CS6650 Final Project Report

File Collaborative Editing Tool

Yu Feng, Yifan Xu, Huichen Liu, Yuting Sun

Abstract

In this project, we implemented a collaborative text file editor from scratch using key concepts and algorithms of distributed systems.

In this file collaborative editing tool, team members can edit files on the local computer, and the updates will be synced among all servers when editing is done. Multiple users can make modifications to the different sections of the same file, changes will be concatenated and synced. In addition, this tool also supports user authentication. Users can register/login with username and password. For safety concerns, users can only edit files they have access to. To improve remote working efficiency, we designed a group chatting feature in this tool. When a file is being edited, all users who are working on the same file will enter a chat group where they can send and receive group messages.

AOD and Design Description

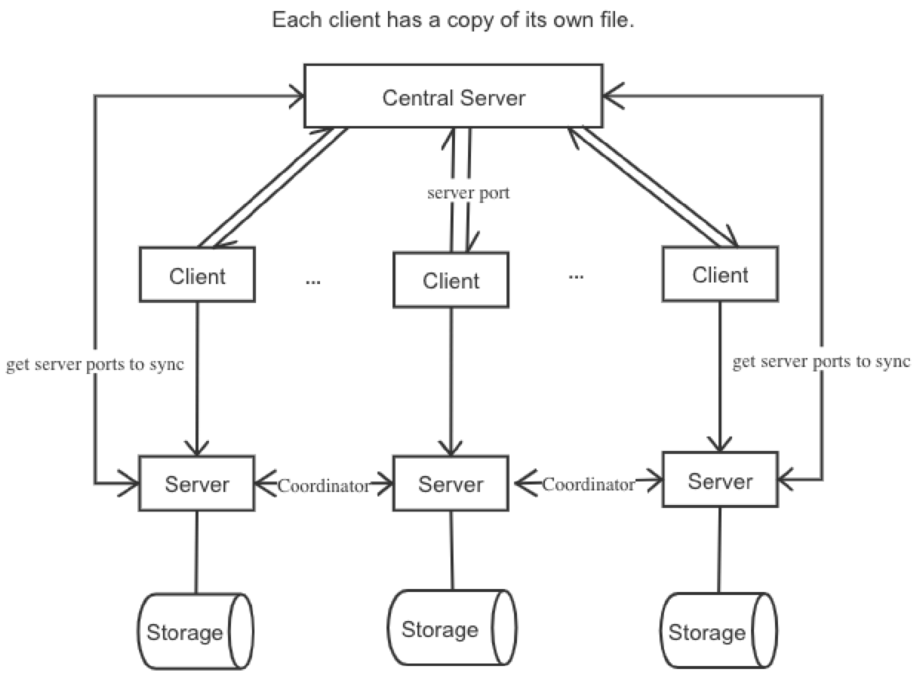

Figure 1. Architecture Overview Diagram

Main Classes and Function Overview

- **`CentralServer`**: Provides central service that supports adding/removing servers and clients, and other housekeeping tasks. It contains the information of a list of servers and their current status. Furthermore, it takes the responsibility to find servers, check servers' status, and provide register service for servers. We assume that the central server never fails.
- **`Client`**: Client constructs a connection with Central Server first. With a unique central server, the clients can be distributed to available servers at the same time. Client will leverage the port received from Central Server and connect to one of the servers.
- **`Server`**: After receiving the request from Client, Server requests Central Server a list of alive server ports, which will be used to conduct the synchronization / transaction among the server storage replicas.
- **Storage Classes**: Each server is in charge of its own replica. Server will take care of the global consistency among replicas by monitoring the common success of updating on directory replicas.

Package Structures

Table1. Package Descriptions in General

Package	**Description**	**Key Classes**
`server`	Implements classes and methods for central servers, replica servers and replica databases.	`CentralServer, Admin, Server`
`client`	Implements client classes and methods.	`Client`
`chat`	Implements classes and methods that enables users to chat.	`ChatManager, Sender, Receiver`
`model`	Implements helper classes of this project, including commit arguments, models for user and documents, etc.	`CommitParams, Documents, User, Results, etc`

Package `com.distributed.server`:

Class `CentralServer`: `Admin` can trigger `CentralServer` to add or remove servers, and other housekeeping tasks. A client can request `CentralServer` to be assigned to an arbitrary alive server. Assumption: the central server and admin never fails.

