CS425 MP2: Design Document

Bhopesh Bassi, Zhongshen Zeng

April 10, 2017

Contents

1 Overview

In this document, we describe the design of an in-memory key-value store which can tolerate k simultaneous failures and also balance load dynamically as machines join and leave the system. We start by discussing various aspects of the design and our approach followed by discussion of scalability and implementation details.

We use Cassandra like ring topology with SHA-1 hash function for lookups and SWIM for failure detection. Our system is number of simultaneous failures driven and we taking this as input to decide how many replicas to make, how to store data etc.

2 Key storage and lookup

2.1 Protocols considered.

a. **Coordinator based lookup:** In this approach we would have one coordinator(chosen via leader election) who will decide which key is stored where, it would also take care of partitioning and load balancing in case of hotspots. This approach is similar to HBase where you have master server and many slave servers. We didn't use this approach because it was complicated for just an in-memory key value store and also required writing a leader election protocol.

b. **Chord/Cassandra like ring based lookup:** This is the lookup algorithm that we discussed in class. We used this approach because it seemed simpler than coordinator based approach. Because of consistent hashing, the algorithm also gives us good load balancing. Lookup time and bandwidth is $O(1)$ because you just have to hash the key to find out which machine is storing it and then directly go to that machine. We use SHA-1 for hashing and experimented with multiple different values of m(no of bits used to map a number on ring) and decided to use 64 as it was giving fairly uniform distribution of processes across ring.

2.2 Multiple writes of the same key.

We use system time and keep different versions of values for same key. As long as we have time synchronization protocol running in our system, we can rely on system time to get latest value for a key.

3 Process failures.

We implemented SWIM failure detector for MP1 and using the same for this MP. Implementation details of the failure detector stay same as MP1, the only difference is that ping-ack timeouts are smaller because we are not using tc to delay the packets. Please refer to design document of MP1 for theoretical analysis of failure detector.

3.1 Write strategy.

To deal with two simultaneous failures, we store each key at three machines: machine which lies immediate clockwise of *hash(key)*, and also at predecessor and successor of this machine. Whenever a machine sees failure of it's successor or predecessor, it redistributes keys stored locally so that we have sufficient copies to deal with more failures.

3.2 Read strategy.

Read from three machines(machines are determined using same process as in write strategy) and then return the latest value out of three. If a machine storing the key just failed and we are in middle of rebalancing, one or two of machines selected might not have gotten the key yet but still we will return correct value from third machine.

3.3 Processes failing at the same time a read/write request is issued

If the timing is unfortunate, remote server could fail as we are about to read/write data. Retrying is a solution but in the worst case each time you try to write/read, you could have a new failure. Currently we try seven times before giving up because that is the maximum number of failures that can happen one after other in our system. This is controlled by the parameter *kv.try.count* in the file *kvapp.properties*.

4 Process joins.

Each process has a static list of all other machines in the group. This list is used as list of default gateways. A newly joining process contacts first alive process in the static list and gets list of currently online processes. Once it has list of all the online processes, it pulls the data from it's to be predecessor and successor, informs everyone that it has joined the system, and then waits for atleast three processes in total to be online before starting user interface.

While pulling data, we decided not to delete keys from original process because we think it's okay to keep an extra copy or two rather than messing up the delete and losing data altogether. We just want to be very strict about deleting any data. However, if this process is fails and then comes back again, it won't have the irrelevant keys.

5 Scalability

a. **Capacity:** Our system's capacity will increase with more machines if a good hash function is used. A good hash function will distribute processes and keys uniformly across the ring and hence, roughly each machine should be storing equal number of keys. In figure 1 we have a total of 10000 keys in system and we see that average number of keys stored at each node decreases as number of nodes increase. If the maximum capacity of each node is 10000 keys, then overall capacity of system increases because we store one key only on 3 nodes. If hash function is good, actual number of keys at node will be almost equal to average in graph.

b. **Memory usage per node:** As discussed in previous point, a good hash function will give us good load balancing and memory usage per node will be approximately equal. As we add more machines but don't add more keys, memory usage per node will decrease with number of nodes. Figure 1 illustrates this point too because for same number of keys, memory usage per node is decreasing with more nodes.

