COMP3331/9331 Computer Networks and Applications

Assignment for Term 1, 2022

Version 1.0

Due: 11:59am (noon) Friday, 22 April 2022 (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 March 2022.

2. Goal and learning objectives

Online discussion forums are widely used as a means for large groups of people to hold conversations on topics of mutual interest. A good example is the online forum used for this course. In this assignment, you will have the opportunity to implement your own version of an online discussion forum application. Your application is based on a client server model consisting of one server and multiple clients communicating either sequentially (i.e., one at a time) or concurrently. The client and server should communicate using both UDP and TCP. Your application will support a range of functions that are typically found on discussion forums including authentication, creation and deletion of threads and messages, reading threads, uploading and downloading files. However, unlike typical online forums that are accessed through HTTP, you will be designing a custom application protocol. Most functions should be implemented over UDP except for uploading and downloading of files, which should use TCP.

2.1 Learning Objectives

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

1. Detailed understanding of how online discussion forums work.
2. Expertise in socket programming.
3. Insights into designing an application layer protocol and a fully functioning networked application.

The assignment is worth **20 marks**. We will test it in two distinct configurations. In the first instance, we will test the interaction between the server and a SINGLE active client. All outlined functionality will be tested. Multiple clients will connect to the server but sequentially – one client connects, interacts, exits, the second client connects, interacts, exits and so on. **The first configuration is worth 14 marks (70% of the total mark)**. In the second instance, we will test the interaction of the server with multiple **concurrent** clients. All outlined functionality will be tested. **The second configuration is worth 6 marks**. Submissions from CSE students will be tested in both configurations. Submissions from non-CSE students will only be tested in the first configuration. The marking guidelines are thus different for the two groups and are indicated in Section 7.

Non-CSE Student: The rationale for this option is that students enrolled in a program that does not include a computer science component have had very limited exposure to programming and

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

3. Assignment Specification

In this programming assignment, you will implement the client and server programs of a discussion forum application, similar in many ways to the discussion forum we use for this course. The difference being that your application is not web-based (i.e. non-HTTP) but uses a custom application layer protocol which you will design. The client and server must communicate using both UDP and TCP as described in the rest of the specification. Your application will support a range of operations including creating a new user account, creating and deleting a new thread, posting a message on a thread, editing or deleting messages, uploading and downloading files to/from a thread, reading a thread, and listing all threads. You will implement the application protocol to implement these functions. Most of the communication between the client and server will take place over UDP, except for file uploads and downloads which must use TCP. The server will listen on a port specified as the command line argument and will wait for a message from the client. When a user executes a client, the authentication process will be initiated. 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. The user may execute a series of commands (one after the other) and eventually quit. Both the client and server MUST print meaningful messages at the command prompt that capture the specific interactions taking place. You are free to choose the precise text that is displayed. Examples of client server interactions are given in Section 8. All communication between the client and server must happen over UDP. The only exception is uploading and downloading files to/from a thread, where the file transfer process must use TCP. The implementation of these operations will require your client to initiate a TCP connection with the server. The server would first need to open a TCP port using the port number specified as the command line argument and listen for connection requests. Note that, UDP and TCP port numbers are distinct (e.g., UDP port 53 is distinct from TCP port 53). Thus, your server can concurrently open a UDP and TCP port with the specified port number (as the command line argument). Once the TCP connection is established, the file upload or download should be initiated. The TCP connection **should be immediately closed** after the file transfer is completed.

The assignment will be tested in two configurations. In the **first configuration**, the server will interact with a single client at any given time. The interaction will involve authentication followed by the execution of several commands, one after the other. Multiple clients can connect with the server in a serial fashion, i.e., one client is initiated, the client authenticates and executes several commands one after the other and quits, a second client is initiated, authenticates, executes several commands, and quits, and so on. The server design is significantly simplified (i.e., you won't need to deal with concurrency) if you only wish to implement this portion of the assignment. A correct implementation of this first part is worth **70% of the assignment marks** (14 marks, see Section 7). In the **second configuration**, the server must interact with multiple clients concurrently. The client design should not require any changes to meet this requirement.

The server design, however, would require a significant change, in that, the server would need to send and receive messages to and from multiple clients concurrently. We strongly recommend using **multi-threading** to achieve this. The interaction with a single client, would however be similar as in the first configuration. Note that, a correctly implemented multi-threaded server should also be able to interact correctly with a single client at any given time. So, if you design your client and server to achieve all functionality expected for the second configuration, it should work as expected in the first configuration.

The server program will be run first followed by one or more instances of the client program (each instance supports one client in the second configuration). 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 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. Submission instructions are in Section 5.

