DSC204A: Scaleable Data Systems Winter 2025

24: The Chord System

Instructor: Umesh Bellur *Scribe: Jingjing Zhu, Junjie Sun, Xiangyi Deng*

Failstop (from last lecture)

Fail-Stop Failures: A process halts permanently and does not recover. Messages from/for a failed process are lost. Our goal is to ensure all correct (non-faulty) processes receive the same set of multicasts, even if some processes fail.

Approach to Handling Fail-Stop Failures:

- **Reliability vs. Ordering:** Reliability ensures messages are delivered to all correct processes. Ordering (FIFO, causal, total) ensures messages are delivered in a specific sequence. These are orthogonal.
- **The "Trick":** Have receivers help the sender.
  1. Sender sends a multicast message M via reliable unicast to all recipients.
  2. Receivers that receive M forward it to all other processes via reliable unicast.

  This redundancy ensures M reaches all correct processes even if the sender fails after sending M.

1 Problem

Considering a scenario: A publisher stores data (e.g., an MP3 file) under a key (e.g., "title"). Clients query the network with Lookup("title") to retrieve the data as Figure 1. However, there are some challenges to do it. The internet comprises distributed nodes ($N_1, N_2, \ldots, N_6$, etc.). Nodes dynamically join/leave, making it difficult to efficiently locate the node storing the key-value pair. To tackle the problem, Chord system is introduced.

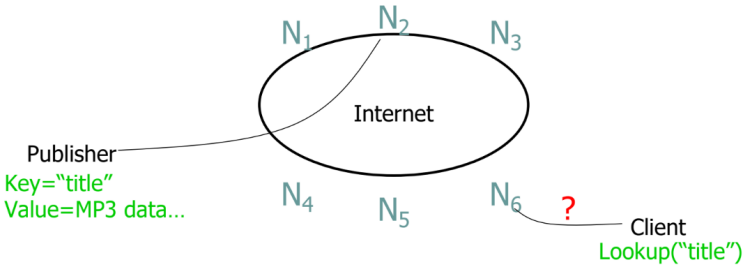

Figure 1: Problem Scenario

1.1 Chord Definition

Definition: A decentralized peer-to-peer (P2P) protocol for key lookup in dynamic networks. It solves problem of distributing and locating a data item in a collection of distributed nodes.

Core Function: Maps a key to the node responsible for storing its value.

Key Features:

- Operates efficiently even as nodes join/leave.
- Supports a single operation: lookup(key) $\rightarrow$ node.

Chord Characteristics:

- Simplicity, provable correctness, and provable performance.
- Each Chord node needs routing information about only a few other nodes.
- Resolves lookups via messages to other nodes (iteratively or recursively).

1.2 Challenges Adressed:

- Load balance: distributed hash function, spreading keys evenly over nodes
- Decentralization: chord is fully distributed, no node more important than other, improves robustness
- Scalability: logarithmic growth of lookup costs with number of nodes in network, even very large systems are feasible
- Availability: chord automatically adjusts its internal tables to ensure that the node responsible for a key can always be found
- Flexible naming: no constraints on the structure of the keys – key-space is flat, flexibility in how to map names to Chord keys

2 Construction of the Chord Ring

2.1 Consistent Hashing

Hash function assigns each node and key an m-bit identifier using a base hash function such as SHA-1.

Hashing Nodes and Keys:

- Nodes: Hashed using properties IP and port.

$$ID_{\text{node}} = \text{hash(IP, Port)}$$

- Keys: Hashed directly.

$$ID_{\text{key}} = \text{hash(key)}$$

- Both are uniformly distributed and exist in the same ID space (e.g., SHA-1)

Identifier Circle: Identifiers are arranged on an identifier circle modulo 2^m => Chord ring. Example: For $m = 3$, IDs range from 0 to 7.

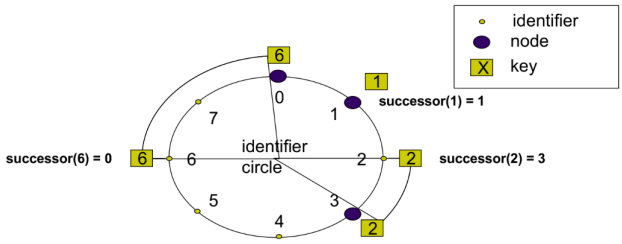

Figure 2: Chord Ring

2.2 Key-Node Assignment

Sucessor Rule: A key k is assigned to the node whose identifier is equal to or greater than the key's identifier. This node is the successor(k) and is the first node clockwise from k.

