Multi-threaded Key-Value Store using RPC

I. Assignment Overview

The current project implements the basic functionalities of Remote Method Invocation (RMI) for key-value storing. The focus is on implementing sample clients to concurrently communicate with the server and perform three basic operations: 1) PUT (key, value), 2) GET (key), and 3) DELETE (key). Five different instances of server are replicated to increase Server bandwidth and ensure availability. Client is able to to access any of the five key-value servers instead of a single server, and get consistent data back from any of the replicas.

The current project consists of four test runs of multithreaded client-server communication (through Coordinator.java (under the folder of Server) and client.java (under the folder of Client)). For each test run, a ConcurrentHashmap is created in each of the replicas in order to store the requests of the three operations and ensure thread-safety in a high concurrency environment.

A protocol is designed to dictate whether a request by a client is valid (see details in the Technical Impression). The server is multi-threaded such that it responds to requests from multiple clients one at a time. Under the client-server program, the server runs forever whereas the client will exit and disconnect with the server until it loops through all the requests as specified in the sample text file.

The client side of the current project considers the following main features:

(1) The client is able to contact any of the five key-value replica servers instead of one single server by specifying the desired port numbers;
(2) The protocol designed to communicate packet contents ensures that the server is robust to malformed datagram packets, as mainly reflected in the get(), put() and delete() methods;
(3) Time-stamped client log is formated in both success and error messages.
(4) Multiple clients can read data very fast, and write data with a lock to ensure database consistency.

The server side of the current project considers the following features:

(1) The server and its replicas listen on ports (which should be specified in command line);
(2) The server runs forever unless external signal is enforced;

(3) The server display client's request and sends time-stamped response to valid requests back to the client in a human readable fashion;
(4) The server is robust to malformed datagram packets and reports the error in the server log in a human-readable format.
(5) Mutual exclusion of threads is handled in a mulitple-read-single-write fashion and the priority of write is on top of the reads.
(6) Implementing a two-phase commit protocol ensures that client is able to issue PUT, DELETE and get consistent data back from any of the replicas (when server(s) is functioning).

II. Technical Impression

1. Language and Tools

I used Java as the programming language. RMI is used for the key-value store service for the following reasons:

(1) The service needs to be safe for concurrent access in case multiple updates happen at the same time;
(2) The service needs to be able to scale up to use the available hardware.
(3) The service needs to be fast.
(4) The service can take advantage of efficient serialization, a simple IDL, and easy interface updating.

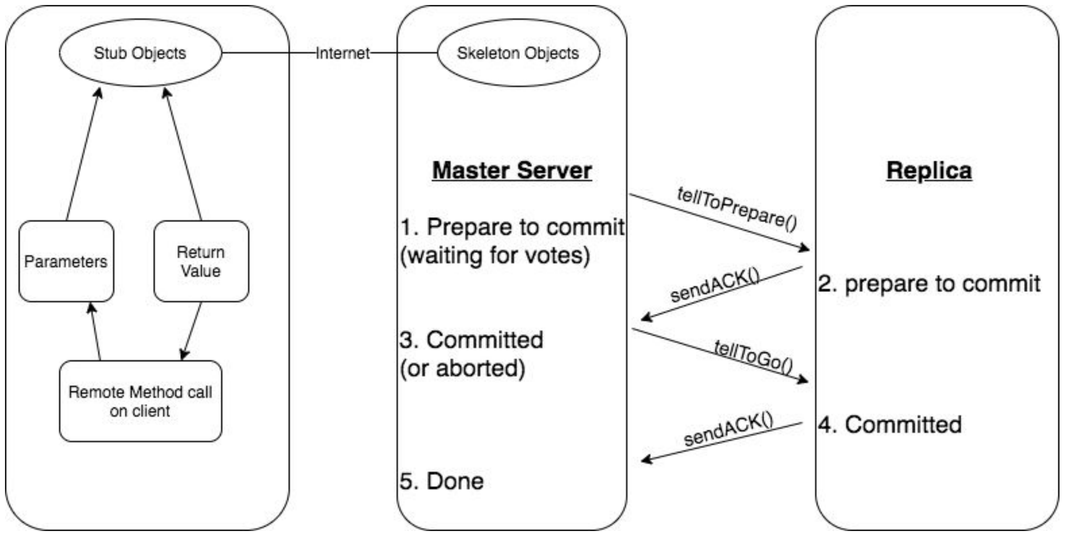
2. Setup

In order to implement all the features in the sending and receiving end-points as discussed in the previous sections, I designed four main classes, two helper classes, and one interface:

- The ***Coordinator*** class under the server folder manages startup/shutdown of the key value store server which parses arguments into port numbers so that a server and its four replicas could listen on ports as specified. Please note that I coded the program such that the Coordinator brings down the server on Port 4001 after 2 minutes of running. This is to test the availability of servers and response to the clients (see Results for details). The whole program continues to run (without server 4001) thereafter.
- The ***Client*** class under the client folder simulates user(s) of the key-value store system. It runs a sample text file for putting, getting and deleting keys and values. Four sample text files are created for sample runs. You could specify the file pathname as needed. For example, ClientRequest.txt is a sample file for

testing all three operations. The other three txt files are for running a unit test of the PUT, GET and DELETE functions for multi-thread concurrent access respectively. You will need to specify the path name for each file.

- The ***Server*** class implements the KeyValStoreInterface under the keyValService folder. The following figure shows the architecture overview of the server class. When a client accesses a server (here I call master server), the server would follow the 5 steps in figure 1. Whenever a server is initiated, four ConcurrentHashMap are used to indicates in the current master server (1) what is the user update request (*pendingRequest*) in the temporary storage (as in step 1 in figure 1), (2) how do each of the replicas say about their preparation about the update (*pendingPrepareAcks*, as in step 2 in figure 1)), (3) how do each of the replicas say about committing the changes requested by the user (*pendingGoAcks*, as in step 4 in figure 1), and (4) the key value store in the current master server (*store*). For concurrent hashmaps (2) and (3), the master server goes through a collection of votes process that determines if the PREPARE ACK and GO ACK are added up to 4 (meaning 4 replicas) to do the final commit. This voting collection process will experience a random 50 to 100 milliseconds delay to make the message transferring process more like a real-life network. Three attempts are made before decisions to commit or abort are made.



- Figure 1. Architecture overview

- The ***keyValStoreInterface*** defines the key-value store service. It describes what clients can expect from the service with the three operations (PUT, GET, DELETE).

3. Data Design and Data Structures

As described in the Setup section, I used Java ConcurrentHashmap for storing pending changes, pending ACKs (prepare and go messages), key-value store database that is used for storing, deleting, and checking key and value. I used ConcurrentHashMap because I need very high concurrency in the current project. It has four advantage over hashmap:

(1) It is thread-safe without synchronizing the whole map.
(2) Reads (e.g., get) can happen very fast while write (e.g., delete, put) is done with a lock.
(3) There is no locking at the object level.
(4) The locking is at a much finer granularity at a hashmap bucket level.

4. Protocol for PUT, GET, and DELETE operations

Every client must follow the syntax of <operation> <key> for GET and DELETE, and <operation> <key> <value> for PUT. If invalid request is sent, the server will send time-stamped error message such as “Malformed Request from [IP: " + clientAddress + ", Port: "+ clientPort + "]." + " Syntax: <operation> <key> OR <operation> <key> <value>. For example: get apple".

The protocol of the three operations are as follows:

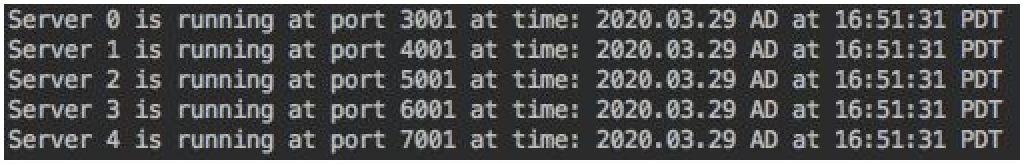
(1) PUT inserts a key-value pair into the database, and if key exists, no change is made, and error message of “key already exists” is sent to client. With ConcurrentHashmap, only one client can do the put operation once at a time.
(2) GET retrieves the value by key in the database. If no put or delete is in action, multiple clients can get the value concurrently without blocking each other.
(3) DELETE removes the key-value pair by key in the database. If no key exists, error message such as “Key does not exist” is sent over to the client. Only one client can do DELETE once at a time, with all other threads which intend to do DELETE being blocked until lock is released.

