**COP5615- Distributed Operating System Principles**

**Project 3: Chord**

**Rishitha Reddy Kallu**

**UFID: 20015890**

**Abhijith Chandra Mergu UFID: 33991217**

**Input:**

The input to the program is numNodes and numRequests where numNodes is the number of peers to be created and numRequests is the number of requests which each peer should make.

**Output:**

The output would be average hop count of the nodes which are to be traversed for the message to be delivered.

**What is working:**

1. We have implemented the Chord protocol which is mentioned in the research paper.
2. We implemented different actors like finding successor, searching for closest preceding node, stabilize the nodes, update finger table periodically, notify nodes about predecessor change.
3. Average number of hops is calculated as total number of hops divided by product of number

of nodes and number of requests.

1. The average hop count is done in O(log n) time complexity.

**Implementation:**

The program involves the following major components:

1. Closest Preceding Node- This function looks for the highest predecessor of the node in the finger table.
2. Find Successor- This function returns the successor for the search node if search node is present between node and the successor, else it will return the successor of the closest preceding node.
3. Node- When started, create a new chord ring or network with no predecessor and successor as itself. If new node comes to join a chord ring, add it to the ring with no predecessor and the value of the successor from find successor function.
4. Stabilize- Function to check the node’s successor to update the finger table and later notifies the successor with the node.
5. Notify- If predecessor is null or new node is present between the predecessor and the node, then predecessor is set to the new node.
6. Fix Fingers- This function is called periodically to change the finger table entries to fix the table.

The workflow for the program is as follows:

* When the program starts, start will initialize monitor and node spawner.
* Node spawner will spawn the nodes and sends monitor to add the node to its list.
* We create a new network or chord ring and fill the finger table for the nodes present in the network size.
* A worker is spawned, then we stabilize and fix the finger table for the nodes with the help of fix fingers function.
* For stabilizing, get the predecessor of the node and compare to check if it is present in between the node and the successor. Then update the finger table of the node with the new successor. Later notify the successor with the node.
* In notify, we check if predecessor is null or if the new node is present in between predecessor and the node, then we update the predecessor with the new node.
* If a new node tries to join the chord ring, the predecessor is initialized as nil and we call a function to find successor.
* For finding successor, we check if the key node is present in between the node and the successor. If it is true, we return the successor else we return the successor of the closest preceding node.
* For finding the closest preceding node, we search for the highest predecessor of the node in the whole finger table.
* After getting the successor and updating the finger table, a new worker is spawned, then the finger table is stabilized again using fix fingers function.
* In fix fingers, we check if the finger index(size of the finger table) is equal to 1 added to the network size, then find the successor for the node with search node as node added to 2 power of network size.
* The worker which is spawned will return the node to the monitor
* Monitor will keep on adding nodes into the NodeList and then inserts the key into KeyList.
* Then key search will start for searching keys and the hop count will get increased.
* When the key is found, keyFound message is sent to the monitor where we calculate the average number of hops as total number of hops divided by product of number of nodes and number of requests.
* After all the searches are done, searchKeys function will send searchDone to the monitor where we will print the average hops.

**How to run:**

1. Unzip the Project3 folder.
2. Open the file using the IntelliJ editor.
3. Open terminal and initialize the erlang shell using the erl command.
4. Execute the command “project3: start(<numNodes>, <numRequests>)” where the values are number of peers and num of requests respectively.
5. The output on the terminal will display the average hop count.

**Statistics:**

|  |  |  |  |
| --- | --- | --- | --- |
| Peers | Message Requests | | Average Hop Count |
| 10 | 20 | 0.79 | |
| 50 | 35 | 1.8297 | |
| 100 | 25 | 2.2668 | |
| 200 | 20 | 2.746 | |
| 500 | 30 | 3.4742 | |
| 1000 | 35 | 3.9716 | |
| 2000 | 30 | 4.4534 | |
| 3000 | 45 | 4.7535 | |
| 4000 | 35 | 4.9845 | |
| 5000 | 30 | 4.9487 | |
| 6000 | 25 | 5.2453 | |
| 7000 | 30 | 5.3286 | |
| 8000 | 30 | 5.4753 | |

**Largest network managed:**

Network with 8000 nodes and 30 message requests is done with an average hop count of 5.4753

**Outputs:**

Input: numNodes= 2000, numRequests= 40

Command: project3: start(2000,40).

Output:



Input: numNodes= 3000, numRequests= 45

Command: project3: start(3000,45).

Output:



Input: numNodes= 4000, numRequests= 35

Command: project3: start(4000,35).

Output:



Input: numNodes= 5000, numRequests= 30

Command: project3: start(5000,30).

Output:



Input: numNodes= 6000, numRequests= 25

Command: project3: start(6000,25).

Output:



Input: numNodes= 7000, numRequests= 30

Command: project3: start(7000,30).

Output:



Input: numNodes= 8000, numRequests= 30

Command: project3: start(8000,30).

Output:

