# CS162
# Operating Systems and Systems Programming
# Lecture 25

## P2P Systems and Review

November 28, 2012

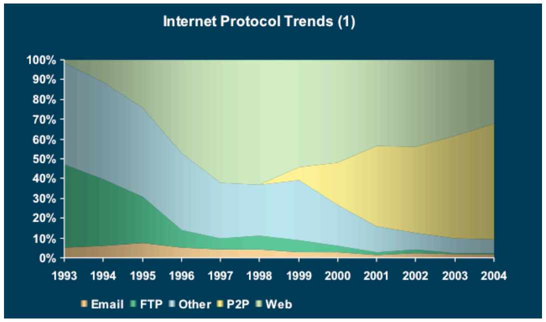
Ion Stoica

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

---

## P2P Traffic

- 2004: some Internet Service Providers (ISPs) claimed > 50% was p2p traffic

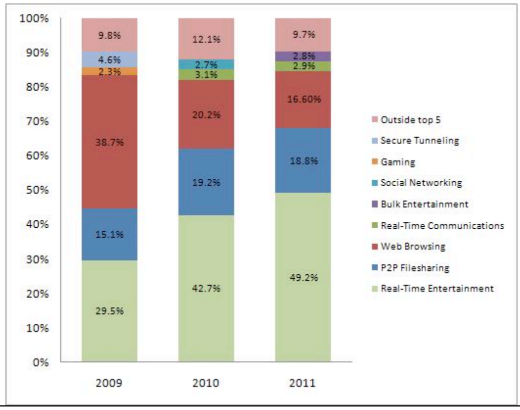





---

## P2P Traffic

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






---

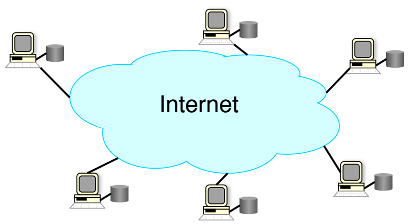
## Peer-to-Peer Systems

- What problem does it 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

## How Did it Start?

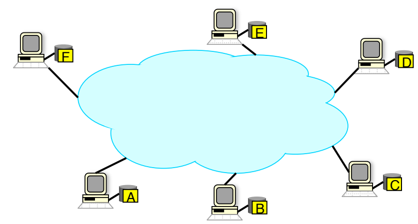
- A killer application: Napster (1999)
  - Free music over the Internet
- Use (home) user machines to store and distribute songs






## Model

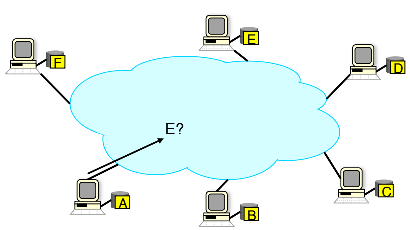
- Each user stores a subset of files
- Each user has access (can download) files from all users in the system






## Main Challenge

- Find a “good” node storing a specified file
- By “good” we mean:
  - Has correct content
  - Can get content fast
  - …






## 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

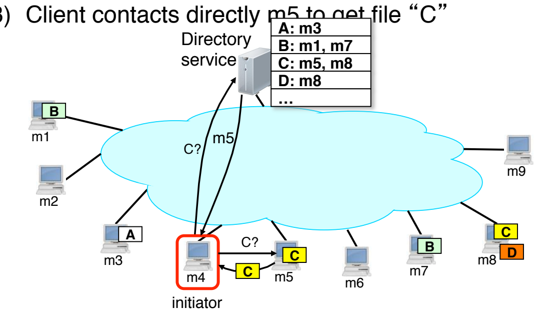
## Napster

- 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

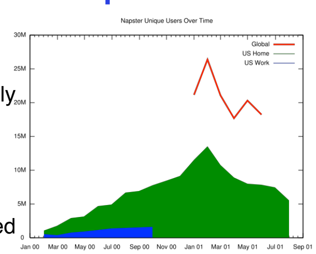
1) A client (initiator) contacts directory service to get file "C"
2) Directory service returns a (possible) close by and lightly loaded peer (m5) storing "C"
3) Client contacts directly m5 to get file "C"






