

Faculty of Engineering and Computer Science Expectations of Originality

This form has been created to ensure that all students in the Faculty of Engineering and Computer Science comply with principles of academic integrity prior to submitting coursework to their instructors for evaluation: namely reports, assignments, lab reports and/or software. All students should become familiar with the University's Code of Conduct (Academic) located at http://web2.concordia.ca/Legal_Counsel/policies/english/AC/Code.html

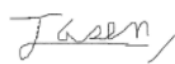


Please read the back of this document carefully before completing the section below. This form must be attached to the front of all coursework submitted to instructors in the Faculty of Engineering and Computer Science.

Course Number: COEN 366 **Instructor:** Ferhat Khendek

Type of Submission (Please check off responses to both a & b)

- a. ☒ Report ☐ Assignment ☐ Lab Report ☐ Software
 b. ☐ Individual submission ☒ Group Submission (All members of the team must sign below)

Having read both sides of this form, I certify that I/we have conformed to the Faculty's expectations of originality and standards of academic integrity.

Name:	ID:	Signature:	Date:
Jasen Ratnam	40094237		20/11/2021
Bahnan Danho	40044767		20/11/2021
Farhan-Nabil Alamgir	40045344		20/11/2021

Do Not Write in this Space – Reserved for Instructor

EXPECTATIONS OF ORIGINALITY & STANDARDS OF ACADEMIC INTEGRITY

ALL SUBMISSIONS must meet the following requirements:

1. The decision on whether a submission is a group or individual submission is determined by the instructor. Individual submissions are done alone and should not be identical to the submission made by any other student. In the case of group submissions, all individuals in the group must be listed on and must sign this form prior to its submission to the instructor.
 2. All individual and group submissions constitute original work by the individual(s) signing this form.
 3. Direct quotations make up a very small proportion of the text, i.e., not exceeding 5% of the word count.
 4. Material paraphrased from a source (e.g., print sources, multimedia sources, web-based sources, course notes or personal interviews) has been identified by a numerical reference citation.
 5. All of the sources consulted and/or included in the report have been listed in the Reference section of the document.
 6. All drawings, diagrams, photos, maps or other visual items derived from other sources have been identified by numerical reference citations in the caption.
 7. No part of the document has been submitted for any other course.
 8. Any exception to these requirements are indicated on an attached page for the instructor's review
- REPORTS and ASSIGNMENTS must also meet the following additional requirements:**

1. A report or assignment consists entirely of ideas, observations, information and conclusions composed by the student(s), except for statements contained within quotation marks and attributed to the best of the student's/students' knowledge to their proper source in footnotes or references.
2. An assignment may not use solutions to assignments of other past or present students/instructors of this course or of any other course.
3. The document has not been revised or edited by another student who is not an author.
4. For reports, the guidelines found in Form and Style, by Patrick MacDonagh and Jack Borden (Fourth Edition: May 2000, available at <http://www.encs.concordia.ca/scs/Forms/Form&Style.pdf>) have been used for this submission.

LAB REPORTS must also meet the following requirements:

1. The data in a lab report represents the results of the experimental work by the student(s), derived only from the experiment itself. There are no additions or modifications derived from any outside source.
2. In preparing and completing the attached lab report, the labs of other past or present students of this course or any other course have not been consulted, used, copied, paraphrased or relied upon in any manner whatsoever.

SOFTWARE must also meet the following requirements:

1. The software represents independent work of the student(s).
2. No other past or present student work (in this course or any other course) has been used in writing this software, except as explicitly documented.
3. The software consists entirely of code written by the undersigned, except for the use of functions and libraries in the public domain, all of which have been documented on an attached page.
4. No part of the software has been used in previous submissions except as identified in the documentation.
5. The documentation of the software includes a reference to any component that the student(s) did not write.
6. All of the sources consulted while writing this code are listed in the documentation.

Peer to Peer File System

Final Report

A Report

Presented to

The Department of Electrical & Computer Engineering

Concordia University

In Partial Fulfillment

of the Requirements

Of COEN 366

By

Jasen Ratnam

ID: 40094237

Bahnann Danho

ID:40044767

Farhan-Nabil Alamgir

ID:40045344

Professor: Ferhat Khendek

Concordia University

December 08, 2021

Table of Content

1. Introduction	1
2. Analysis	2
3. Design and Implementation	8
3.1 Client	9
3.1.1 Client to Server Communication	12
3.1.2 Client to Client Communication	13
3.2 Server	15
3.2.1 Server to client Communication	16
3.2.2 Backup	17
3.3 Multi-Threading	18
3.4 Data & Request Modelling	19
3.5 Handlers	21
3.5.1 Writer	21
3.5.2 Sender	21
3.5.3 Server Handler in the client	21
3.5.4 Client Handler in the server	22
3.5.5 TCP Server Handler in the client	22
4. Testing	23
5. Limitations	26
6. Contribution	27
Jasen Ratnam	27
Bahnan Danho	27
Farhan-Nabil Alamgir	27
7. Conclusion	29
Appendix	30

