COMP3331/9331 Computer Networks and Applications

Assignment for Term 3, 2023

Version 5.0

16:59 PM on Sunday, November 12, 2023 (AEST) (Week 9)

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

1. Version 4.0 was released on October 23, 2023.
2. Section 3.2: Clarifications on failed login attempts.
3. Section 3.3: Clarifications on timestamps and messages for sending and receiving ends.
4. Section 3.3: Further explanations and clarifications on group chat service-related commands.
5. Section 3.3: Clarification on Private chat log username.
6. Section 3.3: Clarification on Group chat log username.
7. Fixing timestamp inconsistencies in displayed messages.
8. Changes to the deadline.
9. Added sample interactions on group commands and fixed typos.

2. Goal and learning objectives.

Messenger and WhatsApp are widely used as a method for large groups of people to have conversations online. In this assignment, you will have the opportunity to implement your own version of a messaging and video talk 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 messaging apps including authentication, message one participant (i.e., private chat), build the group chat for part of the participants, and 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 designing and 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 to 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 mark**. **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 limited exposure to programming and in particular working on complex programming assignments. A Non-CSE Student is one 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 do not qualify. Any student who meets this criterion and wishes to avail themselves of this option MUST email cs3331@cse.unsw.edu.au to seek approval before **5 pm**, **13 October 2023 (AEST) (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**.

In this programming assignment, you will implement the client and server programs of a messaging application, similar in many ways to the Messenger/WhatsApp applications that we use for this course. The difference being that your application will not 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, exchanging messages with the server, send private message to another particular participant, 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 can choose the precise text displayed. Examples of client server interactions are given in Section 8.

Allowed libraries include **all standard libraries** (such as socket, datetime, re, logging...etc. for Python 3 and equivalent libraries for other languages). Any library that needs to be installed in advance will not be accepted. It is recommended to test your code in the VLAB environment, where it will be tested.

3.2 Authentication

When a client requests a connection to the server, e.g., to attend 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. Usernames 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 logged in and a welcome message is displayed. You should make sure that writing 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 (username or password), the user is prompted to retry. After several 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; 01 Jun 2022 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 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; 01 Jun 2022 21:30:04; Yoda; 129.64.1.11; 6666
```

For simplicity, a user will log in once at any given time, e.g., **multiple logins concurrently are not allowed, and we will not test this case**. Blocking should be based on the user's username, not the session created. For instance, if a user has three failed login attempts on one terminal and then two failed attempts on another terminal (after closing the first terminal), the username should be blocked for a specific duration.

About logging

Log files, such as *'userlog.txt'* and *'messagelog.txt,'* must be generated automatically when clients or servers start. Students should refrain from creating them manually. You have the choice to employ a built-in logging library, such as Python's 'logging' module, or create a custom logging function.

3.3. Text message commands

Following successful login, the client displays a message to the users informing them of all available commands and prompting them to select one command. The following commands are available:

- **/msgto:** Private message, which the user launches a private chat with another active user and send private messages,
- **/activeuser:** Display active users,
- **/creategroup:** Group chat room service, which user can build group chat for multiple users
- **/joingroup:** Group chat room service, which user can join the already created group chat
- **/groupmsg:** Group chat message, user can send the message to a specific group and all the users in the group will receive the message,
- **/logout:** Log out
- **/p2pvideo:** send video file to another active user directly via UDP socket (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 lower-case (/msgto, /logout 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 :mm yyyy hh:mm:ss such as 0 1 Oct 2023 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.

Private Message

`/msgto` *`USERNAME MESSAGE_CONTENT`*

The message body should be included as the argument. Please note that the message may contain white spaces (e.g., 'Hi Wen, how are you'), but the entire message cannot consist of only white spaces. The client should send the command (i.e., /msgto), 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(receiver), 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). Please note that since there is a single "messagelog.txt" for all users, the message numbers are global across all users.

```
messageNumber; timestamp; username; message

1; 01 Jun 2022 21:39:04; Yoda; do or do not, there is no try
```

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

Active user list

```
/activeuser
```

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 `/activeuser` command. If so, the server should send the usernames, timestamp since the users became active (the time when users logged in), (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 `/activeuser` 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.

Group Chat Service

This function is composed of three commands: /creategroup group chat building, /joingroup group chat join, and /groupmsg group message sending. For simplicity, the functions such as exit a group chat and offline message will not be required. See below details for these commands.

