COMP3331/9331 Computer Networks and Applications

Assignment for Term 3, 2021

Version 2.0

Due: 11:59 am (noon) Friday, 19 November 2021 (Week 10)

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 5th October 2021.
Version 2.0 released on 18th October 2021: when a user timeouts, the notification that the user has logged out should be sent to all other users.

2. Goal and learning objectives

Instant messaging applications such as WhatsApp, WeChat, Telegram, Signal, etc. are widely used with millions of subscribers participating in them globally. In this assignment, you will have the opportunity to implement your own version of an instant messaging application. In addition to basic messaging functionality, you will also implement many additional services that are available in many of the aforementioned applications. Your application is based on a client server model consisting of one server and multiple messaging clients. **The clients communicate using TCP (UDP cannot be used for this assignment)**. The server is mainly used to authenticate the clients and direct the messages (online or offline) between clients. Besides, the server also has to support certain additional functions (presence notification, blacklisting, timeout, etc.). You should also implement functionality that allows clients to send peer-to-peer messages to each other that bypasses the server.

2.1 Learning Objectives

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

1. Detailed understanding of how instant messaging services work.
2. Expertise in socket programming.
3. Insights into designing an application layer protocol.

3. Assignment Specification

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 each other through the server. The second part asks you implement additional functionality whereby two users can exchange messages with each other directly (i.e. bypassing the server) in a peer-to-peer fashion. This first part is self-contained (Sections 3.1-3.3) and is worth **14 marks**. Implementing peer-to-peer messaging (Section 3.4) is worth **6 marks**. **CSE** students are expected to implement both functionalities. **Non-CSE** students are only required to implement the first part (i.e. no peer-to-peer messaging). The marking guidelines are thus different for the two groups and are indicated in Section 7.

Non-CSE Student Version: 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, 15th October (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.

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 different terminals on the **same machine** (so you can use localhost, i.e., 127.0.0.1 as the IP address for the server and client in your program). All interaction with the clients will be through a command line interface.

3.1. Server

The server has the following responsibilities -

User Authentication -

You may assume that a credentials file called *credentials.txt* will be available in the current working directory of the server with the correct access permissions set (read and write). This file is NOT available at the client. The file will contain username and passwords of authorised users. They contain uppercase characters (A-Z), lowercase characters (a-z), digits (0-9) and special characters (~!@#$%^&*_-+=`|\(){}[]:;"'<>,.?/). An example *credentials.txt* file is provided on the assignment page. We will 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 separating the two. There will only be one password per username. There will be no empty lines in this file.

Upon execution, a client should first setup a TCP connection with the server. Assuming the connection is successful, the client should prompt the user to enter a username. The username should be sent to the server. The server should check the credentials file (*credentials.txt*) for a match. If the username exists, the server sends a confirmation message to the client. The client prompts the user to enter a password. The password is sent to the server, which checks for a match with the stored password for this user. The server sends a confirmation if the password matches or an error message in the event of a mismatch. An appropriate message (welcome or error) is displayed to the user. In case of a mismatch, the client is asked to enter the password again (see discussion on blocking later). If the username does not exist, it is assumed that the user is creating a new account and the sever sends an appropriate message to the client. The client prompts the user to enter a new password. You may assume the password format is as explained above (no need to check). The password is sent to the server. The server creates a new username and password entry in the credentials file (appending it as the last entry). A confirmation is sent to the client. The client displays an appropriate welcome message to the user. 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). After successful authentication, the client is considered as logged in (i.e., online).

When your assignment is tested with multiple concurrent clients, the server should also check that a new client that is authenticating with the server does not attempt to login with a username that is already being used by another active client (i.e., a username cannot be used concurrently by two clients). The server should keep track of all active users and check that the username provided by an authenticating client does not match with those in this list. If a match is found, then a message to this effect should be sent to the client and displayed at the prompt for the user and they should be prompted to enter a username.

As noted above, on entering an invalid password, the user is prompted to retry. After 3 consecutive failed attempts for a particular username, this user is blocked for a duration of *block_duration* seconds (*block_duration* is a command line argument supplied to the server) and cannot login during this duration. The client should quit in this instance.