5. Usage

The current project runs on two main classes. To start, in your CLI, type in the following:

```
java -jar [path directory of the coordinator jar] 3001 4001 5001 6001 7001
```

The following should appear:



```
Server 0 is running at port 3001 at time: 2020.03.29 AD at 16:51:31 PDT
Server 1 is running at port 4001 at time: 2020.03.29 AD at 16:51:31 PDT
Server 2 is running at port 5001 at time: 2020.03.29 AD at 16:51:31 PDT
Server 3 is running at port 6001 at time: 2020.03.29 AD at 16:51:31 PDT
Server 4 is running at port 7001 at time: 2020.03.29 AD at 16:51:31 PDT
```

Then open up another CLI window, and type in the following:

```
java -jar [path directory of the client jar] [port numbers separated by white space]
[sampe run txt file path]
```

For example, if the sample run txt file is ClientRequest.txt, specify it in your last argument, such as:
Java -jar 3001 4001 /Users/Sam/Desktop/project3/out/artifacts/Client/ClientRequest.txt
To switch to other sample runs (e.g., ClientRequestGetOnly.txt, ClientRequestPutOnly, ClientRequestDeleteOnly), specify as instructed.
The ClientRequest.txt sample run consists of a mix of at least five of each operation: 5 PUTs, 5 GETs, 5 DELETEs.

6. Test Result and Output:

Screenshots of the server and client sides for the key value store are shown below:

Scenario 1

1) Run all 5 servers (runs on ports 3001 4001 5001 6001 7001):

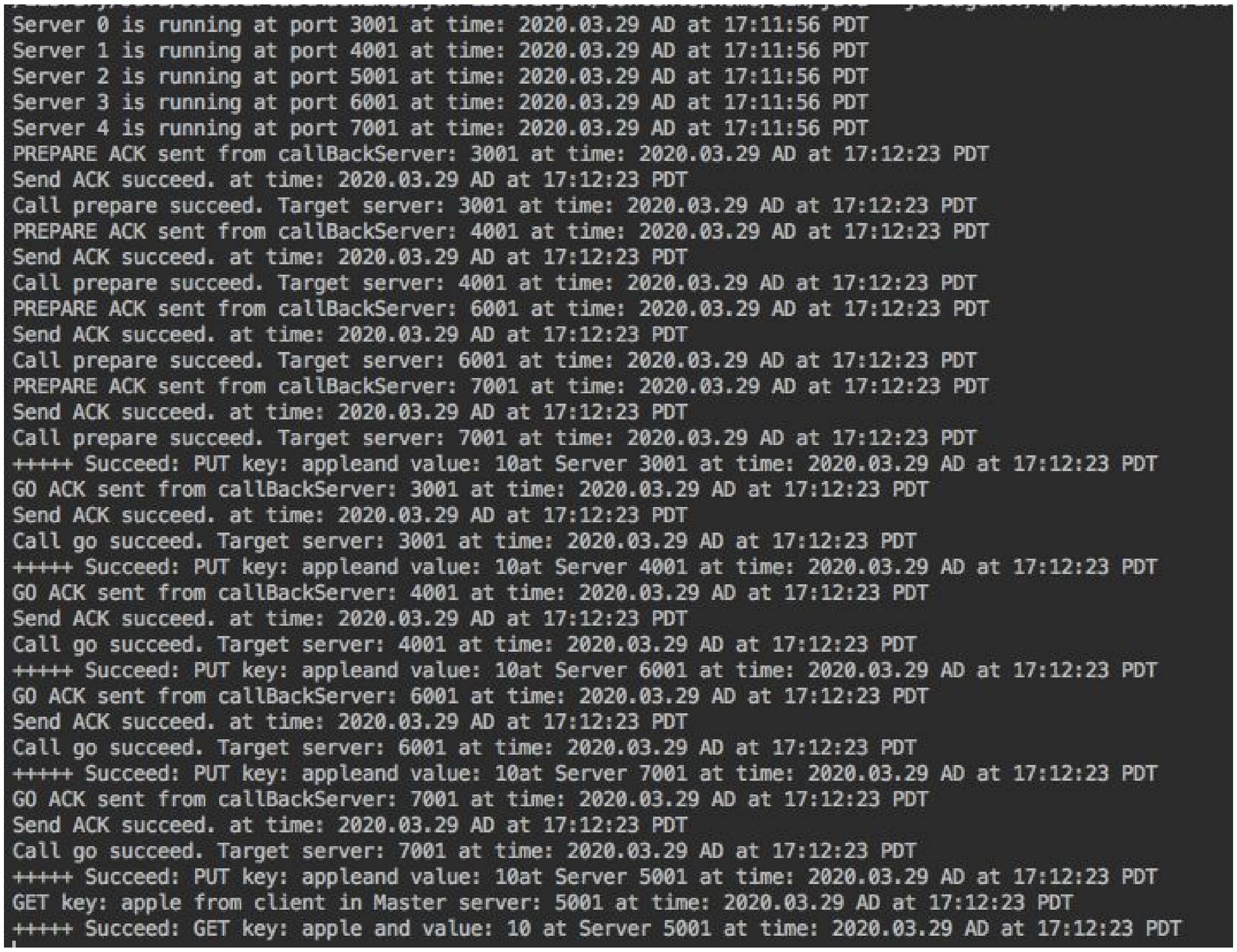
```
java -jar [path directory of the coordinator jar] 3001 4001 5001 6001 7001
```

