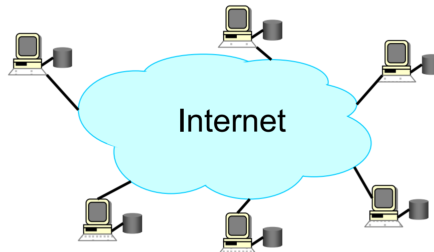
Peer-to-Peer Networks and Distributed Hash Tables

This lecture is based on a lecture by **Ion Stoica (UC Berkeley).** Slides used with permission from author. All rights remain with author.

How Did it Start?

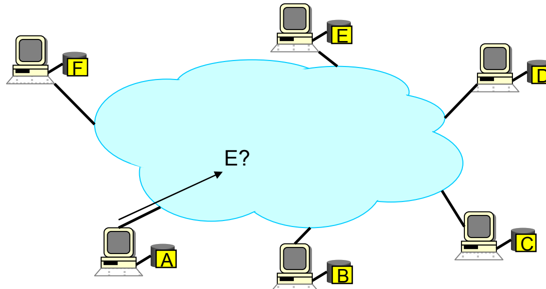
- A killer application: Napster
  - Free music over the Internet
- Key idea: share the **content**, storage *and* bandwidth of individual (home) users



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

Challenges

- Find where a particular file is stored

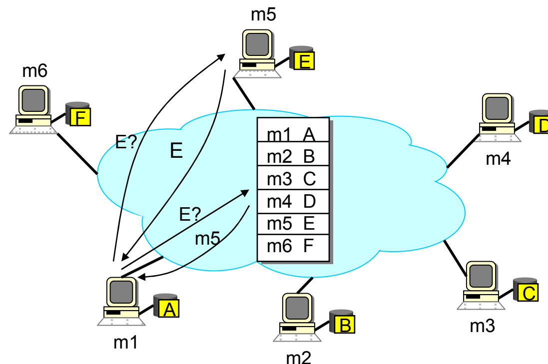


- Scale: up to hundred of thousands or millions of systems
- Dynamicity: systems can come and go any time

Napster

- Assumes a centralized index system that maps files (songs) to machines that are "alive"
- Procedure to access a file:
  - Query the index system, which returns a machine that stores the required file
    - Ideally this is the closest/least-loaded machine
  - Copy the file with ftp
- Advantages:
  - Simplicity, easy to implement sophisticated search engines on top of the index system
- Disadvantages:
  - All disadvantages of a centralizes design

Napster: Example

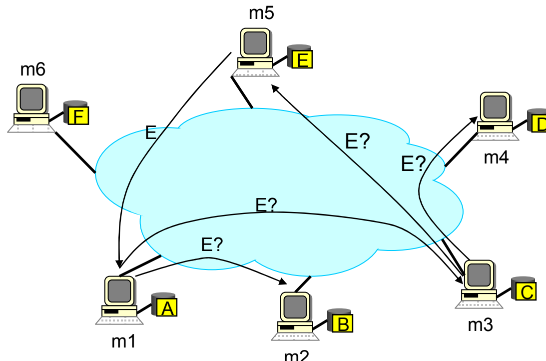


Gnutella

- Distribute file location problem
- Idea: Flood the request
- Procedure to access a file:
  - Sites are setup as an overlay network
  - User send request to all neighbors in the overlay network
  - Neighbors recursively send request to their own 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 request (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;…



Freenet

- Addition goals to file location:
  - Provide publisher anonymity, security
  - Resistant to attacks – a third party shouldn't be able to deny the access to a particular file (data item, object), even if it compromises a large fraction of machines
- Architecture:
  - Each file is identified by a unique identifier
  - Each machine stores a set of files, and maintains a "routing table" to route the individual requests

Data Structure

- Each node maintains a common stack
  - *id* – file identifier
  - *next_hop* – another node that store the file id
  - *file* – file identified by *id* being stored on the local node
- Forwarding:
  - Each message contains the file *id* it is referring to
  - If file *id* stored locally, then stop;
  - If not, search for the “closest” *id* in the stack, and forward the message to the corresponding *next_hop*

