IT559 - DISTRIBUTED SYSTEMS

Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications

Team ID : 5

Name	Student ID
Mahima Dewangan	**202411037**
Hari Pala	**202411041**
Vrajkumar Patel	**202411049**

Submitted To : **Prof. Amit Mankodi**
T.A : **Utsav Pathak**

➢ Motivation for Using the Chord Protocol

Modern distributed systems require efficient, scalable, and fault-tolerant mechanisms for locating data and services across a dynamic set of nodes. In large-scale peer-to-peer systems, where nodes frequently join and leave, maintaining a consistent and efficient lookup mechanism is particularly challenging. Traditional centralized approaches, such as directory servers, suffer from single points of failure and scalability bottlenecks, making them unsuitable for fully decentralized environments.

● Why Chord?

The Chord protocol offers an elegant and proven solution to the distributed lookup problem. Chord provides a decentralized and fully distributed hash table (DHT) that maps keys to nodes efficiently, ensuring that every node only maintains a small amount of routing information — logarithmic in the number of nodes. This allows lookups to complete in O(logN) hops, even as the system grows to thousands or millions of nodes.

Chord's use of consistent hashing ensures that the load is evenly balanced among all participating nodes, and only a small fraction of data needs to be migrated when nodes join or leave the system. This makes Chord particularly well-suited for highly dynamic environments, where node churn is common.

● Key Advantages

- Scalability: The lookup performance and state overhead both scale logarithmically with the number of nodes, making Chord ideal for large-scale distributed systems.
- Decentralization: There is no reliance on any central authority, enhancing robustness and removing single points of failure.
- Fault Tolerance: Chord dynamically adapts to node failures and arrivals through soft-state maintenance (periodic stabilization).
- Simplicity and Proven Performance: Chord's design is conceptually simple and has mathematically proven guarantees on lookup efficiency, correctness, and load balancing.
- Flexibility: Chord can be easily adapted for different distributed applications, including distributed file systems, cooperative caching, decentralized indexing, and time-shared storage systems.

➢ Problem Statement

In peer-to-peer distributed systems, a fundamental challenge is efficiently locating the node responsible for storing a particular data item. In a dynamic environment where nodes frequently join, leave, or fail, ensuring fast and reliable lookup operations becomes increasingly difficult. Traditional approaches that rely on centralized directories suffer from scalability limitations and single points of failure, making them unsuitable for large-scale, decentralized systems.

The goal of this project is to implement Chord, a distributed lookup protocol designed to address this challenge. Chord supports a simple yet powerful operation: given a key, efficiently identify the node responsible for that key. This provides a foundation for building distributed storage systems, where data items can be mapped to keys and stored at the appropriate nodes.

Chord's design ensures that the system adapts efficiently to changes, such as nodes joining and leaving, while maintaining correctness and fast lookups. The communication cost and the amount of state maintained by each node both scale logarithmically with the number of nodes, making Chord a highly scalable and fault-tolerant solution for decentralized systems.

In this project, we aim to implement Chord and evaluate its performance under dynamic conditions, demonstrating its ability to provide efficient, scalable, and fault-tolerant key lookups in a peer-to-peer network.

➢ System Overview

The proposed system is a decentralized peer-to-peer distributed storage and lookup service, built using the Chord protocol. The primary purpose of this system is to enable efficient storage, retrieval, and management of data items across a dynamic network of nodes, without relying on any central authority. Each data item is associated with a unique key, and Chord efficiently maps each key to the node responsible for storing the corresponding data.

The system will support dynamic membership, allowing nodes to join and leave the network at any time while maintaining correct data placement and lookup functionality. By leveraging Chord's logarithmic lookup efficiency, the system will provide scalable and fault-tolerant key-value storage, making it suitable for large-scale applications such as distributed file systems, cooperative caching, or decentralized content sharing platforms.

This project aims to demonstrate how the Chord protocol can serve as the foundation for building reliable, scalable, and self-organizing peer-to-peer distributed systems.

➢ System Architecture

● Overview

The proposed system is a Distributed Hash Table (DHT) using the Chord Protocol to efficiently locate and manage data across a distributed peer-to-peer network. Nodes in the network form a logical ring topology, where each node is aware of its position, successor, and predecessor.