Timeout - The server should check that all logged on users are active. If the server detects that the user has not issued any valid command for interacting with the server or for peer-to-peer messaging for a period of *timeout* seconds (*timeout* is a command line argument supplied to the server), then the server should automatically log this user out. The receipt of a message or typing an invalid command does not count. There are several ways in which you can implement the timeout mechanism. We will leave the design to you.

Presence Broadcasts - The server should notify the presence/absence of other users logged into the server, i.e., send a broadcast notification to all online users when a user logs in and logs out. Note that, when a user is logged off due to timeout, a broadcast notification is sent to all online users.

List of online users - The server should provide a list of users that are currently online in response to such a query from a user.

Online history – The sever should provide a list of users that logged in for a user specified time in the past (e.g., users who logged in within the past 15 minutes).

Message Forwarding - The server should forward each instant message to the correct recipient assuming they are online.

Offline Messaging - When the recipient of a message is not logged in (i.e. is offline), the message will be saved by the server. When the recipient logs in next, the server will send all the unread messages stored for that user (timestamps are not required).

Message Broadcast – The server should allow a user to broadcast a message to all online users. Offline messaging is not required for broadcast messages.

Blacklisting - The server should allow a user to block / unblock any other user. For example, if user A has blocked user B, B can no longer send messages to A i.e. the server should intercept such messages and inform B that the message cannot be forwarded. Blocked users also do not get presence notifications i.e., B will not be informed each time A logs in or logs out. Blocked users are also unable to check the online status of the user blocking them, i.e., B will not be able to see if A is online currently or in the past (i.e., online history).

While marking, the server will be executed first and will remain online for the entire duration of the tests. We will not abruptly kill the server or client processes (CTRL-C). The server is not expected to maintain state from previous executions. When it is executed, it is assumed that no users are active.

3.2. Client

The client should interact with the user via the command line. The client has the following responsibilities -

Authentication - The client should provide a login prompt to enable the user to authenticate with the server. The authentication process was discussed in detail earlier.

Message - The client should allow the user to send a message to any other user and display messages sent by other users. The client should also allow the user to send a broadcast message to all online users. The message may contain uppercase characters (A-Z), lowercase characters (a-z), digits (0-9), special characters (~!@#$%^&*_-+=`|\(){}[]:;"'<>,.?/) and white spaces. During marking we will use short messages that are a few words long.

Notifications - The client should display presence notifications sent by the server about users logging in and out from the server.

Find users online - The client should provide a way for the user to obtain a list of all the users currently online from the server.

Find online history – The client should provide a way for the user to obtain a list of all users who had logged in within a user specified time period.

Blacklist – The client should allow a user to block a user from sending any further messages, receive presence notifications or check if they are currently online or were online in the past. The client should also allow a user to unblock a user that was earlier blocked.

3.3 Commands supported by the client

After a user is logged in, the client should support all the commands shown in the table below. For the following, assume that commands were run by user Yoda. You may assume that during marking the commands will be issued in the correct format as noted below with the appropriate arguments and a single white space between arguments. The <message> can however contain multiple words separated by white spaces. The message can contain uppercase characters (A-Z), lowercase characters (a-z), digits (0-9) and special characters (~!@#$%^&*_-+=`|\(){}[]:;"'<>,.?/). For commands where <user> is specified, the server should check if this <user> is present in the credentials file. If an entry is not found (referred to as an invalid user), an appropriate error message should be displayed.

Command	**Description**
message <user> <message>	Send <message> to <user> through the server. If the user is online then deliver the message immediately, else store the message for offline delivery. If <user> has blocked Yoda, then a message to that effect should be displayed for Yoda. If the <user> is invalid or is self (Yoda) then an appropriate error message should be displayed. The <message> used in our tests will be a few words at most.
broadcast <message>	Send <message> to all online users except Yoda and those users who have blocked Yoda. Inform Yoda that message could not be sent to some recipients.
whoelse	This should display the names of all users that are currently online excluding Yoda and any users who may have blocked

