CSE 513 Programming Assignment 1: Replicated Key-Value Store with Two Different Consistency Models

Out: October 10, 2020 Due: November 10, 2020

Hamidreza (Shahrooz) Zare and Bhuvan Urgaonkar

1 Overview

In this assignment, you will implement a distributed key-value store, which will replicate data across multiple machines. The store will be "geo-distributed" – the machines used in your evaluation will come from specified locations from Google's public cloud (described in Section 3). You will implement two consistency models – both to be covered in class – in your key-value store: (i) linearizability and (ii) causal consistency. For implementing linearizability, you will use an algorithm by Attiya, Bar-Noy, and Welch (simply "ABD algorithm" henceforth). For implementing causal consistency, you will use an algorithm by Ahmad et al. (simply "Ahmad's algorithm" henceforth). Slides/papers describing both these algorithms will be made available on canvas shortly after the project is released. In your experimental evaluation you will compare certain aspects of the performance of your implementations of (i) and (ii). A side benefit of this assignment will be that it will allow you to learn about RPCs (the ideas underlying RPCs are covered in the pre-recorded lecture 9).

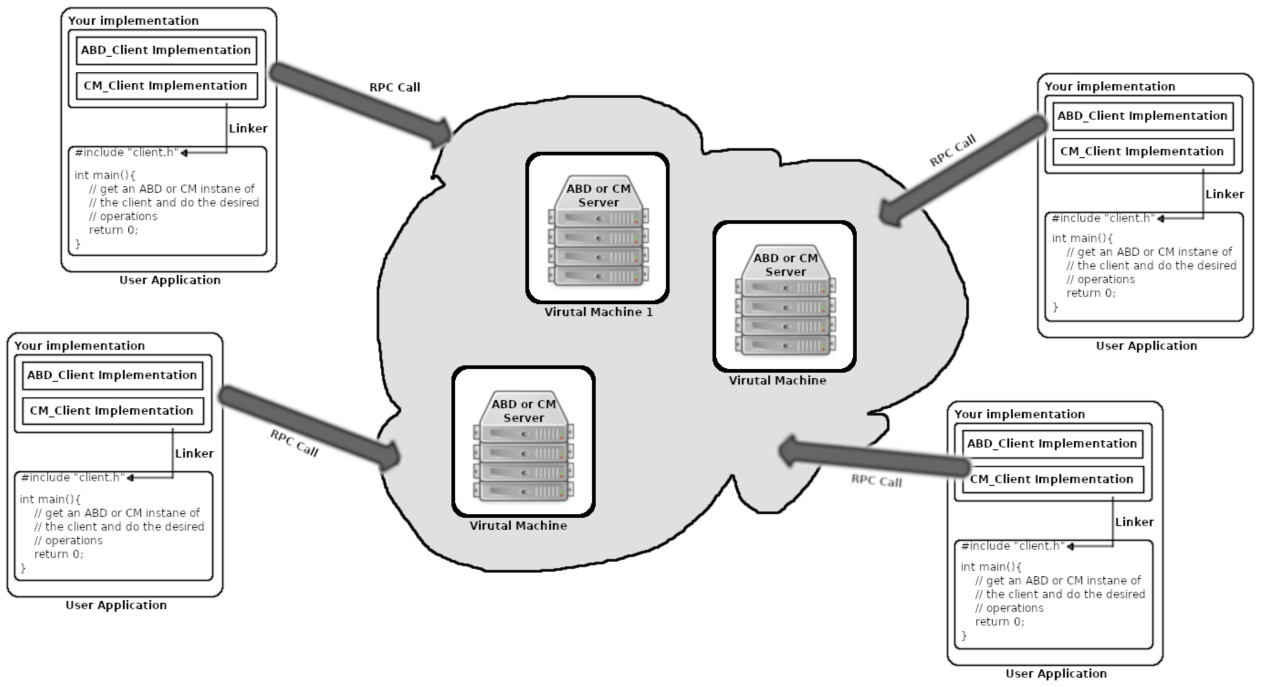
2 Design and Implementation

As Figure 1 shows, our geo-distributed key-value store will reside within a set of machines (nodes within the grey cloud) running protocol-specific server-side logic (simply "server" henceforth). User applications will access this key-value store as they would a local store using the API described in Section 2.1. A user application will consist of two parts: (i) code provided by us (simply "user program" henceforth) that makes use of the key-value store's API and (ii) protocol-specific client-side logic (simply "client" henceforth) that you will implement as a library that the user code will link against. The client will create the API expected by the user and will interact with servers using RPCs.

