands. All commands require a simple request response interaction between the client and server or two clients (**CSE Students only**). The user may execute a series of commands (one after the other) and eventually quit. Both the client and server MUST print meaningful messages at the command prompt that capture the specific interactions taking place. You are free to choose the precise text that is displayed. Examples of client server interactions are given in Section 8.

3.2 Authentication

When a client requests for a connection to the server, e.g., for attending a video conference, the server should prompt the user to input the username and password and authenticate the user. The valid username and password combinations will be stored in a file called credentials.txt which will be in the same directory as the server program. An example *credentials.txt* file is provided on the assignment page. Username and passwords are case-sensitive. We may use a different file for testing so DO NOT hardcode this information in your program. You may assume that each username and password will be on a separate line and that there will be one white space between the two. If the credentials are correct, the client is considered to be logged in and a welcome message is displayed. You should make sure that write permissions are enabled for the credentials.txt file (type "**chmod +w credentials.txt**" at a terminal in the current working directory of the server).

On entering invalid credentials, the user is prompted to retry. After a *number* of consecutive failed attempts, the user is blocked for a duration of 10 seconds (*number* is an integer command line argument supplied to the server and the valid value of *number* should be between 1 and 5) and cannot login during this 10 second duration (even from another IP address). If an invalid *number* value (e.g., a floating-point value, 0 or 6) is supplied to the server, the server prints out a message such as "Invalid number of allowed failed consecutive attempt: *number.* The valid value of argument number is an integer between 1 and 5".

For non-CSE Students: After a user logs in successfully, the server should record a timestamp of the user logging in event and the username in the active user log file (*userlog.txt,* you should make sure that write permissions are enabled for *userlog.txt*). Active users are numbered starting at 1:

```
Active user sequence number; timestamp; username

1; 19 Feb 2021 21:30:04; yoda
```

For CSE Students: After a user logs in successfully, the client should next send the UDP port number that it is listening to the server. The server should record a timestamp of the user logging in event, the username, the IP address (how?) and port number that the client listens to in the active user log file (*userlog.txt*):

```
Active user sequence number; timestamp; username; client IP address;
client UDP server port number

1; 19 Feb 2021 21:30:04; yoda; 129.64.1.11; 6666
```

For simplicity, a user will log in once in any given time, e.g., multiple logins concurrently are not allowed, and we won't test this case.

3.3. Text message operation

Following successful login, the client displays a message to the user informing them of all available commands and prompting to select one command. The following commands are available: MSG: Post Message, DLT: Delete Message, EDT: Edit Message, RDM: Read Message, ATU: Display active users, OUT: Log out and UPD: Upload file (for **CSE Students only**). All available commands should be shown to the user in the first instance after successful login. Subsequent prompts for actions should include this same message.

If an invalid command is selected, an error message should be shown to the user and they should be prompted to select one of the available actions.

In the following, the implementation of each command is explained in detail. The expected usage of each command (i.e., syntax) is included. **Note that, all commands should be upper-case (RDM, MSG, etc.)**. All arguments (if any) are separated by a single white space and will be one word long (except messages which can contain white spaces and timestamps that have a fixed format of `dd Mmm yyyy hh:mm:ss` such as `23 Feb 2021 16:01:20`). **You may assume that the message text may contain uppercase characters (A-Z), lowercase characters (a-z) and digits (0-9) and the following limited set of special characters (!@#$%.?,).**

If the user does not follow the expected usage of any of the operations listed below, i.e., missing (e.g., not specifying the body of a message when posting the message) or incorrect number of arguments (e.g., inclusion of additional or fewer arguments than required), an error message should be shown to

the user and they should be prompted to select one of the available commands. Section 8 illustrates sample interactions between the client and server.

There are 6 commands for **Non-CSE Students** and 7 commands for **CSE Students** respectively, which users can execute. The execution of each individual command is described below.

