CS162
Operating Systems and Systems Programming
Lecture 25

Capstone: P2P Systems, Review

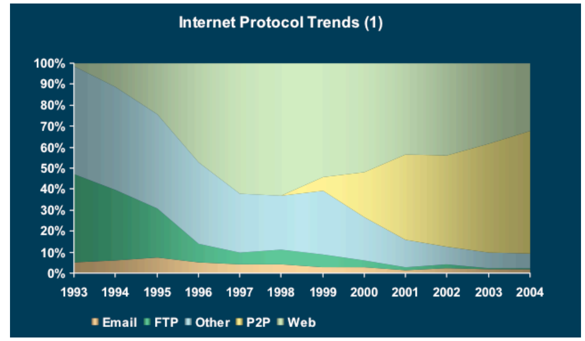
May 1, 2013

Anthony D. Joseph

http://inst.eecs.berkeley.edu/~cs162

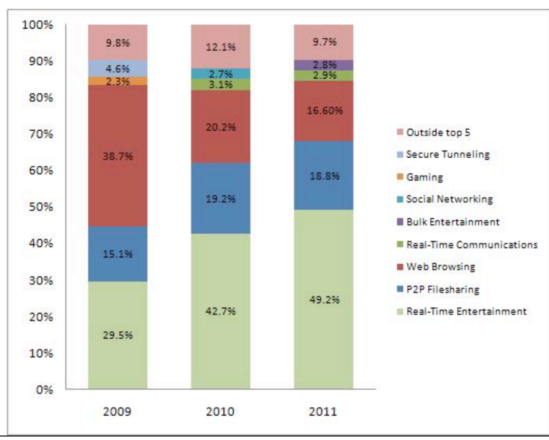
P2P Traffic

- 2004: some Internet Service Providers (ISPs) claimed that over 50% of their traffic was peer-to-peer traffic



P2P Traffic

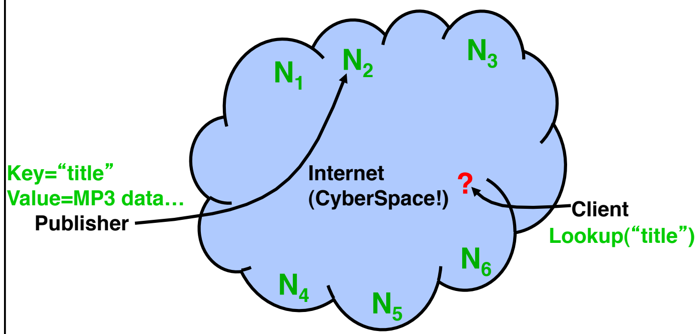
- Today, around 18-20% (North America)
- Big chunk now is video entertainment (e.g., Netflix, iTunes)



Peer-to-Peer Systems

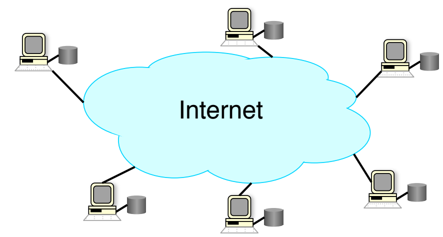
- What problem does P2P try to solve?
  - Provide highly scalable, cost effective (i.e., free!) services, e.g.,
    - » Content distribution (e.g., Bittorrent)
    - » Internet telephony (e.g., Skype)
    - » Video streaming (e.g., Octoshape)
    - » Computation (e.g., SETI@home)
- **Key idea:** leverage "free" resources of users (that use the service), e.g.,
  - Network bandwidth
  - Storage
  - Computation

The Lookup Problem



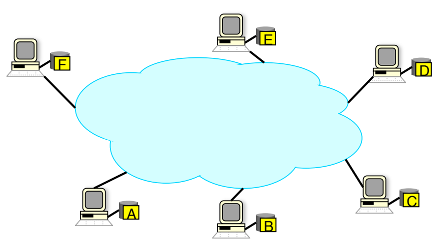
How Did it Start?