c. **Latency and bandwidth per lookup:** As our lookup algorithm is $O(1)$, adding more machines won't change latency per lookup. Similarly, bandwidth usage for a lookup will also remain same. In figure 2, we show bandwidth usage while doing 10k GETs vs number of machines. As we can bandwidth usage stays more or less constant and is not affected by number of machines.

d. **Failure detector bandwidth and detection time:** As we are using SWIM style failure detector, bandwidth usage per node is $O(1)$ and won't change as we add more nodes to system(figure 3). Failure detection time for SWIM is $O(logN)$ protocol periods, so it will grow very slowly with number of nodes.

6 Implementation

- Just like MP1, we have tried to keep code modular and build up from small pieces. You will see lots of interfaces/classes in the code but each of them are quite small.
- Whole system is *number of simultaneous failures* driven and nowhere we have hardcoded two as maximum number of simultaneous failures. System gets number of failures as input and stores one copy more than that. Although we have not tested, but code should work as it is for more number of simultaneous failures.
- Failure detector timeout was chosen keeping in mind the 20 sec restriction but at the same time controlling for false positives. With a maximum load of 100000 keys on entire system, various values for ping-ack timeout were tried and 1s seemed to work the best.
- Executing one command from batch file at a time was taking so much time, now we execute multiple commands together and go over network once for every 5000 commands. Now ordering of commands is an issue but we tackle by only executing one type of command in one trip over network e.g. if you have 4000 SETS and 1000 GETS, they all won't be executed in one batch of 5000 but rather two batches. If there were 5000 GETS or SETS, then they would have been executed in one batch.
- Our system has a limitation that when a node joins the system, it tries to pull all the relevant data from neigbours at once. This didn't work when we tried with total load of 1 million keys(each machine having to pull 300000 keys at joining) but it works with less load: have tested with 100000 keys total. In a real world system, we would need something like JDBC cursor which can read in batches.

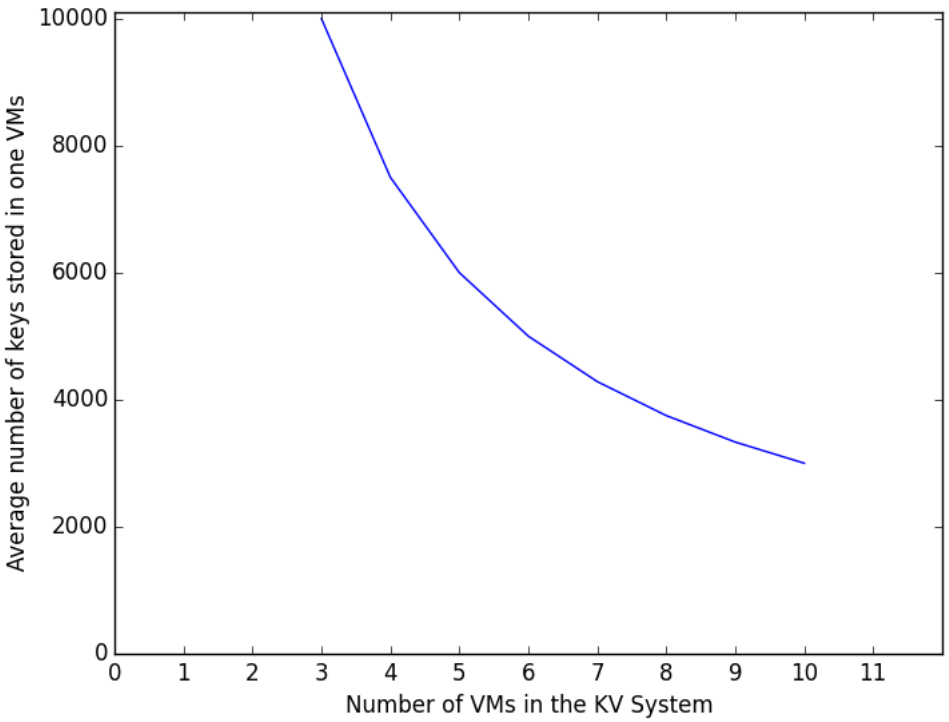

Figure 1: Average keys per node with total of 10000 keys.

