COMP3331/9331 Computer Networks and Applications

Assignment for Term 1, 2025

Version 1.0

Due: 11:59am (noon) Thursday, 24 April 2025 (Week 10)

Any updates, including corrections and clarifications, will be posted on the subject website. Please make sure to check the website regularly for updates.

1. Change Log

Version 1.0 released on 7th March 2025.

2. Goal and learning objectives

Online discussion forums are widely used for large groups of people to converse on topics of mutual interest. A good example is the online forum used for this course. In this assignment, you will implement an online discussion forum application. Your application is based on a client-server model consisting of one server and multiple clients communicating sequentially (i.e., one at a time) or concurrently. The client and server should communicate using both UDP and TCP. Your application will support various functions typically found on discussion forums, including authentication, creating and deleting threads and messages, reading threads, and uploading and downloading files. However, unlike typical online forums accessed through HTTP, you will design a custom application protocol. Most functions should be implemented over UDP except for uploading and downloading 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 server's interaction 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 programs lacking a computer science component have had limited exposure to programming, especially when

tackling complex assignments. A Non-CSE student is defined as someone not enrolled in a CSE program (either single or dual degree). Examples include students enrolled solely in single-degree programs such as Mechatronics, Aerospace, Actuarial Studies, or Law. Students in dual degree programs that include a CSE program as one of their degrees **do not qualify**. Neither do students visiting from overseas universities who are enrolled in a CSE program at their home university. Any student who meets this criterion and wishes to take advantage of this option MUST email cs3331@cse.unsw.edu to seek approval before **5 pm on 4th April (Friday, Week 7)**. By default, we will assume all students are attempting the CSE version of the assignment unless they have received explicit permission. **No exceptions**.

3. Assignment Specification

In this programming assignment, you will implement the client and server programs of a discussion forum application like the discussion forum used in this course. The key difference is that your application is not web-based (i.e., non-HTTP) but instead uses a custom application layer protocol you will design. The client and server must communicate using both UDP and TCP, as detailed in the remainder of the specification. Your application will support various operations, including creating a new user account, creating and deleting a thread, reading a thread, listing all threads, posting messages on a thread, editing or deleting messages, and uploading or downloading files to and from a thread. To implement these functions, you will develop the application protocol. Most communication between the client and server will occur over UDP, except for file uploads and downloads, which must use TCP. The server will listen on a port specified as a command line argument and wait for messages from the client. When a user runs the client, the authentication process will begin. The client will interact with the user through the command line interface. After successful authentication, the user may initiate one of the available commands. All commands require a simple request-response interaction between the client and the server. The user can execute a series of commands sequentially, ultimately opting to quit. The client and server must print meaningful messages at the command prompt to capture specific interactions. You have the liberty to choose the exact text that is displayed. Examples of client-server interactions are provided in Section 8. All communication between the client and server must take place over UDP, with the sole exception being file uploads and downloads, which require TCP. Implementing these operations will necessitate that your client initiates a TCP connection with the server. The server must first open a TCP port using the port number indicated as a command line argument and listen for connection requests. It is important to note that UDP and TCP port numbers are distinct (e.g., UDP port 53 differs from TCP port 53). Therefore, your server can concurrently open both a UDP and a 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, and the TCP connection must **be promptly closed** once the file transfer is complete.

The assignment will be evaluated in two configurations. In the **first configuration**, the server will interact with one client at a time. This interaction will involve authentication, followed by executing several commands in succession. Multiple clients can connect to the server sequentially; that is, one client is initiated, authenticates, executes several commands in order, and then quits. Subsequently, a second client is initiated, authenticates, executes its commands, and quits, and so on. The server design is significantly simplified (i.e., you won't need to handle concurrency) if you only aim to implement this part of the assignment. Correctly completing this first section is worth **70% of the assignment mark**s (14 marks, see Section 7). In the **second configuration**, the server must interact with multiple clients concurrently. The client design should remain unchanged to meet this requirement. However, the server design will need

adjustments, as it must send and receive messages to and from multiple clients simultaneously. One approach to achieving this is **multi-threading**, although **it is not mandatory.** Note that a correctly implemented server that services multiple clients should also be able to interact appropriately with a single client at any given time. Therefore, if your client and server are designed to fulfil all the functionality required for the second configuration, they should also work as intended in the first configuration.

