# CS 6650 Final Project Report

## Team Members

- Shubham Atul Parulekar
- Shashwat Mehul Rathod
- Stephen Okeleke

## Project Objectives

Our project is a distributed user document storage. The users of the project can sign up and login with this system. Once a user is logged in they can upload files, modify currently uploaded files and see files they have already uploaded.

The project uses multiple servers to serve all the above-stated operations for users. Servers are expected to be geographically separated and ideally, the user contacts the server which is nearest to the user so as to get the data as quickly as possible. For this, the user data that is login information is replicated amongst all the servers. This is viable because user profile data occupies relatively less storage. However, the data that the user uploads can be large. Due to this reason uploaded data or user content is sharded and kept in the server on which the content was originally uploaded. This also ensures that users are able to access their data quickly because of the geographical proximity to the server. If the user relocates to a different location, let's say a different country, then the user's data won't be found on the server nearest to it. In this case, the server finds the data on the older server, get the data on its own machine and then start serving the client. Although this is a heavy operation, the number of times this occurs won't be that large.

All the servers communicate with a coordinator. Whenever a server starts up, it produces a message on a message queue to inform the coordinator that the server has started at a particular location. The coordinator then handles any talks between two servers.
When a new server starts and it connects to the coordinator, the coordinator fetches the user data from one of the existing servers and sends it to the new server. This is done so that all servers will have the replicated user data at startup. A new server need not have the sharded data, as on the first call for each user that sharded data will be brought back to the server.

Though not part of the project, data that has been sharded can be replicated if multiple servers are being hosted for a single geographical location. If this is done then the sharded data from a geographically similar server also needs to be copied into a newly created server with the already copied user data.

## Overview

As explained above, the objective of this project is to build a scalable fault-tolerant distributed document storage system. To achieve this objective, we needed a way to deploy and synchronize the application on multiple computers to achieve horizontal scaling, in the event that we have to process a large number of concurrent requests. This objective brought about the need to achieve **Interprocess Communication** between the nodes of our distributed system. For now we will focus on Client-Server communication, and discuss how we achieved communication between our Coordinator and Server later in this report. We considered various forms of communication that we learned in the material such as sending information via sockets with the TCP or UDP protocol, Remote Procedure Calls (RPC), and Remote Method Invocation (RMI). We decided to use RMI for several reasons. Before stating the reasons, we would like to briefly go over the meaning of RMI as we learned from the course. To understand RMI, it is paramount to speak about RPC. In RPC, procedures in processes on remote nodes can be executed as if they lived in our local address space. RMI is built on the same foundations as RPC but in a world of distributed objects. RMI follows Java's object oriented paradigm, and lets us ship objects from Client to Server with minimal fuss. For this reason, we can define behavior very easily by creating and implementing interfaces for a specific purpose. RMI also lets us take advantage of Java's built in security mechanisms. Finally, a very important feature is that RMI is multi-threaded which allows us to exploit threads to process concurrent client requests efficiently.

We could have used this same method to achieve communication between our Server and our Coordinator, but we wanted to decouple those two services in such a way that they could communicate without necessarily knowing anything about each other. When we came across this problem, it became apparent that it was identical to a topic we had learned in the course called **Indirect Communication.** Indirect communication is done through a third party, allowing a high degree of decoupling between senders and receivers. This can be categorized as we learned, into two methods. Space uncoupling and time uncoupling. In space uncoupling, senders do not need to be aware of who they are sending to, and receivers do not need to be aware of who they are receiving from. In time uncoupling, senders and receivers do not need to be online simultaneously. There are multiple ways we could have achieved indirect communication including group communication, publish-subscribe systems, distributed shared memory, tuple spaces and message queues. We went with **Message Queues** due to the ease of use to set up, as well as the fact that it fits perfectly with our use case. Through the use of a message queue, activemq to be precise, we set up a way for our Server and Coordinator to communicate such that they do not need to be aware of each other. This is useful because the Coordinator needs to know how many servers exist in the cluster. Whenever a new Server is added, all it has to do is send a message to a predefined topic informing the Coordinator that it is now in the cluster. The same is done when a Server leaves the cluster.

To achieve scalability and high performance, we decided to implement **Replication** enabled by **Distributed Transactions.** Replication provides us with enhanced performance, high availability and fault tolerance. Documents stored by users are replicated across all our server instances. When a server fails, the user is still able to access with the most recent copies of the documents they uploaded, which proves that the system is fault tolerant to a degree. An important requirement of replicated data is consistency. Operations performed on one instance must be performed on all instances.