The server should accept one argument:

- `server_port`: this is the port number which the server will use to communicate with the clients. Since TCP and UDP ports are distinct, the server can open a UDP and TCP port with the same port number. Recall that UDP is connectionless, so the server should be able to communicate with multiple clients (in the second configuration) through a single server-side UDP socket. Also, a TCP socket is NOT uniquely identified by the server port number. It should thus be possible for multiple TCP connections to use the same server-side port number (when multiple clients are simultaneously uploading/downloading files to/from the server in the second configuration).

The server should be executed before the clients. You may assume that the server will remain executing forever. While marking, 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 logged on and that the discussion forum is empty, i.e., there are no threads or messages. The spec indicates that we will test your code in two configurations. We The server will remain executing for the entire duration that encompasses all tests. In other words, the server will not be restarted after concluding the tests for the first configuration.

It should be initiated as follows:

If you use Java:

```
java Server server_port
```

If you use C:

```
./server server_port
```

If you use Python:

```
python server.py server_port OR
python3 server.py server_port
```

The client should accept one 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. You should allow the OS to pick a random available port (for both UDP and TCP). 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 OR
python3 client.py server_port
```

3.2 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 will contain username and passwords of authorised users. They contain uppercase characters (A-Z), lowercase characters (a-z) and digits (0-9) and special characters (~!@#$%^&*_-+=`|\(){}[]:;"'<>,.?/). **Username and passwords are case-sensitive**, so Yoda and yoda are distinct usernames. An example *credentials.txt* file is provided on the assignment page. 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. Neither the username nor password will contain white spaces. There will only be one password per username. The credentials.txt file will terminate with a '\n' (newline character). There will be no other empty lines in this file.

Upon execution, a 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 prompts the user to enter a username and the process explained above is repeated. You may assume that there is no limit to the number of login attempts that a user may try if they keep entering a wrong password. If the username does not exist, it is assumed that the user is creating a new account and the server 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 in the file). A confirmation is sent to the client. The client displays an appropriate 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 assumed to be logged in. All messages exchanged for implementing authentication should use UDP.

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., the same username cannot be used concurrently by two clients). The server should keep track of all users that are currently logged on 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 server and displayed at the prompt for the user and they should be prompted to enter a username.

3.3 Discussion Forum Operations

Following successful login, the client displays a message to the user informing them of all available commands and prompting to select one command. The following commands are available: CRT: Create Thread, LST: List Threads, MSG: Post Message, DLT: Delete Message, RDT: Read Thread, EDT: Edit Message, UPD: Upload File, DWN: Download File, RMV: Remove Thread, XIT: Exit. All available commands should be shown to the user in the first instance after successful login. Subsequent prompts for action should include this same message.

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

In the following, the implementation of each command is explained in detail. The expected usage of each command (i.e. syntax) is included. **Note that, all commands should be upper-case (CRT, MSG, etc.)**. All arguments (if any) are separated by a single white space and will be one word long (except messages which can contain white spaces). **You may assume that all arguments including thread names, file names and the message text may contain uppercase characters (A-Z), lowercase characters (a-z), digits (0-9) and the '.' special character. The message text can additionally contain white spaces**. You are not required to check if the names or message text adhere to this format.

If the user does not follow the expected usage of any of the operations listed below, i.e., missing (e.g., not specifying the title of the thread when creating a thread) or additional arguments, 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.

The error checking described above can be readily implemented in the client program.

The application can support 10 different commands. 8 of these (excluding file upload and download) should use UDP for communication. Note that UDP segments can be occasionally lost, so you should implement some simple mechanisms such as a retransmission timer to deal with the possibility of packet loss. We will leave the specifics for you to decide. We have discussed several mechanisms for implementing reliable data transfer in the lectures. The file upload and download commands (UPD and DWN) should use TCP for transferring the file. The server should first open a TCP socket on the port number specified in the command line argument (UDP and TCP ports are distinct, so the server can simultaneously use UDP port X and TCP port X). The client should initiate the establishment of the TCP connection. Once the TCP connection is established, the file transfer should be initiated from the client (UPD) or the server (DWN). The TCP connection should be closed immediately after the file transfer concludes. Since the file transfer takes place over TCP, you do not have to worry about reliable transfer of the file.

The execution of each individual command is described in detail below.

CRT: Create Thread

```
CRT threadtitle
```

The title of the new thread (*threadtitle*) should be included as an argument with this command. Thread titles are **one word long and case sensitive.** The client should send the command (CRT), the title of the thread and the username to the server. Each thread is represented as a text file in the current working directory of the server with the same file name as the thread title (*threadtitle*, DO NOT add ".txt" extension to the name). The first line of the file should contain the username who created the thread. Each subsequent line should be a message, added in the chronological sequence in which they were posted. The server should first check if a thread with this title exists. If so, an error message should be conveyed to the client and displayed at the prompt to the user. If the thread does not exist, a new file with the provided title should be created as per the convention noted above (the first line of this file should be the username of the creator). You may assume that the server program will have permission to create a file in the current working directory. A confirmation message should be sent to the server and displayed at the prompt to the user. The client should next prompt the user to select one of the available commands.

MSG: Post Message

