Programming Assignment 3: Key-Value Store with Configurable Consistency

Due: May 1st, 2018 23:59:59pm

In this assignment, you will implement a distributed key-value store that borrows designs from Dynamo and Cassandra using **Python, C/C++, Java**. For communication among different entities in the system, you can either use the Apache Thrift framework we have used in Assignment 1 or Google Protobuf in Assignment 2. This assignment is worth **13.3%** of your total score.

This is a **group assignment**. Every group can have **at most 3 students**. It is also okay if you prefer to work individually on this assignment. However, you will not receive any extra credit for working individually.

If you choose to work in a group, at least one member in the group **must** send the instructor and cc the TA an email listing the names and email addresses of the both members of your group by **April 19th, 2018** by the end of day. If we do not receive your email by this deadline, we will assume that you prefer to work individually on this assignment. **No exception will be made**.

1 Key-Value Store

Each replica server will be a key-value store. Keys are unsigned integers between 0 and 255. Values are strings. Each replica server should support the following key-value operations:

- get key – given a key, return its corresponding value
- put key value – if the key does not already exist, create a new key-value pair; otherwise, update the key to the new value

For simplicity, each replica only needs to store key-value pairs in its memory. That is, there is no need to flush the memory content to persistent storage.

As with Cassandra, to handle a write request, the replica must first log this write in a write-ahead log on persistent storage before updating its in-memory data structure. In this way, if a replica failed and restarted, it can restore its memory state by replaying the disk log.

2 Configurable Consistency

Your distributed key-value store will include **four replicas**. Each replica server is pre-configured with information about all other replicas. Keys are assigned to replica servers using a partitioner similar to the `ByteOrderedPartitioner` in Cassandra. Each replica server is expected to be assigned equal portions of the key space. The replication factor will be **3** – every key-value pair should be stored on three out of four replicas. Three replicas are selected as follows: the first replica is determined by the partitioner, and the second and third replicas are determined by going clockwise on the partitioner ring.

Every client request (get or put) is handled by a coordinator. Client can select any replica server as the coordinator. Therefore, any replica can be a coordinator.

Consistency level. Similar to Cassandra, consistency level is configured by the client. When issuing a request, put or get, the client explicitly specifies the desired consistency level: `ONE` or `QUORUM`. For example, receiving a write request with consistency level `QUORUM`, the coordinator will send the request to all replicas for a key (may or may not include itself). It will respond successful to the client once the write has been written to quorum replicas – i.e., two in our setup. For a read request with consistency level `QUORUM`, the coordinator will return the most recent data from two replicas. To support this operation, when handling a write request, the coordinator should record the time at which the request was received and include this as a timestamp when contacting replica servers for writing.

When the consistency level is set to `QUORUM` during both get and put, we have strong consistency. However, this is not the case when the client uses `ONE`. With `ONE` consistency level, different replicas may be inconsistent. For example, due to failure, a replica misses one write for a key k. When it recovers, it replays its log to restore its memory state. When a read request for key k comes next, it returns its own version of the value, which is inconsistent. To ensure that all replicas eventually become consistent, you will implement the following two procedures, and your key-value store will be **configured to use either of the two**.

Read repair. When handling read requests, the coordinator contacts all replicas. If it finds inconsistent data, it will perform "read repair" in the background.

Hinted handoff. During write, the coordinator tries to write to all replicas. As long as enough replicas have succeeded, `ONE` or `QUORUM`, it will respond successful to the client. However, if not all replicas succeeded, e.g., two have succeeded but one replica server has failed, the coordinator would store a "hint" locally. If at a later time the failed server has recovered, it might be selected as coordinator for another client's request. This will allow other replica servers that have stored "hints" for it to know it has recovered and send over the stored hints.

If not enough replicas of a key are available, e.g., consistency level is configured to `QUORUM`, but only one replica of the key is available, then the coordinator should return an exception to the issuing client. NOTE that this is different from the "sloppy quorum" in Dynamo.

3 Client

You should also implement a client that issues a stream of get and put requests to the key-value store. Once started, the client should **act as a console**, allowing users to issue a stream of requests. The client selects one replica server as the coordinator for all its requests. That is, all requests from a single client are handled by the same coordinator. You should be able to launch multiple clients, potentially issue requests to different coordinators at the same time.

4 Demonstration

After the submission deadline, every group will sign up for a demonstration time slot with the TA. All group members have to be present during the demonstration. You will show the capabilities of the key-value store that your group has implemented.

5 How to submit

To submit the assignment, you should first create a directory containing your BU email IDs. For example, if a group has two students, and their email IDs are `jdoe@binghamton.edu` and `jsmith@binghamton.edu`, you should create a directory called `jdoe-jsmith-p3`.

You should put the following files into this directory:

1. Your source code.
2. A `Makefile` to compile your source code into executables.

3. (Optional) A `Readme` file if there is anything you want the TA to be aware of when grading.
4. A `Task` file, listing the tasks each group member worked on in this assignment.
5. Multiple `STATEMENT` files, signed by each group member individually, containing the following statement:

   "I have done this assignment completely on my own and in collaboration with my partners. I have not copied my portion of the assignment, nor have I given the project solution to anyone else. I understand that if I am involved in plagiarism or cheating I will have to sign an official form that I have cheated and that this form will be stored in my official university record. I also understand that I will receive a grade of **0** for the involved assignment and my grade will be reduced by one level (e.g., from A to A- or from B+ to B) for my first offense, and that I will receive a grade of **"F" for the course** for any additional offense of any kind."

Compress the directory (e.g., `tar czvf jdoe-jsmith-p3.tgz jdoe-jsmith-p3`) and submit the tarball (in `tar.gz` or `tgz` format) to myCourses. You should name your submission: `jdoe-jsmith-p3.tar.gz` or `jdoe-jsmith-p3.tgz`. Failure to use the right file name will result in a 5% point deduction.

Your assignment will be graded on the CS Department computers `remote.cs.binghamton.edu`. It is your responsibility to make sure that your code compiles and runs correctly on these remoteXX computers.

Your assignment must be your original work. We will use MOSS[1] to detect plagiarism in programming assignments.

[1] `https://theory.stanford.edu/~aiken/moss/`