**Distributed Operation System**

**Report**

**Chord Protocol Implementation**

**Submitted By:**

**Akshay Sehgal (UFID: 14167988)**

**Aswin Suresh Krishnan (UFID: 18901173)**

1. **Implement network join**

Our application is designed to join nodes into the network one by one. For each node, it calls a function called “nodeJoin” which places the node at its appropriate location and sets the successor of the new node to the node in the ring and also updates the predecessor of the node. Please refer the diagram below:

Setting Successor

Updating Predecessor

Now when the stabilize works on node 8. It updates the successor of 8 to include node 4 into the ring. The function of stabilizer will be explained later.

Already set at the time of node join

Updating Successor by Stabilizer

After the stabilizer works on the node preceding the new node, it just updates the successor of the node to the new node and thus the new node becomes the part of the ring.

1. **Stabilizer**

Stabilizer works as an independent process under a different GenServer and it keeps working in loop irrespective of state of the network.

How it works?

* The Stabilizer works on each node, going from one node to another by following its successor. It does show so in loop infinitely.
* When the stabilizer reaches a node it calls the successor of the node and check whether the predecessor of the successor node is same as the node the stabilizer is stabilizing. If it is so, it just moves to the next node.
* If the predecessor of the successor is some other node, which will be if some new node is about to join. Then the stabilizer updates the successor of the node to the new node as explained above.

1. **Routing**

The first thing in routing is setting the Finger Table.

**How our application is setting the finger table?**

The “fixFinger” is responsible for setting and updating the finger table of each node which joins the network dynamically. The call to fixFinger has been made from the Stabilizer module which runs for each node and stabilizes it by setting the correct value of the finger table by referring to only the active nodes.

Each update takes at most logN time where N is the number of nodes.

Screenshot of finger table output:

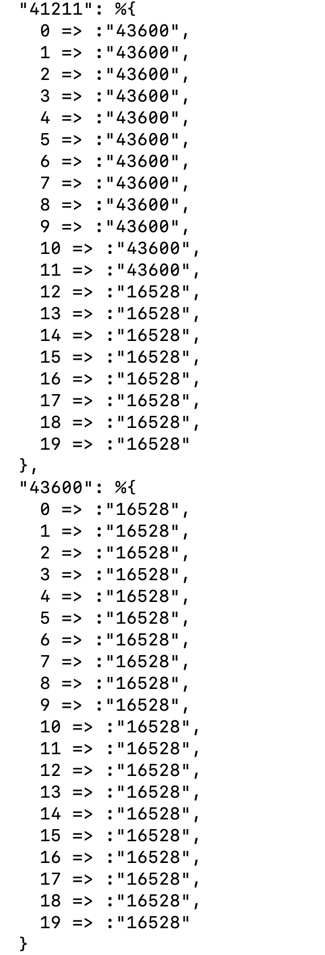


Figure 1Finger Table

**Look Up/ Routing**

* The find successor module tries to find the immediate node that is lesser than the final node to be searched. This it does by going through the finger table and finding the nearest value to the final key and calling find successor on that node.
* Each time we do that we move closer to the key and in the worst case scenario we will have to take logN hops to find it. This is when key resides in the node preceding the node from where we start the search.

**Explanation of the number of hops to find the key.**

Suppose that node n wishes to resolve a query for the successor of k. Let p be the node that immediately precedes k. We analyze the number of query steps to reach p. Recall that if n 6= p, then n forwards its query to the closest predecessor of k in its finger table. Consider the “i” such that node p is in the interval [n + 2i−1, n + 2i). Since this interval is

not empty (it contains p), node n will contact its ith finger, the first node f in this interval. The distance (number of identifiers) between n and f is at least 2i−1. But f and p are both in the

interval [n + 2i−1, n + 2i), which means the distance between them is at most 2i−1. This means f is closer to p than to n, or equivalently, that the distance from f to p is at most half the distance from n to p.

If the distance between the node handling the query and the predecessor p halves in each step, and is at most 2m initially, then within m steps the distance will be one, meaning we have

arrived at p.