## 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**

## 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



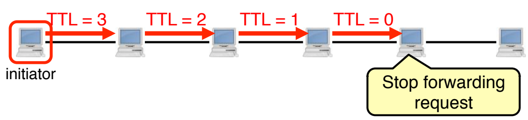
## Gnutella (2000)

- 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)

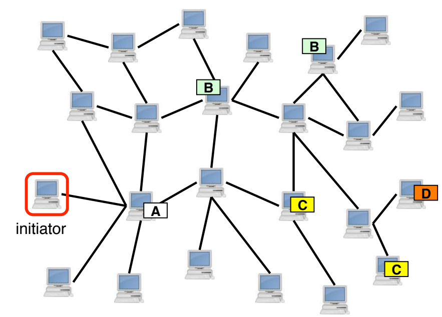
- When the client (initiator) sends a request, it associates a TTL with the request
- When a node forwards the request it decrements the TTL
- When TTL reaches 0, the request is no longer forwarded
- Typically, Gnutella uses TTL = 7
- Example: TTL = 3





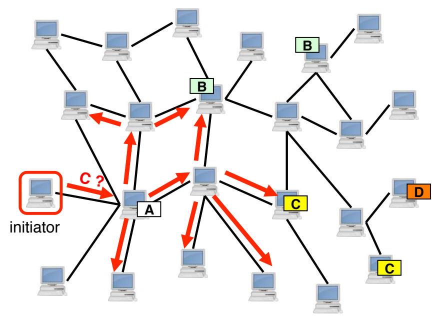

## Gnutella: Example

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



## Gnutella: Example

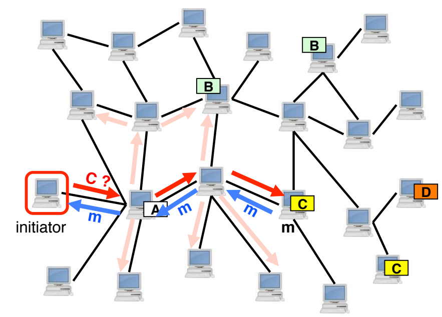
- Initiator send request to its neighbor(s)…
- … which recursively forward the request to their neighbors
- At the 3rd hop request is dropped






## Gnutella: Example

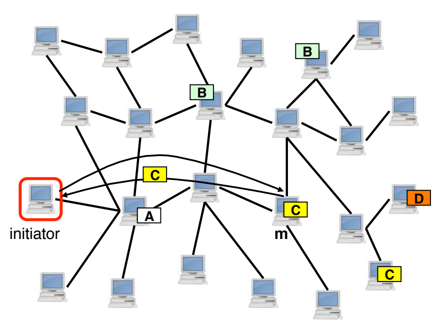
- If node has the requested file it sends a reply back
  – along the reverse path of the request, or
  – directly to initiator






## Gnutella: Example

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





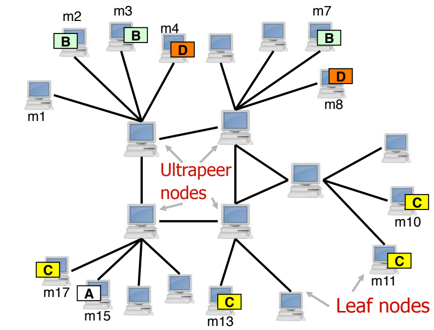

## Two-Level Hierarchy

- What problem does it try to solve?
  – Inefficient search

- Main idea: organize the p2p system in a two level hierarchy
  – Flooding happens only at the top level

## Two-Level Hierarchy

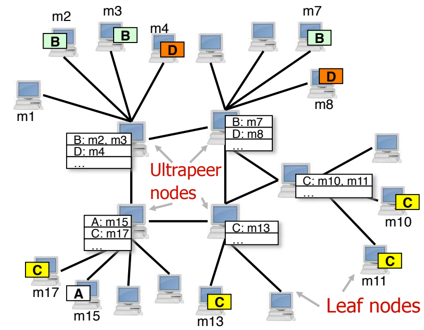
- KaZaa, subsequent versions of Gnutella
- Leaf nodes are connected to a small number of ultrapeers (supernodes)






