# Chord Protocol Implementation and Avg Hop Testing

**Chord Protocol:** https://pdos.csail.mit.edu/papers/chord:sigcomm01/chord_sigcomm.pdf
**Follow the instructions to run the code**
sbt compile;
sbt “run <args>”
Arguments will be:  <number of nodes> <number of requests>

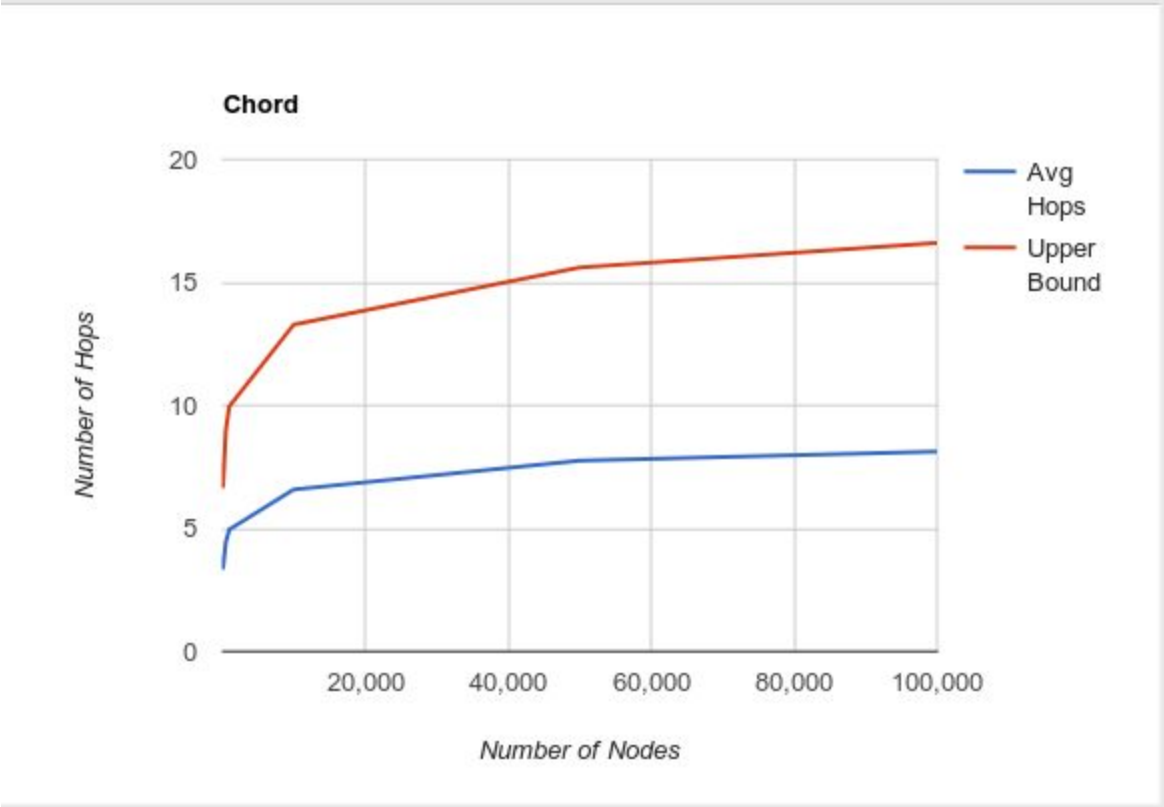
**Example:**
sbt “run 1000 5”