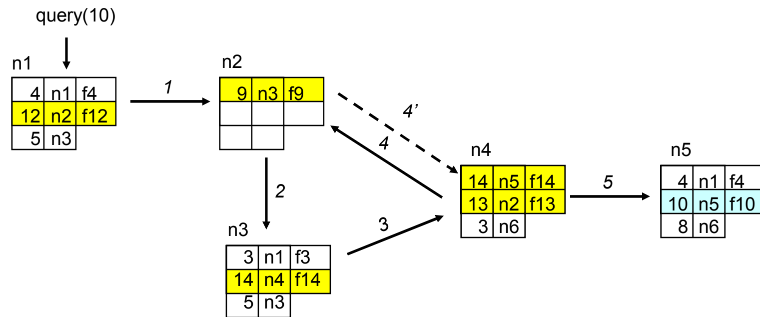
id	*next_hop*	*file*
↓	⋮ ↓	

Query

- API: *file* = query(*id*);
- Upon receiving a query for document *id*
  - Check whether the queried file is stored locally
    - If yes, return it
    - If not, forward the query message
- Notes:
  - Each query is associated a TTL that is decremented each time the query message is forwarded.

    To obscure distance to originator:
    - TTL can be initiated to a random value within some bounds
    - When TTL=1, the query is forwarded with a finite probability
  - Each node maintains the state for all outstanding queries that have traversed it → help to avoid cycles
  - When file is returned, the file is cached along the reverse path

Query Example



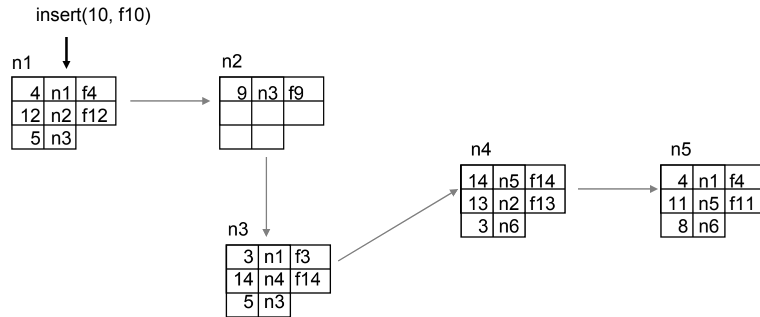
- Note: doesn't show file caching on the reverse path

Insert

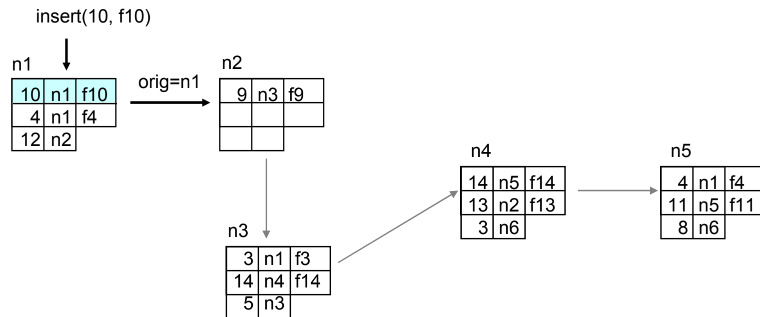
- API: insert(*id*, *file*);
- Two steps
  - Search for the file to be inserted
  - If not found, insert the file
- **Searching:** like query, but nodes maintain state after a collision is detected and the reply is sent back to the originator
- **Insertion**
  - Follow the forward path; insert the file at all nodes along the path
  - A node probabilistically replace the originator with itself; obscure the true originator

