Bayou Project

CS 380D Due December 1, 2015

1 Introduction

In this project you will implement an eventually consistent key-value store in the spirit of Bayou. The key-value store will contain a playlist with each entry consisting of <songName, URL>where songName is the name of the song and URL is the URL of the song. This system will consist of clients and servers. Servers will store the playlist and perform updates when asked by the clients. Clients will be able to perform the following operations:

- **Put** a new song to the playlist or update the value if present
- **Get** the URL of a song in the playlist
- **Delete** an existing song from the playlist

The Bayou protocol will make the playlist eventually consistent. Note that the system's membership is not static: nodes can freely join or leave the system. Of course, when nodes leave the system, they should follow the "retirement" protocol and when nodes reenter, they should be brought up to date.

2 Technical Details

The implementation can be written in any language or languages you like. The two requirements are that no two client or server processes can share memory and all processes be able to run concurrently. Other than that, the rest is up to you. We will not provide any starter Bayou implementation code.

2.1 Conflict Resolution

In the case where many conflicting puts or deletes are made, these conflicts will be resolved by the order the primary replica receives and marks them. The primary replica must be the first server in the system and if the primary replica retires, it should hand off its duties to the server it informs of its retirement. If the primary is not accessible (for instance when it is in a separate partition), nodes should resolve conflicts by the ordering of their names. Servers should follow the recursive naming protocol specified in the section describing the Creation and Retirement Writes in the Bayou paper. Please do not use the id's used to start servers as their ids. These only serve to make testing and startup easier.

3 Test cases

Below are several test cases that your implementation should handle correctly. This is **not** an exhaustive list of all the tests we will be running—but you can count on the fact we will run at least these.

- As nodes create updates and communicate with each other, it should be possible to "freeze" the execution and examine the nodes' logs. Logs should be consistent: every stable log should be a prefix of the longest stable log. Moreover, the tentative logs should be in the correct order (and of course the correct mechanism for transitioning updates from tentative to stable should be in place).
- The system should guarantee eventual consistency. After having created and exchanged updates for a while, nodes should stop creating updates and just exchange them with one another. Eventually it should be possible to verify that all updates have become stable and all logs are identical. If the log is incapable of becomming stable, you should ensure that the order of the tentative logs for nodes in the same partition converges to the same ordering.

- The system should guarantee the four session guarantees ("Read Your Writes", "Writes Follow Reads", "Monotonic Reads", and "Monotonic Writes") to its clients. If a client tries to perform a GET operation when one of the read properties is not satisfied, the replica that doesn't satisfy the client's dependencies should return "ERR_DEP" (more on this below). If a client tries to perform a write operation when one of the write session guarantees is not satisfied, the write should be dropped by the replica.
- The system should handle nodes that "gracefully" leave the system by running the "retirement" protocol. Here is a scenario you should handle. Say that node R that intends to retire. You should make R communicate with another node (A) by gossiping a number of updates that R has just issued. Then R should retires by writing to itself a "retirement" update. R then should talk to another node, say B, to whom it gives all of its updates, including the "retirement" update. Upon processing the retirement update, B should delete R from its timestamp vector. So far, so good. Now, for the kicker: right after deleting R, B should gossip with A and receive any update that A has. Your protocol should handle R's retirement correctly: in particular, B should not end up re-adding R to its timestamp vector as a result of communicating with A.

3.1 The Master Program

To help you test your implementation and assist with the evaluation, each submission must include a master program which will provide a programmatic interface with the playlist. The master program will keep track of and will also send command messages to all servers and clients. More specifically, the master program will read a sequence of newline delineated commands from standard input ending with EOF which will interact with the playlist and, when instructed to, will display output from the playlist to standard out. We will be using this master program to test your implementation of Bayou, not to trick you up on API edge cases. When you turn in the master program, please also turn in a one line file that contains the command to run your master program. The API for the master program is defined on the following page. All ids mentioned in the API are in the same namespace. If for example there are 3 servers and 2 clients, the ids 0 - 4 would be handed out.

3.2 Clean Up (everybody do your share...)

An impatient user, change of plans, or unforeseen bug in your code may cause some number of processes in your system to crash, while leaving others up and running. To allow us to move smoothly between different test cases, please (a.k.a. you must :-)) provide some kind of script (automated process) to clean up such orphaned processes.

In addition, your processes should be respectful of the world in which they live. For example, they should not produce massive log files when run. If you find it necessary to produce log files or debug output, please provide a flag to turn this output on; have it off by default.

3.3 Master Program API Specification

Command	Summary
joinServer [id]	This commands starts a server and will connect this server to all other servers in the system.
retireServer [id]	This command will gracefully retire a server from the system. This command should block until it is able to tell another server of its retirement.
joinClient [clientId] [serverId]	This command will start a client and connect the client to the specified server.
breakConnection [id1] [id2]	This command will break the connection between a client and a server or between two servers
restoreConnection [id1] [id2]	This command will restore the connection between a client and a server or between two servers
pause	This command will pause the system and not allow any Anti-Entropy messages to propagate through the system
start	This command will resume the system and allow Anti-Entropy messages to propagate through the system
stabilize	This command will block until there are enough Anti-Entropy messages for all values to propagate to all possible servers. In general, the time that this function blocks for should increase linearly with the number of servers in the system.
printLog [id]	This command will print out a server's operation log in the format described below.
put [clientId] [songName] [URL]	This command will tell a client to associate the given URL with the songName. This command should block until the client communicates with one server.
get [clientId] [songName]	This command will tell a client to attempt to get the URL associated with the given songName. The value or error returned should be printed to standard-out in the format specified below. This command should block until the client communicates with a server and the master script
delete [clientId] [songName]	This command will tell a client to delete the songName and associated URL in the playlist. This command should block until the client communicates with a server

4 Print Format

4.1 Log format

The "printLog" operation instructs the server to print out it's current view of the log. Each log entry should be on a separate line sorted by oldest entries printed first. Each entry should be printed in the following format:

OP_TYPE:(OP_VALUE):STABLE_BOOL

"OP_TYPE" is either "PUT", or "DELETE". "OP_VALUE" is "songName, URL" for put operations and "songName" for deletes (wrapped in parentheses). "STABLE_BOOL" is "TRUE" if the operation is stable or "FALSE" if not.

4.2 Get format

The "get" operation instructs the client to print out the value for a songName that the server that it is connected to currently has. The value returned by the server should be formatted as follows:

songName:URL

Where URL is the URL associated with the songName. In the case that the server does not have an entry for the key, URL instead should be: "ERR_KEY". In the case that the server that the client is trying to connect to does not satisfy the client's dependencies, URL instead should be: "ERR_DEP".

5 What to Turn In

1. Source code for your implementation of Bayou
2. A makefile that compiles your implementation on the CS Machines (if necessary)
3. A master program and a file named "COMMAND" that contains on one line the command to run your master program
4. A README file that details your implementation of Bayou, your name(s), uteid(s), and utcs id(s), major design decisions, instructions on how to use Bayou by hand, as well as any other information you think is relevant to the grading of your service

Skeleton code for the master program, example COMMAND files, and sample test cases will be provided.