Table of Figures and Tables

Table 1: List of Assumptions	2
Figure 1: UML Diagram of the Clients	9
Figure 2: UML Diagram of the Server	15
Figure 3: UML Diagram of Data Model	19
Table 2: Test Cases	23

1. Introduction

The Peer to Peer File System (P2FS) uses a server-client based architecture that communicates over UDP and a client-client based architecture that communicates over TCP. The system consists of one server with a capacity to handle multiple clients via UDP and the clients are capable of communicating with each other via TCP to download files from each other. The system allows the clients to register to the server, and publish files it has available to the other clients via the server. The server is used to store all the information about the registered clients, that is their available files along with how they can be reached via TCP and UDP and share them to other clients. As required in the project description, initially, the client sends a register request to the server and the server responds with a confirmation or a denial. Clients can update their IP addresses and/or Socket ports. In addition, users can update their lists of files available to the server.

2. Analysis

The protocol design for the Peer to Peer File System as defined in the project description provides detailed implementation details needed to create a successful UDP based client-server and TCP based client-client communication. The protocol provides details for client to client as well as server to client communication. However, during the implementation we ran into situations that were ambiguously described in the project description and required certain assumptions to be made and implemented into our project. The assumptions we have made are described below.

Table 1: List of Assumptions

Case	Ambiguity	Assumption
Client receives a response from server with a RQ# that does not correspond to any of its (pending) requests?	Ignore the message? Let the user know?	Clients can get a response with RQ that does not correspond to any of its (pending) requests. Client will display an error message and ignore it.
Users Updating their information.	Do we allow users to manually change their IP address and port? When user tries to connect to a server from a new client	The user is able to manually change his ip address and port once he is connected from a new client IP and port of client is automatically gotten, and updated to an existing client. On update, the system will get the new IP and port of the new location. Clients' names cannot be changed.

User enter wrong server IP address / port number	<p>Should the users stay connected to the wrong server's addresses. Given an opportunity to enter the correct server's locations?</p> <p>Will they know that they are not connected to the server?</p>	<p>The client follows UDP and assumes the IP and port of the server is correct and sends all requests to it without knowing if it reached or not.</p> <p>If the wrong IP or port is entered, it can't be changed, the application will have to be restarted to enter a new server connection.</p>
Name of client	<p>Is the client name the host name?</p> <p>Is the name linked to the IP?</p> <p>Is the name unique?</p>	<p>The client name is assumed to be unique to the client. Can get back clients by using names. Once the client name is set, it cannot be changed. Name is not linked to the IP, it is a random given name</p>
What happens if the server crashes?	<p>What happens with the client requests?</p> <p>What happens when the server restarts?</p>	<p>Since we are using UDP, clients will not know that the server is disconnected, and will continue sending requests assuming it is still alive.</p> <p>The requests sent will be lost, while the clients assume it reaches the server.</p> <p>Whatever is in memory will be lost.</p> <p>When the server restarts, the server will reload all saved information from the backup.csv</p> <p>The backup will hold the UDP port of the server and information about all registered clients.</p>

What if a client tries to download a file from its own IP address	Should TCP connection to itself be done? Should file download be allowed?	Clients will not be allowed to download a file from themselves. Will check if the given IP is not their own IP before doing a TCP connection. TCP connection will not be done if it's the same IP
What happens during a TCP download.	Should the client be blocked? Should the client be able to send a download and download at the same time? How is the download done?	Clients should not be locked while doing transfers. Clients should be able to do multiple things at the same time using multithreading. Only files that have been published should be able to be downloaded. If it is not published, download will be denied Only one client can download from a given client at a time
Client tries to deregister itself but the message does not go through the server. Server is disconnected or cannot handle the message for some reason	Should the client not deregister?	Following UDP, the client will assume the message has been received by the server and since Deregister does not send a confirmation, it will assume it has been successfully deregistered.
Clients tries to send commands to server before registering	Should the client be able to send the commands? Should the server handle the commands? Should the client send the commands if it knows it is not registered?	Clients will be allowed to send commands that require registration before it is registered to the server. Server will ignore any commands from an unregistered client
Logging of activities in clients and server	Should be printed onto the console? Should it create log files?	We will print onto the console as well as create a log file

