Northeastern University
CS6650 Scalable Distributed Systems Fall 2022
Nisarg Patel - patel.nisargs@northeastern.edu

BookKeeper – Final Project

Introduction:

BookKeeper is a simple e-commerce distributed application. It is a platform that allows user to keep track of the books that they own all from a single application. Using BookKeeper, user can store the details of the books they have in their bookshelf, sell them or buy new books all from the same place. User needs to create an account to interact with the application.

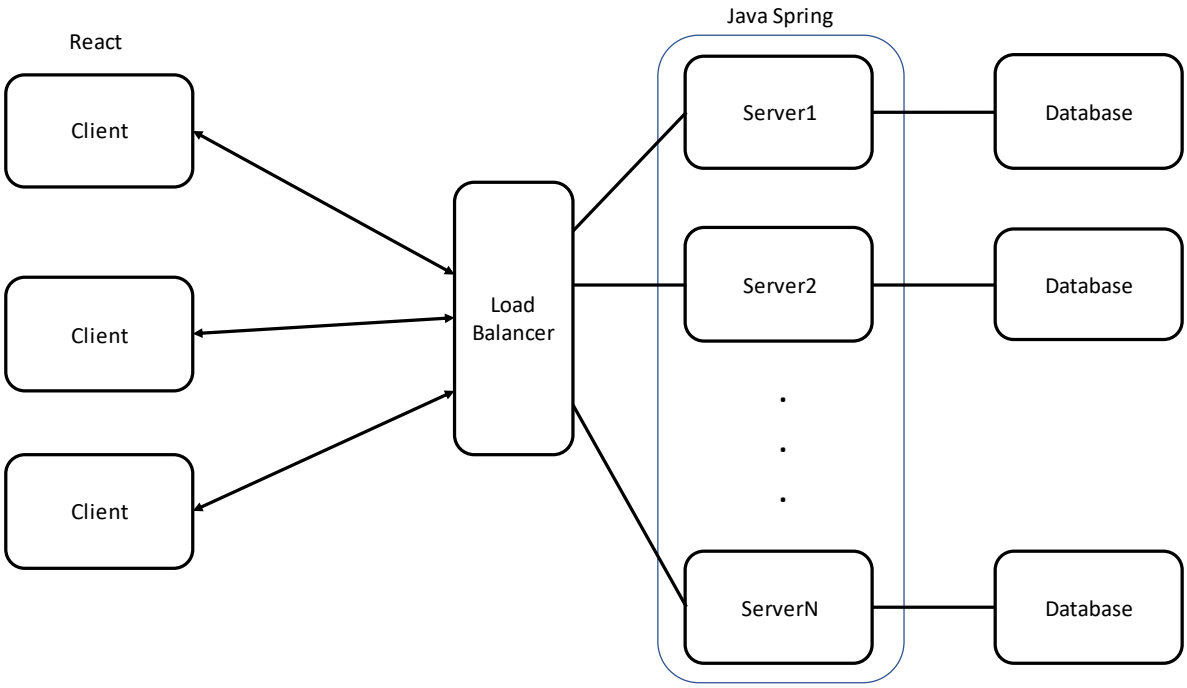
This project is focused mainly on making a distributed application, thus making the functionality of the application simple but sufficient to be completed in the required time.

Problem:

BookKeeper is a simple distributed application to store the information of the books that user own. User can have multiple shelves to store the book, but all the need to keep track of it is by adding the book to the application. It also acts as an e-commerce application, where books that other users put on sale can be bought by the user.

Architecture Overview and Design Description:

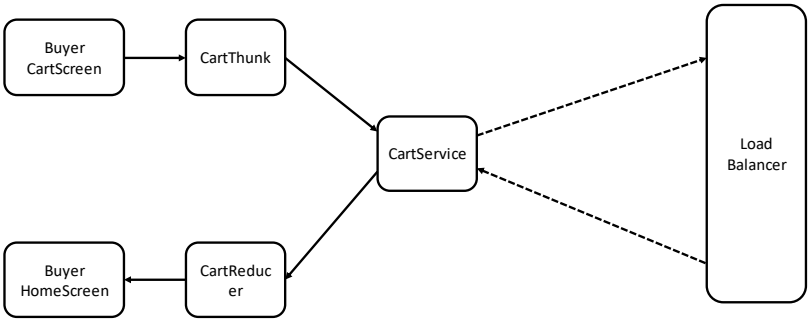
For this project, I used ReactJS to create a front-end application to interact with the clients. The client send a request to the load balancer that forwards that request to one of the many replicated servers. The request and response communication uses REST APIs. The servers are implemented in Spring and uses Spring JPA to store persistent data. The database is implemented using H2 Relational Database. The architectural overview of the implemented application is as follows:



Multiple users can interact with the application at the same time, and the state of the application is replicated among multiple servers.

