COP5615 - Distributed Operating System Principles Programming Assignment 2

Bhavana Aleti
bhavanaaleti@ufl.edu
UF ID: 8243-3275

Abhinav Mandala
abhinav.mandala@ufl.edu
UF ID: 2415-2509

Jyothi Santoshi Karuturi
karuturi.j@ufl.edu
UF ID: 2941-1031

Pranay Kumar Reddy Pinninti
p.pinninti@ufl.edu
UF ID: 6539-0190

1 Compile and Run Instructions

- **Software Requirements -** In order to compile and run the application, you must ensure that your system is equipped with .NET version 7.
- Build the code using the command
  dotnet build
- Run the program using the command
  dotnet run $< numNodes > < numRequests >$

2 Program Files

- **Team_30.fsproj** - This is a .NET project file, usually named YourProjectName.fsproj, where "YourProjectName" matches your project's name. It stores metadata and configuration settings for your .NET project.
- **Program.fs** - The method **main** in this file, serves as the program's entry point and expects two command-line arguments: the number of nodes and the number of requests. If the correct arguments are provided, it starts the Chord network simulation by calling the function **SimulatorToSendRequests** using these parameters. If the arguments are missing or incorrect, it displays a usage message.
- **chord.fsx** - In this program file we implemented simulation of the Chord protocol using the Akka.NET actor model framework. It creates a Chord ring with a specified number of nodes. The nodes join the ring dynamically, and the simulation includes processes for stabilizing the ring, fixing finger tables, and handling key lookups with hop count calculations. The average number of hops for key lookups across the nodes is then calculated and printed.

3 What is working

Chord is a distributed network protocol that maps keys to nodes using a consistent hash function, ensuring even load distribution and minimal key movement when nodes change. In N-Node network each node stores information about O(log N) others, and key lookups require O(log N) messages, providing efficient scalability for large networks.

Below are the functionalities implemented in Chord Protocol -

- **Create Chord** - The function **Create** creates a chord space of size 2^m.The function initializes a Chord node by setting its predecessor and then populating its finger table with its node ID, and scheduling two recurring tasks. The first task, **StabilizeChord**, ensures network stability, while the second task, **FixFingerTables**, maintains finger table accuracy.
- **Join Node** - This function is used when a new Chord node wants to join an existing Chord network. First, it sets the predecessor of the new node to -1, indicating it doesn't have a predecessor yet. Then, it retrieves the actor reference of the existing node by constructing the actor's path. Next, it sends a **GetSuccessorNode** message to the existing node to retrieve its successor node information, helping the new node establish its position in the Chord network. This process facilitates the integration of the new node into the Chord ring.
- **Scalable Key Lookup** - Chord uses a routing mechanism to optimize lookups. This involves a routing table called the finger table. The i-th entry in the finger table for node n points to the first node, denoted as s, that comes after n by at least $2^{(i-1)}$ on the identifier circle. This mechanism optimizes lookups by allowing each entry in the finger table to identify a node located at a specific power of two distance from the current node on the identifier circle. This ensures that Chord's routing is efficient and minimizes message overhead.
- **Stabilization** - In the function **stabilizeChord**, we retrieve the information about the successor of the current node in a Chord network. The **fingerTable** is an essential data structure in Chord, and the first entry (index 0) in this table represents the immediate successor of the node. To obtain details about the successor node, we access its Chord identifier and its network path. Using this identifier and path, we construct a reference to the successor node. Afterward, we send a message to the successor node, specifically asking for its predecessor's information by invoking the **GetPredecessorNode** message. This communication helps maintain accurate routing information and supports the Chord network's stability and efficiency.
- **Output** - We examine each node within the simulation, checking whether the node's request count matches the total number of requests. When these counts align, we increment the tally of nodes with completed requests. After all nodes have finished processing their requests, we calculate the average number of hops. This average is determined by dividing the sum of hops across all requests and nodes by the product of the total request count and the number of nodes in the simulation.

4 Execution Results

The image displays the relationship between the number of nodes and the corresponding number of hops required. Notably, it shows a consistent number of requests (i.e., 10) for each node. The data ranges from 5 nodes to as many as 15,000 nodes.

