COMP3331/9331 Computer Networks and Applications

Assignment for Term 3, 2024

Version 1.1

Due: 11:59am (noon) Friday, 8 November 2024 (Week 9)

Updates to the assignment, including any corrections and clarifications, will be posted on the course website. Please make sure that you check the course website regularly for updates.

Table of Contents

BitTrickle File Sharing System

1. Goal and Learning Objectives

In this assignment you will have the opportunity to implement *BitTrickle*, a permissioned, peer-to-peer file sharing system with a centralised indexing server. Your applications are based on both client-server and peer-to-peer models, consisting of one server and multiple clients, where:

1. The **server** authenticates users who wish to join the peer-to-peer network, keeps track of which users store what files, and provides support services for users to search for files and to connect to one another and transfer those files directly.
2. The **client** is a command-shell interpreter that allows a user to join the peer-to-peer network and interact with it in various ways, such as making files available to the network, and retrieving files from the network.

All client-server communication **must** be over **UDP**. All peer-to-peer (i.e. client-to-client) communication **must** be over **TCP**.

You will additionally submit a short report.

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

1. A deeper understanding of both client-server and peer-to-peer architectures.
2. Socket programming with both UDP and TCP transport-layer protocols.
3. Designing an application layer protocol.

The assignment is worth **20%** of your final grade.

2. Assignment Specification

To provide some context, a high-level example of BitTrickle's primary file sharing functionality is shown in Figure 1. In addition to the central indexing server, there are two authenticated, active peers in the network, "A" and "B". The exchange of messages are:

1. "A" informs the server that they wish to make "X.mp3" available to the network.
2. The server responds to "A" indicating that the file has been successfully indexed.
3. "B" queries the server where they might find "X.mp3".
4. The server responds to "B" indicating that "A" has a copy of "X.mp3".
5. "B" establishes a TCP connection with "A" and requests "X.mp3".
6. "A" reads "X.mp3" from disk and sends it over the established TCP connection, which "B" receives and writes to disk.

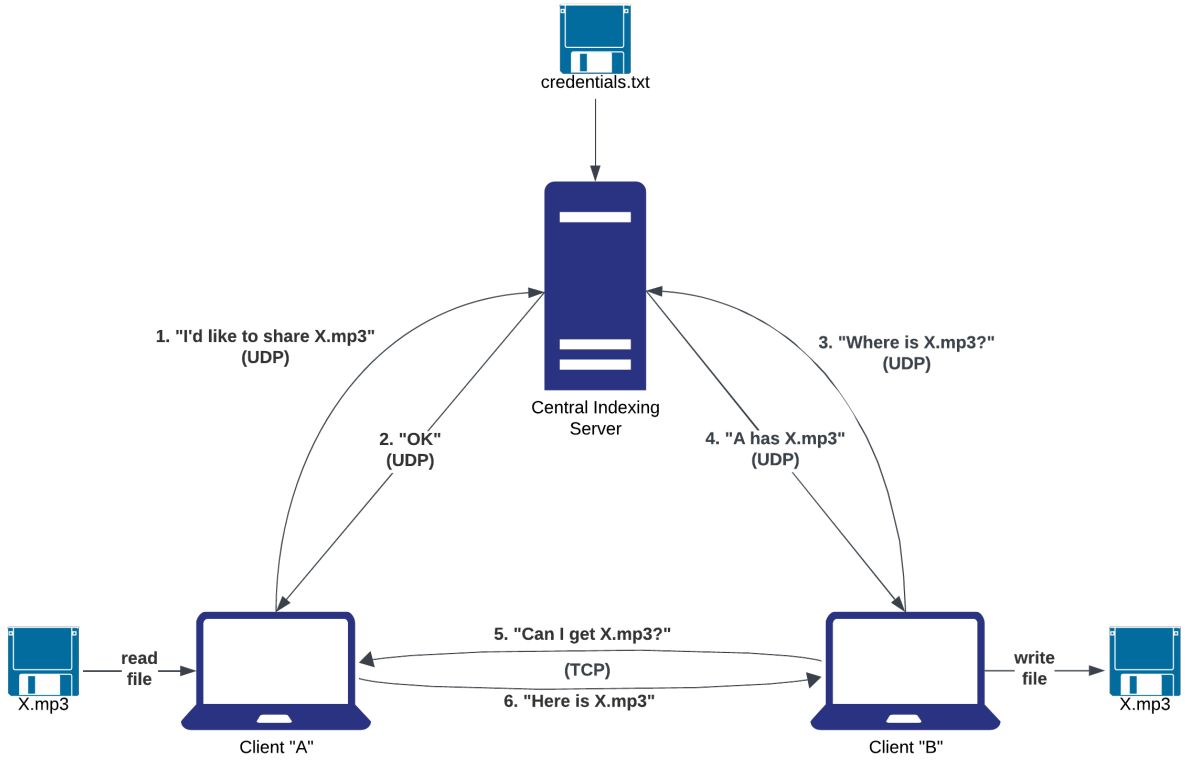

Figure 1: Sharing a file with BitTrickle

2.1. Authentication

Before a user can join the network and become an active peer, they must first authenticate with the server. When the client is executed, it should prompt the user to enter their username and password. These should be sent to the server, with the server response indicating success or failure. Authentication should fail if:

- The username is unknown; or
- The username is known, but the password does not match; or
- The user is currently active (see 2.2. Heartbeat Mechanism).

Should authentication fail, an appropriate message should be printed, and the user should again be prompted to enter their username and password.

