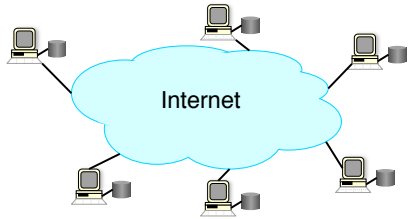
CS162
Operating Systems and Systems Programming
Lecture 24

Peer-to-Peer Networks

November 21, 2011
Anthony D. Joseph and Ion Stoica
http://inst.eecs.berkeley.edu/~cs162

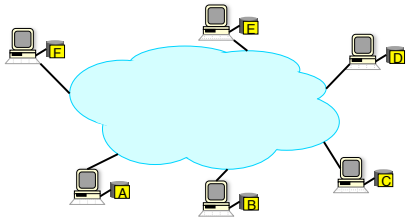
How Did it Start?

- A killer application: Napster (1999)
  - Free music over the Internet
- Key idea: share the storage *and* bandwidth of individual (home) users



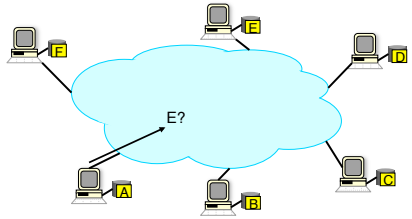
Model

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



Main Challenge

- Find nodes storing a specified file



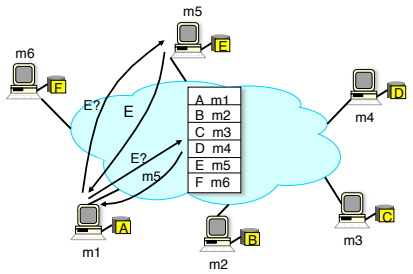
Other Challenges

- Scale: up to hundred of thousands or millions of machines
- Dynamicity: machines can come and go at any time

Napster

- Assume a **centralized** lookup/directory service that maps files (songs) to machines that are alive
- 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) the file
- Advantages:
  - Simplicity, easy to implement sophisticated search engines on top of a centralized lookup service
- Disadvantages:
  - Robustness, scalability (?)

Napster: Example



The Aftermath

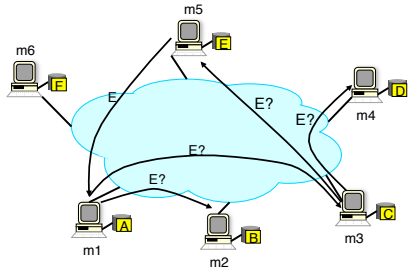
- **"Recording Industry Association of America (RIAA) Sues Music Startup Napster for $20 Billion" –** *December 1999*
- **"Napster ordered to remove copyrighted material" –** *March 2001*
- **Main legal argument:**
  - **Napster owns the lookup service, so it is directly responsible for disseminating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

Gnutella (2000)

- Distribute file location
- Idea: broadcast the request
- How to find a file?
  - Send request to all neighbors
  - Neighbors recursively multicast the request
  - 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)

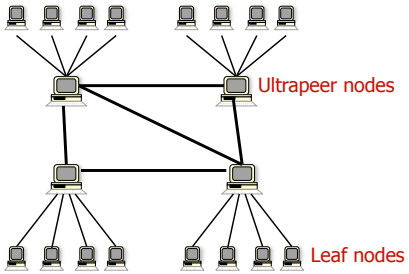
Gnutella: Example

- Assume: m1's neighbors are m2 and m3; m3's neighbors are m4 and m5;…



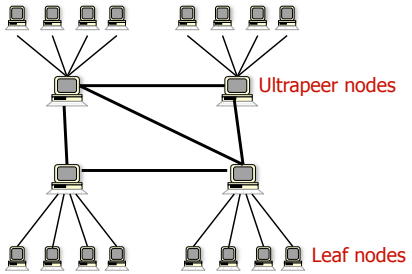
Two-Level Hierarchy

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

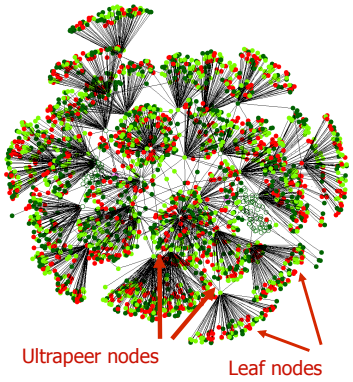


Two-Level Hierarchy

- Query
  - A leaf sends query to its ultrapeers
  - If ultrapeers don't know the answer, they flood the query to other ultrapeers

