

600.413 Topics in P2P Network Systems

Andreas Terzis
terzis@cs.jhu.edu

4/8/03

1



This week

■ Chord

- Presentation from Robert Morris at SIGCOMM 2001

■ CAN

- Presentation from Sylvia Ratnasamy at SIGCOMM 2001

4/8/03

2

Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications

Robert Morris
Ion Stoica, David Karger,
M. Frans Kaashoek, Hari Balakrishnan

MIT and Berkeley

4/8/03

3

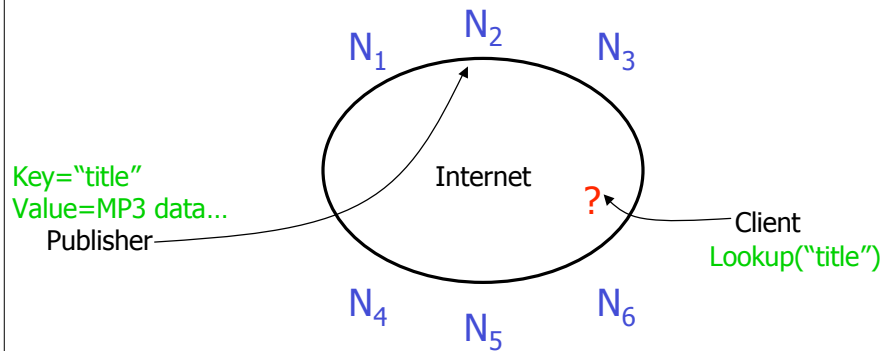
A peer-to-peer storage problem

- 1000 scattered music enthusiasts
- Willing to store and serve replicas
- How do you find the data?

4/8/03

4

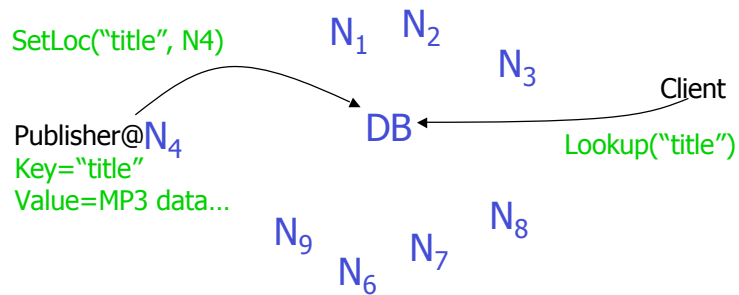
The lookup problem



4/8/03

5

Centralized lookup (Napster)

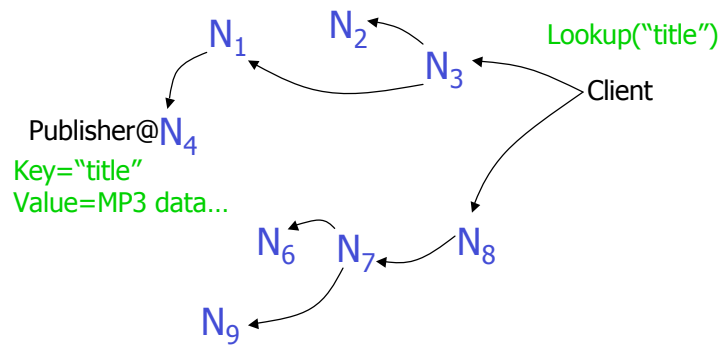


Simple, but $O(N)$ state and a single point of failure

4/8/03

6

Flooded queries (Gnutella)

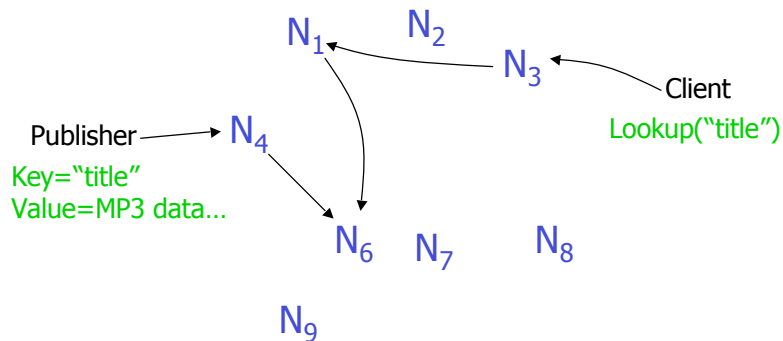


Robust, but worst case $O(N)$ messages per lookup

4/8/03

7

Routed queries (Freenet, Chord, etc.)



4/8/03

8

Routing challenges

- Define a useful key nearness metric
- Keep the hop count small
- Keep the tables small
- Stay robust despite rapid change

- Freenet: emphasizes anonymity
- Chord: emphasizes efficiency and simplicity

4/8/03

9

Chord properties

- Efficient: $O(\log(N))$ messages per lookup
 - N is the total number of servers
- Scalable: $O(\log(N))$ state per node
- Robust: survives massive failures
- Proofs are in paper / tech report
 - Assuming no malicious participants

4/8/03

10

Chord overview

- Provides peer-to-peer hash lookup:
 - Lookup(key) \rightarrow IP address
 - Chord does not store the data
- How does Chord route lookups?
- How does Chord maintain routing tables?

4/8/03

11

Chord IDs

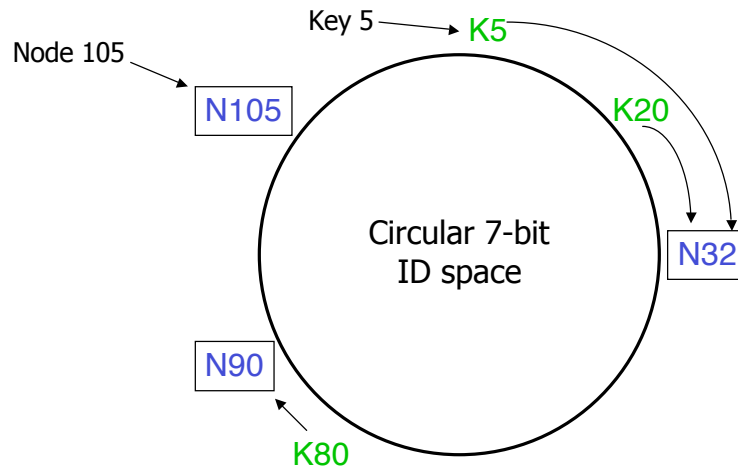
- Key identifier = SHA-1(key)
- Node identifier = SHA-1(IP address)
- Both are uniformly distributed
- Both exist in the same ID space

- How to map key IDs to node IDs?

