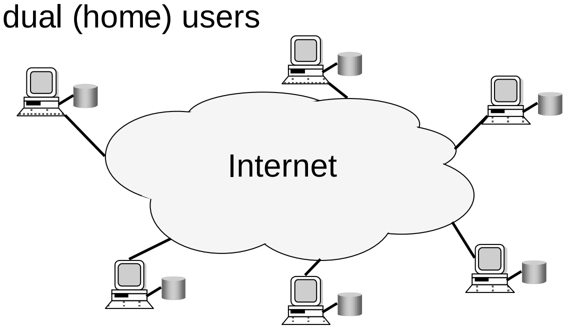
CS 268: Lecture 20 Distributed Hash Tables (DHTs)

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

1

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



2

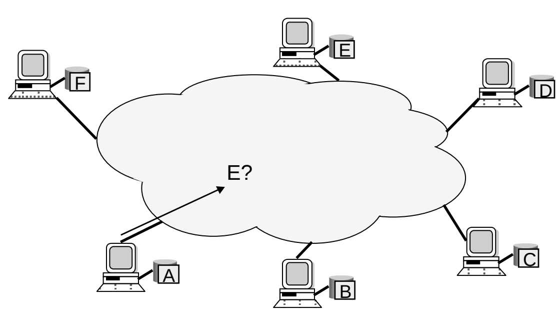
Model

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

3

Main Challenge

- Find where a particular file is stored



4

Other Challenges

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

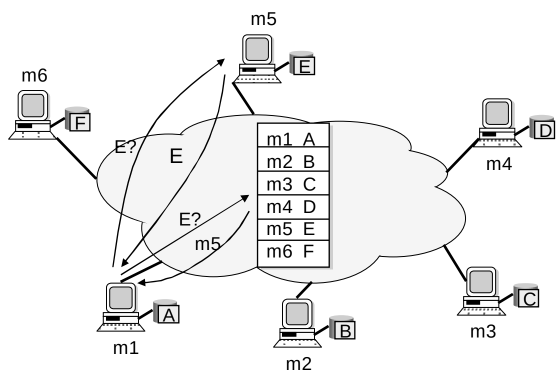
5

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 (?)

6

Napster: Example



7

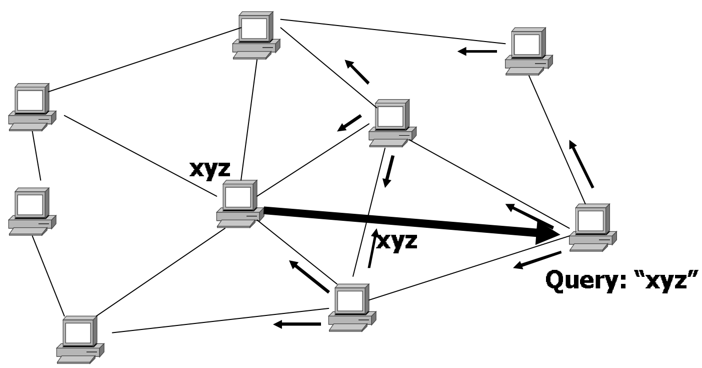
Gnutella

- 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)

8

Gnutella

- Ad-hoc topology
- Queries are flooded for bounded number of hops
- No guarantees on recall



9

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

10

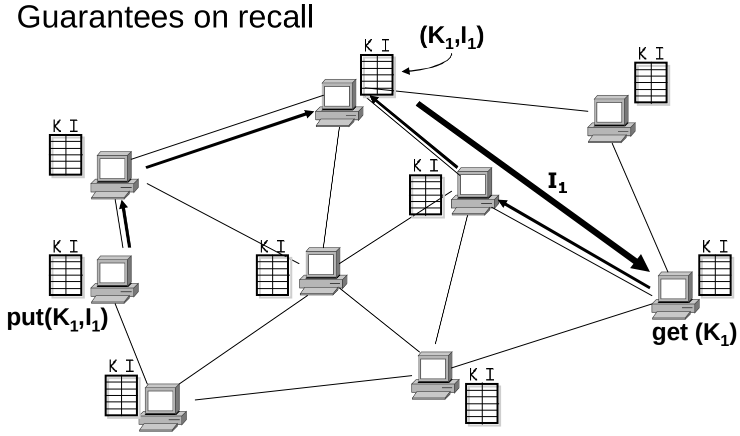
DHT Design Goals

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

11

Structured Networks

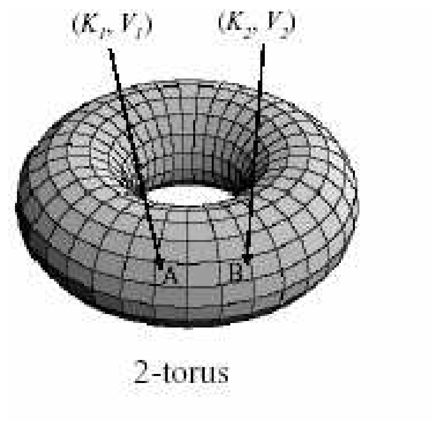
- Distributed Hash Tables (DHTs)
- Hash table interface: **put**(key,item), **get**(key)
- O(log n) hops
- Guarantees on recall