The server program will run first, followed by one or more instances of the client program (each instance supports one client in the second configuration). They will be executed from different terminals on the **same machine** (allowing you to use localhost, i.e., 127.0.0.1, as the IP address for the server and client in your program). All interactions with the clients will occur 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 can 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`: the port number the server will use to communicate with clients. Since TCP and UDP ports are distinct, the server can open both a UDP and a TCP port with the same port number. Recall that UDP is connectionless, allowing the server to communicate with multiple clients (in the second configuration) through a single server-side UDP socket. Additionally, the server port number does not uniquely identify a TCP socket, making it possible for multiple TCP connections to utilise the same server-side port number when multiple clients simultaneously upload or download files to or from the server in the second configuration.

The server should be launched before the clients. You may assume that the server will run indefinitely. During testing, we will not abruptly terminate the server or client processes (CTRL-C). The server is not expected to retain the state from previous executions. When it is launched, it is assumed that no users are logged on and that the discussion forum is empty, meaning there are no threads or messages. The specification indicates that we will test your code in two configurations. The server will remain running for the entire duration, covering all tests. In other words, the server will not be restarted after completing 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 randomly available port (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 server's current working directory with the correct access permissions set (read and write). This file will contain the usernames 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 the usernames and passwords 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 there is no limit to the number of log in attempts a user may try if they keep entering the wrong password. If the username does not exist, the user is assumed to be 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 ensure that write permissions are enabled for the credentials.txt file (type "`chmod +w credentials.txt`" at a terminal in the server's current working directory). 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 ensure that any new client authenticating with it does not attempt to log in using a username that is already

in use by another active client (i.e., the same username cannot be used concurrently by two clients). The server should maintain a record of all users currently logged in and verify that the username provided by an authenticating client does not match any in this list. If a match is found, a notification should be sent to the server and displayed at the prompt for the user, who should then be asked to enter a different username.

3.3 Discussion Forum Operations

Following successful login, the client displays a message informing the user of all available commands and prompting them 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. After successful login, all available commands should be shown to the user in the first instance. Subsequent prompts for action should include this same message.

If an invalid command is entered, the user should receive an error message and 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 in uppercase (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 occasionally be lost, so you should implement simple mechanisms such as a retransmission timer to deal with the possibility of packet loss. We will leave the specifics for you to decide. In the lectures, we discussed several mechanisms for implementing reliable data transfer. The file upload and download commands (UPD and DWN) should use TCP to transfer 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 client (UPD) or the server (DWN) should initiate the file transfer. 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 the file being transferred reliably.

The execution of each command is described in detail below.

CRT: Create Thread

```
CRT threadtitle
```

The new thread's title (*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 server's current working directory with the same file name as the thread title (*threadtitle*, DO NOT add the ".txt" extension to the name). The first line of the file should contain the username of the person who created the thread. Each subsequent line should be a message added to 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 per the convention noted above (the first line of this file should be the creator's username). 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 to the user at the prompt. 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 to which the message should be posted and the message itself should be included as arguments. The message may contain whitespace (e.g., "Hello, how are you"). The client must send the command (MSG), the title of the thread, the message, and the username to the server. **We will only use short messages (a few words long) in our tests**. 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 specified format, along with the message number (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 to the user at the prompt. 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 should be included as arguments. **A message can only be deleted by the user who originally posted it**. The client sends the command (DLT), the thread title, the message number, and the username to the server. The server must verify if a thread with this title exists, if the corresponding message number is valid, and if this user initially posted the message. If any of these checks fail, an appropriate error message should be sent to the client and displayed as a prompt to the user. If all checks are successful, the server should delete the message, which involves removing the line containing it in the corresponding thread file (all subsequent messages and information about uploaded files in the thread file should be shifted up by one line, and the