```
MSG threadtitle message
```

The title of the thread that the message should be posted to and the message should be included as arguments. Note that, the message may contain white spaces (e.g. "hello how are you"). The client should send the command (MSG), the title of the thread, the message and the username to the server. **In our tests, we will only use short messages (a few words long)**. The server should first check if a thread with this title exists. If so, the message and the username should be appended at the end of the file in the format, along with the number of the message (messages within each thread are numbered starting at 1):

```
messagenumber username: message
```

An example:

```
1 yoda: do or do not, there is no try
```

A confirmation message should be sent to the server and displayed to the user. If the thread with this title does not exist, an error message should be sent to the client and displayed at the prompt to the user. The client should next prompt the user to select one of the available commands.

DLT: Delete Message

```
DLT threadtitle messagenumber
```

The title of the thread from which the message is to be deleted and the message number within that thread to be deleted should be included as arguments. **A message can only be deleted by the user who originally posted that message**. The client sends the command (DLT), the title of the thread, the message number and the username to the server. The server should check if a thread with this title exists and if the corresponding message number is valid and finally if this user had originally posted this message. In the event that any of these checks are unsuccessful, an appropriate error message should be sent to the client and displayed at the prompt to the user.

If all checks pass, then the server should delete the message, which entails deleting the line containing this message in the corresponding thread file (all subsequent messages and information about uploaded files in the thread file should be moved up by one line and the message numbers should be updated appropriately) and a confirmation should be sent to the client and displayed at the prompt to the user. The client should next prompt the user to select one of the available commands.

EDT: Edit Message

```
EDT threadtitle messagenumber message
```

The title of the thread from which the message is to be edited, the message number within that thread to be edited and the new message should be included as arguments. **A message can only be edited by the user who originally posted that message**. The client should send the command (EDT), the title of the thread, the message number, the new message and the username to the server. The server should check if a thread with this title exists and if the corresponding message number is valid and finally if the username had posted this message. In the event that any of these checks are unsuccessful, an appropriate error message should be sent to the client and displayed at the prompt to the user. If all checks pass, then the server should replace the original message in the corresponding thread file with the new message (the rest of the details associated with this message, i.e. message number and username should remain unchanged) and a confirmation should be sent to the client and displayed at the prompt to the user. The client should next prompt the user to select one of the commands.

LST: List Threads

```
LST
```

There should be no arguments for this command. The client sends the command (LST) to the server. The server replies back with a listing of all the thread titles. Only the thread titles should be listed, not the messages. The client should print the list on the terminal (one thread per line). If there are no active threads, then a message to that effect should be displayed at the prompt to the user. The client should next prompt the user to select one of the available commands.

RDT: Read Thread

```
RDT threadtitle
```

The title of the thread to be read should be included as an argument. The client should send the command (RDT) and the title of the thread to be read to the server. The server should check if a thread with this title exists. If so, the server should send the contents of the file corresponding to this thread (excluding the first line which contains the username of the creator of the thread) to the client. The client should display all contents of the file including messages and information about uploaded files (see next action) at the terminal to the user. If the thread with this title does not exist, an error message should be sent to the client and displayed at the prompt to the user. The client should next prompt the user to select one of the available commands.

UPD: Upload file

```
UPD threadtitle filename
```

The title of the thread to which the file is being uploaded to and the name of the file should be included as arguments. Thread titles and file names are case sensitive. 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 (e.g., text file), i.e., assume that it is a **binary file**. Be careful to not use functions for file access (reading, writing, etc.) that assume the file to be in text format. The client should send the command (UPD), the title of the thread, the username, and the name of the file being uploaded to the server. The server should check if a thread with this title exists. If it does not, then an appropriate error message should be sent to the client and displayed at the prompt to the user. The server should also check if a file with the provided file name already exists. If it does, an appropriate error message should be conveyed to the client and displayed at the prompt to the user (Note that the same file can be uploaded to different threads). If the thread exists and the file has not already been uploaded to the thread, then a confirmation message should be sent to the client. Following this, the client should transfer the contents of the file to the server. All communication between the client and server described so far should happen over UDP. TCP should only be used for transferring the contents of the file. The TCP connection should be immediately closed after completion of the file transfer. The file should be stored in the current working directory of the server with the file name threadtitle-filename (DO NOT add an extension to the name. If the filename has an extension in the name, retain it, e.g., *test.exe* should be stored as *threadtitle-test.exe*). File names are **case sensitive** and **one word long**. You may assume that the server program will have permission to create a file in its current working directory. A record of the file should be noted on the thread, i.e., an entry should be added at the end of the file corresponding to the thread title indicating that this user has uploaded a file with the specified name. The format should be as follows (note the lack of a message number which differentiates it from a message):