Should authentication succeed, the peer should become active (see 2.2. Heartbeat Mechanism) and the user may start issuing commands (see 2.3. Client Commands).

The list of users authorised to join the network is stored in a credentials file. You may assume that during marking any username and password entered by the user will be non-blank and well formed, as per the rules of the credentials file (see 2.1.1. Credentials File).

2.1.1. Credentials File

The credentials file is a text file available to the server that contains the usernames and passwords of those authorised to join the network. This file is **not** available at the client.

Usernames and passwords are each limited to 16 characters and shall consist only of printable ASCII characters. Each line of the credentials file contains one username-password pair, separated by a single space, with no leading or trailing whitespace, and is terminated by a single `'\n'` newline character.

You may assume that each username in the credentials file will be unique, but you should not assume any particular ordering of the credentials file. You may assume that the server has sufficient resources to store all credentials in main memory, should you choose to read the file once on server startup.

The credentials file will be named `credentials.txt`. It will be in the working directory of the server, and it will follow the format and rules stated above.

Do not hard code any credentials within your server program. These must be read from the file when the server is executed. Failure to do so will likely result in your applications not passing any testing.

Do not hard code any path to the credentials file, or assume any name other than `credentials.txt`, within your server program. The program must read the file `credentials.txt` from the current working directory. Failure to do so will likely result in your applications not passing any testing.

A sample credentials file is provided on the assignment page of the course website. A different credentials file will be used during marking. You may assume the file will be present in the working directory of the server, will have the same name, will have the necessary permissions, will follow the same format and rules, and will remain present and unmodified for the lifetime of the server.

Your server program is not expected to create new or alter existing accounts, and your server is not expected to modify this file in any way.

2.2. Heartbeat Mechanism

The server provides various support services to the peer-to-peer network by maintaining state as to which peers are currently active. Consider the example in Figure 1. When “B” queries the server where it might find “X.mp3”, the server should only direct “B” to connect to “A” if “A” is active.

To achieve this, BitTrickle utilises a “heartbeat” mechanism. These are messages sent periodically by each client to the server to remind the server that the peer is still active. Specifically, after being authenticated, the client should send such a message every 2 seconds. Meanwhile, the server should only consider a peer active if, within the last 3 seconds, the user authenticated, or the server received a heartbeat from that peer.

You may assume that the 1 second difference is sufficient to account for any reasonable network and server latency. You may also assume that heartbeat messages will be delivered reliably.

Note, the heartbeat mechanism will need to run in a separate thread of the client so that the client can concurrently accept and act upon user input (see 3.2. Client Design).

2.3. Client Commands

The client, upon authenticating the user, should support the following 7 commands. As an interactive shell, the client does not need to accept further user input until the previous command has completed.

While it is good programming practice to validate all user input, you may assume that during marking all user input will be well formed. That is, only valid commands will be entered, in lower case, and where the command takes an argument, then an argument will be given, with a single space separating the command and the argument. The argument will also be well formed, case sensitive, and contain no internal whitespace. Furthermore, user input will contain no leading or trailing whitespace.

2.3.1. get <filename>

The client should query the server for any active peers that have published (i.e. shared) a file with an exactly matching filename. If there are multiple such peers, the server (or client) may select one arbitrarily. The client should then establish a TCP connection with the peer, download the file to its working directory, and print a message indicating success. If there are no active peers with a file that matches, then the client should print a message indicating failure.

You may assume that:

- The file will remain published via the uploading peer until the transfer is complete.
- The uploading peer will remain active until the transfer is complete.
- The uploading peer will have the necessary permissions to read the file in its working directory, and the downloading peer will have the necessary permissions to create a new file in its working directory and write to it.
- The downloading peer will have no pre-existing file in its working directory with a matching name. By extension, the downloading peer will not have published a matching file.
- File names are globally unique, that is, two different files will not share the same name.

Also note:

- The uploading peer must still be able to accept and execute user commands while any uploading is taking place.
- Multiple clients may be downloading files from the same peer at the same time. These files may be the same and/or different.
- As with the heartbeat mechanism, this requires the client code to be multithreaded (see 3.2. Client Design).

2.3.2. lap

The client should query the server for a list of active peers and print out their usernames. The list should not include the peer issuing the query. The usernames may be printed in any order. If there are no active peers, then a suitable message should be printed.

2.3.3. lpf

The client should query the server for a list of files that are currently published (i.e. shared) by the user and print out their names. The file names may be printed in any order. If the user currently has no published files, then a suitable message should be printed.

Note, this is intended as a server request so that the client isn't required to maintain any persistent state. That is, a user can publish one or more files, go offline, and should they come back online, then any files previously published will still be listed without any need to store that information locally or re-publish.

2.3.4. pub <filename>

The client should inform the server that the user wishes to publish a file with the given name. This command should be idempotent, that is, executing it repeatedly with the same argument should have no additional effect. The file should remain published until such time as it may be unpublished, regardless of whether the peer remains active. That is, a user can publish one or more files, go offline, and should they come back online, then any files previously published will still be shared without any need to re-publish.

You may assume that the file exists in the working directory of the client, with the necessary read permissions, and will remain present throughout testing.

2.3.5. sch <substring>

The client should query the server for any files published by active peers that contain the case sensitive substring within their name and print out the names of any such files. The list should not include any files published by the user issuing the query. The file names may be printed in any order. If there are no files published by active peers that contain the substring, then a suitable message should be printed.