<p>Client has gotten information about client X from the server.</p> <p>X has deregistered.</p> <p>Client tries to do TCP connection to X and download a file after it has deregistered</p> <p>What if X has disconnected?</p>	<p>Should TCP connection and download be allowed with unregistered clients?</p> <p>What happens if a connection is attempted with a client that does not exist anymore?</p>	<p>Clients are allowed to attempt a TCP connection with any client they have an IP and TCP port number for, whether it is registered or not.</p> <p>If the targeted client does not exist or cannot connect to it for some reason, there will be an error message and client can try again or choose another command</p>
<p>What if a client tries to publish a file with a name that is already published?</p>	<p>Should duplicate filenames be allowed?</p> <p>Can a client have two files with the same name?</p> <p>Does publish replace all files already published.</p>	<p>Clients will be allowed to publish whatever filename they want.</p> <p>It can be duplicate filenames or files that do not exist on the client.</p> <p>A publish sent to the Server will replace the current list of files with the new list of files received. List of files sent by client will replace previous list sent.</p> <p>Server will assume that any given filename is valid.</p>
<p>Multiple clients on the same PC (same IP address)?</p>	<p>Can we have multiple clients on the same IP address?</p> <p>How would the server distinguish them?</p>	<p>Only one client is allowed to be registered per pc (IP)</p> <p>If another client registers with an IP already registered with another name, it will be rejected.</p> <p>It will be rejected even if you try to register with the same name.</p>

If server gets a unknown request type or if client gets a unknown response type	Should the server or client try to handle them? Should it ignore them?	If a request or response is received that is not one of the known types (register, deregister, update,...) it will be ignored
Sockets gets closed abruptly while downloading/uploading (Clients disconnect before completing download)/	Should the download be paused, cancelled?	TCP connections and data transfer will be lost. Error message will be shown.
What is 'Name' when a client publishes and removes files?	Is it the name of the client sending the publish request? Can the user choose the name of the client that publishes the files?	When the client chooses the publish option, the name of the registered client will be added to the message with a given filename automatically. Users cannot select a client name to send. Server will add a given filename on the client with a matching name.
What to do when a request denied is received from server at a client	Should Client try again? Should it ignore it?	When a client gets a request denied, it will post an error message to the user and allow the user to try again by themselves or do something else. No automatic retrial.
If clients tries to remove 1 file that is published and 1 file that is not published	Should it deny both files from being removed ? Should it delete the file that exists and deny the one that does not exist ?	Server will deny both files, and will not remove both files. All files in arraylist sent to be removed must be published on the server

3. Design and Implementation

The implementation of the project requirements is our design for the project. Following the Peer to Peer File System project guidelines, we have main Server and Client classes that are used to configure the UDP server and start the operation of the system. The Server class receives any requests from a client and sends it to a handler class, while the Client class is responsible to send any requests to the server. Both these classes use UDP to communicate in the LAN. In order to assist client and server classes to communicate we created other classes called handlers. On the client end, we use ServerHandler class to handle server responses and Sender class to send requests to the server via UDP. On the server end, we use ClientHandler class to handle requests and Sender class to send responses to the client via UDP. There is also a Writer class in the handler package for both server and clients that logs each transaction to a text file and the command line. The requests/responses messages have been constructed as java objects before being sent using bytestreams. Request is the parent class that is extended by several other subclasses to create all requests and responses. The design of individual components is discussed in detail below.