- A killer application: Napster (1999)
  - Free music over the Internet
- Use (home) user machines to store and distribute songs



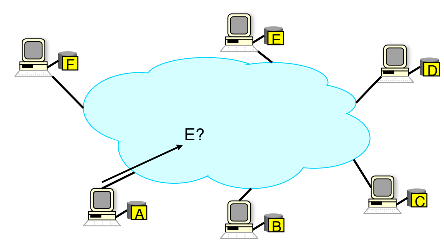
Model

- Each user stores a subset of files
- Each user has access (can download) files from all users in the system



Main Challenge

- Find a "good" node storing a specified file
- By "good" we mean:
  - Has correct content
  - Can get content from quickly
  - …



Other Challenges

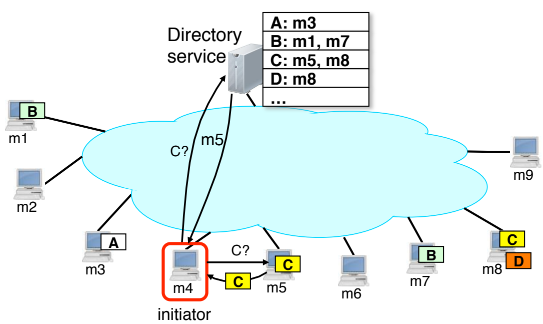
- **Scale**: up to hundred of thousands or millions of machines
- **Dynamicity**: machines can come and go at any time
- **Heterogeneity**: nodes with widely different resources and connectivity

Napster

- Implements a **centralized** lookup/directory service that maps files (songs) to machines currently in the system
- How to find a file (song)?
  - Query the lookup service → return a machine that stores the required file
    - » Ideally this is the closest/least-loaded machine
  - Download (ftp/http) the file
- Advantages:
  - Simplicity, easy to implement sophisticated search engines on top of a centralized lookup service
- Disadvantages:
  - Robustness, scalability (?)

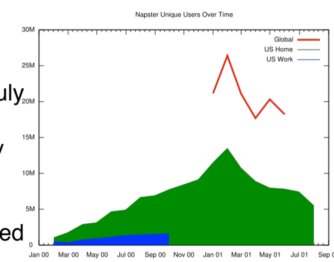
Napster: Example

1) A client (initiator) contacts directory service to get file "C"
2) Directory service returns a (possible) close by and lightly loaded peer (m5) storing "C"
3) Client contacts directly m5 to get file "C"



The Rise and Fall of Napster

- Founded by Shawn Fanning, John Fanning, and Sean Parker
- Operated between June 1999 and July 2001
  - More than 26 million users (February 2001)
- Several high profile songs were leaked before being released:
  - Metallica's "I Disappear" demo song
  - Madonna's "Music" single
- But, also helped made some bands successful (e.g., Radiohead, Dispatch)
- (Reemerged as music store in 2008)

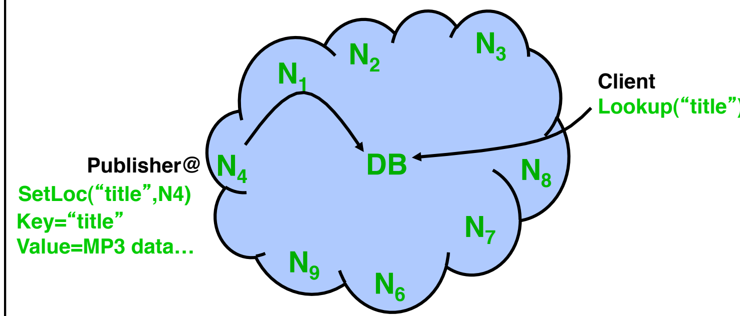


(Source: http://en.wikipedia.org/wiki/File:Napster_Unique_Users.svg)

The Aftermath

- **"Recording Industry Association of America (RIAA) Sues Music Startup Napster for $20 Billion" –** *December 1999*
- **"Napster ordered to remove copyrighted material"** **–** *March 2001*
- **Main legal argument:**
  - *Napster owns the lookup service, so it is directly responsible for disseminating copyrighted material*

Summary: Centralized Lookup (Napster)



Simple, but O(*N*) state and a single point of failure

Gnutella (2000)

- What problem does it try to solve?
  - Get around the legal vulnerabilities by getting rid of the *centralized* directory service
- Main idea: Flood the request to peers in the system to find file