```
/creategroup groupname username1 username2 ..
```

The client needs to practice /creategroup command to request the server to build a group chat room. Arguments are usernames which the current client wants to include in the separate chat room. If there is no argument after the /creategroup command, the client should display an error message before prompting the user one of the available commands. There is no limit to the number of usernames you can put but, in our testing, we will limit this to maximum 3 active users. If usernames that do not meet the specified conditions (invalid or offline) are used, an appropriate error message should be displayed, and the group should not be created.

The server is expected to check if the provided group chat name exists or not. The group chat name must be unique and only consist of letter a-z, A-Z and digit 0-9 (i.e. no space, symbol allowed). If the group chat is created already, the server should inform the client with a message of "a group chat (Name: Comp3331) already exist".

The server also needs to create a corresponding log file (e.g., GROUPNAME_messagelog.txt) to record the messages in the separate chat room. The server also needs to store the group chat members (either in a list, dictionary or file), the creator of the group chat will be the first member in the group. Finally, the server should reply to the client to confirm that the group chat room has been successfully created, for example, "Group chat room has been created, room name: Comp3331, users in this room: username1 username2 .."

After creating the group, the initially added users must join to the group before sending messages using the following command.

```
/joingroup groupname
```

If a user was not initially added to the group by the group creator (using /creategroup command), an appropriate error message should be displayed after running above command. Similarly, if a user has been initially added to the group but hasn't joined using the /joingroup command, an appropriate error message must also be displayed.

Once the /joingroup command is successful, a confirmation message should be displayed on the joining user's terminal, and the user is then ready to use the following command to send a message to the group.

```
/groupmsg groupname message
```

The group chat name and message body should be included as the argument. If there is no argument after the /groupmsg command. The client should display an error message before prompting the user to select one of the available commands.

The group chat name as the first argument indicates which group chat the client wants to send a message (assume the client may be in multiple group chats). The server should firstly check if the group chat with provided group chat name. If the room with that group chat name does not exist, the server should reply to the client with a message of "The group chat does not exist." And the server also needs to check if the client (the client who sends the /groupmsg commands) is a member of the group chat (both added and joined), if the client is not in this group chat the server should reply to the client with a message of "You are not in this group chat." Similarly, if the user was added but not joined yet, appropriate error message should be displayed.

If everything is good, the server should append the message, the username (of the sender), and a timestamp at the end of the message log file (file *GROUPNAME_messageLog.txt*, you should make sure that write permissions are enabled for *GROUPNAME_messageLog.txt*) in the format, along with the number of the messages (messages are numbered starting at 1 for each new created group chat):

```
messageNumber; timestamp; username; message
```

```
1; 01 Jun 2022 21:39:04; Yoda; do or do not, there is no try
```

After the message is successfully received by the server, a confirmation message is sent from the server to the client, indicating that the group message has been sent and client should display it in the terminal. The receiving user(s) should display the timestamp, group name, sender's username, and the message in their terminals. This should be done for all active group members except the sender. If there are no other active users in the group, the message should be logged.

After using any commands of the group chat service and receiving messages, the client should prompt the user to select from the available commands.

Display Messages

```
No command required.
```

The client should receive the messages from the server and display them automatically.

For private messages, the format:
`Time, Username: message content`
(e.g. 01 Jun 2023 15:00:01, Yoda: How are you today?)

For group message, the format is:
`Time, GroupChatName, Username: message content`
(e.g. 01 Jun 2023 18:00:01, Comp3331,Yoda: How are you guys?)

Log out

`/logout`

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 return 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 changed 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 sharing, 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.

You can safely assume that the network is reliable, meaning that metadata, including file names, will not be lost during transmission. You are also not required to guarantee the sequential delivery of packets. You can assemble the packets in the order you receive them at the receiver's end.

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

P2PVIDEO: Video File Sharing

```
/p2pvideo 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 `/activeuser)`. 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 `/activeuser)` 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) We will be testing whether the sent video file at presenters and received video file at the audience is identical. 4) Your code will be tested in the VLAB environment, it is recommended to test your code in your VLAB environment before the submission. 5) You must use UDP sockets for this function. We will test this and using TCP for file transfer will cause **ZERO** marks for this part. 6) You don't need to maintain the server's state. In the event of a server restart, users should log in again; automatic login after a server restart is not required. This applies for private messages, groups, and group messages as well.

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.

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 /activeuser 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 10.2.1, Java 11.0.20, Python 2.7.18 and 3.9.2. 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 Ed Forum 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 have arranged a programming specific Tutorial on Week 7 during normal lab hours for discussing the multithreaded component of the assignment. More details to follow.
- We will arrange for additional consultation hours in Weeks 7 - 9 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 realize 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 as possible 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:

- **Deadline: 16:59 PM on Sunday, November 12, 2023 (AEST)**
- 1 day after the deadline: 16:59 PM on Monday, November 13, 2023 (AEST) - 5% reduction
- 2 days after the deadline: 16:59 PM on Tuesday, November 14, 2023 (AEST) - 10% reduction
- 3 days after the deadline: 16:59 PM on Wednesday, November 15, 2023 (AEST) - 15% reduction
- days after the deadline: 16:59 PM on Thursday, November 16, 2023 (AEST) - 20% reduction
- or more days late: Not accepted.

NOTE: The above penalty is applied to your final total. For example, if you submit your assignment on or before 16:59 PM on Monday, November 13, 2023 (AEST) and your score on the assignment is 10, then your final mark will be 10 – 0.5 (5% penalty) = 9.5.

6. Plagiarism

You are to write all the code for this assignment yourself. Submission of work even partly written by any other person or AI is not permitted. All source code is 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 an appropriate penalty for detected cases of plagiarism. The penalty would be to reduce the assignment mark to **ZERO**. We are aware that a lot of learning takes place in student conversations, and do not 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 paper 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)	1.5	1.5
Successful log in for multiple clients (from multiple terminals)	1	2
Correct Implementation of /activeuser: Display active users	1	1
Correct Implementation of /msgto: Private message	1	2
Correct Implementation of /creategroup : Group chat building	1	2
Correct Implementation of /joingroup : Group chat join	1	1
Correct Implementation of /groupmsg: Group chat message	2	3
Correct Implementation of Display message	2	3
Properly documented report	2	2
Code quality and comments	2	2
Peer to peer communications including Correct Implementation of /p2pvideo: 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 will not be testing your code under extraordinarily 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 6000 3
```

Terminal 2

For Non-CSE Students:

```
>java Client 10.11.0.3 6000 (assume that server is executing on 10.11.0.3)

> Please login

> Username: Yoda

> Password: comp9331

> Welcome to TESSENGER!

> Enter one of the following commands (/msgto, /activeuser, /creategroup,
  /joingroup, /groupmsg, /logout):

>
```

For CSE Students:

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

> Please login

> Username: Yoda

> Password: comp9331

> Welcome to TESSENGER!

> Enter one of the following commands (/msgto, /activeuser, /creategroup,
  /joingroup, /groupmsg, /p2pvideo ,/logout):

>
```

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)
> Please login
> 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)
> Please login
> 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)
> Please login
> Username: Yoda
> Password: comp9331
> Welcome to Tessenger!

> Enter one of the following commands (/msgto, /activeuser, /creategroup,
  /joingroup, /groupmsg, /logout):