2) Run client 1 and make it connect to server 1 (port 3001), and run client 2 and make it connect to server 2 (port 4001).

```
java -jar [path directory of the client jar] 3001 4001 [sampe run txt file path]
```

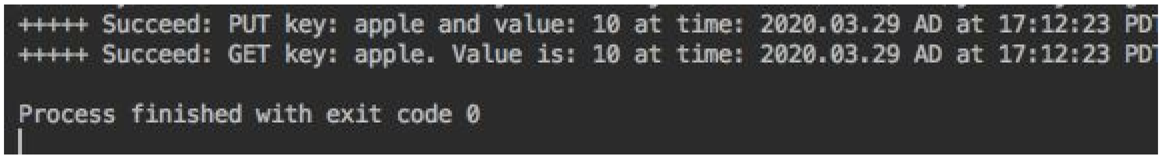
Please replace [sampe run txt file path] with the path of **ClientRequestGetOnly.txt** file.
3) Do put and get from client 1 on server 1, and do put and get from client 2 on server 2

We should be able to put key-value pair by client 1 on server 1 only, and we should be able to get the key put in from both clients 1 and 2.



```
Server 0 is running at port 3001 at time: 2020.03.29 AD at 17:11:56 PDT
Server 1 is running at port 4001 at time: 2020.03.29 AD at 17:11:56 PDT
Server 2 is running at port 5001 at time: 2020.03.29 AD at 17:11:56 PDT
Server 3 is running at port 6001 at time: 2020.03.29 AD at 17:11:56 PDT
Server 4 is running at port 7001 at time: 2020.03.29 AD at 17:11:56 PDT
PREPARE ACK sent from callBackServer: 3001 at time: 2020.03.29 AD at 17:12:23 PDT
Send ACK succeed. at time: 2020.03.29 AD at 17:12:23 PDT
Call prepare succeed. Target server: 3001 at time: 2020.03.29 AD at 17:12:23 PDT
PREPARE ACK sent from callBackServer: 4001 at time: 2020.03.29 AD at 17:12:23 PDT
Send ACK succeed. at time: 2020.03.29 AD at 17:12:23 PDT
Call prepare succeed. Target server: 4001 at time: 2020.03.29 AD at 17:12:23 PDT
PREPARE ACK sent from callBackServer: 6001 at time: 2020.03.29 AD at 17:12:23 PDT
Send ACK succeed. at time: 2020.03.29 AD at 17:12:23 PDT
Call prepare succeed. Target server: 6001 at time: 2020.03.29 AD at 17:12:23 PDT
PREPARE ACK sent from callBackServer: 7001 at time: 2020.03.29 AD at 17:12:23 PDT
Send ACK succeed. at time: 2020.03.29 AD at 17:12:23 PDT
Call prepare succeed. Target server: 7001 at time: 2020.03.29 AD at 17:12:23 PDT
+++++ Succeed: PUT key: appleand value: 10at Server 3001 at time: 2020.03.29 AD at 17:12:23 PDT
GO ACK sent from callBackServer: 3001 at time: 2020.03.29 AD at 17:12:23 PDT
Send ACK succeed. at time: 2020.03.29 AD at 17:12:23 PDT
Call go succeed. Target server: 3001 at time: 2020.03.29 AD at 17:12:23 PDT
+++++ Succeed: PUT key: appleand value: 10at Server 4001 at time: 2020.03.29 AD at 17:12:23 PDT
GO ACK sent from callBackServer: 4001 at time: 2020.03.29 AD at 17:12:23 PDT
Send ACK succeed. at time: 2020.03.29 AD at 17:12:23 PDT
Call go succeed. Target server: 4001 at time: 2020.03.29 AD at 17:12:23 PDT
+++++ Succeed: PUT key: appleand value: 10at Server 6001 at time: 2020.03.29 AD at 17:12:23 PDT
GO ACK sent from callBackServer: 6001 at time: 2020.03.29 AD at 17:12:23 PDT
Send ACK succeed. at time: 2020.03.29 AD at 17:12:23 PDT
Call go succeed. Target server: 6001 at time: 2020.03.29 AD at 17:12:23 PDT
+++++ Succeed: PUT key: appleand value: 10at Server 7001 at time: 2020.03.29 AD at 17:12:23 PDT
GO ACK sent from callBackServer: 7001 at time: 2020.03.29 AD at 17:12:23 PDT
Send ACK succeed. at time: 2020.03.29 AD at 17:12:23 PDT
Call go succeed. Target server: 7001 at time: 2020.03.29 AD at 17:12:23 PDT
+++++ Succeed: PUT key: appleand value: 10at Server 5001 at time: 2020.03.29 AD at 17:12:23 PDT
GET key: apple from client in Master server: 5001 at time: 2020.03.29 AD at 17:12:23 PDT
+++++ Succeed: GET key: apple and value: 10 at Server 5001 at time: 2020.03.29 AD at 17:12:23 PDT
```

Figure 2. Result of clientRequestGetOnly.txt(Server side).



```
+++++ Succeed: PUT key: apple and value: 10 at time: 2020.03.29 AD at 17:12:23 PD
+++++ Succeed: GET key: apple. Value is: 10 at time: 2020.03.29 AD at 17:12:23 PD

Process finished with exit code 0
```

Figure 3. Result of clientRequestGetOnly.txt (client side)