2.3.6. unp <filename>

The client should inform the server that it wishes to unpublish a file with the given name and print out a message indicating success or failure. Failure would occur if the user has no published file with a matching name.

2.3.7. xit

The client should gracefully terminate. Note, this requires no server communication. Given the client will no longer send heartbeats, the server should infer that the peer has left the network.

2.4. File Names and Execution

The main code for the server and client must be contained in one of the following files, based on your choice of language:

Language	Client	Server

C	`client.c`	`server.c`
Java	`Client.java`	`Server.java`
Python	`client.py`	`server.py`

You are free to create additional helper files and name them as you wish.

Both the server and client should accept the following command-line argument:

- `server_port`: this is the UDP port number on which the server should listen for client messages. We recommend using a random port number between 49152 and 65535 (the dynamic port number range).

Both applications should be given the same port argument. During marking, you can assume this to be the case, and that the port argument will be valid.

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

C:

```
$ ./server server_port
```

Java:

```
$ java Server server_port
```

Python:

```
$ python3 server.py server_port
```

Upon execution, the server should bind to the UDP port number given as a command-line argument.

Note, you do not need to specify the port to be used by the client to communicate with the server. The client program should allow the operating system to allocate any available UDP port, which the server will learn upon receipt of a message.

Similarly, the TCP port on which each client will listen for peer connections does not need to be specified in advance. Each client can establish a TCP socket, allowing the operating system to allocate any available port, and then communicate the allocated port number to the server. The server can then store this information and share it with other clients as needed to support peer-to-peer file sharing.

Each client should be initiated in a separate terminal as follows:

C:

```
$ ./client server_port
```

Java:

```
$ java Client server_port
```

Python:

```
$ python3 client.py server_port
```

The server and all clients will be tested on the same host, so you may hardcode the interface as 127.0.0.1 (localhost). As we'll be using the loopback interface, you can assume that no UDP packets will be lost, corrupted, or re-ordered "in flight".

2.5. Application Output

We do not mandate the exact text that should be displayed by the client or the server. However, you must make sure that the displayed text is easy to comprehend and adequately captures the behaviour of each application and their interactions.

Some examples illustrating client-server interaction are provided in H. Sample Interactions.

Please **do not** email course staff or post on the forum asking if your output is acceptable. This is not scalable for a course of this size. If you are unsure of your output, then you should follow the example output.

3. Program Design Considerations

3.1. Server Design

The server program should be less involved than the client. Most commands require simple request/response interactions over UDP between the client and server. On the server side this can be achieved with a single thread of execution and a single socket. The main server complexity will be in defining and utilising data structures to manage the state of the peer-to-peer network.

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 the time when each peer was last active (or the time it will be deemed inactive), the current address of their TCP welcoming socket, the files that are published on the network, and a list of users who have published each of those files.

As you may have learnt in your programming courses, it is not good practice to arbitrarily assume a fixed upper limit on such data structures. Thus, we strongly recommend allocating memory dynamically for all the data structures that are required.

Should you choose to implement a multithreaded server, then you should be particularly careful about how multiple threads will interact with the various data structures.

3.2. Client Design

While the client program won't need to maintain much state, it will be more complex than the server, as it will need to manage multiple sockets and multiple threads of execution. Figure 2 illustrates the sockets involved in the example of Figure 1. Each client minimally has a UDP socket to communicate with the server, and a TCP welcoming socket to listen for file download requests from other peers.

When a download takes place, in this case with client “B” requesting a file from client “A”, “B” creates a new TCP client socket and initiates a three-way handshake with the welcoming socket of “A”. In this context, “B” can be viewed as the client and “A” as the server in a client-server model. Once the handshake is complete, a new connection socket will be created at “A”, and the data transfer can commence.

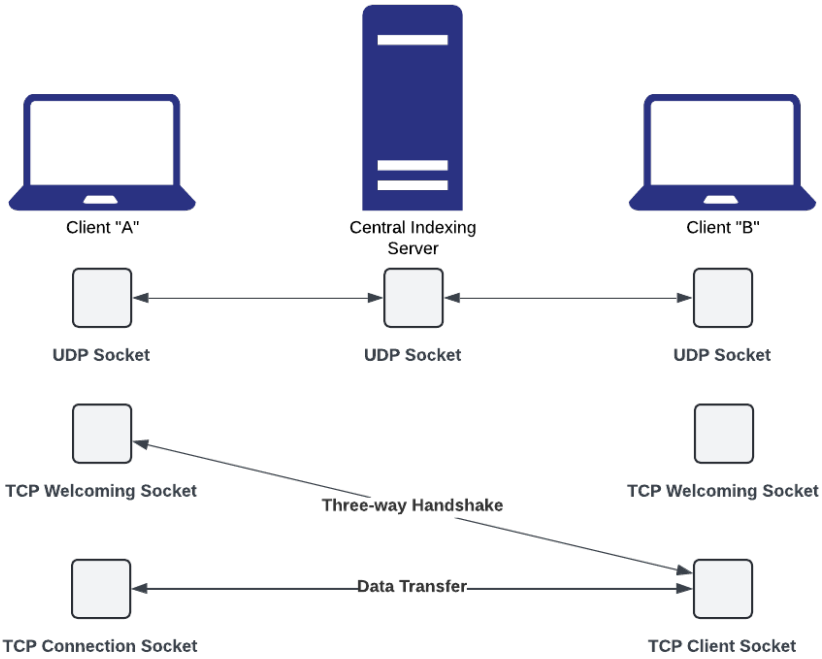

Figure 2: BitTrickle sockets from the example of Figure 1

Of course the peer-to-peer network may be more complex. “A” may be managing any number of connection sockets to support multiple peers that are downloading files simultaneously. Meanwhile “A” should be sending heartbeats periodically to the server and might also be downloading a file itself from another peer in response to a user command. A robust way to achieve this is to use **multithreading**.