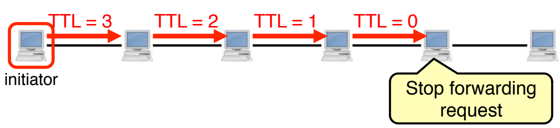
Gnutella (2000)

- How does request flooding work?
  - Send request to all neighbors
  - Neighbors recursively send request to their neighbors
  - Eventually a machine that has the file receives the request, and it sends back the answer
- Advantages:
  - Totally decentralized, highly robust
- Disadvantages:
  - Not scalable; the entire network can be swamped with requests (to alleviate this problem, each request has a TTL)
    - » TTL (Time to Leave): request dropped when TTL reaches 0

Gnutella: Time To Live (TTL)

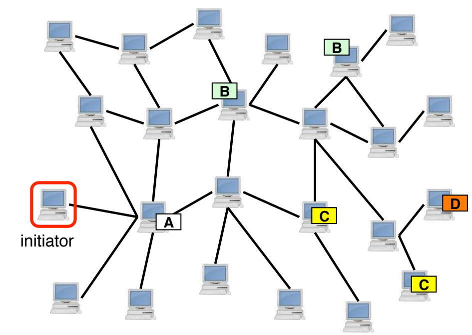
- When the client (initiator) sends a request, it associates a TTL with the request
- When a node forwards the request it decrements the TTL
- When TTL reaches 0, the request is no longer forwarded
- Typically, Gnutella uses TTL = 7

- Example: TTL = 3



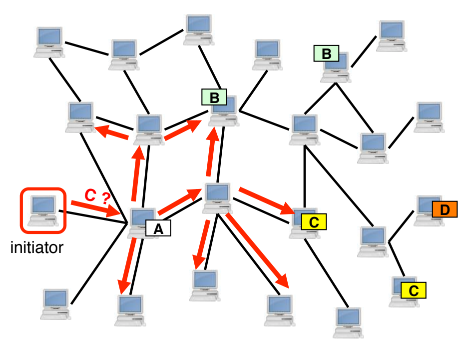
Gnutella: Example

- Assume a client (initiator) asks for file "C"
- Assume TTL=2



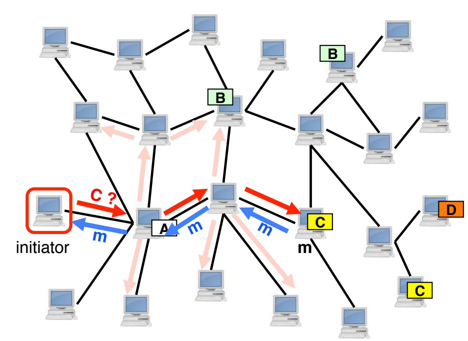
Gnutella: Example

- Initiator send request to its neighbor(s)…
- … which recursively forward the request to their neighbors
- At the 3rd hop request is dropped



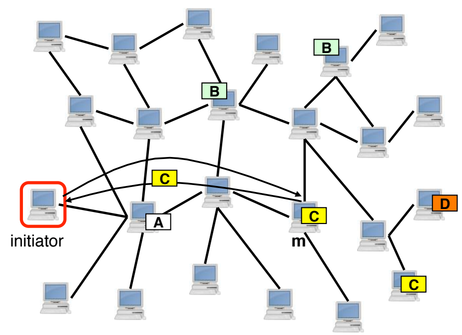
Gnutella: Example

- If node has the requested file it sends a reply back
  - along the reverse path of the request, or
  - directly to initiator



Gnutella: Example

- Initiator request file "C" from node "m"
  - Initiator may pick one of several machines if receive multiple replies

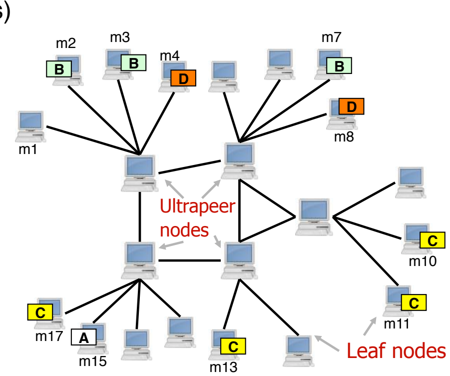


Two-Level Hierarchy

- What problem does it try to solve?
  - Inefficient search
  - Heterogeneous nodes
  - Dynamicity
- Main idea: organize the p2p system in a two level hierarchy
  - Flooding happens only at the top level