Implementation Approach (Front End):

The font-end application is created using ReactJS, with dependencies on React-Router, React-Redux and reduxjs-toolkit for state management. The frontend consists of 3 main services: UserService, BookService and CartService. UserService is used to maintain cuser signup/login information. BookService handles all the book operations of a user and CartService handles the cart operations. The Services creates Thunks that is used to perform asynchronous operations with the load balancer and maintain states on the client. A simple BuyCart flow within the application is as follows:



Implementation Approach (Back End):

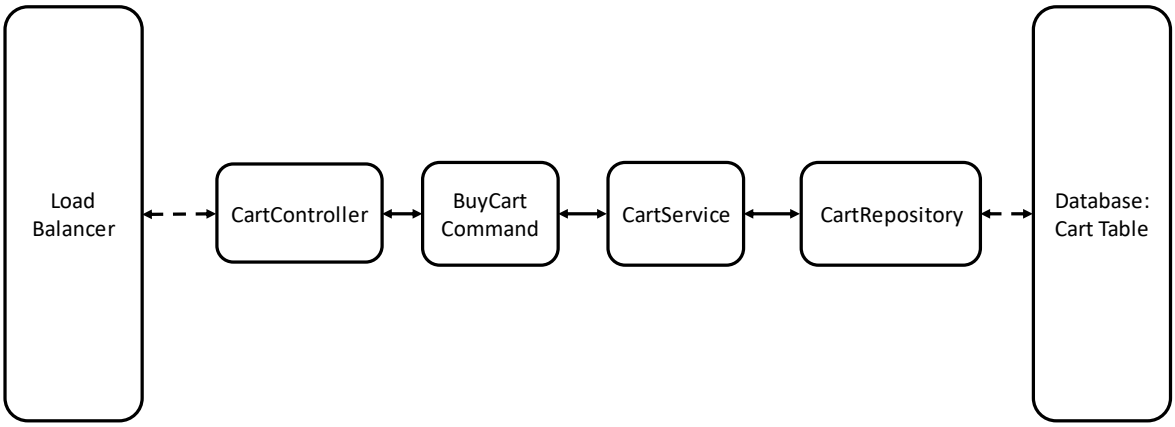
The servers are implement using Spring JPA, with database using Spring JDBC, H2 Database.

All the requests from client come to Load Balancer(Coordinator), which then redirects the request to one of the active servers controllers. The server than performs the operations and sends back the response. In between various distributed algorithms are performed to maintain the state of the system. These algorithms will be discussed in the next section.

The REST requests from the clients are handled by one of the server controllers specific to the request.

1) The UserController handles the requests of user login/signup and logout
2) The BookController handles the requests of a user adding a new book, deleting a book, updating a book, and search requests of the user.
3) The CartController handles the request of cart items of the user.

These controllers generate the specific command that performs operations within a service. These services interact with the repository, which is used to perform database operations. A flow of BuyCart command from the Cart Controller is as follows:



Distributed Algorithms:

To maintain the state of the application between servers, I implemented various distributed algorithms. The system contains a coordinator that handles various housekeeping tasks. Currently the coordinator is same server as load balancer, but it can be separated as per the needs. The coordinator takes requests from new servers that joins and stores it in the serverlist. Inter server communication is handled by ServerController. Each server has a server controller. Apart from this, the following distributed algorithms are implemented:

1) **Distributed Transactions (Two Phase Commit):**
   To perform transactions, I implemented a two-phase commit protocol, following the pseudocode provided in the textbook. I created a class of Transaction that stores the transaction id and the command of the transaction. The caller server performs a transaction, by calling performTransaction. During the voting phase, the caller servers sends a cancommit rest request. Upon response, the caller server checks if all server agreed to commit. If so, then it sends a docommit request to all the servers. Else it sends a doAbort request.

   Distributed transaction was required to maintain the state of the system among the different servers. Some Commands that uses Distributed Transactions are: Signup and Login.

2) **Paxos(Distributed Consensus and Fault Tolerance)**:
   Since the actual system can lead to server failures, the more practical algorithm to perform distributed consensus is Paxos. I used coordinator as the distinguished proposer. The server crashes are simulated by random failures of accept or prepare requests. While testing the application, it can be seen that the system reaches a consensus when less than half of the server fails.

   Paxos is a very important algorithm as it is used in practical distributed systems to perform the distributed consensus. Some commands that uses the Paxos algorithm are: addBook, deleteBook, buyCart, etc.

3) **Distributed Mutual Exclusion (Ricart-Agrawala algorithm):**
   Mutual exclusion is required to perform operations on a critical section. The updates on the database should occur in the order of the queries. For a distributed application, mutual exclusion can be carried out to maintain the state of the whole application. I implemented the Ricart-Agarwala algorithm for distributed mutual exclusion.

   Ricart-Agrawala mutual exclusion is performed on getBookListOfUser, searchBook, etc.

4) **Time and Clocks:**
   To maintain a timestamp and happened before, Vector Clocks are implemented to maintain the order of the commands. Vector clocks are used in the implementation of Ricart-Agrawala algorithm.

5) **Replicated Data Management:**
   Servers can crash and restart at any point of time, thus to make the server start from the actual state of the system, when the server sends a request to add itself to the list of servers, the coordinator sends a copy request to copy the state of the current system. This way any server can start at a random point. Data replication is also important to increase availability of the system.

Changes from Proposal:

I followed most of the design that I proposed during the project proposal. Some changes include the removal of creating microfrontends due to lack of time. Furthermore, I was not able to include the Lending service as it just made the frontend complicated which was not the purpose of the project. The actual distributed algorithms are implemented as mentioned. Furthermore, currently I have my react to handle requests with a single port 8080, thus the port of the coordinator should not change. Some minor changes include the name of the application.

Conclusion:

I created a simple e-commerce application – BookKeeper, that allows user to maintain the books owned. The important aspect include the implementation of distributed backend to handle multiple client requests. It follows the basic principles of a complex distributed application. The system can be run on any machine. The system is scalable, with many servers can be replicated and new services can be included to make the system more complex. It includes various distributed algorithms to achieve different properties and transparencies of a distributed system.