In this approach there could be three threads that are running for the life of the client:

1. To accept and act upon user input.
2. To periodically send heartbeat messages to the server via UDP.
3. To listen for download requests via the TCP welcoming socket, and, when one is received:
    a. Launch a short-lived thread with the TCP connection socket to perform the file transfer.

When the client starts up it should create a UDP socket to communicate with the server and a TCP socket to accept peer-to-peer download requests. It could then spin off a new thread to listen for download requests on the TCP socket (thread 3 above). The main thread (thread 1 above) could then initiate the authentication process. At some stage the client will need to send the address of its TCP socket to the server. This could be included as part of the authentication exchange or sent immediately after having successfully authenticated. Once successfully authenticated, the client

could spin off a new thread to periodically send heartbeat messages (thread 2 above). The main thread (thread 1 above) could then enter its interactive shell loop, prompting the user for input and executing each command.

3.3. Application Layer Protocol

You are implementing an application layer protocol for realising the BitTrickle file sharing system. You will have to design the format (both syntax and semantics) of the messages exchanged between the participants, and the actions taken by each entity on receiving these messages. We do not mandate any specific requirements regarding the design of your application layer protocol. We are only concerned with the end result, i.e. the outlined functionality of the system. 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.

3.4. Message / Buffer Sizes

You **may** assume that the payload of UDP-based application messages (e.g. a list of active peers or a list of matching files) does not exceed 1024 bytes. You are free to nominate appropriate buffer sizes for these messages based on the additional overhead of your application layer protocol.

You **should not** assume that files transferred via TCP can fit within any nominal buffer. The uploading peer may need to repeatedly read a chunk of the file and write it to the socket, and correspondingly, the downloading peer may need to repeatedly read a chunk from the socket and write it to the file.

3.5. Name, Type, and Size of Published Files

You **may** assume that all files published on BitTrickle will have names no greater than 16 characters in total, consisting only of alphanumeric, dash, dot, and underscore ASCII characters. File names are case sensitive. For example, `a.txt` is distinct from `A.txt`. You may assume that during marking, filename and substring arguments provided as user input will follow these same rules.

You **should not** assume that files published on BitTrickle are of a particular format or encoding. They should be read and written in binary mode as raw bytes, rather than in text mode.

You **should not** assume that files published on BitTrickle are constrained to a particular size limit.

You are encouraged to use tools like `diff` to verify that any received file is an exact duplicate of the original.

4. Report

You should submit a small report (no more than 3 pages) named `report.pdf`. This should include:

- Details of which language you have used (e.g., C) and the organisation of your code (`Makefile`, directories if any, etc.).
- The overall program design, for example, the main components (client, server, helper classes/functions) and how these components interact.

- Data structure design, for example, any data structures used by the server to maintain state about published files and active peers.
- Application layer protocol message format(s).
- Known limitations, for example, if your program does not work under certain circumstances, then report those circumstances.
- Also indicate any segments of code that you have borrowed from the Web or other sources.

Appendices

A. Languages and Permitted Features

You are free to use C, Java, or Python. Please choose a language with which you are comfortable.

Your submission will be tested on the CSE servers. For this reason, it is **critical** that your code can compile and run in that environment, using the standard system installations. This is especially important if you plan to develop and test your code on your personal computer.

For your reference, the CSE machines currently support the following:

- C: gcc v12.2
- Java: openjdk v17.0
- Python: python3 v3.11

If your code does not run on the CSE servers, then it cannot be marked, and it will be awarded ZERO.

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. Attempting to circumvent the intent of the assignment may similarly result in a mark of ZERO.

If you are unsure about anything, please check with the course staff on the forum.

B. Additional Notes

- This is **not** a group assignment. You are expected to work on this individually.
- **Sample Code**: Code snippets are provided on the course website in all 3 programming languages. These snippets provide examples of socket programming with both TCP and UDP, and multithreading. You are not required to use any of the provided snippets, but you are welcome to adapt them as you see fit.
- **Programming Tutorial**: We will run programming tutorials, for all 3 programming languages, during regular lab times in Week 7. These will provide further discussion and practice around socket programming and multithreading. A schedule for these sessions will be announced no later than Week 6.
- **Assignment Help Sessions**: We will run additional consultations, for all 3 programming languages, through Week 7 to 9, to assist with assignment related questions. A schedule will be posted on the assignment page of the course website no later than Week 6. Please note, these sessions are not a forum for tutors to debug your code.
- **Backup and Versioning**: We strongly recommend you back-up your programs frequently. CSE backs up 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 this, which are easy to use. We will **not** entertain any requests for special consideration due to issues related to computer failure, lost files, etc.

- **Debugging**: When implementing a complex assignment such as this, there are bound to be errors in your code. We strongly encourage that you follow a systematic approach to debugging. If you are using an IDE for development, then it is bound to have debugging functionalities. Alternatively, you could use a command line debugger such as `pbd` (Python), `jdb` (Java) or `gdb` (C). Use one of these tools to step through your code, create break points, observe the values of relevant variables and messages exchanged, etc. Proceed step by step, check and eliminate the possible causes until you find the underlying issue. Note that, we won't be able to debug your code on the course forum or in the help sessions.
- 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**.
- You are encouraged to use the course discussion forum to ask questions and to discuss different approaches to solve the problem. However, you **must not** post your solution or any code fragments on the forum, or on any other public medium.

