COP5615: Distributed Operating Systems Principle
Project – 3
(Chord - P2P System and Simulation)

Submitted By: Srishti Jaiswal [UFID- 80385159]

Outline:

I used the Actor Model to construct the chord protocol in Erlang and an object access service was utilized to put the protocol into practice. To store my hash value, I constructed a finger table, and then specified some check criteria to see if it terminated.

Definition:

The protocol and technique for a distributed hash table that is peer-to-peer is called Chord. A distributed hash table holds key-value pairs by allocating keys to various computers (referred to as "nodes"); each node keeps the values for all the keys for which it is accountable. As nodes enter and exit the system, Chord keeps routing information. The purpose of this project is to develop F# and demonstrate its utility using the actor model, the Chord protocol, and a straightforward object access service.

How to use:

1. Cd the project source file
2. Start erl shell and compile main, node_chord_file, finger_table_file and request_file using c(file_name).
3. Run main.erl and allocate the desired parameters (Number of nodes, Number of requests).
4. Run node_chord_file.erl
5. Run finger_table_file.erl
6. Run request_file.erl
7. To begin putting the chord protocol into practice, follow the inputs produced by the code.

The working model includes:

- The simulation of chord p2p lookup is successful.
- Each node has a finger table attached to it that holds information in relation to the node's ID.
- For simulation purposes, every node creates a unique random key for every request.
- After all the nodes have been established, a key may be checked on any node at random. Even if the key isn't locally available on that node, the search is routed to the closest node. Finger tables are utilized for this.
- The number of hops increases with each lookup for each key discovered.
- Each node executes the specified number of requests for a particular amount of time, prints the typical number of hops, and then leaves.
- The typical number of network hops needed to look for the key is calculated.

Largest Network you managed to find:

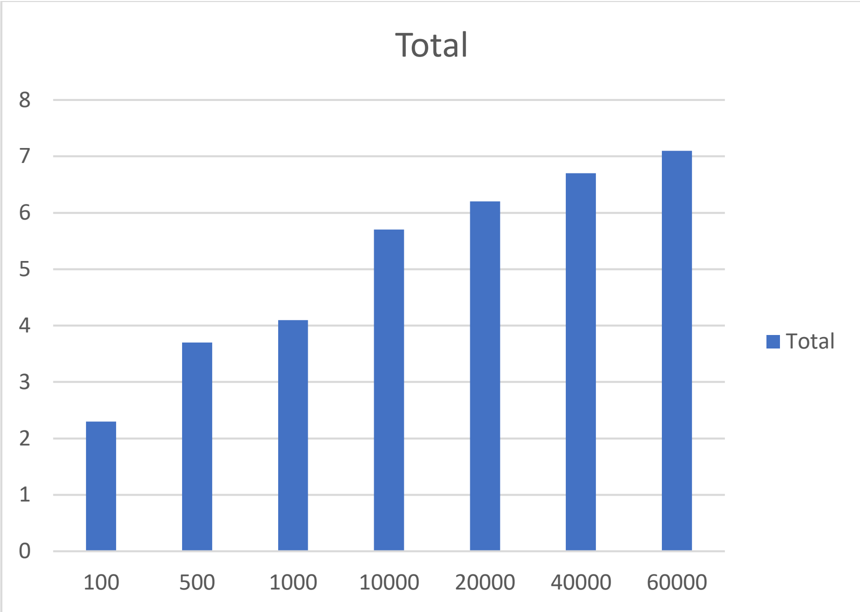
Average hops count = Total hops count / Total number of requests

The largest network we managed to deal with is **60,000** nodes with an average hop count of **7.1**.

