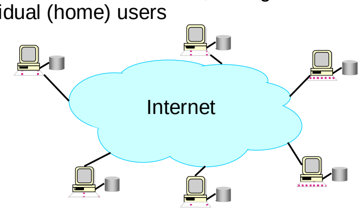
CS 194: Distributed Systems
Distributed Hash Tables

Scott Shenker and Ion Stoica
Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
Berkeley, CA 94720-1776

How Did it Start?

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

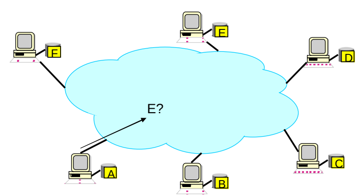


Model

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

Main Challenge

- Find where a particular file is stored



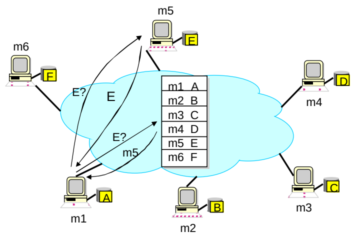
Other Challenges

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

Napster

- Assume a centralized index system that maps files (songs) to machines that are alive
- How to find a file (song)
  - Query the index system → return a machine that stores the required file
    - Ideally this is the closest/least-loaded machine
  - ftp the file
- Advantages:
  - Simplicity, easy to implement sophisticated search engines on top of the index system
- Disadvantages:
  - Robustness, scalability (?)

Napster: Example

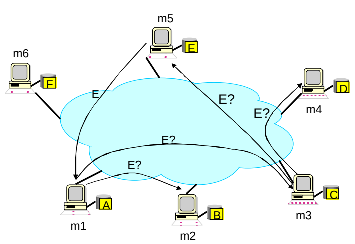


Gnutella

- Distribute file location
- Idea: flood the request
- Hot 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 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;…



Distributed Hash Tables (DHTs)

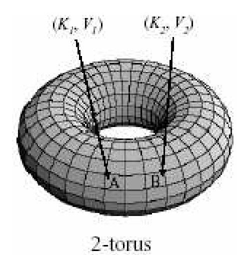
- Abstraction: a distributed hash-table data structure
  - insert(id, item);
  - item = query(id); (or lookup(id);)
  - Note: item can be anything: a data object, document, file, pointer to a file…
- Proposals
  - CAN, Chord, Kademlia, Pastry, Tapestry, etc

DHT Design Goals

- Make sure that an item (file) identified is always found
- Scales to hundreds of thousands of nodes
- Handles rapid arrival and failure of nodes

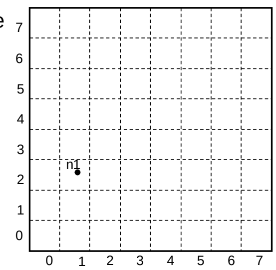
Content Addressable Network (CAN)

- Associate to each node and item a unique *id* in an *d*-dimensional Cartesian space on a *d*-torus
- Properties
  - Routing table size 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

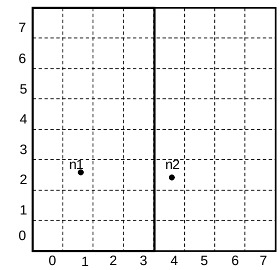
- Space divided between nodes
- All nodes cover the entire space
- Each node covers either a square or a rectangular area of ratios 1:2 or 2:1
- Example:
  - Node n1:(1, 2) first node that joins → cover the entire space



13

CAN Example: Two Dimensional Space

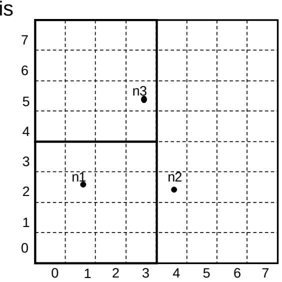
- Node n2:(4, 2) joins → space is divided between n1 and n2



14

CAN Example: Two Dimensional Space

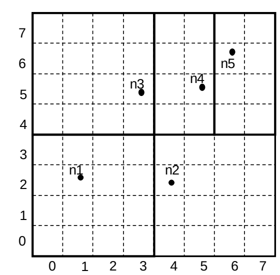
- Node n2:(4, 2) joins → space is divided between n1 and n2