```
Username uploaded filename
```

The entries for file uploads cannot be edited using the EDT command or deleted using the DLT command. They should however be included when a thread is read using the RDT command. Finally, the server should send a confirmation message to the client and a message to this effect should be displayed at the prompt to the user. The client should next prompt the user to select one of the available commands.

DWN: Download file

```
DWN threadtitle filename
```

The title of the thread from which the file is being downloaded and the name of the file should be included as arguments. The client should send the title of the thread and the name of the file to the server. The server should check if a thread with this title exists and if so whether a file with this name was previously uploaded to the thread. If either check does not match, then an appropriate error message should be sent to the client and displayed at the prompt to the user. If a match is found, then the server should transfer the contents of the file to the client. As with the UPD command, all communication between the client and server described so far should happen over UDP. TCP should only be used for transferring the contents of the file. The TCP connection should be immediately closed after completion of the file transfer. The client should write the contents to a local file in the current working directory of the client with the same name (*filename,* DO NOT include *threadtitle* in the file name). You may assume that the client program will have permission to create a file in the current working directory. You may also assume that a file with this same name does not exist in the current working directory of the client. Once the

file transfer is complete, a confirmation message should be displayed at the prompt to the user. The client should next prompt the user to select one of the available commands. Note that, the file should NOT be deleted at the server end. The client is simply downloading a copy of the file.

TESTING NOTES: (1) When you test the operation of the UDP and DWN command, you will likely first upload a test file from the client to the server using the UPD and then try to download the same file from the server using the DWN command. You should make sure that you remove this file from the current working directory of the client between these two commands (to be consistent with the assumption stated in the description above). You can do this by opening a separate terminal and deleting this file from the client's working directory. (2) For similar reasons, when testing your program under the second configuration, make sure that the multiple clients are executed in different working directories.

RMV: Remove Thread

```
RMV threadtitle
```

The title of the thread to be removed should be included as an argument with this action. **A thread can only be removed by the user who originally created that thread.** The client should send the operation (RMV), the title of the thread and the username to the server. The server should first check if a thread with this title exists and if so, whether the user who created the thread matches with the provided username. If either check doesn't match, then an error message should be sent to the client and displayed at the terminal to the user. Else, the thread is deleted including the file storing information about the thread, any files uploaded to the thread and any state maintained about the thread at the server. A confirmation message should be sent to the client which is displayed at the prompt to the user. The client should next prompt the user to select one of the available actions.

XIT: Exit

