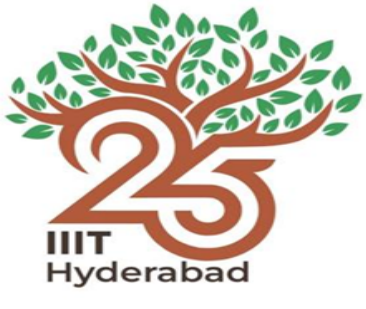


Distributed Systems

Chord Protocol - Fault Tolerance

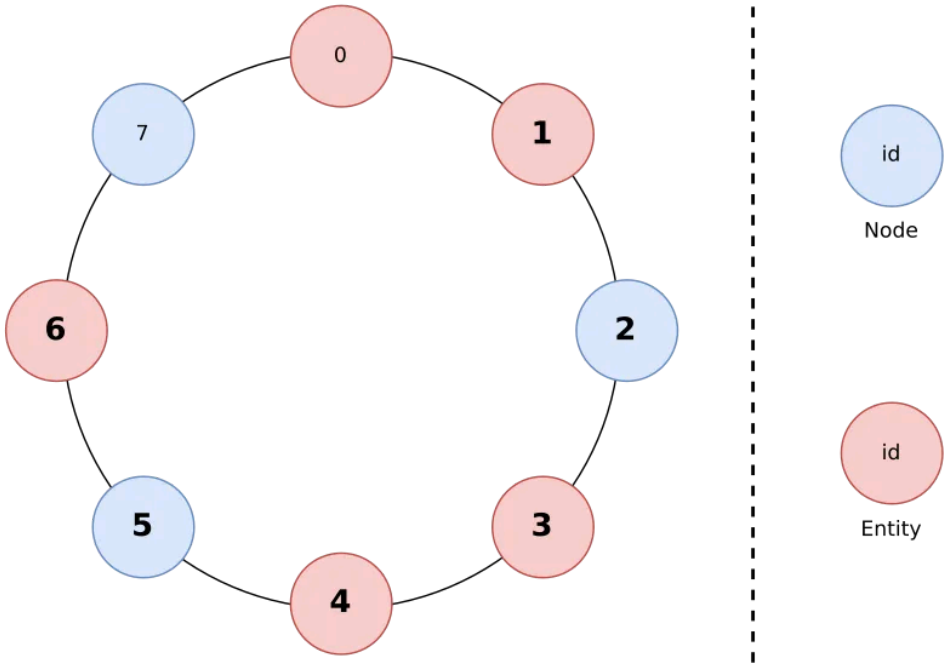
Team 59 members:
Rutuja Deshpande | 2023201032
Shantanu Pathak | 2023202019

Introduction

Chord is a **distributed hash table (DHT)** protocol designed for efficient lookup and retrieval in a decentralized network. It organizes nodes in a logical ring structure, where each node is assigned a unique identifier based on consistent hashing.

Key Features:

1. **Scalability:**
    - Chord scales efficiently with the number of nodes, requiring O(logN)O(\log N)O(logN) time for lookups and O(logN) space per node, where NNN is the total number of nodes.
2. **Key-Value Mapping:**
    - Data keys are hashed into the same identifier space as the nodes, and each key is stored on the node with the next larger identifier (its successor).
3. **Fault Tolerance:**
    - Chord adapts to nodes joining or leaving the system dynamically, maintaining consistent and accurate routing.
4. **Routing Table:**
    - Each node maintains a **finger table** with pointers to other nodes, enabling efficient routing with O(logN) hops.



Why Chord?

Before Chord, Distributed Hash Tables (DHTs) were typically based on a client-server architecture. In this setup, all requests were managed by a single server known as the Coordination Server.

Disadvantages of Client-Server DHT:

1. Uneven responsibility distribution, as not all nodes share equal workloads.
2. Single point of failure, making the system vulnerable if the coordination server fails.

Advantages of Using Chord:

1. Chord is a peer-to-peer DHT that leverages consistent hashing on peer addresses to distribute keys efficiently.
2. Peers take over the role of slaves, ensuring equal participation.
3. Fully distributed architecture eliminates the single point of failure, enhancing robustness and fault tolerance.