4/8/03

12

Consistent hashing [Karger 97]

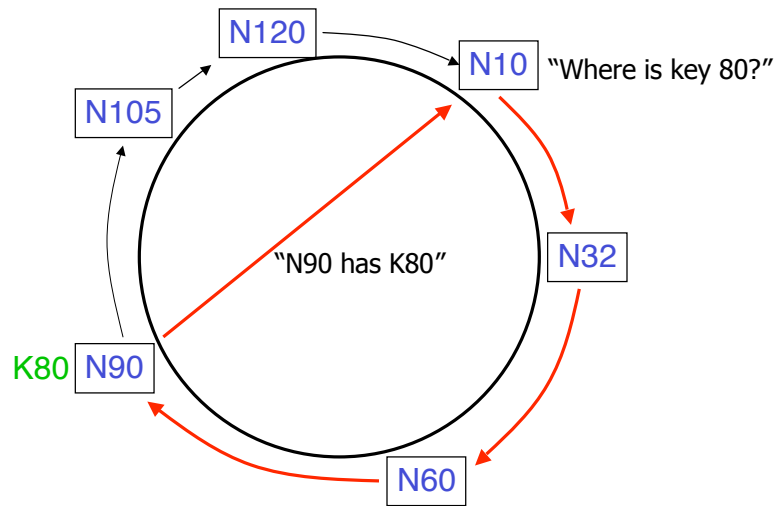


A key is stored at its **successor**: node with next higher ID

4/8/03

13

Basic lookup



4/8/03

14

Simple lookup algorithm

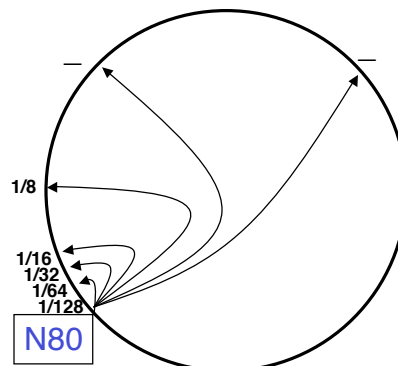
```
Lookup(my-id, key-id)
  n = my successor
  if my-id < n < key-id
    call Lookup(id) on node n // next hop
  else
    return my successor // done
```

- Correctness depends only on successors

4/8/03

15

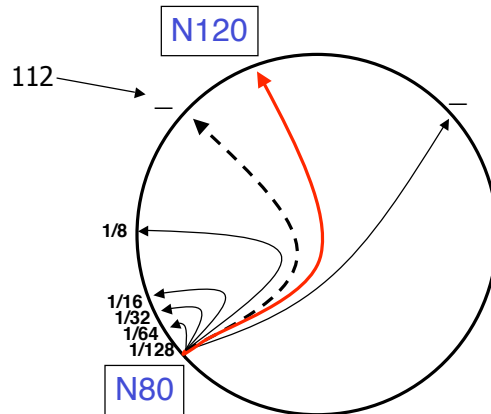
“Finger table” allows $\log(N)$ -time lookups



4/8/03

16

Finger i points to successor of $n+2^i$



4/8/03

17

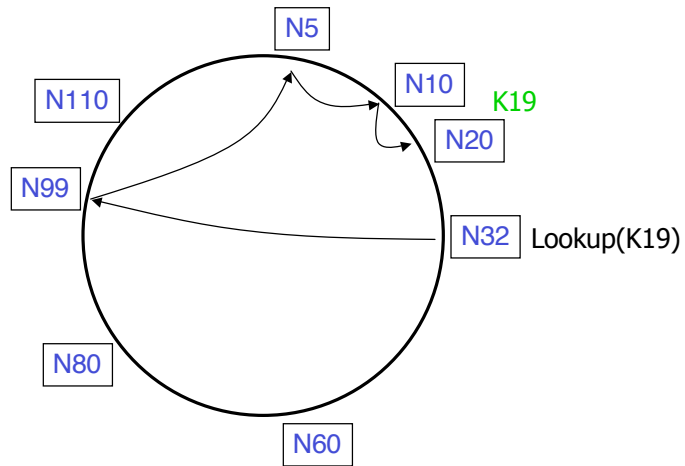
Lookup with fingers

```
Find_successor(my-id, key-id)
  if my-id < key-id < successor
    return my successor      // done
  else
    look in local finger table for
      highest node n s.t. my-id < n < key-id
    call Find_successor(id) on node n // next hop
```

4/8/03

18

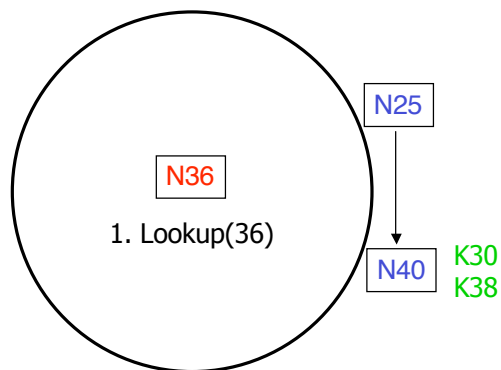
Lookups take $O(\log(N))$ hops



4/8/03

19

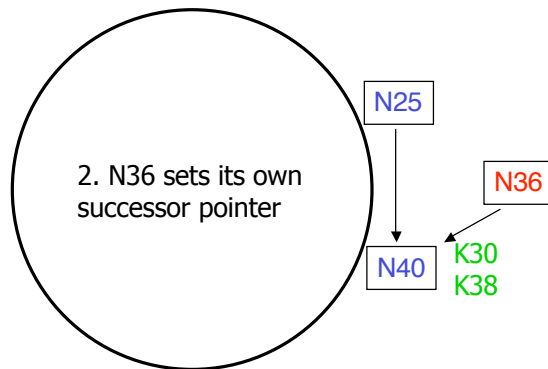
Joining: linked list insert



4/8/03

20

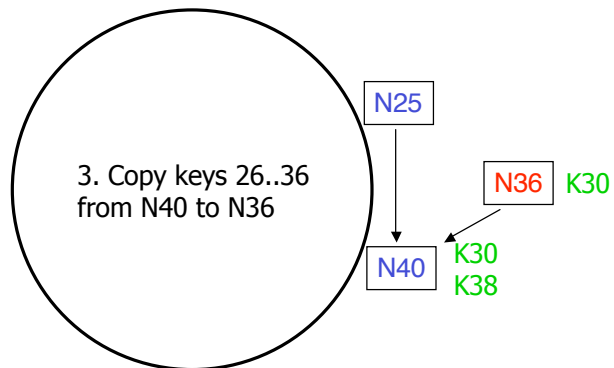
Join (2)



4/8/03

21

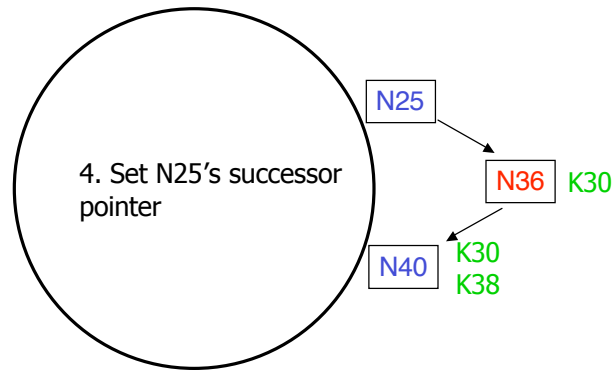
Join (3)



4/8/03

22

Join (4)

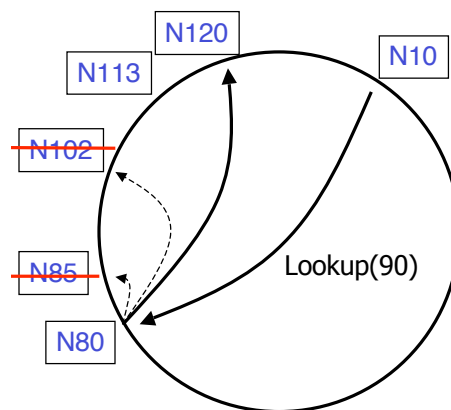


Update finger pointers in the background
Correct successors produce correct lookups

4/8/03

23

Failures might cause incorrect lookup



N80 doesn't know correct successor, so incorrect lookup

4/8/03

24

Solution: successor lists

- Each node knows r immediate successors
- After failure, will know first live successor
- Correct successors guarantee correct lookups

- Guarantee is with some probability

4/8/03

25

Choosing the successor list length

- Assume 1/2 of nodes fail
- $P(\text{successor list all dead}) = (1/2)^r$
 - I.e. $P(\text{this node breaks the Chord ring})$
 - Depends on independent failure
- $P(\text{no broken nodes}) = (1 - (1/2)^r)^N$
 - $r = 2\log(N)$ makes prob. = $1 - 1/N$