MSG: Post Message

```
MSG message
```

The message body should be included as the argument. Note that, the message may contain white spaces (e.g., "hello how are you"). The client should send the command (MSG), the message and the username to the server. **In our tests, we will only use short messages (a few words long)**. The server should append the message, the username, and a timestamp at the end of the message log file (file (*messagelog.txt,* you should make sure that write permissions are enabled for *messagelog.txt*) in the format, along with the number of the messages (messages are numbered starting at 1):

```
Messagenumber; timestamp; username; message; edited

1; 19 Feb 2021 21:39:04; yoda; do or do not, there is no try; no
```

After the message is successfully received at a server, a confirmation message with message number and timestamp should be sent from the server to the client and displayed to the user. If there is no argument after the MSG command. The client should display an error message before prompting the user to select one of the available commands.

DLT: Delete Message

```
DLT messagenumber timestamp
```

The message number to be deleted and the message's timestamp should be included as arguments. **A message can only be deleted by the user who originally posted that message**. The client sends the command (DLT), the message number, its timestamp and the username to the server. The server should check if the message number is valid, if the timestamp is correct, and finally if this user had originally posted this message. In the event that any of these checks are unsuccessful, an appropriate error message should be sent to the client and displayed at the prompt to the user. If all checks pass, then the server should delete the message, which entails deleting the line containing this message in the message log file (all subsequent messages in the file should be moved up by one line and their message numbers should be updated appropriately) and a confirmation should be sent to the client and displayed at the prompt to the user. The client should next prompt the user to select one of the available commands.

EDT: Edit Message

```
EDT messagenumber timestamp message
```

The message number to be edited, the message's timestamp and the new message should be included as arguments. **A message can only be edited by the user who originally posted that message**. The client should send the command (EDT), the message number, the original message's timestamp, the new message and the username to the server. The server should check if the corresponding message number is valid, if the message's timestamp is correct, and finally if the username had posted this message. In the event that any of these checks are unsuccessful, an appropriate error message should

be sent to the client and displayed at the prompt to the user. If all checks pass, then the server should replace the original message with the new message, update the timestamp, and marked the message as edited in the message log file (the rest of the details associated with this message, i.e., message number and username should remain unchanged).

```
Messagenumber; timestamp; username; message; edited

1; 19 Feb 2021 21:39:10; yoda; do or do not; yes
```

A confirmation should be sent to the client and displayed at the prompt to the user. The client should next prompt the user to select one of the commands.

RDM: Read Messages

```
RDM timestamp
```

The timestamp, after which the messages to be read, should be included as an argument. The client should send the command (RDM) and a timestamp to the server. The server should check if there are any new messages (i.e., the timestamps of the messages that are larger/later than the timestamp specified in the RDM message) in the message log file. If so, the server should send these new messages to the client. The client should display all received messages at the terminal to the user. If there is no new message exist, a notification message of "no new message" should be sent to the client and displayed at the prompt to the user. The client should next prompt the user to select one of the available commands.

ATU: Download Active Users

```
ATU
```

There should be no arguments for this command. The server should check if there are any other active users apart from the client that sends the `ATU` command. If so, the server should send the usernames, timestamp since the users are active, (and their IP addresses and Port Numbers, **CSE Students only)** in active user log file to the client (the server should exclude the information of the client, who sends `ATU` command to the server.). The client should display all the information of all received users at the terminal to the user. If there is no other active user exist, a notification message of "no other active user" should be sent to the client and displayed at the prompt to the user. The client should next prompt the user to select one of the available commands.

OUT: Log out

```
OUT
```

There should be no arguments for this command. The client should close the TCP connection, (UDP client server, CSE Students only) and exit with a goodbye message displayed at the terminal to the user. The server should update its state information about currently logged on users and the active user log file. Namely, based on the message (with the `username` information) from the client, the server should delete user, which entails deleting the line containing this user in the active user log file (all subsequent users in the file should be moved up by one line and their active user sequence numbers should be updated appropriately) and a confirmation should be sent to the client and displayed at the prompt to the user. Note that any messages uploaded by the user must not be deleted. For simplicity, we won't test the cases that a user forgets to log out or log out is unsuccessful.