Chord Protocol

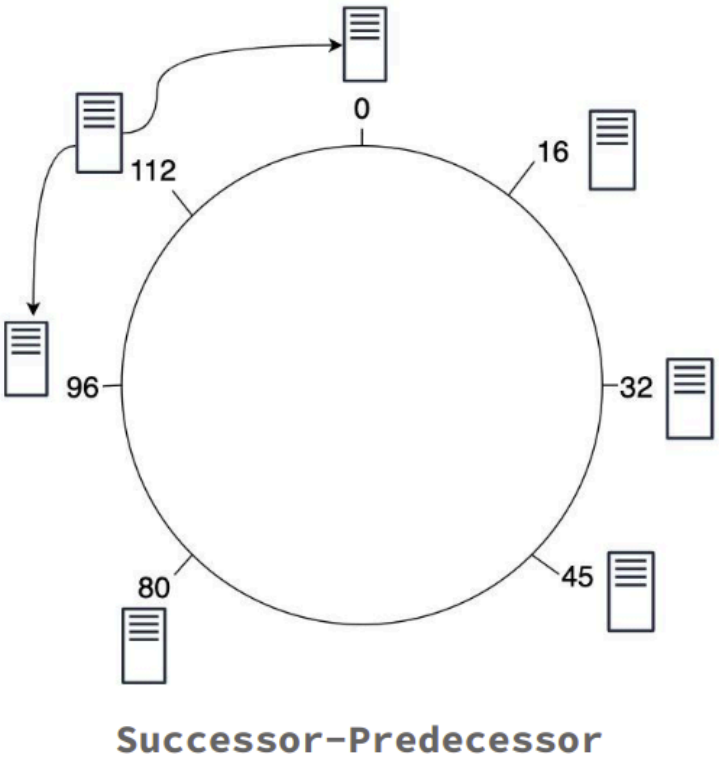
Chord is a peer-to-peer distributed hash table (DHT) protocol that organizes peers in a logical ring structure of size 2^m, where each peer is assigned a unique identifier using consistent hashing. It ensures efficient key lookups and data distribution in a decentralized system.

1. Load Balancing: Any uniform hash function can be used to distribute peers and keys evenly across the ring.
2. Low Conflict Probability: Since $N \ll 2^m$ (where N is the number of peers), key collisions are highly unlikely.

Peer Pointers in Chord:

1. Successor-Predecessor Pointers: Each peer maintains a direct link to its immediate successor and predecessor on the ring for maintaining the structure.
2. Finger Table: Each peer maintains a finger table with pointers to nodes at exponentially increasing distances on the ring. This enables efficient routing, reducing lookup times to O(logN) hops.

Identifiers: Each node and key is assigned a unique ID in the range [0,2^m] using consistent hashing (e.g., SHA-1 or SHA-256).



Finger Table

The **finger table** in a Chord Distributed Hash Table (DHT) is a data structure used by each node to efficiently route requests in the network. It reduces the lookup complexity for a key from O(n) (linear search) to O(logn).

Each node maintains a **finger table**, which contains information about other nodes in the ring. Specifically, the table consists of mmm entries (where m is the number of bits in the node IDs). The size of the table is proportional to m, which determines the identifier space of 2^m nodes.

Entry i in the Finger Table

The ith entry in the finger table of node n points to the **successor node** of the identifier:

$n + 2^{i-1} \bmod 2^m$

Where:

- n: The ID of the current node.
- i: The index of the finger table (1-based).
- 2^m: The maximum size of the identifier space.

1. Successor Lookup:

A node forwards a lookup request to the closest preceding node in its finger table (relative to the target ID). This ensures the request is routed closer to the destination.

Finger[i] = Successor(n + $2^{i-1} \bmod 2^m$)

2. Closest Preceding Node:

To find the closest preceding node for a given key k:

- Traverse the finger table from the highest index down.
- Return the first node f where n < f< k (mod 2^m).

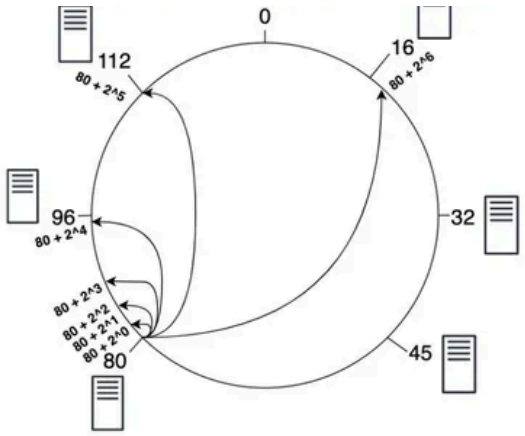
3. Updating the Finger Table:

When a new node joins, it may affect the finger tables of other nodes. The stabilize function and the fix fingers process are responsible for correcting entries.

