CS 7280/4973: Data Str & Alg Scalable Comp	Spring 2025
Lecture 23 — April 7, 2025	
Prof. Prashant Pandey	*Scribe: Ansh Mehta*

1 Overview

In the previous lecture, we discussed the problem of distributing m objects across n nodes within a distributed system. We noticed load balancing and asymptotic scalability as the primary challenges. **Consistent hashing** was introduced as a approach, mapping both data objects and system nodes on a single virtual ring. We also saw how a data structure like a Binary Search Tree (BST) helps in finding the successor node for any given id on the ring.

In this lecture, we look at the problem of where to store this BST (or similar structure) that helps identify successors on the consistent hashing ring and where a request for a specific object should be routed. How does one determine which node is responsible for a particular piece of data without needing a central map or directory?

To address this, we will explore **Distributed Hash Tables (DHTs)**. We will examine **Chord**, which is a DHT protocol and implementation of consistent hashing. Chord provides a solution for managing this node-to-data mapping information in a scalable way.

2 Background

2.1 Motivation

Napster: A peer-to-peer music sharing application. It allowed users to share content and allowed people to search each other's hard drive and transfer songs [1]. Napster used central servers to provide a real-time directory or index [2].

2.2 Problem

Similarly to the Napster application, the concept of sharing files/objects can be extended to any application. At the core of this distributed sharing, the main model is defined below:

Model

1. Each user stores a subset of files.
2. Each user has access to the files of all users in the system.

Hence, the **main challenge** to address is to identify **where a particular file is saved?**

This must be achieved while satisfying the following requirements:

1. **Scalability**: The system should support a large number (e.g., up to millions) of participating nodes.
2. **Dynamicity**: The nodes can join and leave the system, and the system must adapt to these changes.

2.3 Some Solutions

2.3.1 Centralized Index System

Similar to Napster, we can create a **centralized index system** that maps files to the machine that is alive.

How to find a file?: A node queries the central index server(s) with the file identifier. The index returns the network address of the machine(s) storing the required file. (Optionally, the index could store extra information like load or proximity to help choose the best machine).

Advantages

1. **Simplicity:** Easier to manage compared to fully decentralized systems. Only central servers need state management.
2. **Efficiency:** Look-ups can be very fast if the index is well maintained and provisioned.
3. **Functionality:** Easy to implement sophisticated search features (e.g., keyword search) on top of the centralized index.

Disadvantages

1. **Single Point of Failure:** If the central server(s) goes down, the entire system becomes unusable for look-ups. Redundancy can mitigate this, but adds complexity.
2. **Scalability Bottleneck:** The central server(s) must handle all lookup requests and updates (node joins/leaves, file additions/removals). This can become a performance bottleneck as the system grows.

2.3.2 Gnutella

Gnutella is a decentralized peer-to-peer (P2P) protocol, forming an network where peers connect directly to a set of other peers (neighbors).

Key Idea: Flood the request across the peer-to-peer network.

How to find a file?

1. Send request to all neighbors.
2. Neighbors recursively multi-cast.

3. Eventually a machine that has the file receives the request and sends back the file.

Advantages

1. **Decentralized:** No reliance on any central server.
2. **Robustness:** The system can tolerate failures of individual nodes without bringing down the entire search capability.

Disadvantages

1. The entire network is swarmed with requests, the query flood can overwhelm nodes,
   (a) To get rid of this problem one can set a TTL (Time to live) counter on each request. This helps requests to bounded for a set number of hops.
2. **Limited Search Scope:** The TTL mechanism, while necessary to control flooding, means that searches only explore a limited neighborhood of the network. Files held by nodes exceeding the TTL may not be found.

3 The Distributed Hash Table or DHT

Abstraction: A distributed hash table data structure that supports operations like:

1. $Insert(id, item)$: Stores the 'item' associated with id. The DHT automatically determines the responsible node based on the 'id' and stores the mapping there.
2. $item = Query(id)$: Retrieves the 'item' associated with 'id'. The query is efficiently routed through the network to the responsible node, which then returns the result.
   (Note: Item can be anything - a data object, document, file, pointer to a file.)

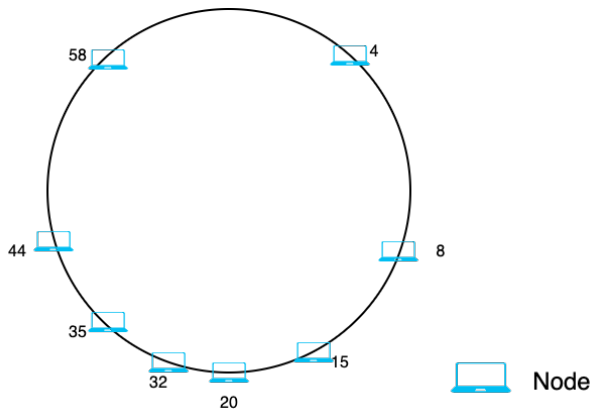


Figure 1:

Design Goals:

1. **Guaranteed Lookup:** Ensure that if an item exists in the DHT, a query for it will successfully locate it.
2. **Scalability:** Efficiently scale to potentially thousands of nodes in terms of search performance.
3. **Robustness:** Handle frequent node arrivals and departures gracefully.

Several DHT protocols were proposed, including CAN, Chord, Kademlia, Pastry, and Tapestry. We will focus on the Chord [3].

3.1 Chord

Chord is a specific DHT protocol that implements consistent hashing on a circular identifier space.