message numbers should be updated accordingly). A confirmation should then be sent to the client and displayed as a prompt to the user. The client should subsequently 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 it**. 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, if the corresponding message number is valid, and if the username has posted this message. If any of these checks are unsuccessful, an appropriate error message should be sent to the client and displayed to the user at the prompt. If all checks pass, the server should replace the original message in the corresponding thread file with the new message (the other details associated with this message, such as message number and username, should remain unchanged), and confirmation should be sent to the client and displayed to the user at the prompt. The client should then 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, which replies with a listing of all the thread titles. Only the thread titles should be listed, not the messages. The client should display the list on the terminal, with one thread per line. If there are no active threads, a message indicating this should be shown to the user at the prompt. The client should then 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 the file's contents, including messages and information about uploaded files (see following 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 to the user at the prompt. 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, 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 client's current working directory with the

correct access permissions set (read). You should not assume the file is in a particular format (e. g., text file); rather, assume it is a **binary file**. Be careful not to use functions for file access (reading, writing, etc.) that assume the file is in text format. The client should send the command (UPD), the thread's title, the username, and the file's name to the server. The server should check if a thread with this title exists. If it does not, an appropriate error message must be sent to the client and displayed at the prompt for the user. The server should also check if a file with the provided name exists. If it does, an appropriate error message should be conveyed to the client and displayed at the prompt for 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 must be sent to the client. Following this, the client should transfer the file's contents to the server. All communication between the client and server described so far must occur over UDP. TCP should only be used to transfer the contents of the file. The TCP connection must be closed immediately after the file transfer is completed. The file should be stored in the server's current working directory with the file name threadtitle-filename (do NOT add an extension to the name. If the filename has an extension, 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; that is, 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 with the EDT command or deleted with the DLT command. However, they should be included when a thread is read using the RDT command. Lastly, the server must send a confirmation message to the client, and a corresponding message should be displayed at the user's prompt. The client should then 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 thread's title and the file's name 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, an appropriate error message should be sent to the client and displayed at the prompt to the user. If a match is found, the server should transfer the file's contents 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 to transfer 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 client's current working directory with the same name (*filename,* DO NOT include *threadtitle* in the file name). You may assume the client program can create a file in the current working directory. You may also assume a file with this exact name does not exist in the client's current working directory. 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 on the server end. The client is simply downloading a copy of the file.

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

RMV: Remove Thread

```
RMV threadtitle
```

The title of the thread to be removed must be included as an argument for this action. **Only the user who created a thread can remove it.** The client should send the operation (RMV), the thread title, and the username to the server. The server should first verify if a thread with this title exists and, if so, whether the user who created the thread matches the provided username. If either check fails, an error message should be sent to the client and displayed in the terminal for the user. Otherwise, the thread will be deleted along with the file that stores information about it, any files uploaded to it, and any state maintained about the thread on the server. A confirmation message should be sent to the client, which will then be displayed at the prompt for the user. The client should subsequently prompt the user to select from the available actions.

XIT: Exit

```
XIT
```

There should be no arguments for this command. The client must inform the server that the user is logging off and exit with a goodbye message displayed on the terminal for the user. The server should update its state information about currently logged-in 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** to communicate 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 to transfer 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 for 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 the maximum size of a UDP segment is 65,535 bytes. The loopback interface (127.0.0.1) on the machine where you are executing the program may have a smaller MSS value. However, it is unlikely that you would need to send very large UDP segments, so this should not be an issue.

Since UDP segments can occasionally be lost, you must implement simple mechanisms to recover from them. In the lectures, we discussed several mechanisms for implementing reliable

data transfer. 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 that 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 do not adhere to the transport layer choices noted in the specification, a significant penalty will be assessed.

Client Design

The client program should be straightforward. It must 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 exact text in the samples. Upon initiation, the client should first execute the user authentication process. After authentication, the user should be prompted to enter one of the available commands. Almost all commands require simple request/response interactions between the client and the server. The client does not need to maintain any state regarding the discussion forum.

We expect no changes to your client design as you advance the implementation from the first to the second configuration.

Server Design

The server code will be more involved than the client, as the server is responsible for maintaining the message forum. However, the server design to implement functionality for the first testing configuration 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., no threads, messages, or uploaded files exist**. 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 occur over TCP. The server must 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 following command once the file transfer process has finished and a confirmation message is displayed to the user, followed by a prompt to enter the next command.