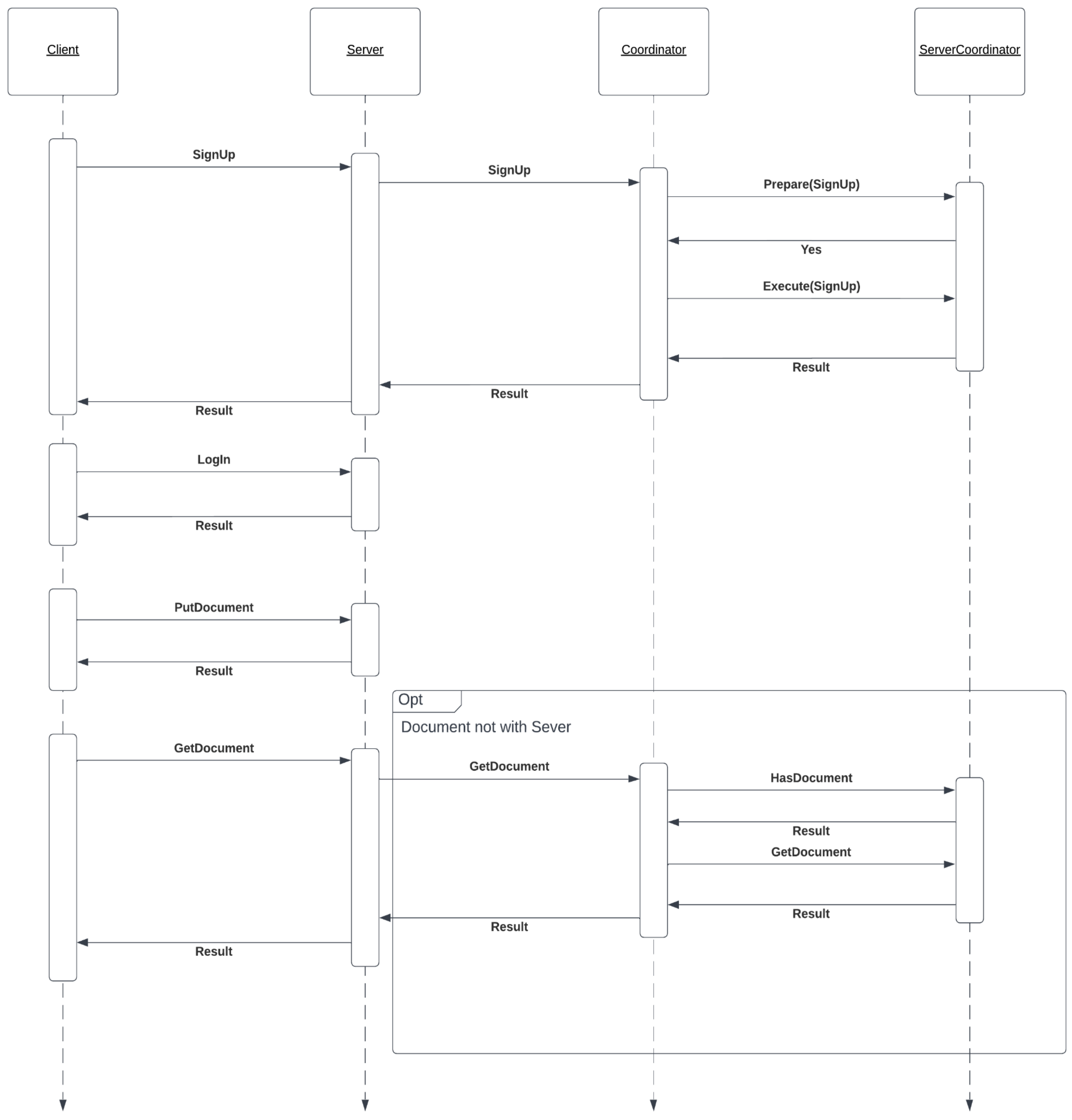
To achieve this consistency, we utilized distributed transactions which we learned in the course. Transactions allow us to process client requests in an atomic nature. There are a number of important factors to consider when dealing with atomicity including : All or Nothing factor, which means that either all operations are completed successfully or the operation must have no effect in the presence of server crashes (such as a get request which doe not modify any data) and the Isolation factor, which means that transactions are performed with interference from other transactions. As we learned from the course ACID is a good mnemonic for remembering these factors. It means Atomicity, Consistency, Isolation, and Durability. To be more specific, we used an atomic commit protocol known as **Two Phase Commit (2PC)** protocol to ensure that our transactions were ACID compliant. In 2PC, a server takes the role of a coordinator whose job is to coordinate the transaction among all servers. When a request is initiated, the coordinator sends a request to all servers asking them to prepare to commit. The servers store the details of the transaction in local storage and send a response of either commit or abort. The coordinator then initiates the second phase of 2PC and tells the servers to commit the transaction if and only if all servers responded with commit, or else it tells the servers to abort the transaction. In the event that a server fails in phase 1, when it recovers, it simply looks up the transaction details from the local server and then commits it pending instructions from the coordinator. This also increases the robustness and fault tolerance of our application.