Insert Example

- Assume query returned failure along “gray” path; insert f10

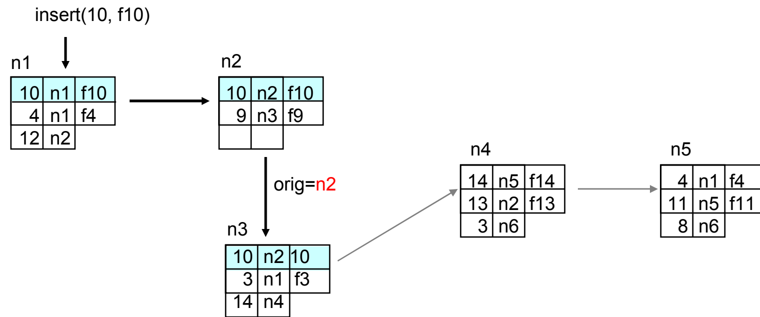


Insert Example



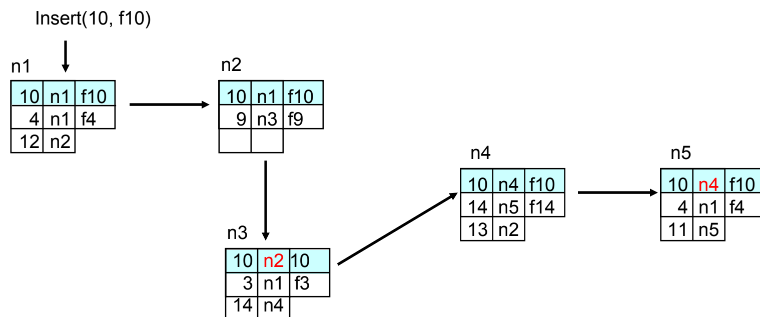
Insert Example

- n2 replaces the originator (n1) with itself



Insert Example

- n2 replaces the originator (n1) with itself



Freenet Properties

- Newly queried/inserted files are stored on nodes storing similar ids
- New nodes can announce themselves by inserting files

istoica@cs.berkeley.edu

Freenet Summary

- Advantages
  - Provides publisher anonymity
  - Totally decentralize architecture → robust and scalable
  - Resistant against malicious file deletion
- Disadvantages
  - Does not always guarantee that a file is found, even if the file is in the network

istoica@cs.berkeley.edu

Other Solutions to the Location Problem

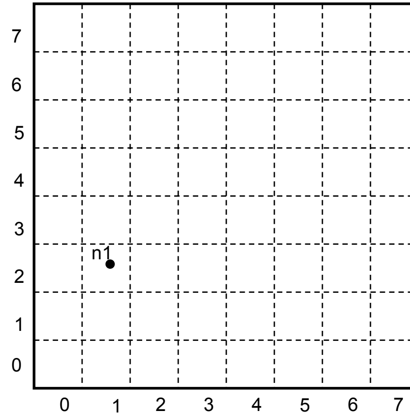
- Goal: make sure that an item (file) identified is always found
- Abstraction: a **distributed hash-table (DHT)** data structure
  - insert(id, item);
  - item = query(id);
  - Note: item can be anything: a data object, document, file, pointer to a file…
- Proposals
  - CAN, Chord, Kademlia, Pastry,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

Content Addressable Network (CAN)

- Associate to each node and item a unique *id* in an *d*-dimensional Cartesian space
- Goals
  - Scales to hundreds of thousands of nodes
  - Handles rapid arrival and failure of nodes
- Properties
  - Routing table size O(*d*)
  - Guarantees that a file is found in at most $d \times n^{1/d}$ steps, where *n* is the total number of nodes

CAN Example: Two Dimensional Space

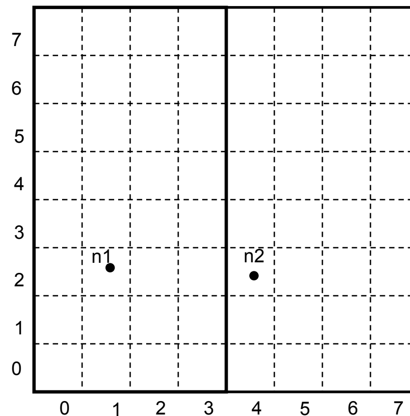
- Space divided between nodes
- All nodes cover the entire space
- Each node covers a zone which is either a square or a rectangular area of ratios 1:2 or 2:1
- Example:
  - Node n1 randomly selects a coordinate, say (1,2):
  - Node n1:(1, 2) first node that joins → cover the entire space



