Distributed Operating System Project 2 Report

PA2_Team 14

Anirudh Atrey, Mudit Arya, Prateek Abbi, Syed Faizan Ali

(59919994, 32141614, 89387132, 26592828)

(aniruddh.atrey@ufl.edu, arya.mudit@ufl.edu, prateekabbi@ufl.edu, syed.ali1@ufl.edu)

How to Run the Program

To run the program, you will need the following frameworks in your system:

- .NET SDK: To run the project, your system must have .NET SDK installed.
- AKKA.NET: Since our project is using AKKA framework and each node is having its own actors, you need to act AKKA.NET framework in your system.

With these SDKs and framework mentioned above, you need only one command to run the code: "**dotnet run <nodes> <requests>".**

Replace <nodes> and <requests> with actual number of nodes and requests respectively.

What is working

The Project is supposed to simulate the Chord Protocol with Actor model i.e., each node in the chord network should have an actor. For this we have used various functions (as described in the given paper). Below mentioned is the description of each function:

- Start_Algo: It is the first message of the chord network from the AKKA model i.e., initially Project starts with this message.
- Create: This function creates the entry in the finger entry table of the Chord protocol.
- Notify: It lets all other node know that the current node should be the predecessor of the receiving node. On receiving the message, it updates its predecessor to the otherID which is passed as an parameter.
- Stabilize: On receiving this as a message following things happen:
  - Checking Successor: By sending out a Requesting_Predecessor message, the present node asks its known successor to identify its immediate predecessor. This is to see whether there is a more deserving candidate to take over as leader.
  - Updating successor: Upon receiving a Predecessor_Response, the querying node will update its successor information if the respondent is a different node that is located closer to the node than its current successor.
  - Notify new successor: Post successor update, the node notifies the newfound successor with a Notify message, expressing its belief to be the predecessor.
  - Periodic Invocation: The ScheduleTellRepeatedly function call sets the stabilising activities to repeat at intervals specified by Stabilizing_Cycle_Time.
- Find_Successor: Each node in a distributed hash table (DHT) built on chords preserves information on the node that comes after it in the chain, known as its successor, and may also include information about other nodes that are spaced out around the ring and stored in its "finger table." An important component of this structure is the Find_Successor function, which helps determine which node in the network will be the immediate successor for a given identifier. Find_Successor is performing the following things:
  - A reference to the actor who started the message, known as IActorRef, and anotherID are sent along with a Find_Successor message. The active node will check to see if this message is the predecessor for the otherID after receiving it. The otherID, the node's ID, and the ID of its successor are compared to arrive at this conclusion.

- The current node recognises that its successor is the right node for the otherID if the otherID is between the current node's ID and that of its successor, or if the current node's ID is greater than that of its successor (this can occur in a ring-shaped arrangement where the IDs have circled around).
- As a result, it returns a Found_Successor message with the ID and a reference to its successor to the actor making the request (ref).
- The node will transmit the Find_Successor message to its own successor if it is not the correct match for anotherID. The same assessment is then carried out by the successor. Iteratively, this method keeps going around the ring until the right node is found.