	Yoda. Users can be displayed in any order.
whoelsesince <time>	This should display the names of all users who were logged in at any time within the past <time> seconds excluding Yoda and any users who may have currently blocked Yoda. Note that this, may include users that may currently be offline. If <time> is greater than the time since when the server has been running, then all users who logged in since the sever initiation time should be listed. This suggests that you will need to keep a login history since the start of the server. Users can be displayed in any order.
block <user>	blocks the <user> from sending messages to Yoda, receive presence notifications about Yoda and be able to check if Yoda is currently online or Yoda's online history. A message should be displayed to Yoda confirming the blocking action. <user> must not be informed that Yoda has blocked them. If <user> is self (i.e., Yoda) or invalid, then an appropriate error message should be displayed. It is not necessary that <user> is currently online. One user may block multiple other users.
unblock <user>	unblocks the <user> who has been previously blocked by Yoda and reverse all the actions manifested by the previous block command. A message should be displayed to Yoda confirming the unblocking action. If <user> is self (i.e., Yoda) or is invalid or was not previously blocked, then an appropriate error message should be displayed. It is not necessary that <user> is currently online.
logout	log out user Yoda. While marking, we will ONLY assume that users will exit by explicitly using the logout command. We will not abruptly kill any client (CTRL-C).

Any command that is not listed above should result in an error message being displayed to the user. The interaction with the user should be via the terminal (i.e., command line). All messages must be displayed in the same terminal. There is no need to create separate terminals for messaging with different users.

We do not mandate the exact text that should be displayed by the client to the user for the various messages. However, you must make sure that the displayed text is easy to comprehend. Please make sure that you DO NOT print any debugging information on the client terminal.

The server program should not print anything at the terminal. If you wish to print to the terminal for debugging purposes, then include an optional debug flag (e.g. –d) as a command line argument for the server. When this optional argument is included, your server can print debugging information to the terminal. This same practice could also be employed for the client program.

Some examples illustrating client server interaction using the above commands are provided in Section 8.

3.4 Peer to Peer Messaging

Some users may prefer to have some privacy during messaging. They may want to message their friends directly without all their conversation being routed via the server. A peer-to-peer messaging client is a good solution for this scenario. In addition to the above functionalities, you should implement peer-to-peer messaging (also referred to as private messaging). Private messages should be exchanged over a TCP connection between the two clients. Assume that client A wants to setup a private chat with client B. Setting up a TCP connection between the two will require certain information about B to be communicated to A via the server. We will not specify it here because there are a few different ways to do that. We will leave the design to you.

To implement this functionality your client should support the following commands (in addition to those listed in Section 3.3). For the following, assume that commands were run by user Yoda. The notion of an invalid use is the same as described in Section 3.3. A p2p messaging session can only be started with an online user.

Command	**Description**
startprivate <user>	This command indicates that user Yoda wishes to commence p2p messaging with <user>. The server should first ask <user> if they are willing to engage in a private chat with Yoda. If <user> declines, then the server should inform Yoda accordingly. If <user> agrees, then Yoda's client should obtain certain information about <user>'s client from the server. If <user> has blocked Yoda, then the server need not query <user> and an appropriate error message should be displayed to Yoda. If <user> is offline, invalid, or self then appropriate error messages should be displayed to Yoda. If the private chat session can proceed, Yoda's client should establish a TCP connection with <user>'s client. A confirmation message should be displayed to Yoda. This TCP connection should remain active till the private chat is stopped or one the user logs off.
private <user> <message>	Send <message> to <user> directly without routing through the server. If the user is no longer online at the port obtained via the previous command, then a message to that effect should be displayed to Yoda. If Yoda has not executed startprivate before this command, then an appropriate error message should be displayed. Note that, Yoda may execute startprivate <user> before <user> blocks Yoda. In this instance, Yoda can still use this command to send a message to <user>. Other error messages (e.g. offline, invalid, etc.) are consistent with those indicated in the above command.
stopprivate <user>	This command indicates that user Yoda wishes to discontinue the p2p messaging session with <user>. Either user can issue this command (i.e., it doesn't have to be the one who initiated the private chat). A message to this effect should be displayed on the terminal for <user>. The TCP connection established between the two end points should be closed. An appropriate error message should be displayed to Yoda if there does not exist an active p2p messaging session with <user> (i.e, startprivate was not executed with this <user>).