4/8/03

26

Lookup with fault tolerance

```
Find_successor(my-id, key-id)
  if my-id < key-id < successor
    return successor
  else
    look in local finger table and successor-list
    for highest node n s.t. my-id < n < key-id
    if n exists
      call Find_successor(id) on node n // next hop
      if call failed,
        remove n from finger table
        return Find_successor(my-id, key-id)
```

4/8/03

27

Chord status

- Working implementation as part of CFS
- Chord library: 3,000 lines of C++
- Deployed in small Internet testbed
- Includes:
 - Correct concurrent join/fail
 - Proximity-based routing for low delay
 - Load control for heterogeneous nodes
 - Resistance to spoofed node IDs

4/8/03

28

Experimental overview

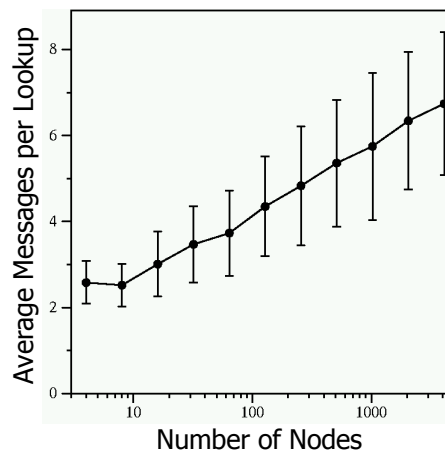
- Quick lookup in large systems
- Low variation in lookup costs
- Robust despite massive failure
- See paper for more results

Experiments confirm theoretical results

4/8/03

29

Chord lookup cost is $O(\log N)$



Constant is 1/2

4/8/03

30

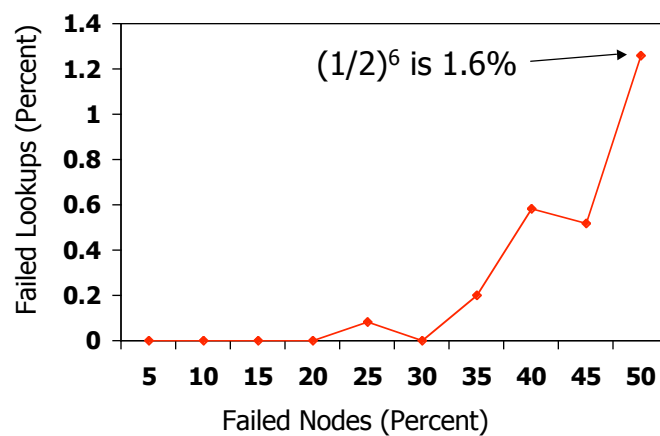
Failure experimental setup

- Start 1,000 CFS/Chord servers
 - Successor list has 20 entries
- Wait until they stabilize
- Insert 1,000 key/value pairs
 - Five replicas of each
- Stop X% of the servers
- Immediately perform 1,000 lookups

4/8/03

31

Massive failures have little impact



4/8/03

32

Chord Summary

- Chord provides peer-to-peer hash lookup
- Efficient: $O(\log(n))$ messages per lookup
- Robust as nodes fail and join
- Good primitive for peer-to-peer systems

<http://www.pdos.lcs.mit.edu/chord>

4/8/03

33

Chord Evaluation

- Deterministic algorithm
 - $O(\log N)$ lookup
 - $O(\log N)$ per-node state
- Performance when nodes enter/leave network?
- Network Spread?
- Performance against adversarial opponents?

4/8/03

34

A Scalable, Content-Addressable Network

Sylvia Ratnasamy^{1,2}, Paul Francis³, Mark Handley¹,
Richard Karp^{1,2}, Scott Shenker¹

¹
ACIRI
4/8/03

²
U.C. Berkeley

³
Tahoe
Networks³⁵

Outline

- Introduction
- Design
- Evaluation
- Ongoing Work

4/8/03

36

Internet-scale hash tables

- Hash tables
 - essential building block in software systems
- Internet-scale distributed hash tables
 - equally valuable to large-scale distributed systems?

4/8/03

37

Internet-scale hash tables

- Hash tables
 - essential building block in software systems
- Internet-scale distributed hash tables
 - equally valuable to large-scale distributed systems?
 - peer-to-peer systems
 - Napster, Gnutella, Groove, FreeNet, MojoNation...
 - large-scale storage management systems
 - Publius, OceanStore, PAST, Farsite, CFS ...
 - mirroring on the Web

4/8/03

38

Content-Addressable Network (CAN)

- CAN: Internet-scale hash table
- Interface
 - insert(key,value)
 - value = retrieve(key)

4/8/03

39

Content-Addressable Network (CAN)

- CAN: Internet-scale hash table
- Interface
 - insert(key,value)
 - value = retrieve(key)
- Properties
 - scalable
 - operationally simple
 - good performance

4/8/03

40

Content-Addressable Network (CAN)

- CAN: Internet-scale hash table
- Interface
 - insert(key,value)
 - value = retrieve(key)
- Properties
 - scalable
 - operationally simple
 - good performance
- Related systems:
Chord/Pastry/Tapestry/Buzz/Plaxton ...

4/8/03

41

Problem Scope

- Design a system that provides the interface
 - scalability
 - robustness
 - performance
 - security
- Application-specific, higher level primitives
 - keyword searching
 - mutable content
 - anonymity