1. **Sample Input**

mix run lib/Initializer.ex 25 2

**Sample Output**

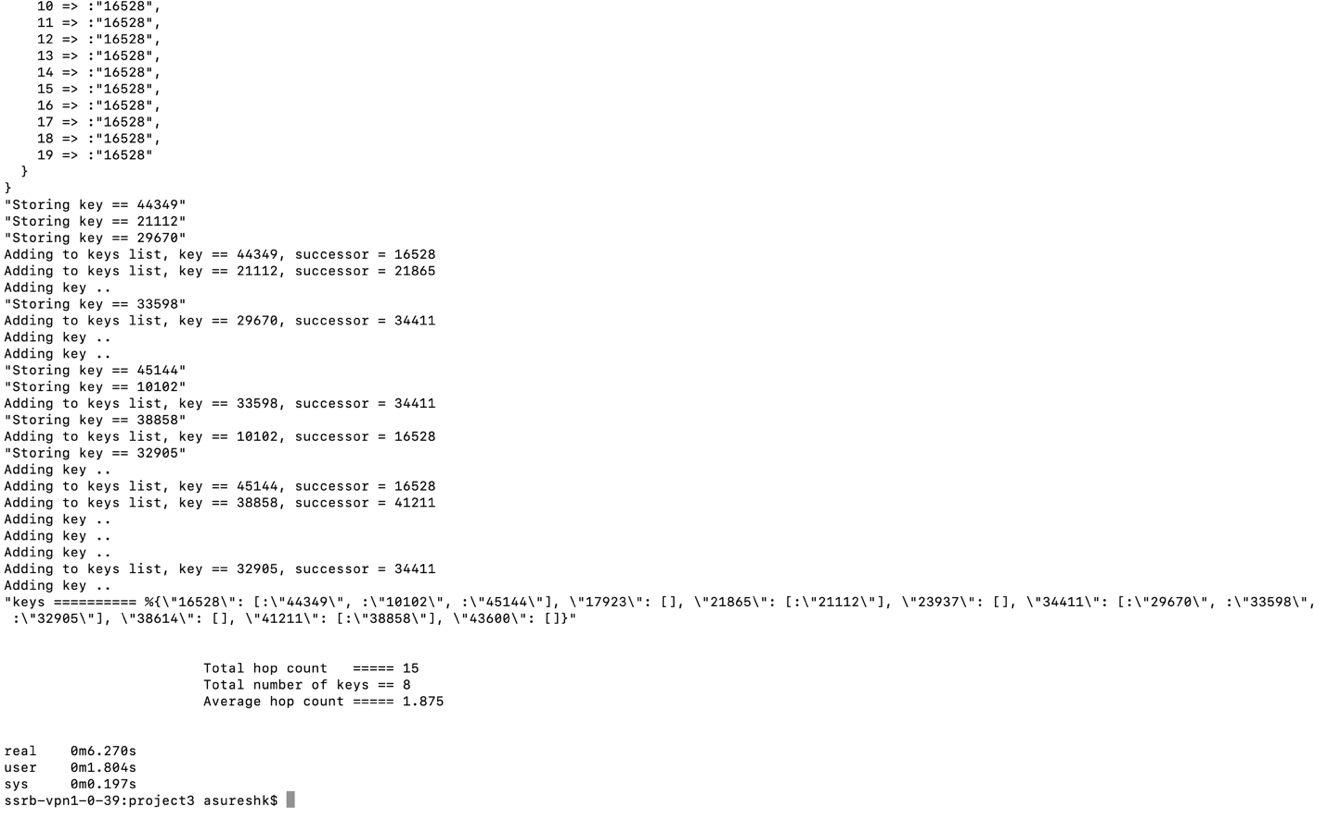


Figure 2 Sample Output

**4.1 Logic for Average Hop Count Calculation:**

* In the program, every time the node sends the request to find successor to the next node, it increases the hopCount by 1. At the end, We have the total keys returned and the total hops made. The program prints the Average Hop Count = Total Hops/Total number of keys returned.

**4.2 NumRequest Implementation**

Each node sends numRequests number of requests concurrently at an interval of 1 ms.

E.g. If numRequest is 3 we are returning number of nodes \* 3 keys from the system by initiating that many requests.