While we do not mandate specific details, you must invest some time considering the design of your data structures. Although this is not an exhaustive list, examples of state information include the total number of online users (and their usernames), the total number of threads and their names, and the posts and files associated with each thread. Upon startup, the server can determine the total number of users by reading the entries in the credentials file. However, it is possible to create new user accounts. Therefore, when defining data structures, you cannot assume a fixed number of users upfront. 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 necessary data structures.

Implementing functionality for the second configuration may require some change as the server must interact with multiple clients simultaneously. One robust way to achieve this is to use **multithreading. However, it is not mandatory**. Various approaches could be employed to achieve this functionality. If you use multiple threads, remember 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 a client, the server should receive a request for a specific command, take all necessary actions, and respond to the client accordingly before processing the subsequent request. This process resembles what you would have implemented to achieve the functionality of the initial configuration.

You may assume that each interaction with a client is **atomic**. For example, if client A initiates an interaction (i.e., any of the 10 commands or the authentication process) with the server, the server cannot be interrupted by a command from another client, B, while processing this interaction. Client B's command will only be executed after the server has finished processing client A's command. In the context of UDP and DWN, this means that at any given time, the server can either receive or send a file to or from at most one client. Thus, the server should manage only one TCP connection at any time.

Consider how multiple threads interact with the various data structures if you choose multi-threading. 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 accomplish all interactions expected in the first configuration correctly.

4. Additional Notes

- This is NOT a group assignment. You are expected to work on this individually.
- **Tips for getting started**: The best way to tackle a complex implementation task is to approach it in stages. We recommend that you first implement the functionality for the initial configuration, meaning the server interacts with a single active client at any given time. A good starting point would be to develop the functionality that allows a single user to log in with the server. Next, add the ability to execute a single command. Ensure you thoroughly test the operation of each command, including typical error conditions, before progressing to the next step. We suggest beginning with more straightforward commands, such as CRT, MSG, and LST, before advancing to more complex commands like UPD and DWN. Once you have thoroughly tested your code for the initial configuration, move on to the second configuration. It is crucial to rigorously test your code to ensure that all possible (and logical) interactions can be executed correctly—t**est, test, and test**.
- **Application Layer Protocol:** Remember that you are implementing an application layer protocol to create a fully functional discussion forum. You need to design the format (syntax and semantics) of the messages exchanged between the client and server and the actions each entity takes upon receiving these messages. We do not impose any specific requirements regarding the design of your application layer protocol. Our primary concern is the outcome, namely the functionality described above. Consider revisiting some of the application layer protocols we have studied (like HTTP, SMTP, etc.) to examine examples of message formats and actions taken.
- You are not required to use the exact wording for the messages displayed to the user on the terminal, as illustrated in the examples in Section 8. However, please ensure that the text is unambiguous.

- **Backup and Versioning:** We strongly recommend backing up your programs frequently. CSE backs up all user accounts nightly. If you are developing code on your personal machine, it is highly advised that you perform daily backups. We also suggest using a reliable versioning system to enable you to roll back and recover from any unintended changes. Many easy-to-use services are available for both tasks. We will NOT entertain any requests for special consideration due to issues related to computer failure, lost files, etc.
- **Language and Platform**: You can implement this assignment using C, Java, or Python. Please choose a language that you are comfortable with. The programs will be tested on CSE Linux machines. So please make sure that your entire application runs correctly in VLAB. This is especially important if you plan to develop and test the programs on your personal computers (which may use a different OS or version or IDE). CSE machines support **gcc version 12.2, Java 17, and Python 3.11.** You may only use the basic socket programming APIs provided 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. It is best to check with the course staff on the forum if unsure.
- **Debugging**: When implementing a complex assignment like this, errors in your code are inevitable. We strongly encourage you to follow a systematic approach to debugging. If you use an IDE for development, it will likely have debugging functionalities. Alternatively, you could use a command-line debugger such as pbd (Python), jdb (Java), or gdb (C). Utilise one of these tools to step through your code, set breakpoints, and observe the values of relevant variables and messages exchanged. Proceed step by step, checking and eliminating possible causes until you identify the underlying issue. Please note that we cannot debug your code on the course forum or during help sessions.
- Multithreading is NOT a requirement. If you choose to implement it, you can refer to the following resources for multi-threading. Note that you won't need to implement very complex aspects of multi-threading for this assignment.
    - Python: https://www.tutorialspoint.com/python3/python_multithreading.htm
    - Java: https://www.javatpoint.com/how-to-create-a-thread-in-java
    - C: https://www.geeksforgeeks.org/multithreading-in-c/
- You are encouraged to use the course forum to ask questions and discuss various approaches to solving the problem. However, please refrain **from** posting your solution or any code snippets on the forums.
- **Programming Tutorial**: In Week 7, we will run programming tutorials during regular lab times to help students learn to program some of the assignment's building blocks. A schedule for these sessions will be announced in Week 6.
- **Assignment Help Sessions:** We will run additional consultations in Weeks 7-10 for all 3 programming languages to assist you with assignment-related questions. A schedule will be posted on the assignment page of the course website. Please note that these sessions are not a forum for tutors to debug your code.

