600.413 Topics in P2P Network Systems

Andreas Terzis
terzis@cs.jhu.edu

This week

- Chord
  - Presentation from Robert Morris at SIGCOMM 2001
- CAN
  - Presentation from Sylvia Ratnasamy at SIGCOMM 2001

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

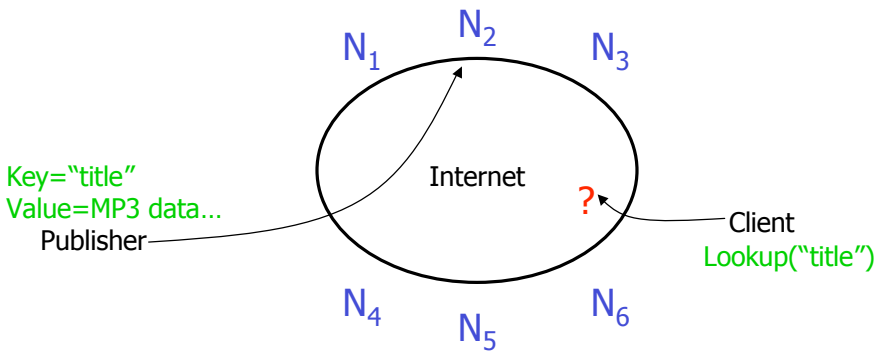
Robert Morris
Ion Stoica, David Karger,
M. Frans Kaashoek, Hari Balakrishnan

MIT and Berkeley

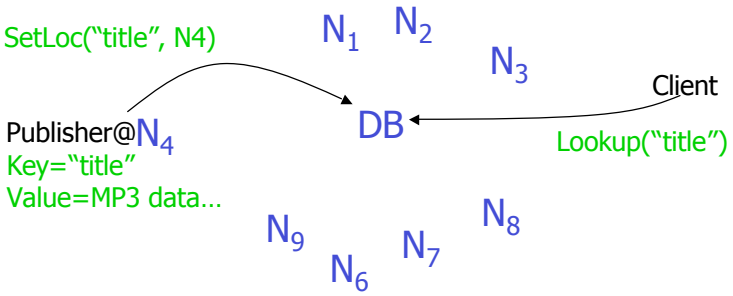
A peer-to-peer storage problem

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

The lookup problem

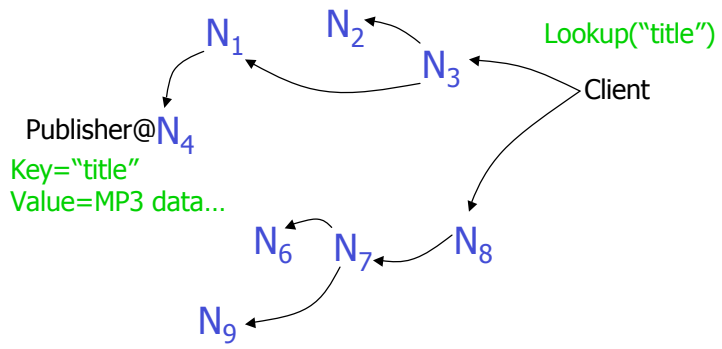


Centralized lookup (Napster)



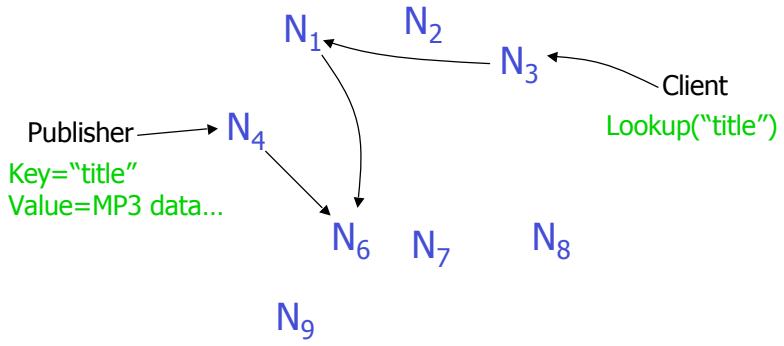
Simple, but O(N) state and a single point of failure

Flooded queries (Gnutella)



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

Routed queries (Freenet, Chord, etc.)



