Design Document for CS425/ECE428 MP1

Zhuo Li[1], Ruiqi Zhong[2]
1. Dept. of Computer Science, UIUC
2. Dept. of Electrical and Computer Engineering, UIUC
Email: {zhuol2, rzhong5@illinois.edu}

1 Overview

In this MP, we implemented an in-memory key-value store with scalability and failure-proof capability. We use membership list in Cassandra to store all nodes and their hashed IDs to guarantee fast speed to handle writes. Also, for the implementation of failure model, we detect the failure of some node in the membership list through catching exception thrown on invocating a remote method of that remote node interface periodically.

2 Algorithm Designs

For both key-value store and failure model, there are more than one way/algorithm to implement. In this section, we will take a deep look at the algorithms we choose and the reason why we choose them other than others in our work. Pseudocodes will be provided for implementation details at the end of each sub-section.

2.1 Basic Functionality of key-value Store

We choose membership table for looking up nodes to store a certain key. We initially considered using finger table for looking up, however, finger table usually works well when the number of nodes is very large. Taking into account the fact that we only have 10 VM clusters and the advantage in speed for handling writing for Cassandra is remarkable, we choose membership table for storing all nodes and looking up.

2.1.1 Algorithm

When a node joins, we register it locally and invoke setupChord function to include all the other nodes alive to its remote communication list. We sort all nodes by their hashed IDs in ascending order. Then we update the predecessor and successor of that node, link the ring and build membership table. Before rebalancing keys for adjacent storage, we check if the joining node's hashed Id is the largest one among all the nodes in membership table. If it is the node with the largest harshed Id, we let this node traverse all the keys in storage of all nodes and reassign all the keys. If it is not the node with the largest harshed Id, we simply make the node with the largest

harshed Id in membership table to traverse all the keys in storage of all nodes and reassign all the keys.

2.1.2 Implementation Details

```
If a node joins
     Set up Chord
     Link the ring
     Build membership table and let the node with the largest hashed
     Id in the membership table to rebalance keys

If a node leaves
     Handover its storage to its successors
     Exit the ring
     Update all nodes' membership table, and remove heartbeat

If SET
     Find the node that the key is to be hashed to
     Put the key and its value to that node's storage

If GET
     Find the key' hashed Id
     Find the node that stores the key in the membership table, and
     get its value from that node's storage

If OWNERS
     Find the node that the key is hashed to
     Check if its predecessor and successor also have the key

If LIST_LOCAL
     Get the local storage of a node

If BATCH
     Execute a list of commands listed above
```

2.2 Failure Model

We let each node alive to periodically invoke the remote method of all other nodes in its membership table and detects failure of the node once the remote method invocation throws exception to keep simplicity.

We use one leader to reassign keys when a new node joins and a node detects the failure of any other node to avoid the exception thrown indicating that concurrent modification of the storage of

one node could not be conducted by several other nodes in network. Also, we set a recovery status for each node in the membership table indicating if it is in recovery status to delay the set operation of keys until all nodes in the network are not in recovery status to avoid mess in storage modification.

2.2.1 Algorithm

On setupChord, we start the timer for this node periodically invoking the remote method of all other nodes alive and in its communication list. Also, we start the process for other nodes communicating with it in similar way. When a node crashes and its failure is detected by some other node, the node that detects the failure should remove the failed node from its membership table and reset the predecessor of the failed node's successor and the successor of the failed node's predecessor. Then, it need to check if it is node with the largest hashed Id in the network. If so, it will travese all the keys, find nodes they are to be hashed to, check and reassign the keys.

Also, the joining of any node will be delayed for 1s once that node detects there exists some node in the membership table that is in recovery state until all the nodes are not in recovery state. Before we put any key in the key-value store, we also check if there exists any node whose recovery status is true. We delay the SET operation until all nodes's recovery status are false to avoid mess in storage.