- Found_Successor: One essential part of the Chord protocol is the Found_Successor message. This crucial message guarantees that every node correctly finds its successor and preserves the integrity of the Chord ring. This procedure is necessary to enable effective key lookups throughout the distributed system.
  - Message Purpose: The Found_Successor message is utilized to signify the discovery of a successor for a particular node ID. In the Chord protocol, each node maintains knowledge of its immediate successor within the ring topology. This message carries the information pertaining to the identified successor, including both the node ID and its associated actor reference (IActorRef).
  - Handling Logic: When a node receives a Found_Successor message, it adjusts its successor details utilizing the node ID and actor reference provided. This action is essential for maintaining the structural coherence of the ring as nodes enter and exit. Subsequent to modifying its successor, the node amends its finger table—a critical routing table utilized by all nodes in the Chord ring to expedite the identification of a specific key's holder—with the new successor information.
  - Scheduling Stabilize and Fix_Fingers: The node sets up periodic tasks for Fix_Fingers and Stabilize upon refreshing its successor details. Fix_Fingers is responsible for maintaining accurate finger table entries, and Stabilize regularly checks the validity of the node's immediate successor. These tasks ensure rapid key discovery in the distributed network and keep the node's data up to date.
  - Notify the Successor: Upon receiving a Found_Successor message, the node completes the process by dispatching a Notify message to the newly identified successor. In this protocol phase, a node communicates to its successor, suggesting itself as the potential predecessor. This action allows the successor to reassess and potentially refresh its predecessor information as required.
- Requesting_Predecessor: On receiving this function as a message, the actor will respond the sender with a “Predecessor_Response” which includes the ID of the predecessor of the current node and the actor reference of the predecessor. Maintaining the Chord ring's structure necessitates awareness of each node's predecessor. This is crucial for handling the inclusion and removal of nodes, revising finger tables, and ensuring the network's equilibrium. Within the Chord ring's architecture, each node is aware of its direct successor and predecessor. This knowledge is essential for overseeing resource distribution and facilitating decentralized searches across the ring's network configuration.
- Predecessor_Response: It is used for the stabilization process i.e., used tor maintaining the consistent distributed hash table. It is doing the following things:
  - When a node receives a Predecessor_Response message, it undertakes an assessment of its own records. If the predecessor ID provided in the message differs from the node's own identifier, the node updates its records. Specifically, it will adjust its successor and refSuccessor to align with the newly identified predecessor. This is a crucial process to ensure that the Chord ring's successor links are correctly established and maintained
  - After updating its successor information, the node communicates back to what was its predecessor's successor—now its own successor—by sending a Notify message. This message serves to inform that node that it should consider the current node as its potential new predecessor if necessary.
  - This process is essential for maintaining the integrity of the Chord network's topology, particularly as nodes join or leave the network.
- Key_Lookup: As the name is suggesting, this function is used to initiate and search the particular key withing the chord network. When this message is received by an actor, following things are happening:
  - Routing Logic:
    - ✓ It checks if the key should logically reside with the node's immediate successor within the structure of the Chord ring.

- ✓ If the successor node has an identifier that sequentially follows the key within the ring's topology, the lookup is considered successful. In this case, a message is dispatched to the printing actor (refofPrinter) noting the hop count with a Found_Key(hc) message. This indicates that either the node or its direct successor has responsibility for the key.
- ✓ In instances where the key does not fall within the immediate successor's range, the node consults its finger table. This table is utilized to determine the closest preceding node to the key within the node's knowledge. Subsequently, it passes on the Key_Lookup message to that node, incrementing the hop count in the process.
- ✓ This process is iterative, either until the message has circulated across the entire ring or until the key is found, whichever comes first.

➢ Termination Condition:
- ✓ The printer actor takes action upon receiving the Found_Key(hc) message, indicating the successful location of the key. This message enables the printer actor to accumulate the total hop counts. Once all lookup operations are complete, it calculates statistical data, including the average number of hops.

➢ Role in Chord Protocol:
- ✓ In the distributed hash table (DHT) model of Chord, both nodes and data identifiers (keys) occupy unique positions in a circular identifier space that resembles a ring. The responsibility for managing the keys falls on the individual nodes, specifically for those keys that are situated between the node itself and its predecessor within the space.
- ✓ To achieve efficient routing within this network—a process that scales logarithmically with the number of nodes, symbolized as O(log N) for a network comprising N nodes—each node is equipped with a finger table. This table contains information about other nodes in the system, facilitating expedited lookups.

➢ Finger table optimization:
- ✓ Instead of progressing sequentially from one node to another, the finger table allows a node to leap a specific interval around the circle with each move. This significantly boosts the speed of the search operation, as each jump has the potential to halve the remaining distance to the target key, leading to a lookup duration that is logarithmic in relation to the number of nodes.

➢ Potential for parallel lookups:
- ✓ Although not explicitly mentioned in the Key_Lookup functionality, the Chord protocol supports simultaneous searches through the finger table, which could potentially shorten lookup times even more.