Note that, a user that actively issues p2p messaging commands must be assumed to be active in the context of the timeout functionality mentioned in Section 3.1.

When a user logs off (or is timed out), any on-going p2p messaging session must conclude. A message to this effect must be displayed to the other user involved in the p2p messaging session.

While marking your assignment, we will initiate at most one p2p messaging session from each client at any given time. We may test the regular commands (not specific to p2p messaging) and the p2p messaging specific commands interchangeably. In other words, Yoda may be simultaneously exchanging messages privately with Leia and via the server with Luke (or even Leia). We may also test a scenario wherein Yoda establishes a p2p messaging session with Luke, closes this session and starts a new p2p messaging session with Leia. If Yoda has an ongoing p2p messaging session with Luke, we will not be testing the scenario where Leia tries to setup a p2p messaging session with either Yoda or Luke.

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 three 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.
- `block_duration`: this is the duration in seconds for which a user should be blocked after three unsuccessful authentication attempts.
- `timeout`: this is the duration in seconds of inactivity after which a user is logged off by the server.

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

If you use Java:

```
java Server server_port block_duration timeout
```

If you use C:

```
./server server_port block_duration timeout
```

If you use Python:

```
python server.py server_port block_duration timeout
```

The client should accept the following argument:

- `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.

Note that, you do not have to specify the port to be used by the client. The client program should allow the operating system to pick a random available port. Each client should be initiated in a separate terminal as follows:

If you use Java:

```
java Client server_port
```

If you use C:

```
./client server_port
```

If you use Python:

```
python client.py server_port
```

Note: The server and multiple clients should all be executed on the same machine on separate terminals. In the client program, use 127.0.0.1 (localhost) as the server IP address.

3.6 Program Design Considerations

Server Design

The server code will be more involved compared to the client. When the server starts up, none of the users are logged on. The server needs to support multiple clients simultaneously. A robust way to achieve this is 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 client, the server should first complete the authentication process. Once a user is authenticated, the server should wait for a request to action a particular command, take necessary action, respond accordingly to the client, and then wait for the next request. Note that, you will need to define several data structures for managing the state of the users. While we do not mandate the specifics, it is critical that you invest some time into thinking about the design of your data structures. Example of state information includes (this is not an exhaustive list) the total # of valid users (and their usernames), total # of online users (and their usernames), the time when each user logged on, the time when each user was last active, the blacklist for each user, offline messages for delivery when the specified user logs on. On start-up the server can determine the total # of valid users (by reading the entries in the credentials file). However, it is possible for new user accounts to be created. Thus, you cannot assume a fixed number of users upfront for defining data structures. As you may have learnt in your programming courses, it is not good practice to arbitrarily assume a fixed upper limit on the number of users. Thus, we strongly recommend allocating memory dynamically for all the data structures that are required. You will also need to use a timer to implement the timeout functionality. Once a client either times out or exits by executing the logout command, the corresponding thread should also be terminated after closing any active TCP connections. 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.

Client Design

The client program should be a little less complicated. 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 client should wait for the use to issue a command. Almost all commands require simple request/response interactions between the client with the server. Note that, the client does not need to maintain any significant state, since the server manages all state maintenance that is necessary. Implementing p2p messaging would require establishing a new TCP connection between the two peers. One way to accomplish this would to be use multithreading. Another approach could be to use non-blocking or asynchronous I/O by using polling, i.e., select().