4/8/03

42

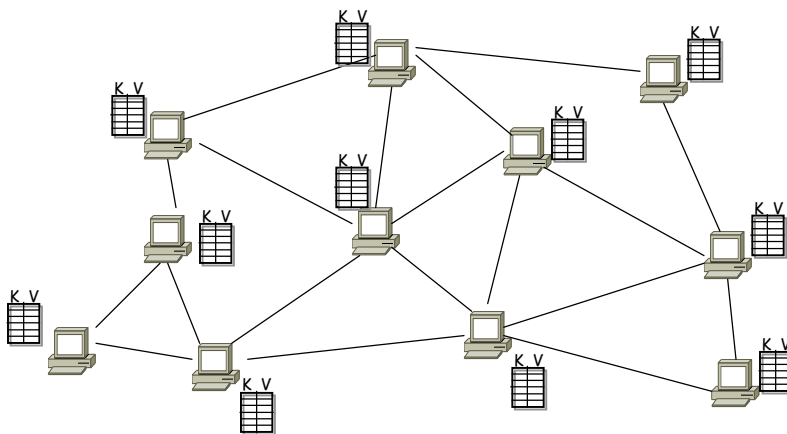
Outline

- Introduction
- **Design**
- Evaluation
- Ongoing Work

4/8/03

43

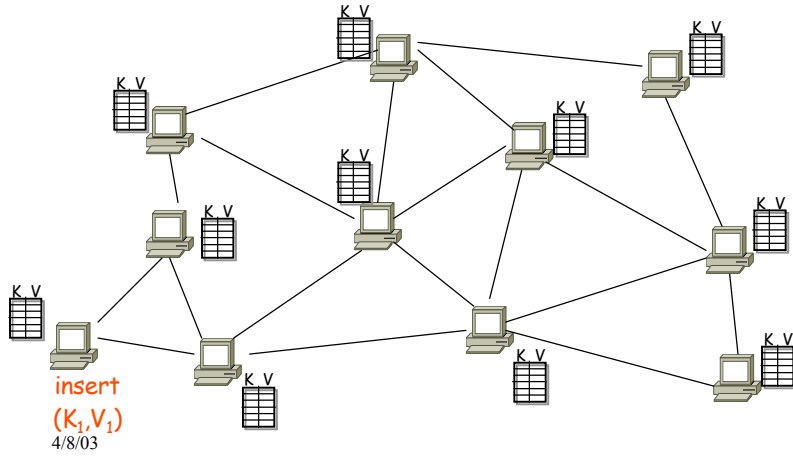
CAN: basic idea



4/8/03

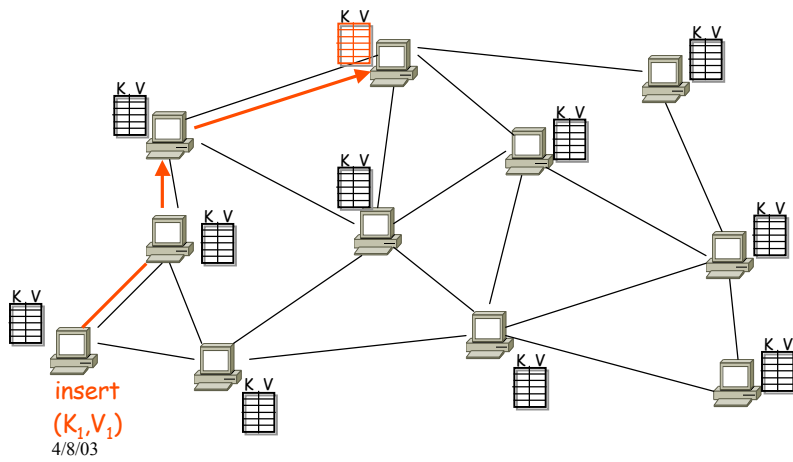
44

CAN: basic idea



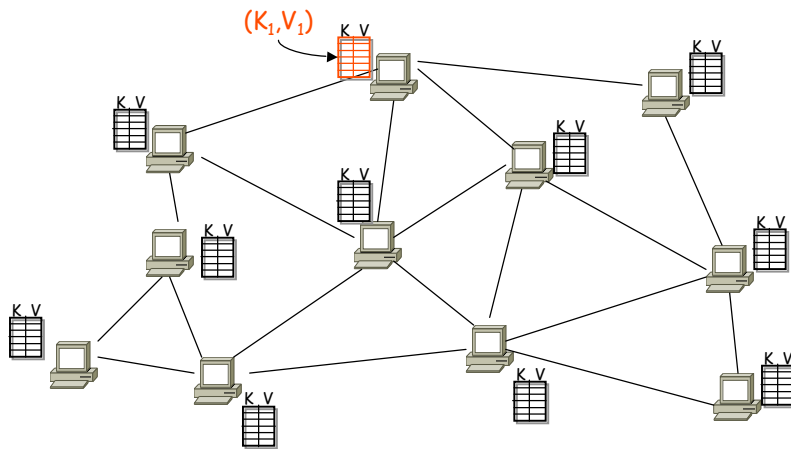
45

CAN: basic idea



46

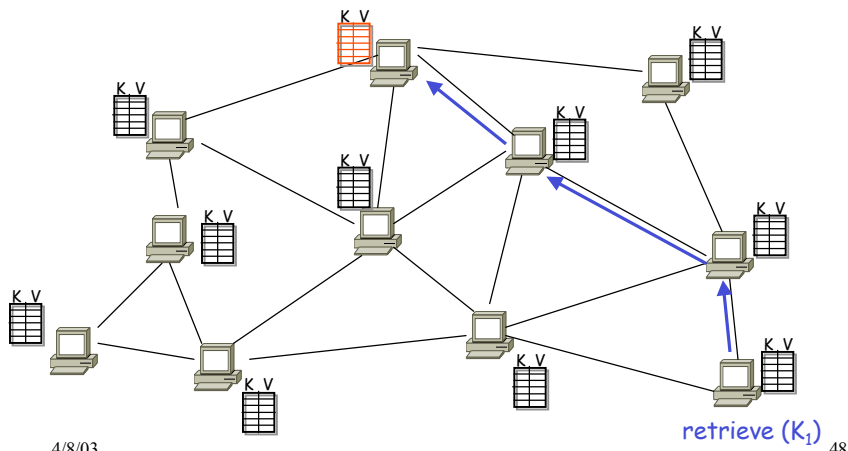
CAN: basic idea



4/8/03

47

CAN: basic idea



4/8/03

48

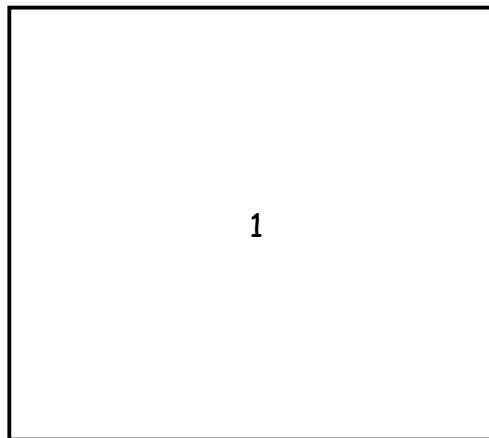
CAN: solution

- virtual Cartesian coordinate space
- entire space is partitioned amongst all the nodes
 - every node "owns" a zone in the overall space
- abstraction
 - can store data at "points" in the space
 - can route from one "point" to another
- point = node that owns the enclosing zone