## Technical Impressions

- As discussed above, our design involves replicating user credentials across each replica and storing the documents on the server they were uploaded to and then fetching them as required.
- For this reason, our design consists of 4 major components: the client-facing server, coordinator, coordinator-facing server, and the client.
- The client-facing server, called the ClientServer, exposes the major functionalities offered by our product like authentication and storage/retrieval of documents to the client.
- The client uses Java's RMI Registry to connect to a server and make remote calls to server methods in order to execute operations.
- The coordinator-facing server, called the CoordinatorServer, exposes the server's internal functionalities to the coordinator, allowing the coordinator to modify the server's state and perform polling operations. These include methods like prepare, execute, abort, hasDocument, etc.
- Each server maintains two datastores of its own: one for storing user credentials, and another one for saving documents.
- The datastore itself will support features to place a lock on resources that are about to be modified. A locked resource cannot be changed unless it is unlocked. This will help maintain the database consistency if multiple processes try to modify the same resource.
- The coordinator is an entity that has access to all the servers running at any point. The coordinator is responsible to ensure that any modifications that may affect shared data are replicated using 2PC across all the server replicas.

Following is the boot-up sequence of system:
- First, the coordinator boots up. A remote object for the coordinator is exported and bound to a name and port in an RMI registry.
- Next, the server boots up.
    - First, the server creates its own data stores (UserDao and LRU).
    - Next, it tries to find the coordinator's remote object in the registry.
    - After finding the coordinator, the server boots up the client-facing server module and exports the object to the registry so that any client can connect to the server.
    - Next, it initializes the coordinator-facing remote server object and asks the coordinator to add this server to its list of servers by publishing a new message onto the message queue.
    - Upon receiving the request, the coordinator finds the remote object for the CoordinatorServer and adds it to the list of servers it maintains.
    - At this point, the server asks the coordinate to synchronize the data so that the data in the newly created server is consistent with the data present with existing servers.
- The server is now ready to accept client requests.

The diagram below provides a high-level view of the sequence of execution of events for each action.



For events that deal with replicated data like user data:

- First, the client calls the required method on the server (e.g. SignUp)
- Since this method deals with the modification of data, the ClientServer calls the corresponding method on the coordinator (SignUp).
- The coordinator then asks all the CoordinatorServers connected to it if they are prepared to execute the operation.

- At this point, each server saves the operation locally.
- The Operation is then responsible to commit itself to the datastore by placing a lock on necessary resource.
- If all the servers are “prepared”, the coordinator asks the servers to execute the operation. This is the point where the states are actually modified.
- If any of the servers are not prepared or unresponsive, the coordinator asks the server to abort the operation at which point the operation is removed from the local memories of servers.
- When a server receives the instruction to execute an operation, the server then asks the Operation to modify the datastore after removing locks from resources.
- The result of this operation is then conveyed by the coordinator to the ClientServer and subsequently, to the client.

For events that deal with accessing documents:

- The client first calls the method on the ClientServer to get the requested document.
- If the client-facing server that the client is connected to has the requested data, it simply returns it to the client.
- If the server doesn’t have this data, it then asks the coordinator to fetch this data from any other servers that may have it.
- The coordinator then requests all the coordinator-facing servers if they have this data.
- If a server responds positively, the coordinator then calls the method on this server to fetch the relevant data.
- The coordinator then sends this data over to the client-facing server.
- Upon receiving this data, the server first saves a copy of this data with itself. And then before returning this data to the user, requests the coordinator to remove it from the original storage destination to avoid any data duplication. The coordinator then uses 2PC to safely remove the data.
- The server finally returns the relevant data to user. Since this data is now stored at the server that the client is directly connected with, any future reads for the same can be performed significantly faster.

## Project Setup

Overview-

The project uses log4j, java rmi and jms as dependencies which are added to the pom.xml file. Generate jars for the 2 drivers in `src\main\java\server\driver` and the client present in `src\main\java\client` and then move to the running instructions. Jars are already generated and are in the `res` folder.

Overview-

Go to the folder with the 3 jar files.

To run the coordinator-

```
java -jar Coordinator.jar -jp <jmsport> -p <rmiport>
```

Example

```
java -jar Coordinator.jar -jp 50001 -p 50000
```

To run the server-

```
java -jar Server.jar -p <selfport> -cp <coordinatorjmsport> -ch <coordinatorhost> -n <servername>
```

Example

```
java -jar Server.jar -p 50003 -cp 50001 -ch localhost -n ser1
```

To run the Client

```
java -jar Client.jar <port> <hostname>
```

Example

```
java -jar Client.jar 50003 localhost
```

Please use a port not being used by some other process.

## Demo video and github repository

There is a video that demonstrates the running of the project.
https://www.youtube.com/watch?v=1IKwztH2NtY

https://github.ccs.neu.edu/shalekar/bsds-proj
currently private to prevent any sort of plagiarism issues. Contact a team member if you want access.