Key-Value Store Client-Server Program - Multithreading RPC

I. Assignment Overview

The current project implements the basic functionalities of Remote Procedure Call (RPC) 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). The current project consists of four test runs of multithreaded client-server communication (through KeyValStoreServer.java and KeyValStoreThreadPool.java). For each test run, a ConcurrentHashmap is created 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 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 uses the default localhost of the server;
(2) The client is robust to server failure by using a timeout mechanism (for example, the code snippet of newBlockingStub.withDeadlineAfter() in KeyValStoreClient.java) to deal with an unresponsive server or stalled connection;
(3) 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;
(4) Time-stamped client log is formated in both success and error messages.
(5) 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 listens on port 9090;
(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.

II. Technical Impression

1. Language and Tools

I used Java as the programming language and used Maven to build and manage the project. gRPC 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 and a protobuf file describing the API:

- The ***KeyValStoreServer*** class manages startup/shutdown of the key value store server which uses serverBuilder as a base and listens on port 9090.
- The ***KeyValStoreClient*** class simulates a user 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 runing 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. For example, to test DELETE function, specify the file as "./src/main/java/ClientRequestDeleteOnly.txt".
- The ***KeyValStoreThreadPool*** class uses ThreadPoolExecutor to separate the task creation and its execution. The reason for using threadpool is to make the server multithreaded. With ThreadPoolExecutor, I only have to implement the Runnable objects and send them to the executor. It is responsible for their execution, instantiation, and running with necessary threads. Specifically, I used Fixed thread pool executor which creates a thread pool that reuses a fixed number of threads to execute any number of tasks. The default pool size is 4, and the default client number is 4, in order to test the functionality of multithreaded communication using RPC (e.g., how four clients access the database concurrently).

- The ***keyValStoreImpl*** class implements the key value store service. It is installed by the gRPC Server to handle the requests issued by the client. In order to simulate storing the keys and values on disk, I added short sleeps (see methods pauseThread() while handling the request. Reads and writes will experience a random 50 to 100 milliseconds delay to make the sample run more like a persistent database. I use RandomLocalThread for randomizing the sleep time of a thread rather than the commonly used Random class. The reason is that Random makes poor performance in a multi-threaded environment due to contention – given that multiple threads share the same Random instance. RandomLocalThread addresses this issue by avoiding any concurrent access to the Random objects.
- The ***KeyValueStoreProto.proto*** is the protocol buffer definition of the key-value store service. It describes what clients can expect from the service with the three operations (PUT, GET, DELETE). It defines the service interface and generates client and server code which acts as glue code between the application logic and the core gRPC library. The code called the gRPC client is referred to as *stub*.

3. Data Design and Data Structures

Java ConcurrentHashmap 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.