**Summary**

- The code is able to join nodes sequentially, and properly create the finger tables of the nodes joining in the network.
- The code is able to update finger tables for all the other nodes in the network who are affected by the new node.
- The code is able to correctly identify the node where our keys can be found from any node.
- The code is able to perform concurrent node joins as part of bonus section by stabilizing the network periodically and updating finger table of each node by calling Fix Fingers functions periodically. Our implementation involves stabilizing and Fix Fingers every 5 milliseconds for all the nodes.
- The code is able to maintain a successor list of size m (2^m = max possible number of nodes). Successor list maintains next m successors of a particular node.
- Whenever a failure occurs and a node fails, all remaining nodes who had their successor as this failing node update their successor to the next successor pointed by their successor list.
- The code is also able to reflect this change in finger tables of all the remaining nodes who had any of the failing nodes in their finger table. This is done by periodically(5 milliseconds) calling the stabilizing and Fix Finger function as mentioned above.
- In this implementation 20 percent of total nodes are failed after initial network is established.
- As part of node failure we established that this system is resilient  because it is successfully reflecting the finger tables of the remaining nodes in the network, and find all the keys correctly.
- The algorithm performs efficiently and the average number of hops is always below the upper bound (log(n)) by a fair margin, where n is the number of nodes.

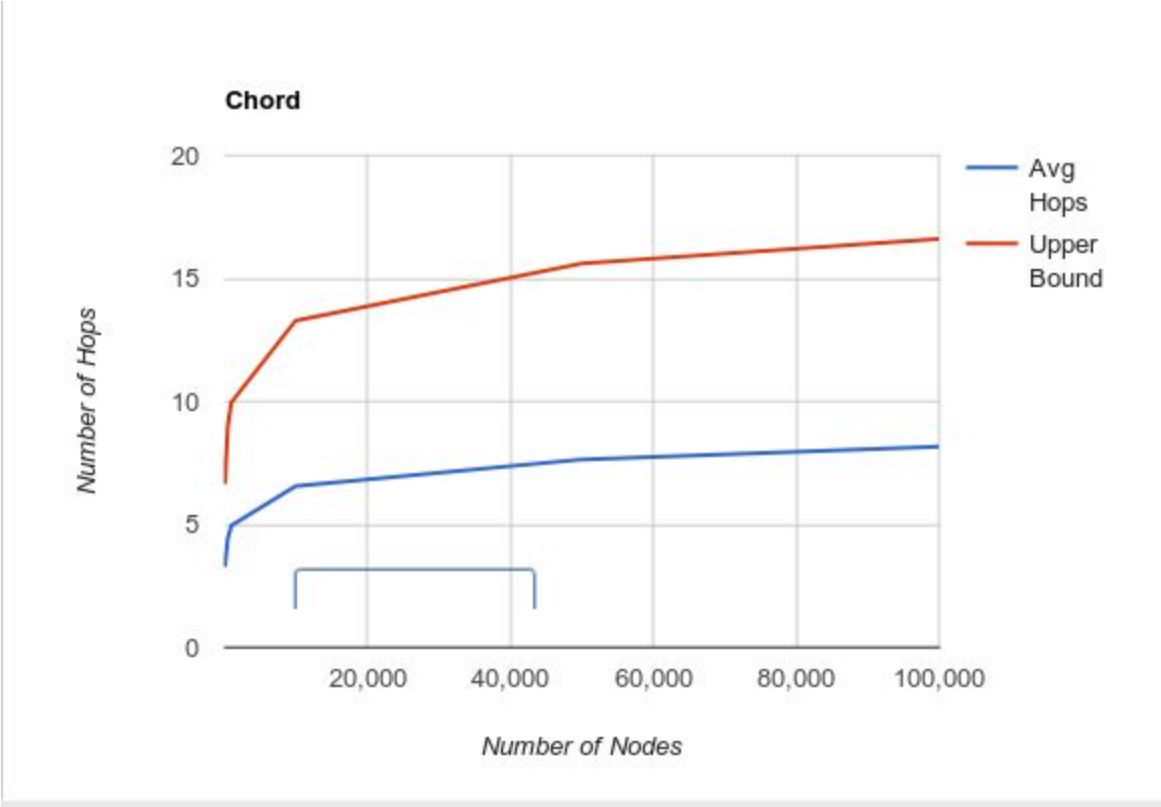
**Largest network we dealt with:**
We were able to create a network containing 100,000 nodes successfully.
**Chord Implementation:** Sequential join
1. Number of Requests: 3