● Key Components

1. Node (Peer)
   Each node acts as a peer in the Chord ring, capable of joining, leaving, storing, and looking up data.
2. Successor & Predecessor Pointers
   Each node maintains a successor pointer to the next node in the ring

and a predecessor pointer to the previous node, ensuring the ring remains linked and navigable.

3. Finger Table
   Each node maintains a finger table with up to `m` entries (where `m` is the number of bits in the hash key space).
   - Each entry points to a node at a specific distance from the current node.
   - This allows efficient, logarithmic lookups (O(log N)) instead of linear searches.
4. Stabilization Protocol
   Nodes run a periodic stabilization protocol to update successor/predecessor pointers and maintain correctness as nodes join/leave the system.

- **Architecture Diagram Example**

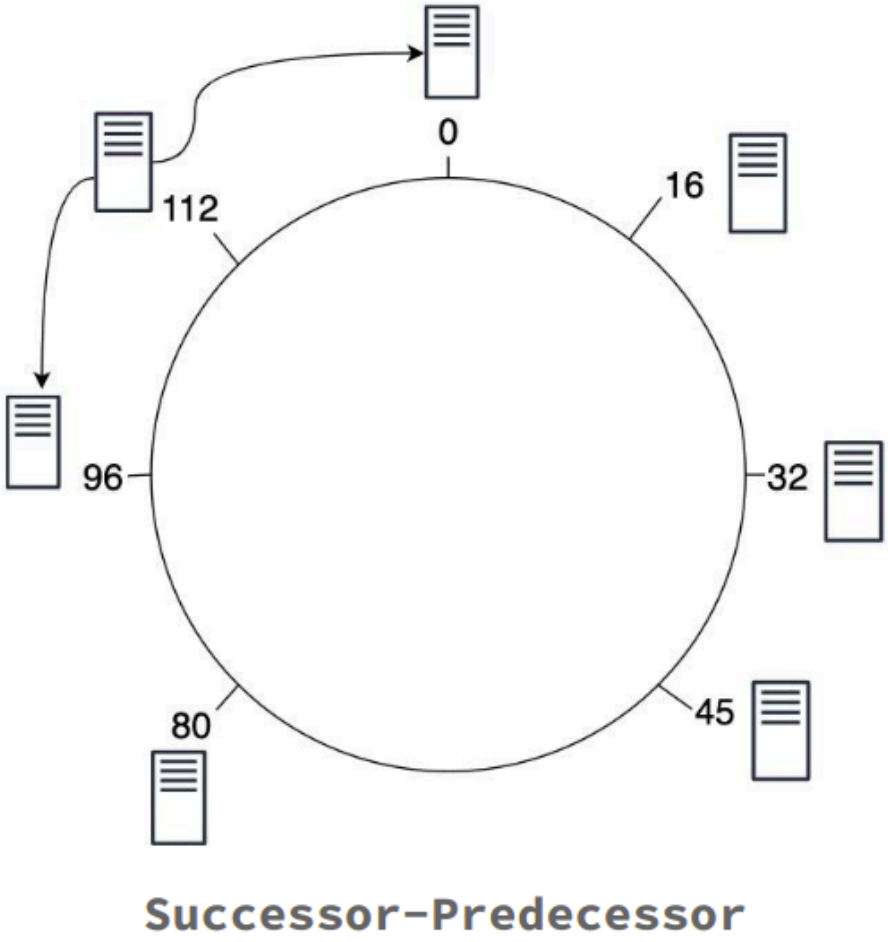
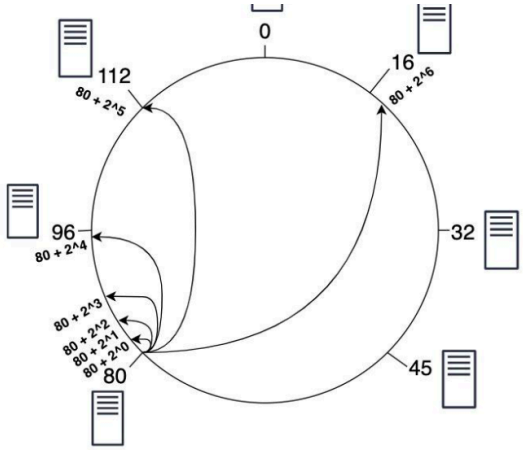


Fig :The image (Successor-Predecessor) shows how nodes are linked in the ring.