>
```

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 /p2pvideo, and /activeuser 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 /msgto command followed by a command that is not supported. Obi-wan executes
/activeuser. Yoda and Obi-wan execute log out.

Yoda's Terminal	Obi-wan's Terminal	Server's Terminal
Enter one of the following commands (/msgto, /activeuser, /creategroup, /joingroup, /groupmsg, /p2pvideo ,/logout): /msgto Obi-wan hello	> Enter one of the following commands (/msgto, /activeuser, /creategroup, /joingroup, /groupmsg, /p2pvideo ,/logout):	> Yoda message to Obi-wan "hello" at 01 Jun 2023 15:00:01.
> message sent at 01 Jun 2023 15:00:01.	01 Jun 2023 15:00:01, Yoda: hello	
> Enter one of the following commands (/msgto, /activeuser, /creategroup, /joingroup, /groupmsg, /p2pvideo ,/logout): whatsyourname > Error. Invalid command! > Enter one of the following commands (/msgto, /activeuser, /creategroup, /joingroup, /groupmsg, /p2pvideo ,/logout):	> Enter one of the following commands (/msgto, /activeuser, /creategroup, /joingroup, /groupmsg, /p2pvideo ,/logout): /activeuser Yoda, active since 01 Jun 2023 15:00:00. Enter one of the following commands (/msgto, /activeuser, /creategroup, /joingroup, /groupmsg, /p2pvideo ,/logout):	> Obi-Wan issued /activeuser command Return messages: Yoda, active since 01 Jun 2023 15:00:00.
/logout		> Yoda logout
> Bye, Yoda!	/logout	> Obi-wan logout
	> Bye, Obi-wan!	

2. Obi-wan executes a valid commands /creategroup, followed by an invalid command /creategroup commands. Yoda executes /joingroup command and interacts with Obi-wan.

Yoda's Terminal	Obi-wan's Terminal	server's Terminal
> Enter one of the following commands (/msgto, /activeuser, /creategroup, /joingroup, /groupmsg, /p2pvideo ,/logout):	Enter one of the following commands (/msgto, /activeuser, /creategroup, /joingroup, /groupmsg, /p2pvideo ,/logout): /creategroup learning3331 > Please enter at least one more active users.	>Yoda is online >Obi-wan is online > Obi-wan issued /creategroup command > Return message: Group chat room is not created. Please enter at least one more active

		users.
	Enter one of the following commands (/msgto, /activeuser, /creategroup, /joingroup, /groupmsg, /p2pvideo ,/logout): /creategroup learning3331 Yoda	
	> Group chat created learning3331	>Obi-wan issued /creategroup command > Return message: Group chat room has been created, room name: learning3331, users in this room: Obi-wan, Yoda
/groupmsg learning3331 hello there Please join the group before sending messages. > Enter one of the following commands (/msgto, /activeuser, /creategroup, /joingroup, /groupmsg, /p2pvideo ,/logout):		> Yoda sent a message to a group chat, but Yoda hasn't been joined to the group.
/joingroup learning3331		> Yoda issued /joingroup command > Return message: Join group chat room successfully, room name: learning3331, users in this room: Obiwan, Yoda
>Joined the group chat: learning3331 successfully. > Enter one of the following commands (/msgto, /activeuser, /creategroup, /joingroup, /groupmsg, /p2pvideo ,/logout): /groupmsg learning3331 hello there >Group chat message sent.		> Yoda issued a message in group chat learning3331: 01 Jun 2023 16:05:00; Yoda; hello there
	> 01 Jun 2023 16:05:00, learning3331, Yoda: hello there	

	> Enter one of the following commands (/msgto, /activeuser, /creategroup, /joingroup, /groupmsg, /p2pvideo ,/logout): /creategroup learning3331 Yoda >Failed to create the group chat learning3331: group name exists!	> /creategroup command issued from Obj-wan > Return message Groupname learning3331 already exists.
> Enter one of the following commands (/msgto, /activeuser, /creategroup, /joingroup, /groupmsg, /p2pvideo ,/logout): /joingroup learning9331		> Yoda tries to join to a group chat that doesn't exist.
Groupchat learning9331 doesn't exist. > Enter one of the following commands (/msgto, /activeuser, /creategroup, /joingroup, /groupmsg, /p2pvideo ,/logout): /joingroup learning3331		> Yoda tries to re-join to a group chat learning3331
You have already joined Groupchat learning3331 > Enter one of the following commands (/msgto, /activeuser, /creategroup, /joingroup, /groupmsg, /p2pvideo ,/logout):		

3. P2P communication via UDP **CSE-students only**. Before `Yoda` uploads a video file lecture1.mp4 to `Obi-wan`, `Yoda` issues the `/activeuser` 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 (/msgto, /activeuser, /createdgroup, /joingroup, /groupmsg, /p2pvideo ,/logout): /activeuser > Obi-wan, 129.129.2.1, 8001, active since 01 Jun 2022 16:00:01. Hans, 129.128.2.1, 9000, active since 01 Jun 2022 16:00:10 Enter one of the following commands (/msgto, /activeuser, /createdgroup, /joingroup, /groupmsg, /p2pvideo ,/logout): /p2pvideo Obi-wan lecture1.mp4 > lecture1.mp4 has been uploaded >> Enter one of the following commands (/msgto, /activeuser, /createdgroup, /joingroup, /groupmsg, /p2pvideo ,/logout):	> Enter one of the following commands (/msgto, /activeuser, /createdgroup, /joingroup, /groupmsg, /p2pvideo ,/logout): > Received lecture1.mp4 from Yoda > Enter one of the following commands (/msgto, /activeuser, /createdgroup, /joingroup, /groupmsg, /p2pvideo ,/logout): (For simplicity, we won't test the scenario that a file is received, when a user is typing/issuing a command.)	> Yoda issued /activeuser command /activeuser. > Return active user list: Obi-wan; 129.129.2.1; 8001; active since 01 Jun 2022 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 01 Jun 2022 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)