CAN Example: Two Dimensional Space

- Node n2:(4, 2) joins

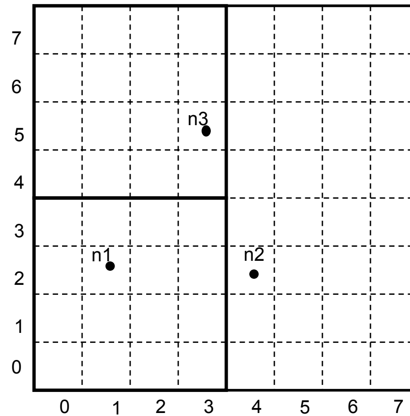
→ n2 finds the owner of the zone where (4,2) lies. This is n1
→ n1 splits the zone with n2



CAN Example: Two Dimensional Space

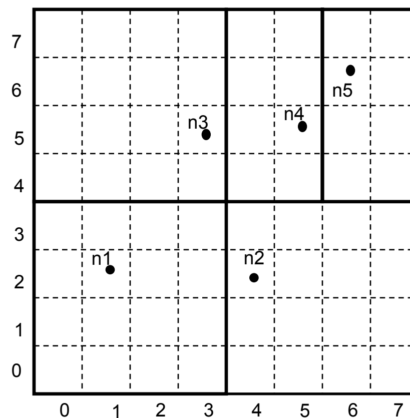
- Node n3:(3,5) joins

→ Node n3 finds the node that owns the zone where (3,5) lies. This is n1.
→ n1 and n3 split their zones



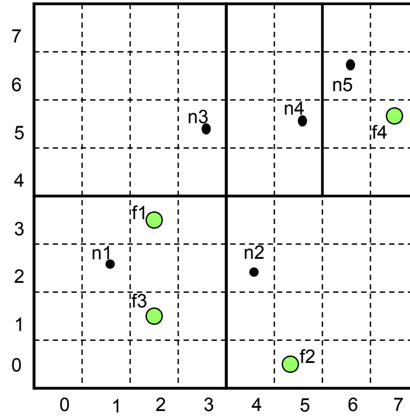
CAN Example: Two Dimensional Space

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



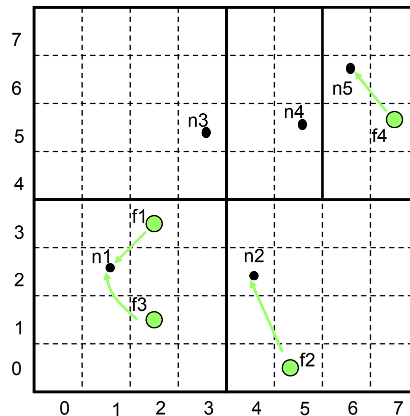
CAN Example: Two Dimensional Space

- Items (files) have an (x,y) key.
- Nodes: n1:(1, 2); n2:(4,2); n3:(3, 5); n4:(5,5);n5:(6,6)
- Items: f1:(2,3); f2:(5,1); f3:(2,1); f4:(7,5);



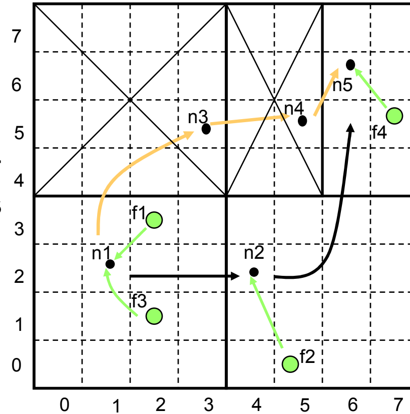
CAN Example: Two Dimensional Space

- 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
- Can route around some failures



istoica@cs.berkeley.edu

Node Failure Recovery

- Simple failures
  - Know your neighbor's neighbors
  - When a node fails, one of its neighbors takes over its zone
- More complex failure modes
  - Simultaneous failure of multiple adjacent nodes
  - Scoped flooding to discover neighbors
  - Hopefully, a rare event