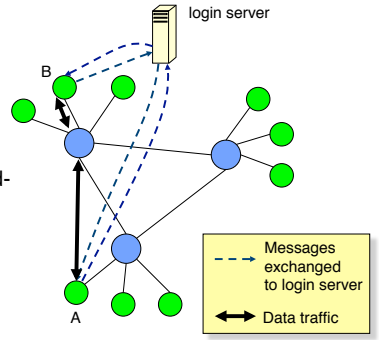


Example: Oct 2003 Crawl on Gnutella



Skype (2003)

- Peer-to-peer Internet Telephony
- Two-level hierarchy like KaZaa
  - Ultrapeers used to route traffic between NATed end-hosts…
  - … plus a login server to
    - » authenticate users
    - » ensure that names are unique across network



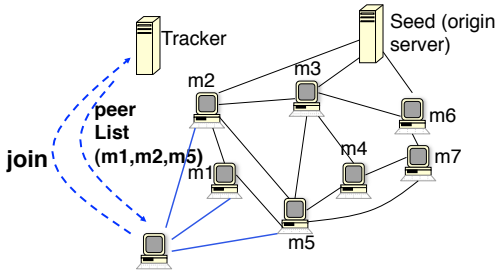
(Note*: probable protocol; Skype protocol is not published)

BitTorrent (2001)

- Allow fast downloads even when sources have low up-link capacity
- How does it work?
  - **Seed** (origin) – site storing the file to be downloaded
  - **Tracker** – server maintaining the list of peers in system
  - Split each file into **pieces** (~ 256 KB each), and each piece into **sub-pieces** (~ 16 KB each)
  - The loader loads one piece at a time
  - Within one piece, the loader can load up to five sub-pieces in **parallel**

BitTorrent: Join Procedure

1) Peer contacts tracker responsible for file it wants to download
2) Tracker returns a list of peer (20-50) downloading same file
3) Peer connects to peers in the list



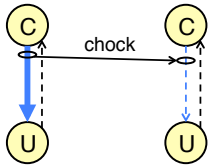
BitTorrent: Download Algorithm

- Download consists of three phases:
- **Start:** get a piece as soon as possible
  - Select a random piece
- **Middle:** spread all pieces as soon as possible
  - Select rarest piece next
- **End:** avoid getting stuck with a slow source, when downloading the last sub-pieces
  - Request in parallel the same sub-piece
  - Cancel slowest downloads once a sub-piece has been received

(For details see: http://bittorrent.org/bittorrentecon.pdf)

Bittorrent: Handling Freeriders

- Freeriders: peers that use the network without contributing (with the upstream bandwidth)
- Solution: chocking, a variant of Tit-for-Tat
  - Each peer has a limited number of upload slots
  - When a peer's upload bandwidth is saturated, it exchanges upload bandwidth for download bandwidth
  - If peer U downloads from peer C and doesn't upload in return, C chokes download to U



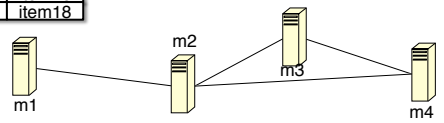
Distributed Hash Tables (DHTs)

- Goal: make sure an item (file) identified is always found
- Abstraction: a distributed hash-table data structure
  - *insert(id, item);*
  - *item = query(id);*
  - Note: *item* can be anything: a data object, document, file, pointer to a file…
- Proposals
  - CAN, Chord, Kademlia, Pastry, Viceroy, Tapestry, etc

