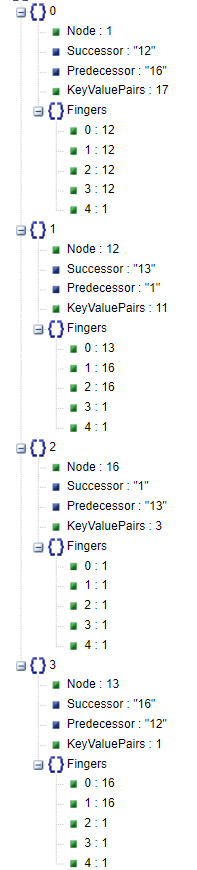
ANALYSIS

1. First, we check if the chord system we setup is according to the CHORD algorithm defined in the paper. We check a basic 4 node system in a 5-bit identifier system whether all the nodes have proper successors and predecessors as well as they are set to the correct fingers.



The four nodes in the system are 1, 12, 13 and 16

Consider Node 1. The node just before it in the circle is 16 and hence correctly as the predecessor. The node just ahead of it in the circle is 12 and rightly as the successor.

Now to check the fingers. Let’s list the lowest value (Or the starting value of the ith index) of finger the node 1 can have at each of the index

1. 1 + 2^0 = 2 – The closest node greater than or equal to 2 is 12 Hence it is rightly set
2. 1 + 2^1 = 3
3. 1 + 2^2 = 5
4. 1 + 2^3 = 9
5. 1 + 2^4 = 17

All except 4th index is before node 12 and hence the fingers are set to node 12. The 5th index starting range is 17 and the closest node after that is 1 (Itself).

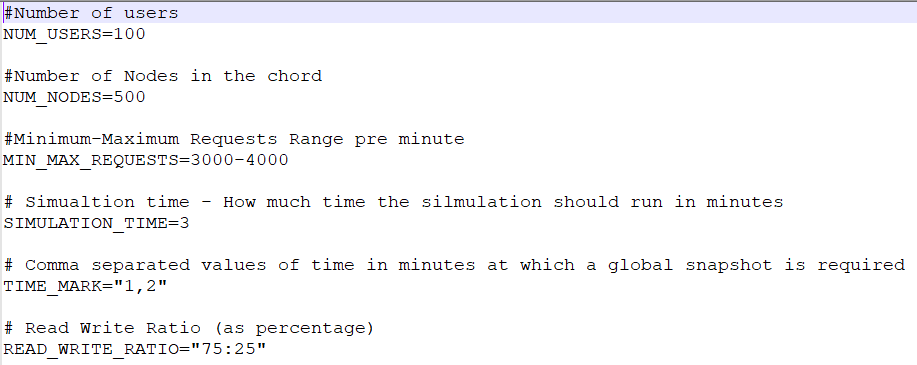
This can be similarly verified for all the other 4 nodes and they come out to be correct indicating a Node joining the CHORD system is placing itself correctly and the other tables are updating their fingers correctly.

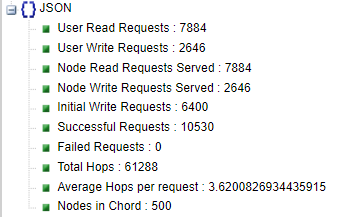
To test larger node system, we could come up with test case to capture all the node’s hash and a calculate their respective states and verify (Can be done but tedious. Can be an addition). But from this base case verification we can confirm the CHORD algorithm implemented is working.

This snapshot Containing the Node, its’s successor, predecessor and finger table along with the size of the data it stores is written to ‘ChordSystem.Json’.

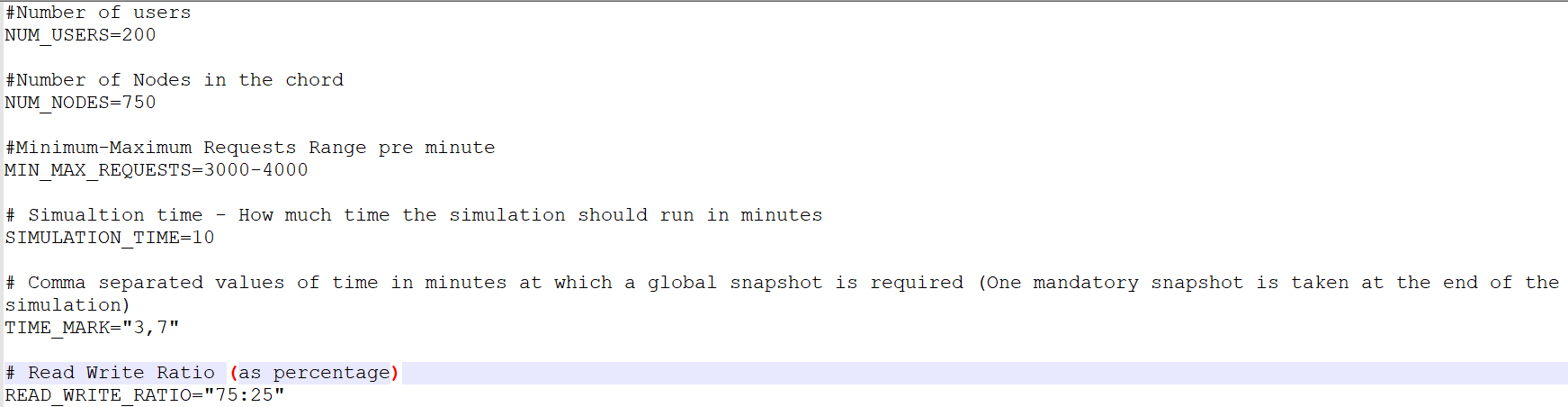
2. Simulation variables and the data from the snapshot. All this data is stored in ‘Global.Json’. This json file is in the path ‘/snapshot/{The time mark}/Global.Json’