3.1 Client

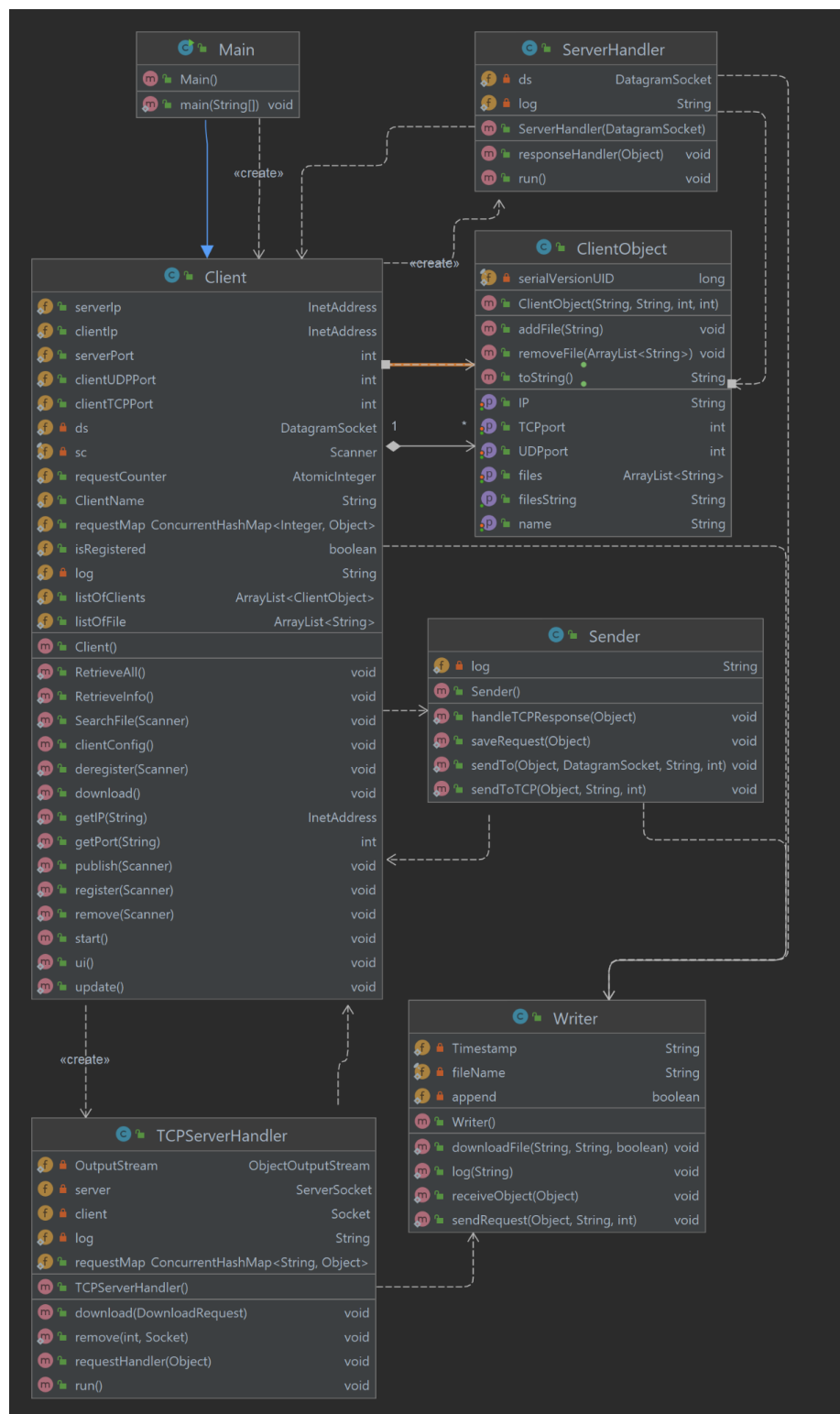


Figure 1: UML Diagram of the Clients

The Client class defines all the variables that are required by the client for set-up and communication. Upon Client start-up, the user is prompted to configure the server addresses and port numbers to establish server-client communication and then the user is prompted for a TCP and UDP port for itself. The Client class once configured starts two threads, one thread for the ServerHandler class and another for the TCPServerHandler class. The Client class also handles the logic to prompt users for any wanted commands and construct user requests such as Register, Update etc. based on the selection. These requests are then sent to the server using the Sender class.

The Sender class consists of a static sendTo method to send the user request to the server using the Server UDP information given at startup. For the download request from a Client, the static sendToTCP method is used to send the user download request to the target Client using TCP. The sendToTCP method establishes a connection to the target client with user prompted TCP port and IP address of target client and TCP port of client doing the request from its startup. Once the download request is sent, the Sender class will wait for a response from the client via the same TCP connection. Users can do other commands while waiting for the response. Multithreading is not used, therefore only one client can download at a time.

As stated previously, while clients can choose different requests to the server, two other threads are running in the background. The `ServerHandler` class is run in one thread, where it listens to responses from the server constantly. It receives any response to requests sent to the server by the user using UDP and handles them. The latter implements the `Runnable` interface that deserializes the server response and passes it to the `responseHandler` method. The latter handles the response based on the `requestType` and displays appropriate output to keep the user updated.

The `TCPServerHandler` class is run in the other thread. This thread listens to a download request from another user constantly. It receives a download request from another client using TCP and handles them, downloads the wanted file if it exists or sends an error message if it can't be downloaded. Similar to the UDP communication, the thread implements the `Runnable` interface that deserializes the client request and passes it to the `requestHandler` method. The latter handles the request based on the `requestType` and does the necessary processing and creates an appropriate response object. The response object is then sent to the client who sent the request using the same TCP connection and the connection is disconnected.

The `Writer` class is used throughout this process to log everything that happens in every class on the command line and on a text file in the same directory of the project. The `ClientObject` class is used to create objects with client information and display them when prompted.

3.1.1 Client to Server Communication

The project description required the Peer to Peer File System system to be composed of a server and a client that must communicate via UDP. To follow the requirements, we set-up a server to client communication using UDP. On the client side, this communication is done by using a datagram socket with the port number that the server is using. Whenever a client wants to communicate to the server, to send a request, it follows the following step.