| Number of Nodes | Average Number of Hops | Max bound of average hops |
|---|---|---|
| 100 | 3.334 | 6.643856 |
| 500 | 4.427 | 8.965784 |
| 1000 | 4.96 | 9.965784 |
| 10000 | 6.585 | 13.287712 |
| 50000 | 7.758 | 15.60964 |
| 100000 | 8.128 | 16.60964 |




2. Number of Requests: 5

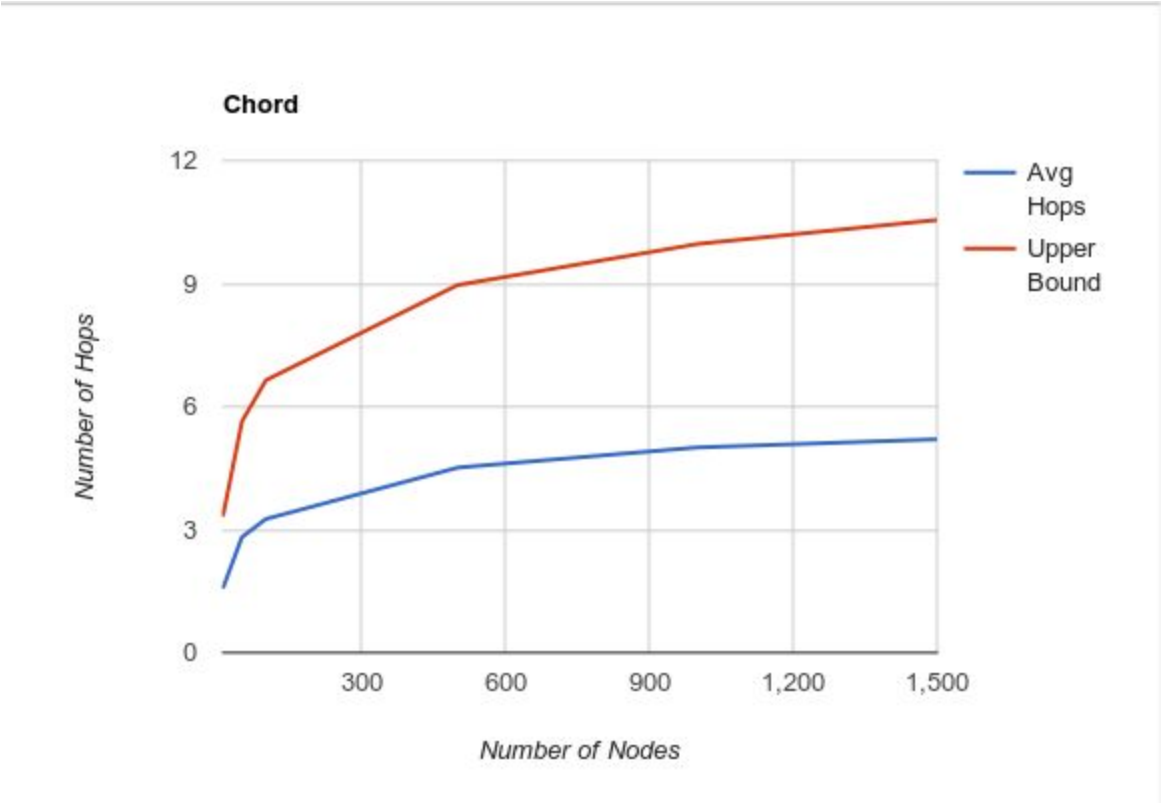
| Number of Nodes | Average Number of Hops | Max bound of average hops |
|---|---|---|
| 100 | 3.278 | 6.643856 |
| 500 | 4.416 | 8.965784 |
| 1000 | 4.956 | 9.965784 |
| 10000 | 6.562 | 13.287712 |
| 50000 | 7.642 | 15.60964 |
| 100000 | 8.163 | 16.60964 |