12

Content Addressable Network (CAN)

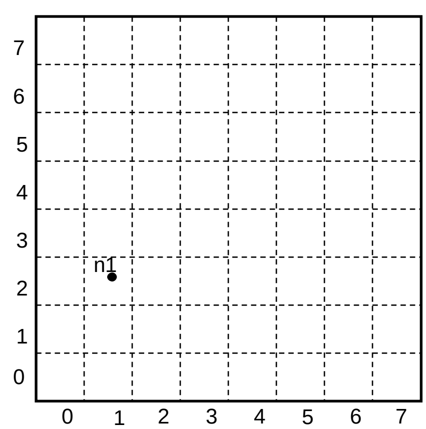
- Associate to each node and item a unique *id* in an *d*-dimensional Cartesian space on a *d*-torus
- Properties
  - Routing table size O(*d*)
  - Guarantees that a file is found in at most $d*n^{1/d}$ steps, where *n* is the total number of nodes



13

CAN Example: Two Dimensional Space

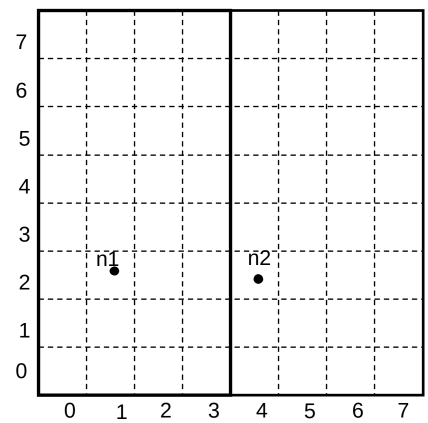
- Space divided between nodes
- All nodes cover the entire space
- Each node covers either a square or a rectangular area of ratios 1:2 or 2:1
- Example:
  - Node n1:(1, 2) first node that joins → cover the entire space



14

CAN Example: Two Dimensional Space

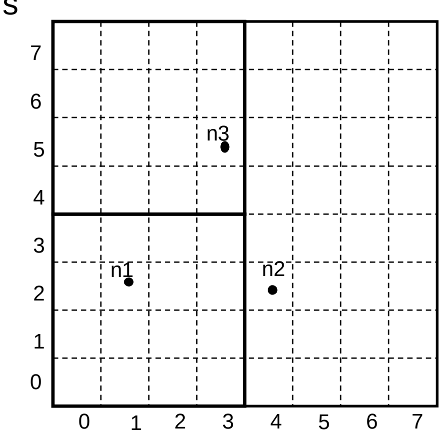
- Node n2:(4, 2) joins → space is divided between n1 and n2



15

CAN Example: Two Dimensional Space

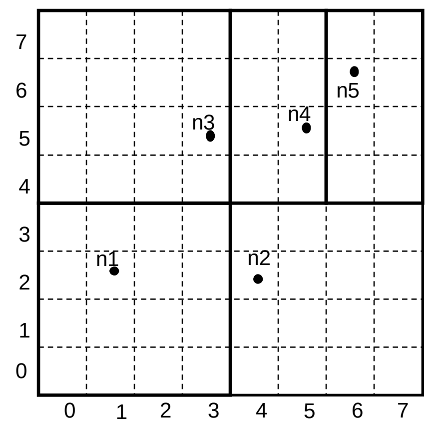
- Node n2:(4, 2) joins → space is divided between n1 and n2



16

CAN Example: Two Dimensional Space

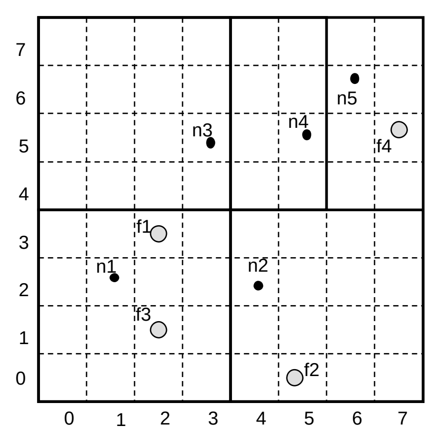
- Nodes n4:(5, 5) and n5:(6,6) join



17

CAN Example: Two Dimensional Space

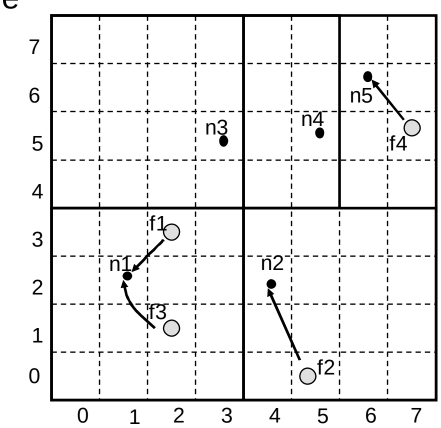
- Nodes: n1:(1, 2); n2:(4,2); n3:(3, 5); n4:(5,5);n5:(6,6)
- Items: f1:(2,3); f2:(5,1); f3:(2,1); f4:(7,5);