The wanted request object, datagram socket along with the server port and IP address are sent to the sender class. Where the request object is serialised into a bit stream using an ObjectOutputStream. Then a datagram packet is created with the object and the necessary server information and finally, the packet is sent to the given server port and IP address via the datagram socket. Since this is a UDP protocol, the client sends the request to the given port and IP without verifying if the server is actually there. It will not wait for any response and will assume that the request has safely arrived at the server. The server can then respond to the request by sending a response to the client UDP port using the Sender class in the server.

3.1.2 Client to Client Communication

The project description also required the Peer to Peer File System system to allow client to client communication to allow file downloads between clients, this communication must be via TCP. In order to accomplish this we set-up a TCP server and a TCP client in every client in our system. The TCP communication between two clients is done in two parts.

All clients will start TCP server at startup using the TCPServerHandler class in a separate thread, this is used to constantly listen to any TCP requests from any other clients. The TCPServer will establish a connection to the other client and handle the request. The TCP server will then send a response to the request using the same connection. In our case, the request will always be a download object and the response may be a failed download object or the requested file being sent.

In addition to running the TCP server, any client can also send a TCP request to the TCP of any other client. The user will select the download command from the options which will then prompt the user for the target client IP address and TCP port. A download object will be created and sent to the target client using the sendToTCP method in the Sender class. A socket connection is established with the target client, if the target client cannot be found and/or the connection fails, an error message is sent and the request will not be sent. Once the connection is established, the request object is serialised and sent using ObjectOutputStream. The client then waits for a response from the TCPServer and finally handles that response. While waiting for a response, the client is free to do any other options. Once the response is received and handled, the connection will be closed. Our implementation only allows one client to download from another client at a time.