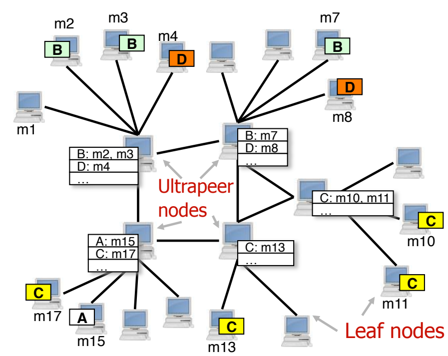
Two-Level Hierarchy

- KaZaa, and subsequent versions of Gnutella
- Leaf nodes are connected to a small number of ultrapeers (supernodes)



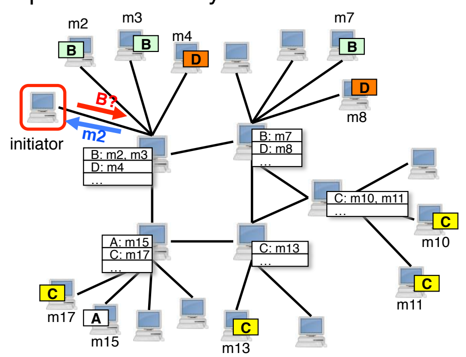
Two-Level Hierarchy

- Each ultra-peer builds a directory for the content stored at its peers



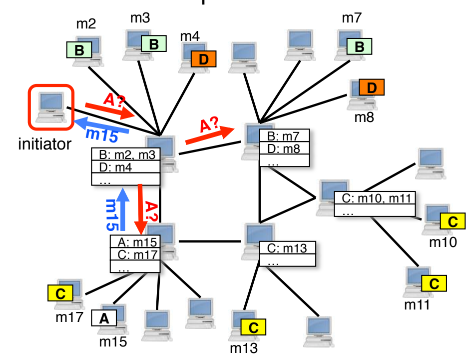
Gnutella: Example

- Query: A leaf sends query to its ultrapeers
- If ultrapeer has requested content in its directory, the ultrapeer replies immediately

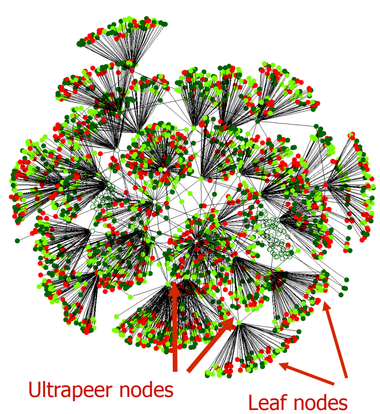


Gnutella: Example

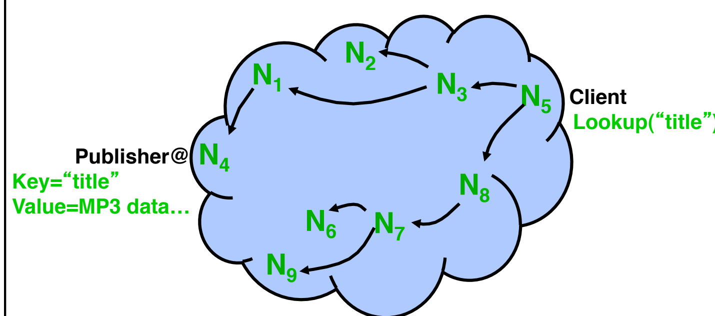
- Query: A leaf sends query to its ultrapeers
- If ultrapeer doesn't have content in its directory, the ultrapeer floods other ultrapeers



Example: Oct 2003 Crawl on Gnutella

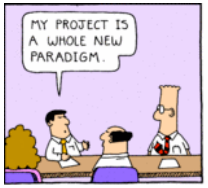


Summary: Flooded queries (Gnutella)