DHTs

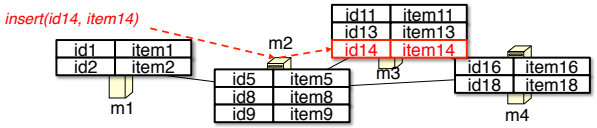
- Partition hash table across nodes

id1	item1
id2	item2
id5	item5
id8	item8
id9	item9
id11	item11
id13	item13
id16	item16
id18	item18



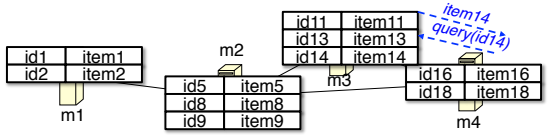
DHT: Insertion

- Call *insert(id4, item14)* at m1
  - Find node responsible for id4, i.e., node m3
  - Insert (id14, item14) at m3



DHT: Query

- Call *query(id4)* at m4
  - Find node responsible for id4, i.e., m3
  - Return (id14, item14) to m4

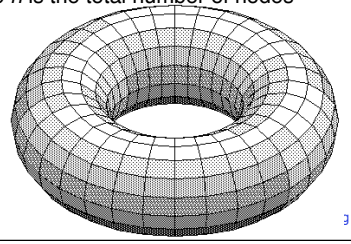


DHT: Lookup Service

- Key primitive: lookup service
  - Given an ID, map it to a host
- Challenges
  - Scalability: hundreds of thousands or millions of machines
  - Instability
    - » Changes in routes, congestion, availability of machines
  - Heterogeneity
    - » Latency: 1ms to 1000ms
    - » Bandwidth: 32Kb/s to 100Mb/s
    - » Nodes stay in system from 10s to a year
  - Trust
    - » Selfish users
    - » Malicious users

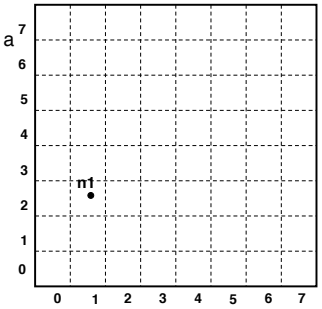
Content Addressable Network (CAN)

- Associate to each node and item a unique *id* in an *d*-dimensional space, e.g., torus
- Properties
  - Routing table size O(*d*), i.e., each node needs to know about O(*d*)
  - Guarantees that a file is found in at most $d*n^{1/d}$ steps, where *n* is the total number of nodes



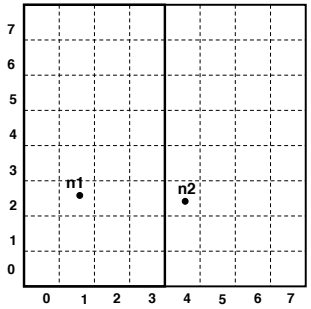
CAN Example: Two Dimensional Space

- Space divided between nodes
- All nodes cover the entire space
- Each node covers either a square or a rectangular area of ratios 1:2 or 2:1
- Example:
  - Assume space size (8 x 8)
  - Node n1:(1, 2) first node that joins → cover the entire space



