COMP3331/9331 Computer Networks and Applications

Assignment for Term 3, 2019

Version 1.0

Due: 11:59am (noon) Friday, 22 November 2019 (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 7th October 2019.

2. Goal and learning objectives

Instant messaging applications such as WhatsApp, WeChat, Facebook Messenger, 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 with the server using TCP. 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 bypass 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 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 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 **15 marks**. Implementing peer-to-peer messaging (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 peer-to-peer messaging). 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.

There is an extension component (outlined in Section 3.5) that is worth **4 marks**. Note that the bonus marks may not be proportional to the amount of extra work that you will have to do. They are there to encourage you to go beyond the standard assignment. The bonus marks can be used to make up for lost marks in the lab exercises but NOT for any of the exams (mid-session and final).

3.1. Server

The server has the following responsibilities -

User Authentication - When a new client requests for a connection, 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 (i.e. online) and a welcome message is displayed. When all messaging is done, a user should be able to logout from the server.

On entering invalid credentials, the user is prompted to retry. After 3 consecutive failed attempts, the 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 (even from another IP address). While a user is online, if someone uses the same username/password to log in (even from another IP address), then this new login attempt is denied.

Timeout - The server should keep track of all online users. If the server does not receive any commands from a user 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. Note that, to be considered active, a user must actively issue a command. The receipt of a message does not count.

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.

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.

3.2. Client

The client has the following responsibilities -

Authentication - The client should provide a login prompt to enable the user to authenticate with the server.

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.

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. 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 A.

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 A, then a message to that effect should be displayed for A. If the <user> is not present in the credentials file (i.e. invalid user) or is self (A) 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 A and those users who have blocked A. Inform A that message could not be sent to some recipients.
whoelse	This should display the names of all users that are currently online excluding A. 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 A. 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 A. A message should be displayed to A confirming the blocking action. If <user> is self (i.e., A) or is invalid, then an appropriate error message should be displayed. <user> must not be informed that A has blocked them.
unblock <user>	unblocks the <user> who has been previously blocked by A. A message should be displayed to A confirming the unblocking action. If <user> is self (i.e., A) or is invalid or was not already blocked, then an appropriate error message should be displayed.
logout	log out user A.

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. console). 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.

We also prefer that you do not print anything at the terminal running the server. We suggest that you use an optional debug flag (e.g. –d) for the server. When this flag is turned on, your server can print debugging information to the terminal.

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).

To implement this functionality your client should support the following commands (in addition to those listed in Section 3.3)

Command	**Description**
startprivate <user>	This command indicates that user A wishes to commence p2p messaging with <user>. The client should obtain the IP address and port number being used by the <user> from the server. The client should try to establish a TCP connection to this IP address and port number combination. A confirmation message should be

	displayed to A. If <user> has blocked A, then server should not provide the IP address and port number and an appropriate error message should be displayed. If <user> is offline, invalid or self then appropriate error messages should be displayed.
private <user> <message>	Send <message> to <user> directly without routing through the server. If the user is no longer online at the address obtained via the previous command then a message to that effect should be displayed for A. If A has not executed startprivate before this command then an appropriate error message should be displayed. Note that, A may execute startprivate <user> before <user> has blocked A. In this instance, A 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 A wishes to discontinue the p2p messaging session with <user>. 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 if there does not exist an active p2p messaging session with <user>

3.5 Extension

Once you have your P2P messaging in place, the next logical step is to utilize the setup to transfer files in a distributed manner (similar to BitTorrent). Assume that a user 'A' has a file (assume a binary format) that they are willing to serve to other users in a P2P manner. User A divides the file into N chunks (we will use N=10 for this part of the assignment) and registers the details of the file (name, size and number of chunks) with the server. Any client can search for the availability of a specific file name in the system from the server. The server will reply back with either 'Not available' or 'Available' along with list of online users that have one or more chunks of the requested file name. A client can also search for specific chunks of a file.

The client will then select peers that have chunks and request the transfer of the selected chunks. Once a chunk has been downloaded, the client will also register the availability of this chunk with the server. The algorithm to select the peers for downloading the chunks is for you to decide. The objective is to finish the download in an efficient and distributed manner. As a rule of thumb, you cannot download all chunks of a file from a single peer, if other peers have some chunks of that file available for download.

You need to provide the following additional features in order to support the file transfer.

Command	**Description**
register <filename> <parameters>	This command indicates that user A wishes to register a <filename> with the server. <parameters> contains names, number & size of chunks. The server will respond with 'Ok' if the file is successfully registered. A confirmation message should be displayed to A. If the file is already registered, the server will respond appropriately and registers the availability of chunks at

	user A.
searchFile <filename>	User A wishes to search for the availability of a filename in the system. The server will reply back with either 'Not available' or 'Available' along with list of online users that have one or more chunks of the requested <filename>.
searchChunk <filename> <chunks>	User A searches for the availability of specific <chunks> for a <filename> in the system. The server will reply back with either 'Not available' or 'Available' along with list of online users that have one or more requested <chunks> of the <file name>.
private <user> <download> <filename> <chunk>	The client should obtain the IP address and port number being used by the <user> from the server. If <user> has blocked A, then the server should not provide the IP address and port number and an appropriate error message should be displayed. If <user> is offline, invalid or self then appropriate error messages should be displayed. The client should try to establish a TCP connection to this IP address and port number combination. Once the connection is established, it requests the download of a <chunk> of a specific <filename>. An appropriate error message should be displayed if there does not exist an active p2p messaging session with <user>.

When running all clients and server on the same machine, make sure that a client only has access to chunks that it has currently downloaded from the system. Only the original owner has access to all chunks from the start.

3.6 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 two 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.

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

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
```

Note: 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 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 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 should use TCP for transferring messages between each client and server. The TCP connection 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 OS pick a random available port.

- **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 on these machines (i.e. your 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 9. 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 consultations in Weeks 7, 8 and 9 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, 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. **If you have attempted the extension then, please clearly indicate so in your report**.

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 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:
- 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 specified interval after 3 unsuccessful attempts (even from different IP)	1	1.5
Successful log in for multiple clients (from multiple machines)	1	1.5
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 automatic logout functionality after inactivity (timeout)	1	1.5
Correct Implementation of broadcast	1	1.5
Correct Implementation of messaging between two online clients	1.5	2
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
Code quality and comments	1	1
Peer to peer Messaging	5	N/A
Extension (p2p file exchange)	4	N/A

The maximum bonus mark that you can get for the extended assignment is **4 marks**. The bonus marks can be used to make up for lost marks in the lab exercises but NOT for any of the exams (mid-session and final).

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

Terminal 1

```
>java Server 4000 60 120
```

Terminal 2

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

```
>Password: wise
>Welcome to the greatest messaging application ever!
>
```

Case 2: Unsuccessful Login (assume server is running on Terminal 1 as in Case 1)

Terminal 2

```
>java Client 10.11.0.3 4000   (assume that server is executing on 10.11.0.3)
>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 should now be blocked for 60 seconds (since *block_time* is 60). The terminal should shut down at this point.

Terminal 2 (reopened before 60 seconds are over)

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

Terminal 2 (reopened after 60 seconds are over)

```
>java Client 10.11.0.3 4000   (assume that server is executing on 10.11.0.3)
>Username: yoda
>Password: wise
>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 sucks		
	>hans: vader sucks	>hans: vader sucks

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 rock >Your message could not be delivered to some recipients		
	>hans: I rock	
>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 stormtroopers		
	>hans: stormtroppers	>hans: stormtroppers

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		
>private luke hey dude		
		>hans(private): hey dude
		>private hans hello
>luke(private): hello		
>message yoda force is strong		
	>hans: force is strong	
>logout		
	>hans logged out	>hans logged out