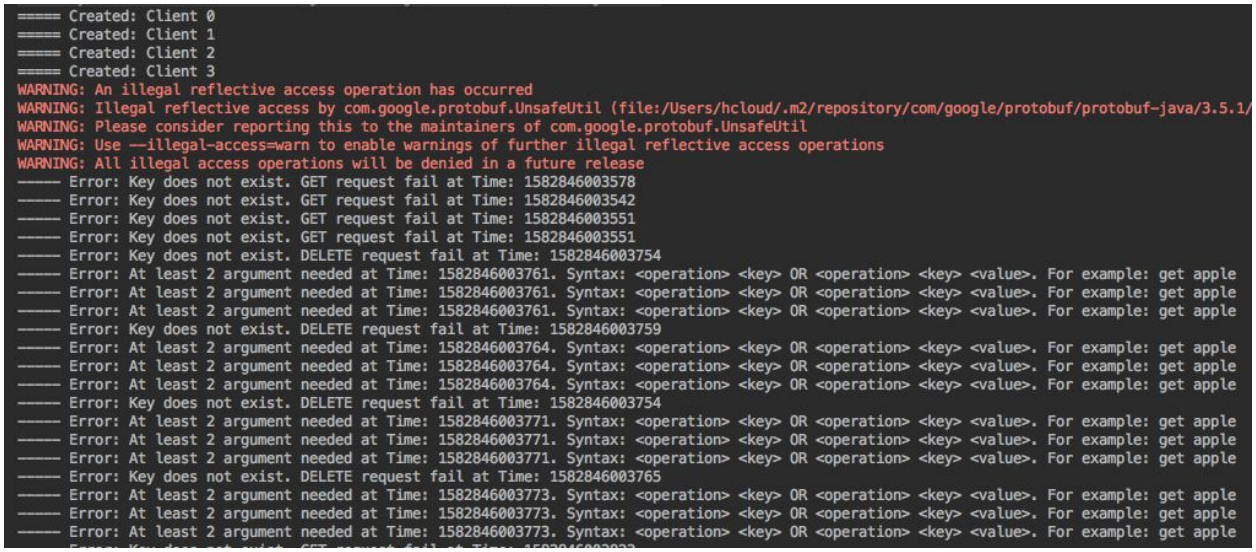
(2) GET retrieves the value by key in the database. If no put or delete is in action, multiple clients can get the value concurrently without blocking each other.
(3) DELETE removes the key-value pair by key in the database. If no key exists, error message such as “Key does not exist” is sent over to 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 main functions. To start, please click Run to run the main function of the *KeyValStoreServer.java*. “Server started at 9090” should appear. Next, click Run to run the main function of the *KeyValStoreThreadPool.java*. The default sample run is on the ClientRequest.txt file, which consists of a mix of at least five of each operation: 5 PUTs, 5 GETs, 5 DELETEs. To change to any of the other 3 sample runs respectively, please specify the path as described above.

6. Test Result and Output:

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



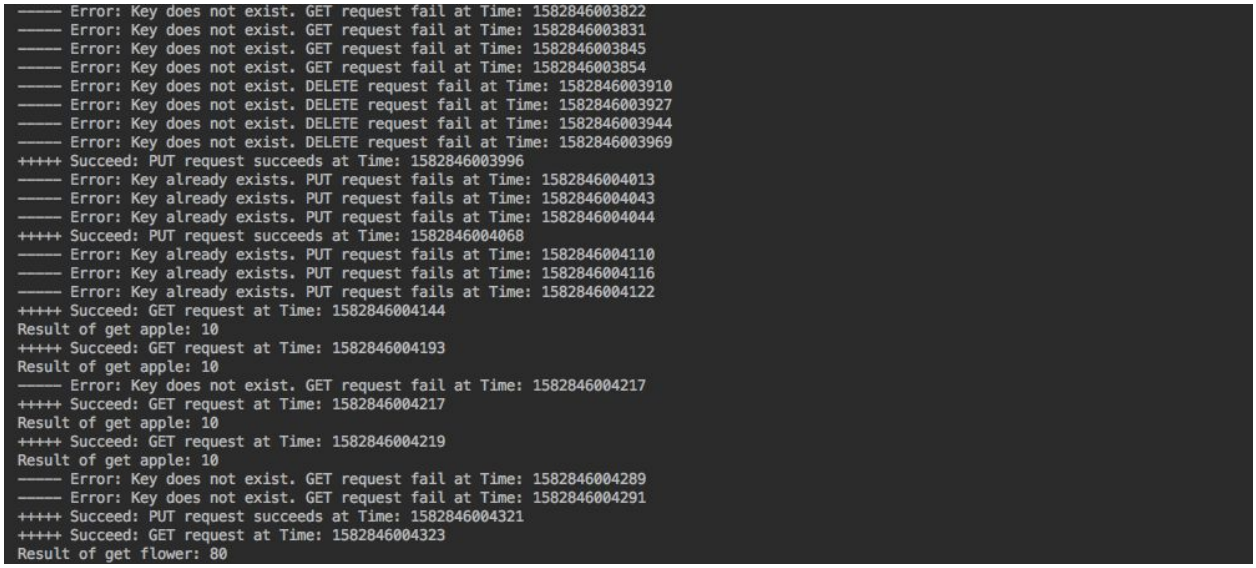


Figure 1. Result of clientRequest.txt (client side)

