Project3 Report

Group member:

Qinxuan Shi	UFID:	8351-8162
Jiahui He	UFID:	7717-1034

How to start:

1. starter:start().
2. project3 numNodes numRequests
3. The result will show in the console like “Average hops for a message:”.

What is working:

This project implements the chord distributed hash table protocol by using the APIs showed in the research paper.

Find_successor(id) function works to find the successor of id; it returns the successor of the current node if the value of id between the current node and it’s successor; or it returns the node which has the largest identifier smaller than id.
Closest_preceding_node(id) function helps the function find_successor() to get the largest identifier from finger table that smaller than id.
Create(), join(), are used to create the new node, the difference is that create() is used for the first node to create the chord ring, and join() is used for other nodes to join the existed node and set its successor.
Stabilize() works to check if the node’s successor’s predecessor is the node, if not, this function is used to update and call the function notify() to update the predecessor.
Notify() is used to update the predecessor of a node after the stabilize().
Fix_fingers() is used to refresh the finger table of each node by calling periodically.

To each node the request generates a random number and find the correct node to store the number. After finding the correct node, the node will send the hops the request travels to count process. The request will send one time for each second.

When a new node joins the chord ring, the node that calls join() function will also call find_successor() function to look for the correct successor for the new node. Then they will call stabilize() and notify() to update their successor and predecessor. The fix_fingers() function will also be called to refresh the finger table.

As is required in the instruction, we use the SHA256 to hash the pid of a node as its id and hash the key value as its id.

As is shown in the next section, the growing of the average hops for a message is getting slower when the number of nodes is getting bigger, which in accordance with the complexity of the chord protocol using scalable key location, O(logN). The maximum number of nodes we managed to deal with is 20000 and each node will send two requests.

Although the complexity of average hops for a message is O(logN), it takes a really long time for nodes to stabilize and update their finger table.

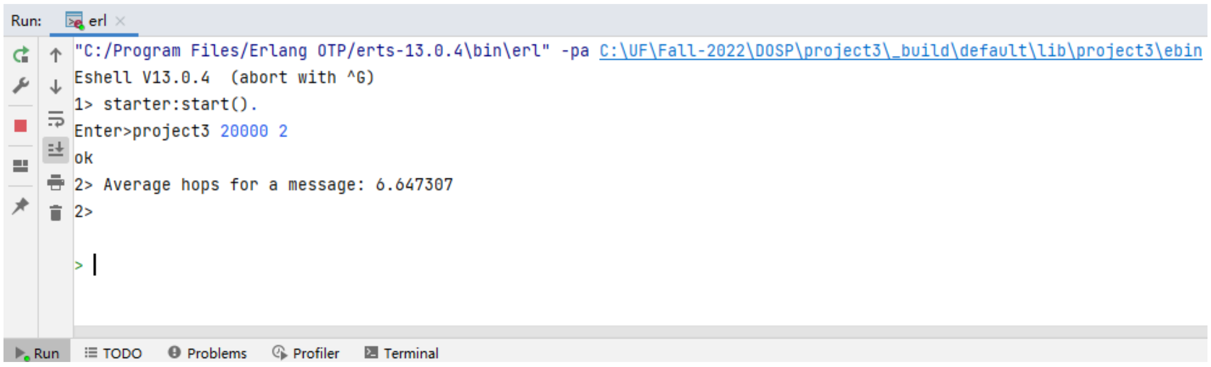
The chord protocol works much better to the larger number of nodes compared to small number of nodes.

Results:

We set the number of requests to 2, and tried several times with different number of nodes, here is the results:

Number of Nodes	Average hops for a message
10	0.89
50	2.02
100	2.73
500	4.04
1000	4.47
2000	4.94
5000	5.63
10000	6.15
20000	6.64

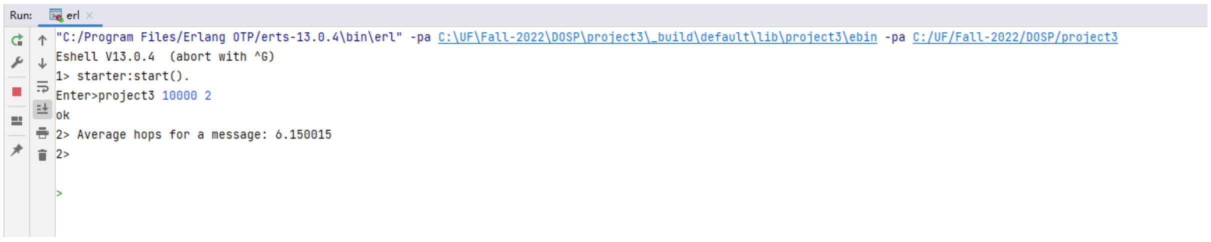
Here is the screenshot for the largest network we managed to deal with, the largest number of nodes is 20,000 and the request number is 2.



```
"C:/Program Files/Erlang OTP/erts-13.0.4\bin\erl" -pa C:\UF\Fall-2022\DOSP\project3\_build\default\lib\project3\ebin
Eshell V13.0.4  (abort with ^G)
1> starter:start().
Enter>project3 20000 2
ok
2> Average hops for a message: 6.647307
2>

>
```

This is the screenshot for the node number 10000.



```
"C:/Program Files/Erlang OTP/erts-13.0.4\bin\erl" -pa C:\UF\Fall-2022\DOSP\project3\_build\default\lib\project3\ebin -pa C:/UF/Fall-2022/DOSP/project3
Eshell V13.0.4  (abort with ^G)
1> starter:start().
Enter>project3 10000 2
ok
2> Average hops for a message: 6.150015
2>

>
```