`CentralServer`:

1) Keep track of server status:
   Data Structure: `Map<Integer, Integer> serverStatus`
   Get server information: `getServerStatus(), getPeers()`
2) Method for assigning server to client : `assignAliveServerToClient()`
3) Method for restart replica server: `restartSlaveServer()`

`Admin`: Provides methods to kill and restart a server by remote calling methods of `killSlaveServer()` and `restartSlaveServer()` from Central Server

Class `Server`

1) Methods for user administration:
   `createUser(), login(User user), logout(User user)`

2) Methods for document administration:
   Create and share: `createDocument()`, `shareDoc()`
   Get document informations:
   `showSection()`, `showDocumentContent()`, `listOwnedDocs()`
   Edit documents: `edit()`, `editEnd()`

3) Methods for 2PC
   Coordinator methods: `prepare()`, `commitOrAbort()`
   Participant methods: `receivePrepare()`, `receiveCommit()`, `receiveAbort()`, `executeCommit()`

4) Methods for data recovery
   `recoverData()`, `helpRecoverData()`

Storage Classes:
- `aliveUserDatabase`: stores alive user information in memory.
- `userDatabase`: stores user information with .dat file.
- `documentDatabase`: stores document information with .dat file.

Package `com.distributed.client`:
Class `Client`:
`commandDispatchingLoop()`: for each user input, the client parses the argument and calls the corresponding remote method in Server.

Class `LocalSession`:
- Keeps track of the current user and document information for the current session.

Class `NotiClientRunnable`:
- A thread that runs separately with the client to receive and manage notifications from other clients.

Package `com.distributed.chat`:
Class `ChatManager`: Manages chat database of group chat among clients
Class `Sender`: Implements a sender that sends datagram messages via UDP to other clients in the same chat group.
Class `Receiver`: Implements a receiver class of the chat group.

Package `com.distributed.model`:
Models that related to database:
`User`, `Document`, `Section`, `BackupData`

Models that are related to 2PC:
`CommitParams`, `CommitEnum`, `Request`, `Results`

Models related to chat and notifications:
`Message`, `RemoteInputStreamUtils`

Technical Impression

Implementation Approach

Our system is implemented by Java. Clients communicate with servers using Remote Procedure Calls (RPC) with Java RMI. Client registry is implemented with Java RMI. Files are transported using RMI. Files between servers are transported with byte arrays. Files between server and client are transported using Java RMIIO library[1] that supports streaming large amounts of data using the RMI framework. Group chat is implemented with java.nio library. Chat messages are transported as datagrams using UDP protocol. We use TCP and UDP tools provided in java.net library to implement these features.

Server/Client Management

Client Registry
New client connects to the central server and receives a server port number to connect.
The client sends a connection request to the replica server and establishes a connection.

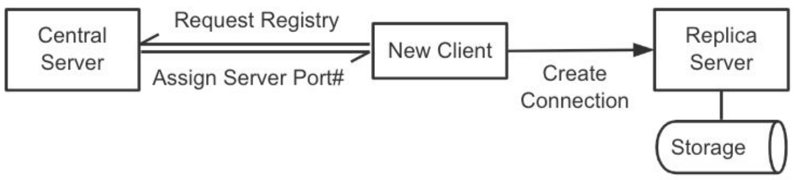

Figure 2. Client Registry

[1] https://openhms.sourceforge.io/rmiio/

Server Kill and Restart

To test fault tolerance of our system, we designed a super server `Admin` that can kill and restart a replica server by remote calling CentralServer methods `killSlaveServer()` and `restartSlaveServer()`.

2-Phase Commit with Fault Tolerance

To enable fault tolerance from the distributed transaction, we implemented a modified 2-phase commit. In this procedure, the coordinator will commit the transaction when it receives ACK from all live peers. If some server fails, the system will continue to work as long as a majority of servers are alive.

The modified 2PC procedure:

When a client sends a request that needs to update the storage, the server starts a 2PC process as a coordinator. The coordinator first gets a list of peer servers from the central server.