3.4 Peer to Peer Communication (Video file upload, CSE Students only)

The P2P part of the assignment enables one client upload video files to another client using UDP. Each client is in one of two states, Presenter or Audience. The Presenter client sends video files the Audience client. Here, the presenter client is the UDP client, while the Audience client is the UDP server. After receiving the video files, the Audience client saves the files and the username of Presenter. Note that a client can behave in either Presenter or Audience state.

To implement this functionality your client should support the following command.

UPD: Upload file

UPD username filename

The Audience user and the name of the file should be included as arguments. You may assume that the file included in the argument will be available in the current working directory of the client with the correct access permissions set (read). You should not assume that the file will be in a particular format, i.e., just assume that it is a **binary file**. The Presenter client (e.g., `Yoda)` should check if the Audience user (indicated by the username argument, e.g., `Obi-wan`) is active (e.g., by issuing command `ATU)`. If `Obi-wan` is not active, the Presenter client should display an appropriate error message (e.g., `Obi-wan` is offline) at the prompt to `Yoda`. If `Obi-wan` is active, `Yoda` should obtain the `Obi-wan`'s address and UDP server port number (e.g., by issuing command `ATU)` before transferring the contents of the file to `Obi-wan` via **UDP**. Here, `Yoda` is the UDP client and `Obi-Wan` is the UDP server. The file should be stored in the current working directory of `Obi-wan` with the file name presenterusername_filename (DO NOT add an extension to the name. If the filename has an extension mp4, e.g., *test.mp4* should be stored as *yoda_test.mp4* in our example). File names are **case sensitive** and **one word long.** After the file transmission, the terminal of `Yoda` should next prompt the user to select one of the available commands. The terminal of `Obi-wan` should display an appropriate message, e.g., a file (*test.mp4*) has been received from `Yoda,` before prompting the user to select one of the available commands.

TESTING NOTES: 1) When you are testing your assignment, you may run the server and multiple clients on the same machine on separate terminals. In this case, use 127.0.0.1 (local host) as the destination (e.g., `Obi-wan's` in our example above) IP address. 2) For simplicity, we will run different clients at different directories, and won't test the scenario that a file is received when a user is typing/issuing a command.

3.5 File Names & Execution

The main code for the server and client should be contained in the following files: `server.c`, or `Server.java` or `server.py`, and `client.c` or `Client.java` or `client.py`. You are free to create additional files such as header files or other class files and name them as you wish.

The server should accept the following two arguments:

- `server_port`: this is the port number which the server will use to communicate with the clients. Recall that a TCP socket is NOT uniquely identified by the server port number. So it is possible for multiple TCP connections to use the same server-side port number.

- `number_of_consecutive_failed_attempts`: this is the number of consecutive unsuccessful authentication attempts before a user should be blocked for **10 seconds**. It should be an integer between 1 and 5.

The server should be executed before any of the clients. It should be initiated as follows:

If you use Java:

```
java Server server_port number_of_consecutive_failed_attempts
```

If you use C:

```
./server server_port number_of_consecutive_failed_attempts
```

If you use Python:

```
python server.py server_port number_of_consecutive_failed_attempts
```

The client should accept the following three arguments:

- `server_IP`: this is the IP address of the machine on which the server is running.
- `server_port`: this is the port number being used by the server. This argument should be the same as the first argument of the server.
- `client_udp_port`: this is the port number which the client will listen to/wait for the UDP traffic from the other clients.