istoica@cs.berkeley.edu

Chord

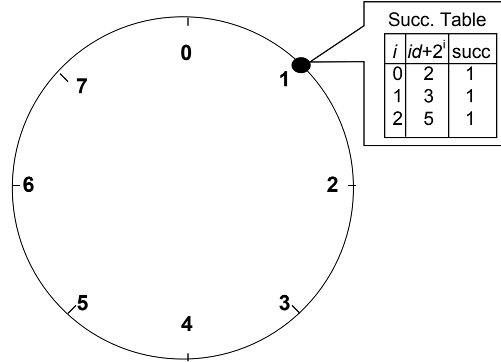
- Associate to each node and item a unique *id* in an *uni-*dimensional space
- Goals
  - Scales to hundreds of thousands of nodes
  - Handles rapid arrival and failure of nodes
- Properties
  - Routing table size O(log(*N*)) , where *N* is the total number of nodes
  - Guarantees that a file is found in O(log(*N*)) steps

Data Structure

- Assume identifier space is $0..2^m$
- Each node maintains
  - **Finger table**
    - Entry *i* in the finger table of *n* is the first node that succeeds or equals $n + 2^i$
  - **Predecessor node**
- An item identified by *id* is stored on the succesor node of *id*

Chord Example

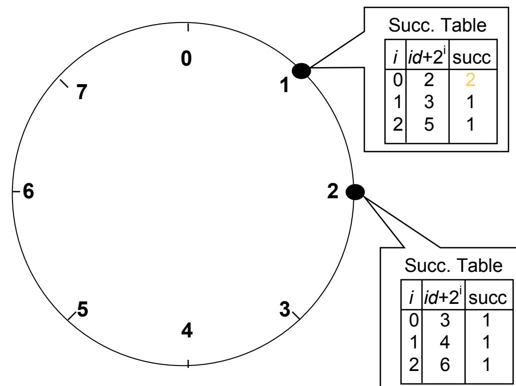
- Assume an identifier space 0..8
- Node n1:(1) joins→all entries in its finger table are initialized to itself