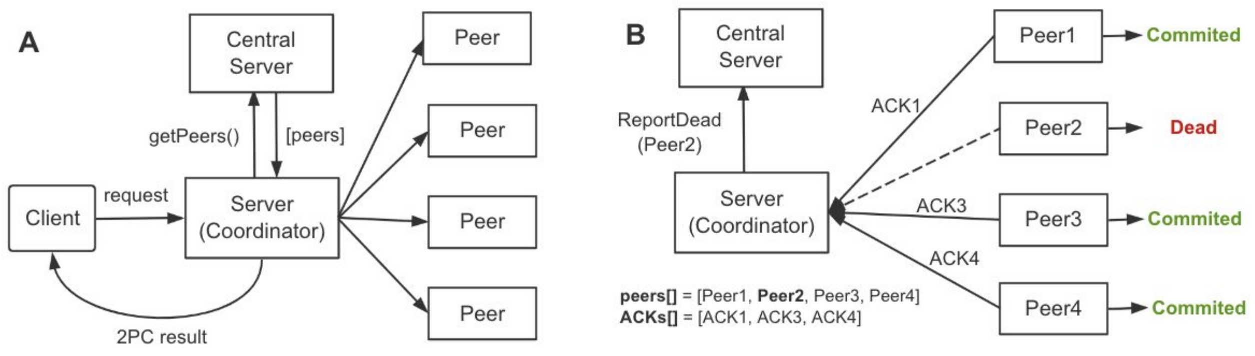

Figure 3. 2PC with fault tolerance. (A) Client sends a request to the server, the server starts the 2PC process as a coordinator and sends the 2PC result back to the client. (B) An example of fault tolerance. If Peer2 fails in the commit phase, other peers still commit the transaction. The coordinator can detect the failed peer and report to the central server.

1) Phase I: the coordinator sends prepare() to all peers and collects votes.
   If any peer votes ABORT, coordinator aborts the transaction.
   If all votes are PREPARED, the coordinator continues to the next phase.
   NOTE: the coordinator does not need to receive from every peer. Even if some peer is dead and fails to respond, the coordinator will continue as long as majority peers are alive and respond with PREPARED.

2) Phase II: the coordinator sends commit() or abort() to all peers based on the voting result of Phase I.
   If a peer receives COMMIT, it calls executeCommit() to update local storage and sends ACK back to the coordinator.
   NOTE: The coordinator saves the ACK from every peer in a hashmap. If a peer is dead, it would fail to send ACK to the coordinator. By comparing its ackMap with the peer list, the coordinator is able to detect a dead peer.

Database Update

When the transaction is committed, the storage database and files of the server is updated. The request types fall into two main categories based on whether they require to update document files.

Table2. Requests types that need to conduct 2-phase commit procedures:

Request Type	Data Updated	Request Names
0	AliveUserDatabase	Login / Logout
1	UserDatabase	Register / ShareDoc / GetNotifications
2	ChatManager	EditDoc / StopEdit
3	DocumentDatabase	CreateDoc / ShareDoc
4	DocumentFiles	EditDoc / StopEdit

As shown in the table above, request type 0-3 do not update doc files, while request type 4 update doc files. These two types follow different procedures when committing a transaction:

Requests without updating documents:
When a coordinator starts a 2PC, it stores the updated database in commitParam (but not update the local database).
When a server (coordinator or peer) commits a transaction, it updates its local database according to the database in commitParams.

Requests that updates documents:
When a coordinator starts a 2PC, it stores arguments such as (docName and fileStreams) in commitParam (not update documentDatabase directly).

When a server commits the transaction, it updates documentDatabase according to the arguments from commitParam.
To update doc files, the server first constructs a fileStream byte array according to the arguments from commitParam (docName, secName, filePath, etc). Then the server writes the fileStream to the files that need updating.

