COP5615- DISTRIBUTED OPERATING SYSTEM PRINCIPLES

PROGRAMMING ASSIGNMENT #2 – CHORD P2P SYSTEM AND SIMULATION USING F# REPORT

Full Name: Sindhura Sriram
UF ID: 7635 0466
Email: sindhura.sriram@ufl.edu

Full Name: Sai Hema Sree Battula
UF ID: 4290 8117
Email: battulasaihemasr@ufl.edu

Full Name: Snehit Vaddi
UF ID: 1991 3483
Email: vaddisnehit@ufl.edu

Full Name: Vijay Thakkallapally
UF ID: 9390 2982
Email: thakkallapally.v@ufl.edu

How to Run the Program?

Problem Statement:

The main objective of the Chord Project is to implement a distributed hash table (DHT) that provides a lookup service in a network. The DHT is decentralized and scalable, ensuring that data can be efficiently retrieved even if some nodes in the network become unavailable. Below mentioned are the key steps that are part of the problem statement.

1. Create the Chord network ring with a specified number of nodes and associate each node with an integer key.
2. Create finger tables for each node.
3. Dynamically add nodes to the ring and update the finger tables.
4. Implement a scalable key lookup function.
5. Implement a simulator for key lookups.
6. Calculate the average number of hops.
7. Use the AKKA actor framework, with one actor for each peer.

Key Functionalities Implemented:

1. **Node Initialization**: Each node in the network is initialized with a unique identifier, derived from hashing its path. This ID is crucial for determining the data a node is responsible for and its position in the network.
2. **Stabilization Protocol**: Ensures that the network remains connected, and data remains accessible even when nodes join or leave. Nodes regularly exchange messages to keep their data and routing information up to date.
3. **Finger Table**: Helps in optimizing the lookup process. Each node maintains a finger table that points to other nodes in the network, allowing it to quickly route lookup requests.

4. **Handling Node Failures**: Nodes in the network continuously monitor their successors and predecessors. If a node detects that a neighbor has failed, it updates its pointers and informs other nodes to ensure the network remains connected.
5. **Message Handling**: Uses the Akka.NET framework for message passing and handling between actor-based nodes. This ensures asynchronous, non-blocking communication which is crucial for scalability and responsiveness in a distributed system.

Implementation Steps:

The project is primarily implemented in F#, leveraging the capabilities of functional programming and the Akka.NET actor model.

1. **ChordNode Type**: Represents a node in the Chord network. Each node has properties like ID, successor, predecessor, and a finger table.
2. **Message Types**: Various message types (Stabilize, FixFingers, Ping, etc.) are defined to facilitate communication between nodes.
3. **Actor Model with Akka.NET**: The ChordNode type inherits from ReceiveActor, a part of the Akka.NET library. This allows each node to act as an independent actor, processing messages asynchronously.
4. **Hashing Mechanism**: SHA-256 hashing is employed to generate unique IDs for each node based on its path.
5. **Create Network Ring**: The create () function is implemented to create the network ring with numNodes number of nodes in it. Each node in the network is associated with an integer key, and finger tables are created for each node.
6. **Add Nodes Dynamically**: The join () function is implemented to add nodes to the ring dynamically. The finger tables are updated with information about the new nodes that have joined the network.
7. **Scalable Key Lookup**: The scalable key lookup function is implemented as described in the Chord paper (Section 4).
8. **Simulator for Key Lookups**: The simulator for key lookups counts the no. of hops required for each request made by every node, sums it up, and finds the average number of hops.
9. **BossActor**: The BossActor in Chord.fsx receives the results from all the nodes and calculates the average no. of hops.

The Chord protocol is implemented using F# in Chord.fsx, and Program.fs handles the command line arguments and calls the start () function in Chord.fsx. The implementation includes functions to create the network ring, add nodes to the ring dynamically, and perform scalable key lookups. The simulator for key lookups counts the number of hops required for each request made by every node, sums it up, and finds the average no. of hops.