- Fix_Fingers: This function in responsible for periodically maintaining the finger table of each node in the network. The purpose of the Fix_Fingers routine is to upkeep the routing information within the Chord network, enabling efficient search operations. In Chord's distributed hash table design, one of the key features is a finger table, structured to ensure that each entry points progressively farther around the network ring. This design ensures that, on average, a search can be performed in O(log N) steps within a network comprising N nodes.
- Find_Ith_Successor: This message (or function) initiates a request to find the ith entry for the finger table of the node.
  - ➢ Message Contents: The message holds three pieces of data: the index 'i', which indicates a specific position within the finger table; the 'key', which is the target identifier for which the successor is being sought; and the 'ref', which is the reference to the node that initiated this lookup process.
  - ➢ Determining the Immediate Successor: When the current node's successor has a smaller identifier than the node itself, and the given key is either greater than the node's identifier or smaller than the node's successor, the function concludes that the successor for the key must be the node's own immediate successor. Consequently, it responds with a Found_Finger_Entry message.
  - ➢ Handling Keys within a Specific Range: If the key falls within the range that is greater than the node's identifier and less than or equal to the node's successor, the function again responds with a Found_Finger_Entry message, suggesting that the successor to this node is the direct successor for the key.
  - ➢ Beyond Simple Cases: If neither of the above scenarios apply, the function embarks on a more comprehensive search. It traverses the finger table in reverse, from the last entry towards the first. This loop continues until a suitable finger entry is located that is closest preceding to the key in question. When such an entry is found, the message is forwarded to the corresponding node associated with that finger entry. If the traversal reaches the end without finding a suitable entry, the Find_Ith_Successor message is passed to the current node's immediate successor for further handling.
  - ➢ Addressing Circular Network Topology: Recognizing the circular structure of the Chord network, the function takes into account scenarios where the key's numerical value is lower than the node's identifier. To

accommodate the ring's wrap-around nature, the size of the entire hash space is added to the key before making comparisons, ensuring the algorithm respects the circular continuity of the network space.

- Found_Finger_Entry: When this function is called, or when node receives this as a message, it creates the new entry in the finger table with the otherID and ref passed as an argument to the function. After doing so it updates the i-th position in its finger table with the new entry.
- Start_Lookups: When node receives this as a message or when this function is called, the algorithm simulates the scenario where each node in the network initiates multiple queries to locate random keys within the distributed hash table. This could be part of a test to measure the efficiency and speed of the Chord DHT system, particularly looking at metrics like the average number of hops or the time it takes to respond. The total number of hops needed to find each key is communicated back through Found_Key messages, and these are aggregated to calculate the average number of hops across all searches.
- Found_Key: The Akka.NET actor model framework is leveraged to create a simulation of the Chord system, and within this simulation, the Found_Key message plays a significant role. This message type is associated with the successful location of a key within the Chord distributed hash table (DHT) and is accompanied by a hop count (hc), which represents the number of steps or nodes traversed to find the key.
  The printHC function is responsible for processing the Found_Key message. It orchestrates an actor that monitors the accumulation of Found_Key messages and the summation of their respective hop counts. Once the actor ascertains that the count of Found_Key messages matches the product of the total requests and the total nodes, it proceeds to calculate the average number of hops per key lookup. Upon determination, this average is then displayed, providing insight into the efficiency of the key location process within the simulation.