18

CAN Example: Two Dimensional Space

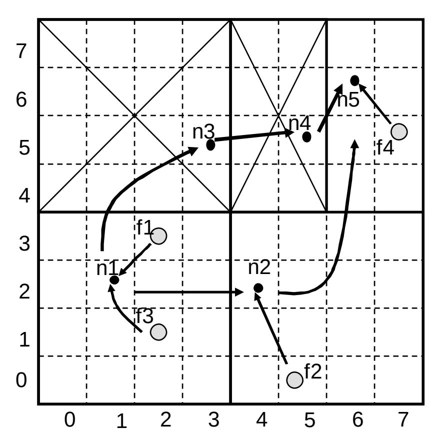
- Each item is stored by the node who owns its mapping in the space



19

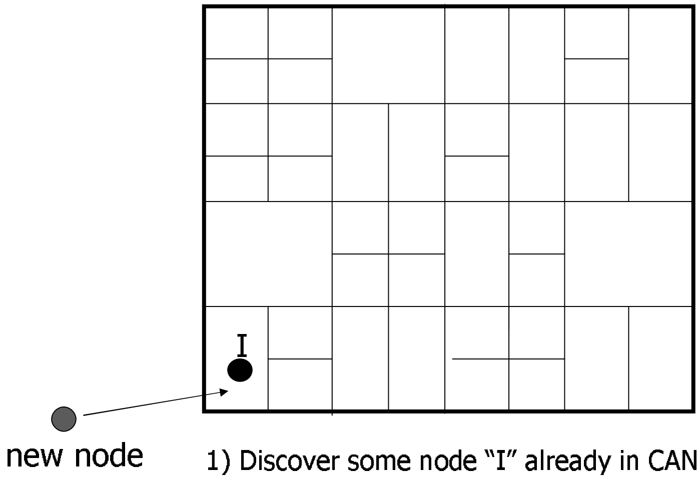
CAN: Query Example

- 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



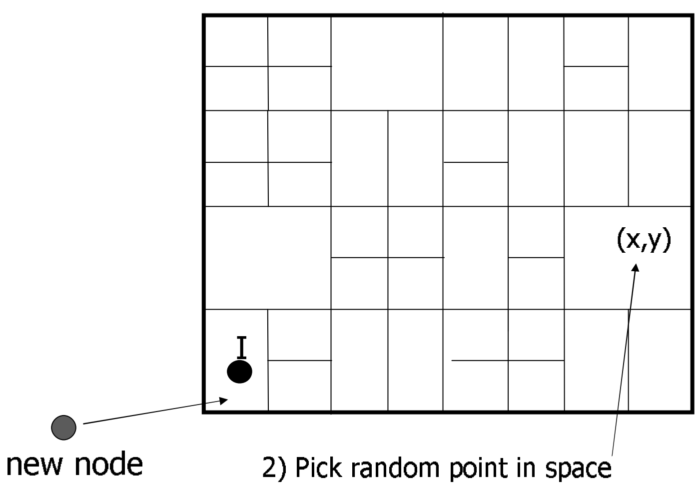
20

CAN: Node Joining



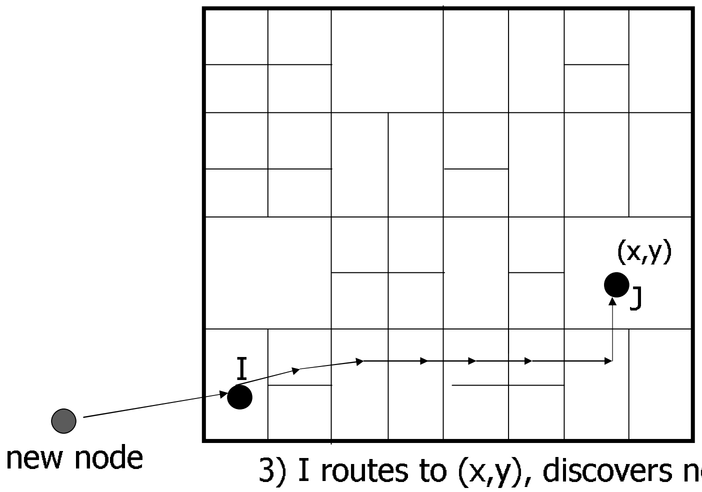
21

CAN: Node Joining



22

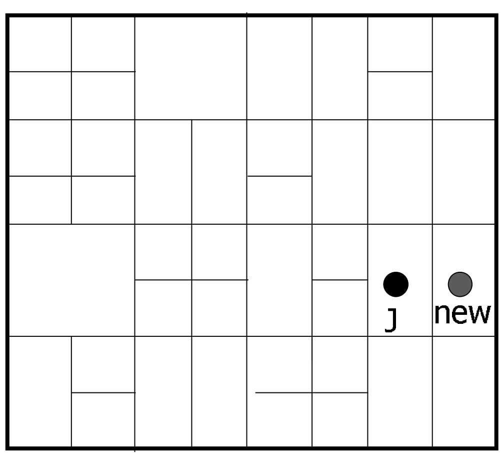
CAN: Node Joining



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

23

CAN: Node Joining



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

24

Node departure

- Node explicitly hands over its zone and the associated (key,value) database to one of its neighbors
- In case 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

