Project 1: Distributed Key-Value Store

1. Introduction

In this project you will implement an eventually consistent key-value store. Each entry in the key-value store will be a pair of binary strings. The system will consist of clients and servers. Servers will store the data and perform updates when asked by the clients. Clients will be able to perform the following operations:

- **Put** a new entry into the store.
- **Get** the value associated with a key.

You will design a protocol that guarantees eventual consistency plus two session guarantees: *Read Your Writes* and *Monotonic Read*s. Eventual consistency means that eventually all clients will see all writes, but no other guarantees are provided.

- Read Your Writes - If a client has written a value to a key, it will never read an older value.
- Monotonic Reads - If a client has read a value, it will never read an older value.

Eventual consistency is usually implemented through gossip - servers periodically exchanging updates with other servers and bringing their logs to a consistent state. We will discuss consistency models in the first weeks of class, and you can read the UW slides on consistency and the book chapter on replication for more information about consistency and session guarantees, both linked on the class schedule. The papers under week 4 (PNUTS, Consistency at Facebook, and Bayou) detail real-world implementations of systems providing various forms of consistency.

1.1 Logistics

This project is due on March 2. You should work in groups of 2-3 people.

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.

3. Performance

Part of your grade will be a performance component. We will benchmark your implementation and you will receive points based on how many requests you can handle per second. This section is worth 20 points out of the total 100. We will test 5 clients, each connected to one of 5

servers. The clients will rotate issuing put requests, then we will call ‘stabilize’ at the end. Points will be awarded as follows:

- 5 points for 20 requests/second
- 10 points for 30 requests/second
- 20 points (full credit) for 40 requests/second

Since there will be variability in the results, we’ll use the median of five runs.

4. Testing

You should ensure that your system is able to pass correctness tests. Below are a couple of cases that you can be sure we will run, though this isn’t an exhaustive list.

- 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 propagated to all nodes and the key-value stores at all servers should be identical. If it is impossible for the stores to fully become identical, you should ensure that the stores for nodes in the same partition converge to the same state. A system with 5 servers shouldn’t take longer than 5 seconds to converge.
- The system should guarantee the two session guarantees (*Read Your Writes* and *Monotonic Reads*) to its clients. If a client tries to perform a GET operation when one of the properties isn’t satisfied, the replica that doesn’t satisfy the client’s dependencies should return “ERR_DEP”.

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 key-value store. 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 key-value store and, when instructed to, will display output from the playlist to standard out. We will be using this master program to test your implementation, not to trick you up on API edge cases. Your Makefile for the project should compile this to an executable named “master”. If your implementation doesn’t require a Makefile, turn in an empty file. The API for the master program is defined on the next 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

An impatient user, change of plans, or unforseen 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 provide some kind of script to clean up such orphaned processes. This should kill all processes that are still alive in your system.

3.4 Master Program API Specification

Command	Summary
joinServer [id]	This command starts a server and will connect this server to all other servers in the system
killServer [id]	This command immediately kills a server. It should block until the server is stopped.
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.
createConnection [id1] [id2]	This command will create or restore the connection between a client and a server or between two servers.
stabilize	This command will block until all values are able to propagate to all connected servers. This should block for a max of 5 seconds for a system with 5 servers.
printStore [id]	This command will print out a server's key-value store in the format described below.
put [clientId] [key] [value]	This command will tell a client to associate the given value with the key. This command should block until the client communicates with one server.
get [clientId] [key]	This command will tell a client to attempt to get the key associated with the given value. 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.

3.5 Print Format

The “printStore” and “get” operations instruct the process to print out key-value pairs. These should be printed in the format:

key:value

For a “get” operation, if the server doesn’t have a value for the key, the value 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, the value should instead be: “ERR_DEP”.

5. What to Turn In

One member of your group should submit everything to Canvas as a zip file.

1. Source code for your implementation (client, server, and master)
2. A makefile that compiles your implementation on the CS machines (can be empty)
3. A README file that details your implementation, your names, UT EIDs, and UTCS ids, a description of your protocol, a description of your tests, instructions on how to use your system, and any other information you think is relevant to the grading of your service.

Thanks to Dr. Lorenzo Alvisi and Dr. Michael Swift whose projects influenced the creation of this document.