```
XIT
```

There should be no arguments for this command. The client should inform the server that the user is logging off and exit with a goodbye message displayed at the terminal to the user. The server should update its state information about currently logged on users. Note that, any messages and files uploaded by the user must not be deleted.

3.3 Program Design Considerations

Transport Layer

You MUST use **UDP** for communicating between the client and server to implement the authentication process and 8 of the 10 commands, i.e., excluding UPD and DWN. TCP should only be used for transferring the contents of the file. All other message exchanges required to implement UDP and DWN should use UDP. Remember that UDP is connectionless and that your client and server programs must explicitly create UDP segments containing your application messages and send these segments to the other endpoint. The client has the socket information about the server (127.0.0.1 and `server_port`). The server program should extract the client-side socket information from the UDP segment sent by the client. The responses sent by the server to the client should be addressed to this socket. Note that a maximum size of UDP segment is 65,535 bytes. The loopback interface (127.0.0.1) on the machine you are executing

the program on may have a smaller MSS value. Either way, it is unlikely that you would need to send very large UDP segments, so this should not be an issue.

Since, UDP segments can be occasionally lost, you must implement some simple mechanisms to recover from it. We have discussed several mechanisms for implementing reliable data transfer in the lectures. You are free to use one or more of those in your implementation. We do not specifically mandate the mechanisms to be used.

The file transfer process associated with the UPD and DWN commands should use TCP.

The server port is specified as a command line argument (`server_port`). Note that TCP and UDP ports are distinct, so the server can open a TCP port at `server_port` and a UDP port at `server_port`. The client port does not need to be specified. Your client program should let the OS pick a random available port.

If you no not adhere to the choice of the transport layer as noted in the specification, then a significant penalty will be assessed.

Client Design

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

We don't anticipate that any changes would be required to your client design as you progress the implementation from the first configuration to the second configuration.

Server Design

The server code will be fairly involved compared to the client as the server is responsible for maintaining the message forum. However, the server design to implement functionality for the first configuration of testing should be relatively straightforward as the server needs to only interact with one client at a time. **When the server starts up, the forum is empty – i.e., there exist no threads, no messages, no uploaded files**. The server should open a UDP socket and wait for an authentication request from a client. Once authentication is complete, the server should service each command issued by the client sequentially. This will require the client and server to exchange application layer messages (which you will design) with each other, encapsulated within UDP segments.

Recall, that the file transfer for the UPD and DWN commands should take place over TCP. The server should first open a TCP socket on `server_port` and wait for the client to establish a TCP connection. Once the TCP connection is established, the file transfer should be initiated. The TCP connection should be closed immediately after the file transfer is complete. Note that, the user can only initiate the next command once the file transfer process has completed, and a confirmation message is displayed to the user followed by a prompt to enter the next command.

While we do not mandate the specifics, it is critical that you invest some time into thinking about the design of your data structures. Examples of state information includes (this is not an exhaustive list): the total # of online users (and their usernames), the total of threads and their names, the posts and files associated with each thread. On start-up, the server can determine the

total number of 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.

Implementing functionality for the second configuration will require a significant change as the server must interact with multiple clients simultaneously. A robust way to achieve this to use **multithreading**. We do not mandate a specific approach. There are various approaches that could be employed to achieve this functionality. Do note that, it is feasible for multiple threads to send and receive data from the same UDP socket. Thus, you should not need to create multiple UDP sockets at the server.

When interacting with one client, the server should receive a request for a particular command, take all necessary action and respond accordingly to the client and then process the next request. This process is exactly like what you would have implemented to meet the functionality of the first configuration.

You may assume that each interaction with a client is **atomic**. Consider that client A initiates an interaction (i.e., any of the 10 commands or the authentication process) with the server. While the server is processing this interaction, it cannot be interrupted by a command from another client B. Client B's command will be acted upon after the server has finished processing the command from client A. In the context of UDP and DWN, this means that at any given time, the server can either be receiving or sending a file from/to at most one client. Thus, the server should only be managing at most one TCP connection at any given time.