Scenario 2

1) Run all 5 servers

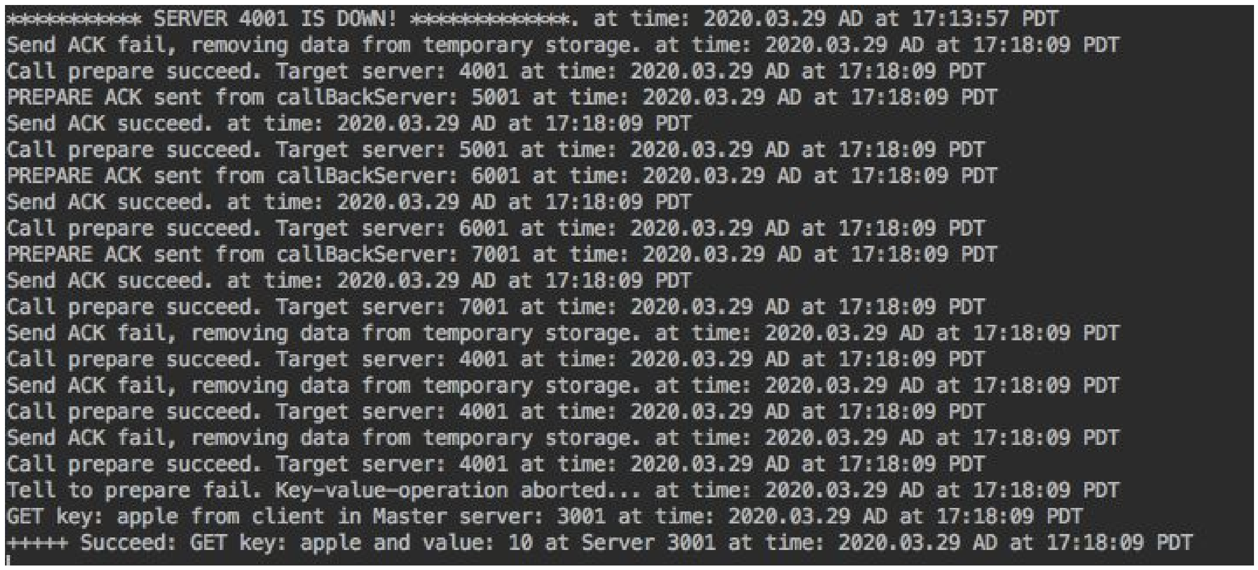
Same as Scenario 1 step (1).

2) Run client 1 and make it connect to server 1, bring down server2 (the current program brings down server 4001 automatically after 2 minutes of running the servers, so you don't need to do anything here except to wait for the server to crash).

3) Do PUT and GET from client on server1 - The PUT operation should abort since server2 is down, whereas the GET operation is still working.

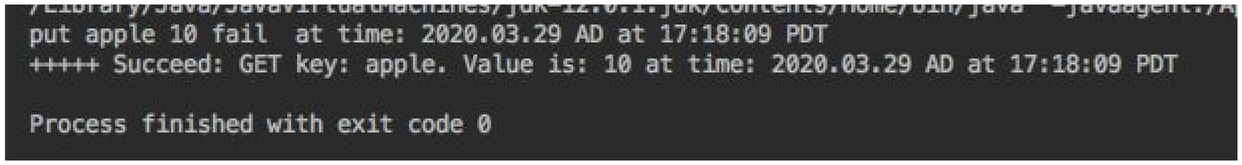
```
java -jar [path directory of the client jar] 3001 [sampe run txt file path]
```

Please replace [sampe run txt file path] with the path of **ClientRequestGetOnly.txt** file.



```
*********** SERVER 4001 IS DOWN! *************. at time: 2020.03.29 AD at 17:13:57 PDT
Send ACK fail, removing data from temporary storage. at time: 2020.03.29 AD at 17:18:09 PDT
Call prepare succeed. Target server: 4001 at time: 2020.03.29 AD at 17:18:09 PDT
PREPARE ACK sent from callBackServer: 5001 at time: 2020.03.29 AD at 17:18:09 PDT
Send ACK succeed. at time: 2020.03.29 AD at 17:18:09 PDT
Call prepare succeed. Target server: 5001 at time: 2020.03.29 AD at 17:18:09 PDT