4. Additional Notes

- This is NOT a 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 3 unsuccessful attempts. You could then proceed to the timeout functionality (i.e. automatically logout a user after inactivity. Then extend this to handle multiple clients. Once your server can support multiple clients, implement the functions for presence notification, whoelse and whoelsesince. Your next milestone should be to implement messaging (via the server) between users. Start with broadcast, then move to online messaging and finally offline messaging. Once you have ensured that all of the above are working perfectly, add the blacklist functionality. Note that, this will require changing the implementation of some of the functionality that you have already implemented. Once messaging via the server is working perfectly, you can move on to peer-to-peer messaging. 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 realising instant messaging services. You will have to design the format (both syntax and semantics) of the messages exchanged between the client and server and the actions taken by each entity on receiving these messages. We do not mandate any specific requirements with regards the design of your application layer protocol. We are only considered with the end result, i.e. the functionality 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 must use TCP for transferring messages between each client and server and between two clients for p2p messaging. The TCP connection with the server should be setup by the client during the login phase and should remain active until the user logs out or the server logs out the user due to inactivity (i.e., timeout). The server port is specified as a command line argument. The client port does not need to be specified. Your client program should let the operating system pick a random available port. There are several ways to facilitate the establishment of a TCP connection between two clients for p2p messaging. We will leave the specifics to you.
- **Debugging**: When implementing a complex assignment such as this, there are bound to be errors in your code. We strongly encourage that you follow a systematic approach to debugging. If you are using an IDE for development, then it is bound to have debugging functionalities. Alternately you could use a command line debugger such as pbd (python), jdb (java) or gdb (c). Use one of these tools to step through your code, create break points, observe the values of relevant variables and messages exchanged, etc. Proceed step by step, check and eliminate the possible causes until you find the underlying issue. Note that, we won't be able to debug your code on the forum or even in the help sessions.
- **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 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 in 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. If you are unsure, it is best you check with the course staff on the forum.

- 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 strongly encouraged to use the discussion 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 will arrange for additional consultations in Weeks 7-10 to assist you with assignment related questions. Information about the consults will be announced via the website.

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, data structure design, the application layer message format and a brief description of how your system works. Also discuss any design trade-offs considered and made. If your program does not work under any circumstances, 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 VLAB before submission. We appreciate that you may choose to develop the code natively on your machine and use an integrated development environment. However, your code will be tested in VLAB through command line interaction as noted in this document. 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 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 can 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:
- Up to 24 hours after deadline: 10% reduction
- More than 24 hours but less than 48 hours after deadline: 20% reduction
- More than 48 hours but less than 72 hours after deadline: 30% reduction
- More than 72 hours but less than 96 hours after deadline: 40% reduction
- More than 96 hours after deadline: NOT accepted

NOTE: The above penalty is applied to your final total. For example, if you submit your assignment 24 hours 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 terms. In addition, each submission will be checked against all other submissions of the current term. 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 award a ZERO mark for the assignment and reporting your name to the CSE plagiarism officer (which would mean adding your name to the school plagiarism register or escalating to a higher level if the name is already on it). 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 and creation of new account	1	1.5
Blocking user for specified interval after 3 unsuccessful attempts	1	1.5
Successful log in for multiple clients	0.5	1
Correct implementation of presence notification	1	1.5
Correct implementation of whoelse	1	1.5

Correct implementation of whoelsesince	1	1.5
Correct implementation of timeout functionality after inactivity	1	1.5
Correct implementation of broadcast	1	1.5
Correct implementation of messaging between two online clients	1.5	2.5
Correct implementation of offline messaging	2	2.5
Correct implementation of user blocking and unblocking and its effects	2	2.5
Properly documented report	1	1
Peer to peer Messaging • Correct implementation of startprivate accounting for all different considerations (declined chat, blocked user, etc.) • Correct implementation of p2p messaging • Correct implementation of stopprivate • Correct implementation of timeout accounting for p2p interaction • Correct implementation of simultaneous execution of p2p messaging and regular commands sent to the server.	(6) 2.5 2 0.5 0.5 0.5	N/A

NOTE: While marking, we will be testing for typical usage scenarios for the functionality described in Section 3 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, you should 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.