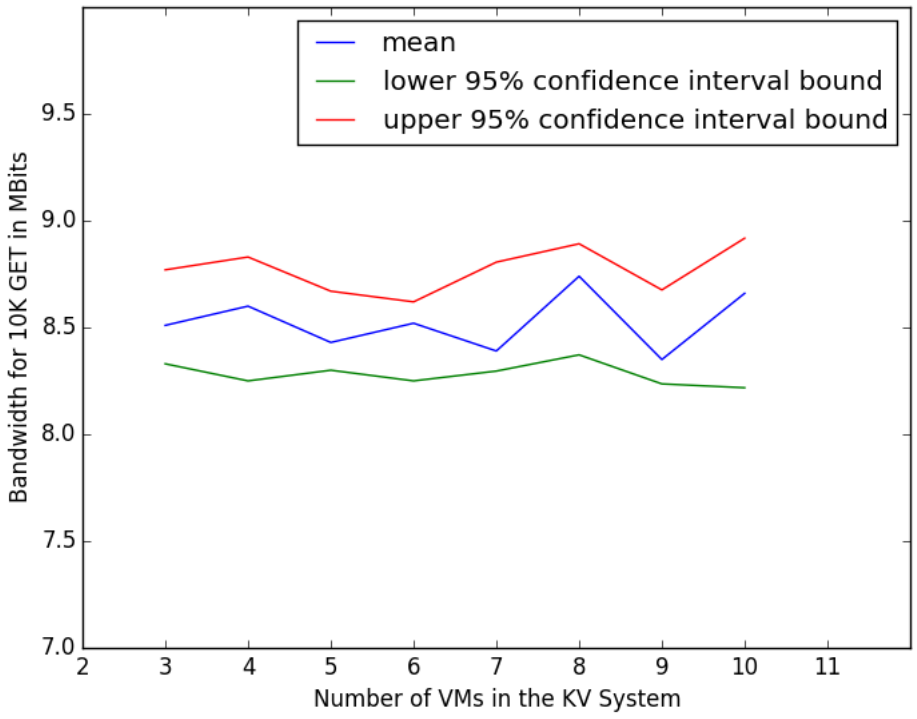

Figure 2: Bandwidth usage for doing 10000 GETs.

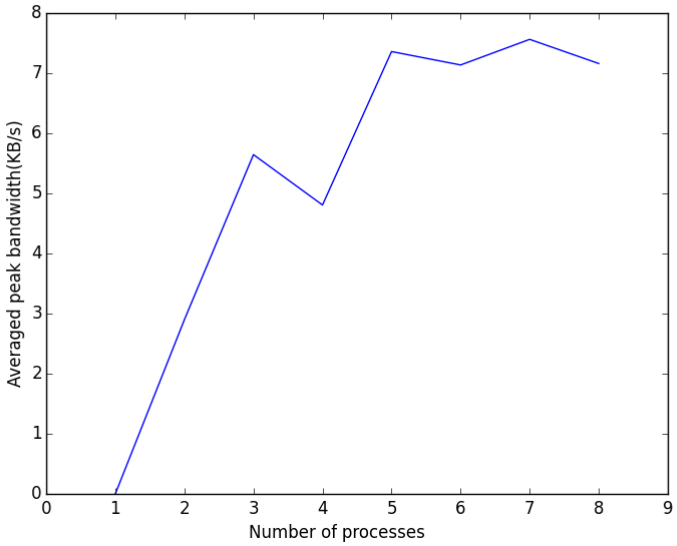

Figure 3: Failure detector bandwidth(per process) stays constant with more and more processes.

7 Acknowledgement

We would like to thank the entire staff for staying on top of our Piazza (public and private) posts and always providing useful feedback and hints. We would especially like to thank Xuewei for his help over unofficial channels like phone and Facebook.