3.2 Server

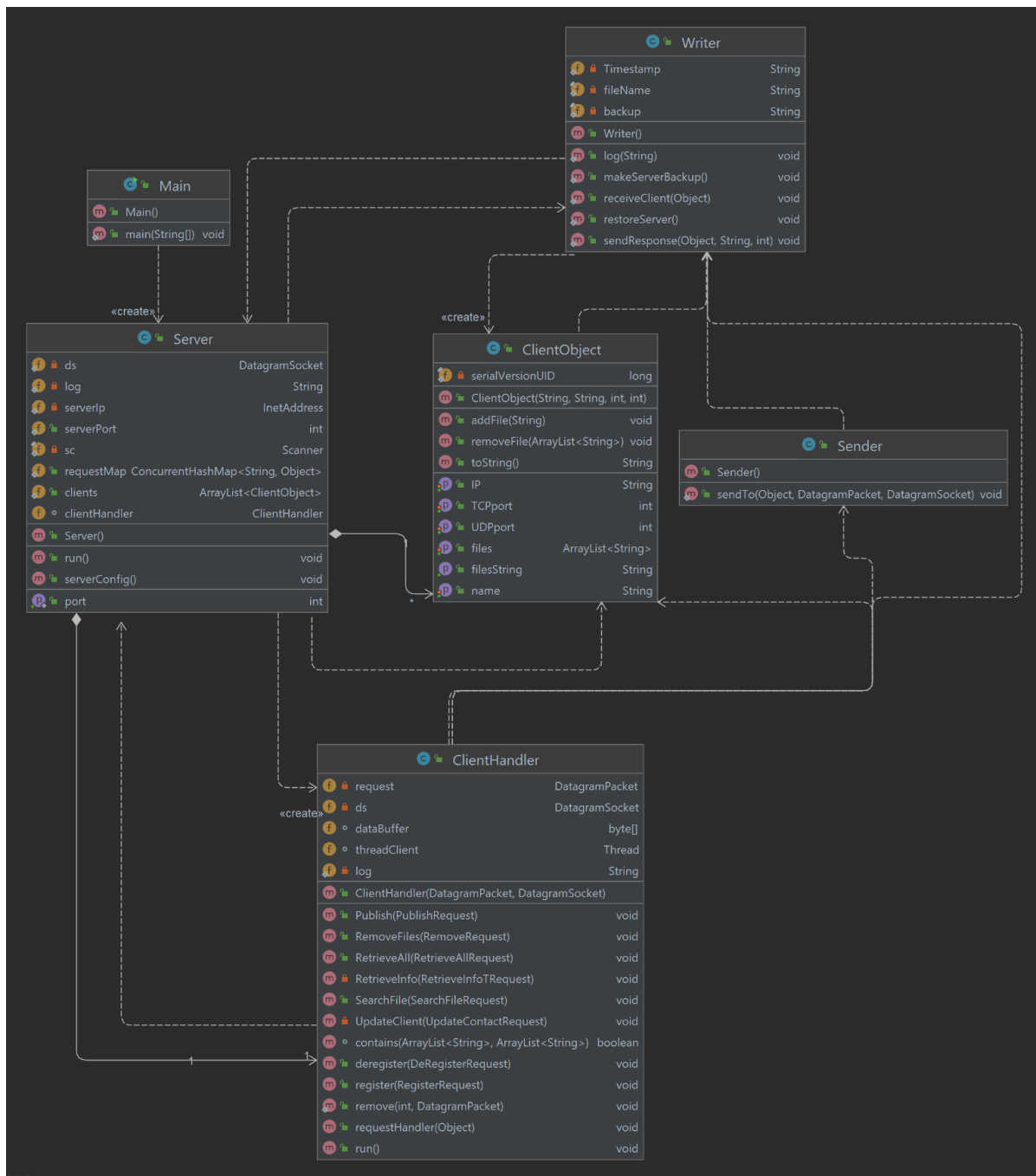


Figure 2: UML Diagram of the Server

The Server class defines all the variables that are necessary to configure the server for communication. When the server starts, the user will be prompted to enter the UDP port number of the server and then its IP address and port number will be displayed. When Clients are run, the server port number and IP address needs to be entered to be able to communicate with the Server. The server is designed in such a way that if the server crashes and you restart your server, it will be persistent. The server will look for and read a backup file and restore itself to its previous status. The port number and all registered clients will be restored. The role of the Server is to receive any request from the clients, handle them and send a response if needed.

The Sender class contains a static `sendTo` method which will send the response of the requests to the client using the UDP port of the client. The server class has a class called `ClientHandler`, a new thread is created with this class for every request the server receives. Every request from a client goes in its own thread.

The `ClientHandler` class is run in a thread for every request received, where it handles the request. It receives any requests sent by the user using UDP and handles them. The latter implements the `Runnable` interface that deserializes the client requests and passes it to the `requestHandler` method. The latter handles the request based on the `requestType` and displays appropriate output to keep the user updated. If a response is required, a response object is created and sent back to the client using the Sender class `sendTo` method via UDP.

3.2.1 Server to client Communication

The project description required the system to be made of a UDP communication protocol between the Server and the Client. On the server side, the communication is done the same way as the client: a datagram socket with the IP and UDP port number of the client. Whenever a server wants to communicate to the client, to send a response, it follows the following steps.

The wanted response object, datagram socket along with the client UDP port and IP address are sent to the sender class. Where the response object is serialised into a bit stream using an `ObjectOutputStream`. Then a datagram packet is created with the object and the necessary client information and finally, the packet is sent to the given client via the datagram socket.

3.2.2 Backup

The server contains a backup system which allows us to save the information that has been provided up until now. In the `Writer` class, there are two methods that make this backup process possible. The first method is the `makeServerBackup` that saves the UDP port and IP address of the server along with the registered users information into a *backup.csv* file. The second method used is the `restoreServer` which reads the backup the CSV files and restores the Server to its previous status. This allows the Server to be persistent. If the server crashes for some reason and needs to be restarted, it will read the *backup.csv* file and restore itself to its previous state. The server's port number and information about all registered clients will be restored.

3.3 Multi-Threading

Multi-Threading is adopted in our design by both client and server. Server is required to listen and respond to multiple client requests. It also needs to handle the UI requests simultaneously. These requirements can be handled in a multi-threaded environment. Therefore, in our implementation the server spawns a new thread for every new incoming client request and passes the data to ClientHandler class to be handled. The ClientHandler implements the runnable interface and processes the request thread until completion. Similarly, the client code was required to make server requests and simultaneously listen to server responses. However, our client implementation spawns one thread for server responses at startup and all responses will be received and passed to the ServerHandler that implements the runnable interface. Since there is only one thread to handle all responses from the Server in the Client, only 1 response can be handled at a time. But clients can send requests while handling responses since they are both in different threads. There is also a separate thread on the client side that runs a TCP server using the TCPServerHandler, this thread handles all requests received from another client. Similar to the ServerHandler this only uses one thread for all requests. Hence, clients can only handle one download request at a time.