Robust, but worst case O(*N*) messages per lookup
Two-level hierarchy helps, but only reduces *N*

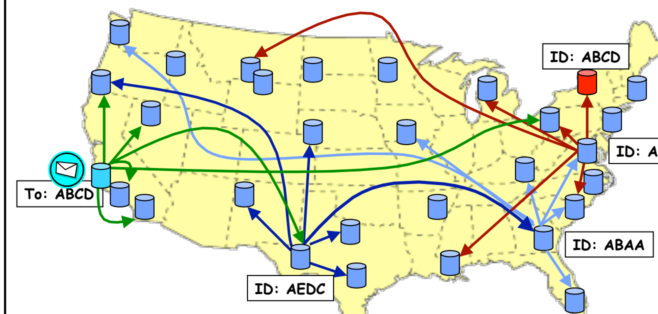
Research Community View of Peer-to-Peer



- Old View:
  - A bunch of flakey high-school students stealing music
- New View:
  - A philosophy of systems design at extreme scale
  - Probabilistic design when it is appropriate
  - New techniques aimed at unreliable components
  - A rethinking (and recasting) of distributed algorithms
  - Use of Physical, Biological, and Game-Theoretic techniques to achieve guarantees

Structured Peer-to-Peer Overlays

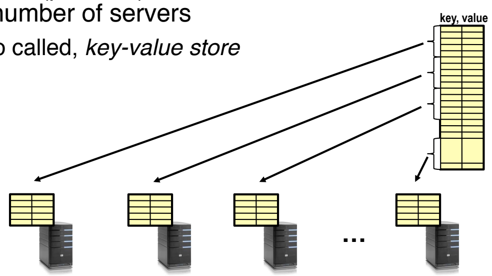
- Highly scalable protocol for routing to a name or node
  - "rule-based" incremental routing towards destination ID
  - each node has small set of outgoing routes, e.g. prefix routing
  - *log (n)* neighbors per node, *log (n)* hops from any *X* to *Y*



- 2001: Tapestry, CAN, Chord, Pastry; many others followed

Recall: Distributed Hash Tables (DHTs)

- Distribute (partition) a hash table data structure across a large number of servers
  - Also called, *key-value store*



- Two operations
  - **put**(key, data); // insert "data" identified by "key"
  - data = **get**(key); // get data associated to "key"

Recall: DHTs (cont'd)

- **Lookup service**: given a key (ID), map it to node n
  n = **lookup**(key);
- Can invoke **put()** and **get()** at any node m

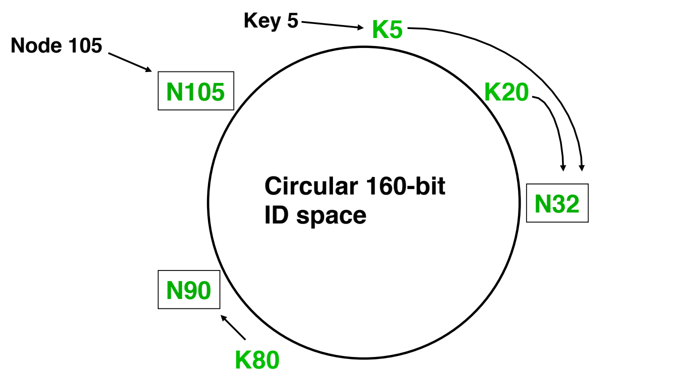
```
m.put(key, data) {
    n = lookup(key); // get node "n" mapping "key"
    n.store(key, data); // store data at node "n"
}

data = m.get(key) {
    n = lookup(key); // get node "n" storing data associated to "key"
    return n.retrieve(key); // get data stored at "n" associated to
    "key"
}
```