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

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

Chord overview

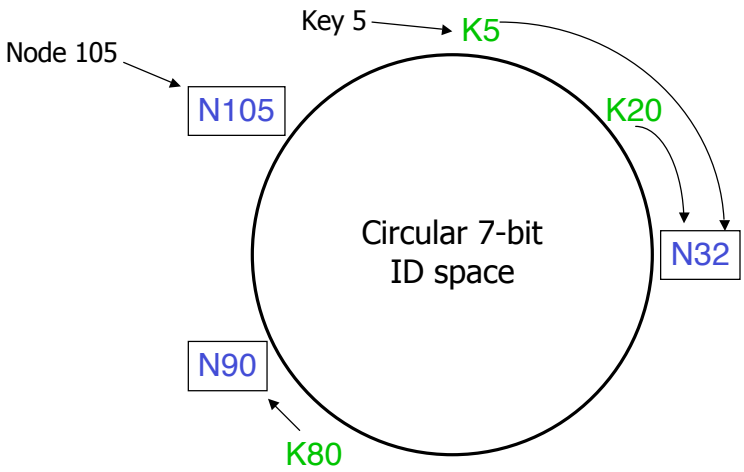
- Provides peer-to-peer hash lookup:
  - Lookup(key) → IP address
  - Chord does not store the data
- How does Chord route lookups?
- How does Chord maintain routing tables?

Chord IDs

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

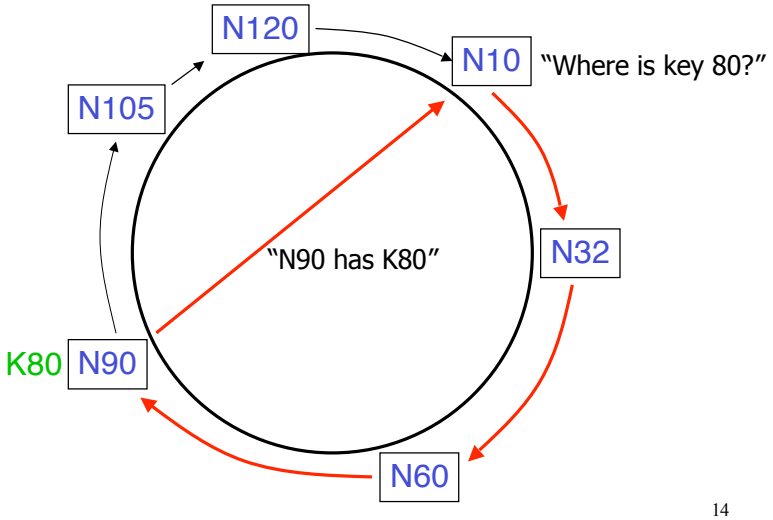
- How to map key IDs to node IDs?

Consistent hashing [Karger 97]



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

Basic lookup

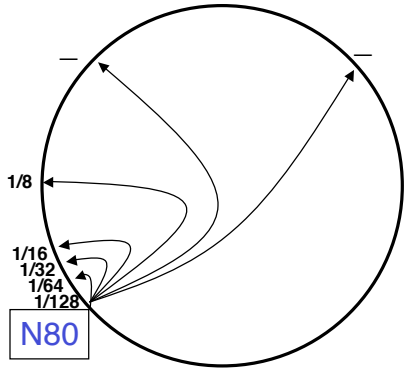


Simple lookup algorithm

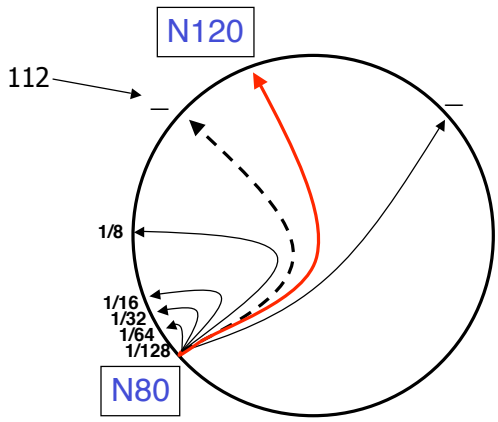
```
Lookup(my-id, key-id)
  n = my successor
  if my-id < n < key-id
      call Lookup(id) on node n   // next hop
  else
      return my successor         // done
```