1. **Some Important Information:**
   * We have set the value of “m” in the program which is used to generate different node numbers by using SHA-1 hash generator. E.g. the value of m equal to 8 allows only 256 different codes to be generated.
   * For our final implementation, we have set “m” as 20.
   * Maximum value of number of nodes run: 2000

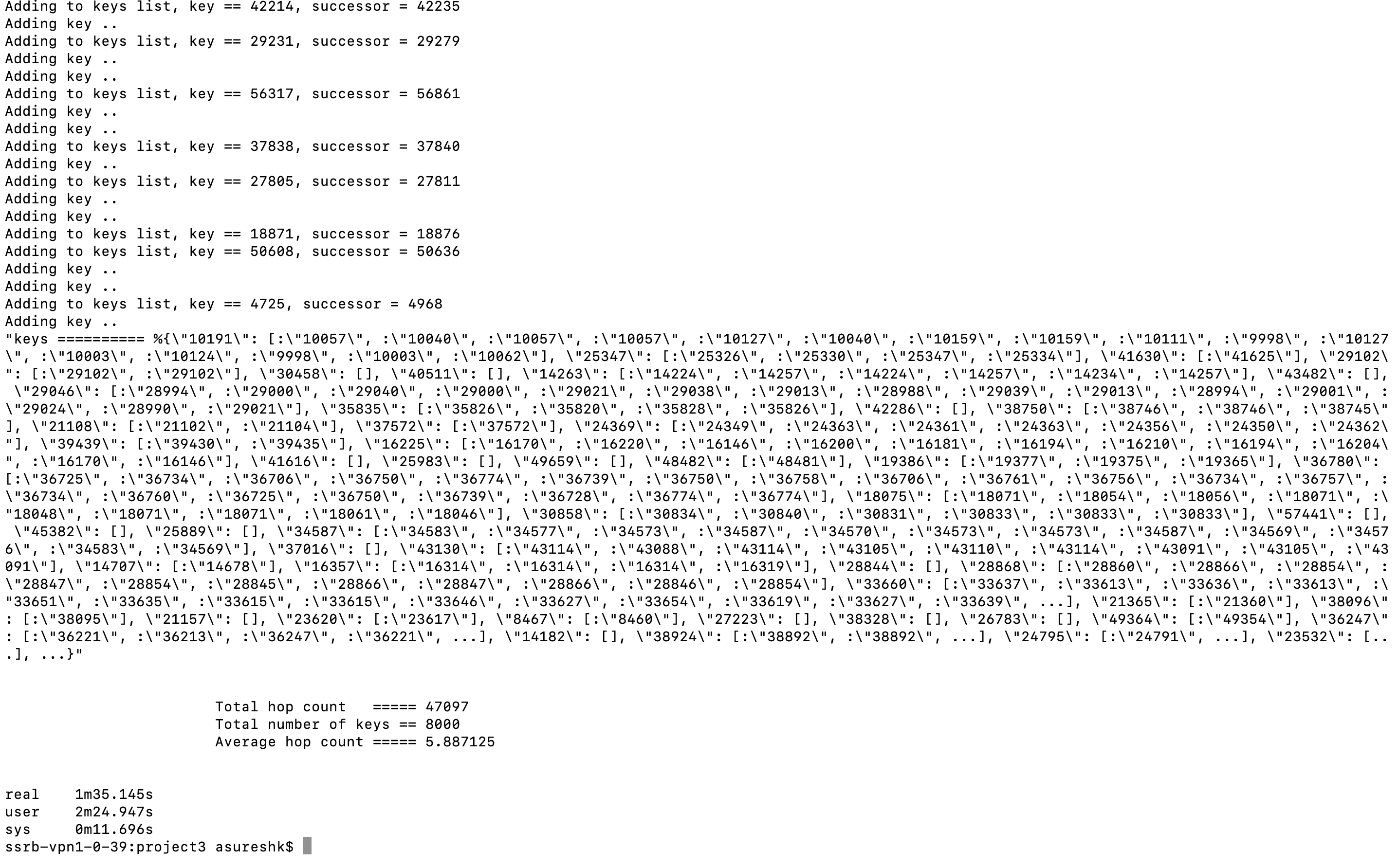


Figure 4 Results with NumNodes 2000

1. **Analysis**

We have analyzed data for one set of values.