C. Submission

Please ensure that you use the mandated file names of `report.pdf` and for the entry point of each application. These are:

Language	Client	Server
C	`client.c`	`server.c`
Java	`Client.java`	`Server.java`
Python	`client.py`	`server.py`

If you are using C or Java, then you **must** additionally submit a `Makefile`. This is because we need to know how to resolve any dependencies. After running `make`, we should have the following executable files:

Language	Client	Server
C	`client`	`server`
Java	`Client.class`	`Server.class`

You should **not** submit a `credentials.txt`, any build artefacts, or any files used to test file transfer between peers.

If your submission does not rely on a directory structure, then you may submit the files directly. For example, assuming the relevant files are in the current working directory:

```
$ give cs3331 assign client.c server.c Makefile report.pdf
```

However, if your submission does rely on some directory structure, then you **must** first `tar` the parent directory as `assign.tar`. For example, assuming a directory `assign` contains all the relevant files and sub-directories. Open a terminal and navigate to the parent directory of `assign`, then:

```
$ tar -cvf assign.tar assign
$ give cs3331 assign assign.tar
```

Upon running `give`, ensure that your submission is accepted. You may submit often. Only your last submission will be marked.

Emailing your code to course staff will not be considered as a submission.

Submitting the wrong files, failing to submit certain files, failing to complete the submission process, or simply failing to submit, will not be considered as grounds for re-assessment.

If you wish to validate your submission, you may execute:

```
$ 3331 classrun -check assign  # show submission status
$ 3331 classrun -fetch assign  # fetch most recent submission
```

Important: It is your responsibility to ensure that your submission is accepted, and that your submission is what you intend to have assessed. **No exceptions**.

D. Late Submission Policy

Late submissions will incur a 5% per day penalty, for up to 5 days, calculated on the achieved mark. Each day starts from the deadline and accrues every 24 hours.

For example, an assignment otherwise assessed as 12/20, submitted 49 hours late, will incur a 3 day x 5% = 15% penalty, applied to 12, and be awarded 12 x 0.85 = 10.2/20.

Submissions after 5 days from the deadline will not be accepted unless an extension has been granted, as detailed in E. Special Consideration and Equitable Learning Services.

E. Special Consideration and Equitable Learning Services

Applications for Special Consideration **must** be submitted to the university via the Special Consideration portal. Course staff do **not** accept or approve special consideration requests.

Students who are registered with Equitable Learning Services **must** email cs3331@cse.unsw.edu.au to request any adjustments based on their Equitable Learning Plan.

Any requested and approved extensions will defer late penalties and submission closure. For example, a student who has been approved for a 3 day extension, will not incur any late penalties until 3 days after the standard deadline, and will be able to submit up to 8 days after the standard deadline.

F. Plagiarism

You are to write all the code for this assignment yourself. All source code is subject to strict checks for plagiarism, via highly sophisticated plagiarism detection software. These checks may include comparison with available code from Internet sites and assignments from previous 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 the

appropriate penalty for detected cases of plagiarism. The most likely penalty will 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 or accept any programming code in writing, as this is likely to be used exactly as is, and lead to plagiarism penalties for both the supplier and the copier of the code. 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 short snippets of 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 (as comments) where the code appears. 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.

Generative AI Tools: It is prohibited to use any software or service to search for or generate information or answers. If its use is detected, it will be regarded as serious academic misconduct and subject to the standard penalties, which may include 00FL, suspension and exclusion.

G. Marking Policy

G.1. Preliminary Checks

We will perform both automated and manual checks to assess:

1. That the submitted code is original. See F. Plagiarism.
2. That the submitted code can compile and run in the standard CSE lab environment, and only uses permitted features. See A. Languages and Permitted Features.
3. That the submitted code utilises UDP for all client-server communications and TCP for all peer-to-peer communications.

Failing any of the above will likely result in a mark of ZERO, and in the case of plagiarism, possibly further disciplinary action.

G.2. Marking Process

While marking, the server will be executed first and will remain online for the entire duration of the tests. We will not abruptly kill the server or client processes (Ctrl-C). Should the server crash due to a bug, then it and any clients will be restarted before marking continues. Should the server need to be restarted, it is not expected to maintain any state from previous executions. However, to maximise your mark, you are urged to test your code thoroughly and resolve any bugs prior to submission.

A typical marking session will last for approximately 15 minutes during which we will initiate at most 5 clients. However, you should not hard code any specific limits in your programs. We will be testing under typical usage scenarios for the functionality described in the Assignment Specification. We won’t be testing your code under very complex scenarios or extreme edge cases.

We will verify that the descriptions in your report reflect the actual implementations in your programs.

G.3. Marking Rubric

Functionality	Marks
Authentication (see 2.1. Authentication)	2
Heartbeat mechanism (see 2.2. Heartbeat Mechanism)	1
List active peers (see 2.3.2. lap)	2
Publish a file (see 2.3.4. pub <filename>)	1
List published files (see 2.3.3. lpf)	2
Search for files (see 2.3.5. sch <substring>)	2
Download a file (see 2.3.1. get <filename>)	6
Unpublish a file (see 2.3.6. unp <filename>)	1
Exit (see 2.3.7. xit)	1
Report (see 4. Report)	1
Properly documented and commented code	1
Total Marks	**20**

No particular coding style guide is mandated, but you might consider following:

- CSE C Coding Style Guide
- Google Java Style Guide
- PEP 8 - Style Guide for Python Code

H. Sample Interactions

Note that the following may not be exhaustive but should be useful to get a sense of what is expected. We are assuming Python as the implementation language.

