Project4 Executive Summary

Assignment Overview

The major purpose of project 4 is to use PAXOS to replicate distinct servers, instead of 2-phase-commit protocol in project 3, to **achieve fault tolerance** during concurrent handling among multiple servers, and therefore building a more scalable distributed Key Value Store system. Through the implementation process of PAXOS protocol, we were able to make the clients be able to contact any one of the created servers and get consistent data back from any of the replicas. Similar to project 3, only PUT and DELETE operations are needed to consider in the new implementation of the server since they will be writing/modifying data. However, project 4: PAXOS, known as a **consensus** algorithm, manages to handle failures. This is achieved by creating a coordinator to connect to the server replicas, and having three roles inside a server: Proposer, Acceptor and Learner. These three roles perform **several phases** below and justify in the Acceptor that if a majority of the acceptors promise and accept the proposal.
Phase 1a: proposer-prepare
Phase 1b: acceptor-promise
Phase 2a: proposer-propose
Phase 2b: acceptor-accept
Phase 3: learner-learn/commit

In addition, random failures(10% probability) were arranged in the Acceptor that peer servers may fail during the process of Phase 1b and Phase 2b. When a server is down, it can also be restarted manually by another current live server. These are all simulation forms of failures that can test if the system is fault-tolerant.

Practices contain:

1. **Design and programming:**

- Java multi-thread programming with handling mutual exclusion(*synchronized* keyword)
- Java concurrency implementation by using executor framework, future and callable
- Java RMI usage
- Java object-oriented design
- Algorithm/Protocol design
- Code refactoring

2. **(Still) Learning to use new tools:**

- Docker(learn to use a container, pack the application to fit different Operating Systems aligns with the idea of building distributed systems)
- Shell scripts

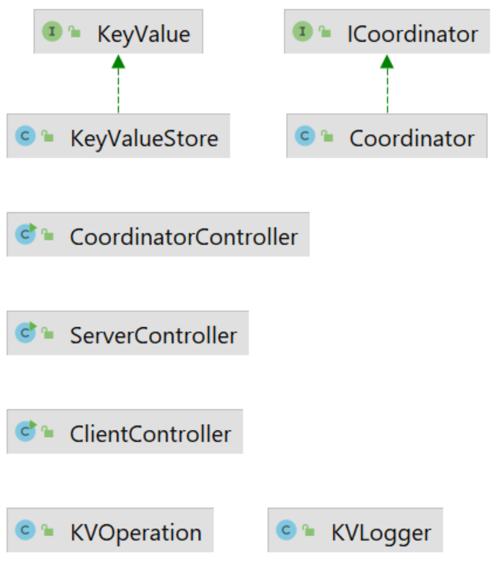
Technical Impression

Features:

1. **Put** operation: for users to put a key-value pair into the store. Putting in an already-in-the-store key with a new value will override the old value. This will be consistently updated by all of the server replicas.
2. **Get** operation: for users to get the value by giving a specific key. Getting a key that is not in the store will prompt a warning log message to the user.
3. **Delete** operation: for users to delete a key-value pair into the store by giving a specific key. Deleting a key that is not in the store will prompt a warning log message to the user. This will be consistently updated by all of the server replicas.
4. **Multi-threaded Replica Servers:** The project supports multi-threaded servers(i.e. In the test screenshots, five servers were created but it can be any number) and they can consistently handle multiple requests from different clients, just acting like one single server.
5. **Log messages**: The program includes human-readable log messages for users to interact with correct commands input.
6. **Fault-tolerance**: The multi-threaded replica servers are fault-tolerant, which means put and delete may randomly fail in some servers, and if a majority of live peer servers are down, the consensus cannot be reached and the put/delete won't be successful, but it will never force client down and can restart the server quickly if a server downs.

Assumptions & Design Choices:

1. OOD Design structure clarifications



The **client** package includes:
- Client App/Controller with the main method: *ClientController*

The **server** package includes:
- KeyValue store and operations with related files: interface *KeyValue*, class *KeyValueStore* *Notice*** In class KeyValueStore, three PAXOS roles Proposer, Acceptor and Learner are represented by inner classes.*
- Server App/Controller with the main method: *ServerController*

The **coordinator** package includes:
- Coordinator for leading the server and store a groups of peer servers' information with related files: interface *ICoordinator*, class *Coordinator*
- Coordinator App/Controller with the main method: *CoordinatorController*

The **utils** package includes:
- KeyValue store operation of PUT, GET and DELETE: *KVOperation*
- Logs print out in human-readable format with timestamps of current system time: *KVLogger*

2. Protocol Design and Assumptions

The PAXOS Protocol design is based on the below assumptions:

(1) Transactions(the sequences of operations) are not being considered.

The response/request parsing design:

Since there is no specific requirement for the project, I kept a super rough design - performing regular expressions with strings to deal with requests by splitting them with space. As for responses, I used ";" as a separator to handle code and response repsectively. This was highly based on the assumption that we only include a limited type of requests, code and reponses.

3. Implementing details(more can be found in comments and documentation in code)
- For the put method, putting in an already-in-the-store key with a new value will override the old value.

- Prepopulated data sets operations will be performed at the time that any client starts.

- In order to test the main improvement/functionality of this project 4 easily, CoordinatorController is set by using RMI via a hard-coded port number 1111 to let server replicas connect with. But notice that the communication between servers and clients are still via command line arguments.

Enhancements for future works:

1. Scalability and Modularity consideration for Protocol Design. Instead of handling strings and performing regular expressions, we can create an encapsulation class for request and

response to handle them specifically. Thus in future implementations if we need to handle a larger number/type of requests and responses, we have more flexibility on customizing the design.

2. Scalability consideration for multithreaded programs especially when the situations are not as simple as current. In the future, we may be required to handle synchronization based on specific conditions, so setting locks explicitly may be necessary.

3. Scalability considerations for handling transactions in a distributed environment. In order to do this, we can maintain another container to store the sequence of operations of transactions, and in PAXOS, a transaction to represent a proposal with a proposal Id, instead of only a single operation for a proposal.

4. Thread-safe program consideration. With the increasing usage of threads in the program, we should make sure at caution that we are implementing a thread-safe program. For example, in the future, we may be using Collections.synchronizedMap or ConcurrentHashMap instead of a normal HashMap to store the Key-Value pairs.