Programming Assignment 3: Replicated Key-Value Store with Configurable Consistency

Due: December 2nd, 2020 23:59:59pm

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

This is a **group assignment**. Every group can have **at most 2 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 an email listing the names, email addresses, and Github IDs of the both members of your group by **November 18, 2020** end of day.

If you choose to work individually on this assignment, please still send the instructor an email with your Github ID so that a repository can be created for you.

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 Replicated Key-Value Store

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. A 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, upon receiving a

write request with consistency level `QUORUM`, the coordinator will send the request to all replicas of 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.

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. If the consistency level is configured to `ONE`, but none of the key's three replicas are available, the coordinator should also return an exception.

With `ONE` consistency level, a replica may miss write operations. 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 out-of-date.

To ensure that all replicas become consistent, you will implement the hinted handoff mechanism as described below:

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 will send a message to all replicas. This will allow other replica servers that have stored "hints" for it to know it has recovered and send over the stored hints.

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 arbitrary 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 Demo

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 Github repository

Once you have formed your group, please email the instruct about your group. Please include in your email both group members' names, BU email addresses, and Github IDs. An empty group repository will be setup on Github, allowing both group members to make commits.

To add a file to the next commit, use the `git add` command. To commit your code, use the `git commit` command. Be sure to also push your commit to the Github repository after every commit using the `git push` command.

We expect each repository to have **at least five commits**, with the first one and the last one **more than 72 hours apart**. **This requirement will be strictly enforced. Submissions that do not meet the five commits / 72 hours requirement will not be accepted or graded.**

6 How to submit

To submit, commit and push your latest code to the private Github repository. Your commit should contain the following files:

1. Your source code.
2. A `Makefile` to compile your source code into executables.
3. A `Readme` file describing:
   - how to compile and run your code on `remote.cs.binghamton.edu` computers,
   - the tasks both group members worked on in this assignment,
   - completion status of the assignment, e.g., what has been implemented and tested, what has not,
   - anything else that might be helpful.
4. Two `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 partner. 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."

After pushing your final commit to the Github repository, please submit your **commit hash** to myCourses. This helps us know your submission is ready for grading and which of your commits we should grade. We will not grade your assignment unless you have submitted the commit hash to myCourses before the assignment submission deadline.

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/`