CPSC 426 Final Project: Using Amazon Dynamo's Design to Increase Availability of Sharded Key-Value Store

Joonhee Park
Yale University

1 Introduction

In the CPSC 426 Lab 4 implementation of a sharded Key-value store (which administered Get, Set, and Delete APIs), when the client would store a particular Key-value in the system, the client would perform Set calls to all the nodes that host this key (i.e. all the nodes that host the shard that they key maps to). If any of the Set calls failed, then the client would return an error. Additionally, there were no consistency guarantees; writes could fail partially, which means that multiple Get calls to the same key could return different values. For my final project, using the principles of sloppy quorum, hinted handoff, and vector clock conflict resolution (from Amazon's Dynamo Paper), I expanded upon Lab 4 to build a sharded Key-value store with greater write availability and, in the case where replicas of a particular key-value diverge across servers, designed a conflict resolution strategy.

2 Design

As alluded to previously, in Lab 4, the client was responsible for fanning out Set calls to all the nodes that hosts a particular key; in other words, the client was responsible for replicating a key-value to all N servers that it mapped to without any form of "backup." Instead of this, the sloppy quorum design dictates that one "coordinator node" is responsible for replicating to the first N "healthy" servers. In my implementation, the client serves as the coordinator node, and if a Set call to a server fails, then the client connects to one of the nodes that the key is not originally mapped to and places the key-value in a temporary storage in the healthy server using a different, TemporarySet API.

I chose the client to be my coordinator to minimize the number of server connections in the worst case. More specifically, if my design had a coordinator node assigned to each shard, but connection to the coordinator node failed, then I would have to assign a new coordinator node among the other nodes that host this shard (and check again if connection succeeds). Then, in the case where all coordinator nodes fail except the last one, this last server would have to retry connection to all other nodes that host this shard. My design skips the part of finding a valid coordinator node by making the client the coordinator. Of course, one possible drawback to this design is if the client can establish a valid connection to only one server, but all of the servers are able to communicate with each other. In the case where a key-value maps to a set of servers including one of the healthy ones, my design would likely fail, since the client will be able to replicate the key-value on only one server (and there are zero other healhty nodes that the client can temporarily store on). However, for a different design, it's easy to see that if the client can establish connection to the one healthy node and assign it the coordinator role, that node can successfully replicate the key-value to all other nodes, resulting in a successful Set call.

Each server has a regular key-value storage for regular Set calls. My design added another distinct key-value storage specifically for calls to TemporarySet. Using a garbage collector strategy, the server would periodically go through this storage of temporary values. For each key-value in the storage, the server tries to connect to the server that the key-value originally

should have been replicated to, and upon successful connection and replication to the original server, the key-value is removed from the temporary store. This is inspired from the hinted-handoff technique in the Dynamo paper.

Lastly, my system resolves all versions of an object (from different servers that replicate it) using vector clocks. Each key-value is associated with a list of (node, counter) pairs, where node denotes the server that has Set this value and counter denotes the version history. When a new key is introduced to the system, each node that is assigned to it initializes a vector clock with a list of length N (the number of servers), with each node having a counter of 0. When the node (say n1) assigns a value to this key, it increments n1's counter by 1.

In Lab 4, when the client calls Get on a key, it connects to a random server assigned to the key. In my impelementation, the client calls Get to all of the servers assigned to the key and adds all the values (with their associated vector clock) to a list. The client then goes through this list and removes all values that are ancestors according to their vector clocks. More specifically, the Dynamo paper cites that if comparing two objects and "the counterson the first object's clock are less-than-or-equal to all of the nodes in the second clock, then the first is an ancestor of the second and can be forgotten." Then, from the list of concurrent values, the client chooses an arbitrary values and returns it.

An improvement upon this implementation could take better advantage of vector clocks, where the client's Get API would be updated to return a list of concurrent values instead of a single one. Then, the application built on top of the service could, for example, merge these values (i.e. a shopping-cart-like application could merge conflicting values into one, unified shopping cart). Due to limited time and the complexities of rewriting the test-setup and stress-test files, my implementation does not explore these possibilities.

3 Implementation

The API used to implement the sloppy quorum and hinted handoff strategies is TmpSet. It takes in the same inputs as the regular Set (key, value, time-to-live) as well as a string denoting the original node that the key-value belongs to. For each server, the regular storage is compartmentalized by shard; for the temporary storage, it is partitioned by node. When TmpSet is called, the server places the key-value in the temporary storage with respect to the original node it belongs to. Then, using a Ticker that fires every 2 seconds, the server goes through the temporary storage (by node) and attempts to replicate each temporarily-stored key-value on its original node.