2.2.2 Implementation Details

```
If a node joins
     Set up Chord, initialize timer for this node to
     periodically invoke the remote methods of other nodes in
     network. Start the similar process for other nodes
     communicating with it periodically.
     Link the ring
     Build membership table and let the node with the largest
     hashed Id in the membership table to rebalance keys

If SET
     Wait until the recovery status of all nodes in the network
     become false
     Find the node that the key is to be hashed to
     Put the key and its value to that node's storage

If any node detects the failure of another node
     Delete the failed node from the membership table
     Reset its predecessor's successor and its successor's
     predecessor
```

```
Check if itself is the node with the largest hashed Id. If
so, traverse all the keys and reassign all the keys. If
not, let the node with the largest hashed Id do the
above-mentioned operation.
```

3 System Evaluation

3.1 Theoretical Estimation

For our distributed key-value store, with the increasing of number of nodes in the network, the network capacity for key-value pairs will grow. The memory usage per node will decrease with the increase number of VM clusters because the key-value pairs will distribute more sparsely in more VM clusters with different hashed Ids.

The bandwidth usage per lookup is always O(1) because we only find from one machine and get the target node by the harshed Id of the key. The failure detection bandwidth usage is O(N) because we make all nodes to invoke the remote method of all other nodes in network. So the failure detection bandwidth will increase as the number of machines in the network increases.

The trend of the latency per look up and failure detection time with the number of clusters is plotted in 3.2. We can conclude from the figures in 3.2 that our system is scalable because the performance is relatively stable with the increase of number of machines. Also, the time consumed by set operation, lookup operation and failure detection is very relatively fast.

3.2 Figures and Tables

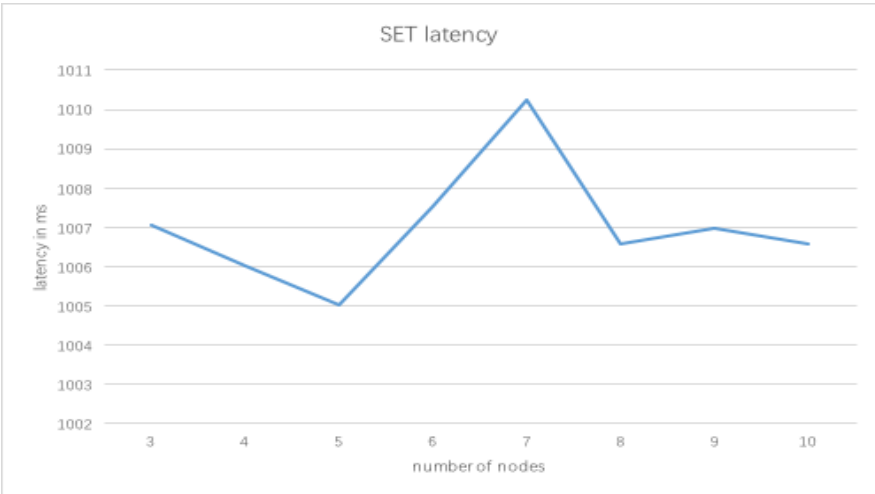

Figure 1 SET Latency vs. Number of Nodes in Network

	3	4	5	6	7	8	9	10
95% Confidence Interval / ms	[1003.538611, 1010.544872]	[1004.614154, 1007.385114]	[1004.065192, 1005.991821]	[1006.0886, 1008.996108]	[1008.155195, 1012.302167]	[1005.766422, 1007.397677]	[1005.031085, 1008.935842]	[1005.532331, 1007.631022]