Identifier Space and Consistent Hashing:

- Chord uses a m bit identifier space, arranged as a circle (ring) from 0 to $2^m - 1$.
- A consistent hash function maps both node identifiers and item keys to points on this ring.

Properties

1. Routing table of size $O(\log n)$, where n is the number of nodes.
2. Guarantees that a file is found in the $O(\log n)$ steps.
3. Each node maintains a pointer to its successor node.

Chord supports two operations to support these properties, LOOKUP and JOIN operations.

Joining Operation

We use the successor pointer and two messages - *stabilize()* and *notify()*.

Steps

1. Each node (say A) periodically sends the *stabilize()* message to its successor (say B).
2. Upon receiving a *stabilize* message, the Node B returns its predecessor $B^{'} = predeccesor(B)$ to A by sending *notify($B^{'}$)* message.
3. When node A receives notify($B^{'}$) from B,
   (a) If B' between A and B: A new node node was added and A needs to update its successor to B'.
   (b) Else: Do not do anything.

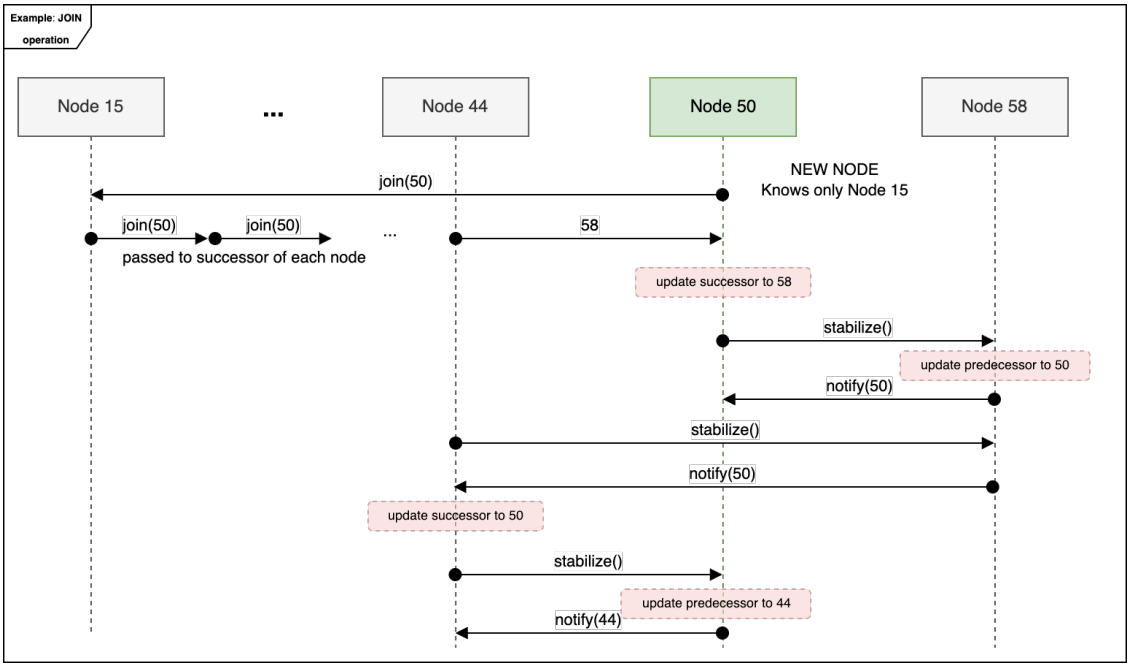


Figure 2: Example of JOIN operation (a new node is added to the system)

Figure 2 demonstrates a join operation.

Lookup Operation

Idea 1: Linear Scan - Perform a linear scan starting from a node and traversing to each successor, checking if the file is found on each of the nodes. The total number of hops in this approach is O(N)

Idea 2: Sparsification: Chord employs "finger tables". This is a form of index sparsification where instead of a central node maintaining full information about each node, we maintain at each node some extra information about a subset of nodes.

Finger Table

Each node n maintains a finger table ft with up to m entries (where m is the number of bits in our ring representation). The i-th entry in the finger table of node n points to the first node that succeeds n by at least 2^{i-1} in the identifier circle (modulo 2^m).

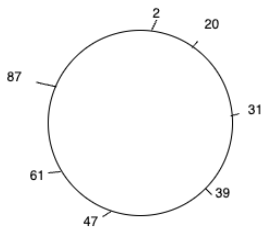
Specifically:

$ft[i] = successor((n + 2^{i-1}) \mod 2^m)$ for $i = 1, ..., m$.

Each node maintains a finger table of size $\log N$, and we can route any request to the correct node in at most $\log N$ steps.

Example:

We built a finger table with the ring topology given in Figure 3 for the node identified with $Nid = 20$.



i	ft[i] (calculation)	ft[i]
0	succ((31 + 0)mod128)	31
1	succ((31+2)mod128)	31
2	succ((31+4)mod128)	31
3	succ((31+8)mod128)	39
4	succ((31+16)mod128)	47
5	succ((31+32)mod128)	87
6	succ((31+64)mod128)	2

Figure 3: Finger table for the Nid = 20 in the ring.

References

[1] Peter Jan Honigsberg. The Evolution and Revolution of Napster. *University of San Francisco Law Review*, 36:473–508, 2002.

[2] Stefan Saroiu, Krishna P. Gummadi, Steven D. Gribble. A Measurement Study of Peer-to-Peer File Sharing Systems. *Proc. SPIE Multimedia Computing and Networking*, 4673:156–170, 2002.

[3] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. *SIGCOMM Computer Communication Review*, 31(4):149–160, August 2001.