
Report: Project 3 – Chord Protocol in Scala

(Both Failure and Non Failure Model)

Students: Chiranjib Sur (UFID : 2531 4396)

Pramit Dutta (UFID : 7513 2433)

Course: COP 5615: Distributed Operating Systems
Principles

Chord Protocol is a lookup protocol to peer to peer applications. At the heart of Chord is a distributed hash table that stores key-value pairs. A Node (a peer) stores the values for all the keys for which it is responsible. The protocol specifies how keys are assigned to the nodes and how one node can discover a key by first locating the node that is responsible for that key.

Nodes and keys are assigned an m -bit identifier by using consistent hashing. SHA-1 algorithm is used in this case. Consistent hashing eliminates the chance of two nodes colliding in the identifier space. This allows nodes to join and leave the network without any hindrance. Nodes and keys are arranged in an identifier circle that has at most 2^m nodes, ranging from 0 to $2^m - 1$. Each node has a successor and a predecessor. Keys can also have successors. The successor node of a key k is the first node whose id equals to k or succeeds k in the identifier circle. Every key is assigned to its successor node.

The basic usage of a peer to peer network is to search a key from a node. If the key is found in a particular node it returns it, else that node forwards the message to its successor to continue the search operation. In Chord protocol each node maintains a finger table containing m entries. The i th entry of node ' n ' will contain successor $((n + 2^{i-1}) \bmod 2^m)$. When a node wants to lookup a key k , it will pass the query to its nearest successor or predecessor as recorded in the finger table and this process will propagate until a node finds out the key in its immediate successor.

The following are maximum number of nodes our system can handle to process in limited time.

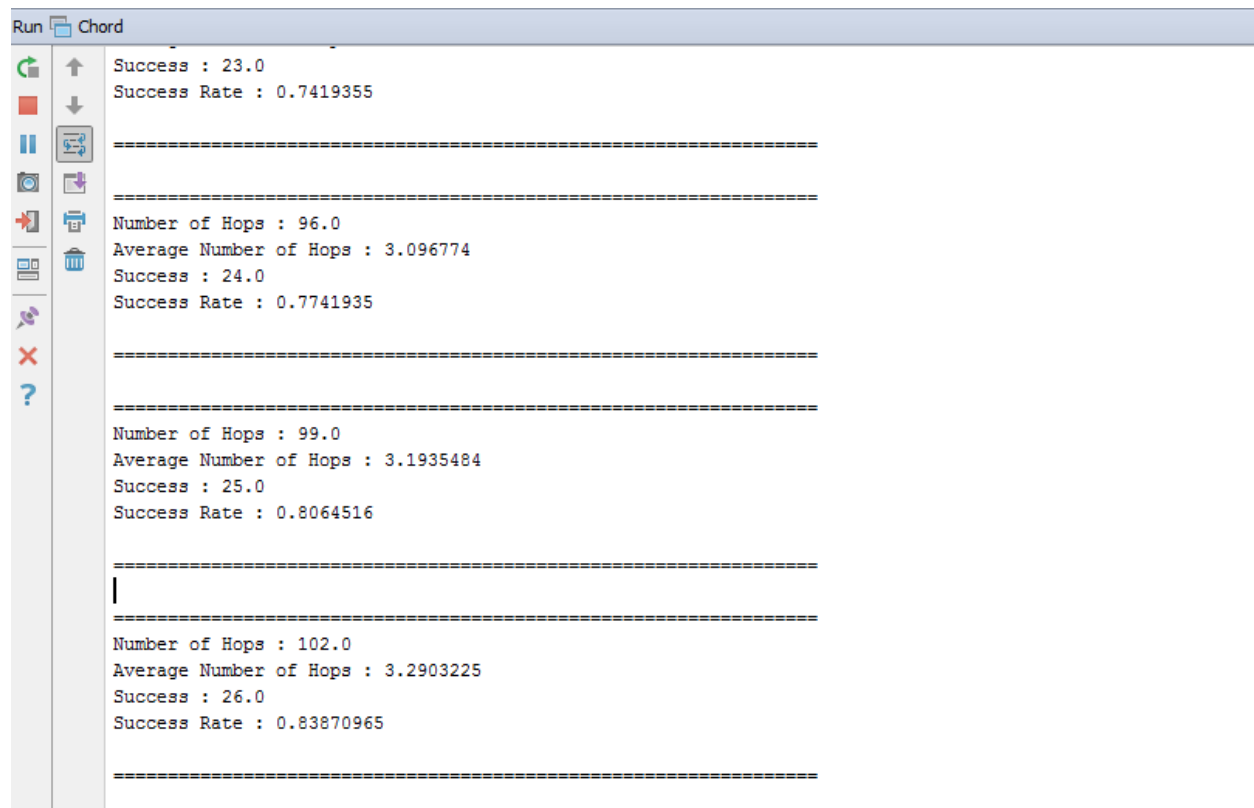
Largest networks dealt with: Maximum number of nodes for which the Chord Protocol manage to run in our system: (We tried with integers ranging from 10 to 20000, increasing 10/100/1000/5000 at a time and recorded these floors of convergences)

15000 nodes

However it may happen that the system can handle more records, but it takes time to run as the number of nodes increases and all the others have to update their fingerTable.