PREPARE ACK sent from callBackServer: 6001 at time: 2020.03.29 AD at 17:18:09 PDT
Send ACK succeed. at time: 2020.03.29 AD at 17:18:09 PDT
Call prepare succeed. Target server: 6001 at time: 2020.03.29 AD at 17:18:09 PDT
PREPARE ACK sent from callBackServer: 7001 at time: 2020.03.29 AD at 17:18:09 PDT
Send ACK succeed. at time: 2020.03.29 AD at 17:18:09 PDT
Call prepare succeed. Target server: 7001 at time: 2020.03.29 AD at 17:18:09 PDT
Send ACK fail, removing data from temporary storage. at time: 2020.03.29 AD at 17:18:09 PDT
Call prepare succeed. Target server: 4001 at time: 2020.03.29 AD at 17:18:09 PDT
Send ACK fail, removing data from temporary storage. at time: 2020.03.29 AD at 17:18:09 PDT
Call prepare succeed. Target server: 4001 at time: 2020.03.29 AD at 17:18:09 PDT
Send ACK fail, removing data from temporary storage. at time: 2020.03.29 AD at 17:18:09 PDT
Call prepare succeed. Target server: 4001 at time: 2020.03.29 AD at 17:18:09 PDT
Tell to prepare fail. Key-value-operation aborted... at time: 2020.03.29 AD at 17:18:09 PDT
GET key: apple from client in Master server: 3001 at time: 2020.03.29 AD at 17:18:09 PDT
+++++ Succeed: GET key: apple and value: 10 at Server 3001 at time: 2020.03.29 AD at 17:18:09 PDT
```