Note that, you do not have to specify the TCP port to be used by the client. You should allow the OS to pick a random available port. Similarly, you should allow the OS to pick a random available UDP source port for the UDP client. Each client should be initiated in a separate terminal as follows:

For non-CSE Students:

If you use Java:

```
java Client server_IP server_port
```

If you use C:

```
./client server_IP server_port
```

If you use Python:

```
python client.py server_IP server_port
```

For CSE Students:

If you use Java:

```
java Client server_IP server_port client_udp_server_port
```

If you use C:

```
./client server_IP server_port client_udp_server_port
```

If you use Python:

```
python client.py server_IP server_port client_udp_server_port
```

Note: 1) The additional argument of client_udp_server_port for **CSE Students** for the P2P UDP communication described in Section 3.4. In UDP P2P communication, one client program (i.e., Audience) acts as UDP server and the other client program (i.e., Presenter) acts as UDP client. 2) When you are testing your assignment, you can run the server and multiple clients on the same

machine on separate terminals. In this case, use 127.0.0.1 (local host) as the server IP address.

3.6 Program Design Considerations

Client Design

The client program should be fairly straightforward. The client needs to interact with the user through the command line interface and print meaningful messages. Section 8 provides some examples. **You do not have the use the exact same text as shown in the samples.** Upon initiation, the client should establish a TCP connection with the server and execute the user authentication process. Following authentication, the user should be prompted to enter one of the available commands. Almost all commands require simple request/response interactions between the client with the server. Note that, the client does not need to maintain any state about the videoconferencing.

For CSE Students, the client program also involves P2P communication using UDP. Similar to above, the user should be prompted to enter the available P2P communication command: UPD. This function should be implemented using a new thread since the user may want to interact with the client program when the file is uploading. The thread will end when the upload finishes. Similarly, the client UDP server should be implemented with another thread. However, this thread should be run until the client logs off since it is a UDP server thread. You should be particularly careful about how multiple threads will interact with the various data structures. Code snippets for multi-threading in all supported languages are available on the course webpage.

Server Design

When the server starts up, the videoconference is empty – i.e., there exist no users. The server should wait for a client to connect, perform authentication and service each command issued by the client sequentially. Note that, you will need to define a number of data structures for managing the current state of the videoconference (e.g., active users and posts) and the server must be able to interact with multiple clients simultaneously. A robust way to achieve this to use multithreading. In this approach, you will need a main thread to listen for new connections. This can be done using the socket accept function within a while loop. This main thread is your main program. For each connected client, you will need to create a new thread. When interacting with one particular client, the server should receive a request for a particular operation, take necessary action and respond accordingly to the client and wait for the next request. You may assume that each interaction with a client is atomic. Consider that client A initiates an interaction (i.e., a command) with the server. While the server is processing this interaction, it cannot be interrupted by a command from another client B. Client B's command will be acted upon after the command from client A is processed. Once a client exits, the corresponding thread should also be terminated. You should be particularly careful about how multiple threads will interact with the various data structures. Code snippets for multi-threading in all supported languages are available on the course webpage.

4. Additional Notes

- This is **NOT** group assignment. You are expected to work on this individually.
- **Tips on getting started**: The best way to tackle a complex implementation task is to do it in stages. A good place to start would be to implement the functionality to allow a single user to login with the server. Next, add the blocking functionality for a number of unsuccessful attempts. Then extend this to handle multiple clients. Once your server can support multiple clients,

implement the functions for posting, editing, reading and deleting messages. Finally, implement the features of downloading active users and logging off. Note that, this may require changing the implementation of some of the functionalities that you have already implemented. Once the communication with the server is working perfectly, you can move on to peer-to-peer communication (**CSE Students only**). It is imperative that you rigorously test your code to ensure that all possible (and logical) interactions can be correctly executed. Test, test and test.