If the system configuration is better and is made to run for a long time, it can process the creation of tables for high number of nodes with percentage of spread to much higher rate.

Snap Shots of the Program output



```
Run Chord
Success : 23.0
Success Rate : 0.7419355
=====
=====
Number of Hops : 96.0
Average Number of Hops : 3.096774
Success : 24.0
Success Rate : 0.7741935
=====
=====
Number of Hops : 99.0
Average Number of Hops : 3.1935484
Success : 25.0
Success Rate : 0.8064516
=====
|
=====
Number of Hops : 102.0
Average Number of Hops : 3.2903225
Success : 26.0
Success Rate : 0.83870965
=====
```

Fig 1: Snap Shot for one case where the final output (for average hops and success rate) is being shown for the non failure model

For the failure model, the output format is same, but there is change in code.

Results and Discussion

In the following figure (Fig 2) we have plotted the average number of hops that are required for searching a key in the chord ring for the different number of nodes and the trend clearly shows that with the increasing number of nodes, the average number of hops also increases. We must also include that the number of data files also increased with the number of the nodes, but logically it has very little to say about the increase in the number of hops. It can be easily seen that the non-failure topology performs best though both of them has the same level of connectivity. This is due to communication failure it faces during search and there are more failure search and hopping, it has faced and there may be much traffic and loss of data as the link fail to forward the packet to the required destination. Also in failure model, as the system fails, there is no message for failure and thus it has not contributed to the average number of hops whereas in case of non-failure model, at least the master has got a message that the searching has failed.

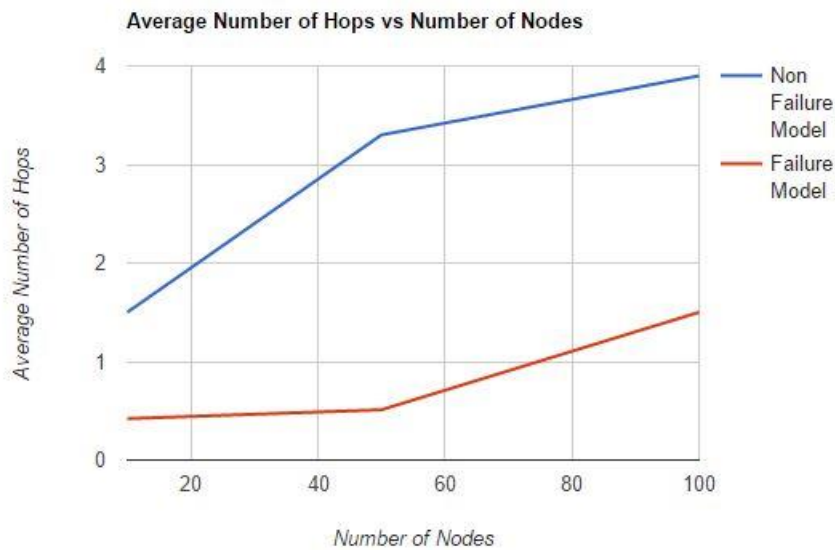


Fig 2: Plot for Average Number of Hops vs Number of nodes (for both the non-failure and failure model)

In case of fig 3, we can see that the success ratio increases for the failure model with the increase in the number of nodes. This is due to the fact that with the increase in the number of nodes, the connectivity is more for each individual as the option in finger table is much more. However for the non failure model, the efficiency decreases as the number of connectivity can cause more spread for the search patrol message and may end up in failure.

However there are cases where the message loops between nodes and thus result in failure in search even though the data file is present. This is a failure in the routing and forwarding table and the distribution of the nodes in the ring.

A properly distributed node ring can help in better distribute the load and also help in proper search mechanism.

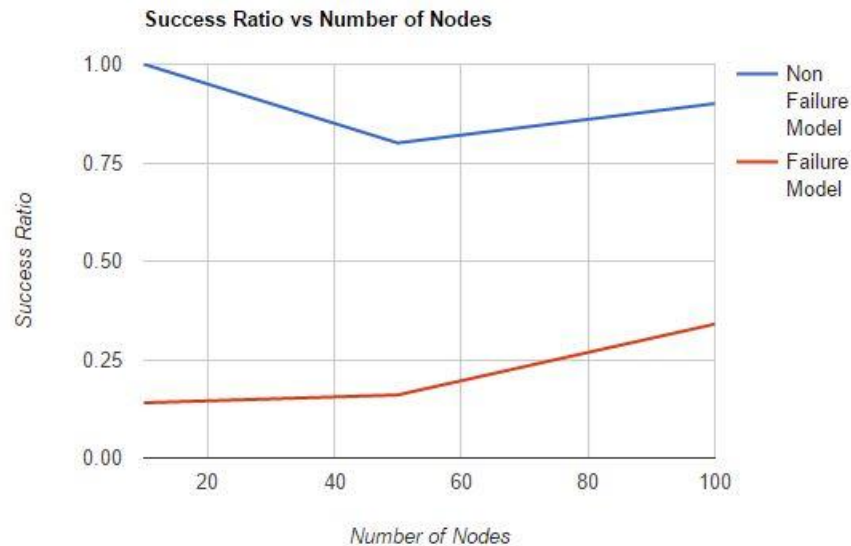


Fig 3: Plot for Success Ratio vs Number of nodes (for both the non-failure and failure model)