4/8/03

49

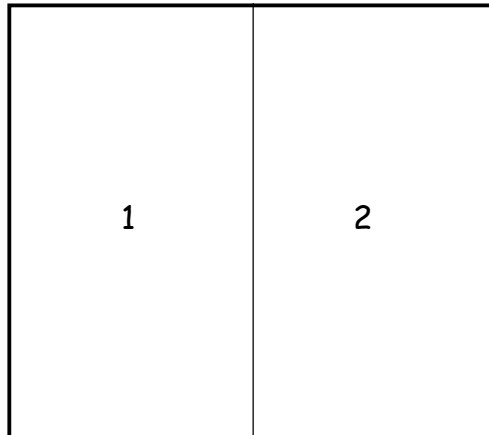
CAN: simple example



4/8/03

50

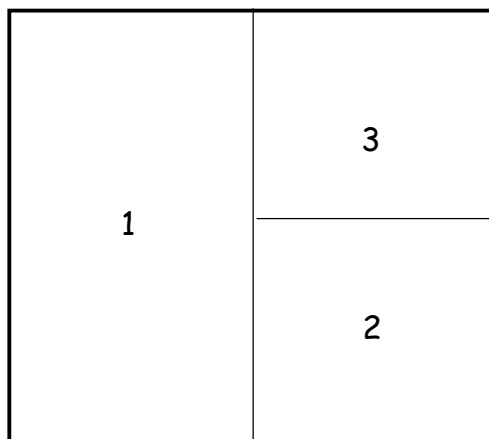
CAN: simple example



4/8/03

51

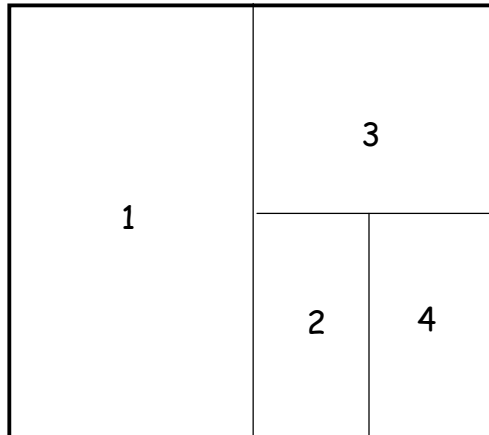
CAN: simple example



4/8/03

52

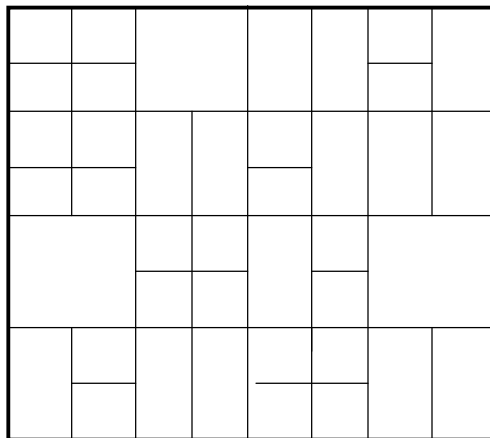
CAN: simple example



4/8/03

53

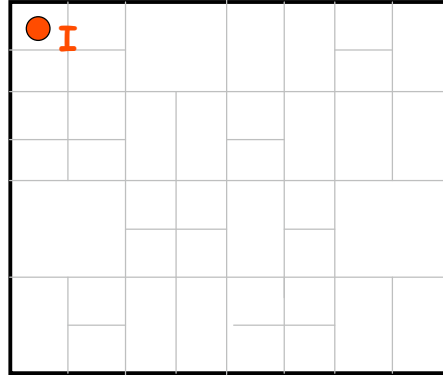
CAN: simple example



4/8/03

54

CAN: simple example

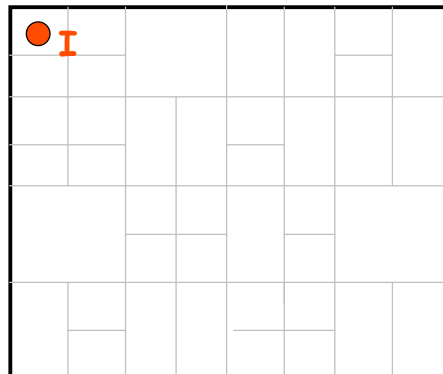


4/8/03

55

CAN: simple example

node I::insert(K,V)



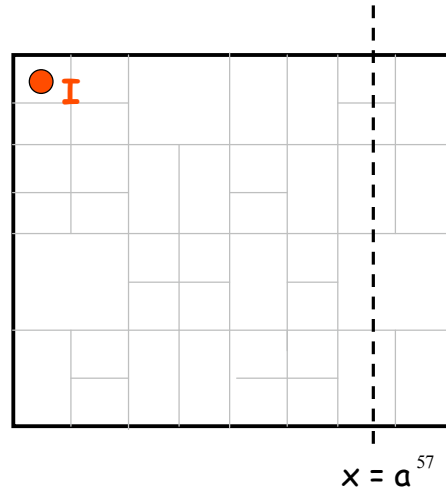
4/8/03

56

CAN: simple example

node I::insert(K,V)

(1) $a = h_x(K)$



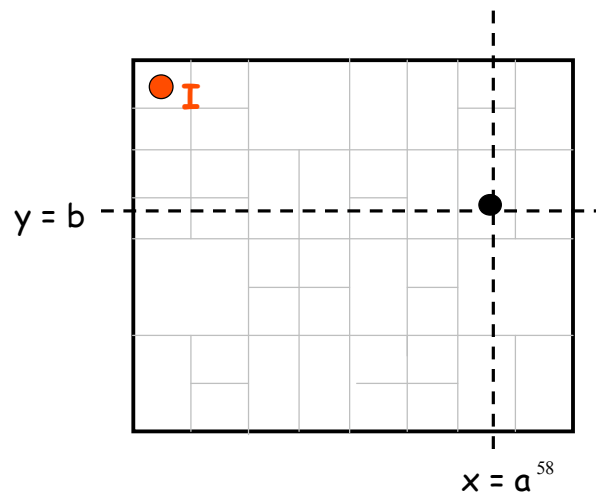
4/8/03

CAN: simple example

node I::insert(K,V)

(1) $a = h_x(K)$

$b = h_y(K)$



4/8/03

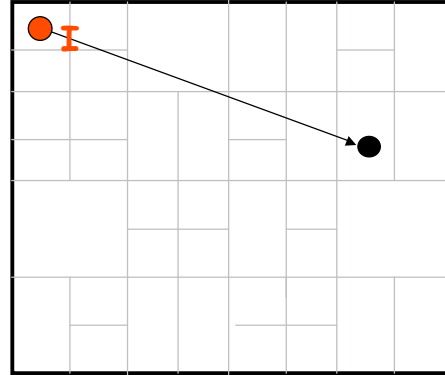
CAN: simple example

node I::insert(K,V)

(1) $a = h_x(K)$

$b = h_y(K)$

(2) route(K,V) \rightarrow (a,b)



4/8/03

59

CAN: simple example

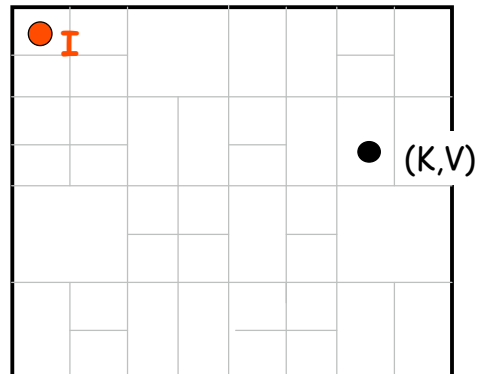
node I::insert(K,V)

(1) $a = h_x(K)$

$b = h_y(K)$

(2) route(K,V) \rightarrow (a,b)

(3) (a,b) stores (K,V)



4/8/03

60

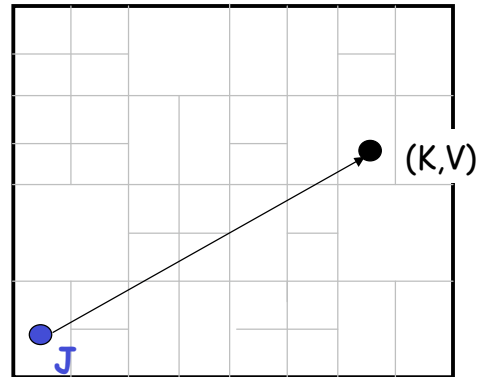
CAN: simple example

node J::retrieve(K)

(1) $a = h_x(K)$

$b = h_y(K)$

(2) route "retrieve(K)" to (a,b)



4/8/03

61

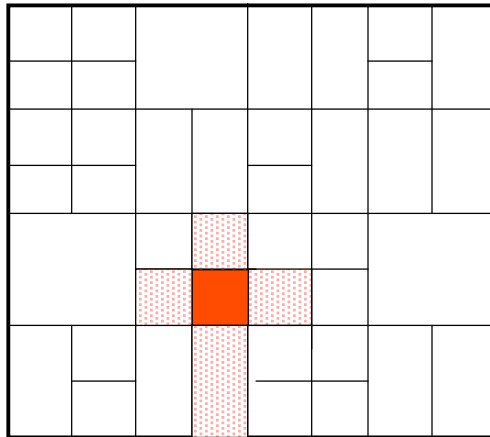
CAN

Data stored in the CAN is addressed by name (i.e. key), not location (i.e. IP address)