To make it easier for the client (a.k.a. the coordinator) to find other nodes to temporarily store a value in the case of server connection failure, I added another function called NodesNotForShard, which returns all the nodes that a key (which is mapped to a shard) is not assigned to. If replication to one server fails, the client then chooses a node from NodesNotForShard to temporarily store the key-value, and upon success, removes the node it temporarily stored to from NodesNotForShard. Since the client replicates concurrently, my implementation updates NodesNotForShard in a thread-safe manner using Golang mutexes; otherwise, two different threads could call TmpSet to the same node in NodesNotForShard, which would fail to maintain our goal of replicating to the first N healthy, unique nodes.

For Get calls, as previously mentioned, the client collects all values for a key that is replicated across multiple nodes. To reconcile potentially different versions, the client removes stale values by assessing their vector clocks using three functions: isAncestor, isEqual, and isConcurrent. All three functions compare two value's vector clocks. isAncestor returns true if the first's vector clock has counters that are all less than or equal to the second vector clock's counters. isEqual returns true if all counters across the two clocks are equal. isConcurrent returns true if two values are conflicting (there exists a counter that is higher in the first clock and another counter that is higher in the second clock). The algorithm I im-

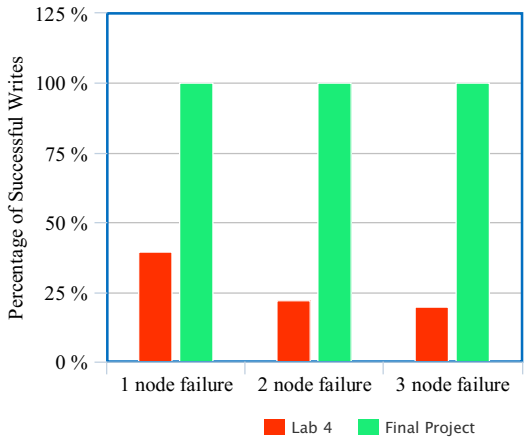

Figure 1: Write Availability with/out sloppy quorum

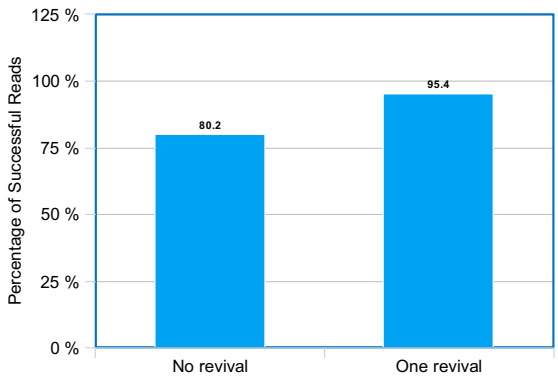

Figure 2: Read Availability with/out hinted handoff

plemented works as such: the client iterates through the list of values and adds the first value to a new list called updatedResponses. If a value's clock is concurrent with or equal to a clock in updatedResponses, we add it to updatedResponses. Otherwise, if a value's clock is not concurrent with or equal to a clock in updatedResponses, and it is not an ancestor of a clock in updatedResponses, we clear updatedResponses and add the value to it.

After the client obtains a list of the most updated responses, it randomly chooses an index within it and returns that value.

4 Demonstration

All tests were performed on a 2020 MacBook Pro with 1.4 GHz Quad-Core Intel Core i5 processor and 16 GB of memory.

Using a five-node cluster with ten shards scattered across five nodes (with no shard being assigned to more than three nodes), Figure 1 compares the write-availability (percentage of successful Sets) between my Lab 4 and final project implementations. Stress tests were performed for a write-heavy workload (1 Get QPS and 1000 Set QPS), in the cases where 1, 2, or 3 nodes were unavailable. Evidently, the final project had complete write-availability in the case of multiple node failures, while the Lab 4 KV-store had much less availability when faced with similar circumstances.

It is important to note, however, that at the 3-node failure mark, the number of successful Get calls reduced for the final project implementation. This is because not all shards are covered, resulting in some shards having zero nodes that the client can connect to. While the system cannot do anything about some keys that cannot connect to any server that it is stored on, we do bring a node "up-to-date" if it is revived using the hinted-handoff strategy. Figure 2 compares the read-availability of the final project implementation when 3 nodes are down the whole way through versus when 3 nodes are down but one of them is revived mid-test. The higher percentage of successful gets demonstrates the effectiveness of hinted handoff, where if a failing node (say n1) is brought back to life, healthy nodes that temporarily stored n1's key-value pairs will update n1 once it detects that n1 is available again.

The functionality of each of these techniques (as well as conflict resolution using vector clocks) is further explored in the test cases provided in kv/test/sloppy_test.go.

5 Related Work

[1] Giuseppe DeCandia et al. Dynamo: Amazon's Highly Available Key-value Store. SOSP 2007:205-220