3.4 Data & Request Modelling

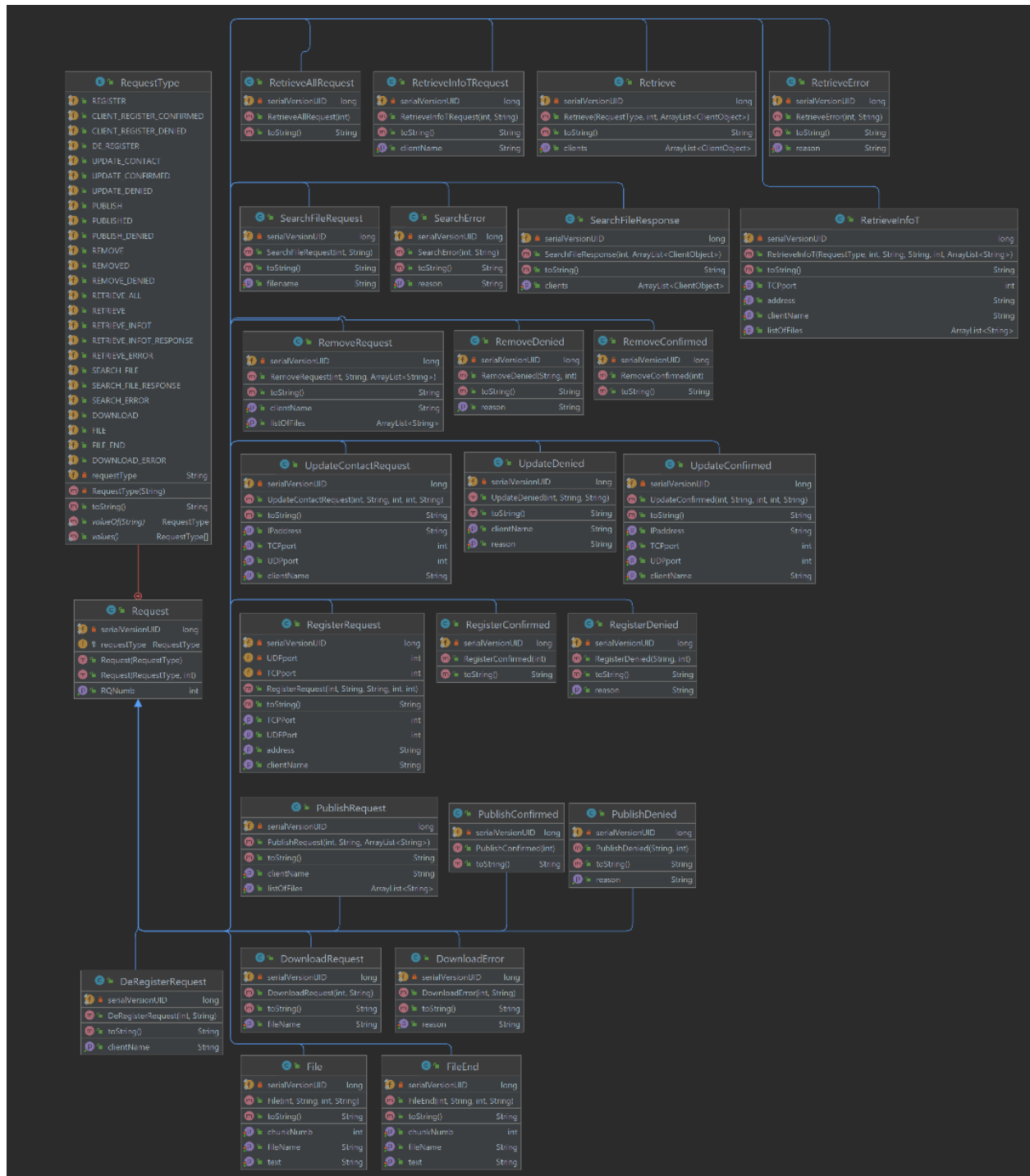


Figure 3: UML Diagram of Data Model

The UML diagram represents the inheritance design adopted to construct the Requests and responses used for communications between Clients and the Server. The Request class is the parent class of all requests and responses. The requests and responses consist of variables that are given for the request/response in the project requirements. The requestType is an enum that is injected when a particular request object is instantiated. It is used to make sure the same spelling and name is used for the request/response everywhere. These objects are serialised before being sent to a server or a client. The requests and responses are kept track of by using the request number ID in a hashmap on both client and server sides. The request ID are removed from the client side when a response is received and from the server side when it sends a response.

3.5 Handlers

3.5.1 Writer

This class is used to log in a text file and the command line everything that happens in the system. It logs any messages received or sent in addition to any log message we want. It is used to keep track of communications with the user entity by logging the results into a text file. This is common in both Server and Client code. In the Client side, the Writer class also is responsible to write the file downloaded from another client.

3.5.2 Sender

The Sender class is used in the Clients to send user requests to the server and is used in the Server to send responses to the Client. The class defines a static `sendTo` method that needs to be passed the object as a Request/Response. It is then sent alongside the socket number to be used to send the request/response. The object is converted to a bytes stream, included in the datagram packet and sent to the server or client a UDP socket. In addition, the Sender class in the Client also has a `sendToTCP` method that is used to send objects to other clients using TCP and a method to handle any responses from that TCP request.

3.5.3 Server Handler in the client

The server handler on the client side is used to constantly listen to any message from the Server through the client's UDP port. This helper class implements a runnable interface and handles the responses to the requests of the client. It constitutes the logic to qualify each response object received by the client. A Message is displayed to the user based on the response.