- $ represents the terminal prompt.
- > represents the client prompt.
- … represents a section of output during which there was no user activity, and the server was only logging heartbeats, that has been removed for clarity.
- The actual terminal windows contain no blank lines. Blank cells have been added to the sample interactions to provide some indication of timing alignment between terminal windows, with file download durations also exaggerated relative to the server log.
- The first two fields of the server log are the time, to millisecond precision, and client port.

Assume we have a directory `assign` that is structured as follows:

```
$ tree --filesfirst assign
assign
├── client.py
├── server.py
├── hans
│   └── Assignment_v1.0.pdf
```

```
├── server
│   └── credentials.txt
├── vader
│   ├── BitTrickle.mp4
│   └── rfc768.txt
└── yoda

5 directories, 6 files
```

This directory contains our client and server code, and four sub-directories. The `server` sub-directory will be the working directory of the server and contains the credentials file. The `hans`, `vader`, and `yoda` sub-directories will be the working directories of the three clients we will launch, each initially containing zero or more files for the user to share. The server and all clients will be executed in separate terminals, each within their own working directory. The server will be executed first, before any clients.

At the end of the sample interactions the directory `assign` will be structured as follows:

```
$ tree --filesfirst assign
assign
├── client.py
├── server.py
├── hans
│   ├── Assignment_v1.0.pdf
│   └── BitTrickle.mp4
├── server
│   └── credentials.txt
├── vader
│   ├── Assignment_v1.0.pdf
│   ├── BitTrickle.mp4
│   └── rfc768.txt
└── yoda
    ├── Assignment_v1.0.pdf
    ├── BitTrickle.mp4
    └── rfc768.txt

5 directories, 11 files
```

We can confirm that the downloaded files are exact duplicates using `diff`:

```
$ cd assign
$ diff -s hans/Assignment_v1.0.pdf vader/Assignment_v1.0.pdf
Files hans/Assignment_v1.0.pdf and vader/Assignment_v1.0.pdf are identical
$ diff -s hans/Assignment_v1.0.pdf yoda/Assignment_v1.0.pdf
Files hans/Assignment_v1.0.pdf and yoda/Assignment_v1.0.pdf are identical
$ diff -s hans/BitTrickle.mp4 yoda/BitTrickle.mp4
Files hans/BitTrickle.mp4 and yoda/BitTrickle.mp4 are identical
$ diff -s hans/BitTrickle.mp4 vader/BitTrickle.mp4
Files hans/BitTrickle.mp4 and vader/BitTrickle.mp4 are identical
$ diff -s vader/rfc768.txt yoda/rfc768.txt
Files vader/rfc768.txt and yoda/rfc768.txt are identical
```

#	server terminal	hans terminal	vader terminal	yoda terminal
1	$ python3 ../server.py 63155			
2		$ python3 ../client.py 63155		
3		Enter username: hans		
4	09:11:23.445: 57820: Received AUTH from hans	Enter password: solo*falcon		
5	09:11:23.445: 57820: Sent ERR to hans	Authentication failed. Please try again.		
6		Enter username: hans		
7	09:11:28.049: 57820: Received AUTH from hans	Enter password: falcon*solo		
8	09:11:28.049: 57820: Sent OK to hans	Welcome to BitTrickle!		
9	09:11:30.055: 57820: Received HBT from hans	Available commands are: get, lap, lpf, pub, sch, unp, xit		
10	09:11:31.249: 57820: Received LAP from hans	> lap		
11	09:11:31.249: 57820: Sent OK to hans	No active peers		
12	09:11:32.058: 57820: Received HBT from hans			
13	09:11:33.854: 57820: Received LPF from hans	> lpf		
14	09:11:33.854: 57820: Sent OK to hans	No files published		
15	09:11:34.063: 57820: Received HBT from hans			
16	09:11:36.068: 57820: Received HBT from hans			
17	09:11:38.074: 57820: Received HBT from hans			
18	09:11:40.079: 57820: Received HBT from hans			
19	09:11:42.084: 57820: Received HBT from hans			
20	09:11:42.343: 57820: Received PUB from hans	> pub Assignment_v1.0.pdf		
21	09:11:42.343: 57820: Sent OK to hans	File published successfully		
22	09:11:44.089: 57820: Received HBT from hans			
23	09:11:46.094: 57820: Received HBT from hans			
24	09:11:48.095: 57820: Received HBT from hans			
25	09:11:48.886: 57820: Received PUB from hans	> pub Assignment_v1.0.pdf		
26	09:11:48.886: 57820: Sent OK to hans	File published successfully		
27	09:11:50.101: 57820: Received HBT from hans			
28	09:11:50.653: 57820: Received LPF from hans	> lpf		
29	09:11:50.653: 57820: Sent OK to hans	1 file published:		
30	09:11:52.106: 57820: Received HBT from hans	Assignment_v1.0.pdf		
31	09:11:54.111: 57820: Received HBT from hans			
32	09:11:56.116: 57820: Received HBT from hans			
33	09:11:58.121: 57820: Received HBT from hans			
34	09:11:59.933: 57820: Received UNP from hans	> unp Assignment_v1.0.pdf		
35	09:11:59.933: 57820: Sent OK to hans	File unpublished successfully		
36	09:12:00.124: 57820: Received HBT from hans			
37	09:12:02.126: 57820: Received HBT from hans			
38	09:12:04.130: 57820: Received HBT from hans			
39	09:12:06.135: 57820: Received HBT from hans			