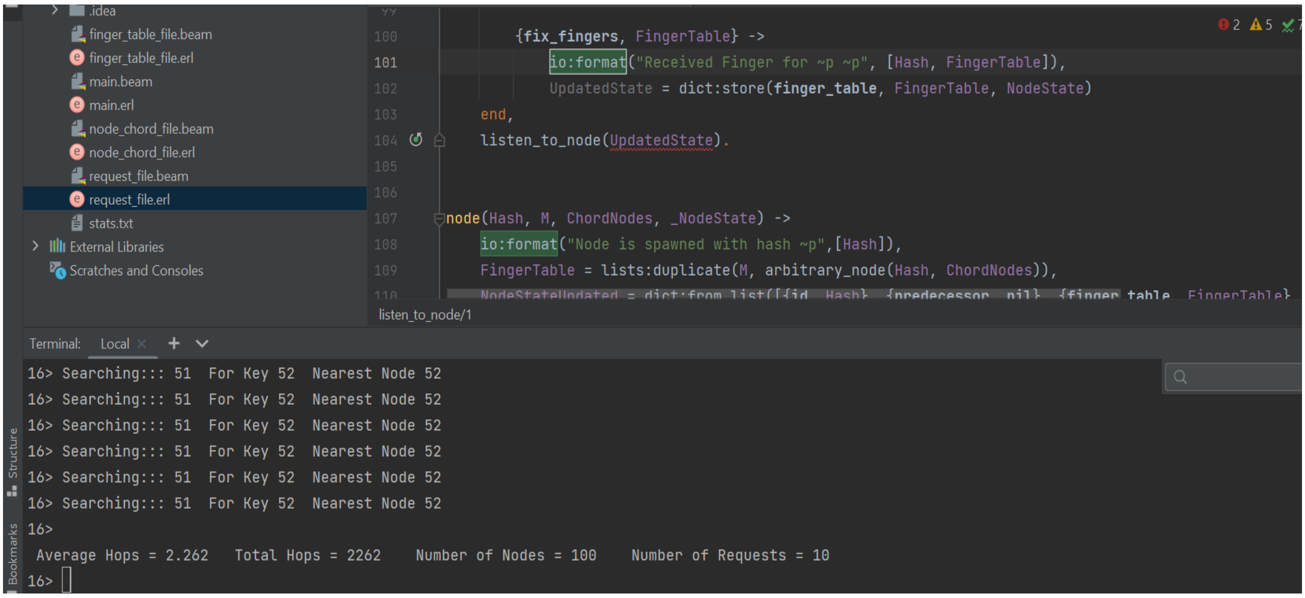
Observations:

- The average number of hops rose as more nodes joined the network overall.
- The average number of hops dropped as the number of requests from each node to the same network rose.
- The average number of hops to reach the target node was in the range of 2.3-7.1 for number of nodes ranging from 100-60000.

Number of Nodes	Average Hops
100	2.2
500	3.7
1000	4.1
10000	5.7
20000	6.2
40000	6.7
60000	7.1


Total
8
7
6
5
4
3
2
1
0
100
500
1000
10000
20000
40000
60000
Total

Results:


.idea
finger_table_file.beam
finger_table_file.erl
main.beam
main.erl
node_chord_file.beam
node_chord_file.erl
request_file.beam
request_file.erl
stats.txt
External Libraries
Scratches and Consoles
{fix_fingers, FingerTable} ->
io:format("Received Finger for ~p ~p", [Hash, FingerTable]),
UpdatedState = dict:store(finger_table, FingerTable, NodeState)
end,
listen_to_node(UpdatedState).
node(Hash, M, ChordNodes, _NodeState) ->
io:format("Node is spawned with hash ~p",[Hash]),
FingerTable = lists:duplicate(M, arbitrary_node(Hash, ChordNodes)),
listen_to_node/1
Terminal: Local
16> Searching::: 51 For Key 52 Nearest Node 52
16> Searching::: 51 For Key 52 Nearest Node 52
16> Searching::: 51 For Key 52 Nearest Node 52
16> Searching::: 51 For Key 52 Nearest Node 52
16> Searching::: 51 For Key 52 Nearest Node 52
16> Searching::: 51 For Key 52 Nearest Node 52
16>
Average Hops = 2.262 Total Hops = 2262 Number of Nodes = 100 Number of Requests = 10
16>


1> main:main(60000,10).
true
2>
27268 <0.82.0>
2> Node is spawned with hash 272682>
33821 <0.83.0>
2> Node is spawned with hash 338212>
40146 <0.84.0>
2> Node is spawned with hash 401462>
63347 <0.85.0>
2> Node is spawned with hash 633472>
58245 <0.86.0>
2> Node is spawned with hash 582452>
44439 <0.87.0>
2> Node is spawned with hash 444392>