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. A server program that correctly implements functionality for the second configuration should be able to correctly accomplish all interactions expected in the first configuration.

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. We recommend that you first implement the functionality for the first configuration, i.e., the server interacts with a single active client at any time. A good place to start would be to implement the functionality to allow a single user to login with the server. Next, add functionality to implement one command. Ensure you thoroughly test the operation of each command, including typical error conditions, and then progress to the next. We recommend that you start with the simpler commands such as CRT, MSG, LST before progressing to more complex commands such as UPD and DWN. Once you have thoroughly tested your code for the first configuration, proceed to the second configuration. 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 a fully functional discussion forum. 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 to the design of your application layer protocol. We are only concerned 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.

- **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 8. However, please make sure that the text is clear and unambiguous.
- You are strongly encouraged to use the course 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 names (see Section 3.1). 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 trade-offs considered and made. If your program does not work under any circumstances, please report this here. If you have implemented functionality for handling multiple concurrent clients, then you should indicate your approach in the report. Also indicate any code segments that were borrowed from the Web or other sources.

You are required to submit your source code and `report.pdf`. You can submit your assignment using the give command through VLAB. 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). The same command should be used for 3331 and 9331.

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 may submit multiple 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 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 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 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. DO NOT POST YOUR CODE TO GITHUB OR ANY OTHER REPOSITORY AND ALLOW PUBLIC ACCESS. THIS WILL BE CONSIDERED AS PLAGARISM.

7. Marking Policy

The following table outlines the marking rubric for both CSE and non-CSE students. For CSE students, **14 marks** are attributed towards testing the interaction between the server and one active client (multiple clients will connect sequentially one after the other as in the sample interaction provided). **6 marks** are attributed towards testing the interaction between the server and multiple concurrent clients. You should test your program rigorously before submission. **All submissions will be manually marked by your tutors and NOT auto marked**. Some helper scripts may be used to assist with the marking. Your submissions will be marked using the following criteria:

Functionality	**Marks (CSE)**	**Marks (Non-CSE)**
Successful authentication for an existing and new user including all error handling	1	1.5
Successful creation of a new thread (CRT command) including all error handling	1	1.5
Successful creation of a new message (MSG command) including all error handling	1	1.5
Successful listing of active threads (LST command) including all error handling	0.5	0.75
Successful reading of an active thread (RDT command) including all error handling	1	1.5
Successful editing of an existing message (EDT command) including all error handling	1	1.5
Successful deletion of an existing message (DLT command) including all error handling	1	1.5
Successful deletion of an active thread (RMV command) including all error handling	1	1.5
Successful uploading of a file to a thread (UPD command) including all error handling	2	3
Successful download of a file from a thread (DWN command) including all error handling	2	3
Successful log off for a logged in user (XIT command) including all error handling	0.5	0.75
Implementation of mechanisms to recover from occasional loss of UDP segments	1	1
Properly documented report	1	1
Successful authentication of multiple concurrent existing and new users including all error handing	0.5	N/A

Successful execution of all 8 commands excluding UPD and DWN and associated error handling (8 x 0.5 marks each)	4	N/A
Successful execution of UPD and DWN and associated error handling (2 x 0.75 marks each)	1.5	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 15-20 minutes. When testing with multiple concurrent clients, we will spawn a maximum of 3 concurrent 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

In the following we provide examples of sample interactions for both configurations to be tested. Your server and client code should display similar meaningful messages at the terminal. You **do not** have to use the same text as shown below. Note that, this is not an exhaustive summary of all possible interactions. Our tests will not necessarily follow this exact interaction shown.

First Configuration

In this configuration, the server interacts with a single client at any given time. It is recommended to execute the client and server in different working directories. Ensure that write permissions are enabled on the credentials file. In the following, two clients with usernames Yoda and Obi-wan connect and interact with the server sequentially in that order. The inputs from the user are shown as underlined in the client terminal. Extra spacing is inserted in the server terminal to align the output with corresponding user interaction at the client end.