Successful Login

Terminal 1

```
java Server 4000 60 120
```

Terminal 2

```
java Client 4000
Username: yoda
Password: wise
Welcome to the greatest messaging application ever!
```

Unsuccessful Login (assume server is running on Terminal 1 as in Case 1)

Terminal 2

```
java Client 4000
Username: yoda
Password: weird
Invalid Password. Please try again
Password: green
Invalid Password. Please try again
Password: password
Invalid Password. Your account has been blocked. Please try again later
```

The user `yoda` should now be blocked for 60 seconds (since *block_time* is 60). The client should exit. Assume that a new terminal is opened before 60 seconds are over.

```
java Client 4000

Username: yoda

Password: wise

Your account is blocked due to multiple login failures. Please try again later
```

The client should exit. Assume now that a new terminal is opened after 60 seconds since when the blocking action was in effect.

```
java Client 4000

Username: yoda

Password: wise

Welcome to the greatest messaging application ever!
```

New User Account Creation (assume server is running on Terminal 1 as in Case 1)

```
java Client 4000

Username: jaba

This is a new user. Enter a password: longtongue

Welcome to the greatest messaging application ever!
```

Example Interactions

Consider a scenario where three users Hans, Yoda and Luke are currently logged in. No one has yet blocked anyone else. In the following we will illustrate the text displayed at the terminals for all three users as they execute various commands. Some other examples with different users are also provided.

1. Hans executes whoelse followed by a command that is not supported

hans's Terminal	**yoda's Terminal**	**luke's Terminal**

`whoelse`
`yoda`
`luke`		

`whatsthetime`
`Error. Invalid command`		

2. Hans messages Yoda and then messages an invalid user

hans's Terminal	**yoda's Terminal**	**luke's Terminal**
`message yoda Hey Dude`		
	`hans: Hey Dude`	

`message bob party time`
`Error. Invalid user`		

3. Hans broadcasts a message

hans's Terminal	**yoda's Terminal**	**luke's Terminal**
`broadcast vader is evil`		
	`hans: vader is evil`	`hans: vader is evil`

4. Luke blocks Hans followed by a few interactions that illustrate the effect of blocking and

unblocking.

hans's Terminal	**yoda's Terminal**	**luke's Terminal**

		`block hans`
		`hans is blocked`

`broadcast I travel solo`
`Your message could not be delivered to some recipients`		

	`hans: I travel solo`	
`whoelse` `yoda`		

`message luke You angry?`
`Your message could not be delivered as the recipient has blocked you`		

`block hans`
`Error. Cannot block self`		

		`unblock yoda`
		`Error. yoda was not blocked`

		`unblock hans`
		`hans is unblocked`

`broadcast jedis are cool`		
	`hans: jedis are cool`	`hans: jedis are cool`

5. Assume that Vader was logged in 5 minutes ago but logged out 2 minutes ago and that R2D2 was logged in 10 minutes ago but logged out 5 minutes ago.

hans's Terminal	**yoda's Terminal**	**luke's Terminal**

		`whoelsesince 200`
		`hans`
		`yoda`
		`vader`

	`whoelsesince 500`
	`hans`
	`luke`
	`vader`
	`r2d2`	

6. Now assume that Hans and Yoda are logged on, but that Luke is currently offline. Luke joins in later and receives a stored message from Hans. It also shows presence notification. Later, Yoda logs out and the corresponding notification is shown to others.

hans's Terminal	**yoda's Terminal**	**luke's Terminal**
`message luke Let's rock`		`(Assume that luke logs in after this message)`

luke logged in	luke logged in	hans: Let's rock
	logout	
yoda logged out		yoda logged out

7. Assume that Hans, Yoda and Luke are currently logged in. Hans first tries to send a private message to Yoda without first executing startprivate. This is followed by the correct sequence of commands for private messaging. Observe that a non-private message (i.e. through the server) can also be sent by a user engaged in a private conversation.

hans's Terminal	**yoda's Terminal**	**luke's Terminal**

`private luke hey dude`
 	
`Error. Private messaging to luke not enabled`		

`startprivate luke`
 	
`Start private messaging with luke`		

		`hans would like to private message, enter y or n: y`
`hans accepts private messaging`		
`private luke hey dude`		
		`hans(private): hey dude`
		`private hans hello man`
`luke(private): hello man`		
`message yoda force is strong`		
	`hans: force is strong`	
`message luke now via server`		
		`hans: now via server`
`logout`		
		`hans logged out, private chat concluding`
	`hans logged out`	`hans logged out`