A Disibuted Key-Value-Store coforming to an AP system

Gabriel Velazquez & Chris Troiano
Shayan Farmanidiznab

June 10, 2016

Our distributed key value store characterizes a primary back up system, where the back ups are still accessible by the client.

Our setup consists of using.

1. Python
2. Flask, a micro framework that handles routing our functions to HTTP
3. Python library Requests

1 Overview

Our setup should satisfy all the requirements of a single-copy consistent storage in the situation that the primary/master node (detailed below) does not crash. This means that in the aforementioned situation, the system should return the same responses as required from ASG2 specs. Our distributed KVS emphasizes and chooses availability over consistency.

Because each node can be contacted by the client, in order to ensure consistency, our model's core backbone relies on keeping 'trust' in a primary / master node. Our master was selected when each node is spun up, automcatically choosing the node with the lowest port umber, the port # being used as the node ID #. This means that whenever a non-master node gets a request from the client, the non-master node acts as an intermediary, and contacts the master node for the appropriate data. We still ensure that each node gets updated with the latest values from PUT, as a GET from the master-node will always update its local KVS. In the case that the master node cannot be contacted, we ensure that the client still gets a response, just from the local KVS, which may not be up to date.

In the case of delete, we read the response, status-code, from the master node on whether the key existed or not. This is to not just properly return the data and correctly adjust the local KVS, but to also get the correct response from the local KVS should the master no longer be alive. This should add a little but more consistency.

We chose this set up, as it allowed us to not have to use clocks. It also allowed us to deal with less infinite responses/loops with a broadcast function. We only ever have to talk to the master which makes inter-node communcation easy. Dealing with setting up Docker, and especially inter-node communication prevent us from having the time to prepare a more fault-tolerant distributed KVS (when the primary/master fails). **Our KVS is thus an AP system.**

2 Details

To ensure that the correct responses would always be delivered, we created 'pure'/ purely 'functional' kvs functions that returned exactly was specified from ASG2. These were moved outside the KVS function called by client from '/kvs'. '/kvs' now initiates test cases deciding if a response came from a client, the

master/primary node, or a non-master/non-primary node. If the request is from a user, and it is a PUT request, it immediately attempts to send the same PUT request to the masternode. The same process is used for GET and DELETE. When a non-primary node is contacted by the client, it immediately asks the master-node for the appropriate request. Thus, any PUT to the KVS to any node will thus update the master, and because every GET firstly talks to the master, it will return the latest response.

2.1 Consistency

If the master fails, the system becomes only as consistent for a certain value as when it was the last time a backup node contacted the master with a GET (or a PUT), as each GET request will also update the local KVS if a key/value is found from the master. Thus each backup does become a consistent backup whenever GET requests are sent. This creates less

2.2 Bandwidth/Throughout

Our system enjoys high throughput and little bandwidth, as our primary/backup model only becomes consistent when a client contacts the system/indivual node, eg, meaning, *to the client*, it resembles a single-copy KVS. This is due to no broadcast, and only contacting the master when required.

3 Broadcast

The broadcast function took much longer than we initially thought it was going to take. The first route that was suggested by fellow classmates, was to use threading. This seemed to take care of the problem except that we couldn't figure out how to stop the broadcast from propagating. Every time that a put request would come through, it would broadcast that same put request to everyone else in the network. An infinite loop of message passing would ensue. We tried using event hooks to issue a put request to the rest of the members upon receiving a put request. This obviously left us with the same problem of an infinite message passing loop.

After wrestling with this problem for a few days, I spotted Nikihil walking around campus and stopped him to ask him about this issue. He suggested that we pass a variable around in the url and decrement it every time that we send a put request to a new node. This is a global variable in our program that is called ttl, short for time to live, which is initialized to five whenever any node gets a put request from a localhost ip. With this we were able to implement a reliable broadcast; every node broadcasts to every other node. This is a bit redundant, but we solved the broadcasting issue pretty late in the game. Since all we needed was to pass a message in the url, we could have solved the infinite message passing with either threads or event hooks. We just so happened to be attempting event hooks when we received the advice. If we had more time, we would implement this with threads because the code reads better and is an all around more elegant solution to the problem than using event hooks.

3.1 Leader Election

After setting up broadcast, we really wanted to use our broadcast function to implement leader election in the case that the node fails. We set up functions that would recieve data from other nodes containing the appropriate ID number of the node to become the new master. In the end, our broadcast function drastically slowed down any request to our program, and subsequently would not allow the unit tests to correctly run. Thus our broadcast function is not in use.

A crude way to set up leader election which we did not implement was to simply change the masternode value on the local node. This was made simple given that the members list is static. The reasoning behind design being simple is that any node that tries to contact the masternode which failed and would not respond, would choose the next avaiable member with the lowest ID #. Because each node starts with the

same members list, and thus the same masternode, the next primary/master node choice would be the same for all node when the first master fails.

4 Faults

Should the master fail, each node becomes a lonely KVS that will deliver old/incorrect (potentially correct) GET requests, and will only PUT to its local KVS.