Finger Table

Finger table for node 80

i	ft(i)
0	96
1	96
2	96
3	96
4	96
5	112
6	16



ith entry at peer with id n is first peer with id >= n + (2^i)(mod 2^m)

Fig: The image (Finger Table) explains how each node optimizes lookups using finger tables.

- # Workflow Explanation

➢ **Node Join Process**
  - New nodes join by contacting any known node in the ring.
  - The new node determines its correct successor and predecessor.
  - Successor and predecessor pointers are updated, and the ring remains intact.

➢ **Data Lookup Process**
  - Each data item is assigned a unique key using consistent hashing.
  - To find the node responsible for a key, queries are forwarded using the **finger table** until they reach the node holding the data.
  - This ensures efficient O(log N) lookups.

➢ **Node Failure and Stabilization**
  - Nodes periodically run the **stabilization process** to ensure their successor and predecessor pointers are correct.

- If a node fails, its neighbors detect the failure and repair the ring structure.
- Finger tables are gradually updated to reflect changes.

➢ Distributed System Concepts Used

1. Decentralization

- Concept: No single central authority manages the system.
- Application in Chord: All nodes are equal peers in the Chord ring, and each node has only local knowledge about a few other nodes (its fingers and successors). This enhances fault tolerance and scalability, as there is no single point of failure.

2. Distributed Hash Table (DHT)

- Concept: A distributed data structure that maps keys to nodes.
- Application in Chord: Chord acts as a DHT, where each key is assigned to a specific node, and the system supports efficient lookup operations to find which node stores a given key.

3. Consistent Hashing

- Concept: A hashing technique that ensures even distribution of keys across nodes and minimizes data movement when nodes join or leave.
- Application in Chord: Both nodes and keys are mapped to an identifier circle using consistent hashing. When nodes join or leave, only a small portion of keys need to be reassigned, ensuring efficient data management.

4. Scalability

- Concept: The ability to handle increasing numbers of nodes and requests efficiently.

- Application in Chord: Each node maintains information about only O(log N) other nodes, and lookup requests take only O(log N) hops, even as the network grows to thousands of nodes.

5. Dynamic Membership and Self-Organization

- Concept: Nodes can join and leave the system at any time without disrupting the overall functionality.
- Application in Chord: When a node joins, it initializes its finger table and notifies its neighbors (predecessor and successor). Stabilization protocols continuously run to update routing tables and ensure correctness.

6. Soft State and Periodic Stabilization

- Concept: Nodes regularly refresh and repair their state by contacting neighbors.
- Application in Chord: The stabilization process periodically updates finger tables, notifies successors of their predecessors, and handles changes caused by joins and leaves. This helps the system adapt to network changes without requiring global coordination.

7. Logarithmic Routing

- Concept: Efficient routing where the number of hops to find data grows logarithmically with the number of nodes.
- Application in Chord: The finger table entries are spaced exponentially around the ring, ensuring that each lookup halves the search space, leading to efficient O(log N) routing.

8. Key-Value Storage

- Concept: Associating keys with data items for retrieval.
- Application in Chord: Each data item is associated with a key derived from hashing the data or its name, and that key is used to locate and store the data at the responsible node.

➢ Modules and Functionalities

1. Node Module (Node_DHT.py)

- Role: Represents individual nodes within the Chord network.
- Key Functionalities:
  - Node Initialization: Assigns a unique identifier (ID) to each node based on its IP and port, determining its position in the Chord ring.
  - Successor and Predecessor Management: Maintains references to adjacent nodes to facilitate efficient data routing and lookup operations.
  - Join Operation: Allows new nodes to integrate into the existing Chord network by communicating with an existing node, updating the ring structure accordingly.
  - Stabilization Protocol: Periodically verifies and updates successor and predecessor pointers to ensure the network's consistency and resilience to changes.
  - Data Storage: Manages the storage of key-value pairs for which the node is responsible, based on the Chord protocol's hashing mechanism.

