Chordy: A Distributed Hash Table

Weherage, Pradeep Peiris

October 14, 2015

1 Introduction

Chordy is a distributed hash-table, which is based on Chord, a distributed lookup protocol.

Chord address the problem of efficiently locating nodes that stores particular data items. Also, it stabilizes with dynamically changes of node join/leave in the system. Chordy implementation in Erlang is based on these features from Chord protocol.

2 Main problems and solutions

As Chord protocol, the main operation here is to locate a node with a given key. And associate any data value with the key and store key/value pairs at the node to which the key map. Also it needs to adopt with nodes that leave and join the system.

A balance data distribution across the nodes is also another basic problem to be addressed. Chord uses a variant of consistent hashing in assigning keys to nodes, which causes each nodes to receive same number of keys, and make balance distribution of data across the nodes. In our implementation we use a basic random number generation as the following, which has some guarantee of uniqueness.

```
generate() ->
     random:seed(now()),
     random:uniform(1000000000).
```

With compare to Chord, our implementation is totally based on message passing, and of course asynchronous. A node is represented with properties of Key, Predecessor Node, Successor Node and Store. The nodes are capable of repairing itself and finding its own position in the system (ring) through messages passing. This is automated via a special scheduled procedure, stabilization. That is, each node in the ring (system) is scheduled to frequently check the node position in the ring is correct. Basically, a node start with

issuing a message to its Successor asking the Predecessor. And in return of the response message, if it is not itself, use relevant messages and actions to repair itself.

As stated previously each node has its own storage, which mapped with associated key. With arrival of new nodes, the storage should be split accordingly. It uses the following function in storage key separation.

```
split(_, _, []) -> {[], []};
split(From, To, Store) ->
  lists:foldl(fun({Key, Value}, {Updated, Rest}) ->
    case key:between(Key, From, To) of
      true -> {Updated, lists:append([{Key, Value}], Rest)};
      false -> {lists:append([{Key, Value}], Updated) , Rest}
    end
  end, {[], []}, Store).
```

3 Evaluation

Chorday as a distributed hash table provides an efficient data access to its keys. Also it has other key features around concepts like flat key space, availability, decentralization, load balance and scalability. This section explains these features of Chorday and evaluation with some test cases.

3.1 Flat key space

The key-space in Chordy is flat and no constraints on the structure of the keys it lookups. This provides a great flexibility for applications to use their own keys on Chordy.

3.2 Availability and Decentralization

The Chordy system is highly available as it automatically adjusts the node ring on failures and arrival of new nodes. Basically the nodes ring can be repaired while still responding to its application layer. And it is decentralized, best suited for distributed peer to peer applications.

3.3 Load balance

As mentioned previously hashing key function (randomly generated keys in our implementation) tends to balance the data load across the nodes. This provides degree of data load balancing. Figure 1 and 2 shows a result of simple data load test. Chordy was initiated with one node and populated 10 000 data items. Then, number of nodes was increased and evaluated how initial data load was distributed across the new nodes. As it is depicted in

