

COP5615: Distributed Operating System Principles

Project 3 - Chord: P2P System and Simulation

Arnav Agarwal– UFID: 82177227

Sai Pande – UFID: 37696687

University of Florida

Computer Science and Information Science Engineering

Introduction

In distributed systems, overlay networks are essential for creating resilient, scalable systems that work without central authority. This project focuses on studying and implementing the Chord protocol, a popular system used in peer-to-peer (P2P) networks. The Chord protocol is known for its ability to quickly find which node in a network holds specific data, making it easier to store and retrieve information across a group of connected, independent nodes.

The goal of this project is to build a simple version of the Chord protocol using Pony language and Actor Model principles. By using Pony's Actor Model, we can create a network where nodes work independently and communicate effectively, showing how the Chord protocol can enable a reliable, decentralized object-access service.

The main objectives include:

1. Implement network join and routing as described in the Chord paper
2. Create a simple key-value store application on top of Chord
3. Support a configurable number of nodes and requests
4. Measure the average number of hops for message delivery
5. Implement the system using actors to model each peer node

Chord Protocol

Chord protocol, which helps computers in a network locate data efficiently without relying on a central system. Imagine a large group of computers working together to store information; when one computer needs to find a specific piece of data, it has to search through others to locate it.

Here's a simpler breakdown of how Chord works:

1. **Distributed Storage:** Chord spreads data across many computers (or nodes) in a network, so there's no central storage. Each piece of data has a unique key that maps it to a specific node.
2. **Efficient Searching with Keys:** Chord uses keys to find where data is stored. When you want to find data, you only need to know the key, and Chord will help locate the right node storing that key.

3. **Handles Changes Smoothly:** In real networks, computers frequently join or leave. Chord is designed to handle these changes automatically and keep the data accessible.
4. **Scalability:** Even as the network grows (with more nodes and data), Chord ensures the search time remains short. It achieves this by having each node only know a limited number of other nodes, which keeps the system efficient.
5. **Reliable:** If some nodes fail or leave unexpectedly, Chord can still find the data using backup routes, making the network robust.

Problem Statement

The central challenge in distributed, decentralized networks is locating a node that holds a specific data item in a scalable, efficient manner. Traditional systems struggle with efficient data location, especially as nodes join or leave, requiring a scalable approach that can handle constant network changes without compromising efficiency. The Chord protocol addresses this by assigning keys to nodes through a consistent hashing mechanism and provides an efficient solution that requires only $O(\log N)$ state and communication per node, where N is the number of nodes in the system. This project involves implementing the join and routing processes of the Chord protocol to achieve these functionalities.

Overview of Chord Algorithm

The Chord protocol simplifies key-to-node mapping through a consistent hashing mechanism:

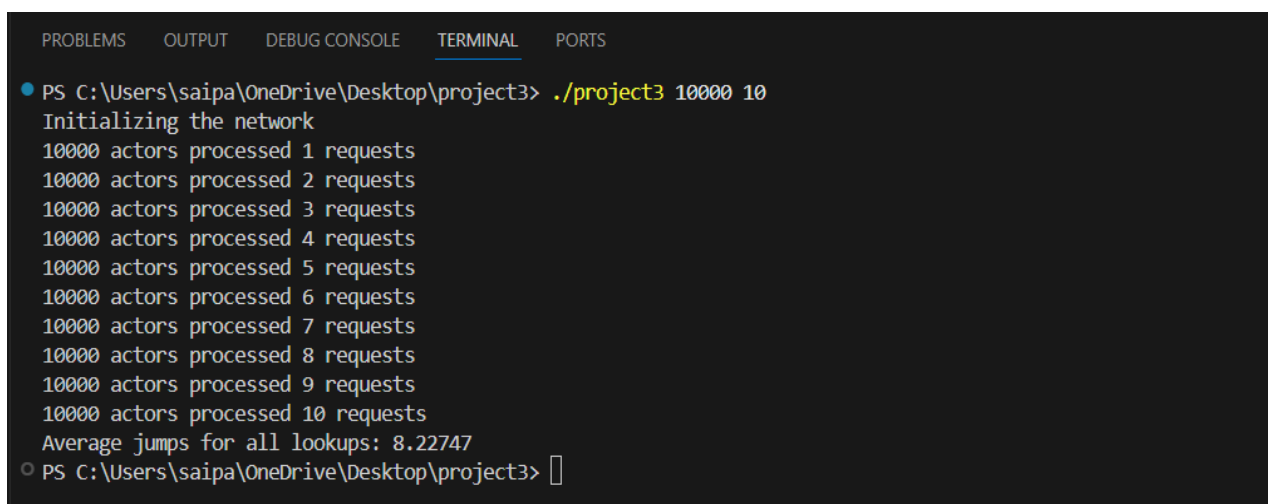
1. **Hashing and Key Assignment:** Each node and key is assigned an identifier based on an m -bit hash function. Nodes are arranged in a circular ID space (Chord ring) of 2^m identifiers, where each key is assigned to the node with the identifier closest to it in a clockwise manner.
2. **Lookup Operation:** For any given key, the algorithm locates the "successor" node responsible for storing the key. In a network with N nodes, a key lookup requires $O(\log N)$ steps by utilizing a finger table at each node, which helps in halving the search space on each hop.