i	$id+2^i$	succ
0	2	1
1	3	1
2	5	1

Chord Example

- Node n2:(3) joins



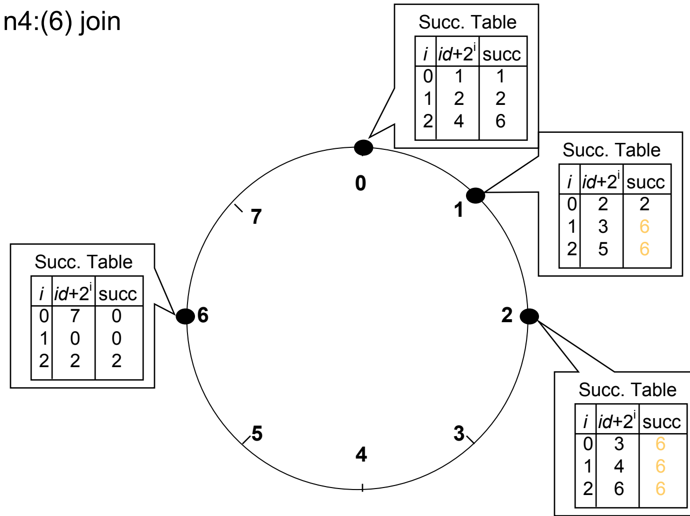
i	$id+2^i$	succ
0	2	2
1	3	1
2	5	1

Succ. Table

i	$id+2^i$	succ
0	3	1
1	4	1
2	6	1

Chord Example

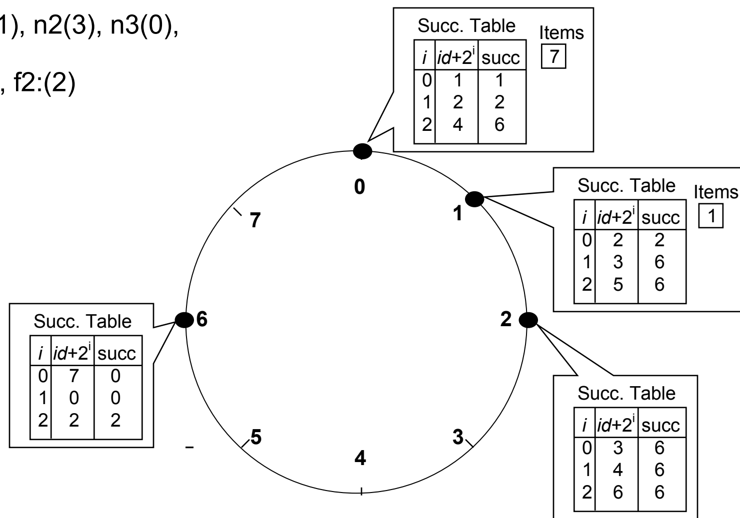
- Nodes n3:(0), n4:(6) join



istoica@cs.berkeley.edu

Chord Examples

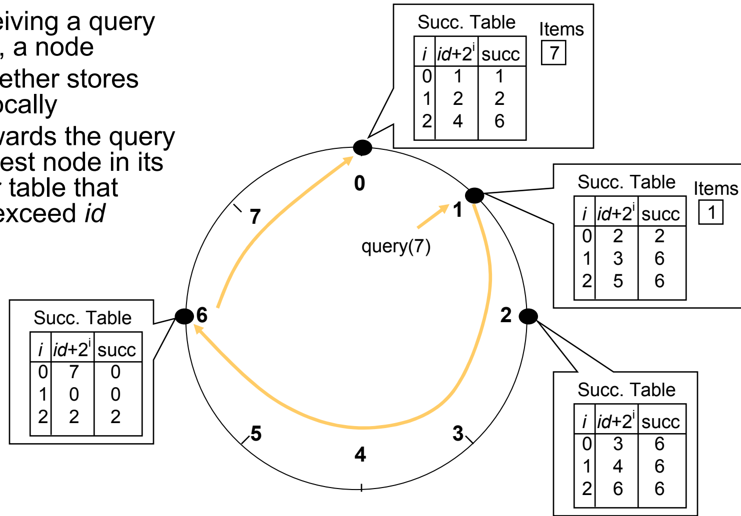
- Nodes: n1:(1), n2(3), n3(0), n4(6)
- Items: f1:(7), f2:(2)



istoica@cs.berkeley.edu

Query

- Upon receiving a query for item *id*, a node
- Check whether stores the item locally
- If not, forwards the query to the largest node in its successor table that does not exceed *id*



Node Joining

- Node n joins the system:
  - n picks a random identifier, id
  - n performs n’ = lookup(id)
  - n->successor = n’

State Maintenance: Stabilization Protocol

- Periodically node n
  - Asks its successor, n', about its predecessor n''
  - If n'' is between n' and n''
    - n->successor = n''
    - **notify** n'' that n its predecessor
- When node n'' receives notification message from n
  - If n is between n''->predecessor and n'', then
    - n''->predecessor = n
- Improve robustness
  - Each node maintain a successor **list** (usually of size 2*log N)

istoica@cs.berkeley.edu

CAN/Chord Optimizations

- Weight neighbor nodes by RTT
  - When routing, choose neighbor who is closer to destination with lowest RTT from me
  - Reduces path latency
- Multiple physical nodes per virtual node
  - Reduces path length (fewer virtual nodes)
  - Reduces path latency (can choose physical node from virtual node with lowest RTT)
  - Improved fault tolerance (only one node per zone needs to survive to allow routing through the zone)
- Several others

istoica@cs.berkeley.edu

Conclusions

- Distributed Hash Tables are a key component of scalable and robust overlay networks
- CAN: O(d) state, O(d*n1/d) distance
- Chord: O(log n) state, O(log n) distance
- Both can achieve stretch < 2
- Simplicity is key
- Services built on top of distributed hash tables
  - p2p file storage, i3 (chord)
  - multicast (CAN, Tapestry)
  - persistent storage (OceanStore using Tapestry)