## Two-Level Hierarchy

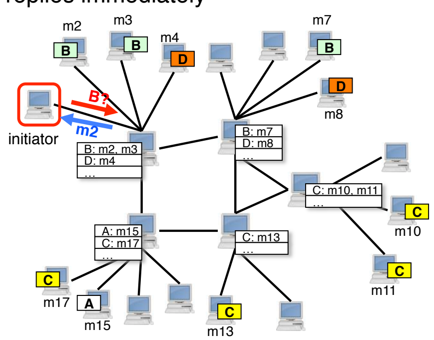
- Each ultra-peer builds a director for the content stored at its peers






## Gnutella: Example

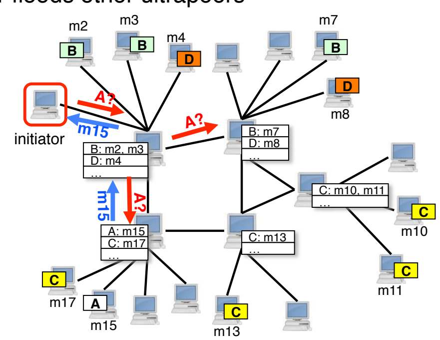
- Query: A leaf sends query to its ultrapeers
- If ultrapeer has requested content in its directory, the ultrapeer replies immediately



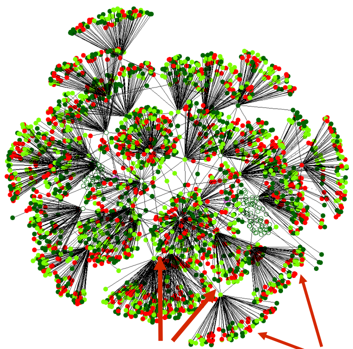



## Gnutella: Example

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



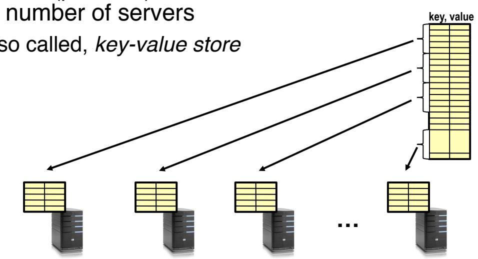
## Example: Oct 2003 Crawl on Gnutella






## 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)

- Project 4: **puts** and **gets** are serialized through a **master**
  – Master knows all nodes (slaves) in the system
  – Master maintains mapping between keys and nodes
  – Simple but doesn't scale for large, dynamic p2p systems
- Next: an **efficient decentralized** lookup protocol
  – Many proposals: CAN, Chord, Pastry, Tapestry, Kademlia, …
  – Used in practice, e.g., eDonkey (based on Kademlia)



## 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"
}
```

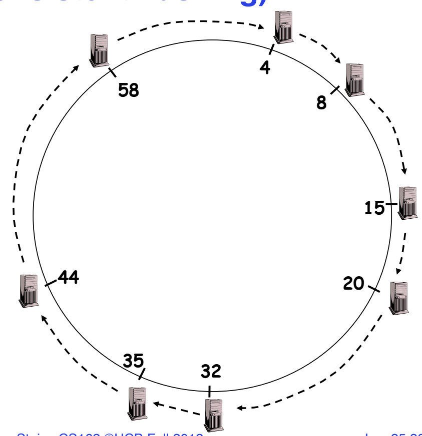
## Chord Lookup Service

- Associate to each node and item a unique *key* in an *uni-*dimensional space 0..$2^m$-1
  - Partition this space across N machines
  - 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



## Identifier to Node Mapping Example (Consistent hashing)

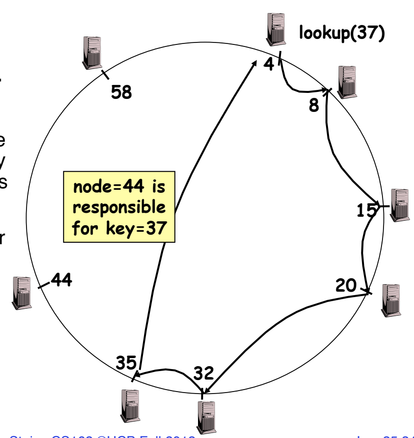
- m = 6; ID range 0..63
- Node 8 maps [5,8]
- Node 15 maps [9,15]
- Node 20 maps [16, 20]
- …
- Node 4 maps [59, 4]
- Each node maintains a pointer to its successor






## Lookup

- Each node maintains pointer to its **successor**
- Route lookup(key) to the node responsible for key using successor pointers
- E.g., node=4 lookups for node responsible for key=37






## Stabilization Procedure

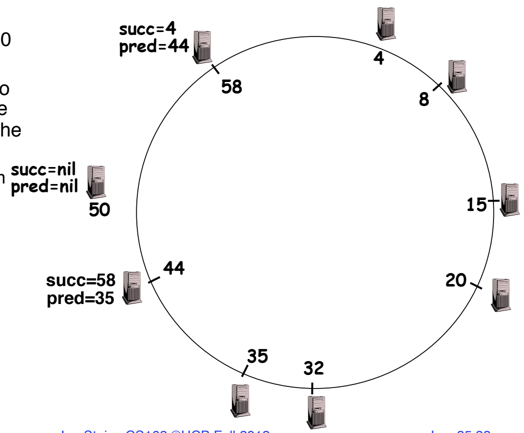
- Periodic operation performed by each node n to maintain its successor when new nodes join the system