Example usage as in the chord ring Figure 2: successor(1) = 1, successor(2) = 3, successor(6) = 8

3 Localization of Nodes in Chord

A critical capability in peer-to-peer systems is determining the node responsible for a given key. Chord addresses this challenge using two levels of localization mechanisms: a simple linear approach and an optimized logarithmic technique.

3.1 Basic Node Localization

In the most basic form of Chord lookup, a node only knows its immediate successor. To find the node responsible for a key k, node n checks whether k lies between itself and its successor:

```
n.find_successor(id):
    if id in (n, successor]:
        return successor
    else:
        return successor.find_successor(id)
```

This approach may require $O(N)$ hops in a system with N nodes.

3.2 Scalable Node Localization via Finger Tables

To reduce lookup costs, Chord introduces the **finger table**, a routing table that enables logarithmic lookup time. Each node maintains up to m entries (where m is the number of bits in the identifier space). The i-th

entry in the finger table of node n points to:

$$\text{finger}[i] = \text{successor}(n + 2^{i-1})$$

This design allows each node to learn exponentially more distant nodes as i increases.

3.3 Key Characteristics

- Each node only stores $O(\log N)$ entries.
- Nodes have better routing knowledge of nearby nodes than distant ones.
- The finger table alone is not sufficient to locate an arbitrary key immediately but helps narrow the search quickly.

3.4 Efficient Lookups Using Finger Tables

To perform a lookup, node n:

1. Scans its finger table for the node f with the highest ID less than the target key.
2. Forwards the lookup request to f.
3. This process continues recursively or iteratively until the responsible node is found.

This process ensures a lookup cost of $O(\log N)$ hops in expectation.

3.5 Conclusion

Chord's localization strategy effectively balances the trade-off between minimal per-node state and fast lookup times. The scalable design of the finger table is essential to Chord's performance and makes it practical for large, dynamic distributed systems.

4 Node Joins and Stabilization

4.1 Node Joins and Stabilization

Identifier Space & Finger Tables Chord arranges nodes on a circle modulo 2^m. Each node n maintains a finger table of size m, where entry i points to successor$(n+2^{i-1} \bmod 2^m)$. This structure supports $O(\log N)$ lookups once stabilized.

Stabilization Protocol Each node n periodically runs:

`stabilize()` 1. Query successor s for its predecessor p.

2. If $p \in (n, s)$, set successor$_n \leftarrow p$.

3. Invoke `notify(`n`)` on the (possibly updated) successor.

`notify(n')` If $n' \in (s.\text{predecessor}, s)$ or no predecessor, set $s.\text{predecessor} \leftarrow n'$ and transfer keys in $(\text{pred}_s, n']$.

`fix_fingers()` On each run, pick next index i, compute

$$\text{start} = (n.\text{id} + 2^{i-1}) \bmod 2^m,$$

then update

$$\text{finger}_n[i] \leftarrow \text{find\_successor}(\text{start}).$$

Node Join Sequence When node N_{26} joins nodes N_{32} and N_{21}:

1. *Locate successor:* $N_{26} \rightarrow \text{find\_successor}(26)$ on N_{32}.
2. *Notify successor:* N_{26} calls `notify(`N_{26}`)` on N_{32}, so $N_{32}.\text{predecessor} \leftarrow N_{26}$.
3. *Key transfer:* N_{26} pulls keys in $(\text{pred}_{32}, 26]$ from N_{32}.
4. *Finger repair:* Concurrent `fix_fingers()` calls on all nodes update finger entries pointing past N_{26}.
5. *Neighbor stabilization:* N_{21}`.stabilize()` sees N_{26} and updates $N_{21}.\text{successor}$.
6. *Mutual notify:* N_{21} calls `notify(`N_{21}`)` on N_{26}, so $N_{26}.\text{predecessor} \leftarrow N_{21}$.