In the following figure (Fig 4), we have plotted to trend of how the failure model works for different percentage of failure. Like say the failure is about 20% of the total number of nodes, then the system will work in this way. However as the statistical significance is not considered, the trend may change for different sampling for different distribution of nodes in the chord ring and also on how the failure distribution is spread in the network. Also since the search is also considered and generated randomly, there is high chance that percentage and average number of hops may change. The average number of hops is very low because there are failure in search process and there is not even any acknowledgement of the failed cases sent to the master, where these values are calculated.

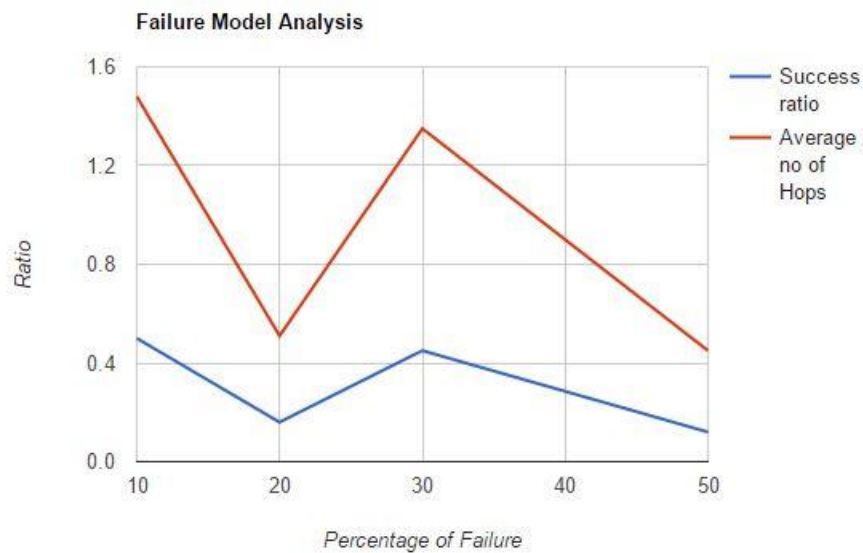


Fig 4: Plot for Success Ratio and Average number of hops with variation in the percentage of failure of nodes (for failure model)

What is working?

The finger table updation in $\log(n)$ time and the efficient searching (in $\log(n)$ time) is the main objective of our model and thus both of these things are working fine and efficiently. This can be seen from the graph as for $m=9m$, we have a descent low average number of hops.

However in case of failure model, when a node dies or leaves, it may not inform the others or the master about it and there is inconsistency in the model and search. In that case the system is not resilient.

Solution to such cases are:

- ➔ Handshaking of network protocol (like in routing table), not possible in our case and also a message overhead
- ➔ Nodes themselves take care of it and update their finger table
- ➔ There is considerably frequent updation of the finger table, initiated by the master if there is frequent loss in the successor and predecessor for any node.

Difficulties faced

While propagating the messages we faced an issue when we tried to pass a message from one node to just another node. The Algorithms were not properly synchronized and behaving crazily. When we increased the number of nodes a particular node can send messages to, the messages propagated fast and the algorithms need to take care of the consistency of the finger table..

- ➔ Codes behave crazily and difficult to debug
- ➔ Convergence of the finger table update
- ➔ Data distribution update
- ➔ Scarcity of propagation, hence increased the propagation number
- ➔ If the number of messages increases, then messages may drop
- ➔ Sleeping is a bad idea for node to communication as it stops one of them and at that moment the other messages sent to them are not received
- ➔ Communication overhead can create a problem and can make us feel that the node has failed to receive
- ➔ Sometime random and efficient nodes can help in better propagation
- ➔ Failure node detection cannot be determined dynamically, unless you try to update the finger table and find change in table entries.
- ➔ Finger table goes to inconsistent state if things are not taken care of.

Findings:

We found that distributed computing is easy to program, but difficult to debug and synchronized, unless there are procedure to make it completely asynchronous. In case of asynchronous system, the message model increases and chance of inconsistency is much.

References : http://opendatastructures.org/ods-java/12_2_AdjacencyLists_Graph_a.html
<http://alvinalexander.com/scala/iterating-scala-lists-foreach-for-comprehension>