**Bonus Implementation: Concurrent Node Joining**

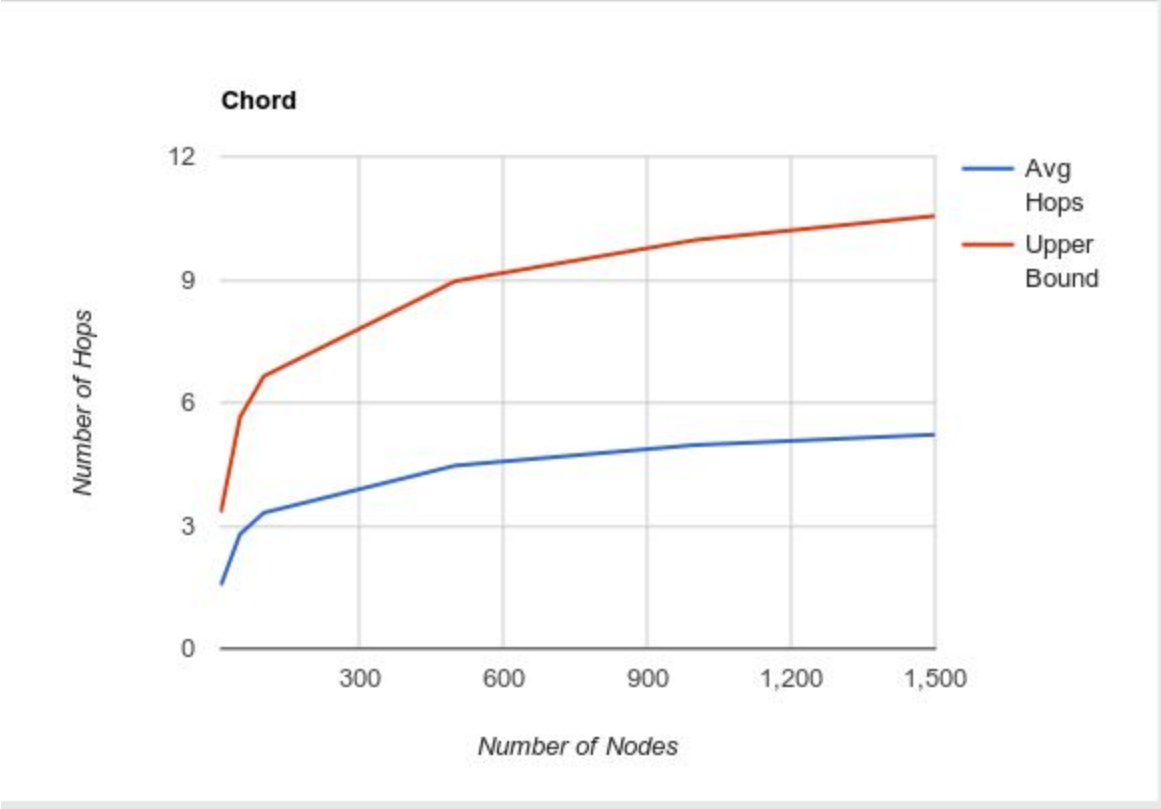
1. Number of Requests: 3

| Number of Nodes | Average Number of Hops | Max bound of average hops |
|---|---|---|
| 10 | 1.566 | 3.321928 |
| 50 | 2.82 | 5.643856 |
| 100 | 3.26 | 6.643856 |
| 500 | 4.512 | 8.965784 |
| 1000 | 5.005 | 9.965784 |
| 1500 | 5.201 | 10.550747 |