3.5.4 Client Handler in the server

The client handler on the server side is used to constantly listen to any message from any Client through the server's UDP port. It is also a helper class that implements a runnable interface and handles the requests received from clients. It constitutes the logic to qualify each request sent by a client, process it and send an appropriate response object back to the client.

3.5.5 TCP Server Handler in the client

The TCP server handler on the client side is used to constantly listen to any message from any other Client through the client's TCP port. It is also a helper class that implements a runnable interface and handles any TCP requests received from other clients. It constitutes the logic to qualify each request sent by a client, process it and send an appropriate response object back to the client. In our case, this will always be a download request and a download fail or a file transfer response.

4. Testing

Table 2: Test Cases

Pre-condition	Steps	Expected Results	Actual Results	Outcome
Start up	User sends a registration request to the server using a unique name	Client gets registered with a name.	Client is registered	-pass test
	Wrong Server information	Client will assume its the right information and send requests to given information	Clients sends request to given server information	-pass test
-User is registered	User sends a registration request	Client will get an error message saying they are already registered	Error message shown in logs	-pass test
	Server crash	Clients can keep sending requests but there won't be any responses. At restart the server will read CSV and restore itself to its previous condition.	Server restarted and restored successfully. Clients requests ignored while server is down.	-pass test
	User sends a deregister request	Clients assume a request has been sent and received and assume it is deregistered. Server removes the client from its list of registered clients	Client is deregistered	-pass test

	Registered user Connects from a different client User sends an Update location request from a different client	Registered users information is changed to new client location in Server	Client location updated	-pass test
	Multiple TCP downloads at once	Will not be allowed, since we are not using multi-threading to maximum potential	Multiple TCP downloads do not happen. Error message shown to failed downloads.	-pass test
	Client attempts download from itself	Client side check will not allow to send a download request from itself	Error message shown, download not allowed	-pass test
-User not registered	User sends a registration request	Client will be registered if given name and IP are unique	Client registered	-pass test
	User sends a registration request to the server using an already registered name	Registration will be denied with a error message displayed on log	Client not registered, error message shown.	-pass test
	User sends a deregister request	Users are allowed to send deregister commands to deregister any name. Server will deregister a client with matching name	User can deregister someone else, if given name does not exist request is ignored	-pass test
	Server Crash	Does not affect a non-registered client. Since its information is not saved yet. A registration request will be lost	Server restored succesfully	-pass test

	Update client	Registered users' information is changed to a new client location in Server. Client becomes registered	Client location updated.	-pass test
--	---------------	---	--------------------------	------------

5. Limitations

- Our client implementation can handle only one active user at one time for download.
 - Only one client can do a TCP connection to another at a time
- There is redundancy in our code design
 - Both client and server code have the same object copy pasted for requests and responses
- Timers are not used to time out requests and responses
- Multithreading is not used to its maximum potential, not allowing multiple TCP downloads and Server responses to be handled at once.

6. Contribution

Jasen Ratnam

Over the course of this project, I was responsible for creating the server backup using a CSV file. I also took part in the implementation of the server to client and client to client communication. This includes the registration and De-registration of clients to the server, the mobility of the clients and the File transfer between peers using TCP. In addition, I developed the Writer handler on both client and server and helped my colleagues to implement the other functionalities of the project.

Bahnan Danho

In this project, I was responsible for the implementation of the information retrieving and file searching functions. This includes the client request of retrieving all the clients, retrieving a specific client or searching for a specific file on the server. The work was done both from the client side as in the server side. I have also collaborated on the file publishing functions with the help of Nabil and Jasen in order to make our project a success. My colleagues were also there to help me if any problem was encountered.

Farhan-Nabil Alamgir

During the development of this project, I have worked on publishing file related information alongside Bahnan, which consists of publishing and retrieving available files and where to download them from. I have performed the work for both the client and server side alongside my teammates. I also worked on the mobility, which was to let the client update their contact information alongside Jasen. Overall throughout the project, we communicated a lot and reviewed each other's code to avoid any fatalities. I also made sure we were having proper versioning and source control throughout the project.

7. Conclusion

The Peer to Peer File System is implemented using a client-server UDP and client-client TCP communications. All the requests and responses objects are serialized for easier transportation. In fact, a server runs on the original thread from the main class that implements runnable. In addition, each request from a client is processed on a new thread. On the other hand, each client uses only one thread. The Server keeps a backup of itself using a CSV file . In case of a server crash, the CSV file is used to access the clients information and the Server information. The latter ensures the portability and reliability of the entire system.

Appendix

How to run the system

- Client:
 - Open command prompt
 - Navigate to the folder where Client.jar is located
 - Run the program using the following command:

```
java -jar Client.jar
```

- Server:
 - Open command prompt
 - Navigate to the folder where Server.jar is located
 - Run the program using the following command:

```
java -jar Server.jar
```