25

Chord

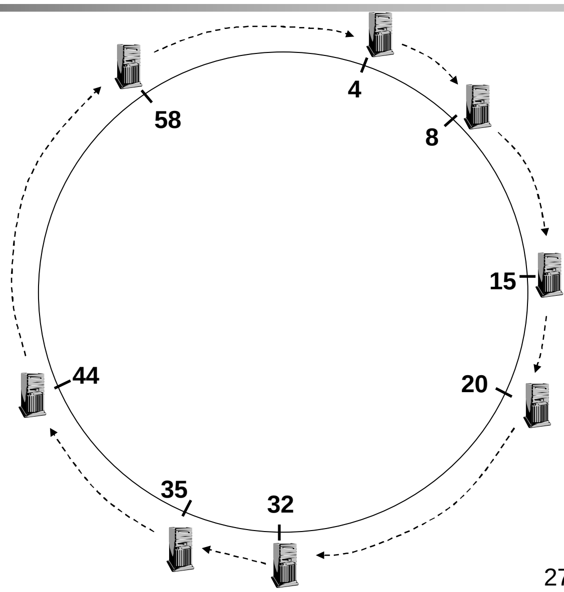
- Associate to each node and item a unique *id* in an *uni-*dimensional space $0..2^m-1$
- Key design decision
  - Decouple correctness from efficiency
- Properties
  - Routing table size O(log(*N*)) , where *N* is the total number of nodes
  - Guarantees that a file is found in O(log(*N*)) steps

26

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

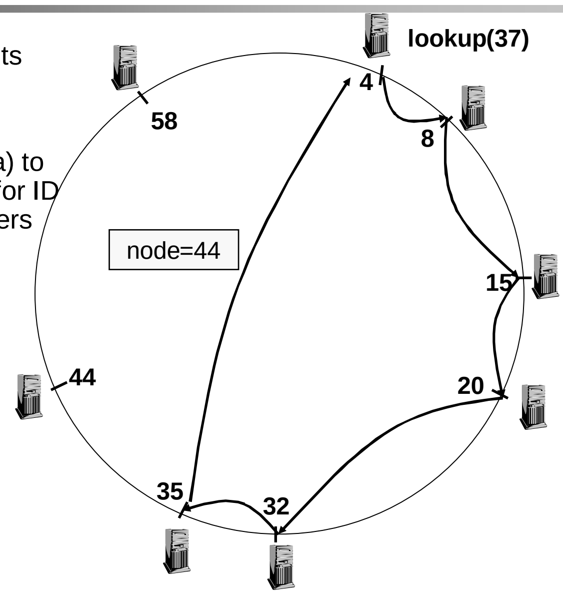


27

Lookup

- Each node maintains its successor

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



28

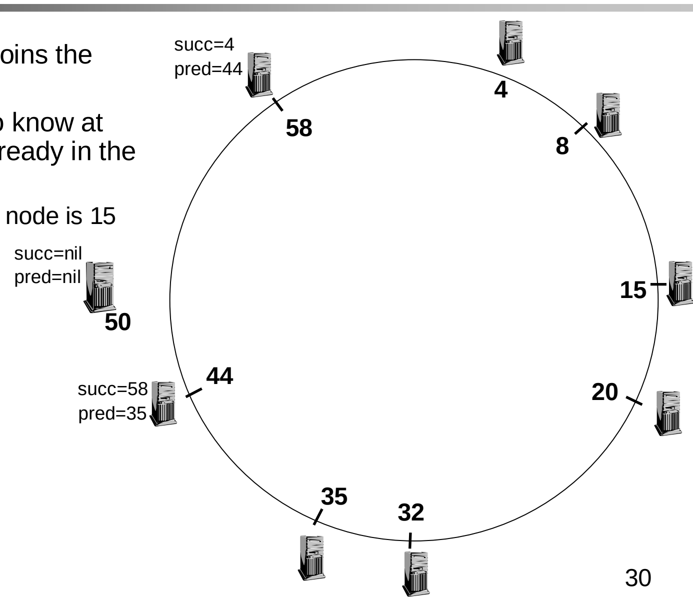
Joining Operation

- 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

29

Joining Operation

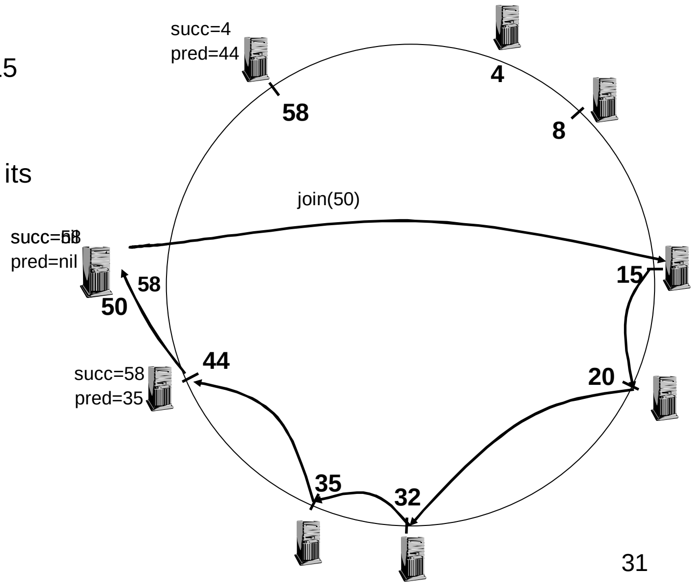
- Node with id=50 joins the ring
- Node 50 needs to know at least one node already in the system
  - Assume known node is 15