2. Number of Requests: 5

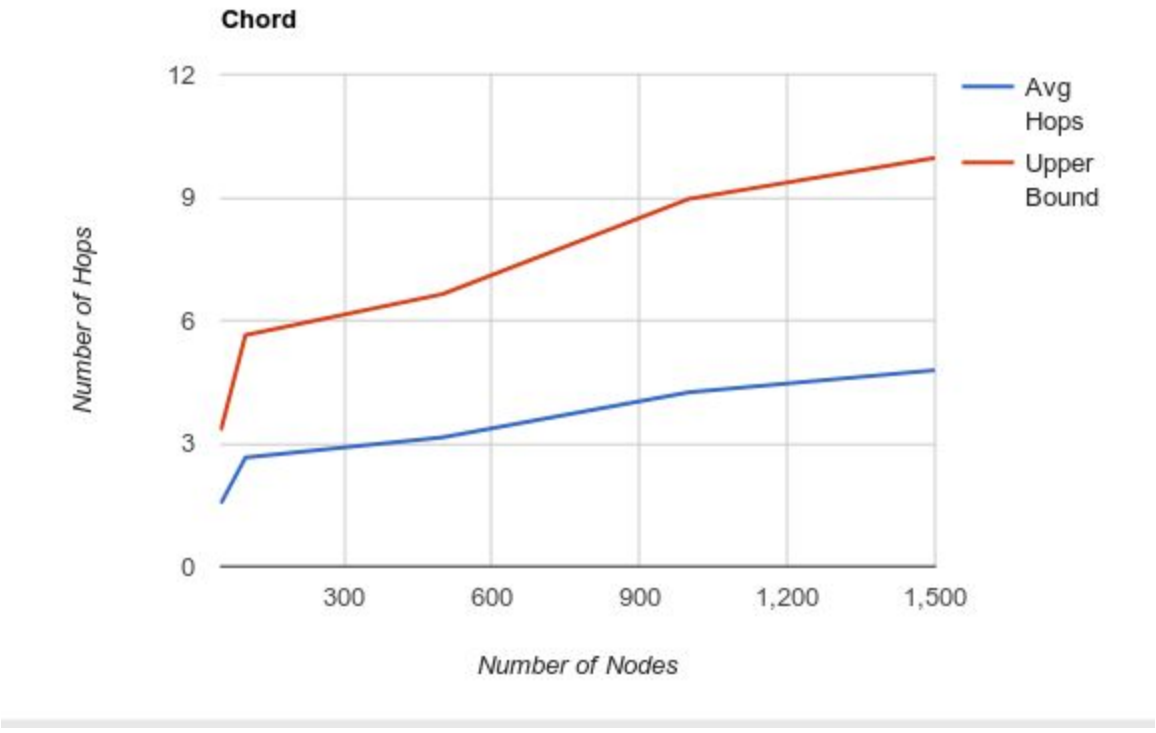
| Number of Nodes | Average Number of Hops | Max bound of average hops |
|---|---|---|
| 10 | 1.54 | 3.321928 |
| 50 | 2.792 | 5.643856 |
| 100 | 3.316 | 6.643856 |
| 500 | 4.465 | 8.965784 |
| 1000 | 4.965 | 9.965784 |
| 1500 | 5.218 | 10.550747 |