An important observation is that the number of hops consistently increases as the number of nodes increases as shown.

```
(base) pranayreddy@Pranays-MacBook-Air Team_30 % dotnet run 5 10
No of Nodes = 5 , No of Requests = 10
Average Number of Hops : 1.280000
(base) pranayreddy@Pranays-MacBook-Air Team_30 % dotnet run 10 10
No of Nodes = 10 , No of Requests = 10
Average Number of Hops : 1.700000
(base) pranayreddy@Pranays-MacBook-Air Team_30 % dotnet run 20 10
No of Nodes = 20 , No of Requests = 10
Average Number of Hops : 2.825000
(base) pranayreddy@Pranays-MacBook-Air Team_30 % dotnet run 50 10
No of Nodes = 50 , No of Requests = 10
Average Number of Hops : 3.900000
(base) pranayreddy@Pranays-MacBook-Air Team_30 % dotnet run 70 10
No of Nodes = 70 , No of Requests = 10
Average Number of Hops : 4.184286
(base) pranayreddy@Pranays-MacBook-Air Team_30 % dotnet run 100 10
No of Nodes = 100 , No of Requests = 10
Average Number of Hops : 5.277000
(base) pranayreddy@Pranays-MacBook-Air Team_30 % dotnet run 200 10
No of Nodes = 200 , No of Requests = 10
Average Number of Hops : 6.220500
(base) pranayreddy@Pranays-MacBook-Air Team_30 % dotnet run 500 10
No of Nodes = 500 , No of Requests = 10
Average Number of Hops : 7.737600
(base) pranayreddy@Pranays-MacBook-Air Team_30 % dotnet run 700 10
No of Nodes = 700 , No of Requests = 10
Average Number of Hops : 9.316429
(base) pranayreddy@Pranays-MacBook-Air Team_30 % dotnet run 1000 10
No of Nodes = 1000 , No of Requests = 10
Average Number of Hops : 9.679600
(base) pranayreddy@Pranays-MacBook-Air Team_30 % dotnet run 2000 10
No of Nodes = 2000 , No of Requests = 10
Average Number of Hops : 10.759550
(base) pranayreddy@Pranays-MacBook-Air Team_30 % dotnet run 5000 10
No of Nodes = 5000 , No of Requests = 10
Average Number of Hops : 11.839522
(base) pranayreddy@Pranays-MacBook-Air Team_30 % dotnet run 10000 10
No of Nodes = 10000 ,No of Requests = 10
Average Number of Hops : 13.473592
(base) pranayreddy@Pranays-MacBook-Air Team_30 % dotnet run 15000 10
No of Nodes = 15000 ,No of Requests = 10
Average Number of Hops : 15.354943
```

