CS 7800 / 4810 — Advanced Data Structures

Lecture Notes: Distributed Hash Tables and Chord

---

1 How Did It Start?

A killer application: **Napster** — free music over the Internet.

The key idea is to share the content, storage, and bandwidth of individual users rather than relying on a central server.

1.1 Model

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

1.2 Main Challenges

- **Lookup:** Find where a particular file is saved.
- **Scale:** Support up to hundreds of thousands or millions of machines.
- **Dynamicity:** Machines can come and go at any time (churn).

2 Napster: Centralized Index

Napster assumes a *centralized index system* that maps files to machines that are alive.

2.1 How to Find a File

Query the index system, which returns a machine that stores the requested file. Ideally, this is the closest or least loaded machine.

2.2 Advantages

Simplicity: easy to implement. One can build sophisticated search engines on top of the index system.

2.3 Disadvantages

The centralized index is a single point of failure. It suffers from poor **robustness** and **scalability** — the central server must handle every lookup from every user in the system.

3 Gnutella: Flooding

Gnutella distributes file location information by *flooding* the request across the network.

3.1 How to Find a File

1. Send the request to all neighbors.
2. Neighbors recursively multicast the request.
3. Eventually, a machine that has the file receives the request and sends back the answer.

3.2 Advantages

Totally decentralized and highly robust — no single point of failure.

3.3 Disadvantages

- **Not scalable:** the entire network can be swamped with requests.
- To alleviate this problem, each request has a **TTL** (Time to Live) that limits the number of hops.
- The topology is ad-hoc: queries are flooded for a bounded number of hops.
- **No guarantees on recall:** a file may exist in the system but never be found if it is beyond the TTL horizon.

4 Distributed Hash Tables (DHTs)

Definition 4.1 (Distributed Hash Table)**.** A **DHT** is a distributed data structure that provides a hash-table-like interface across a network of machines. It supports two operations:

- INSERT$(id, item)$ — store an item under a given key.
- $item =$ QUERY(id) — retrieve the item associated with a given key.

An item can be anything: a data object, a document, a file, or a pointer to a file.

4.1 Proposals

Several DHT designs were proposed in the early 2000s: CAN, **Chord**, Kademlia, Pastry, Tapestry, among others.

4.2 DHT Design Goals

- Make sure that an item identified by a key is *always found*.
- Scale to hundreds of thousands of nodes.
- Handle rapid arrival and failure of nodes (churn).

5 Chord

5.1 Connection to Consistent Hashing

Chord is a direct application of consistent hashing to the peer-to-peer routing problem. In the previous lecture, consistent hashing was presented as a way to assign m items to n caches on a ring so that adding or removing a cache only moves $O(m/n)$ items. Chord takes this idea and turns it into a *fully decentralized lookup protocol*: both nodes and data items are hashed onto the same $[0, 2^M)$ identifier ring, and each item is stored at its successor node — exactly the consistent hashing rule.

The key difference is that in classical consistent hashing a single BST (or a centralized directory) is used to find the successor of any point on the ring. In a distributed setting, no single machine holds the full ring. Chord's contribution is a *decentralized routing algorithm*: each node maintains only $O(\log N)$ pointers (the finger table), and any lookup is resolved in $O(\log N)$ hops by successively halving the remaining distance on the ring. Chord also introduces a STABILIZE protocol that maintains correct successor pointers as nodes join and leave, ensuring that the consistent hashing invariant is preserved under churn.

In short: consistent hashing defines *where* data lives; Chord defines *how to find it* without centralized coordination.

5.2 Overview

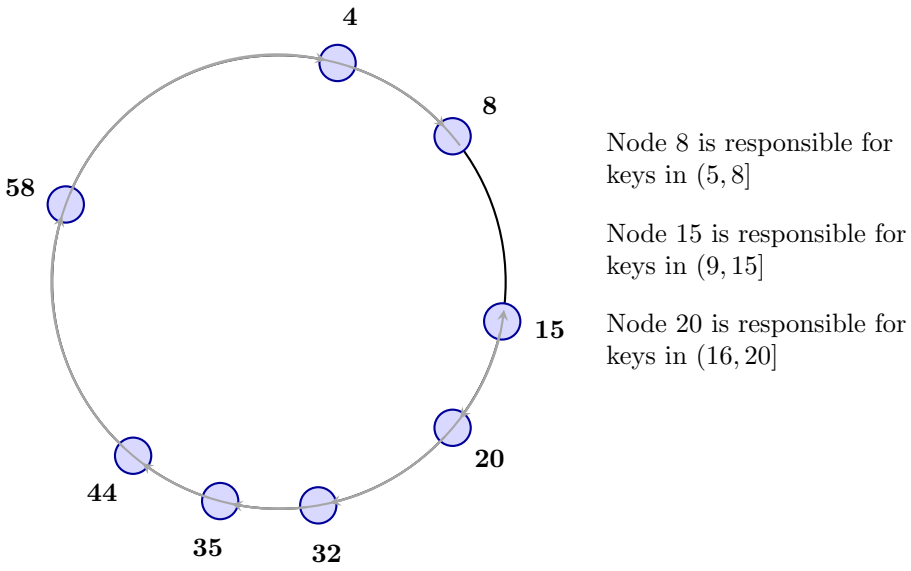
- Associate to each node and item a unique id in a uni-dimensional identifier space $\{0, 1, \ldots, 2^M - 1\}$.
- Key design decision: **decouple correctness from efficiency**. Correctness depends only on successor pointers; the finger table is an optimization for fast lookups.

