Luna Szymanski
6650 Project 3 - Replicated Key/Value Store using 2PC

How to Compile and Test:

Included Files:
jars/ → contains the jar files
src/ → contains .java files used for creating the jar files in jars/
terminal logs/ → Client.Terminal.Log.txt is the log of the client.
→ Coordinator.Terminal.Log.txt is log of coordinator (servers)

Compilation Instructions:

1. Open Two Terminals and go to unzipped proj3 directory in terminal
2. Terminal 1 (Coordinator) type `java -jar jars/coordinator.jar P1 P2 P3 P4 P5`
    a. P1-P5 are 5 port numbers for the servers to run on
    b. Wait for "All servers Ready!" message in Terminal 1
3. Next in Terminal 2(Client) type `java -jar jars/client.jar P1 CMND KEY VAL`
    a. P1 - port to connect to
    b. CMND - 0 for put, 1 for get, 2 for delete
    c. KEY - the key of the key/value pair
    d. VALUE - the value of the key/value pair
4. Now you have a running coordinator in Terminal 1, and in Terminal 2 you can make any combination of commands here for testing. You can also run additional terminals that run clients in the same way as step 3
5. Please see my terminal logs for my client and coordinator in terminal logs/
    a. Client makes 5 puts to one server, then 5 gets to a second server, and then 5 deletes to a third server. You can clearly see the servers interact to make sure every server has a perfect replica of every other server
    b. Coordinator logs show the preparation step of 2PC, before every write command the servers prepares to make sure everything is good to go and after all the servers are prepared (no less and no more or it will abort) then it does the go phase of the commit where it makes the changes to all 5 servers at the same time.

Assignment Overview

The objective of this assignment was to create a replicated key/value store that uses 2 Phase Commit. In other words I built a system of 5 servers, all of which replicated the same data. No matter which server received a transaction request, all 5 servers would uniformly change. In addition I built a two phase commit process, and used the preparation phase to make sure all of my servers were active and responsive. I think it was out of scope to be checking the contents of each server at each write call to make sure that before the request was accepted the data was uniform across all 5 servers. However this would help protect against

corruption and other failures. Also I did not build an independent failure detection system to make sure the servers were live, and I did not include support for changing the number of active servers, though I think having a dynamic server count would be more effective as it could be adjusted based on costs or bandwidth needs.

Technical Impressions

One of my major design decisions was to have my Coordinator class do the majority of the work. My idea was that anytime a server got a write command, it would call the coordinator class to enact the change to all 5 servers at once. I used a two phase commit algorithm, where first the servers all prepare to commit, and only after every server is ready does the algorithm continue to the actual commit. I decided to use the preparation phase to also handle aborting a client request, and the condition I thought made the most sense was when the server count was not equal to 5. If there are less servers then one was dropped, if there are more than 5 then something else is wrong and an abort is also appropriate.

Because I could have multiple clients making requests I used a boolean busy flag to indicate when the 2PC begins and ends. When 2PC begins my code starts by telling the servers to prepare, at which point it only continues if the resources are not being used. As soon as it is allowed to access the resources it marks the busy flag so a basic Mutex is established.

I stored my key/value map in my RMIServer class, because I needed a unique map for every server. Initially I tried keeping the map in DictImpl but because it is an extension of an interface the map was not duplicated and was essentially just one map with 5 access points. Moving it to my server class solved this issue.

Additional Information

Workflow:

- PUT/DEL → Client sends request to a server. 2 Phase commit begins. First Preparation phase during which server contacts the coordinator class to make sure all servers are prepared to commit. When all servers are prepared to commit, the second “Go” phase begins. The server communicates with the coordinator to fulfill the commit.
- GET → Get is not a writing function, it is read only. So it is not part of the 2PC process as no data is being committed. In this case the server simple fulfills the get request from its local map.

Abort Scenario:

- I check for an abort scenario only when I get a write command (PUT/DEL), during the preparation phase of the 2PC. As the servers report their preparation status, I keep count of the servers. If there are not exactly 5 servers then I call an abort on that one request.