4.1 Tabular Results

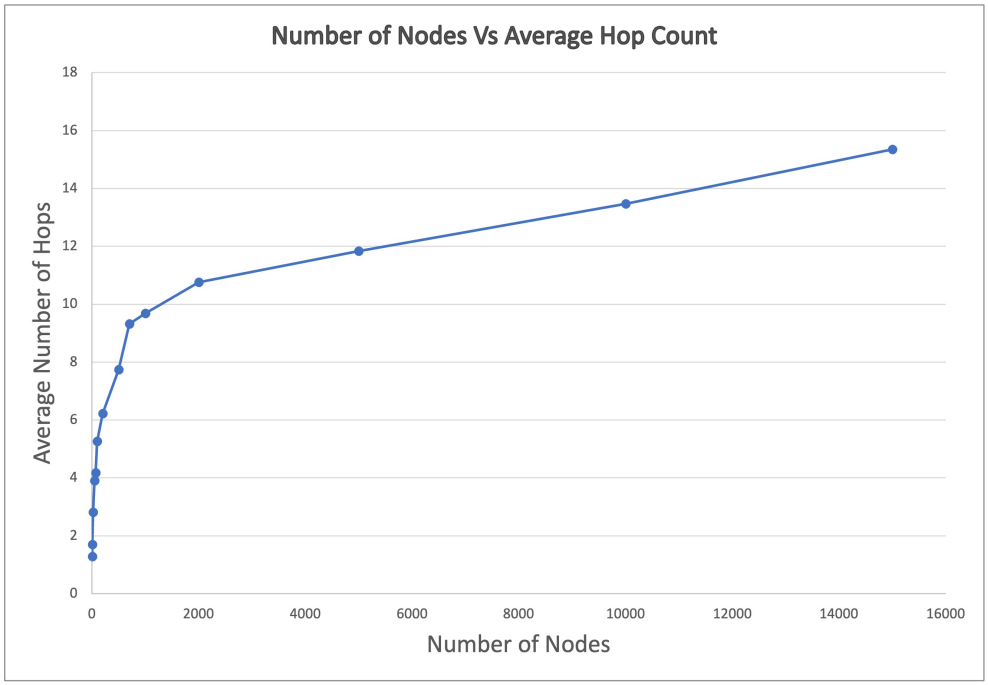
Number of Nodes	Average Hop Count
5	1.28
10	1.70
20	2.82
50	3.90
70	4.18
100	5.27
200	6.22
500	7.74
700	9.32
1000	9.68
2000	10.76
5000	11.84
10000	13.47
15000	15.35

Table 1: Number of Nodes vs Average Hop Count

The table represents a relationship between the number of nodes in a Chord ring network and the corresponding average hop count observed during the simulation of the Chord protocol. The average hop count likely indicates the average number of network hops required to locate a specific key within the Chord ring.

4.2 Graphical Results

Below is the graphical representation of the comparison between Average number of Hops and Number of nodes with number of requests being constant for all the records.



4.3 Assumptions

Below are some of the assumptions that our Chord protocol implementation makes:

- Deterministic Node Creation: The code assumes a Chord ring structure where each node has a unique identifier (nodeID) within a certain range (chordRingRange), typically determined by the number of bits (m) used for identifiers.
- Predefined Network Size: The code assumes a network size of 2^{20} suggested by the requirement.
- Initial Finger Table: The code initializes empty finger tables for each node as they join the network.
- Finger Table Maintenance: The code maintains the finger table to store references to other nodes that can potentially store keys closest to a given key in the Chord ring. It periodically updates and stabilizes the finger table to maintain an accurate representation of the network.
- Fixed Stabilization Interval: The stabilization process is executed continuously with a fixed interval.
- Stabilization Timing: The code does not take network latency into account, assuming that stabilization processes and request handling happen instantly.
- Key Lookup: To perform key lookups, the code utilizes a scalable key lookup algorithm based on finger tables. It initiates a key lookup request from a node to find the node responsible for a specific key in the Chord ring. It determines the number of hops required to locate the key.

4.4 Largest Network:

The largest network we managed to deal is 15000 number of nodes with number of requests for each node being 10. The average number of hops for this network is 15.35 .

5 Conclusion

In our collaborative project, all team members had an equal and important role in the task design and assignment. The project provided us with a comprehensive understanding of the Chord protocol's scalability and reliability. Implementing this protocol unveiled practical complexities, especially in managing dynamic peer-to-peer systems, which required us to address challenges such as maintaining finger tables, ensuring network stability, and accommodating swift additions of nodes. This hands-on experience shed light on the real-world significance of the Chord protocol in distributed systems, emphasizing its role in load balancing and data availability.

Furthermore, our project expanded our knowledge in several key areas:

- We gained insights into consistent hashing, a fundamental technique used across various distributed systems to streamline hashing efficiency.
- Our understanding of scalable key location significantly reduced the complexities associated with lookups, aligning with the Chord protocol's efficiency.

- We acknowledged the Chord protocol's vital role in addressing the intricate challenge of decentralized data item retrieval in distributed peer-to-peer applications. This protocol offers an efficient solution: given a key, it promptly identifies the node responsible for storing the key's value. In a stable network with N nodes, each node maintains routing information for only O(log N) other nodes, resulting in efficient O(log N) message-based lookups to other nodes.