Chord Lookup Service

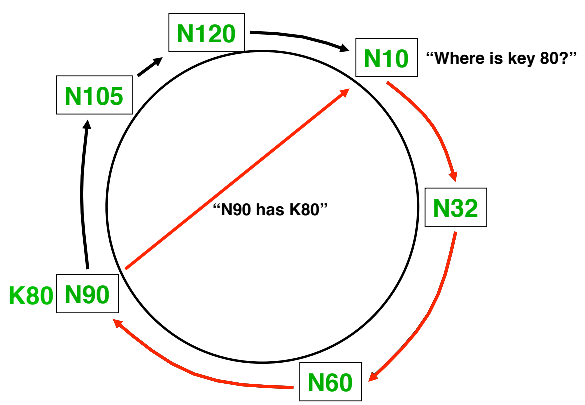
- Associate to each ***node and item*** a unique *key* in the *uni-dimensional* space 0..2^m-1
  - Partition this space across N machines with IDs from 0..2^m-1
  - Each **key** is mapped to the node with the smallest ID larger than the **key** (consistent hashing)
- Design approach: decouple **correctness** from **efficiency**
- Properties
  - Routing table size (# of other nodes a node needs to know about) is O(log(*N*)), where *N* is the number of nodes
  - Guarantees that a file is found in O(log(*N*)) steps

Consistent hashing [Karger 97]



A key is stored at its successor: node with next higher ID

Basic lookup

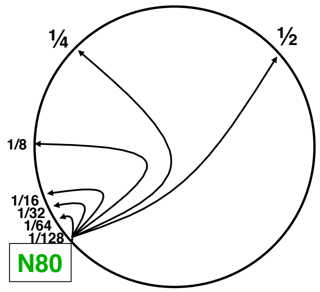


Simple lookup algorithm

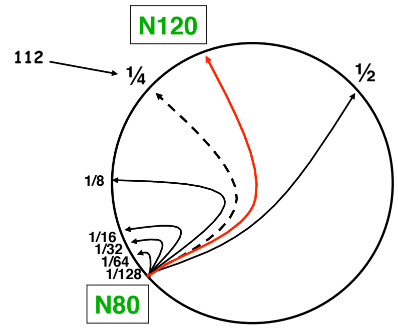
```
Lookup(my-id, key-id)
  n = my successor
  if my-id < n < key-id
     call Lookup(id) on node n   // next hop
  else
     return my successor         // done
```

- Correctness depends only on successors

"Finger Table" Allows log(N)-time Lookups



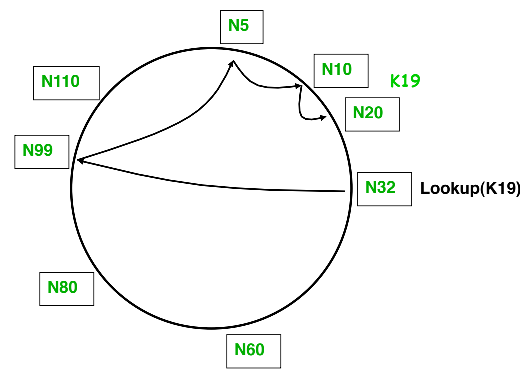
Finger *i* Points to Successor of $n+2^i$



Lookup with Fingers

```
Lookup(my-id, key-id)
  look in local finger table for
      highest node n s.t. my-id < n < key-id
  if n exists
      call Lookup(id) on node n      // next hop
  else
      return my successor            // done
```


Lookups take O(*log(N)*) hops

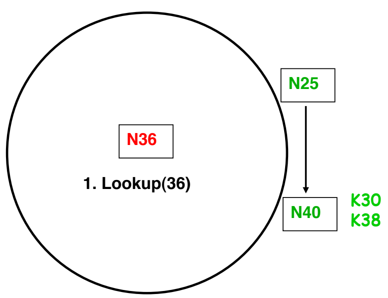


Announcements

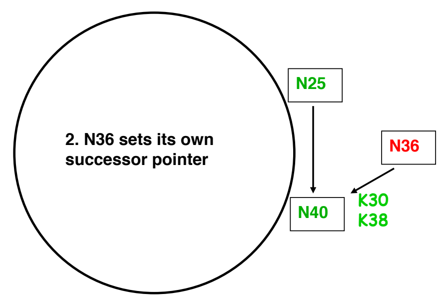
- Project 4 Code due next week Thu 4/9 by 11:59pm
  - Final design doc and group evals due Fri 4/10 by 11:59pm
- My RRR week office hours
  - Monday 5/6, 1-2pm and Wednesday 5/8, 2-3pm in 449 Soda
- Final Exam: Friday 5/17, 8-11am in 1 Pimentel
  - Review: Monday 5/6, 2-5pm in 100 Lewis Hall
  - All material from the course: lectures, sections, projects, hand outs
    - » With more focus on second half (~30%/~70%), but you are still responsible for all the material
  - Two sheets of notes, both sides
  - Dumb calculator allowed