30

Joining Operation

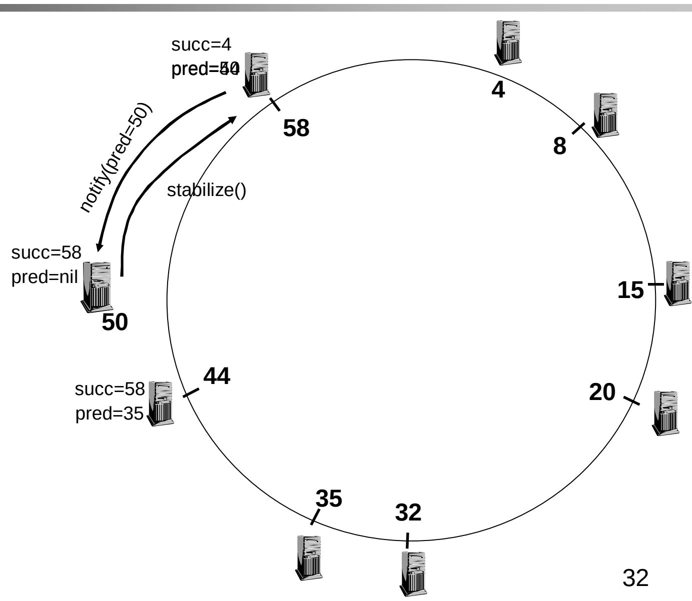
- Node 50: send join(50) to node 15
- Node 44: returns node 58
- Node 50 updates its successor to 58



31

Joining Operation

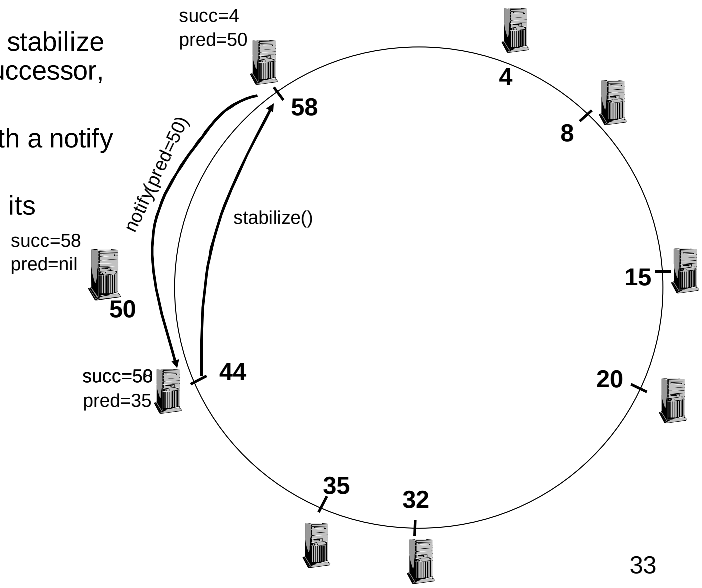
- Node 50: send stabilize() to node 58
- Node 58:
  - update predecessor to 50
  - send notify() back



32

Joining Operation (cont'd)

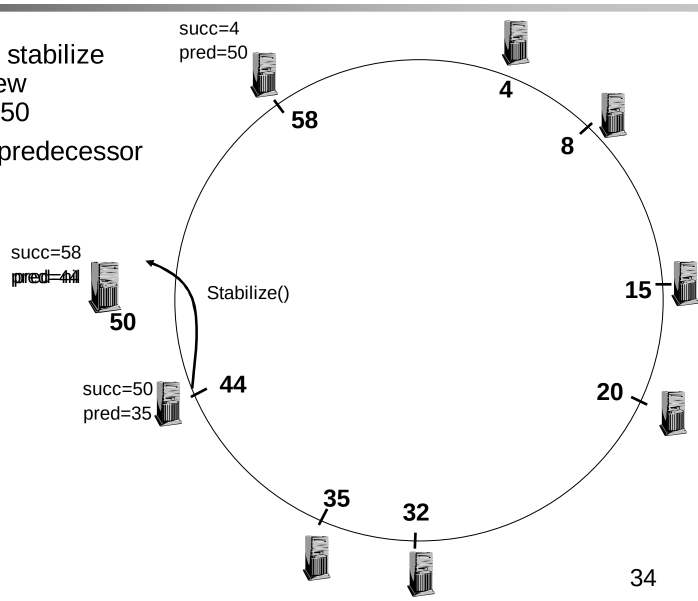
- Node 44 sends a stabilize message to its successor, node 58
- Node 58 reply with a notify message
- Node 44 updates its successor to 50