4/8/03

62

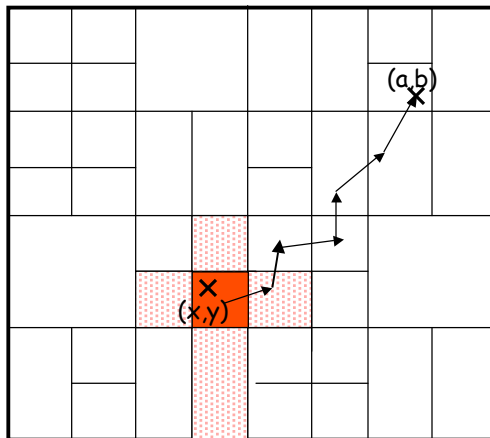
CAN: routing table



4/8/03

63

CAN: routing



4/8/03

64

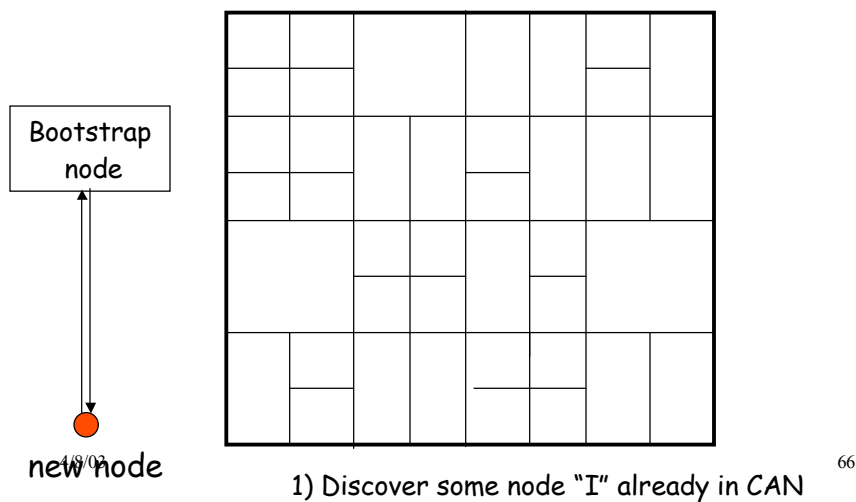
CAN: routing

A node only maintains state for its immediate neighboring nodes

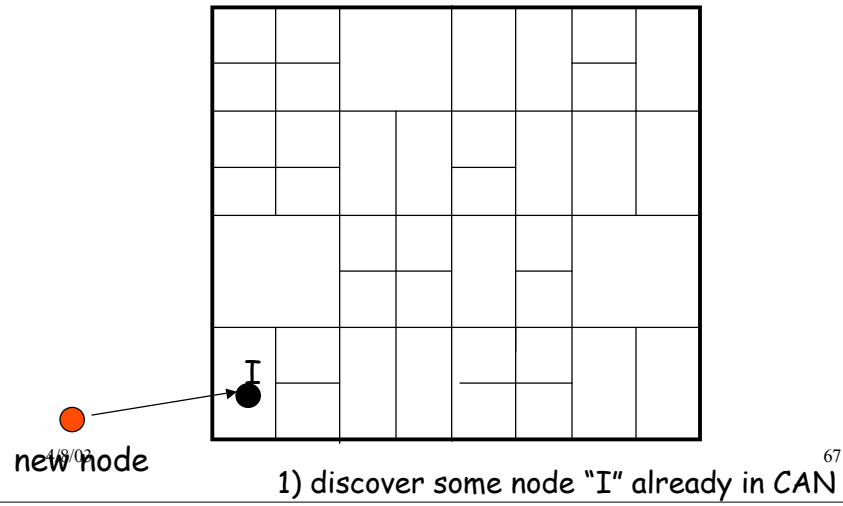
4/8/03

65

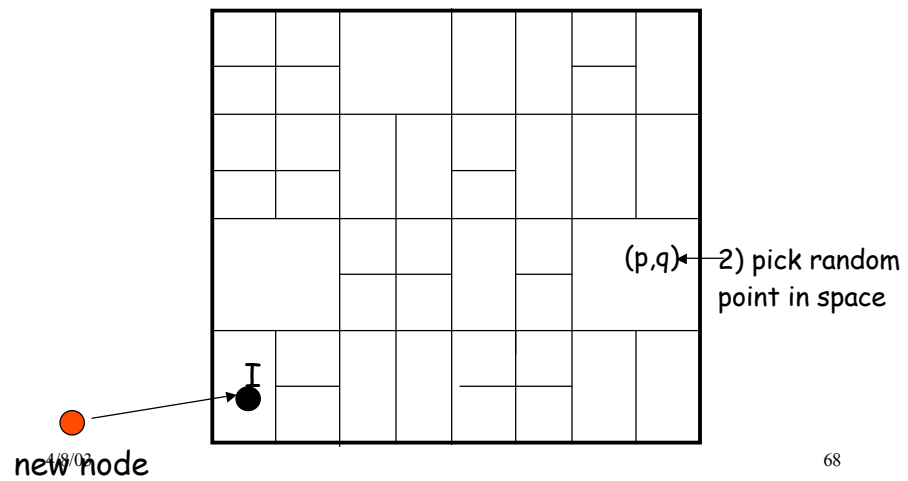
CAN: node insertion



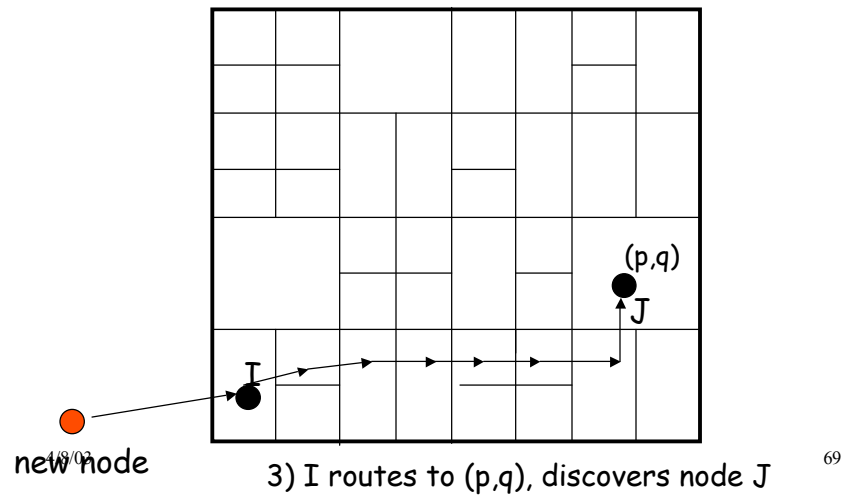
CAN: node insertion



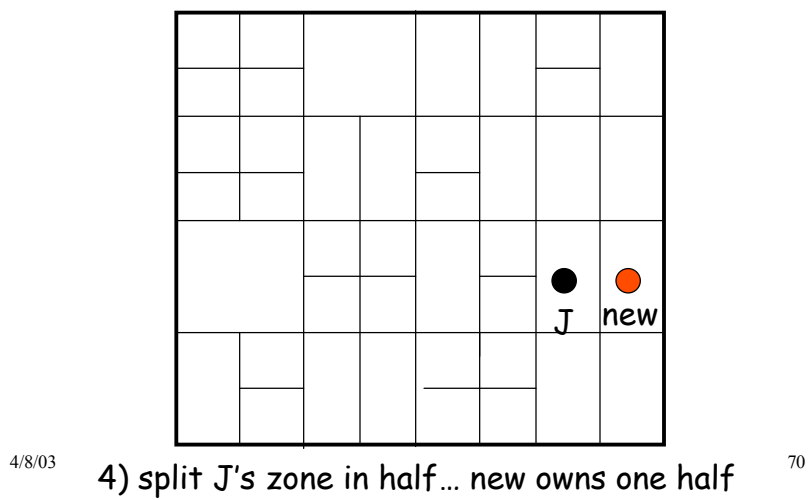
CAN: node insertion



CAN: node insertion



CAN: node insertion



CAN: node insertion

Inserting a new node affects only a single other node and its immediate neighbors

4/8/03

71

CAN: node failures

- Need to repair the space
 - recover database
 - soft-state updates
 - use replication, rebuild database from replicas
 - repair routing
 - takeover algorithm