- **Application Layer Protocol:** Remember that you are implementing an application layer protocol for a videoconferencing software. We are only considered with the end result, i.e., the functionalities outlined above. You may wish to revisit some of the application layer protocols that we have studied (HTTP, SMTP, etc.) to see examples of message format, actions taken, etc.
- **Transport Layer Protocol:** You should use TCP for the communication between each client and server, (and UDP for P2P communication between two clients **CSE Students only**). The TCP connection should be setup by the client during the login phase and should remain active until the user logs off, while there is no such requirement for UDP. The server port of the server is specified as a command line argument. (Similarly, the server port number of UDP is specified as a command parameter of the client **CSE Students only)**. The client ports for both TCP and UDP do not need to be specified. Your client program should let the OS pick up random available TCP or UDP ports.
- **Backup and Versioning:** We strongly recommend you to back-up your programs frequently. CSE backups all user accounts nightly. If you are developing code on your personal machine, it is strongly recommended that you undertake daily backups. We also recommend using a good versioning system such as github or bitbucket so that you can roll back and recover from any inadvertent changes. There are many services available for both which are easy to use. We will NOT entertain any requests for special consideration due to issues related to computer failure, lost files, etc.
- **Language and Platform**: You are free to use C, JAVA or Python to implement this assignment. Please choose a language that you are comfortable with. The programs will be tested on CSE Linux machines. So please make sure that your entire application runs correctly on these machines (i.e., CSE lab computers) or using VLAB. This is especially important if you plan to develop and test the programs on your personal computers (which may possibly use a different OS or version or IDE). Note that CSE machines support the following: **gcc version 8.2, Java 11, Python 2.7 and 3.7. If you are using Python, please clearly mention in your report which version of Python we should use to test your code.** You may only use the basic socket programming APIs providing in your programming language of choice. You may not use any special ready-to-use libraries or APIs that implement certain functions of the spec for you.
- There is **no requirement** that you must use the same text for the various messages displayed to the user on the terminal as illustrated in the examples in Section 8. However, please make sure that the text is clear and unambiguous.
- You are encouraged to use the forums on WebCMS to ask questions and to discuss different approaches to solve the problem. However, you should **not** post your solution or any code fragments on the forums.
- We will arrange for additional consultation hours in Weeks 7 - 10 to assist you with assignment related questions if needed.

5. Submission

Please ensure that you use the mandated file name. You may of course have additional header files and/or helper files. If you are using C, then you MUST submit a makefile/script along with your code

(not necessary with Java or Python). This is because we need to know how to resolve the dependencies among all the files that you have provided. After running your makefile we should have the following executable files: `server` and `client`. In addition, you should submit a small report, `report.pdf` (no more than 3 pages) describing the program design, the application layer message format and a brief description of how your system works. Also discuss any design tradeoffs considered and made. Describe possible improvements and extensions to your program and indicate how you could realise them. If your program does not work under any particular circumstances, please report this here. Also indicate any segments of code that you have borrowed from the Web or other books.

You are required to submit your source code and `report.pdf`. You can submit your assignment using the give command in a terminal from any CSE machine (or using VLAB or connecting via SSH to the CSE login servers). Make sure you are in the same directory as your code and report, and then do the following:

1. Type tar -cvf assign.tar filenames
e.g. `tar –cvf assign.tar *.java report.pdf`

2. When you are ready to submit, at the bash prompt type `3331`

3. Next, type: `give cs3331 assign assign.tar` (You should receive a message stating the result of your submission). Note that, COMP9331 students should also use this command.

Alternately, you can also submit the tar file via the WebCMS3 interface on the assignment page.

Important notes

- The system will only accept `assign.tar` submission name. All other names will be rejected.
- **Ensure that your program/s are tested in CSE Linux machine (or VLAB) before submission. In the past, there were cases where tutors were unable to compile and run students' programs while marking. To avoid any disruption, please ensure that you test your program in CSE Linux-based machine (or VLAB) before submitting the assignment. Note that, we will be unable to award any significant marks if the submitted code does not run during marking.**

- You may submit as many times before the deadline. A later submission will override the earlier submission, so make sure you submit the correct file. Do not leave until the last moment to submit, as there may be technical, or network errors and you will not have time to rectify it.

Late Submission Penalty: Late penalty will be applied as follows:
- 1 day after deadline: 10% reduction
- 2 days after deadline: 20% reduction
- 3 days after deadline: 30% reduction
- 4 days after deadline: 40% reduction
- 5 or more days late: NOT accepted

NOTE: The above penalty is applied to your final total. For example, if you submit your assignment 1 day late and your score on the assignment is 10, then your final mark will be 10 – 1 (10% penalty) = 9.

