

Peer-to-Peer Systems and Distributed Hash Tables



COS 418: *Distributed Systems*
Lecture 10

Kyle Jamieson

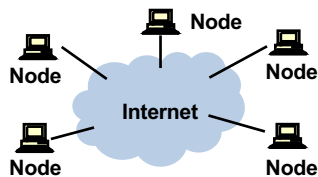
[Credit: Selected content adapted from B. Karp, R. Morris]

Today

1. Peer-to-Peer Systems
 - Napster, Gnutella, BitTorrent, challenges
2. Distributed Hash Tables
3. The Chord Lookup Service
4. Concluding thoughts on DHTs, P2P

2

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

3

Why might P2P be a win?

- **High capacity for services** through parallelism:
 - Many disks
 - Many network connections
 - Many CPUs
- **Absence of a centralized server** or servers may mean:
 - **Less chance** of service overload as load increases
 - Easier **deployment**
 - A single failure **won't wreck** the whole system
 - System as a whole is **harder to attack**

4

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

5

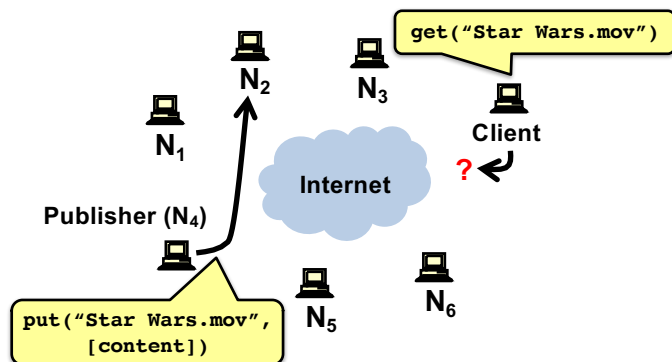
Example: Classic BitTorrent

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

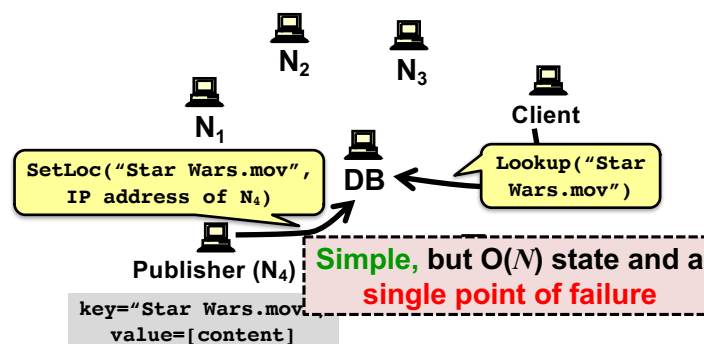
6

The lookup problem



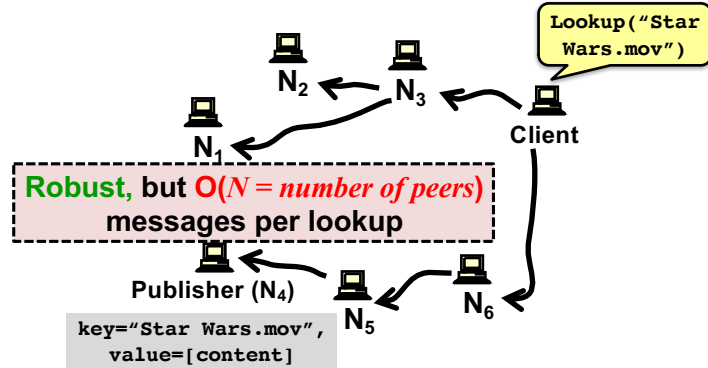
7

Centralized lookup (Napster)