4/8/03

72

CAN: takeover algorithm

- 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

4/8/03

73

CAN: node failures

Only the failed node's immediate neighbors are required for recovery

4/8/03

74

Design recap

- **Basic CAN**
 - completely distributed
 - self-organizing
 - nodes only maintain state for their immediate neighbors
- **Additional design features**
 - multiple, independent spaces (realities)
 - background load balancing algorithm
 - simple heuristics to improve performance

4/8/03

75

Outline

- Introduction
- Design
- **Evaluation**
- Ongoing Work

4/8/03

76

Evaluation

- Scalability
- Low-latency
- Load balancing
- Robustness

4/8/03

77

CAN: scalability

- For a uniformly partitioned space with n nodes and d dimensions
 - per node, number of neighbors is $2d$
 - average routing path is $(dn^{1/d})/4$ hops
 - simulations show that the above results hold in practice
- Can scale the network without increasing per-node state
- Chord/Plaxton/Tapestry/Buzz
 - $\log(n)$ nbrs with $\log(n)$ hops

4/8/03

78

CAN: low-latency

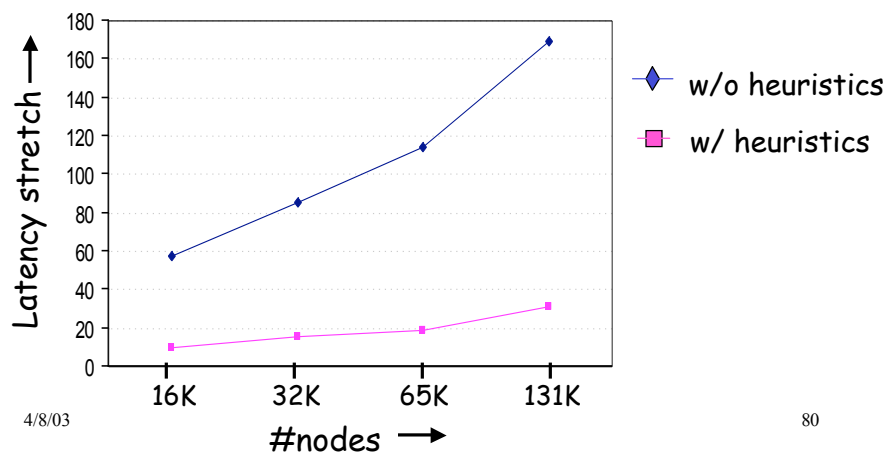
- Problem
 - latency stretch = $\frac{\text{(CAN routing delay)}}{\text{(IP routing delay)}}$
 - application-level routing may lead to high stretch
- Solution
 - increase dimensions
 - heuristics
 - RTT-weighted routing
 - multiple nodes per zone (peer nodes)
 - deterministically replicate entries

4/8/03

79

CAN: low-latency

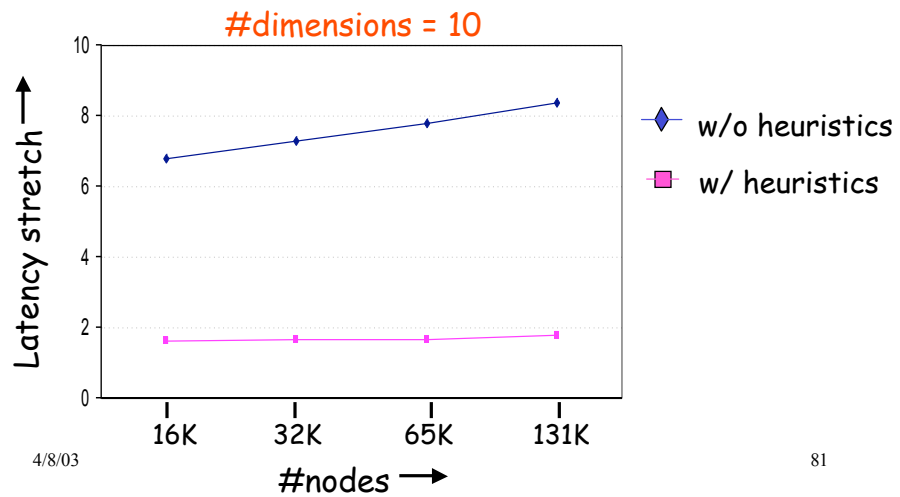
#dimensions = 2



4/8/03

80

CAN: low-latency



CAN: load balancing

- Two pieces
 - Dealing with hot-spots
 - popular (key,value) pairs
 - nodes cache recently requested entries
 - overloaded node replicates popular entries at neighbors
 - Uniform coordinate space partitioning
 - uniformly spread (key,value) entries
 - uniformly spread out routing load

4/8/03

82

Uniform Partitioning

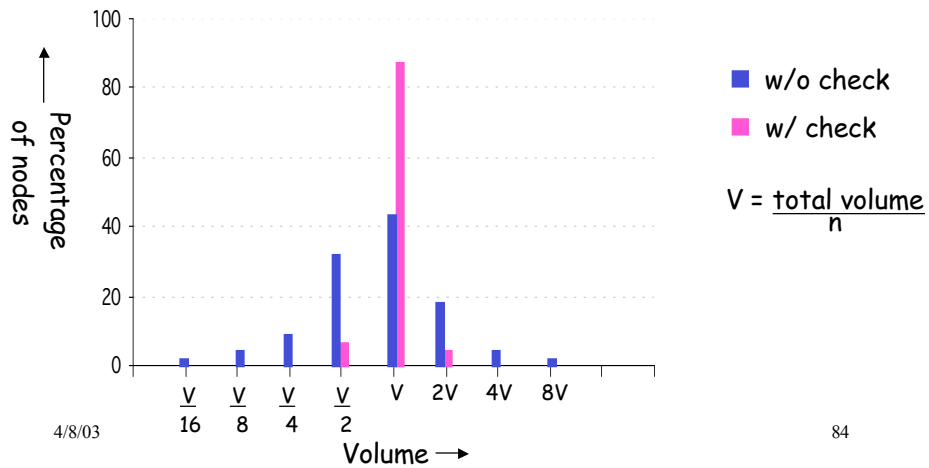
- Added check
 - at join time, pick a zone
 - check neighboring zones
 - pick the largest zone and split that one

4/8/03

83

Uniform Partitioning

65,000 nodes, 3 dimensions



84

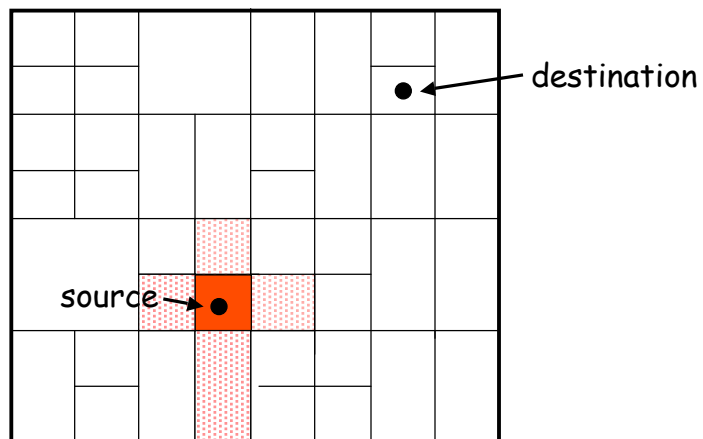
CAN: Robustness

- Completely distributed
 - no single point of failure
- Not exploring database recovery
- Resilience of routing
 - can route around trouble

