CS 425/ECE 428
Distributed System

MP2 Key-Value Store

Design Documents

Group Member:

Name: Rui Xia
NetID: ruixia2
UIN:660935771

Name: Youjie Li
NetID: li238
UIN: 668129373

Spring 2017
April 10, 2017

Algorithm

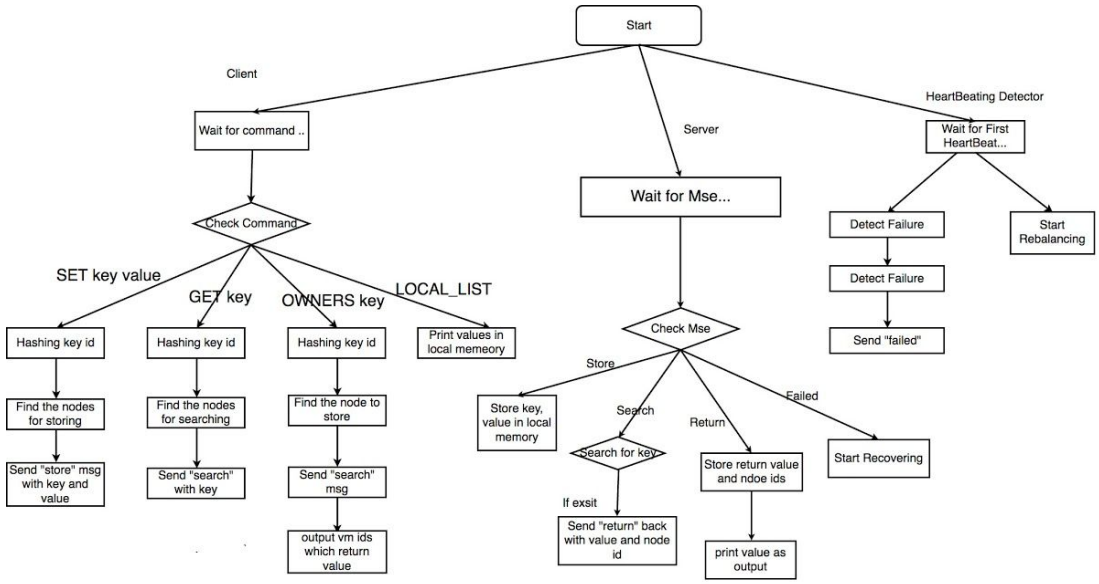

Figure 1 Flow Chart for Algorithm Design

Algorithm Selected

For this mp, we used cassandra, which means Ring-Based DHT without finger table or routing key, to implement the distributed key-value store on 10 vm. The reason why we used it is that the scale of nodes in this mp is managed, each node know the information of each other in the whole community. So the time complexity for search will be O(1). Server can directly hash the data and find the right node to store.

We set the hashing bits as 5 and mapped 10 node id from 0 to 30. The structure of the ring-based DHT is shown in Figure 2

All to all heartbeating are sent between each pair of nodes periodically for failure detection and new join detection.

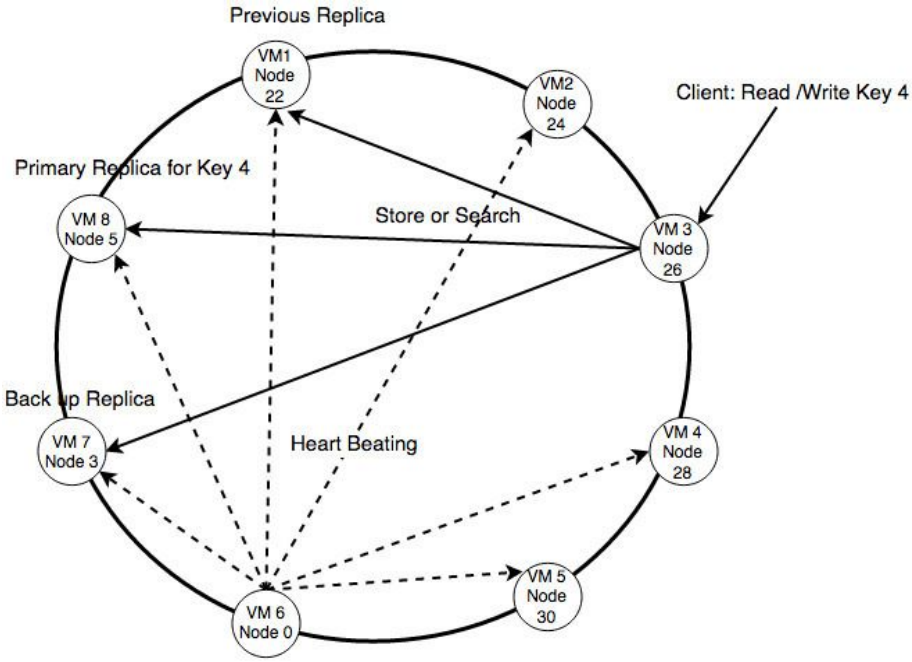

Figure 2 Structure of DHT

Impelimentation Details:

A. Implement Commands

- **SET**
  Hashing key id directly and find three nodes which the replicas should be stored in (primary, back up and previous). Then send "store" msg to those nodes'. Server who recevied a "store" msg should store the data into local memory.
- **GET**
  Similar to SET, hashing and find replicas, then ask for the corresponding nodes with"search" msg. Nodes received "search" will search the key and send back the value with the local time if the key is stored in local memory.
  This method will return the latest value sent back among those three replicas, in order to make sure the update will not be lost.
- **LOCAL_LIST**
  Each node return all data in their local memory.
- **OWNERS**
  Similar to GET, finding replicas and ask for response. Return all VM IDS which respond the "search" msg

B. Recovery

- When failure happened, the precessors and successors should copy replicas in their local memory to pre-precessors and suc-successors in order to keep three replicas. The copy will be implemented as send "store" msg as well.

- In order to deal with two simultaneous failures, we let the server to wait for second failures in a period of heartbeating. If second failure come in one period, we consider them as simultaneous failure and recover them one by one.

C. Rebalance

- We detected a new come up nodes if delivered the first heart beating msg.
- Similar as recovery, if new join detected, the successors should copy parts of its local memory to the new joined nodes. At the same time, precessors and suc-successors should delete the corresponding data from their local memory in order to keep the replicas stored in the three correct nodes.
- In order to deal with two simultaneous come up, we lock the join process, which means allow one new nodes joined at one time. We have a flag "rebalancing", which will be true is the last rebalancing haven't been finished. We will not accepted the first heartbeating if the flag is true, which help to avoid mixed rebalancing.
- Moreover, the rebalancing flag are also set as true during recovering because the requirement said a new rebalance should be delayed until recovery completed.

Implementation Parameters:

- Period of Heart Beating: T= 3 s
- Timeout for one or two failures: T + delta <= 6s (T=3s)
- Hashing bits: m = 5
- Number of replicas: k = 3

Theoretical Estimation for the N-node System

capacity = number of hashed IDs can be stored = 2^m (> N)
memory usage per node = O(N)
bandwidth usage per lookup = 2k (<< N)
latency per lookup = one message RTT
failure detection bandwidth usage = O(N) per node
failure detection time = T + delta (delta relates to max one-way delay)

Evaluation

We have implemented our key-value system using the parameters mentioned above. In order to test the scalability and performance of our system, three major metrics are evaluated, which are the worst-case failure detection time, failure detection bandwidth per node, latency per lookup. Firstly, we have tested multiple independent failures in the system and measured the worst failure detection time. Secondly, we have evaluated the heartbeat bandwidth usage of each node over 10 mins. Thirdly, the latencies of the search operation which returns the values from all replicas are measured as well.

# of Nodes	Worst-Case Failure Detection Time (ms)		Failure Detection Bandwidth per Node (# of msg / second)		Latency per Lookup (ms)	
	Average	95% Confidence Interval	Average	95% Confidence Interval	Average	95% Confidence Interval
4	6252.732913	(6102.732913 to 6402.732913)	0.9995831791	(0.9994431791 to 0.9997231791)	248.673738	(246.9673738 to 249.9673738)
5	6264.083982	(6094.083982 to 6434.083982)	1.332691237	(1.332181237 to 1.333201237)	226.0504484	(209.0504484 to 243.0504484)
6	6303.951931	(6143.951931 to 6463.951931)	1.700566257	(1.677566257 to 1.723566257)	249.8318354	(218.8318354 to 280.8318354)
7	6297.753255	(6127.753255 to 6467.753255)	2.005982028	(2.000882028 to 2.011082028)	347.5894451	(291.5894451 to 403.5894451)
8	6536.640338	(6356.640338 to 6716.640338)	2.332193098	(2.3318230979 to 2.332563098)	293.282485	(254.282485 to 332.282485)
9	6320.541819	(6170.541819 to 6470.541819)	2.665224527	(2.664554527 to 2.665894527)	274.186945	(240.186945 to 308.186945)
10	6521.601701	(6452.601701 to 6590.601701)	2.954707147	(2.950607147 to 2.958807147)	267.3580117	(246.3580117 to 288.3580117)

As for the worst-case failure detection time, the theoratical value is (T + 2*max-one-way-delay) while the practical value is (Timeout + max-one-way-delay + program overhead) which is around 6300 ms. As can be seen, this factor is scalable with an increasing number of nodes.

As for the failure detection bandwidth, it increases with the size of the system, which is as expected, because all-to-all heartbeat is adopted in this design in order to support arbitrary number of simultaneous failure and join.

As for the latency per lookup, it depends only on the RTT and program overhead. As can be seen, this factor is also scalable.