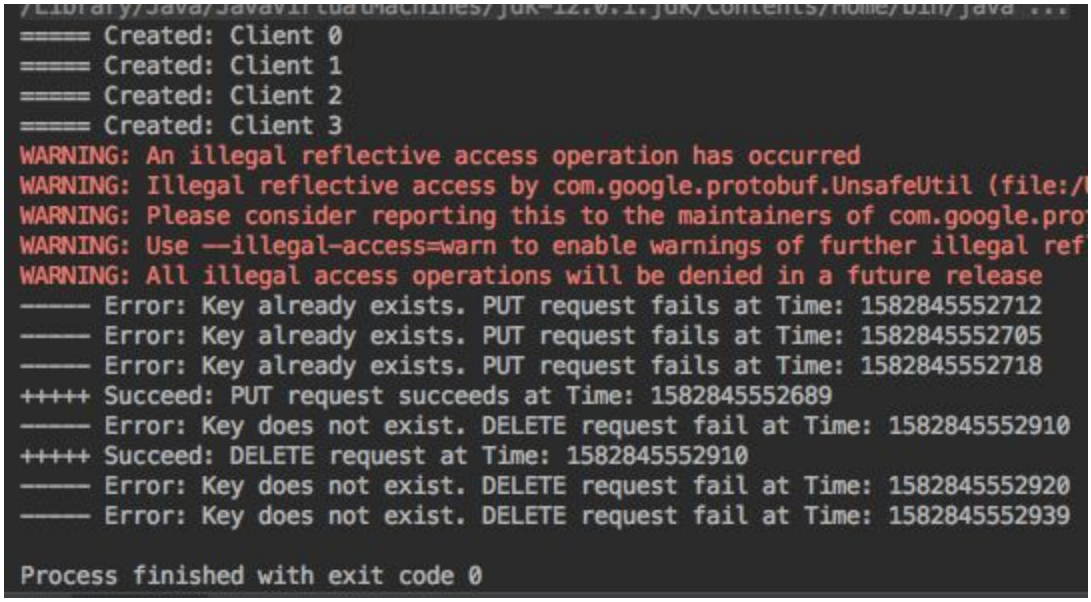


Figure 2. Result of clientDeleteOnly.txt(client side)