6. Plagiarism

You are to write all of the code for this assignment yourself. All source codes are subject to strict checks for plagiarism, via highly sophisticated plagiarism detection software. These checks may include comparison with available code from Internet sites and assignments from previous semesters. In addition, each submission will be checked against all other submissions of the current semester. Do not post this assignment on forums where you can pay programmers to write code for you. We will be monitoring such forums. Please note that we take this matter quite seriously. The LIC will decide on appropriate penalty for detected cases of plagiarism. The most likely penalty would be to reduce the assignment mark to **ZERO**. We are aware that a lot of learning takes place in student conversations, and don't wish to discourage those. However, it is important, for both those helping others and those being helped, not to provide/accept any programming language code in writing, as this is apt to be used exactly as is, and lead to plagiarism penalties for both the supplier and the copier of the codes. Write something on a piece of paper, by all means, but tear it up/take it away when the discussion is over. It is OK to borrow bits and pieces of code from sample socket code out on the Web and in books. You MUST however acknowledge the source of any borrowed code. This means providing a reference to a book or a URL when the code appears (as comments). Also indicate in your report the portions of your code that were borrowed. Explain any modifications you have made (if any) to the borrowed code.

7. Marking Policy

You should test your program rigorously before submitting your code. Your code will be marked using the following criteria:

The following table outlines the marking rubric for both CSE and non-CSE students:

Functionality	**Marks (CSE)**	**Marks (non-CSE)**
Successful log in and log out for single client	0.5	0.5
Blocking user for 10 seconds after specified number of unsuccessful attempts (even from different IP)	1	1.5
Successful log in for multiple clients (from multiple machines)	1	2
Correct Implementation of MSG: Post Message	1.5	2
Correct Implementation of DLT: Delete Message	1.5	2
Correct Implementation of EDT: Edit Message	1.5	2
Correct Implementation of EDT: Read Message	2	3
Correct Implementation of ATU: Display active users	2	3
Properly documented report	2	2
Code quality and comments	2	2
Peer to peer communications including Correct Implementation of UPD: Upload file	5	N/A

NOTE: While marking, we will be testing for typical usage scenarios for the above functionality and some straightforward error conditions. A typical marking session will last for about 15 minutes during which we will initiate at most 5 clients. However, please do not hard code any specific limits in your programs. We won't be testing your code under very complex scenarios and extreme edge cases.

8. Sample Interaction

Note that the following list is not exhaustive but should be useful to get a sense of what is expected. We are assuming Java as the implementation language.

Case 1: Successful Login (underline denotes user input)

Terminal 1

```
>java Server 4000 3
```

Terminal 2

For Non-CSE Students:

```
>java Client 10.11.0.3 4000 (assume that server is executing on 10.11.0.3)

> Username: Yoda

> Password: comp9331

> Welcome to TOOM!

> Enter one of the following commands (MSG, DLT, EDT, RDM, ATU, OUT):

>
```

For CSE Students:

```
>java Client 10.11.0.3 4000 8000 (assume that server is executing on 10.11.0.3)

> Username: Yoda

> Password: comp9331

> Welcome to TOOM! (the client should upload the UDP port number 8000, i.e., the
second argument, to the server in the background after a user is log in).

> Enter one of the following commands (MSG, DLT, EDT, RDM, ATU, OUT, UPD):

>
```

Case 2: Unsuccessful Login (assume server is running on Terminal 1 as in Case 1, underline denotes user input)

The unsuccessful login examples below are for **Non-CSE Students.** For **CSE Students**, the client program should have an additional argument client_udp_server_port (see the example above with UDP port number 8000).

Terminal 2

```
>java Client 10.11.0.3 4000   (assume that server is executing on 10.11.0.3)
```

```
> Username: Yoda
> Password: comp3331
> Invalid Password. Please try again
> Password: comp8331
> Invalid Password. Please try again
> Password: comp7331
> Invalid Password. Your account has been blocked. Please try again later
```

The user should now be blocked for 10 seconds since the specified number of unsuccessful login attempts is 3. The terminal should shut down at this point.

