Distributed Key-Value Store Report

Chia-Chen Hsu, UT EID:ch43899, UTCS id: cchsu
Xinrui, UT EID:xh3426, UTCS id: xh3426
Fu Li, UT EID: lf9397, UTCS id: fuli2015

1. The description of our protocol

Server

Each client maintains two important variables:

1. server.vclock, the vector clock of the server. The increment method makes the server's logical clock catch up with the fastest clock and increments it by one. For vector clocks from servers, the merge method updates server's vector by taking the maximum of the value in its own vector clock and the vector for merging. For vector clocks from clients, the merge method only updates the client logical clock in server's vector.
2. server.writeLog, the write log of all the put operations. The total order of these operation is dertermined by (serverTime, serverId), which is unique with each other. The write log is sorted in the total order.

Each client has two operations:

1. server.put
    a. receive (key, value, vclock) from client
    b. merge server.vclock with vclock and increase server.vclock
    c. append (serverTime, serverId, (key, value)) to server.writeLog
    d. send (server.vclock, (serverTime, serverId)) to client
2. server.get
    a. receive (key, vclock, (serverTime, serverId)) from client
    b. merge server.vclock with vclock and increase server.vclock
    c. search the key in the bottom-up way and compare the two total orders
    d. send (server.vclock, value, (serverTime, serverId)) to client
3. server.antiEntropy
    a. receive (writeLog, vclock) from server
    b. merge server.vclock with vclock
    c. merge server.writeLog with writeLog in the total order

Client

Each client maintains two important variables:

1. client.vclock, the vector clock of this client. The increment method makes the client's logical clock catch up with the fastest clock and increments it by one. The merge method updates each element in client's vector by taking the maximum of the value in its own vector clock and the value in the vector for merging.
2. client.lastOpDict, the dictionary of total order of the last operation performed on the key. Each key corresponds to an order (serverTime, serverId).

Each client has two operations:

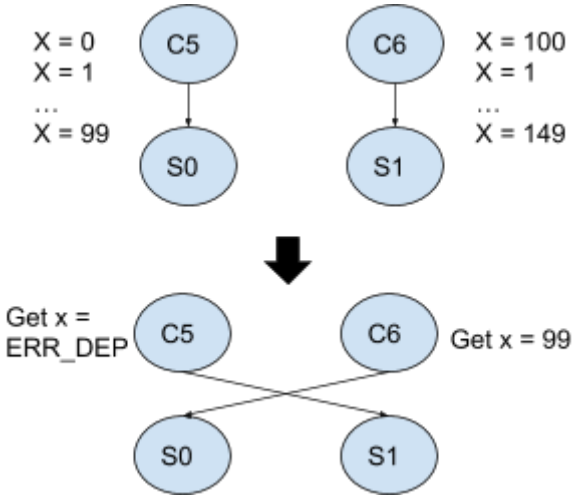
1. client.put

    a. increase client.vclock
    b. send (key, value, client.vclock) to server
    c. receive (vclock, serverTime, serverId) from server
    d. merge client.vclock with vclock and update client.lastOpDict[key] by (serverTime, serverId)
2. client.get
    a. send (key, client.vclock, client.lastOpDict[key]) to server
    b. receive (vclock, value, (serverTime, serverId)) from server
    c. merge client.vclock with vclock and update client.lastOpDict[key] by (serverTime, serverId)
    d. return value

2. The description of our tests

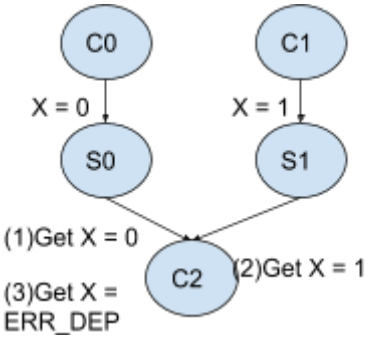
Our test involves two part: Correctness and Performance. As for correctness, we perform unit test for each API to ensure the functionality of each API. We also demonstrate sample examples for testing three properties-- Eventual consistency, read-after-write, and Monotonic Reads. Please check testing.py file to see all our test cases. Here we introduces these three properties cases:

1. Eventual Consistency. We generate 10k random key-value pairs and put them randomly into 5 servers. After calling stabilize(), we compare the key/value info within each servers and the values of every keys should be the same. Then, we separate the fully-connected servers by isolating server 0, following by connecting it back and do stabilize and comparing the value again. This test cases also demonstrate that we could handle large scale operations.
2. Read-your-write. We design scenario that 2 clients connected to separate servers and perform some put operations to same key. Then, switch their connection and check the get value of the key. If read-your-write holds, we expect to get one latest value (depends on our total order) and one ERR_DEP.



3. Monotonic Reads. We have 2 clients put values We generate 10k key-value pairs and put them into 5 servers. The value for each key maintains monotonic order. We

perform get operation during the every put ops and check if the get also maintain monotonic order.



4. Test on different topology. We use break connections/create connections to separate servers into several graphs. By following with stabilize(), we can check the key/value pair within each connected graph and see the consistency within each graph and the discrepancy between graphs.

As for performance, we evaluate the frequency for APIs by tests as follows:
We tests on 2 set of data:

a. 100 puts for single key * 5 servers = 500 ops
b. 100 puts for 100 keys * 5 servers = 500 ops

The table below demonstrate our results.This results shows that our system could perform up to 1900 instructions per second. Also, when the system need to exchange 500 distinct keys, it need less than 5 secs for stabilization.

Senario	put	get	stabilize
A	0.2ms	3ms	0.18 sec
B	0.3ms	1.1ms	4.16 sec

3. Instructions on how to use your system

1. Directly input the API command in the testing.py file, like joinServer(2) or stabilize().
2. Put your commands in a file, name it by testCommands.txt, and run master.py, with the testCommands.txt file as argument.
   ```
   python master.py command.txt
   ```

The server ids are in the range {0,1,2,3,4} and the cliens ids in {5,6,7,8,9}.

4.Other Information

Compared with Bayou system, the advantage of our system is that we allow clients can write on any server without access control.