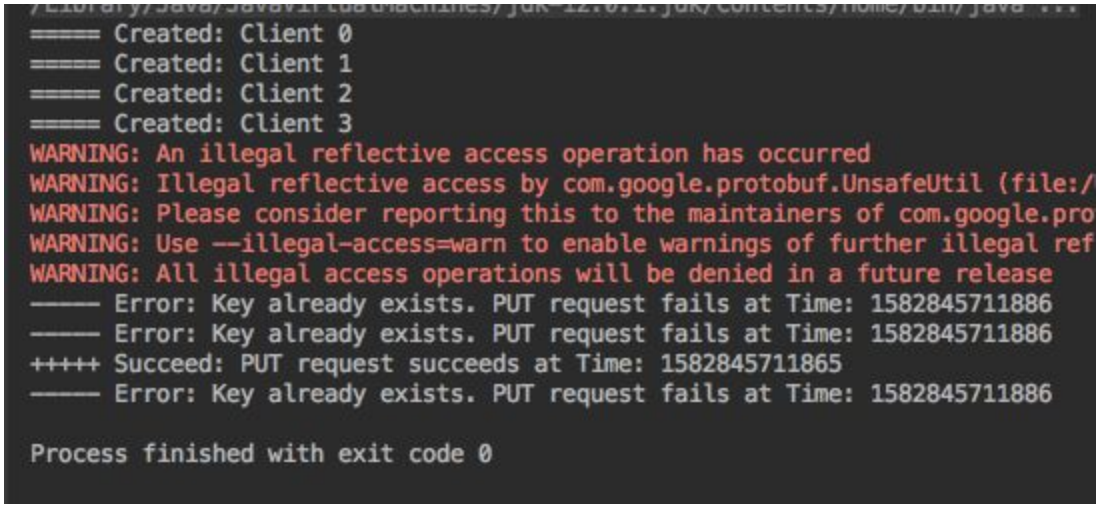


Figure 3. Result of ClientRequestPutOnly.txt (client side)

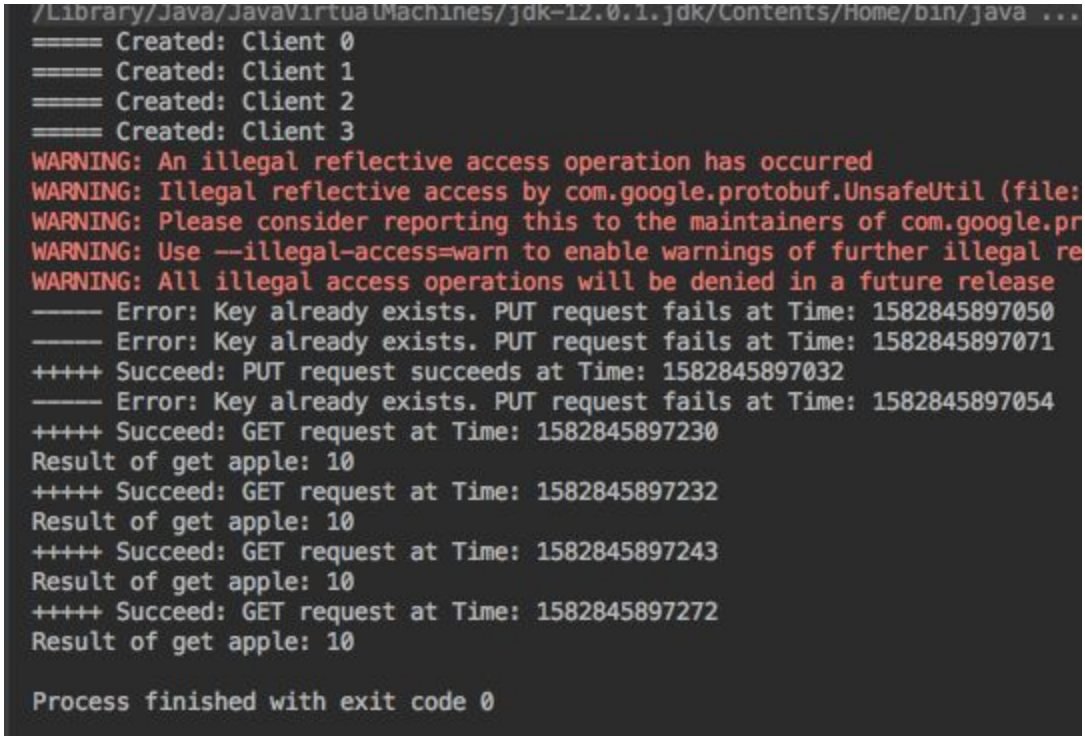


Figure 4. Result of ClientRequestGetOnly.java

7. Area of Improvement

A number of improvements could be made to make the project cleaner and closer to real-life scenario applications. First, given the complexity of multiple dependencies in pom.xml as a Maven project, and the addition of proto file, I have not converted the current project into JAR. Next step could focus on converting the project to JAR so as to better able to run the project with no concern about environment or IDEs. 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 is fixed size thread pool 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 other thread pools (e.g., schedule thread pool) should perform as compared to fixed size thread pool. Lastly, 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.