Client Terminal	Server Terminal
	>java Server 5000
>java Client 5000	Waiting for clients
Enter username: Yoda	Client authenticating
Enter password: sdrfdfs12	Incorrect password
Invalid password	
Enter username: Yoda	
Enter password: jedi*knight	Yoda successful login
Welcome to the forum	
Enter one of the following commands: CRT, MSG, DLT, EDT, LST, RDT, UPD, DWN, RMV, XIT: LST	
No threads to list	Yoda issued LST command
Enter one of the following commands: CRT, MSG, DLT, EDT, LST, RDT, UPD, DWN, RMV, XIT: HELLO	
Invalid command	
Enter one of the following commands: CRT, MSG, DLT, EDT, LST, RDT, UPD, DWN, RMV, XIT: CRT 3331	Yoda issued CRT command

Thread 3331 created	Thread 3331 created
Enter one of the following commands: CRT, MSG, DLT, EDT, LST, RDT, UPD, DWN, RMV, XIT: CRT 3331	
	Yoda issued CRT command
Thread 3331 exists	Thread 3331 exists
Enter one of the following commands: CRT, MSG, DLT, EDT, LST, RDT, UPD, DWN, RMV, XIT: CRT 9331	
	Yoda issued CRT command
Thread 9331 created	Thread 9331 created
Enter one of the following commands: CRT, MSG, DLT, EDT, LST, RDT, UPD, DWN, RMV, XIT: LST 3331	
Incorrect syntax for LST	
Enter one of the following commands: CRT, MSG, DLT, EDT, LST, RDT, UPD, DWN, RMV, XIT: LST	
The list of active threads:	Yoda issued LST command
3331	
9331	
Enter one of the following commands: CRT, MSG, DLT, EDT, LST, RDT, UPD, DWN, RMV, XIT: MSG 3331 Networks is awesome	
	Yoda issued MSG command
Message posted to 3331 thread	Message posted to 3331 thread
Enter one of the following commands: CRT, MSG, DLT, EDT, LST, RDT, UPD, DWN, RMV, XIT: RDT	
Incorrect syntax for RDT	
Enter one of the following commands: CRT, MSG, DLT, EDT, LST, RDT, UPD, DWN, RMV, XIT: RDT 9331	
	Yoda issued RDT command
Thread 9331 is empty	Thread 9331 read
Enter one of the following commands: CRT, MSG, DLT, EDT, LST, RDT, UPD, DWN, RMV, XIT: RDT 3331	
	Yoda issued RDT command
1 Yoda: Networks is awesome	Thread 3331 read
Enter one of the following commands: CRT, MSG, DLT, EDT, LST, RDT, UPD, DWN, RMV, XIT: UPD 3331 test.exe	
	Yoda issued UPD command
test.exe uploaded to 3331 thread	Yoda uploaded file test.exe to 3331 thread
Enter one of the following commands: CRT, MSG, DLT, EDT, LST, RDT, UPD, DWN, RMV, XIT: RDT 3331	
	Yoda issued RDT command
1 Yoda: Networks is awesome	Thread 3331 read
Yoda uploaded test.exe	
Enter one of the following commands: CRT, MSG, DLT, EDT, LST, RDT, UPD, DWN, RMV, XIT: RMV 9331	
	Yoda issued RMV command
Thread 9331 removed	Thread 9331 removed

Enter one of the following commands: CRT, MSG, DLT, EDT, LST, RDT, UPD, DWN, RMV, XIT: XIT	
Goodbye	Yoda exited
	Waiting for clients
>java Client 5000	
Enter username: Obi-wan	Client authenticating
New user, enter password: r2d2	New user
Enter one of the following commands: CRT, MSG, DLT, EDT, LST, RDT, UPD, DWN, RMV, XIT: CRT 9331	Obi-wan successfully logged in
Thread 9331 created	Obi-wan issued CRT command
Enter one of the following commands: CRT, MSG, DLT, EDT, LST, RDT, UPD, DWN, RMV, XIT: MSG 9331 Networks exam PWNED me	Thread 9331 created
Message posted to 9331 thread	Obi-wan issued MSG command
Enter one of the following commands: CRT, MSG, DLT, EDT, LST, RDT, UPD, DWN, RMV, XIT: MSG 3331 Networks exam PWNED me	Obi-wan posted to 9331 thread
Message posted to 3331 thread	Obi-wan issued MSG command
Enter one of the following commands: CRT, MSG, DLT, EDT, LST, RDT, UPD, DWN, RMV, XIT: LST	Obi-wan posted to 3331 thread
The list of active threads:	
3331	Obi-wan issued LST command
9331	
Enter one of the following commands: CRT, MSG, DLT, EDT, LST, RDT, UPD, DWN, RMV, XIT: RDT 331	Obi-wan issued RDT command
Thread 331 does not exist	Incorrect thread specified
Enter one of the following commands: CRT, MSG, DLT, EDT, LST, RDT, UPD, DWN, RMV, XIT: RDT 3331	Obi-wan issued RDT command
1 Yoda: Networks is awesome	Thread 3331 read
Yoda uploaded test.exe	
2 Obi-wan: Networks exam PWNED me	
Enter one of the following commands: CRT, MSG, DLT, EDT, LST, RDT, UPD, DWN, RMV, XIT: DWN 9331 test.exe	Obi-wan issued DWN command
File does not exist in Thread 9331	test.exe does not exist in Thread 9331
Enter one of the following commands: CRT, MSG, DLT, EDT, LST, RDT, UPD, DWN, RMV, XIT: DWN 3331 test.exe	Obi-wan issued DWN command
test.exe successfully downloaded	test.exe downloaded from Thread 3331
Enter one of the following commands: CRT, MSG, DLT, EDT, LST, RDT, UPD, DWN, RMV,	

XIT: EDT 3331 1 I PWNED Networks exam	Obi-wan issued EDT commend
The message belongs to another user and cannot be edited	Message cannot be edited
Enter one of the following commands: CRT, MSG, DLT, EDT, LST, RDT, UPD, DWN, RMV, XIT: EDT 3331 2 I PWNED Networks exam	Obi-wan issued EDT commend
The message has been edited	Message has been edited
Enter one of the following commands: CRT, MSG, DLT, EDT, LST, RDT, UPD, DWN, RMV, XIT: RDT 3331	Obi-wan issued RDT command
1 Yoda: Networks is awesome	Thread 3331 read
Yoda uploaded test.exe	
2 Obi-wan: I PWNED Networks exam	
Enter one of the following commands: CRT, MSG, DLT, EDT, LST, RDT, UPD, DWN, RMV, XIT: RMV 3331	Obi-wan issued RMV command
The thread was created by another user and cannot be removed	Thread 3331 cannot be removed
Enter one of the following commands: CRT, MSG, DLT, EDT, LST, RDT, UPD, DWN, RMV, XIT: RMV 9331	Obi-wan issued RMV command
The thread has been removed	Thread 9331 removed
Enter one of the following commands: CRT, MSG, DLT, EDT, LST, RDT, UPD, DWN, RMV, XIT: LST	Obi-wan issued LST command
The list of active threads:	
3331	
Enter one of the following commands: CRT, MSG, DLT, EDT, LST, RDT, UPD, DWN, RMV, XIT: XIT	Obi-wan exited
