Project3 Executive Summary

Assignment Overview

The major purpose of project 3 is to understand how to replicate distinct servers, instead of using only a single server, for the Key Value store to increase Server bandwidth and ensure availability. Through the implementation process of **Two Phase Commit 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, although the server which the client is contacting, is the "main" server that will give more information than other replicas. Compared with Project 2, only PUT and DELETE operations are needed to consider in the new implementation of the server since they will be writing/modifying data. In other words, whenever a client issues a PUT or DELETE operation to any of the all created servers, that receiving "main" server will ensure the updates have been received and committed across all of the replicas. This is achieved by creating a coordinator to connect to the server replicas, by performing:
(1) Phase 1: prepare(voting to see if all the replicas agreed the operation, but a timeout mechanism is set to check server failures);
(2) Phase 2: commit or abort based on the voting result.

In addition, 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
- 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.

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*
- Server App/Controller with the main method: *ServerController*

The **coordinator** package includes:
- Coordinator for two phase commit protocol 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 Two Phase Commit Protocol design is based on the below assumptions:

(1) We will not have any server failures to ensure the protocol will not stall, although I also set a timeout mechanism of 5 seconds to roughly handle this.
(2) 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 3 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.

- Under the assumption that we don't handle transactions, the serverAbort method in KeyValueStore(the server class) is only logging an info message for the current version without any "aborting logic details".

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 therefore we will implement the serverAbort method body in detail.

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.