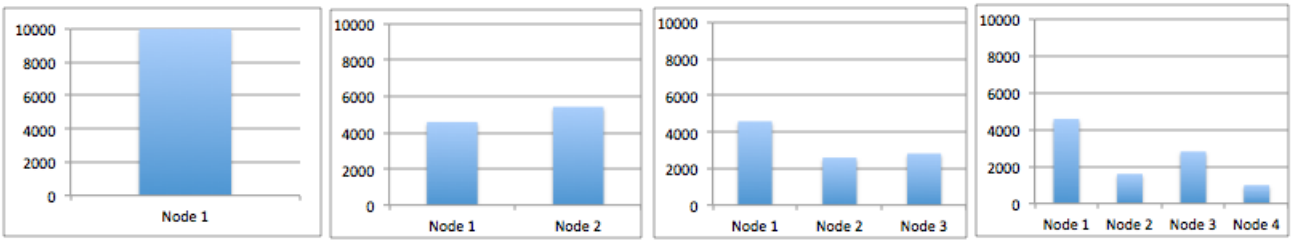

Figure 1: Test case1, data distribution 4 nodes

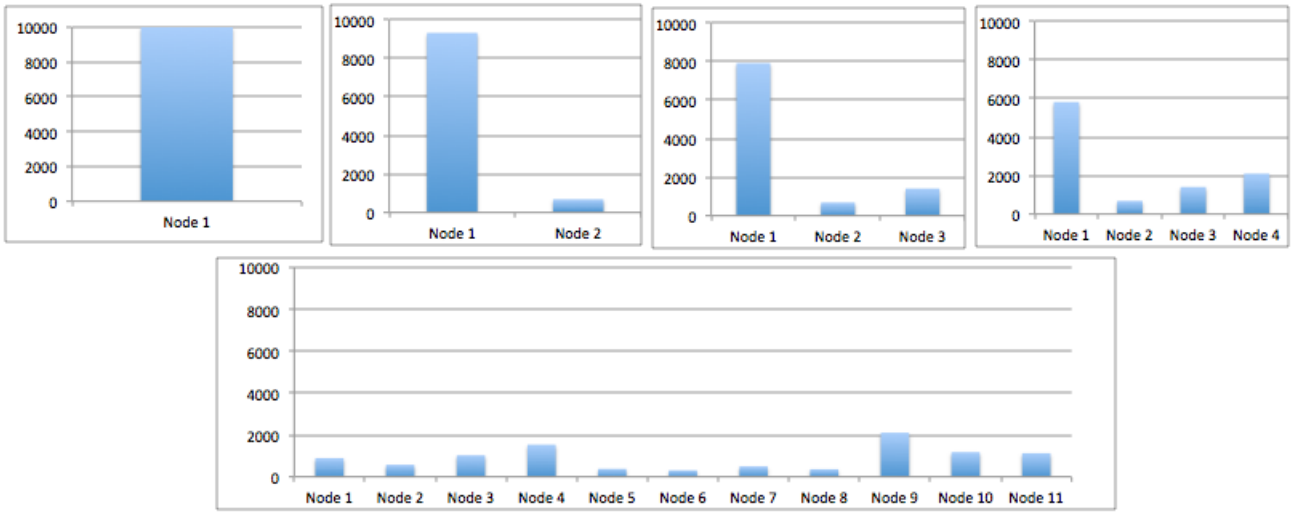

Figure 2: Test case2, data distribution with 11 nodes

the Figure 1, in first test case, Node 2 has caused Node 1 to share data more evenly with Node 2. But the load remain unchanged in Node 1 even with introduction of Node 3 and Node 4. But any new future nodes may caused its data separation. In second test case, data was not spread evenly with first 3 nodes. But, data tends to distribute across all nodes more evenly with introduction of more nodes.

3.4 Scalability

According Chord, the data lookup grows as log of the number of nodes. As a test case (Test case 3) for our implementation, the response lookup time is evaluated over 10000 items. Figure 3 shows the response time, starting from one node and increasing number of nodes up 8. As it represents, the

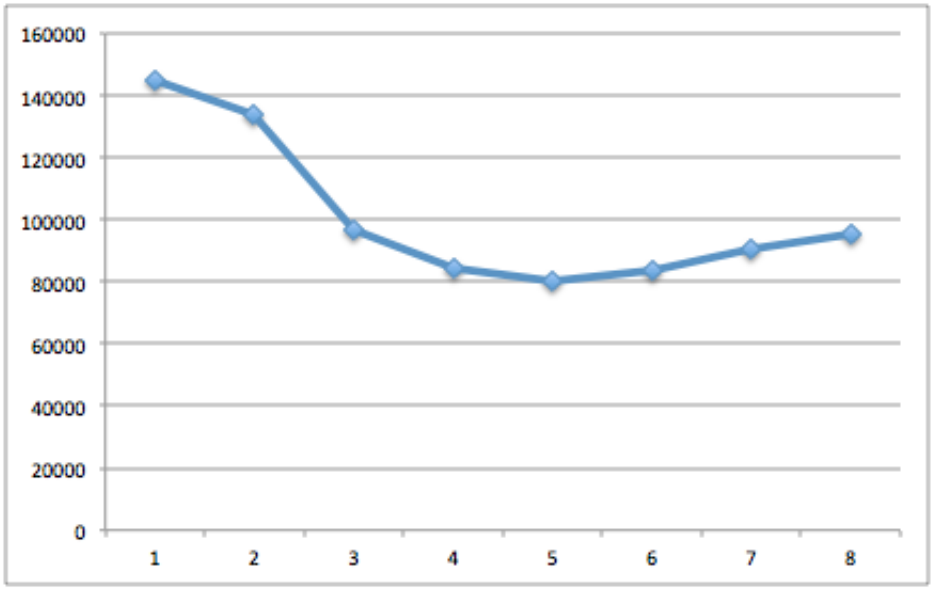

Figure 3: Response Time vs Number of nodes

response time tends to improve with more nodes but up to some level. In our test case, after 5 nodes the response time tends to be consisted. Also, it is noticed a good response time when data is evenly distributed. That is, if we take the response time for 10000 items lookup in Test Case 1 (Figure 1) and Test Case 2 (Figure 2) with 4 nodes, Test Case 1 has a better response time as data is more evenly distributed compared to Test Case 2 with 4 nodes.

4 Conclusions

In current implementation, it was considered only inserting new nodes to the ring and data storage. And it was evaluated the response time with same data load with different number of nodes. Also the response time with data distribution across nodes with our basic hash-key function. Basically, more evenly distributed data courses a better response time. Chordy should be further improved to handle failures and repair the ring in such failures. It is an important factor data replication, so that there wouldn't be any data lose.