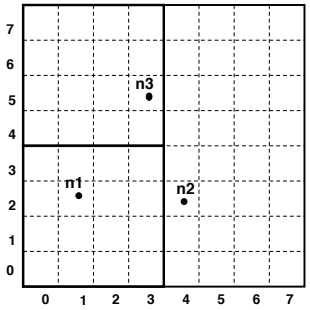
CAN Example: Two Dimensional Space

- Node n2:(4, 2) joins → space is divided between n1 and n2



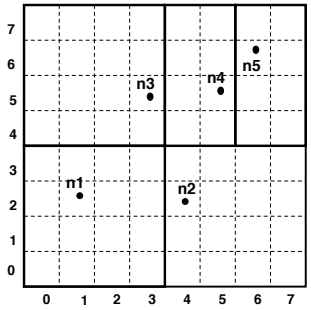
CAN Example: Two Dimensional Space

- Node n2:(4, 2) joins → space is divided between n1 and n2



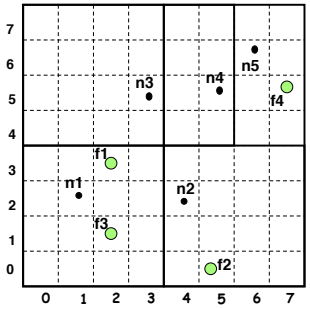
CAN Example: Two Dimensional Space

- Nodes n4:(5, 5) and n5:(6,6) join



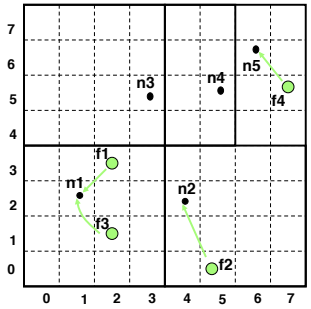
CAN Example: Two Dimensional Space

- Nodes: n1:(1, 2); n2:(4,2); n3:(3, 5); n4:(5,5);n5:(6,6)
- Items: f1:(2,3); f2:(5,0); f3:(2,1); f4:(7,5);



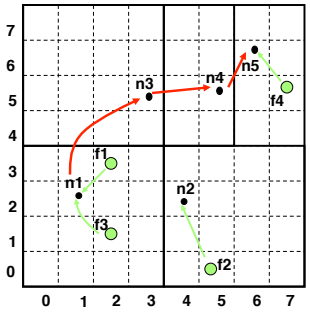
CAN Example: Two Dimensional Space

- Each item is stored by the node who owns its mapping in the space



CAN: Query Example

- Each node knows its neighbors in the *d*-space
- Forward query to the neighbor that is closest to the query *id*
- Example: assume n1 queries f4



5min Break

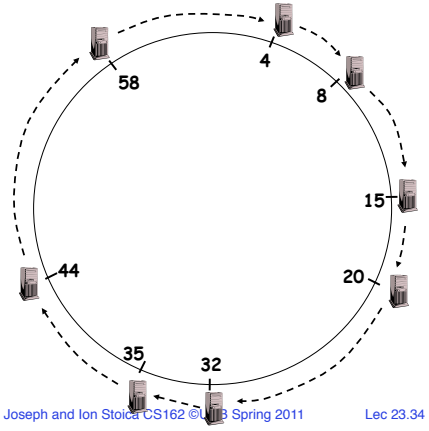
Chord

- Associate to each node and item a unique *id* in an *uni-*dimensional space $0..2^m-1$
- Key design decision
  - Decouple correctness from efficiency
- 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

Identifier to Node Mapping Example

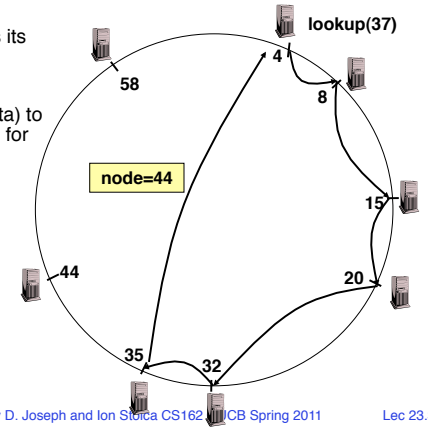
- Node 8 maps [5,8]
- Node 15 maps [9,15]
- Node 20 maps [16, 20]
- …
- Node 4 maps [59, 4]

- Each node maintains a pointer to its successor



Lookup

- Each node maintains its successor
- Route packet (ID, data) to the node responsible for ID using successor pointers



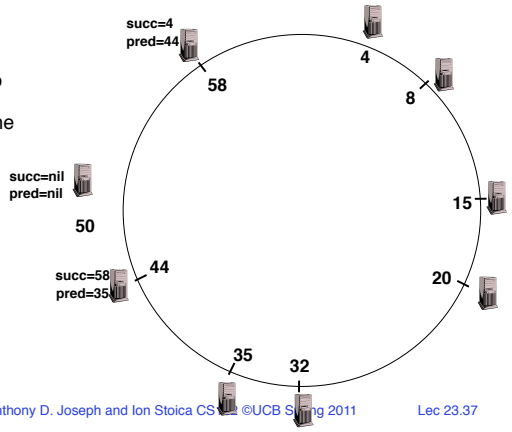
Stabilization Procedure

- Periodic operation performed by each node N to handle joins

```
N: periodically:
    send GET_PRED to N.successor;

M: upon receiving GET_PRED from N:
    reply PRED(M.predecessor) to N;

N: upon receiving PRED(M')
    if (M' between (N, N.successor))
          N.successor = M';
    send NOTIFY to N.successor;

M: upon receiving NOTIFY from N:
    if ((N between (Mpredecessor, M)) ll M.predecessor = NULL)
       predecessor = N;
```