```
n.stabilize()
  x = succ.pred;
  if (x ∈ (n, succ))
     succ = x;     // if x better successor, update
  succ.notify(n); // n tells successor about itself

n.notify(n')
  if (pred = nil or n' ∈ (pred, n))
     pred = n';     // if n' is better predecessor, update
```

## Joining Operation

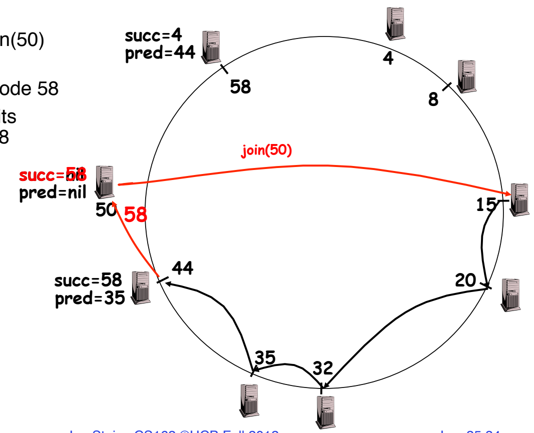
- Node with key=50 joins the ring
- Node 50 needs to know at least one node already in the system
  - Assume known node is 15






## Joining Operation

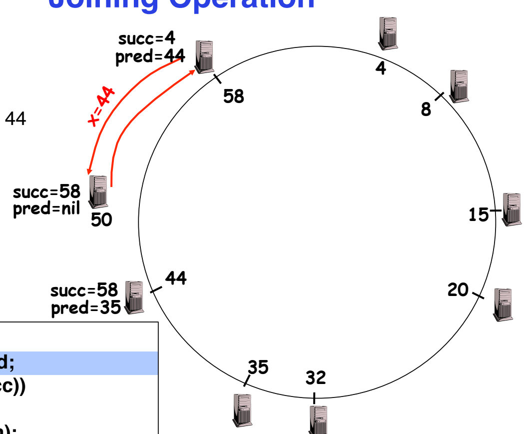
- n=50 sends join(50) to node 15
- n=44 returns node 58
- n=50 updates its successor to 58






## Joining Operation

- n=50 executes stabilize()
- n's successor (58) returns x = 44

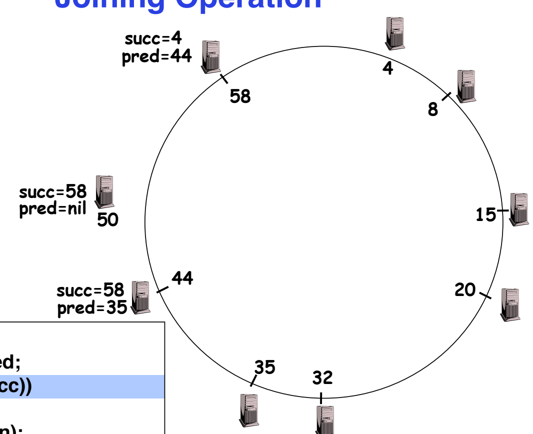