2.1 API Provided to User Program

Your code will provide an API comprising the functions described below to the user program. See the file `client.h` for more details on these functions and the related structures.
The following two functions are for creating and destroying clients:

- `struct Client* client_instance(uint32_t id, char* protocol, struct Server_info* servers, uint32_t number_of_servers)`: This function will instantiate a client and initialize all the necessary variables. `id` is the unique identifier of the client. `protocol` can



```
#include "client.h"

int main(){
    // get an ABD or CM instane of
    // the client and do the desired
    // operations
    return 0;
}
```

```
#include "client.h"

int main(){
    // get an ABD or CM instane of
    // the client and do the desired
    // operations
    return 0;
}
```

```
#include "client.h"

int main(){
    // get an ABD or CM instane of
    // the client and do the desired
    // operations
    return 0;
}
```

```
#include "client.h"

int main(){
    // get an ABD or CM instane of
    // the client and do the desired
    // operations
    return 0;
}
```

Figure 1: Overview of your servers and user applications using your clients, and their relations. The servers will be run them within virtual machines in different Google cloud regions. The user applications will be run within PSU lab machines.

be "ABD" or "CM" indicating the type of the client. `servers` is an array of attributes of each server, and `number_of_servers` determine the number of elements in that array. It returns a pointer to the created client. It returns `NULL` if an error occurs.

- `int client_delete(struct Client* c)`: This function will destroy all the memory that might have been allocated during the client instantiation and needs to be cleaned up. It returns 0 on success, and -1 on error.

To perform operations, the user program will use the following functions:

- `int put(struct Client* c, char* key, uint32_t key_size, char* value, uint32_t value_size)`: This function will write the value specified in the variable `value` into the key specified by the variable `key`. The number of characters expected in `key` and `value` is given by `key_size` and `value_size`, respectively. It returns 0 on success, and -1 on error.
- `int get(struct Client* c, char* key, uint32_t key_size, char** value, uint32_t *value_size)`: This function will read the value of the key specified by the variable `key`. The number of characters in the key is given by `key_size`. The read value will be written to a character array pointed by the variable `value` of length `value_size`. Again here, the function also takes a pointer to a structure of Client. This is the pointer returned by the client_instance function. It returns 0 on success, and -1 on error.

Note that we consider keys and values in our key-value store to be strings. You should implement your servers such that all the keys have an initial value of string("00000"). That is, if a client tries to read the value of a key that no clients have yet written, it should receive as the value string("00000").

2.2 Client-Server Interactions and Usage Expectations

You should prepare makefiles to create server application and client object files as described below.

2.2.1 Server

A server should be started by issuing the following command:

```
./server [ip] [port] [protocol]
```

Here `ip`, `port`, and `protocol` have the obvious meaning. In particular, the above command will start the RPC service on the specified `ip` and `port` to respond to protocol-specific client requests. You will use gRPC. Note that you can only have one gRPC service since you have only one `ip` and `port`. This service will support different functions and protocol buffers for client-server communication. The basics of RPC were covered in Lecture 9. You are expected to learn about gRPC on your own. A good reference is this link. The TA will be happy to help you with difficulties you may encounter with using gRPC, so do make good use of his office hours!

2.2.2 Client

For a given user program code (which has the `main` function), your makefile should link object file of the protocol-specific client implementation to create the final user application executable.

For your convenience, a userprogram as a sample is provided for you in Canvas. This program can be executed by the following commands:

```
./userprogram ABD #To create and run operations through an ABD Client
./userprogram CM  #To create and run operations through a CM Client
```

This sample program will create 3 clients of the specified type in the argument and does three concurrent write operations and then three concurrent read operations on just one key. Please note that before running the program you should set the ip and ports of your servers in the code, and then run your servers and compile and run the userprogram. You can refer to the code implementation for more information.