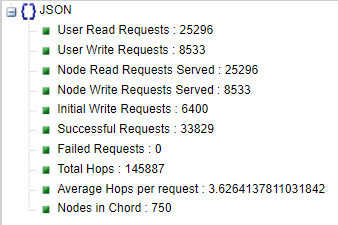
Simulation 1 :



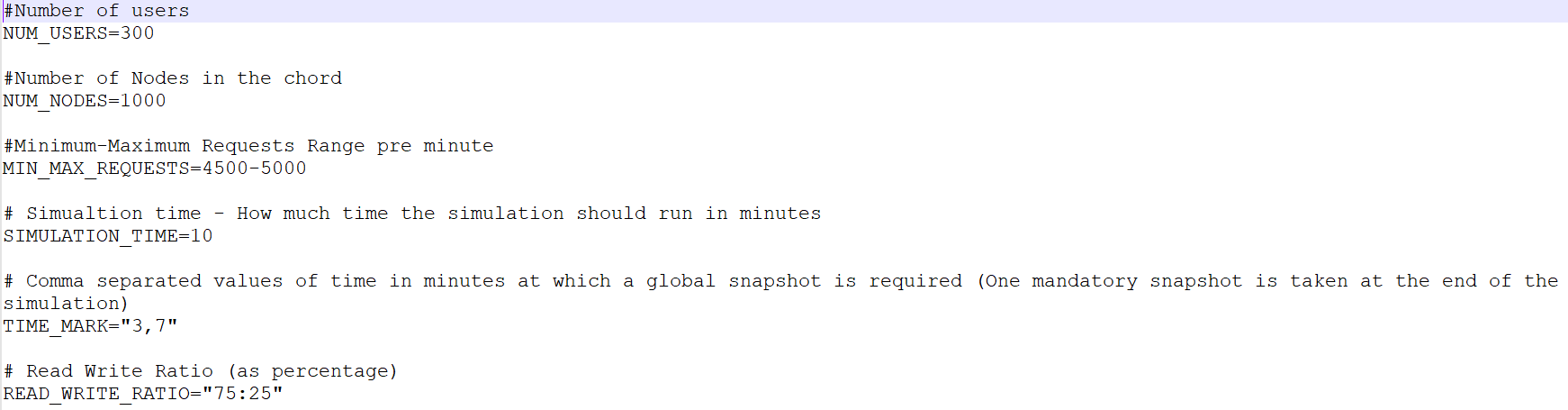


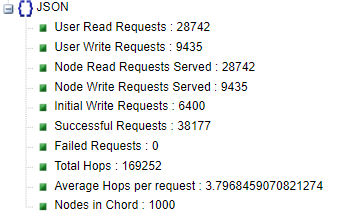
Simulation 2 :





Simulation 3 :





* The number of Users, nodes and the range of requests were changed for each simulation.
* Simulation 1 had 500 nodes and 3000-4000 requests. 100 Users – 3-minute Simulation
* Simulation 2 had 750 nodes and 3000-4000 requests. 200 Users – 10-minute Simulation
* Simulation 3 had 1000 nodes and 4500-5000 requests. 300 Users – 10-minute Simulation
* Each simulation was randomized to generate 75% read requests (As these are the blocking ones as they wait for the response from the server) and 25% write requests (Non-blocking as they fire and forget). This was done to test the load capacity.
* All the simulation fetches the same number of records and from the database and write a fixed portion of it initially to the Chord system (‘Initial Write Requests’ : Here 6400 records).

**Observations**

The main Idea of using finger tables to reduce the number of hops by trying to jump to finger connected nodes rather than each consecutive node in the circle if only successor node is being maintained.

* Without finger tables the number of hops would be a worst case of O(N) or an average case of N/2 at least (N being number of nodes in the system). From our simulation results it can be seen even when the nodes are increased from 500 to 750 to 1000 in consecutive simulations, the average number of hops remained at ~ **3.6**, which is a very small number. In systems with hundreds of thousands of nodes this number will make a huge difference.

Number of hops is calculated **as [ Total Hops] / [User Read Request + User Write Requests + Initial Write Requests]**

The circle for each of the simulation has total space of 2^18 = 262,144. Also, each node maintains a m reference to the finger tables. Therefore, each node has 18 nodes.

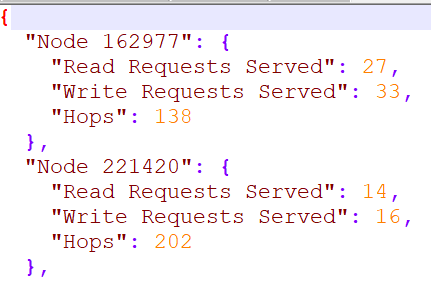
The number of failed requests were 0 in each simulation .We did not implement the scenario of a node leaving/failing which would have caused requests to fail. The sum [ Read/Write Requests Served ] – [Read Write Requests Made by the user] is the number of successful requests

In some instances (1 or 2) of simulation which were not captured, the system did face timeout issues though. This mostly happened when the number of users were set to a low number, which created a bottleneck in the Actor system.

The number of nodes also plays an important role. In instances where nodes were set to 25-50 keeping the request/min at 4000, we faced bottlenecks when fetching the data.

3. Every snapshot also contains ‘Nodes.json’ and ‘Users.json’ in the same path of Global.json’

Each ‘Nodes.json’ file stores data for each node like the number of read/write requests it served (That is how many keys were read and written to which it was directly responsible for). It also maintains the number of requests passed through the node.



Similarly, ‘Users.json’ stores data like the read and write requests it made.