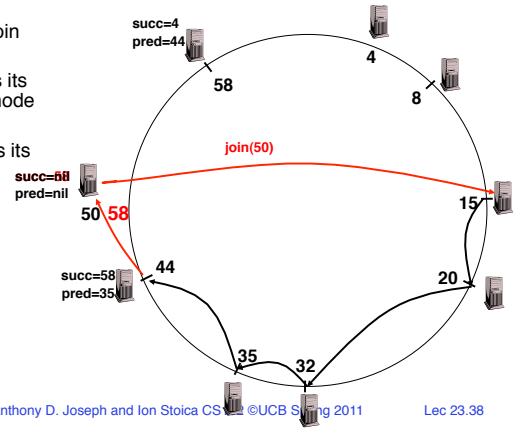
Joining Operation

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



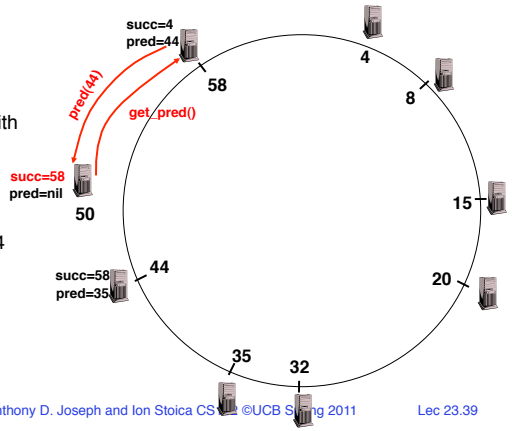
Joining Operation

- Node 50: send join (50) to node 15
- Node 44: returns its successor, i.e., node 58
- Node 50 updates its successor to 58



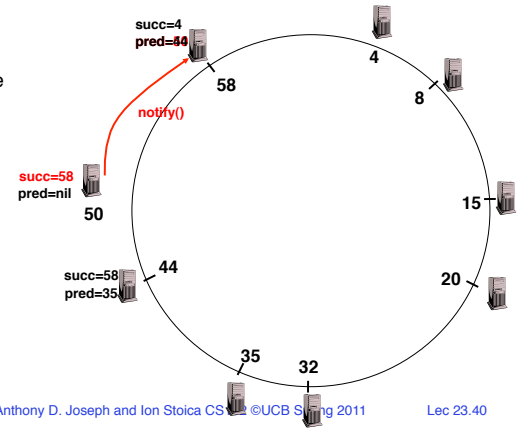
Joining Operation

- Node 50: ask its successor (node 58) for its predecessor
- Node 58: reply with its predecessor, i.e., node 44
- Node 50: doesn't do anything as 44 not in (50,58)



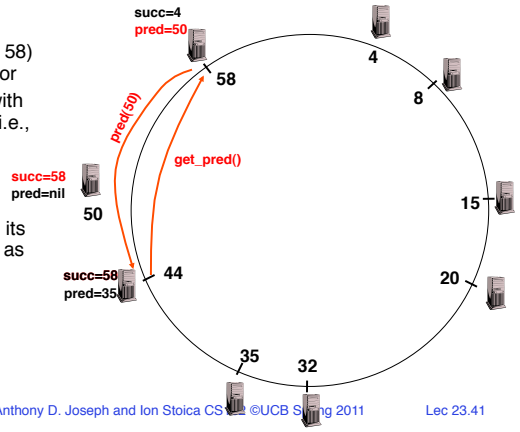
Joining Operation

- Node 50: send NOTIFY to 58
- Node 58: update its predecessor to 50 as 50 is in (44, 58)