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)

Node join

When a new node N′ wants to join the Chord ring:

1. **Find the Successor of the New Node:**
    - Identify N″s **successor** in the ring by finding the node responsible for N″s ID.
    - This is achieved by querying any existing node N in the ring with the find_successor(ID) operation.
    - Successor(ID) = First node N such that N.ID ≥ ID (mod 2^m)
2. **Update Successor and Predecessor Links:**
    - N′ sets its **successor** to the node found in step 1.
    - The new node notifies its successor to update its **predecessor** to N′.
3. **Transfer Keys to New Node:**
    - The successor transfers keys that N′ is now responsible for.
    - Keys hashed to K belong to N′ if: K is in (Predecessor.ID, N′.ID]
4. **Stabilize the Ring:**
    - N′ calls the stabilize() method to inform its predecessor and successor about its existence and updates its **finger table**.

Stabilization Post Join

The stabilization process ensures the ring structure is correct after a node joins:

- Each node periodically verifies that its **successor** is accurate.
- Each node notifies its **successor** to update their predecessor links if needed.
- The **finger table** is updated to reflect the new node's presence.

Routing

Route a query for a key k to the node responsible for it (its successor).

Process:

1. **Direct Successor Check:** If k lies between the current node's ID and its successor's ID, the current node routes the query directly to its successor.
2. **Finger Table Lookup:** If k does not fall in the above range, the node routes the query to the closest preceding node (based on finger table entries) whose ID is less than k.
3. **Recursive Routing:** The query is forwarded recursively until the target node (successor of k) is reached.

Routing Complexity: In an N-node network, the lookup requires O(logN) hops on average due to the logarithmic structure of the finger table.

Stabilization

Goals of Stabilization

1. Correct Successor and Predecessor Pointers: Ensure that each node's successor and predecessor pointers correctly reflect the node's neighbors in the ring.
2. Handle Node Joins: Incorporate new nodes into the ring by updating the successor and predecessor pointers of affected nodes.
3. Handle Node Failures or Departures: Adjust the ring to bypass nodes that have failed or left unexpectedly.

1. Finding and Verifying Successor

The successor of a node N is the first node clockwise in the identifier space from N.

- If successor is not correct (or missing), find the correct successor: successor(N) = find_successor(N+1)
- Ask the current successor for its predecessor.
- If this predecessor is a better fit (closer to N), update N's successor.

2. Notifying the Successor

- A node N notifies its current successor of its existence, allowing the successor to update its predecessor.
- Send a message to the successor: notify(successor, N)
- The successor checks if N is closer than its current predecessor. If true, the predecessor is updated to N.

3. Transferring Keys

When a new node N′ joins between N and N.successor, some keys previously managed by N.successor now belong to N′.

N.successor transfers keys for which: $ID(key) \in (ID(N), ID(N.successor)]$

4. Fixing the Ring

Stabilization identifies and corrects inconsistencies caused by joins, failures, or incorrect pointers.

Periodic Tasks:

- Verify and update the successor.
- Notify the successor to update its predecessor.

Server Leaving the Chord Ring

A node in a Chord ring can leave voluntarily (graceful exit) or fail unexpectedly (crash). The graceful departure process involves:

Step 1: Data Transfer

- When a node leaves the ring, it transfers its data to its immediate successor. This ensures no key-value pairs are lost.
- Keys managed by a node n satisfy: Keys={k ∣ predecessor(n)<k≤n} where:
    - predecessor(n) is the ID of the predecessor of node n.
    - The comparison is modulo 2^m, where m is the number of bits in the hash space.

Logical Flow:

1. The departing node identifies its successor.
2. It transfers all keys in its data store to the successor using a send_keys message.
3. Informing its predecessor about its successor.
4. Informing its successor about its predecessor.
5. Updating the finger tables and stabilizing the ring.

Step 2: Notify Predecessor

- The departing node informs its predecessor about its successor so that the predecessor can update its successor pointer.

Logical Flow:

1. The node sends a notify message to its predecessor with the successor's information: notify(predecessor.id,successor.id)
2. The predecessor updates its successor pointer: successor=successor of departing node

Step 3: Notify Successor

- The departing node informs its successor about its predecessor so that the successor can update its predecessor pointer.

Logical Flow:

1. The node sends a notify message to its successor with the predecessor's information: notify(successor.id,predecessor.id)
2. The successor updates its predecessor pointer: predecessor=predecessor of departing node

Step 4: Update Finger Tables

- The finger tables of other nodes in the ring may need to be updated to account for the departing node.

For each node i in the ring:

finger[j]=successor(n + 2^{j-1})

where 1 ≤ j≤m and n is the ID of the node.