3 Experimental Evaluation

You will use Google Cloud Platform (GCP) to run the server component of your code and your computer to run the client component of your code. As it is stated in the ABD algorithm, a client cannot do simultaneous operations, so to increase the request rate, you should create several instances of the client application.

To run your implementation, you should run your servers first on VMs in GCP, and then run the user application to do get or put operations using your implemented clients.

To use GCP, you need to create an account, and you can use the following commands to create or terminate virtual machines on GCP. Please install its CLI tool by using the following command:

```
sudo apt-get install gcloud
```

```
\\ Authentication, you need to run the command bellow just once.
gcloud auth login

\\ Create a project
gcloud projects create project1

\\ Set the project as your default project, also please
\\ note to connect your project to a billing account
gcloud config set project project1

\\ Create a Machine
gcloud compute instances create s5 --machine-type=e2-micro --zone=us-east1-b

\\ Please use the following regions for your servers (the first three for the
\\ experiment of 3 servers and all of them for the experiment of 5 servers):
\\ 1.us-east 2.us-central 3.europe-west 4.asia-east 5.europe-north

\\ Terminate all the machines
gcloud compute instances delete $(gcloud compute instances list --uri)

\\ Please remember to terminate your machines while you are not using them.
```

You are required to test your implementation with 3, and 5 servers, each with 10%, 50%, and 90% read_write ratio with the rate of 10 requests per second. Please set the size of values to 1KB. In each case, you should report the avarage latency of read and write operations during running the clients for 10 seconds (10 req/s * 10s = 100 requests in total).

Unfortunately, checking the linearizability of a given trace is a NP-Hard problem. However, there are some applications that attempt to show that the history is not linearizable, and if it is not the case, we can say it is linearizable with some degree of confidence. One of the simplest application for this verification is Kossos[1]. To use this application, you need to produce a trace of your implementation with the following format sample:

```
{:process 0, :type :invoke, :f :read, :value nil}
{:process 1, :type :invoke, :f :write, :value 3}
{:process 1, :type :ok, :f :write, :value 3}
{:process 0, :type :ok, :f :read, :value 3}
```

Please note that linearizability is composable. If an implementation is linearizable for one key, the whole system is linearizable. Thus, you need to feed the traces of only one key to Knossos. Each client should perform some portion of the total operations. All clients will issue requests constantly and asynchronously with respect to the other clients. This way you will ensure that system has concurrent operations. You will collect the logs accumulated at each client, append them to one common log file with the formate described above and provide that file to the knossos framework. When you run the log file with the framework, it should return a value of true if your logs are linearizable.

[1]Available at https://github.com/jepsen-io/knossos

To install Knossos, you need OpenJDK version 8. You can install it with the following command:

```
sudo apt-get install openjdk-8-jre
```

Please clone Knossos from its GitHub repository and also Leiningen from this link.

You should make "lein" script executable and put it in the directory of Knossos. Then you can verify your output traces using the following command

```
./lein run --model cas-register <log-file-name>.edn
```

You may encounter this error "Unrecognized VM option 'UnlockCommercialFeatures'". If that's the case, please comment out "-XX:+UnlockCommercialFeatures" in the file project.clj in the Knossos directory.

4 Submission and Grading

You will submit your code and a short report containing the result of your experimental evaluation and whether they match your expectations. Your code must have a Makefile for compiling. The Makefile will produce a library (named "libProject1.a") containing the Client and Server classes which our driver program can take an instance of and run them.

Your code will be evaluated by the TA using a combination of:

- The correctness of your code (50%) - we will run your code against some test cases and check the linearizability and causality of your traces.
- The use of gRPC (25%)
- Your report (25%)

5 Some final remarks

- Academic integrity policy: You are prohibited from copying any content from the Internet, sharing ideas, code, configuration, text, or anything else or getting help from anyone in or outside of the class. Consulting online sources is acceptable, but under no circumstances should anything be copied. Failure to abide by this requirement will result in the consequences described in our course syllabus at course page.
- For further information regarding the project please refer to Canvas and emails regarding the project.
- It is very important that you start early. Please seek as much help as you need from the instructor and the TA.
- Good luck and happy coding!