33

Joining Operation (cont'd)

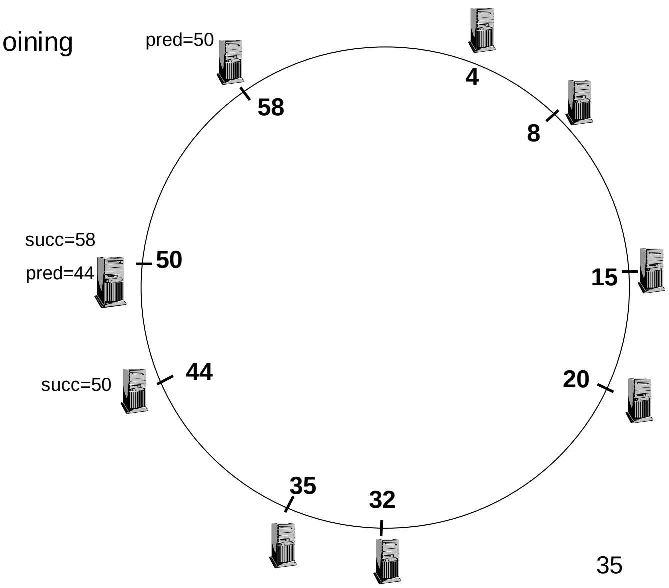
- Node 44 sends a stabilize message to its new successor, node 50
- Node 50 sets its predecessor to node 44



34

Joining Operation (cont'd)

- This completes the joining operation!



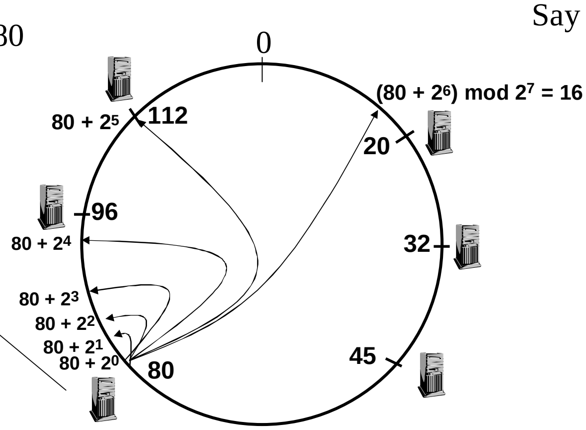
35

Achieving Efficiency: *finger tables*

Say $m=7$

Finger Table at 80

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



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

36

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

37

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

38

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)

39

One Other Papers

- Krishna Gummadi et al, "The Impact of DHT Routing Geometry on Resilience and Proximity", SIGCOMM'03

40

Motivation

- New DHTs constantly proposed
  - CAN, Chord, Pastry, Tapestry, Plaxton, Viceroy, Kademlia, Skipnet, Symphony, Koorde, Apocrypha, Land, ORDI …
- Each is extensively analyzed but in isolation
- Each DHT has many algorithmic details making it difficult to compare

Goals:

a) *Separate fundamental design choices from algorithmic details*

b) *Understand their affect reliability and efficiency*

41

Our approach: Component-based analysis

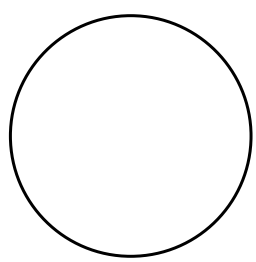
- Break DHT design into independent components
- Analyze impact of each component choice separately
  - compare with black-box analysis:
    - benchmark each DHT implementation
    - rankings of existing DHTs vs. hints on better designs
- Two types of components
  - *Routing-level* : neighbor & route selection
  - *System-level* : caching, replication, querying policy etc.

42

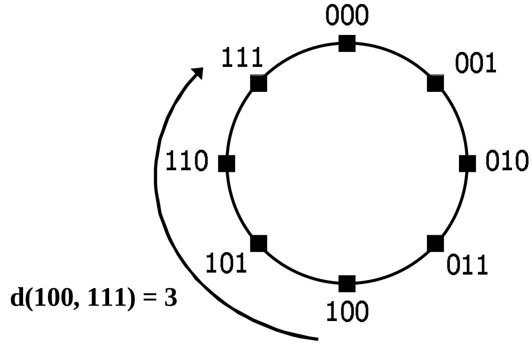
Three aspects of a DHT design

1) **Geometry**: smallest network graph that ensures correct routing/lookup in the DHT
   - Tree, Hypercube, Ring, Butterfly, Debruijn
2) **Distance function**: captures a geometric structure
   - d(id1, id2) for any two node identifiers
3) **Algorithm**: rules for selecting neighbors and routes using the distance function