**Bonus Implementation: Concurrent Node Joining with Failure**

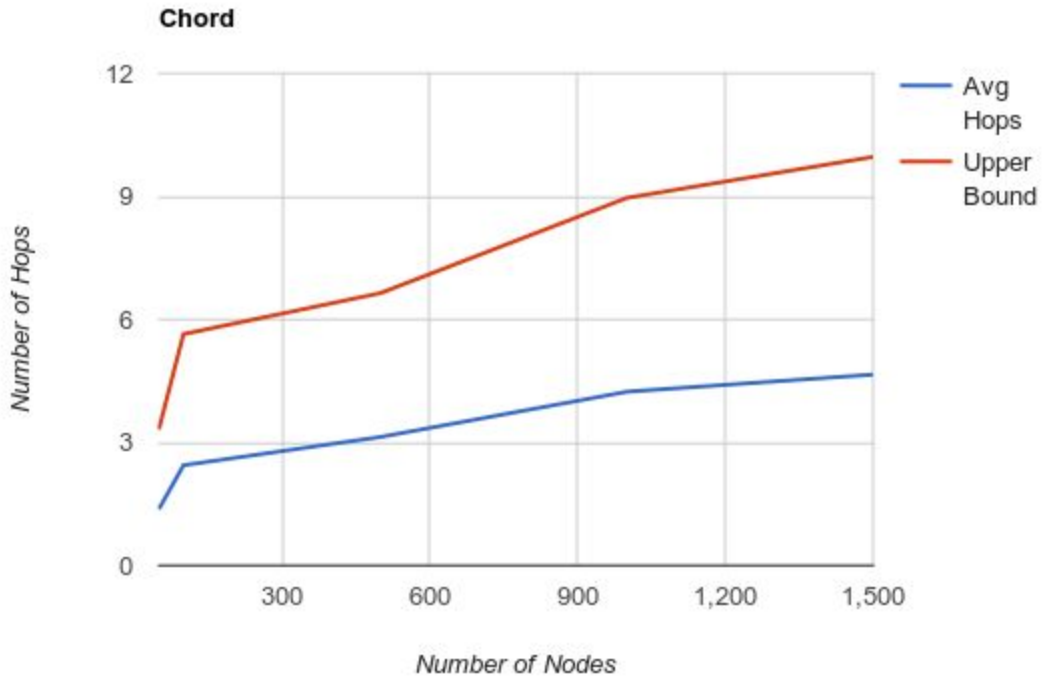
1. Number of Requests: 3

| Total Number of Nodes | No of Failed Nodes (20% of Nodes) | Average Number of Hops | Max bound of average hops |
|---|---|---|---|
| 10 | 2 | 1.541 | 3.321928 |
| 50 | 10 | 2.658 | 5.643856 |
| 100 | 20 | 3.15 | 6.643856 |
| 500 | 100 | 4.249 | 8.965784 |
| 1000 | 200 | 4.787 | 9.965784 |
| 1500 | 300 | 4.841 | 10.550747 |



2. Number of Requests: 5

| Total Number of Nodes | No of Failed Nodes (20% of Nodes) | Average Number of Hops | Max bound of average hops |
|---|---|---|---|
| 10 | 2 | 1.375 | 3.321928 |
| 50 | 10 | 2.445 | 5.643856 |
| 100 | 20 | 3.135 | 6.643856 |
| 500 | 100 | 4.238 | 8.965784 |
| 1000 | 200 | 4.657 | 9.965784 |
| 1500 | 300 | 4.895 | 10.550747 |




**Note:**

- The average number of hops are always less than log(n) to the base 2.

- Whenever a request is successfully completed, we print the key(request) and the node at which it was found.

**Note for concurrent node join and failure implementation:**

- When failure is implemented, we are terminating 20% of the total nodes and we receive log dead letters on those terminated nodes. Log dead letters comes when these active nodes are sending message to terminated nodes.
- These logs do not affect this system, and only signify that the message was sent to a node which we terminated as part of failure.
- After system shut down the console is flooded with log dead letters. Please scroll a little above to see the average number of hop counts.
- After failing the nodes, this network waits for 4 seconds as a buffer time to let the network stabilize.

**Output:**

In addition to the average number of hops in the ChordSequentialNodeJoin.scala output, we are also printing:

1. The names of the worker created
2. The finger tables of all nodes
3. Key ID and the node in which it can be found.

In addition to the average number of hops in the ChordConcurrentNodeJoinandFailure.scala output, we are also printing:

1. The names of the worker created
2. The name of nodes failed
3. Key ID and the node in which it can be found