Figure 4. Result of clientRequestGetOnly.txt (Server side after server 4001 is down).



```
/Library/Java/JavaVirtualMachines/jdk-12.0.1.jdk/Contents/Home/bin/java -javaagent:/A
put apple 10 fail  at time: 2020.03.29 AD at 17:18:09 PDT
+++++ Succeed: GET key: apple. Value is: 10 at time: 2020.03.29 AD at 17:18:09 PDT

Process finished with exit code 0
```

Figure 5. Result of clientRequestGetOnly.txt (Client side).

Scenario 3

1) Run all 5 servers

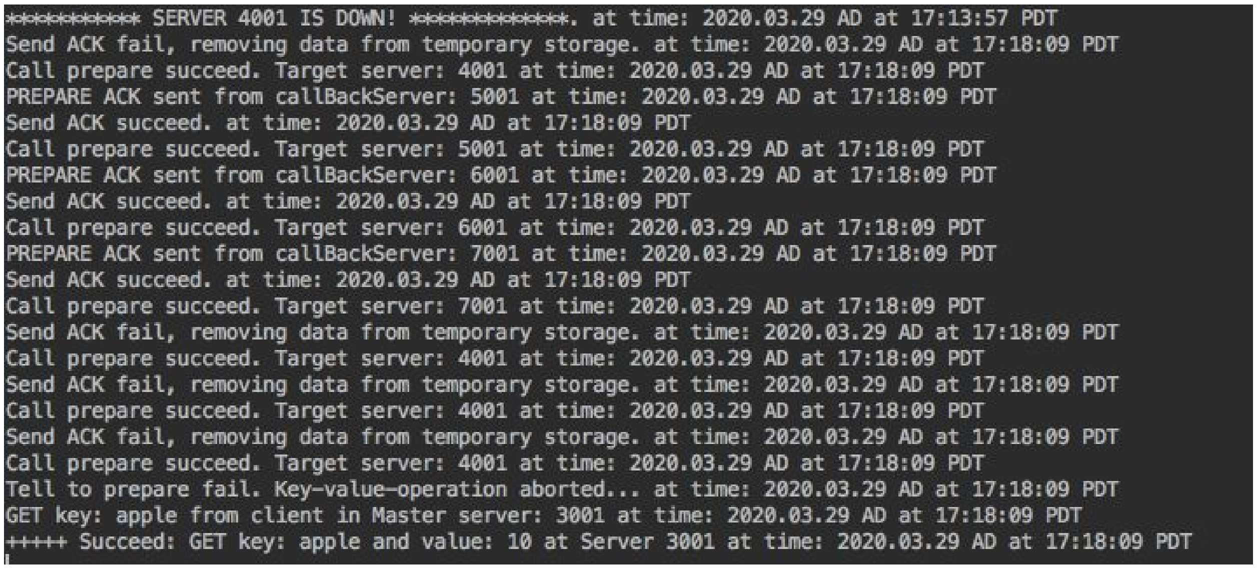
Same as Scenario 1 step (1).

2) Run client 1 and make it connect to server 2, bring down server2 (the current program brings down server 4001 automatically after 2 minutes of running the servers, so you don't need to do anything here except to wait for the server to crash).

3) Do PUT and GET from client on server2 - The PUT and GET operations should abort since server2 is down. The logs of the server side should be unchanged, since server 2 is down!