Chord DHT has Ring Geometry



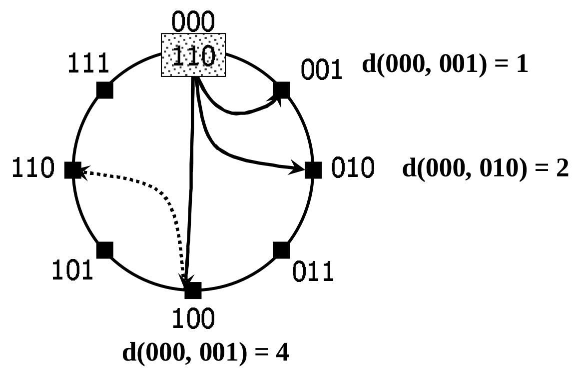
Chord Distance **function captures Ring**



- Nodes are points on a clock-wise Ring
- **d(id1, id2) = length of clock-wise arc between ids**
  = (id2 – id1) mod N

45

Chord Neighbor and Route selection Algorithms



- Neighbor selection: **i^th^** neighbor at **2^i^** distance
- Route selection: pick neighbor closest to destination

46

One Geometry, Many Algorithms

- *Algorithm* : exact rules for selecting neighbors, routes
  - Chord, CAN, PRR, Tapestry, Pastry etc.
  - inspired by geometric structures like Ring, Hyper-cube, Tree
- *Geometry* : an algorithm's underlying structure
  - *Distance* function is the formal representation of Geometry
  - Chord, Symphony => Ring
  - many algorithms can have same geometry

Why is Geometry important?

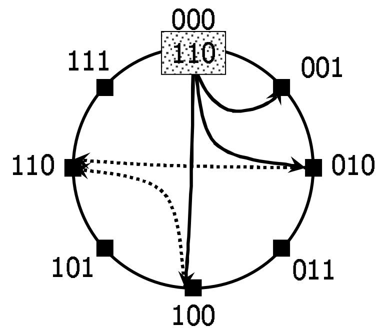
47

Insight:
Geometry => Flexibility => Performance

- Geometry captures *flexibility* in selecting algorithms
- Flexibility is important for routing performance
  - Flexibility in selecting routes leads to shorter, reliable paths
  - Flexibility in selecting neighbors leads to shorter paths

48

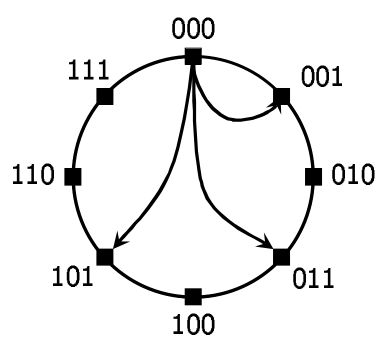
Route selection flexibility allowed by Ring Geometry



- Chord algorithm picks neighbor closest to destination
- A different algorithm picks the best of alternate paths

49

Neighbor selection flexibility allowed by Ring Geometry



- Chord algorithm picks **i^{th}** neighbor at **2^i** distance
- A different algorithm picks **i^{th}** neighbor from **$[2^i , 2^{i+1})$**

50

Geometries we compare

Geometry	Algorithm
Ring	Chord, Symphony
Hypercube	CAN
Tree	Plaxton
Hybrid = Tree + Ring	Tapestry, Pastry
XOR d(id1, id2) = id1 XOR id2	Kademlia

51

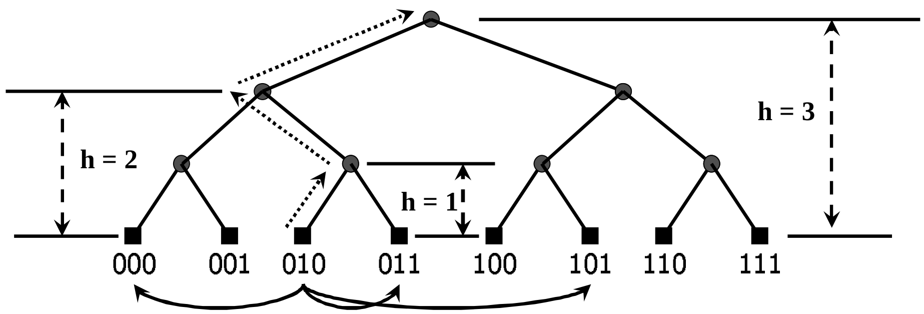
Metrics for flexibilities

- **FNS**: Flexibility in Neighbor Selection
  = number of node choices for a neighbor
- **FRS**: Flexibility in Route Selection
  = avg. number of next-hop choices for all destinations

- Constraints for neighbors and routes
  - select neighbors to have paths of **O(logN)**
  - select routes so that each hop is closer to destination

52

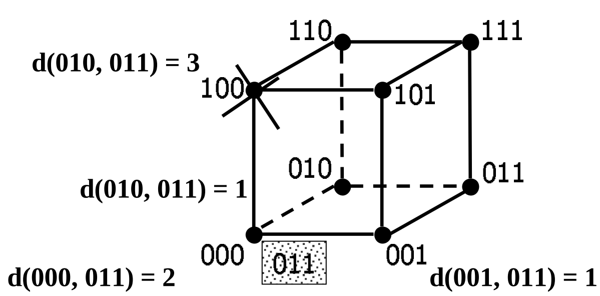
Flexibility in neighbor selection (FNS) for Tree