Terminal 2 (reopened before 10 seconds are over)

```
>java Client 10.11.0.3 4000 8000 (assume that server is executing on 10.11.0.3)
> Username: Yoda
> Password: comp9331
> Your account is blocked due to multiple login failures. Please try again later
```

Terminal 2 (reopened after 10 seconds are over)

```
>java Client 10.11.0.3 4000 8000 (assume that server is executing on 10.11.0.3)
> Username: Yoda
> Password: comp9331
> Welcome to Toom!

> Enter one of the following commands (MSG, DLT, EDT, RDM, ATU, OUT):

>
```

Example Interactions (underline denotes user input)

Example Interactions 1 and 2 below are for **Non-CSE Students.** For **CSE Students**, the client command prompt has one more command UPD, i.e., (MSG, DLT, EDT, RDM, ATU, OUT, UPD), and ATU command returns extra active users' information including IP addresses and UDP port numbers. Please see Example Interaction 3 (P2P communication via UDP).

Consider a scenario where users `Yoda` and `Obi-wan` are currently logged in. In the following we will illustrate the text displayed at the terminals for all users and the server as the users execute various commands.

1. `Yoda` executes `MSG` command followed by a command that is not supported. `Obi-wan` executes `MSG`. `Yoda` and `Obi-wan` execute `log out`.

Yoda's Terminal	Obi-wan's Terminal	server's Terminal
> Enter one of the following commands (MSG, DLT, EDT, RDM, ATU, OUT): MSG Hello > Message #1 posted at 23 Feb 2021 15:00:01. > Enter one of the following commands (MSG,		> Yoda posted MSG #1 "Hello" at 23 Feb 2021 15:00:01.

DLT, EDT, RDM, ATU, OUT): whatsthedate > Error. Invalid command! > Enter one of the following commands (MSG, DLT, EDT, RDM, ATU, OUT):		
	> Enter one of the following commands (MSG, DLT, EDT, RDM, ATU, OUT): RDM 23 Feb 2021 15:00:00 > #1; Yoda: "Hello" posted at 23 Feb 2021 15:00:01. > Enter one of the following commands (MSG, DLT, EDT, RDM, ATU, OUT):	> Obi-Wan issued RDM command > Return messages: #1 Yoda: "Hello" posted at 23 Feb 2021 15:00:01.
> Enter one of the following commands (MSG, DLT, EDT, RDM, ATU, OUT): OUT > Bye, Yoda!		> Yoda logout
	> Enter one of the following commands (MSG, DLT, EDT, RDM, ATU, OUT): OUT > Bye, Obi-wan!	> Obi-wan logout

2. `Obi-wan` executes a series of valid commands followed by a `RDM` command issued by `Yoda`.

Yoda's Terminal	Obi-wan's Terminal	server's Terminal
	> Enter one of the following commands (MSG, DLT, EDT, RDM, ATU, OUT): MSG Hello > Message #2 posted at 23 Feb 2021 16:00:01. > Enter one of the following commands (MSG, DLT, EDT, RDM, ATU, OUT): MSG Compute Network Rocks > Message #3 posted at 23 Feb 2021 16:01:01. > Enter one of the following commands (MSG, DLT, EDT, RDM, ATU, OUT): EDT #3 23 Feb 2021 16:01:01 Computer Network Rocks (note computer instead of compute in the	> Obi-Wan posted MSG #2 "Hello" at 23 Feb 2021 16:00:01. > Obi-Wan posted MSG #3 "Compute Network Rocks" at 23 Feb 2021 16:01:01. > Obi-Wan edited MSG #3 "Computer Network Rocks"