Table 1 95% Confidence Interval for SET Latency

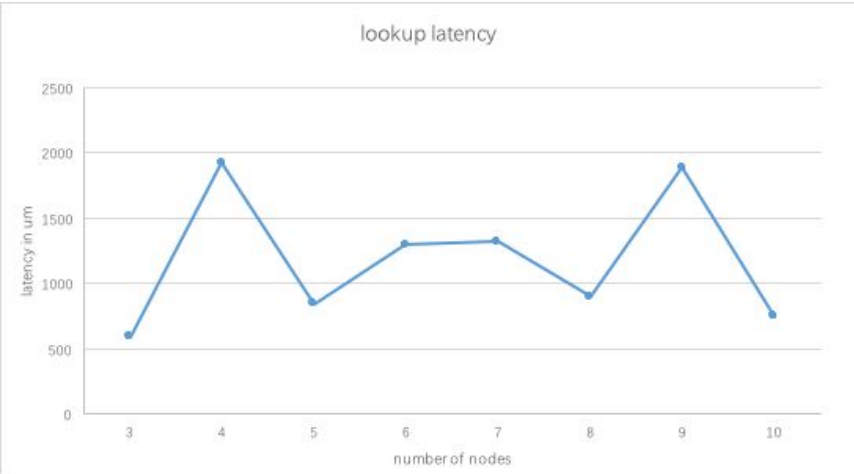

Figure 2 Lookup Latency vs. Number of Nodes in Network

	3	4	5	6	7	8	9	10
95% Confidence Interval / us	[338.5085256, 843.4066744]	[789.4022136, 3057.520786]	[572.7496014, 1117.210399]	[741.5132687, 1848.564531]	[711.5990187, 1932.505181]	[537.1977658, 1265.104234]	[963.5435981, 2812.937202]	[549.9407112, 943.9774888]

Table 2 95% Confidence Interval for Lookup Latency

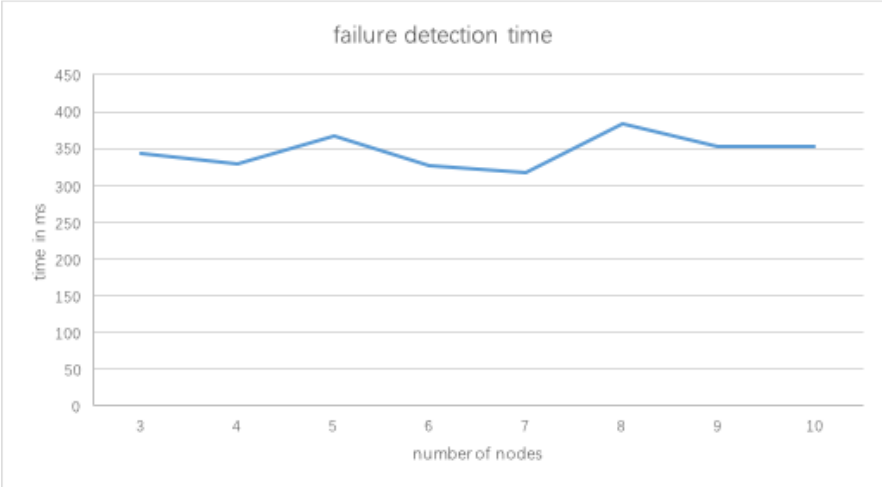

Figure 3 Failure Detection Time vs. Number of Nodes in Network

	3	4	5	6	7	8	9	10
95% Confidence Interval / ms	[310.2317704, 375.1682296]	[306.9203058, 351.4796942]	[331.3233328, 401.6766672]	[306.0585989, 344.7414011]	[306.6911966, 326.1088034]	[352.1894829, 416.0105171]	[322.993828, 384.406172]	[326.5502878, 380.6497122]

Table 3 95% Confidence Interval for Failure Detection Time

3.3 Description of Testing Approach

We uses a batch file which set several numbers of key-value pairs to the system. In order to cover all possible cases, we designed our test key-value pairs to be almost uniform distributed on all nodes.

We test the cases in which one node fails, two nodes fail simutaniously, one node join, several nodes join at the same time, etc.

To get the required metrics like lookup latency and failure detection time, we get the system nano time before and after we SET/GET a key. We also monitor the key interruption for exit and print the time when exit and heartbeat detection time as well.

4 Summary

In this MP, we built a distributed key-value store which is scalable, failure-proof and in-memory using membership table. We let each node periodically invoke the remote function of other node interfaces and catch exception when the remote node fails. To solve the problem of modifying the storage of a node by several other nodes, we always let the node with the largest hashed Id, which is regarded as the leader, to reassign keys in recovery and joining. We also set recovery status for each node, and add key-value pairs only when all nodes' recovery states are false to avoid mess in storage.