```
n.stabilize()
  x = succ.pred;
  if (x∈(n, succ))
     succ = x;
  succ.notify(n);
```



## Joining Operation

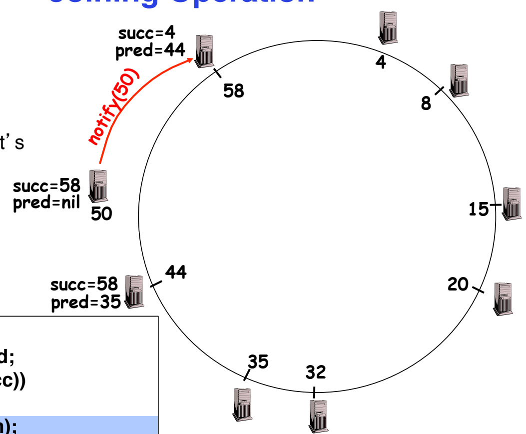
- n=50 executes stabilize()
  - x = 44
  - succ = 58




```
n.stabilize()
  x = succ.pred;
  if (x∈(n, succ))
     succ = x;
  succ.notify(n);
```

## Joining Operation

- n=50 executes stabilize()
  - x = 44
  - succ = 58
- n=50 sends to it' s successor (58) notify(50)

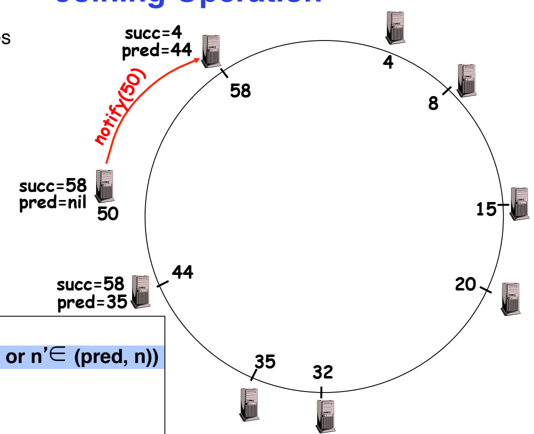



```
n.stabilize()
  x = succ.pred;
  if (x∈(n, succ))
    succ = x;
  succ.notify(n);
```



## Joining Operation

- n=58 processes notify(50)
  - pred = 44
  - n' = 50

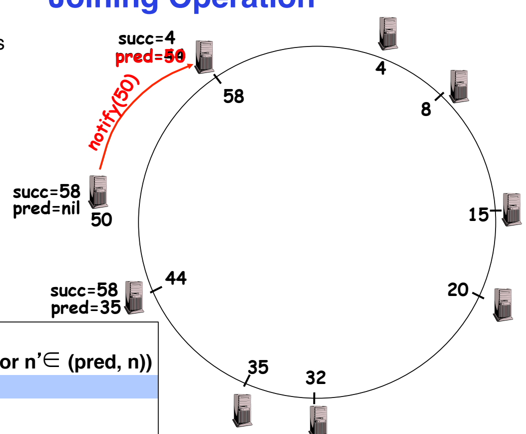



```
n.notify(n')
  if (pred = nil or n'∈ (pred, n))
    pred = n'
```



## Joining Operation

- n=58 processes notify(50)
  - pred = 44
  - n' = 50
- set pred = 50

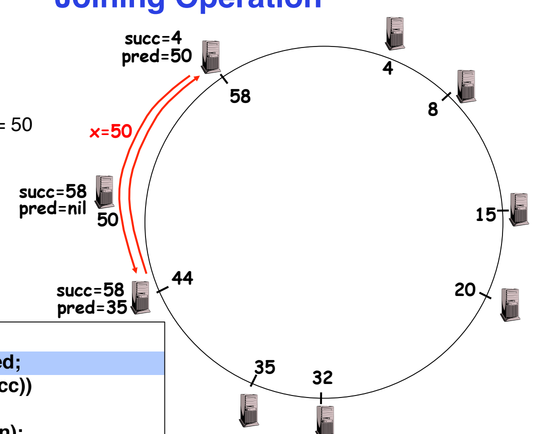