15

CAN Example: Two Dimensional Space

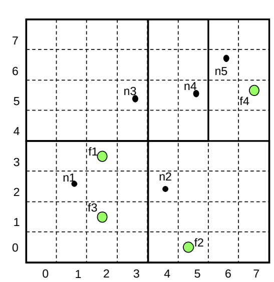
- Nodes n4:(5, 5) and n5:(6,6) join



16

CAN Example: Two Dimensional Space

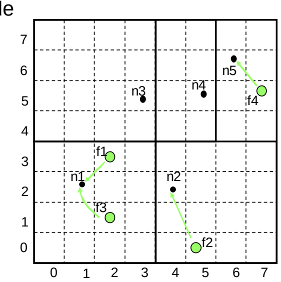
- Nodes: n1:(1, 2); n2:(4,2); n3:(3, 5); n4:(5,5);n5:(6,6)
- Items: f1:(2,3); f2:(5,1); f3:(2,1); f4:(7,5);



17

CAN Example: Two Dimensional Space

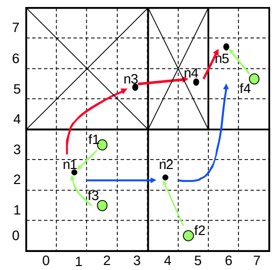
- Each item is stored by the node who owns its mapping in the space



18

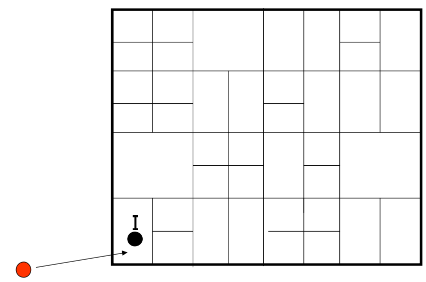
CAN: Query Example

- 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



19

CAN: Node Joining

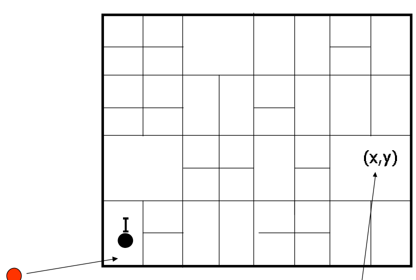


new node

1) Discover some node "I" already in CAN

20

CAN: Node Joining

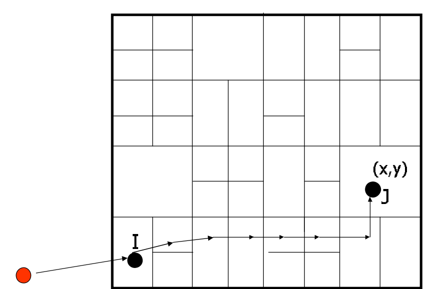


new node

2) Pick random point in space

21

CAN: Node Joining

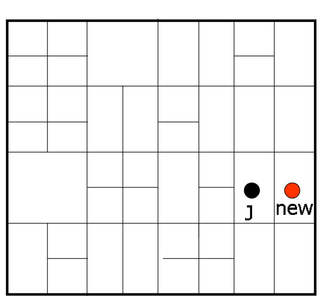


new node

3) I routes to (x,y), discovers node J

22

CAN: Node Joining



4) split J's zone in half… new node owns one half

23

Node departure

- Node explicitly hands over its zone and the associated (key,value) database to one of its neighbors
- Incase of network failure this is handled by a take-over algorithm
- Problem : take over mechanism does not provide regeneration of data
- Solution:
  every node has a backup of its neighbours

24

Chord

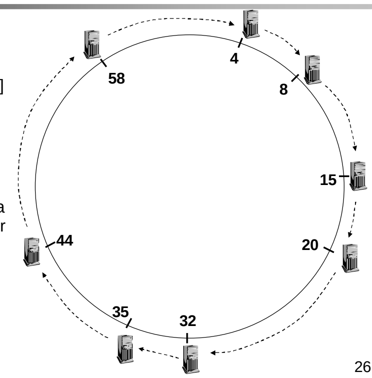
- Associate to each node and item a unique *id* in an *uni-*dimensional space 0..2^m-1
- Goals
  - Scales to hundreds of thousands of nodes
  - Handles rapid arrival and failure of nodes
- Properties
  - Routing table size O(log(*N*)) , where *N* is the total number of nodes
  - Guarantees that a file is found in O(log(*N*)) steps

25

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]

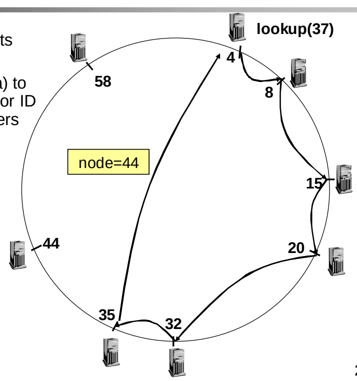
- Each node maintains a pointer to its successor