	original message)	at 23 Feb 2021 16:01:10.
	> Message #3 edited at 23 Feb 2021 16:01:10.	
	> Enter one of the following commands (MSG, DLT, EDT, RDM, ATU, OUT): MSG Database Rocks	> Obi-Wan posted MSG #4 "Database Rocks" at 23 Feb 2021 16:01:20.
	> Message #4 posted at 23 Feb 2021 16:01:20.	
	> Enter one of the following commands (MSG, DLT, EDT, RDM, ATU, OUT): DLT #4 23 Feb 2021 16:01:20	> Obi-Wan deleted MSG #4 "Database Rocks" at 23 Feb 2021 16:01:25.
	> Message #4 deleted at 23 Feb 2021 16:01:25.	
	> Enter one of the following commands (MSG, DLT, EDT, RDM, ATU, OUT): MSG IoT Rocks	> Obi-Wan posted MSG #4 "IoT Rocks" at 23 Feb 2021 16:01:30.
	> Enter one of the following commands (MSG, DLT, EDT, RDM, ATU, OUT):	
> Enter one of the following commands (MSG, DLT, EDT, RDM, ATU, OUT): ATU		> Yoda issued ATU command ATU.
> Obi-wan, active since 23 Feb 2021 16:00:01.		> Return active user list: Obi-wan, active since 23 Feb 2021 16:00:01.
> Enter one of the following commands (MSG, DLT, EDT, RDM, ATU, OUT): RDM 23 Feb 2021 16:01:00		> Yoda issued RDM command
		> Return messages:
> #3, Obi-wan: Computer Network Rocks, edited at 23 Feb 2021 16:01:10.		#3 Obi-wan, Computer Network Rocks, edited at 23 Feb 2021 16:01:10.
#4 Obi-wan, IoT Rocks, posted at 23 Feb 2021 16:01:30		#4 Obi-wan, IoT Rocks, posted at 23 Feb 2021 16:01:30
		(note: 1) Message #3: edited instead of posted, 2) Message #2 was posted earlier than 23 Feb 2021 16:01:00, thus is not included, 3) Message #4 has been deleted and is replaced by a new message.
> Enter one of the following commands (MSG, DLT, EDT, RDM, ATU, OUT):		
EDT #3 23 Feb 2021 16:01:10 AI Rocks		> Yoda attempts to edit MSG #3 at 23 Feb 2021 16:02:10. Authorisation fails.
> Unauthorised to edit Message #3.		
> Enter one of the following commands (MSG,		

DLT, EDT, RDM, ATU, OUT):		

3. P2P communication via UDP **CSE-students only**. Before `Yoda` uploads a video file lecture1.mp4 to `Obi-wan`, `Yoda` issues the ALT command to find out the IP address and UDP server port number of `Obi-wan`.

Yoda's Terminal	**Obi-wan's Terminal**	**server's Terminal**
> Enter one of the following commands (MSG, DLT, EDT, RDM, ATU, OUT, UPD): ATU > Obi-wan, 129.129.2.1, 8001, active since 23 Feb 2021 16:00:01. Han, 129.128.2.1, 9000, active since 23 Feb 2021 16:00:10 > Enter one of the following commands (MSG, DLT, EDT, RDM, ATU, OUT, UPD): UPD Obi-wan lecture1.mp4 > lecture1.mp4 has been uploaded >> Enter one of the following commands (MSG, DLT, EDT, RDM, ATU, OUT, UPD):	> Enter one of the following commands (MSG, DLT, EDT, RDM, ATU, OUT, UPD): > Received lecture1.mp4 from Yoda > Enter one of the following commands (MSG, DLT, EDT, RDM, ATU, OUT, UPD): (For simplicity, we won't test the scenario that a file is received, when a user is typing/issuing a command.)	> Yoda issued ATU command ATU. > Return active user list: Obi-wan; 129.129.2.1; 8001; active since 23 Feb 2021 16:00:01. (assume that the IP address and UDP server port number of Obi-wan are 129.129.2.1 and 8001 respectively.) Hans; 129.128.2.1; 9000; active since 23 Feb 2021 16:00:10 (assume that Hans is active with this details). (note that the server is not aware of the P2P UDP communication between Yoda and Obi-wan)