40	09:12:07.319: 57820: Received UNP from hans	> unp Assignment_v1.0.pdf		
41	09:12:07.319: 57820: Sent ERR to hans	File unpublication failed		
42	09:12:08.140: 57820: Received HBT from hans			
43	09:12:09.181: 57820: Received LPF from hans	> lpf		
44	09:12:09.181: 57820: Sent OK to hans	No files published		
45	09:12:10.145: 57820: Received HBT from hans			
46	09:12:12.149: 57820: Received HBT from hans			
47	09:12:14.154: 57820: Received HBT from hans			
48	09:12:16.159: 57820: Received HBT from hans			
49	09:12:18.164: 57820: Received HBT from hans			
50	09:12:18.320: 57820: Received PUB from hans	> pub Assignment_v1.0.pdf		
51	09:12:18.321: 57820: Sent OK to hans	File published successfully		
52	09:12:20.169: 57820: Received HBT from hans			
53	09:12:20.964: 57820: Received LPF from hans	> lpf		
54	09:12:20.964: 57820: Sent OK to hans	1 file published:		
55	09:12:22.174: 57820: Received HBT from hans	Assignment_v1.0.pdf		
56	09:12:24.177: 57820: Received HBT from hans			
57	...	...	...	...
58	09:12:36.208: 57820: Received HBT from hans			
59	09:12:38.213: 57820: Received HBT from hans		$ python3 ../client.py 63155	
60	09:12:40.218: 57820: Received HBT from hans		Enter username: hans	
61	09:12:40.493: 53589: Received AUTH from hans		Enter password: falcon*solo	
62	09:12:40.493: 53589: Sent ERR to hans		Authentication failed. Please try again.	
63	09:12:42.224: 57820: Received HBT from hans			
64	09:12:44.229: 57820: Received HBT from hans			
65	09:12:46.234: 57820: Received HBT from hans		Enter username: vader	
66	09:12:47.548: 53589: Received AUTH from vader		Enter password: sithlord**	
67	09:12:47.549: 53589: Sent OK to vader		Welcome to BitTrickle!	
68	09:12:48.239: 57820: Received HBT from hans		Available commands are: get, lap, lpf, pub, sch, unp, xit	
69	09:12:49.554: 53589: Received HBT from vader			
70	09:12:49.861: 53589: Received LAP from vader		> lap	
71	09:12:49.861: 53589: Sent OK to vader		1 active peer:	
72	09:12:50.244: 57820: Received HBT from hans		hans	
73	09:12:51.559: 53589: Received HBT from vader			
74	...	...	...	...
75	09:13:00.270: 57820: Received HBT from hans			
76	09:13:01.014: 53589: Received PUB from vader		> pub BitTrickle.mp4	
77	09:13:01.014: 53589: Sent OK to vader		File published successfully	
78	09:13:01.585: 53589: Received HBT from vader			

79	...	...	...	...
80	09:13:10.296: 57820: Received HBT from hans			
81	09:13:11.245: 53589: Received PUB from vader		> pub rfc768.txt	
82	09:13:11.245: 53589: Sent OK to vader		File published successfully	
83	09:13:11.611: 53589: Received HBT from vader			
84	...	...	...	...
85	09:13:14.307: 57820: Received HBT from hans			
86	09:13:15.024: 53589: Received SCH from vader		> sch pdf	
87	09:13:15.024: 53589: Sent OK to vader		1 file found:	
88	09:13:15.621: 53589: Received HBT from vader		Assignment_v1.0.pdf	
89	09:13:16.312: 57820: Received HBT from hans			
90	...	...	...	...
91	09:13:22.327: 57820: Received HBT from hans			
92	09:13:23.142: 53589: Received GET from vader		> get assignment_v1.0.pdf	
93	09:13:23.142: 53589: Sent ERR to vader		File not found	
94	09:13:23.642: 53589: Received HBT from vader			
95	...	...	...	...
96	09:13:41.685: 53589: Received HBT from vader			
97	09:13:42.379: 57820: Received HBT from hans			$ python3 ../client.py 63155
98	09:13:43.690: 53589: Received HBT from vader			Enter username: yoda
99	09:13:43.758: 53966: Received AUTH from yoda			Enter password: wise@!man
100	09:13:43.758: 53966: Sent OK to yoda			Welcome to BitTrickle!
101	09:13:44.384: 57820: Received HBT from hans			Available commands are: get, lap, lpf, pub, sch, unp, xit
102	09:13:45.695: 53589: Received HBT from vader			
103	09:13:45.763: 53966: Received HBT from yoda			
104	09:13:45.824: 53966: Received LAP from yoda			> lap
105	09:13:45.824: 53966: Sent OK to yoda			2 active peers:
106	09:13:46.389: 57820: Received HBT from hans			hans
107	09:13:47.700: 53589: Received HBT from vader			vader
108	09:13:47.768: 53966: Received HBT from yoda			
109	09:13:48.394: 57820: Received HBT from hans			
110	09:13:48.830: 53966: Received SCH from yoda			> sch i
111	09:13:48.830: 53966: Sent OK to yoda			2 files found:
112	09:13:49.705: 53589: Received HBT from vader			Assignment_v1.0.pdf
113	09:13:49.773: 53966: Received HBT from yoda			BitTrickle.mp4
114	09:13:50.399: 57820: Received HBT from hans			
115	...	...	...	...
116	09:14:47.920: 53966: Received HBT from yoda			
117	09:14:48.402: 53589: Received GET from vader	> get BitTrickle.mp4	> get Assignment_v1.0.pdf	> get BitTrickle.mp4
118	09:14:48.402: 53589: Sent OK to vader			