26

Lookup

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



27

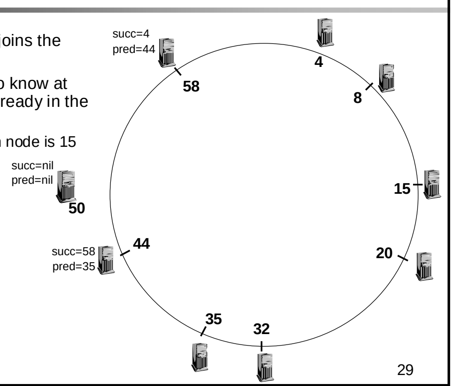
Joining Operation

- Each node A periodically sends a stabilize() message to its successor B
- Upon receiving a stabilize() message node B
  - returns its predecessor B'=pred(B) to A by sending a notify(B') message
- Upon receiving notify(B') from B,
  - if B' is between A and B, A updates its successor to B'
  - A doesn't do anything, otherwise

28

Joining Operation

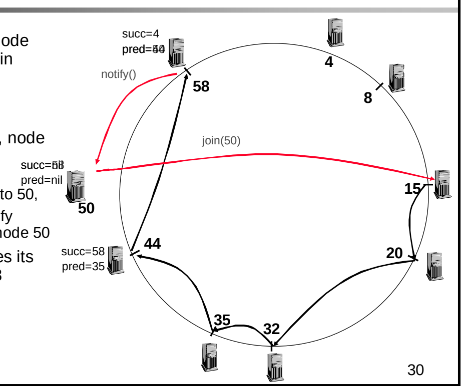
- Node with id=50 joins the ring
- Node 50 needs to know at least one node already in the system
  - Assume known node is 15



29

Joining Operation

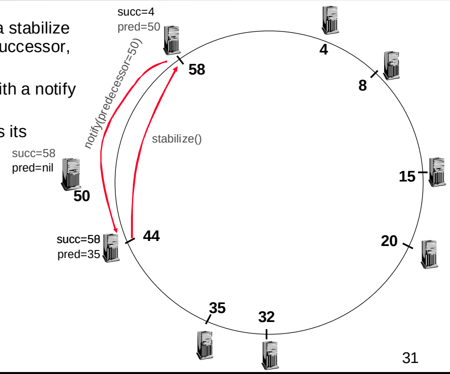
- Node 50 asks node 15 to forward join message
- When join(50) reaches the destination (i.e., node 58), node 58
  1) updates its predecessor to 50,
  2) returns a notify message to node 50
- Node 50 updates its successor to 58



30

Joining Operation (cont'd)

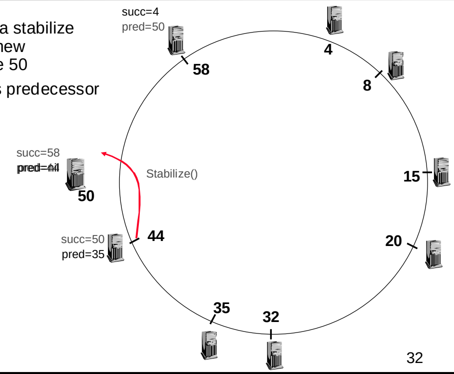
- Node 44 sends a stabilize message to its successor, node 58
- Node 58 reply with a notify message
- Node 44 updates its successor to 50



31

Joining Operation (cont'd)

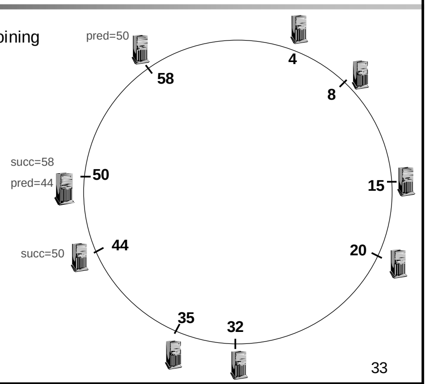
- Node 44 sends a stabilize message to its new successor, node 50
- Node 50 sets its predecessor to node 44



32

Joining Operation (cont'd)

- This completes the joining operation!



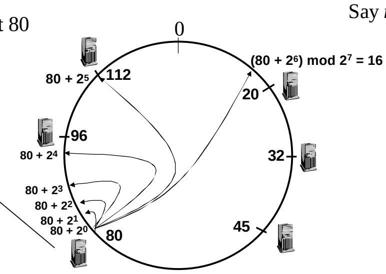
33

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 >= $n+2^i \pmod{2^m}$

34

Achieving Robustness

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

35

CAN/Chord Optimizations

- Reduce latency
  - Chose finger that reduces expected time to reach destination
  - Chose the closest node from range [N+2^{i-1},N+2^i) as successor
- Accommodate heterogeneous systems
  - Multiple virtual nodes per physical node

36

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
  - persistent storage (OpenDHT, Oceanstore)
  - p2p file storage, i3 (chord)
  - multicast (CAN, Tapestry)

37