M=20

Total keys searched in each round = 512

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | |  | |  | |
| **numNodes** | **numRequests** | | **Total Hop Count** | | **Average Hop Count** | |
| 4 | 128 | | 544 | | 1.0625 | |
| 8 | 64 | | 839 | | 1.6387 | |
| 16 | 32 | | 1045 | | 2.041 | |
| 32 | 16 | | 1718 | | 3.3554 | |
| 64 | 8 | | 2069 | | 4.041 | |
| 128 | 4 | | 2182 | | 4.2617 | |
| 256 | 2 | | 2822 | | 5.5117 | |
| 512 | 1 | | 3243 | | 6.34 | |

1. **Bonus Implementation**

**For the Bonus Implementation.**

* We have implemented Node failure in addition to NodeJoin in the initial network.

**Implementation:**

* As per the readme of the bonus part, the command to run the bonus has an extra input from the user which is the number of error nodes. The system will set up the network with the NumNodes and assign keys.
* After the normal setting up of network, the nodes will start to fail and the network would be stabilized by the Stabilizer and fixFinger module running continuously as a separate GenServer process.
* The leaving node update the successor of its predecessor to it’s successor and also assign the predecessor of its successor to nil.
* This is done to let the stabilizer know that the node has left and that is why the predecessor of the following node has been set to nil.
* When the Stabilizer visits the predecessor of the node that left. It checks for the predecessor of the following node which in this case will be nil. Having encountered such a situation, the Stabilize will update the predecessor of the node to itself that is the predecessor of the successor of node that left is set to the predecessor of the left node.

Please refer to the diagram:

Setting Predecessor to NIL



Updating Successor of the predecessor

Leaving Node



Now when the Stabilizer will visit Node 7, it will update the predecessor of 5 which is NIL, to 7 that is the new preceding node in the network.

The implementation screenshots are attached for better understanding:

**Bonus Program Input:**

mix run lib/Initializer.ex 25 2 10

**Bonus Program Output:**

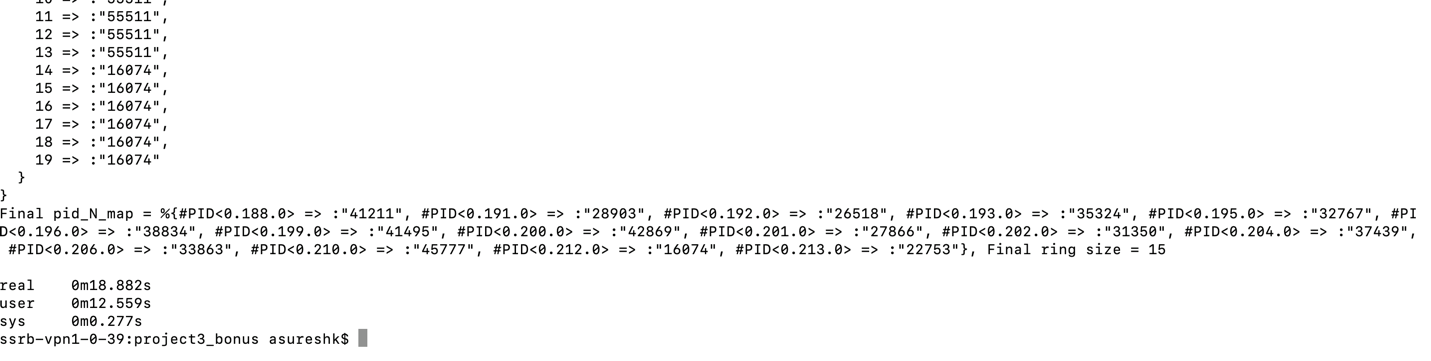
****

Figure 3 Bonus Implementation Output

**Explanation:**

The network will first get initialized with 25 nodes and 100 keys, we have set keys 4 times the number of nodes in our network. After completing set up, the 10 nodes will be removed from the network that is why the final ring size is 15 as you can see in the network. The stabilizer worked in the background to set the finger table for each and every node and hence stabilizing the network.