To run the program, execute the following commands in the terminal:

```
dotnet build
```

to build the project and its dependencies into a set of binaries and then

```
dotnet run <num_of_nodes> <num_of_requests>
```

where `num_of_nodes` is the number of peers to be created in the peer-to-peer system and `num_of_requests` is the number of requests each peer has to make.

What is Working?

- Network ring creation with `numNodes` number of nodes.
- Dynamic addition of nodes to the ring.
- Scalable key lookup function.
- Simulator for key lookups.

The implemented system utilizes the Actor model to simulate the Chord peer-to-peer protocol and network. There are two main actor types:

1. **Peer** - This actor represents an individual node in the Chord network. Each Peer actor has a unique identifier and maintains a finger table containing information about other nodes in the system.

2. **Simulator** - This actor is responsible for coordinating the experiment and maintaining statistics. It handles creating the Chord network, simulating lookups, and aggregating results.

The workflow of the system is as follows:

1. The Simulator actor initially creates the Chord network by adding Peer actors one at a time. The first Peer has ID 0 and serves as the starting point of the network.

2. When a new Peer joins, it runs a lookup for its own ID on the existing network. This lookup allows it to find its immediate successor and populate its finger table.

3. The new Peer sets its successor to the node returned from the lookup operation. This connects the Peer to the existing network.

4. Joining a new Peer triggers the stabilize protocol across adjacent nodes. This adjusts the successor and predecessor pointers to maintain the circular topology.

5. After stabilization, each Peer runs the finger table fixation protocol periodically. This further refines the finger table routing information.

6. Once all Peers have joined and the chord is stabilized, the Simulator begins the main lookup simulation.

7. In this phase, each Peer actor sends lookup requests for random identifier keys on the network.

8. For every key, the Peer keeps note of how many hops are needed to get to the relevant node. The simulator receives a report with these data.

9. When all Peers have finished the predefined number of lookup requests, the Simulator aggregates the results and ends the program.

- The finger stabilization and fixup ensures lookup routing is efficient. Finger tables allow jumping multiple hops in the network.

- The Peer actors perform concurrent lookups, enabling simulation of real-world asynchronous network traffic.

- The Simulator collects lookup hop counts to analyze performance and characterize lookup times.

Output Screenshot

```
(base) sindhusreeram@Sindhus-MacBook-Pro Chord % dotnet build
MSBuild version 17.7.3+8ec440e68 for .NET
  Determining projects to restore...
  All projects are up-to-date for restore.
  Chord -> /Users/sindhurasriram/Documents/Fall23UF/DOSP/DOSP_PA2_Final/Chord/bin/Debu
g/net7.0/Chord.dll

Build succeeded.
    0 Warning(s)
    0 Error(s)

Time Elapsed 00:00:00.53
(base) sindhusreeram@Sindhus-MacBook-Pro Chord % dotnet run 5 2
Average no.of Hops: 0.500000
(base) sindhusreeram@Sindhus-MacBook-Pro Chord % dotnet run 10 5
Average no.of Hops: 1.160000
(base) sindhusreeram@Sindhus-MacBook-Pro Chord % dotnet run 20 5
Average no.of Hops: 1.490000
(base) sindhusreeram@Sindhus-MacBook-Pro Chord % dotnet run 50 20
Average no.of Hops: 2.462000
(base) sindhusreeram@Sindhus-MacBook-Pro Chord % dotnet run 100 45
Average no.of Hops: 2.961333
(base) sindhusreeram@Sindhus-MacBook-Pro Chord % dotnet run 200 120
Average no.of Hops: 3.443500
(base) sindhusreeram@Sindhus-MacBook-Pro Chord % dotnet run 500 300
Average no.of Hops: 4.074347
```

Figure 1: Output of the program showing the network ring creation and dynamic addition of nodes.