119	09:14:48.402: 53966: Received GET from yoda			
120	09:14:48.402: 53966: Sent OK to yoda			
121	09:14:48.402: 57820: Received GET from hans			
122	09:14:48.402: 57820: Sent OK to hans			
123	09:14:48.539: 57820: Received HBT from hans			
124	09:14:49.848: 53589: Received HBT from vader		Assignment_v1.0.pdf downloaded successfully	
125	09:14:49.925: 53966: Received HBT from yoda			
126	09:14:50.544: 57820: Received HBT from hans			
127	09:14:51.853: 53589: Received HBT from vader			
128	09:14:51.930: 53966: Received HBT from yoda			
129	09:14:52.549: 57820: Received HBT from hans	BitTrickle.mp4 downloaded successfully		BitTrickle.mp4 downloaded successfully
130	09:14:53.858: 53589: Received HBT from vader			
131	...	...	...	...
132	09:15:18.615: 57820: Received HBT from hans			
133	09:15:19.750: 53589: Received PUB from vader		> pub Assignment_v1.0.pdf	
134	09:15:19.750: 53589: Sent OK to vader		File published successfully	
135	09:15:19.924: 53589: Received HBT from vader			
136	...	...	...	...
137	09:15:22.626: 57820: Received HBT from hans			
138	09:15:22.696: 53589: Received SCH from vader		> sch pdf	
139	09:15:22.696: 53589: Sent OK to vader		No files found	
140	09:15:23.934: 53589: Received HBT from vader			
141	...	...	...	...
142	09:15:42.676: 57820: Received HBT from hans			
143	09:15:43.801: 53966: Received GET from yoda			> get Assignment_v1.0.pdf
144	09:15:43.801: 53966: Sent OK to yoda			
145	09:15:43.985: 53589: Received HBT from vader			
146	09:15:44.059: 53966: Received HBT from yoda			
147	09:15:44.681: 57820: Received HBT from hans			
148	09:15:45.990: 53589: Received HBT from vader			
149	09:15:46.064: 53966: Received HBT from yoda			
150	09:15:46.687: 57820: Received HBT from hans			Assignment_v1.0.pdf downloaded successfully
151	09:15:47.995: 53589: Received HBT from vader			
152	...	...	...	...
153	09:15:54.707: 57820: Received HBT from hans			
154	09:15:56.016: 53589: Received HBT from vader	> xit		
155	09:15:56.087: 53966: Received HBT from yoda	Goodbye!		
156	09:15:58.021: 53589: Received HBT from vader			
157	09:15:58.092: 53966: Received HBT from yoda		> xit	
158	09:16:00.097: 53966: Received HBT from yoda		Goodbye!	

159	09:16:02.103: 53966: Received HBT from yoda			
160	...	...	...	...
161	09:16:08.118: 53966: Received HBT from yoda			
162	09:16:08.738: 53966: Received GET from yoda			> get rfc768.txt
163	09:16:08.738: 53966: Sent ERR to yoda			File not found
164	09:16:10.123: 53966: Received HBT from yoda			
165	...	...	...	...
166	09:16:30.174: 53966: Received HBT from yoda			
167	09:16:32.180: 53966: Received HBT from yoda		$ python3 ../client.py 63155	
168	09:16:34.185: 53966: Received HBT from yoda		Enter username: vader	
169	09:16:34.259: 54347: Received AUTH from vader		Enter password: sithlord**	
170	09:16:34.259: 54347: Sent OK to vader		Welcome to BitTrickle!	
171	09:16:36.190: 53966: Received HBT from yoda		Available commands are: get, lap, lpf, pub, sch, unp, xit	
172	09:16:36.264: 54347: Received HBT from vader			
173	09:16:36.708: 54347: Received LPF from vader		> lpf	
174	09:16:36.708: 54347: Sent OK to vader		3 files published:	
175	09:16:38.195: 53966: Received HBT from yoda		Assignment_v1.0.pdf	
176	09:16:38.270: 54347: Received HBT from vader		BitTrickle.mp4	
177	09:16:40.200: 53966: Received HBT from yoda		rfc768.txt	
178	09:16:40.275: 54347: Received HBT from vader			
179	...	...	...	...
180	09:16:42.280: 54347: Received HBT from vader			
181	09:16:42.631: 53966: Received SCH from yoda			> sch rfc
182	09:16:42.631: 53966: Sent OK to yoda			1 file found:
183	09:16:44.211: 53966: Received HBT from yoda			rfc768.txt
184	09:16:44.285: 54347: Received HBT from vader			
185	09:16:48.221: 53966: Received HBT from yoda			
186	09:16:48.296: 54347: Received HBT from vader			
187	09:16:49.936: 53966: Received GET from yoda			> get rfc768.txt
188	09:16:49.936: 53966: Sent OK to yoda			
189	09:16:50.226: 53966: Received HBT from yoda			rfc768.txt downloaded successfully
190	09:16:50.301: 54347: Received HBT from vader			
191	...	...	...	...
192	09:16:58.247: 53966: Received HBT from yoda		> xit	
193			Goodbye!	> xit
194				Goodbye!