Screenshots of the output

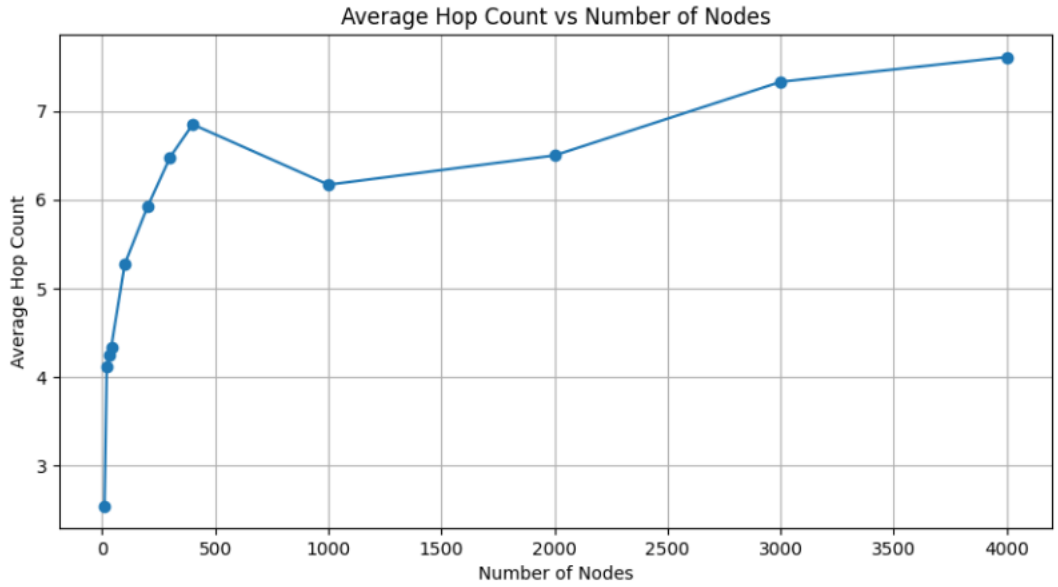
```
PS C:\Users\aryam\OneDrive\Desktop\UF\SEM 2 FALL 2023\DOSP\Chord\Chord> dotnet run 10 10
Average HOPCOUNT = 2.55
PS C:\Users\aryam\OneDrive\Desktop\UF\SEM 2 FALL 2023\DOSP\Chord\Chord>
PS C:\Users\aryam\OneDrive\Desktop\UF\SEM 2 FALL 2023\DOSP\Chord\Chord> dotnet run 20 10
Average HOPCOUNT = 4.12
PS C:\Users\aryam\OneDrive\Desktop\UF\SEM 2 FALL 2023\DOSP\Chord\Chord> dotnet run 30 10
Average HOPCOUNT = 4.25
PS C:\Users\aryam\OneDrive\Desktop\UF\SEM 2 FALL 2023\DOSP\Chord\Chord> dotnet run 40 10
Average HOPCOUNT = 4.34
PS C:\Users\aryam\OneDrive\Desktop\UF\SEM 2 FALL 2023\DOSP\Chord\Chord> dotnet run 200 10
Average HOPCOUNT = 5.92
PS C:\Users\aryam\OneDrive\Desktop\UF\SEM 2 FALL 2023\DOSP\Chord\Chord> dotnet run 100 10
Average HOPCOUNT = 5.28
PS C:\Users\aryam\OneDrive\Desktop\UF\SEM 2 FALL 2023\DOSP\Chord\Chord> dotnet run 300 10
Average HOPCOUNT = 6.48
PS C:\Users\aryam\OneDrive\Desktop\UF\SEM 2 FALL 2023\DOSP\Chord\Chord> dotnet run 400 10
Average HOPCOUNT = 6.85
PS C:\Users\aryam\OneDrive\Desktop\UF\SEM 2 FALL 2023\DOSP\Chord\Chord> dotnet run 4000 10
Average HOPCOUNT = 7.61
PS C:\Users\aryam\OneDrive\Desktop\UF\SEM 2 FALL 2023\DOSP\Chord\Chord>
```

Table of Avg. hop Count results with varying number of nodes

Number of Nodes	Number of Requests	Average Hop Count
10	10	2.55
20	10	4.12
30	10	4.25
40	10	4.34
100	10	5.28
200	10	5.92
300	10	6.48
400	10	6.85
1000	10	6.17

2000	10	6.50
3000	10	7.33
4000	10	7.61

Graph of "Number of nodes" vs "Avg. Hop Count"



Assumptions about the protocol

While developing the project, and after reading the given research paper we assumed the following things about the Protocol:

- Consistent Hashing
  As mentioned in the paper that the protocol uses consistent hashing for the optimal lookups. We have implemented the similar function that maps the nodes to a chord ring of size 2^m where m is always constant (20 – as mentioned in the PDF).
- Finger Tables
  In the chord protocol, each node has its own entry known as finger table which tells us the next node for hopping. This allows for the looking up for any node in O(log N) time complexity.
- Node Join
  Nodes can join the ring even after successfully creating the ring. When nodes join, they initialize the successor and predecessor and then based on it, successor and predecessor of the joining node update the entry in the finger tables.
- Stabilization
  After every few seconds (100ms in our case), the ring stabilizes itself to correct and update the entries in the finger tables and to update the successor and predecessor.
- Lookup Requests
  Using the finger tables, algorithm can lookup for the desired keys, provided that the algorithm gets the correct successor/predecessor information to route the request through the network, while its looking for the desired key, it counts the number of hops it takes to resolve the lookup.
- Asynchronous and Event Driven
  Since we are assuming that, algorithm adds multiple keys concurrently and asynchronously, we are using AKKA framework of F# to simulate that, for handling concurrency and message passing between nodes.
- Fault Tolerance
  Since, ring is stabilizing itself after every few seconds, we are assuming that the ring has the ability to deal with node failures.
- Random Node Generation

Within the limit, it is generating the nodes randomly and not sequentially.

Largest Network

We were able to make largest network with 4000 nodes and 10 requests. We could have gone more but it was taking too much of time. So, we decided to stop at 4000 nodes only.