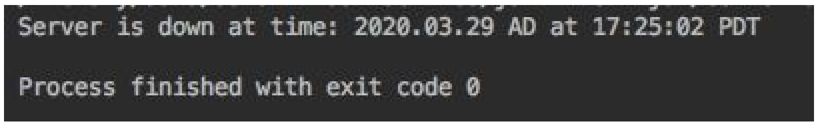
```
java -jar [path directory of the client jar] 4001 [sampe run txt file path]
```

Please replace [sampe run txt file path] with the path of **ClientRequestGetOnly.txt** file.



```
*********** SERVER 4001 IS DOWN! *************. at time: 2020.03.29 AD at 17:13:57 PDT
Send ACK fail, removing data from temporary storage. at time: 2020.03.29 AD at 17:18:09 PDT
Call prepare succeed. Target server: 4001 at time: 2020.03.29 AD at 17:18:09 PDT
PREPARE ACK sent from callBackServer: 5001 at time: 2020.03.29 AD at 17:18:09 PDT
Send ACK succeed. at time: 2020.03.29 AD at 17:18:09 PDT
Call prepare succeed. Target server: 5001 at time: 2020.03.29 AD at 17:18:09 PDT
PREPARE ACK sent from callBackServer: 6001 at time: 2020.03.29 AD at 17:18:09 PDT
Send ACK succeed. at time: 2020.03.29 AD at 17:18:09 PDT
Call prepare succeed. Target server: 6001 at time: 2020.03.29 AD at 17:18:09 PDT
PREPARE ACK sent from callBackServer: 7001 at time: 2020.03.29 AD at 17:18:09 PDT
Send ACK succeed. at time: 2020.03.29 AD at 17:18:09 PDT
Call prepare succeed. Target server: 7001 at time: 2020.03.29 AD at 17:18:09 PDT
Send ACK fail, removing data from temporary storage. at time: 2020.03.29 AD at 17:18:09 PDT
Call prepare succeed. Target server: 4001 at time: 2020.03.29 AD at 17:18:09 PDT
Send ACK fail, removing data from temporary storage. at time: 2020.03.29 AD at 17:18:09 PDT
Call prepare succeed. Target server: 4001 at time: 2020.03.29 AD at 17:18:09 PDT
Send ACK fail, removing data from temporary storage. at time: 2020.03.29 AD at 17:18:09 PDT
Call prepare succeed. Target server: 4001 at time: 2020.03.29 AD at 17:18:09 PDT
Tell to prepare fail. Key-value-operation aborted... at time: 2020.03.29 AD at 17:18:09 PDT
GET key: apple from client in Master server: 3001 at time: 2020.03.29 AD at 17:18:09 PDT
+++++ Succeed: GET key: apple and value: 10 at Server 3001 at time: 2020.03.29 AD at 17:18:09 PDT
```

Figure 6. Result of clientRequestGetOnly.txt (server side).



```
Server is down at time: 2020.03.29 AD at 17:25:02 PDT

Process finished with exit code 0
```

Figure 7. Result of clientRequestGetOnly.txt(client side)

7. Area of Improvement

A number of improvements could be made to make the project cleaner and closer to real-life scenario applications. First, using gRPC instead of RMI is encouraged. Second, the current situation only considers key of any size and no restrictions are imposed on the type of keys. Future work should consider other user inputs and provide validation of possible values. Third, the current project lets user to specify ports to latch on in order to spawn multiple threads.I also required client to sleep for 50 to 100 milliseconds for each request to control possible incoming data flows. Future work should focus on how thread pools (e.g., schedule thread pool and fixed size thread pool) should perform. Fourth, two mechanisms are used to allow single-write and multiple reads: (1) ReadWriteLock class to impose locks on reads and writes respectively, and (2) ConcurrentHashmap was created for storing key-value pairs for single-write and multiple reads. Potential improvements in accessibility such as caching of the most recently used key-value pairs could enhance user experience and improve I/O speeds. Lastly, the client robustness to server failure could be enhanced by using a timeout mechanism to deal with an unresponsive server or stalled connection.