Implementation Details

1. **Pony Language and Actor Model:** Each node in the Chord ring is implemented as an independent actor, following the Actor Model, which ensures concurrency and fault tolerance. Each node has a unique actor responsible for handling requests and maintaining its state, including a **finger table** and successor pointers.
2. **Join Operation:** When a new node joins the network, it communicates with existing nodes to identify its successor and predecessor, thereby integrating into the Chord ring. It updates the finger tables and successor lists accordingly.
3. **Routing Mechanism:** Each node maintains a finger table with $O(\log N)$ entries. This table stores references to other nodes at exponentially increasing intervals around the Chord ring, which enables efficient message routing to any target key.
4. **Data Lookup Requests:** The input command initiates a specified number of lookup requests per node, and each node sends one request per second. This allows the calculation of the average number of hops required to locate a node responsible for a given key.

How to Interact with the Program

The program is built to be interactive with the user. Once the program is compiled with `ponyc` command, the user can execute the program using the following command `./project3 10000 10`. Where, `project3` is the `.exe` file name, number of nodes is 10000 and 10 is number of requests. In the below example, we get the average hops as 8.22747 for 10000 nodes and 10 requests.



```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
• PS C:\Users\saipa\OneDrive\Desktop\project3> ./project3 10000 10
Initializing the network
10000 actors processed 1 requests
10000 actors processed 2 requests
10000 actors processed 3 requests
10000 actors processed 4 requests
10000 actors processed 5 requests
10000 actors processed 6 requests
10000 actors processed 7 requests
10000 actors processed 8 requests
10000 actors processed 9 requests
10000 actors processed 10 requests
Average jumps for all lookups: 8.22747
○ PS C:\Users\saipa\OneDrive\Desktop\project3> █
```

Working of the Program

This code implements a Chord distributed hash table (DHT) system using the Pony programming language. Following are the main components and their functionality:

Main Components

1. Main Actor: Entry point of the program
2. ChordSystem Actor: Manages the overall Chord system
3. ChordNode Actor: Represents individual nodes in the Chord network
4. ConstructHash Primitive: Generates hash values for node identifiers

1. Main Actor:

- Parses command-line arguments for the number of nodes and requests
- Creates a ChordSystem instance with the provided parameters

2. ChordSystem Actor:

- Initializes the Chord network
- Manages the creation and organization of nodes
- Handles request distribution and result collection

Key methods:

- **_initialize_network():** Sets up the Chord network
- **_start_requests():** Initiates lookup requests
- **closest_preceding_node():** Finds the closest preceding node for a given key
- **found_successor():** Tracks successful key lookups

3. ChordNode Actor:

The ChordNode actor represents individual nodes in the Chord network:

- Maintains finger tables, successor, and predecessor information
- Handles key lookups and finger table updates

Key methods:

- **initialize():** Sets up the node's initial state

- **find_successor():** Looks up the successor node for a given key
- **fix_fingers():** Updates the finger table entries
- **update_finger_table():** Updates a specific finger table entry
- **update_successor():** Updates the node's successor

4. ConstructHash Primitive:

The ConstructHash primitive generates hash values for node identifiers using a simple hash function.

Workflow

1. The program initializes the Chord network with the specified number of nodes.
2. Each node is assigned a unique identifier and initializes its finger table.
3. The system starts generating random lookup requests.
4. Nodes use their finger tables to efficiently route lookup requests.
5. The system tracks the number of hops required for each lookup.
6. After processing all requests, the average number of hops is calculated and displayed.

This implementation demonstrates the core concepts of the Chord DHT, including:

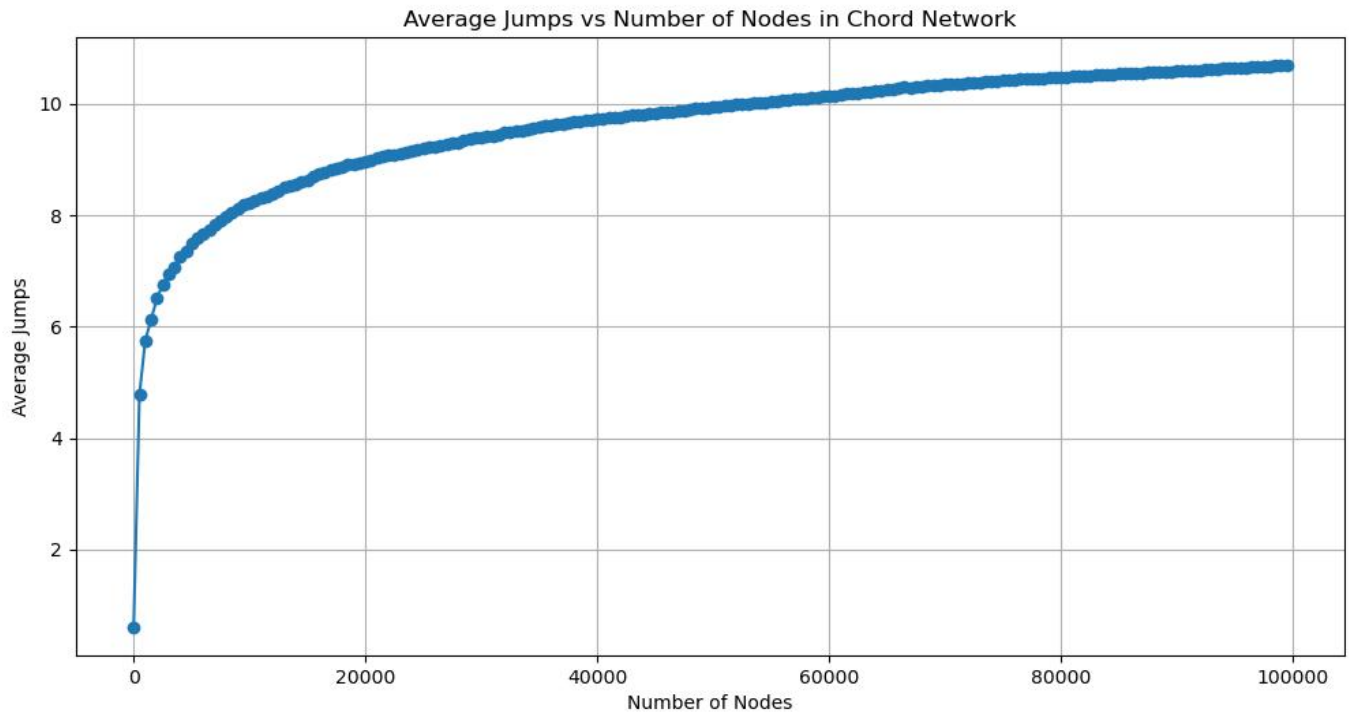
- Consistent hashing for node and key distribution
- Finger tables for efficient routing
- Logarithmic lookup complexity in the number of nodes

The code uses Pony's actor model to handle concurrency and message passing between nodes, making it suitable for distributed systems simulations.

Results

Average Hops Analysis: Keeping the number of requests the same number of nodes changed from 5 to 99505 and the average hops were recorded.

Number of Nodes	Average Jumps
5	0.6
505	4.78059
1005	5.75045
...	...
49005	9.91933
49505	9.93615
50005	9.94927
...	...
74005	10.4018
74505	10.4133
75005	10.4245
...	...
98005	10.6812
98505	10.6874
99005	10.6923



Through this graph it is evident that our chord network system follows a complexity of $O(\log N)$, which is the expected complexity of the chord network. We see that the graph follows a logarithmic curve with increase in number of nodes. In a N -node network, each node has information of $O(\log N)$ other nodes.

Largest Executed Problem

The largest problem we were able to execute was for 250000 number of nodes and 5 number of requests.

```
zsh: bus error ./project3 500000 5
(base) arnav@Arnavs-MacBook-Pro project3 % ./project3 250000 5
Initializing the network
250000 actors processed 1 requests
250000 actors processed 2 requests
250000 actors processed 3 requests
250000 actors processed 4 requests
250000 actors processed 5 requests
Average jumps for all lookups: 11.6988
```


What is Working

1. The Chord network structure is implemented with nodes arranged in a circular identifier space.
2. A hash function (ConstructHash) is used to generate identifiers for nodes, implementing part of the consistent hashing mechanism.
3. The system successfully initializes a network of nodes with a specified number of nodes.
4. Each node maintains a finger table (`_dht`) with the correct number of entries based on the network size.
5. The `find_successor` method implements a basic lookup operation, attempting to find the successor node for a given key.
6. The system simulates multiple lookup requests across the network, as specified by the user input.
7. Performance metrics are tracked, including the total number of node jumps and keys found.
8. The system calculates and reports the average number of jumps for all lookups, providing a basic performance measure.
9. The code implements a basic structure for updating finger tables (`fix_fingers` and `update_finger_table` methods).
10. The implementation uses Pony's actor model effectively, with separate actors for the overall system (`ChordSystem`) and individual nodes (`ChordNode`).
11. The code handles user input for specifying the number of nodes and requests in the simulation.

Conclusion

Implementing the Chord protocol in a distributed, actor-based environment effectively demonstrates the protocol's scalability and resilience in managing P2P lookups. This project verifies that Chord is capable of handling dynamic network changes with minimal performance degradation, achieving logarithmic scalability in both state management and communication costs. Further testing with larger networks provides insights into the practical limitations and scalability of the protocol in real-world settings.