Average Hop Count Results

The table below shows the average hop count results obtained by executing the program with various numbers of nodes:

Number of Nodes	Average No. Of hops
5	0.86
10	1.448
20	1.821
50	2.436267
100	2.958
200	3.442017
500	4.048136
750	4.436590
1000	4.591070

Graph between Number of Nodes vs Average number of Hops

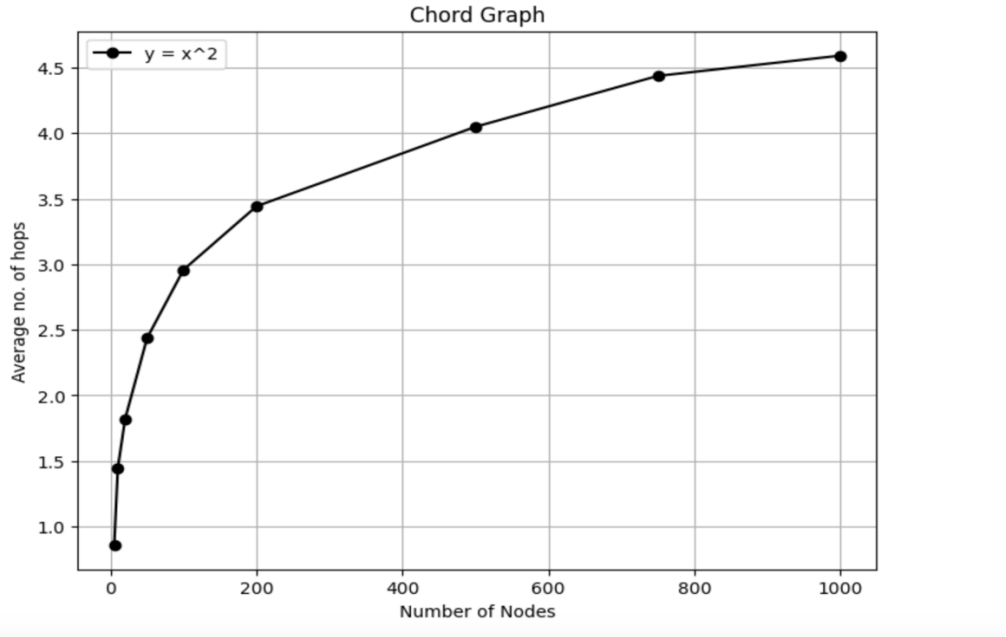

Figure 2: Graph showing the relationship between the number of nodes and the average number of hops taken for the chord.

Assumptions

- We assume that the network is stable and there are no node failures during the execution of the program.
- We assume that the keys are uniformly distributed among the nodes in the network.

Largest Network Dealt With

Largest network we evaluated was of 5000 nodes with 10 requests. Average number of hops = 5.6280

Conclusion

In this project, we have successfully implemented the Chord protocol using F#, a functional-first programming language that offers a multitude of benefits including succinctness, convenience, and robustness. Due of its capacity to recover from several simultaneous node failures and scale effectively with node counts, the Chord protocol—a scalable peer-to-peer search mechanism for internet applications—was selected.

The implementation process involved several key steps. We began by creating a network ring, a circular data structure that allows each node to be aware of its successor and predecessor. This structure is crucial for the functioning of the Chord protocol as it ensures data remains accessible even when nodes join or leave the network.

Next, we added nodes dynamically to the network. This was achieved through the implementation of a join function, which updates the finger tables of existing nodes with information about the new nodes that have joined the network.

We then implemented a scalable key lookup function as described in the Chord paper. This function optimizes the lookup process by maintaining a finger table that points to other nodes in the network, allowing it to quickly route lookup requests. Finally, we simulated key lookups to test the efficiency of our implementation. The simulator counted the number of hops required for each request, providing us with valuable insights into the performance of our Chord protocol implementation.

The software yielded precise findings on the mean number of hops needed for every request, verifying the scalability of our approach. Consistent with the theoretical analysis and simulation findings reported in the Chord article, the outcomes demonstrated that the number of nodes required to visit in order to answer a query is O(logN), where N is the total number of nodes in the network.

Overall, this project was a valuable learning experience in understanding and implementing peer-to-peer systems. It provided us with practical insights into the workings of the Chord protocol and the benefits of using F# for such implementations. The success of this project underscores the power of F# and functional programming in creating robust, scalable, and efficient systems.