5min Break

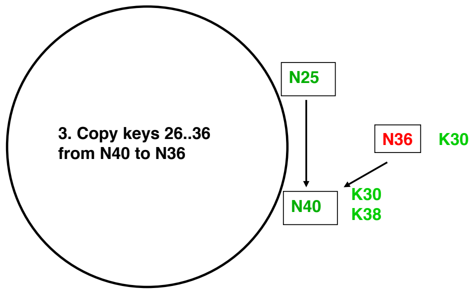
Joining: Linked List Insert of Node N36



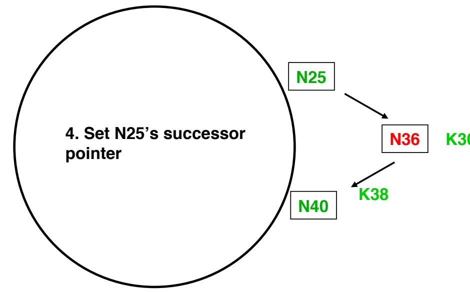
Join (2)



Join (3)

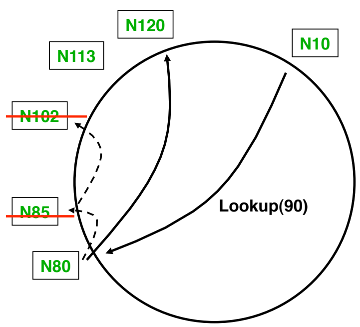


Join (4)



Update finger pointers in the background
Correct successors produce correct lookups

Challenge: Failures Might Cause Incorrect Lookup



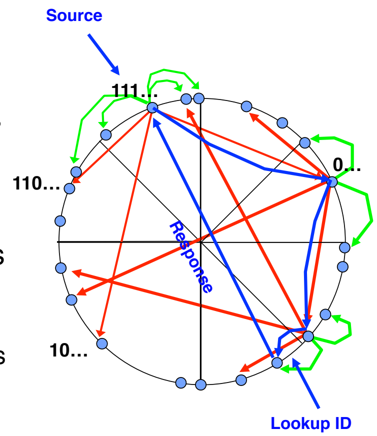
N80 doesn't know correct successor, so incorrect lookup

Solution: successor lists

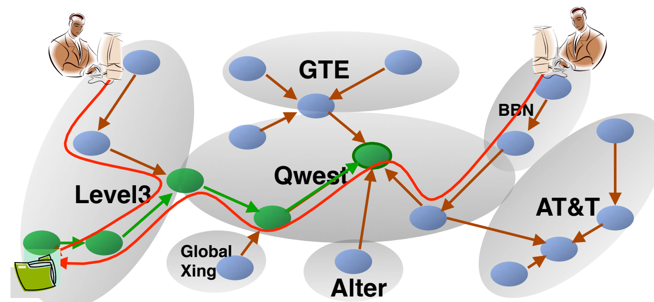
- Each node knows r immediate successors
  - After failure, will know first live successor
  - Correct successors guarantee correct lookups *with some probability*
- Many systems use a "leaf set"
  - The set of nodes around the "root" node that can handle all of the data/queries that the root nodes might handle
- When node fails:
  - Leaf set can handle queries for dead node
  - Leaf set queried to recreate missing data
  - Leaf set used to reconstruct new leaf set

Lookup with Leaf Set

- Assign IDs to nodes
  – Map hash values to node with closest ID
- Leaf set is successors and predecessors
  – All that's needed for correctness
- Routing table matches successively longer prefixes
  – Allows efficient lookups