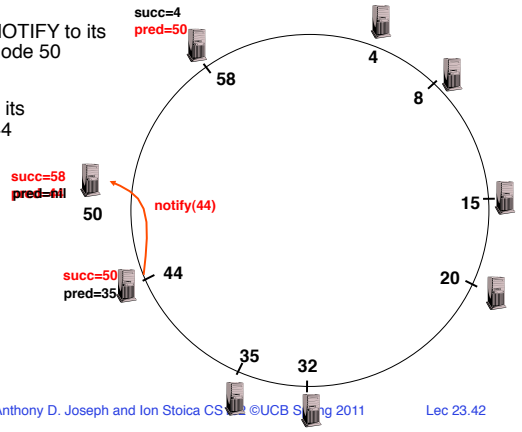
Joining Operation (cont'd)

- Node 44: ask its successor (node 58) for its predecessor
- Node 58: reply with its predecessor, i.e., node 50
- Node 44: update its successor to 50, as 50 in (44, 58)



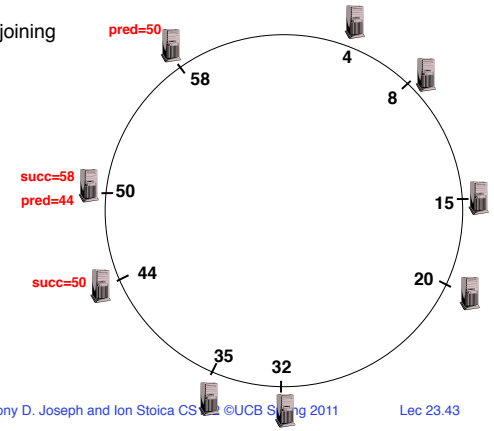
Joining Operation (cont'd)

- Node 44: send NOTIFY to its successor, i.e., node 50
- Node 50: update its predecessor to 44



Joining Operation (cont'd)

- This completes the joining operation!

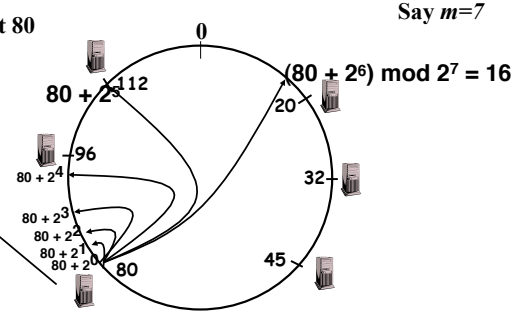


Achieving Efficiency: *finger tables*

Finger Table at 80

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

Say $m=7$



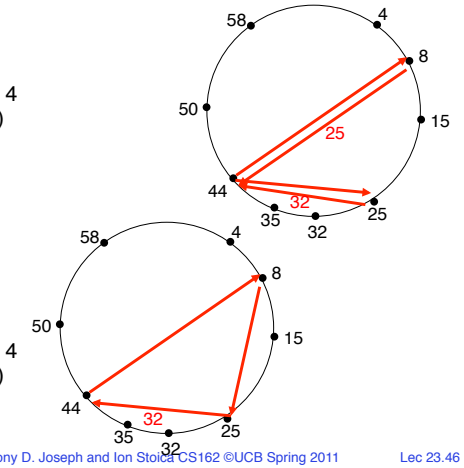
ith entry at peer with id n is first peer with id >= $n + 2^i \pmod{2^m}$

Achieving Robustness

- To improve robustness each node maintains the k (> 1) immediate successors instead of only one successor
- In the pred() reply message, node A can send its k-1 successors to its predecessor B
- Upon receiving pred() message, B can update its successor list by concatenating the successor list received from A with its own list

Iterative vs. Recursive Queries

- Iteratively:
  - Example: node 4 issue query(31)
- Recursively
  - Example: node 4 issue query(31)



Performance

- Routing in the overlay network can be more expensive than in the underlying network
  - Because usually there is no correlation between node ids and their locality;
    - » A query can repeatedly jump from Europe to North America, though both the initiator and the node that store the item are in Europe!
- Solutions: CAN and Chord maintain multiple copies for each entry in their routing tables and choose the closest in terms of network distance

Conclusions

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