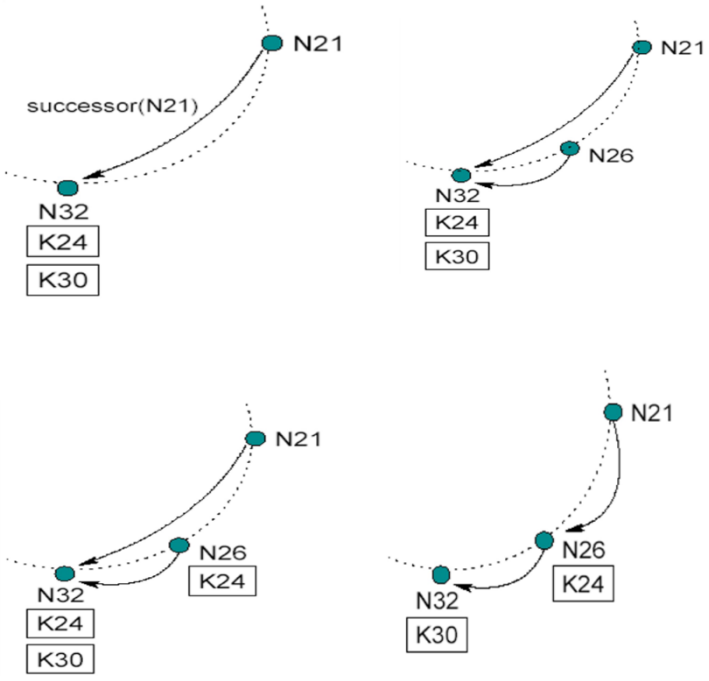

Figure 3: Node join and stablization

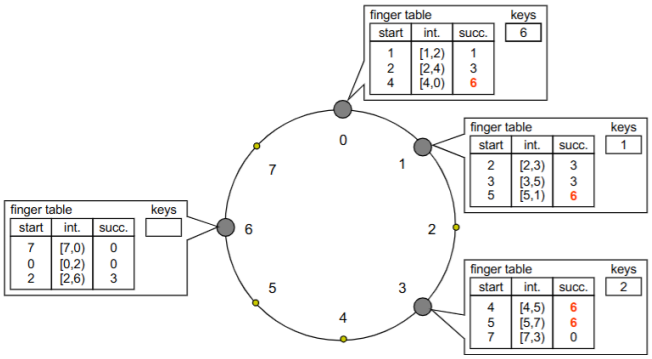

Figure 4: Node joins with figure tables

4.2 Node Departures

When a node departs (voluntary or faulty), its neighbors and successor list handle repair:

- Each node keeps an r-entry *successor list* of immediate successors for failover.
- On detecting failure (via `check_predecessor()`), nodes skip dead entries.
- Periodic `stabilize()` and `fix_fingers()` rebuild pointers.

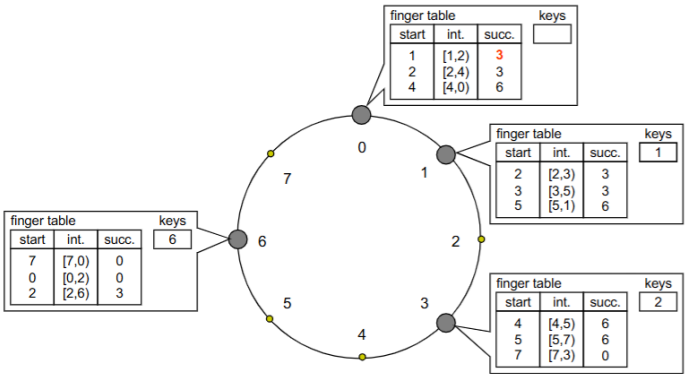

Figure 5: Node departures with figure tables

4.3 Impact of Node Joins on Lookups

Immediately after a join, routing tables may be inconsistent:

- **All pointers correct:** $O(\log N)$ hops.
- **Successors correct but fingers stale:** Correct but via linear hops until fingers update.

- **Incorrect successors:** Initial lookup may contact wrong node and backtrack, still terminates correctly.

Eventually, after stabilization and finger fixes, performance returns to $O(\log N)$. Temporary degradation is bounded by the stabilization period (usually $O(\log N)$ rounds).