Logical Flow:

1. Other nodes periodically run the stabilization process, which updates their finger tables.
2. The departing node does not need to handle this directly; it will occur naturally as part of the stabilization process.

Fault Tolerance in Chord

1. Maintain Redundant Successors (Successor List):

- Instead of maintaining just one successor, maintain a list of successors (e.g., r successors) for each node.
- If the immediate successor fails, the node can refer to the next available successor in the list.

2. Periodic Health Checks (Ping Mechanism):

- Use a heartbeat or ping to check if the immediate successor and predecessor are alive.

- If a node fails to respond within a timeout, remove it from the successor list and update pointers.

3. Replicate Data:

- Replicate data across the r successors to ensure it remains accessible if a node fails.
- When inserting or deleting keys, update all replicas in the successor list.

4. Graceful Handling of Failures:

- Nodes should gracefully handle situations where both the predecessor and immediate successor fail, falling back on successor lists and stabilization.

Time Complexity Analysis

1. Successor List Maintenance:

- Updating the successor list during stabilization involves contacting r nodes, making the time complexity for stabilization: O(r). The value of r is typically a small constant (e.g., 3-5), so the cost is low.

2. Failure Detection (Ping Mechanism):

- Periodically pinging the predecessor and successors involves sending O(r) messages. Each ping takes constant time, resulting in: O(r)
- Again, with a small r, the cost is minimal.

3. Data Replication:

- For a single key operation (insert/delete), replicating across r successors involves: O(r)
- This scales linearly with the replication factor.

4. Lookup Operations:

- The Chord protocol ensures lookup time is: O(logN). With fault tolerance, if the immediate successor fails, the system may need to traverse the successor list, adding an O(r) term to lookups. The overall complexity remains: O(r+logN) ≈ O(logN) (since r is constant).

Benefits of This Approach

1. **High Availability:** The system continues to function even when multiple nodes fail.
2. **Data Reliability:** Data remains accessible due to replication.
3. **Low Overhead:** The added cost is minimal due to the small, constant r.

Conclusion

The fault-tolerant enhancements implemented in the Chord protocol improve the reliability and robustness of the distributed system, ensuring it can withstand node failures without significant disruptions. By introducing features like:

1. **Successor Lists**: Maintaining multiple successors ensures the ring remains connected even when the immediate successor fails.
2. **Failure Detection (Ping Mechanism)**: Periodic health checks proactively identify and handle node failures.
3. **Data Replication**: Storing redundant copies of data across successors safeguards against data loss.
4. **Graceful Recovery**: Stabilization and fallback mechanisms ensure the system self-heals from failures and maintains consistency.

These mechanisms enhance the system's availability and scalability, making it more resilient to real-world challenges like network partitions, hardware failures, and high churn rates.

Possible Future Work

Dynamic Replication Factor (r): Explore dynamically adjusting the replication factor based on node availability, network conditions, or data-criticality requirements.

Advanced Failure Detection: Incorporate more sophisticated failure detection mechanisms, such as:

- Gossip protocols for distributed heartbeat checks.
- Adaptive timeouts based on network latency.

Load Balancing: Implement load-aware mechanisms to distribute data and query loads more evenly across nodes, reducing hotspots and improving performance.

Security Enhancements: Add secure communication (e.g., TLS) and node authentication to protect against malicious nodes in an open network.

References

The following papers and resources are essential for understanding Chord and its fault tolerance challenges:

- "Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications" by Stoica et al., 2001. This foundational paper details the Chord protocol and its use of consistent hashing for DHTs.
- “An adaptive stabilization framework for distributed hash tables” by G Ghinita, YM Teo, 2006. This paper introduces an adaptive stabilization framework for distributed hash tables (DHTs) that intelligently adjusts stabilization rates based on real-time network conditions, aiming to enhance performance and reduce communication overhead during topology changes.
- “Hybrid-Chord: A Peer-to-Peer System Based Chord” by P Flocchini, A Nayak, M Xie, 2005. Thepaper introduces Hybrid-Chord, a peer-to-peer model that significantly reduces lookup hops and enhances routing performance and data availability while maintaining the same complexity as the original Chord system.
- “Towards fault-tolerant chord P2P system: Analysis of some replication strategies” by R Kapelko, 2013. This paper addresses unexpected node failures in Chord-based P2P systems by investigating successor lists and multiple Chord rings replication schemes, demonstrating that high data availability can be maintained with both methods and analyzing their impact on fault tolerance and information recovery.