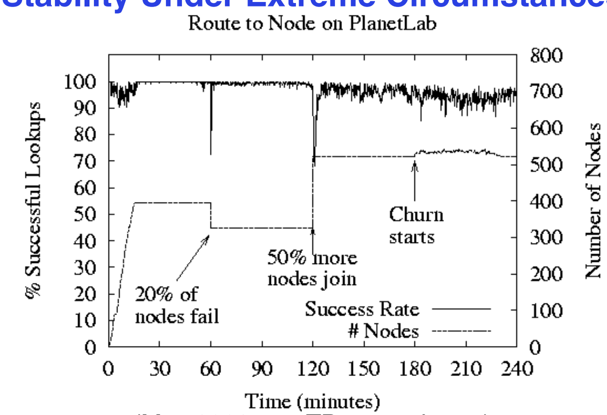


Decentralized Object Location & Routing



- Server "publishes" object in infrastructure like a yellow-pages
  – Object's root node determined by Hash(object name)
  – Overlay distributes location pointers to *log (n)* nodes towards Root
- Clients route messages towards object's root node
  – Message routes towards root, redirect when location pointer found

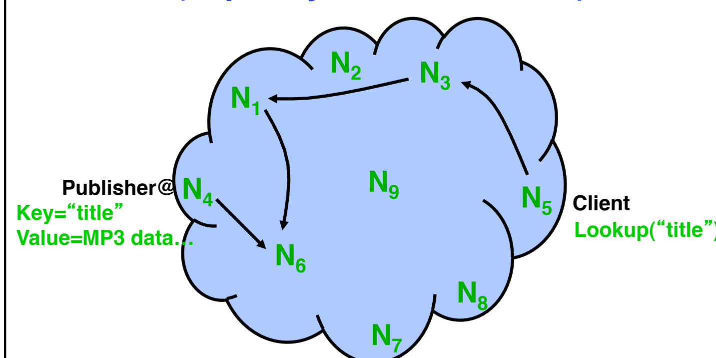
Stability Under Extreme Circumstances



(May 2003: 1.5 TB over 4 hours)

DOLR Model generalizes to many simultaneous apps

Summary: Routed Queries (Tapestry, Chord, CAN, ...)



Can be O(log *N*) messages per lookup (or even O(1))
Potentially complex routing state and maintenance.

P2P Summary

- The key challenge of building wide area P2P systems is a scalable and robust directory service
- Solutions
  - Naptser: centralized location service
  - Gnutella: broadcast-based decentralized location service
  - CAN, Chord, Tapestry, Pastry: intelligent-routing decentralized solution
    - » Guarantee correctness

CS162: Summary

- OS functions:
  - Manage system resources
  - Provide services: storage, networking, …
  - Provide a VM abstraction to processes/users: give illusion to each process/user that is using a dedicated machine
- Challenges
  - Virtualize system resources
    - » Virtual Memory (VM): address translation, demand paging
    - » CPU scheduling
  - Arbitrate access to resources and data
    - » Concurrency control, synchronization
    - » Deadlock prevention, detection

Key Concept: Synchronization

- Allow multiple processes to share data
- Why it is challenging?
  - Want high utilization: need fine grain sharing
  - Avoid non-determinism
- Many primitives/mechanisms
  - Locks, Semaphores, Monitors (condition variables)
- Many examples:
  - Producer-consumer (bounded buffer, flow control)
  - Reader/Writer problem
  - Transactions

Most likely concept you'll use in your job

OS is Evolving

The Network is the Computer™

- Vast majority of apps are distributed today
  - E.g., mail, Facebook/Twitter, Skype, Google docs, …
- More and more OSes integrate remote services
  - E.g., iOS (iCloud), Chrome OS (Google Drive), Windows 8 (SkyDrive)
- One example in this class (project 4): reliable and consistent key-value store
  - Give you taste of challenges of building a distributed system
  - Why hard?
    - » Nodes can fail: may lose data, render service unavailable
    - » Network can get congested or partitioned: slow/unavailable service
    - » Scale: a p2p network can consists of million of nodes

Conclusion

- OS inherently covers many topics
  - More and more services migrate into OS (e.g., networking, search)
- If you want to focus on some of these topics
  - Database class (CS 186)
  - Networking class (EE 122)
  - Security class (CS 161)
  - Software engineering class (CS 169)
- If you want to focus on OS
  - Advanced OS class, CS 194 (John Kubiatowicz), Spring 2014
  - Undergraduate research projects in the AMP Lab
    - » Akaros and Mesos projects