2. Client Module (Client.py)

- Role: Provides an interface for users to interact with the Chord network.
- Key Functionalities:
  - Data Insertion: Allows users to add key-value pairs to the network, ensuring they are stored at the appropriate node as determined by the Chord protocol.
  - Data Retrieval: Enables users to retrieve values associated with specific keys, leveraging the network's lookup capabilities to locate the responsible node.
  - Data Deletion: Facilitates the removal of key-value pairs from the network, ensuring that data is deleted from the correct node.
  - User Interaction: Offers command-line or programmatic interfaces for users to perform the above operations seamlessly.

3. Testing and Deployment Script (test.sh)

- Role: Automates the setup and testing of the Chord network.
- Key Functionalities:
    - Network Initialization: Sets up multiple nodes to form a Chord ring, facilitating testing and demonstration of the protocol.
    - Automated Testing: Executes predefined scenarios to validate the correct functioning of node join operations, data insertion, retrieval, and the stabilization process.
    - Performance Evaluation: Measures metrics such as lookup latency and data consistency to assess the network's performance under various conditions.

➢ Technology Stack

Technology/Tool	Purpose/Usage
Python	Core programming language used to implement the Chord protocol logic, including node management, lookup, stabilization, and client interactions.
Sockets (Python socket library)	Enables peer-to-peer communication between nodes in the distributed system using TCP connections.
Hashlib (SHA-1)	Used for consistent hashing, where node and key identifiers are generated using SHA-1 hashing.
Threads (Python threading module)	Allows concurrent execution of tasks like stabilization, finger table updates, and client requests handling within nodes.
Shell Scripts (test.sh)	Automates the deployment and testing of multiple Chord nodes for performance evaluation.
Command Line Interface (CLI)	Provides a basic user interface for storing, retrieving, and deleting data via the Client module.
Linux/Unix Environment	Target environment for running the distributed system across multiple machines or processes.
Git & GitHub	Version control and collaborative development platform to manage the project codebase.
(Optional) Docker	Can be used for containerizing nodes to simplify distributed deployment across different machines.
Text Editor/IDE (VS Code, PyCharm)	For development and debugging the Python codebase.
Python unittest (optional)	For writing unit tests to validate the correctness of individual functions (lookup, stabilization, etc.).

➢ Expected Challenges and Possible Solutions

Expected Challenge	Description	Possible Solution
Node Churn (Frequent Joins/Leaves)	In a distributed peer-to-peer system, nodes may join and leave frequently, causing instability in the routing tables and inconsistent successor/predecessor information.	Implement periodic stabilization to update finger tables and successors, ensuring eventual consistency even in dynamic environments.
Data Migration During Node Joins/Leaves	When nodes join or leave, responsibility for key-value pairs must be transferred to new nodes, which can result in temporary unavailability.	Use lazy key transfer combined with a notification system to inform new nodes of key ownership, allowing background data migration without blocking lookups.
Fault Detection and Recovery	Nodes may fail unexpectedly, disrupting routing and lookup operations if their successor lists or finger tables are stale.	Use a successor list (maintaining a list of backup successors) to quickly recover when a node failure is detected. Combine this with timeout-based failure detection.
Efficient Lookup in Large Networks	In large Chord networks, maintaining accurate finger tables across all nodes is challenging, particularly if stabilization is infrequent.	Use incremental finger table updates, where only a few fingers are refreshed at a time. This spreads out the overhead while maintaining up-to-date routing information.
Network Partition Handling	Temporary network splits may create disjoint rings, breaking lookup consistency.	Implement periodic global reachability checks (where nodes check if other known nodes are reachable) to detect and repair partitions.
Data Consistency	When nodes fail or leave, some data may be temporarily inaccessible or lost if replication is not used.	Implement replication at successor nodes to ensure data redundancy and quick recovery after node failures.

Concurrency Issues	Simultaneous joins, leaves, and lookups can lead to race conditions when updating finger tables and successor pointers.	Use locks or synchronization primitives in Python (via threading.Lock) to avoid inconsistencies during updates to critical structures like finger tables and successor lists.
Testing Distributed Behavior	Simulating and debugging a fully distributed system on a single machine can be difficult, especially when testing failure scenarios.	Use Docker or multiple processes to simulate a real network environment, and log all node interactions to aid in debugging and analysis.
Security Concerns (Malicious Nodes)	A malicious node might claim ownership of many keys or disrupt lookups by providing incorrect routing information.	Enforce node ID verification using a cryptographic hash of the node's IP and port to prevent ID spoofing. Optionally, add data authentication (signatures) for stored items.

