ATTRIBUTION

• These slides are released under an Attribution-NonCommercial-ShareAlike 3.0 Unported (CC BY-NC-SA 3.0) Creative Commons license

• These slides incorporate material from:
  • Kyle Jamieson, Princeton University (also under a CC BY-NC-SA 3.0 Creative Commons license)
ANNOUNCEMENTS
OUTLINE

1. Peer-to-peer overview
2. DHTs
3. Chord
WHAT IS A PEER-TO-PEER (P2P) SYSTEM?

- A **distributed** system architecture:
- **No centralized control**
- Nodes are **roughly symmetric** in function
- **Large** number of **unreliable** nodes
P2P ADOPTION

• Successful adoption in *some niche areas* –

1. Client-to-client (legal, illegal) **file sharing**
   • Popular data but owning organization has no money

2. **Digital currency**: no natural single owner (Bitcoin)

3. **Voice/video telephony**: user to user anyway
   • Issues: Privacy and control
EXAMPLE: CLASSIC BITTORRENT

1. User clicks on download link
   - Gets **torrent** file with content hash, IP addr of **tracker**

2. User’s BitTorrent (BT) client talks to tracker
   - Tracker tells it **list of peers** who have file

3. User’s BT client downloads file from one or more peers

4. User’s BT client tells tracker it has a copy now, too

5. User’s BT client serves the file to others for a while

Provides huge download bandwidth, without expensive server or network links
THE LOOKUP PROBLEM

Publisher (N₄)

put(“Star Wars.mov”, [content])

Internet

N₂

N₃

N₅

N₆

Client

get(“Star Wars.mov”)
CENTRALIZED LOOKUP (NAPSTER)

Publisher (N4)

SetLoc("Star Wars.mov", IP address of N4)

db

key="Star Wars.mov", value=[content]

Client

Lookup("Star Wars.mov")

Simple, but $O(N)$ state and a single point of failure
Robust, but $O(N = \text{number of peers})$ messages per lookup
Can we make it robust, reasonable state, reasonable number of hops?
OUTLINE

1. Peer-to-peer overview
2. DHTs
3. Chord
WHAT IS A DHT (AND WHY)?

• Local hash table:
  
  key = Hash(name)

  put(key, value)

  get(key) → value

• Service: Constant-time insertion and lookup

How can I do (roughly) this across millions of hosts on the Internet?
Distributed Hash Table (DHT)
WHAT IS A DHT (AND WHY)?

• Distributed Hash Table:
  
  key = hash(data)

  lookup(key) \rightarrow IP addr (Chord lookup service)

  send-RPC(IP address, put, key, data)

  send-RPC(IP address, get, key) \rightarrow data

• Partitioning data in truly large-scale distributed systems
  • Tuples in a global database engine
  • Data blocks in a global file system
  • Files in a P2P file-sharing system
WHY MIGHT DHT DESIGN BE HARD?

• Decentralized: no central authority

• Scalable: low network traffic overhead

• Efficient: find items quickly (latency)

• Dynamic: nodes fail, new nodes join
OUTLINE

1. Peer-to-peer overview
2. DHTs
3. Chord
CHORD LOOKUP ALGORITHM PROPERTIES

• **Interface:** lookup(key) → IP address

• **Efficient:** $O(\log N)$ messages per lookup
  • $N$ is the total number of servers

• **Scalable:** $O(\log N)$ state per node

• **Robust:** survives massive failures
CHORD IDENTIFIERS

- **Key identifier** = SHA-1(key)

- **Node identifier** = SHA-1(IP address)

- SHA-1 distributes both uniformly

- *How does Chord partition data?*
  - *i.e.*, map key IDs to node IDs
CONSISTENT HASHING [Karger ‘97]

Key is stored at its successor: node with next-higher ID.
CHORD: SUCCESSOR POINTERS
**SIMPLE LOOKUP ALGORITHM**

\[
\text{Lookup}(\text{key-id})
\]

\[
\text{succ} \leftarrow \text{my successor}
\]

\[
\text{if } \text{my-id} < \text{succ} < \text{key-id} \quad \text{/**next hop**}
\]

\[
\text{call Lookup(\text{key-id}) on succ}
\]

\[
\text{else} \quad \text{/**done**}
\]

\[
\text{return succ}
\]

- **Correctness** depends only on **successors**
• **Problem:** Forwarding through successor is slow

• **Data structure is a linked list:** $O(n)$

• **Idea:** Can we make it more like a binary search?
  • Need to be able to halve distance at each step
• Skip Lists (Pugh, 1989)
• Consider a linked list:

*Head* → 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10 → *NIL*

• Lookup time: O(n)
**CHORD INTUITION**

- Skip Lists (Pugh, 1989)
- Consider a linked list:
  - Add 2nd row of pointers spaced further apart
    - Still $O(n)$, but more efficient
    - Use 2nd row to get as close as possible without going over
    - Then last row to get to the desired element
**CHORD INTUTION**

- Skip Lists (Pugh, 1989)
- Consider a linked list:

  ![Linked List Diagram](image)

- Add $\log(N)$ rows
  - Get as close as possible on top row, then drop down a row, then drop down another row, until the bottom row
  - $O(\log N)$ lookup time
“FINGER TABLE” ALLOWS LOG N-TIME LOOKUPS
FINGER $I$ POINTS TO SUCCESSOR OF $N + 2^I$
FINGER TABLE EXAMPLE
**LOOKUP WITH FINGER TABLE**

**Lookup**(key-id)

look in local finger table for highest n: my-id < n < key-id

if n exists

call Lookup(key-id) on node n  //next hop

else

return my successor //done
LOOKUPS TAKE $O(\log N)$ HOPS
AN ASIDE: IS LOG(N) FAST OR SLOW?

• For a million nodes, it’s 20 hops

• If each hop takes 50 milliseconds, lookups take a second

• If each hop has 10% chance of failure, it’s a couple of timeouts

• So in practice log(n) is better than O(n) but not great
JOINING: LINKED LIST INSERT

1. Lookup(36)

N36

N25

N40

K30

K38
2. N36 sets its own successor pointer
JOIN (3)

3. Copy keys 26..36 from N40 to N36
NOTIFY MESSAGES MAINTAIN PREDECESSORS

notify N25

N25

N40

N36

notify N36
"My predecessor is N36."
JOINING: SUMMARY

- Predecessor pointer allows link to new node
- Update finger pointers in the background
- Correct successors produce correct lookups
WHAT DHTS GOT RIGHT

• **Consistent hashing**
  - Elegant way to divide a workload across machines
  - Very useful in clusters: actively used today in Amazon Dynamo and other systems

• **Replication** for high availability, efficient recovery after node failure

• **Incremental scalability**: “add nodes, capacity increases”

• **Self-management**: minimal configuration

• **Unique trait**: no single server to shut down/monitor
UC San Diego