- **logN** neighbors in sub-trees of varying heights
- **FNS = 2^{i-1}** for **i**th neighbor of a node

53

Flexibility in route selection (FRS) for Hypercube



- Routing to next hop fixes one bit
- **FRS =Avg. (#bits destination differs in)=logN/2**

54

Summary of our flexibility analysis

Flexibility	Ordering of Geometries
Neighbors (FNS)	**Hypercube << Tree, XOR, Ring, Hybrid** (logN) (2^{i-1})
Routes (FRS)	**Tree << XOR, Hybrid < Hypercube < Ring** (1) (logN/2) (logN/2) (logN)

How relevant is flexibility for DHT routing performance?

55

Analysis of *Static Resilience*

Two aspects of robust routing

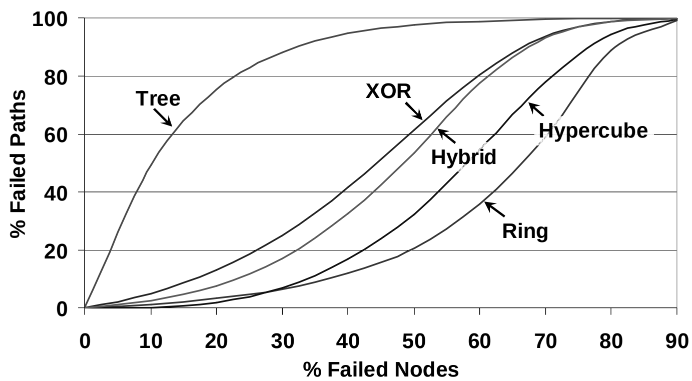
- *Dynamic Recovery* : how quickly routing state is recovered after failures
- *Static Resilience* : how well the network routes before recovery finishes
  - captures how quickly recovery algorithms need to work
  - depends on FRS

Evaluation:

- Fail a fraction of nodes, without recovering any state
- Metric: **% Paths Failed**

56

Does flexibility affect Static Resilience?



Tree << XOR ≈ Hybrid < Hypercube < Ring

Flexibility in Route Selection matters for Static Resilience

57

Analysis of *Overlay Path Latency*

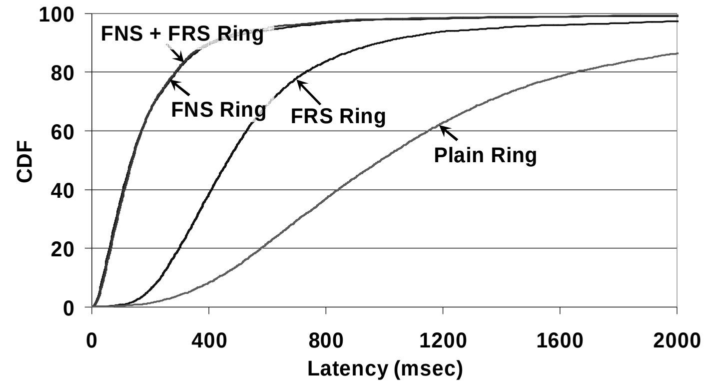
- Goal: Minimize end-to-end overlay path latency
  - not just the number of hops
- Both FNS and FRS can reduce latency
  - Tree has FNS, Hypercube has FRS, Ring & XOR have both

Evaluation:

- Using Internet latency distributions (see paper)

58

Which is more effective, FNS or FRS?

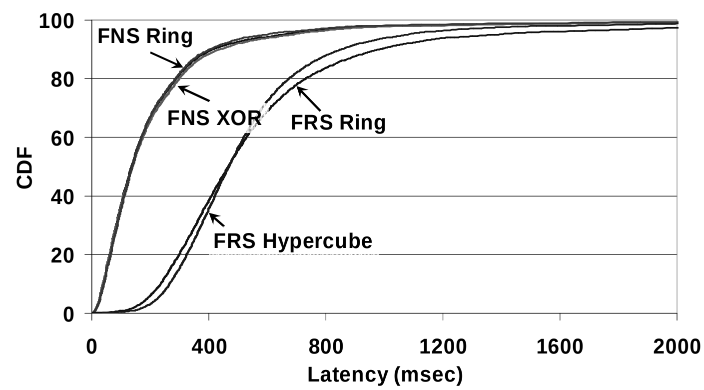


Plain << FRS << FNS ≈ FNS+FRS

Neighbor Selection is much better than Route Selection

59

Does Geometry affect performance of FNS or FRS?



No, performance of FNS/FRS is independent of Geometry

A Geometry's support for neighbor selection is crucial

60

Summary of results

- FRS matters for Static Resilience
  - Ring has the best resilience
- Both FNS and FRS reduce Overlay Path Latency
- But, FNS is far more important than FRS
  - Ring, Hybrid, Tree and XOR have high FNS

61

Conclusions

- Routing Geometry is a fundamental design choice
  - Geometry determines flexibility
  - Flexibility improves resilience and proximity
- Ring has the greatest flexibility

62