4/8/03

85

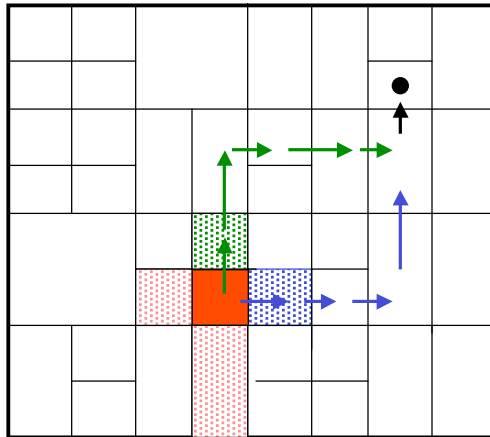
Routing resilience



4/8/03

86

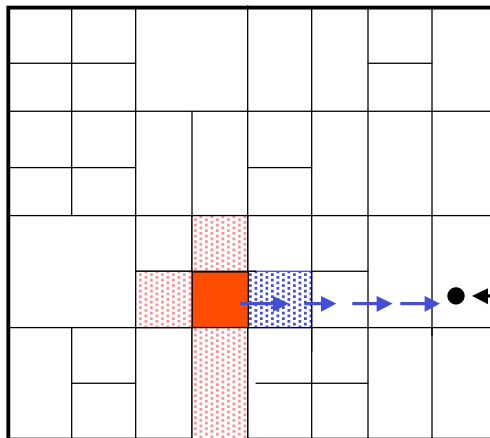
Routing resilience



4/8/03

87

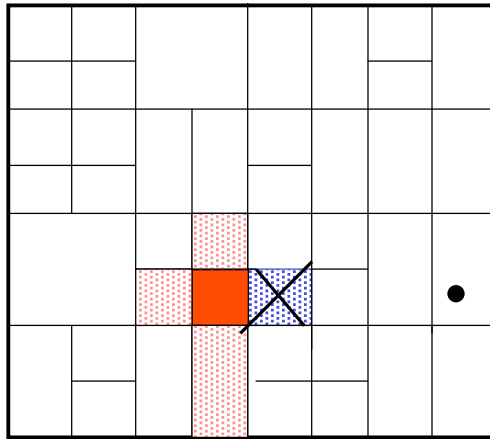
Routing resilience



4/8/03

88

Routing resilience



4/8/03

89

Routing resilience

- Node X::route(D)

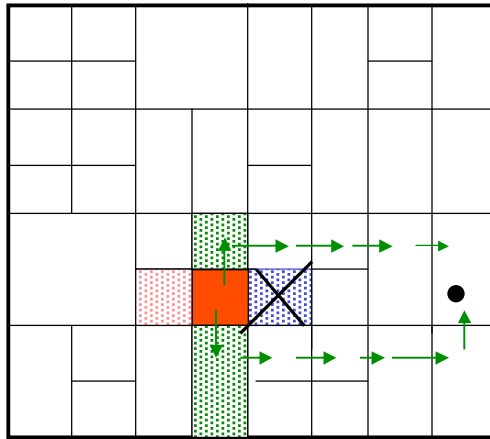
If (X cannot make progress to D)

- check if any neighbor of X can make progress
- if yes, forward message to one such nbr

4/8/03

90

Routing resilience

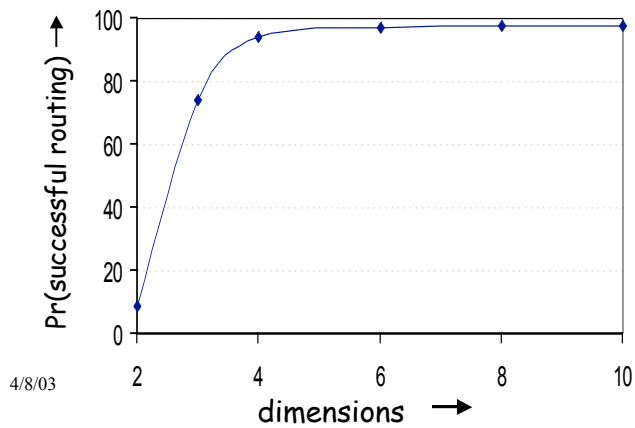


4/8/03

91

Routing resilience

CAN size = 16K nodes
Pr(node failure) = 0.25



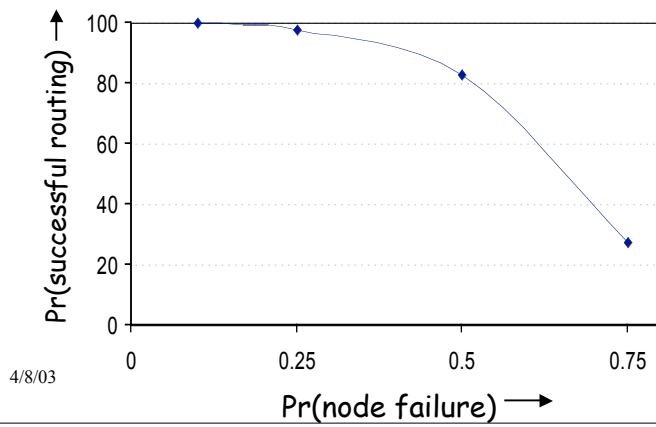
4/8/03

92

Routing resilience

CAN size = 16K nodes

#dimensions = 10



4/8/03

93

Outline

- Introduction
- Design
- Evaluation
- **Ongoing Work**

4/8/03

94

Ongoing Work

- Topologically-sensitive CAN construction
 - distributed binning

4/8/03

95

Distributed Binning

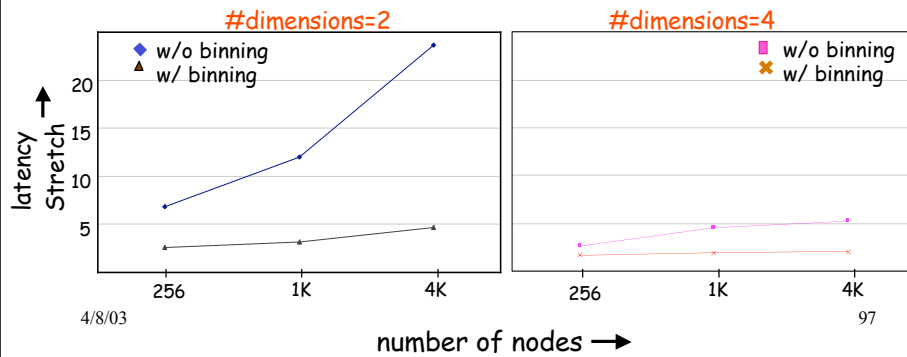
- **Goal**
 - bin nodes such that co-located nodes land in same bin
- **Idea**
 - well known set of landmark machines
 - each CAN node, measures its RTT to each landmark
 - orders the landmarks in order of increasing RTT
- **CAN construction**
 - place nodes from the same bin close together on the CAN

4/8/03

96

Distributed Binning

- 4 Landmarks (placed at 5 hops away from each other)
- naïve partitioning



Ongoing Work (cont'd)

- Topologically-sensitive CAN construction
 - distributed binning
- CAN Security (Petros Maniatis - Stanford)
 - spectrum of attacks
 - appropriate counter-measures

4/8/03

98

Ongoing Work (cont'd)

- *CAN Usage*
 - Application-level Multicast (NGC 2001)
 - Grass-Roots Content Distribution
 - Distributed Databases using CANs
(J.Hellerstein, S.Ratnasamy, S.Shenker, I.Stoica, S.Zhuang)

4/8/03


99

Summary

- *CAN*
 - an Internet-scale hash table
 - potential building block in Internet applications
- Scalability
 - $O(d)$ per-node state
- Low-latency routing
 - simple heuristics help a lot
- Robust
 - decentralized, can route around trouble

4/8/03

100



CAN Evaluation

4/8/03 **101**