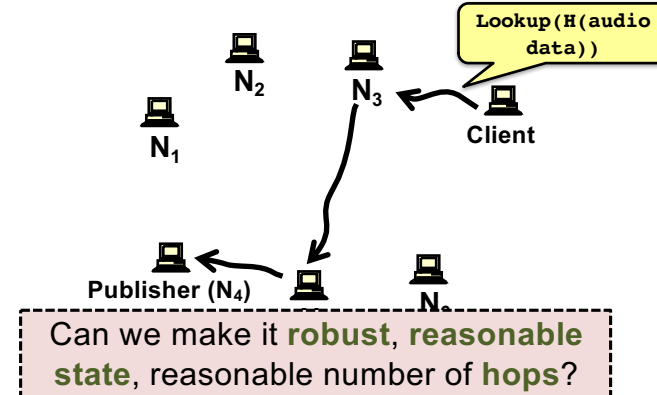
8

Flooded queries (original Gnutella)



9

Routed DHT queries (Chord)



10

Today

1. Peer-to-Peer Systems
- 2. Distributed Hash Tables**
3. The Chord Lookup Service
4. Concluding thoughts on DHTs, P2P

11

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)

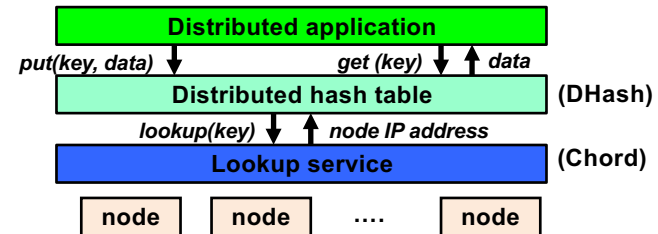
12

What is a DHT (and why)?

- Distributed Hash Table:
key = hash(data)
lookup(key) → IP addr (**Chord lookup service**)
send-RPC(IP address, **put**, key, data)
send-RPC(IP address, **get**, key) → 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

13

Cooperative storage with a DHT



- App may be **distributed** over many nodes
- DHT **distributes data storage** over many nodes

14

BitTorrent over DHT

- BitTorrent can use DHT instead of (or with) a tracker
- BT clients use DHT:
 - Key = **file content hash** (“infohash”)
 - Value = **IP address of peer** willing to serve file
 - Can store multiple values (*i.e.* IP addresses) for a key
- Client does:
 - get (infohash) to find other clients willing to serve
 - put (infohash, my-ipaddr) to identify itself as willing

15

Why might DHT be a win for BitTorrent?

- The DHT comprises a single giant tracker, less fragmented than many trackers
 - So peers more likely to **find each other**
- Maybe a classic tracker too exposed to **legal & c. attacks**

16

Why the put/get DHT interface?

- API supports a wide range of applications
 - DHT imposes no structure/meaning on keys
- Key/value pairs are persistent and global
 - Can store keys in other DHT values
 - And thus build complex data structures

17

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

18

Today

1. Peer-to-Peer Systems
2. Distributed Hash Tables
3. **The Chord Lookup Service**
 - **Basic design**
 - Integration with *DHash* DHT, performance

19

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
- **Simple to analyze**

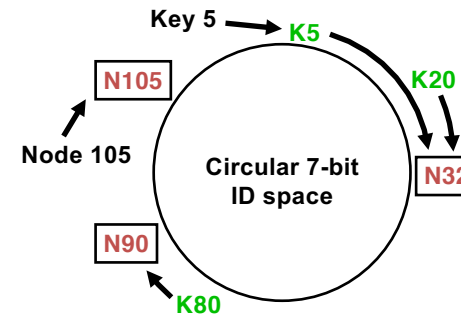
20

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

21

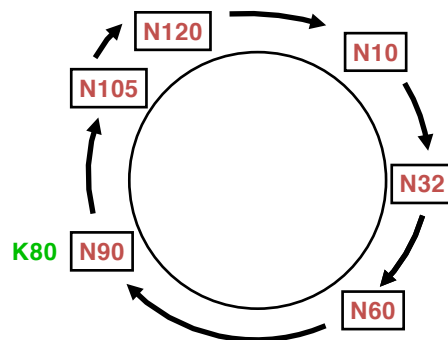
Consistent hashing [Karger '97]