```
n.notify(n')
  if (pred = nil or n'∈ (pred, n))
    pred = n'
```



## Joining Operation

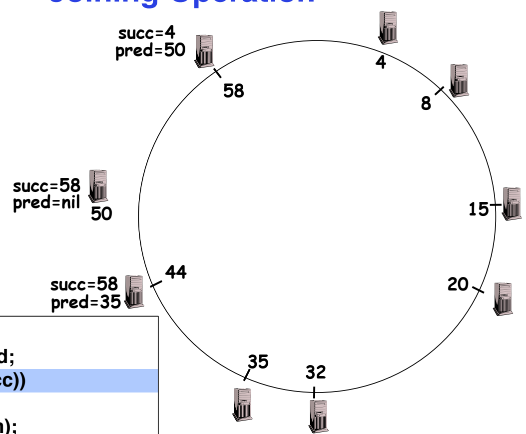
- n=44 runs stabilize()
- n' s successor (58) returns x = 50




```
n.stabilize()
  x = succ.pred;
  if (x∈(n, succ))
    succ = x;
  succ.notify(n);
```

## Joining Operation

- n=44 runs stabilize()
  - x = 50
  - succ = 58

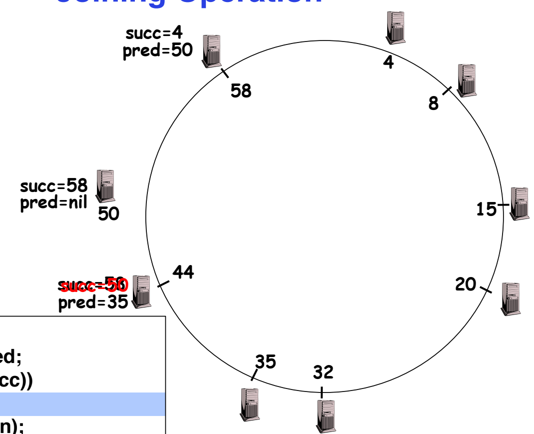



```
n.stabilize()
  x = succ.pred;
  if (x∈(n, succ))
     succ = x;
  succ.notify(n);
```



## Joining Operation

- n=44 runs stabilize()
  - x = 50
  - succ = 58
- n=44 sets succ=50

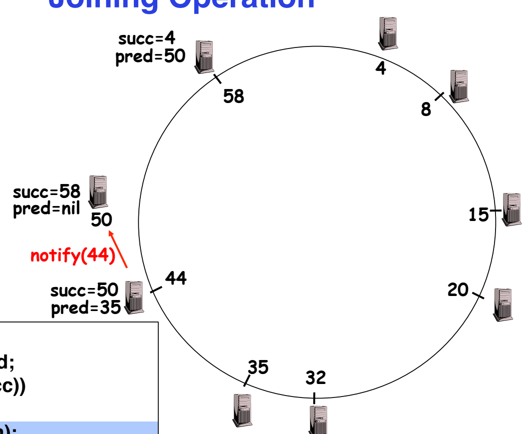



```
n.stabilize()
  x = succ.pred;
  if (x∈(n, succ))
     succ = x;
  succ.notify(n);
```



## Joining Operation

- n=44 runs stabilize()
- n=44 sends notify(44) to its successor

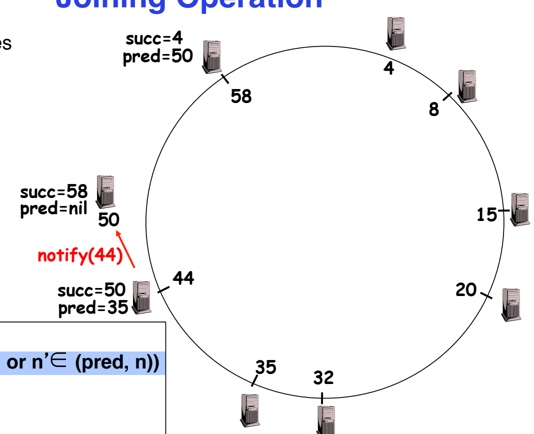