Data Recovery

When a server is restarted, the data can be recovered and remain consistent with other servers. The detailed steps are listed below:

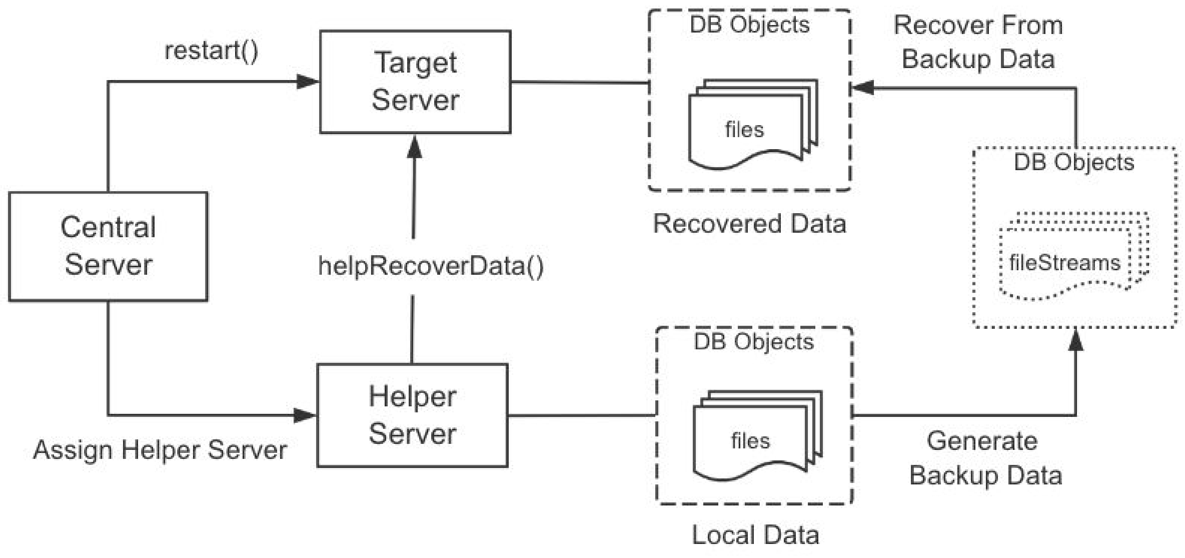

Figure 4. Server Restart and Data Recovery

1) Admin calls `restart(targetServer)` and the central server will restart the target server and assign a live server as Helper Server to help the target server recover data.
2) Helper server calls `helpRecoverData()`

a. The helper server backups its local data to backup data:
- Copies Database Objects (i.e. UserDB, DocDB, etc) to the backup data.
- Copies local files as fileStreams in byte arrays and put to the backup data.

b. The target server calls `recoverData(backupData)` to recover data from backup data:
- Copies Database Objects from backup data.
- Recovers all files from fileStreams in backup data.

Chat Management

This feature allows chatting among multiple users editing the same document. The chatting database is stored in class `ChatManager` with a hashmap that

`{key = docName : val = chatRoomAddress}.`

Chatting messages are transferred as datagrams via UDP.

- When a client is created , a `Receiver` role is created.
- When the client connects to a server, a `Sender` role is created
- When a client starts to edit a doc, the `ChatManager` adds the client to the chat room of the doc.
- The client can send messages to the chat room by calling `sendMessage().` The message is broadcast among all clients in the chatroom.
- The `Receiver` thread of each client runs separately from the client's main thread. The Receiver keeps receiving messages from the chatroom UDP channel and stores them in a `Message` list. When the `Client` enters `"receive"` to see message history, it obtains the `Message` list from `Receiver`. The `receiver` automatically clears the read messages by resetting the message list.

Key Algorithms

Distributed Mutual Exclusion

In this project, distributed mutual exclusion is required when multiple clients are trying to send requests to the servers and update the databases. Specifically, we implemented mutual exclusion using the tools provided in the java `Concurrency` API, which provides implicit locking via the `synchronized` keyword and various explicit locks specified by the `Lock` interface.

Distributed Transactions

Our system allows its concurrent users to perform transactions, while guaranteeing ACID properties. To ensure the consistency among all the replica servers and provide fault tolerance property for our system, we implemented a modified two-phase commit protocol for transaction commits. Details are discussed in the previous section: 2-Phase Commit with Fault Tolerance.

Fault-Tolerance

We designed fault tolerance for our project in two aspects:

1) The system still works even if some servers are down. Details are discussed in the previous section: 2-Phase Commit with Fault Tolerance.
2) When a server restarts from failure, it recovers data including users, docs and chat management from a live server in the system. Details are discussed in the previous section: Data Recovery.

Group Communication

In this project, we designed a feature that allows chatting among multiple users editing the same document. One user can send messages to others and all the users can check the whole chat log and send their own messages. We implemented this feature using group communication, details are discussed in the section: Chat Management.