Key is stored at its **successor**: node with next-higher ID

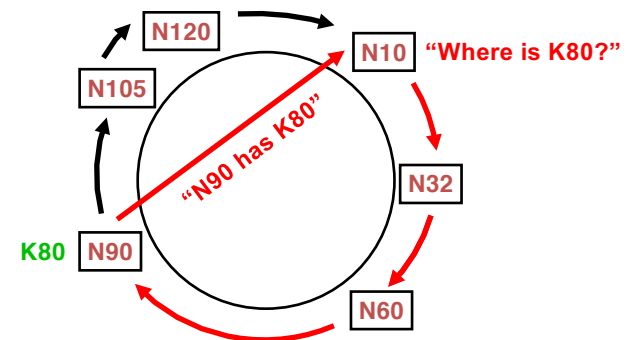
22

Chord: Successor pointers



23

Basic lookup



24

Simple lookup algorithm

```
Lookup(key-id)
  succ ← my successor
  if my-id < succ < key-id //next hop
    call Lookup(key-id) on succ
  else //done
    return succ
```

- **Correctness** depends only on **successors**

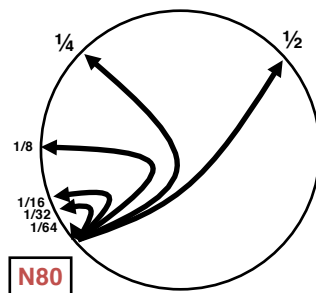
25

Improving performance

- **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

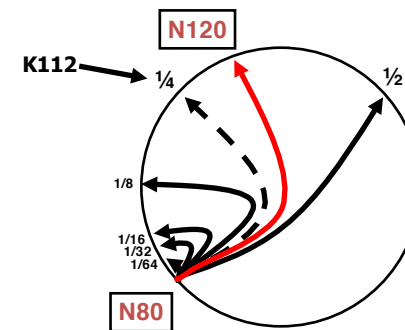
26

“Finger table” allows log N-time lookups



27

Finger i Points to Successor of $n+2^i$



28

Implication of finger tables

- A **binary lookup tree** rooted at every node
 - Threaded through other nodes' finger tables
- This is **better** than simply arranging the nodes in a single tree
 - Every node acts as a root
 - So there's **no root hotspot**
 - **No single point** of failure
 - But a **lot more state** in total

29

Lookup with finger table

Lookup(key-id)

look in local finger table for
highest n : $\text{my-id} < n < \text{key-id}$

if n exists

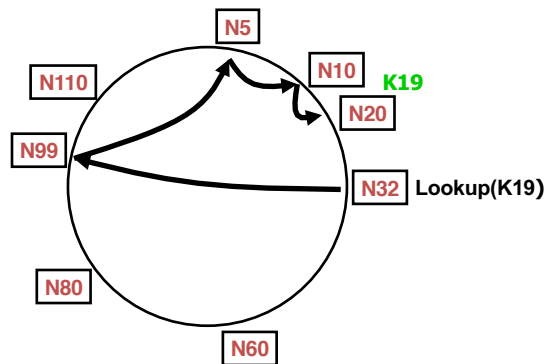
call **Lookup**(key-id) on node n //next hop

else

return my successor //done

30

Lookups Take $O(\log N)$ Hops



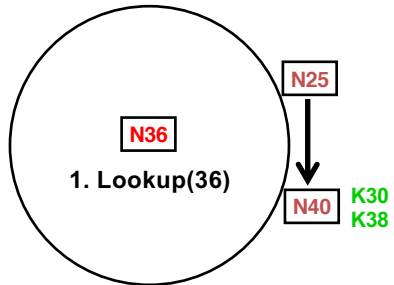
31

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**

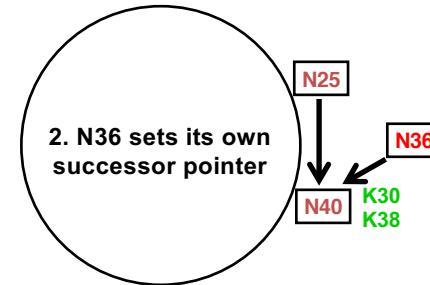
32

Joining: Linked list insert



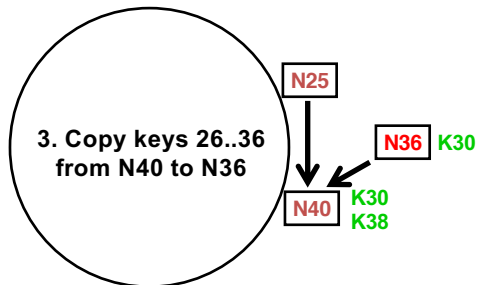
33

Join (2)



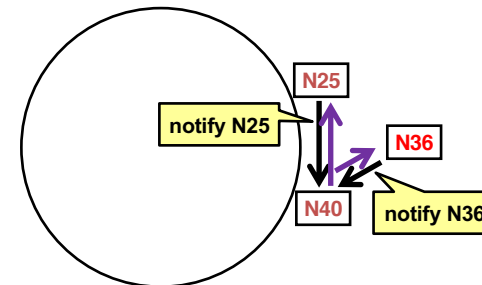
34

Join (3)



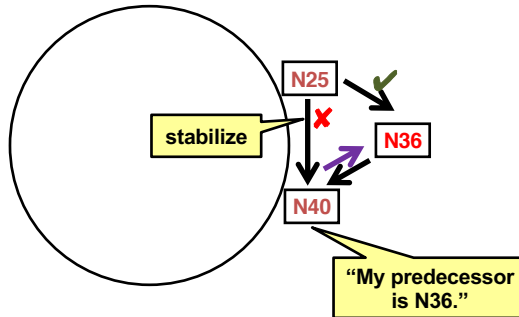
35

Notify messages maintain predecessors



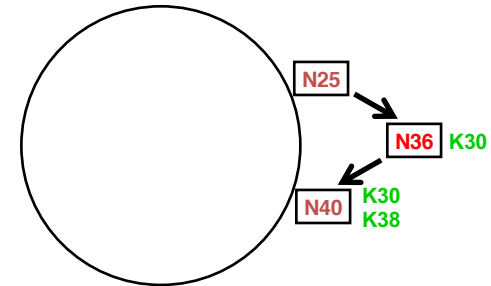
36

Stabilize message fixes successor



37

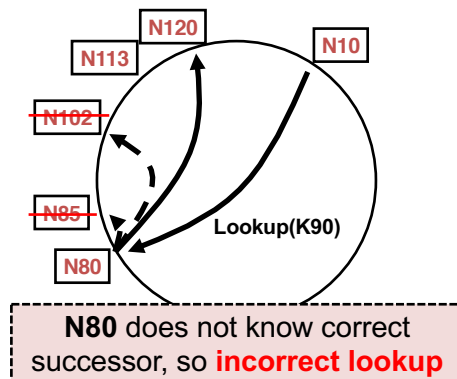
Joining: Summary



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

38

Failures may cause incorrect lookup



39

Successor lists

- Each node stores a **list** of its r **immediate successors**
 - After failure, will know first live successor
 - **Correct successors** guarantee **correct lookups**
 - Guarantee is with some probability

40

Choosing successor list length

- Assume **one half** of the nodes **fail**
- $P(\text{successor list all dead}) = (1/2)^r$
 - *i.e.*, $P(\text{this node breaks the Chord ring})$
 - Depends on independent failure
- Successor list of **size $r = O(\log N)$** makes this probability $1/N$: low for large N

41

Lookup with fault tolerance

```
Lookup(key-id)
  look in local finger table and successor-list
  for highest n: my-id < n < key-id
  if n exists
    call Lookup(key-id) on node n //next hop
    if call failed,
      remove n from finger table and/or
      successor list
      return Lookup(key-id)
  else
    return my successor //done
```

42

Today

1. Peer-to-Peer Systems
2. Distributed Hash Tables
3. **The Chord Lookup Service**
 - Basic design
 - **Integration with DHash DHT, performance**

43

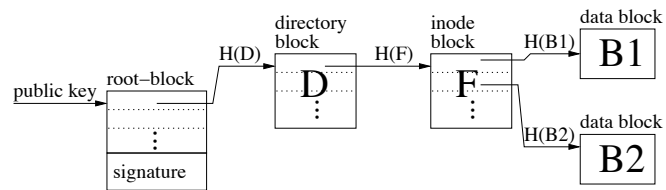
The DHash DHT

- Builds key/value storage on Chord
- **Replicates** blocks for availability
 - Stores k **replicas** at the k **successors** after the block on the Chord ring
- **Caches** blocks for load balancing
 - **Client** sends **copy of block** to each of the servers it contacted along the **lookup path**
- **Authenticates** block contents