Goodbye	Waiting for clients
>	

Second Configuration

In this configuration, the server interacts concurrently with multiple clients. In the following, two clients with usernames Yoda and R2D2 connect and interact with the server concurrently. The inputs from the user are shown as underlined. You MUST execute each individual client in a separate working directory. Ensure that write permissions are enabled on the credentials file. The interaction below shows the server being initiated. However, note that when we test your code, the server would already be executing as we would have conducted tests for the first configuration. The server will not be restarted between the tests for the two configurations. Note that, extra space is added in the two client terminals to simulate some delay before the users enter commands when prompted to do so. This is simply done to improve readability of the output below. You should not make such assumptions in your implementation.

Client 1 Terminal	Client 2 Terminal	Server Terminal
		>java Server 6000
>java Client 6000		Waiting for clients
Enter username: Yoda		Client authenticating
Enter password: jedi*knight		Yoda successful login
Welcome to the forum		
	>java Client 6000	
Enter one of the following commands: CRT, MSG, DLT, EDT, LST, RDT, UPD, DWN, RMV, XIT:	Enter username: Yoda	Client authenticating
	Yoda has already logged in	Yoda has already logged in
(extra space added before user's response)	Enter username: R2D2	
	Enter password: c3p0sucks	R2D2 successful login
	Welcome to the forum	
	Enter one of the following commands: CRT, MSG, DLT, EDT, LST, RDT, UPD, DWN, RMV, XIT:	
LST		
No threads to list		Yoda issued LST command
Enter one of the following commands: CRT, MSG, DLT, EDT, LST, RDT, UPD, DWN, RMV, XIT:	(extra space added before user's response)	
(extra space added before user's response)	CRT 3331	R2D2 issued CRT command
	Thread 3331 created	Thread 3331 created
	Enter one of the following commands: CRT, MSG, DLT, EDT, LST, RDT, UPD, DWN, RMV, XIT:	
CRT 3331		Yoda issued CRT command
Thread 3331 exists		Thread 3331 exists
Enter one of the following commands: CRT, MSG, DLT, EDT, LST, RDT, UPD, DWN, RMV, XIT: CRT 9331	(extra space added before user's response)	Yoda issued CRT command
Thread 9331 created		Thread 9331 created
Enter one of the following commands: CRT, MSG, DLT, EDT, LST, RDT, UPD, DWN, RMV, XIT: MSG 3331 Networks Rocks		Yoda issued MSG command
Message posted to 3331 thread		Message posted to 3331 thread
Enter one of the following commands: CRT, MSG, DLT, EDT, LST, RDT, UPD, DWN, RMV, XIT:	MSG 3331 Yes it does	R2D2 issued MSG command
	Message posted to 3331 thread	Message posted to 3331 thread
(extra space added before user's response)	Enter one of the following commands:	

	CRT, MSG, DLT, EDT, LST, RDT, UPD, DWN, RMV, XIT: RDT 3331	
	1 Yoda: Networks Rocks	R2D2 issued RDT command
	2 R2D2: Yes it does	Thread 3331 read
	Enter one of the following commands: CRT, MSG, DLT, EDT, LST, RDT, UPD, DWN, RMV, XIT:	
UPD 9331 test1.exe	(extra space added before user's response)	
test1.exe uploaded to 9331 thread		Yoda issued UPD command
Enter one of the following commands: CRT, MSG, DLT, EDT, LST, RDT, UPD, DWN, RMV, XIT:		Yoda uploaded file test1.exe to 9331 thread
	UPD 9331 test2.exe	
(extra space added before user's response)	test2.exe uploaded to 9331 thread	R2D2 issued UPD command
	Enter one of the following commands: CRT, MSG, DLT, EDT, LST, RDT, UPD, DWN, RMV, XIT:	R2D2 uploaded file test2.exe to 9331 thread
RDT 9331	(extra space added before user's response)	
Yoda uploaded test1.exe		Yoda issued RDT command
R2D2 uploaded test2.exe		Thread 9331 read
Enter one of the following commands: CRT, MSG, DLT, EDT, LST, RDT, UPD, DWN, RMV, XIT:	DWN 9331 test1.exe	R2D2 issued DWN command
	test1.exe successfully downloaded	test1.exe downloaded from Thread 9331
(extra space added before user's response)	Enter one of the following commands: CRT, MSG, DLT, EDT, LST, RDT, UPD, DWN, RMV, XIT:	
EDT 3331 2 This assignment rocks		
The message belongs to another user and cannot be edited		Yoda issued EDT command
	(extra space added before user's response)	Message cannot be edited
Enter one of the following commands: CRT, MSG, DLT, EDT, LST, RDT, UPD, DWN, RMV, XIT: MSG 3331 This assignment rocks		
Message posted to 3331 thread		Yoda issued MSG command
Enter one of the following commands: CRT, MSG, DLT, EDT, LST, RDT, UPD, DWN, RMV, XIT:		Message posted to 3331 thread
	RDT 3331	
	1 Yoda: Networks Rocks	R2D2 issued RDT

(extra space added before user's response) DLT 3331 2	2 R2D2: Yes it does 3 Yoda: This assignment rocks Enter one of the following commands: CRT, MSG, DLT, EDT, LST, RDT, UPD, DWN, RMV, XIT:	command Thread 3331 read
The message belongs to another user and cannot be edited Enter one of the following commands: CRT, MSG, DLT, EDT, LST, RDT, UPD, DWN, RMV, XIT: DLT 3331 1	(extra space added before user's response)	Yoda issued DLT command Message cannot be deleted
The message has been deleted Enter one of the following commands: CRT, MSG, DLT, EDT, LST, RDT, UPD, DWN, RMV, XIT:	RDT 3331 1 R2D2: Yes it does 2 Yoda: This assignment rocks Enter one of the following commands: CRT, MSG, DLT, EDT, LST, RDT, UPD, DWN, RMV, XIT: RMV 9331	Yoda issued DLT command Message has been deleted R2D2 issued RDT command Thread 3331 read
(extra space added before user's response)	Thread cannot be removed Enter one of the following commands: CRT, MSG, DLT, EDT, LST, RDT, UPD, DWN, RMV, XIT: RMV 3331 Thread removed Enter one of the following commands: CRT, MSG, DLT, EDT, LST, RDT, UPD, DWN, RMV, XIT:	R2D2 issued RMV command Thread 9331 cannot be removed R2D2 issued RMV command Thread 3331 removed
XIT Goodbye >	XIT Goodbye >	Yoda exited R2D2 exited Waiting for clients