```
n.stabilize()
  x = succ.pred;
  if (x∈(n, succ))
     succ = x;
  succ.notify(n);
```



## Joining Operation

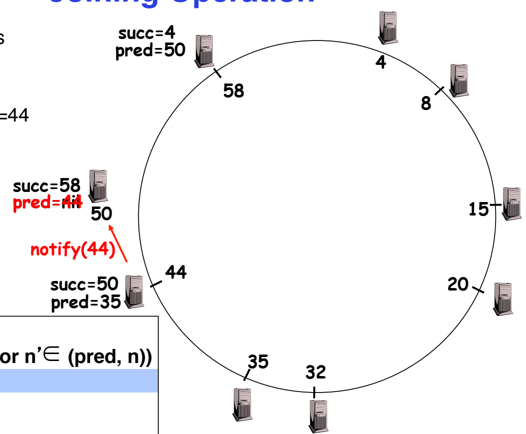
- n=50 processes notify(44)
  - pred = nil




```
n.notify(n')
  if (pred = nil or n'∈ (pred, n))
     pred = n'
```

## Joining Operation

- n=50 processes notify(44)
  - pred = nil
- n=50 sets pred=44

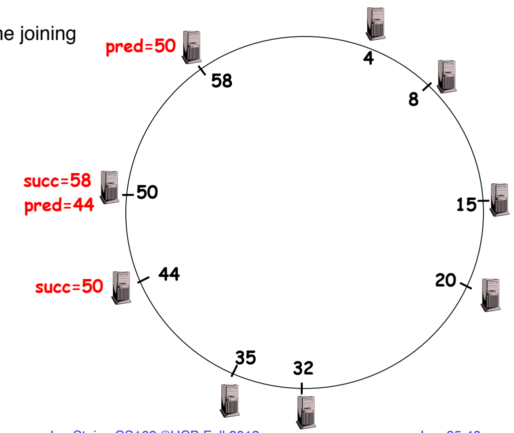



```
n.notify(n')
  if (pred = nil or n'∈ (pred, n))
    pred = n'
```



## Joining Operation (cont' d)

- This completes the joining operation!



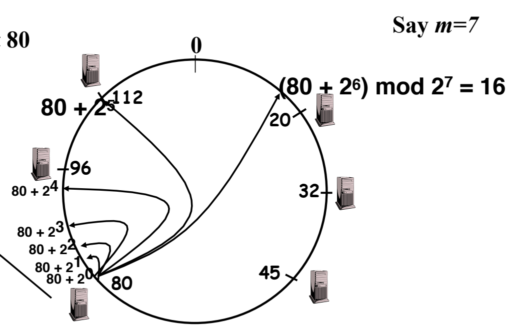



## Achieving Efficiency: *finger tables*

**Finger Table at 80**

| *i* | *ft[i]* |
|---|---|
| 0 | 96 |
| 1 | 96 |
| 2 | 96 |
| 3 | 96 |
| 4 | 96 |
| 5 | 112 |
| 6 | 20 |

Say *m=7*




*i*th entry at peer with id *n* is first peer with id >= $n + 2^i (\bmod 2^m)$



## Details

- Lookup complexity O(*log* N)
  - Every hop the distance to target is at least halved
- To improve robustness each node maintains the k (> 1) immediate successors instead of only one successor
- Successor S of a node N can send its K-1 successors to N during N' s stabilize() procedure

## Announcements

- Final code due: Thursday, Dec 6, 11:59pm
- Final design document and group evaluation due: Friday, Dec 7, 11:59pm
- Final exam: Thursday, Dec 13, 8-11am
  - Close books
  - Double face cheat sheet
  - Comprehensive, but greater focus on material since midterm (30% / 70%)
- Final Review: Wednesday, Dec 5, 6-9pm, 60 Evans Hall



## 5min Break



## 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)
  - Reade/Writer problem
  - Transactions

Most likely concept you'll use in your job



## OS is Evolving

Sun microsystems

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, Windows 8
- 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 an 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
  - New upper-level OS class, CS 194 (John Kubiatowicz), Spring 2013
  - Undergraduate research projects in the AMP Lab
    - » Akaros and Mesos projects