44

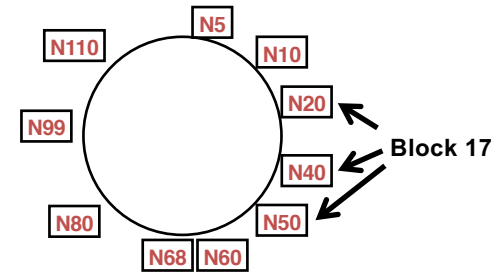
DHash data authentication

- Two types of DHash blocks:
 - Content-hash:** key = SHA-1(data)
 - Public-key:** key is a cryptographic public key, data are signed by corresponding private key
- Chord File System example:



45

DHash replicates blocks at r successors



- Replicas are **easy to find** if successor fails
- Hashed node IDs ensure **independent failure**

46

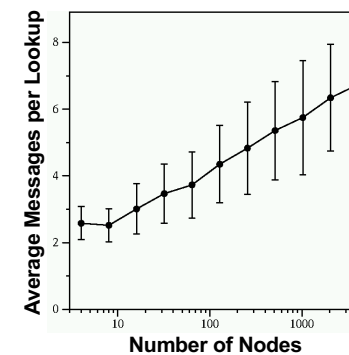
Experimental overview

- Quick lookup** in large systems
- Low **variation** in lookup costs
- Robust** despite **massive failure**

Goal: Experimentally confirm theoretical results

47

Chord lookup cost is $O(\log N)$



Constant is 1/2

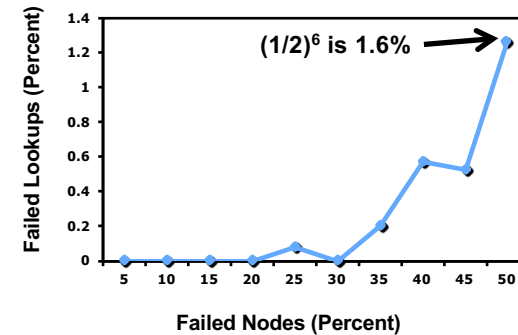
48

Failure experiment setup

- Start **1,000 Chord servers**
 - Each server's **successor list** has 20 entries
 - Wait until they **stabilize**
- Insert 1,000 key/value pairs
 - **Five replicas** of each
- **Stop X%** of the servers, immediately make 1,000 lookups

49

Massive failures have little impact



50

Today

1. Peer-to-Peer Systems
2. Distributed Hash Tables
3. The Chord Lookup Service
 - Basic design
 - Integration with *DHash* DHT, performance
4. **Concluding thoughts on DHT, P2P**

51

DHTs: Impact

- Original DHTs (CAN, Chord, Kademia, Pastry, Tapestry) proposed in 2001-02
- Following 5-6 years saw proliferation of DHT-based applications:
 - Filesystems (e.g., CFS, Ivy, OceanStore, Pond, PAST)
 - Naming systems (e.g., SFR, Beehive)
 - DB query processing [PIER, Wisc]
 - Content distribution systems (e.g., Coral)
 - distributed databases (e.g., PIER)

52

Why don't all services use P2P?

1. **High latency and limited bandwidth** between peers (*cf.* between server cluster in datacenter)
2. User computers are **less reliable** than managed servers
3. **Lack of trust** in peers' correct behavior
 - Securing DHT routing hard, unsolved in practice

53

DHTs in retrospective

- Seem promising for finding data in large P2P systems
- Decentralization seems good for load, fault tolerance

- **But:** the **security problems** are difficult
- **But:** **churn** is a problem, particularly if $\log(n)$ is big

- So DHTs have not had the impact that many hoped for

54

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

55

Wednesday topic:

Eventual Consistency

Pre-reading: Bayou paper (on website)

11:59 PM Wednesday:

Assignment 2 Deadline

56