5. Submission

Please use the mandated file names (see Section 3.1). You may have additional header files and/or helper files. If you are using C, 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 you 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 certain 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.

Alternatively, you can 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 develop the code natively on your machine using 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. We cannot award significant marks if the submitted code does not run during marking.**
- You may submit multiple times before the deadline. A later submission will override an earlier one, so please ensure that you submit the correct file. Avoid waiting until the last moment to submit, as technical or network errors may prevent you from rectifying the issue in time.

Late Submission Penalty: The UNSW standard late penalty, 5% per day of the maximum available mark, will apply for up to 5 days. This assignment is worth 20 marks, so the penalty equates to a 1-mark deduction per day late. Submissions after 5 days will not be accepted.

6. Plagiarism

You are to write all the code for this assignment yourself. All source code is subject to strict plagiarism checks using sophisticated detection software. These checks may include comparisons with available code from internet sites and assignments from previous terms. In addition, each submission will be checked against all other submissions from the current term.

Please 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 very seriously. The LIC will decide on the appropriate penalty for detected cases of plagiarism. The most likely penalty would be to reduce the assignment mark to zero. We recognise that much learning occurs in student conversations and do not wish to discourage those. However, it is essential for both those helping others and those being helped not to provide or accept any programming language code in writing, as this may be used as is and lead to plagiarism penalties for both the provider and copier of the code. Feel free to write something on paper but tear it up or take it away when the discussion ends. Borrowing bits and pieces of code from sample socket code available online and in books is acceptable. However, you must acknowledge the source of any borrowed code. This means referencing a book or a URL when the code appears (as comments). Also, indicate the portions of your code that are borrowed in your report. 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 plagiarism.**

Generative AI Tools: The use of any software or service to search for or generate information or answers is prohibited. If detected, such use will be considered serious academic misconduct and may result in standard penalties, which could include 00FL, suspension, and exclusion.

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 test typical usage scenarios for the aforementioned functionality and some simple error conditions. A typical marking session will last approximately 15 to 20 minutes. We will spawn a maximum of three concurrent clients when testing with multiple concurrent clients. However, please avoid hard coding any specific limits in your programs. We will not test your code under very complex scenarios or extreme edge cases.

8. Sample Interaction

The following provides 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.

First Configuration

In this configuration, the server interacts with a single client at any given time. It is recommended to run the client and server in separate working directories. Ensure that write permissions are enabled for the credentials file. Below, two clients with usernames Yoda and Obi-wan connect and interact with the server sequentially in that order. The inputs from the user are displayed as underlined in the client terminal. Extra spacing is added in the server terminal to align the output with the 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	Client authenticating
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	
Thread 3331 created	Yoda issued CRT command
Enter one of the following commands: CRT, MSG, DLT, EDT, LST, RDT, UPD, DWN, RMV, XIT: CRT 3331	Thread 3331 created
Thread 3331 exists	Yoda issued CRT command
Enter one of the following commands: CRT, MSG, DLT, EDT, LST, RDT, UPD, DWN, RMV, XIT: CRT 9331	Thread 3331 exists
Thread 9331 created	Yoda issued CRT command
Enter one of the following commands: CRT, MSG, DLT, EDT, LST, RDT, UPD, DWN, RMV, XIT: LST 3331	Thread 9331 created
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	
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: RDT	Message posted to 3331 thread
Incorrect syntax for RDT	
Enter one of the following commands: CRT, MSG, DLT, EDT, LST, RDT, UPD, DWN, RMV, XIT: RDT 9331	
Thread 9331 is empty	Yoda issued RDT command
Enter one of the following commands: CRT, MSG, DLT, EDT, LST, RDT, UPD, DWN, RMV, XIT: RDT 3331	Thread 9331 read
1 Yoda: Networks is awesome	Yoda issued RDT command
Enter one of the following commands: CRT, MSG, DLT, EDT, LST, RDT, UPD, DWN, RMV, XIT: UPD 3331 test.exe	Thread 3331 read

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

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	
The thread was created by another user and cannot be removed	Obi-wan issued RMV command
Enter one of the following commands: CRT, MSG, DLT, EDT, LST, RDT, UPD, DWN, RMV, XIT: RMV 9331	Thread 3331 cannot be removed
The thread has been removed	Obi-wan issued RMV command
Enter one of the following commands: CRT, MSG, DLT, EDT, LST, RDT, UPD, DWN, RMV, XIT: LST	Thread 9331 removed
The list of active threads:	Obi-wan issued LST command
3331	
Enter one of the following commands: CRT, MSG, DLT, EDT, LST, RDT, UPD, DWN, RMV, XIT: XIT	
Goodbye	Obi-wan exited
>	Waiting for clients

Second Configuration

In this configuration, the server interacts concurrently with multiple clients. In the following example, two clients with usernames Yoda and R2D2 connect and interact with the server simultaneously. The inputs from the users are shown as underlined. You must execute each 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 will already be executing, as we will have conducted tests for the first configuration. The server will not be restarted between the tests for the two configurations. Additionally, extra space is added in the two client terminals to simulate some delay before the users enter commands when prompted. This is done to improve the 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
CRT 3331	Enter one of the following commands: CRT, MSG, DLT, EDT, LST, RDT, UPD, DWN, RMV, XIT:	
Thread 3331 exists		Yoda issued CRT command
Enter one of the following commands: CRT, MSG, DLT, EDT, LST, RDT, UPD, DWN, RMV, XIT: CRT 9331		Thread 3331 exists
	(extra space added before user's response)	
Thread 9331 created		Yoda issued CRT command

Enter one of the following commands: CRT, MSG, DLT, EDT, LST, RDT, UPD, DWN, RMV, XIT: MSG 3331 Networks Rocks		Thread 9331 created
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:	MSG 3331 Yes it does	Message posted to 3331 thread
(extra space added before user's response)	Message posted to 3331 thread	R2D2 issued MSG command
	Enter one of the following commands: CRT, MSG, DLT, EDT, LST, RDT, UPD, DWN, RMV, XIT: RDT 3331	Message posted to 3331 thread
	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)	Yoda issued UPD command
test1.exe uploaded to 9331 thread		Yoda uploaded file test1.exe to 9331 thread
Enter one of the following commands: CRT, MSG, DLT, EDT, LST, RDT, UPD, DWN, RMV, XIT:	UPD 9331 test2.exe	
	test2.exe uploaded to 9331 thread	R2D2 issued UPD command
(extra space added before user's response)	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		
Yoda uploaded test1.exe	(extra space added before user's response)	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		Yoda issued EDT

another user and 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 Enter one of the following commands: CRT, MSG, DLT, EDT, LST, RDT, UPD, DWN, RMV, XIT: (extra space added before user's response) DLT 3331 2 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 The message has been deleted 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) RDT 3331 1 Yoda: Networks Rocks 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: (extra space added before user's response) 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 Thread cannot be removed Enter one of the following commands: CRT, MSG, DLT, EDT, LST, RDT, UPD, DWN, RMV, XIT: RMV 3331	command Message cannot be edited Yoda issued MSG command Message posted to 3331 thread R2D2 issued RDT command Thread 3331 read Yoda issued DLT command Message cannot be deleted Yoda issued DLT command Message has been deleted R2D2 issued RDT command Thread 3331 read R2D2 issued RMV command Thread 9331 cannot be removed

XIT Goodbye >	Thread removed Enter one of the following commands: CRT, MSG, DLT, EDT, LST, RDT, UPD, DWN, RMV, XIT: XIT Goodbye >	R2D2 issued RMV command Thread 3331 removed Yoda exited R2D2 exited Waiting for clients