5.3 Properties

- Routing table size: $O(\log N)$, where N is the total number of nodes.
- Guarantees that a file is found in $O(\log N)$ steps.

5.4 Ring Structure

Nodes are arranged on an identifier ring of size 2^M. Each node maintains a pointer to its **successor** (the next node clockwise on the ring).



6 Lookup

In the simplest form, each node maintains only its successor. To route a packet $(ID, data)$, forward it along successor pointers until it reaches the node responsible for ID.

This naïve approach is correct but requires $O(N)$ hops in the worst case.

7 Achieving Efficiency: Finger Tables

To achieve $O(\log N)$ lookup, each node maintains a **finger table** with M entries (where $M = \lceil \log_2 N_{\max} \rceil$ is the number of bits in the identifier space).

Definition 7.1 (Finger Table)**.** The i-th entry in the finger table of a node with identifier n is:

$$ft[i] = \text{first node with id} \geq (n + 2^i) \bmod 2^M, \qquad i = 0, 1, \ldots, M-1.$$

That is, the i-th finger points to the successor of the point $n + 2^i$ on the ring.

Example 7.1 (Finger Table at Node 80, $M = 7$)**.** The identifier space is $\{0, 1, \ldots, 127\}$. The finger table at node 80:

i	Start: $(80 + 2^i) \bmod 128$	$ft[i]$
0	81	96
1	82	96
2	84	96
3	88	96
4	96	96
5	112	112
6	16	20

Notice that the first five fingers all point to node 96 because there is no node between 81 and 96. Finger 5 jumps to 112, and finger 6 wraps around the ring to node 20.

7.1 Lookup with Finger Tables

To look up key k at node n:

1. If $k \in (n, \mathit{successor}(n)]$, the successor holds the key. Return it.
2. Otherwise, find the *largest* finger $ft[i]$ such that $ft[i] \in (n, k)$ on the ring, and forward the query to $ft[i]$.

Each hop at least halves the remaining distance on the ring, so the lookup completes in $O(\log N)$ hops.

Theorem 7.1. *With high probability, a Chord lookup visits $O(\log N)$ nodes.*

8 Joining Operation

When a new node joins the ring, it must integrate itself into the successor chain. Chord uses a STABILIZE protocol to accomplish this.

8.1 Protocol

- Each node A periodically sends a STABILIZE() message to its successor B.
- Upon receiving STABILIZE(), node B returns its predecessor $B' = \text{pred}(B)$ to A by sending a NOTIFY(B') message.
- Upon receiving NOTIFY(B') from B:
  - If B' is between A and B on the ring, then A updates its successor to B'.
  - Otherwise, A does nothing.

8.2 Worked Example: Node 50 Joins

Consider the ring with existing nodes $\{4, 8, 15, 20, 32, 35, 44, 58\}$. Node 50 wants to join and knows that node 15 is already in the system.

1. **Node 50 sends Join(50) to Node 15.** The request is routed around the ring. Node 44 returns node 58 as the successor of 50. Node 50 sets its successor to 58.
2. **Node 50 sends Stabilize to Node 58.** Node 58 sees that its current predecessor is 44, but 50 is between 44 and 58. Node 58 updates its predecessor to 50 and sends NOTIFY(50) back.
3. **Node 44 sends Stabilize to Node 58.** Node 58 replies with NOTIFY(50) (its new predecessor). Node 44 sees that 50 is between 44 and 58, so it updates its successor to 50.
4. **Node 44 sends Stabilize to Node 50.** Node 50 sees that it has no predecessor, so it sets its predecessor to 44.
5. **Joining complete.** The ring now correctly includes node 50: $\ldots \to 44 \to 50 \to 58 \to \ldots$

Remark 8.1. The STABILIZE protocol runs continuously and in the background. Multiple concurrent joins are handled correctly because each round of stabilization makes local progress toward the correct ring topology.

9 Summary: Napster vs. Gnutella vs. Chord

	Napster	**Gnutella**	**Chord**
Architecture	Centralized index	Fully decentralized	Structured DHT
Lookup cost	$O(1)$ (index)	$O(N)$ flood	$O(\log N)$ hops
State per node	—	Neighbor list	$O(\log N)$ fingers
Guaranteed recall	Yes	No (TTL-bounded)	Yes
Robustness	Single point of failure	Highly robust	Robust (stabilize)
Scalability	Poor	Poor (flooding)	Good