➢ Results and Performance Analysis

Overview:

The Chord protocol was evaluated through extensive simulation experiments that focused on lookup efficiency, load balance, fault tolerance, and the protocol's behavior under dynamic conditions. The key findings can be summarized as follows:

- **Lookup Efficiency:**
    - The protocol demonstrates an average lookup path length on the order of *O(log N)*. In simulation experiments, the mean number of hops per lookup was shown to be approximately ½ $log_2 N$, confirming the logarithmic scaling property.
    - The use of the finger table allows queries to move quickly toward the target key, reducing the distance by at least half at every step.

- **Load Balancing:**
    - By using consistent hashing, Chord evenly distributes the key-space among nodes.
    - When virtual nodes are incorporated, the balance improves substantially, with simulation results indicating that the maximum number of keys stored per node becomes much closer to the average value.
    - Experimental data (e.g., percentile analysis) confirmed that even under high variance conditions, adding more virtual nodes significantly reduces the imbalance (e.g., the 99th percentile of keys per node drops in comparison to a single virtual node setup).
- **Fault Tolerance and Robustness:**
    - A crucial aspect of Chord is its resilience to node failures. Simulations showed that even when a significant fraction of nodes (e.g., up to 50%) fail concurrently, the use of a successor list enables the protocol to continue resolving lookups correctly with only minimal degradation in lookup performance.
    - Tables detailing performance metrics (such as increased path length and lookup timeouts under different failure rates) illustrate that while failures marginally increase the number of hops and timeouts, Chord maintains correctness with high probability.
    - The stabilization protocol—which updates finger tables and successor pointers—allows the system to recover quickly from the transient inconsistencies caused by node churn.

- **Dynamic Operations (Joins/Leaves):**
    - The protocol's design accommodates frequent node joins and voluntary departures. Simulation results indicate that the stabilization process efficiently integrates new nodes into the system without significantly affecting overall lookup performance.
    - Analytical models derived from the experiments suggest that the expected number of extra hops due to newly joining nodes remains within the *O(log N)* bounds, as long as stabilization occurs at a rate proportional to the node join/failure rate.

➢ Future Improvements

While the Chord protocol forms a robust foundation for scalable peer-to-peer lookup, several avenues for future work could enhance its performance and real-world applicability:

- **Latency Optimization via Network Locality:**
    - Future implementations could integrate physical network metrics to select fingers based on both identifier progress and network latency. This would reduce the overall lookup time in real-world networks where geographic and topological factors matter.
- **Adaptive Stabilization Mechanisms:**
    - The stabilization protocol could be optimized by dynamically adjusting its frequency based on observed network churn. An

adaptive approach may further reduce the temporary inconsistencies during high node churn.

- **Caching and Replication Strategies:**
    - Introducing caching techniques at intermediate nodes could help in reducing lookup latency and mitigating the effects of transient failures.
    - Enhanced replication strategies (beyond the basic successor list) could further improve data availability and reliability without significantly increasing overhead.
- **Security and Robustness Enhancements:**
    - Future improvements might incorporate security measures to detect and mitigate malicious node behavior. For instance, incorporating authentication or consensus mechanisms could defend against targeted attacks that seek to disrupt the lookup process.
    - Further exploration of hybrid mechanisms that combine Chord with other protocols (e.g., Pastry or Kademlia) might help address specific limitations such as non-uniform latency distribution in heterogeneous networks.
- **Real-World Deployment and Testing:**
    - Extensive field tests on large-scale networks or testbeds (as well as comparisons with other peer-to-peer lookup protocols) would provide additional insights into practical challenges and performance trade-offs.

- Investigating the impact of real-world traffic patterns and diverse failure models may reveal additional optimization strategies for maintaining system stability.

➢ References

- CHORD is a simple Peer to Peer protocol which implements a Distributed Hash Table detailed as per the paper - Stoica, Ion, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. "Chord: A scalable peer-to-peer lookup service for internet applications." ACM SIGCOMM Computer Communication Review 31, no. 4 (2001): 149-160.