- Correctness depends only on successors

"Finger table" allows log(N)-time lookups



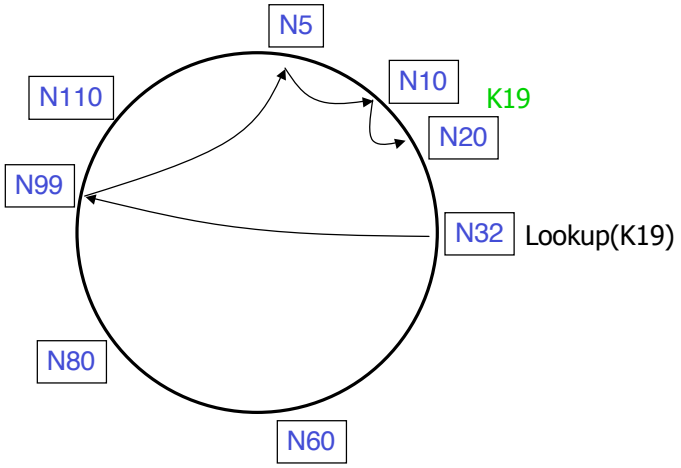
Finger i points to successor of $n+2^i$



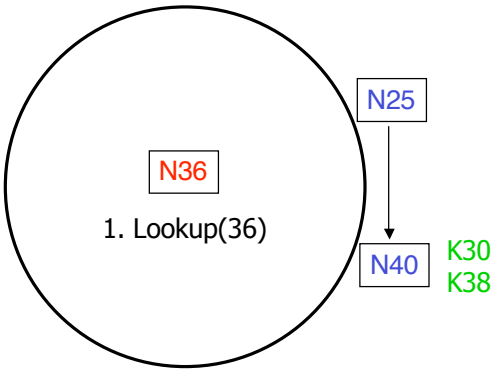
Lookup with fingers

```
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
```

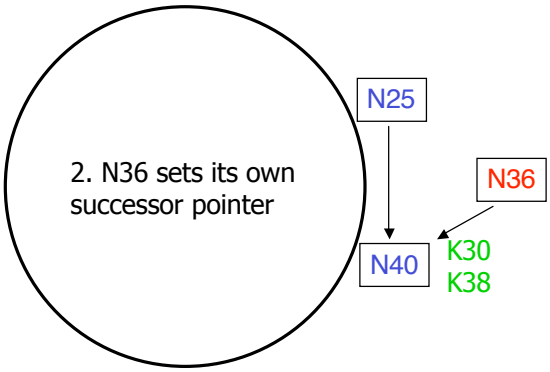
Lookups take O(*log(N)*) hops



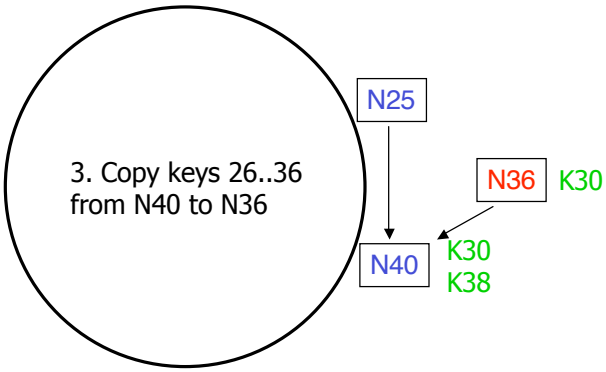
Joining: linked list insert



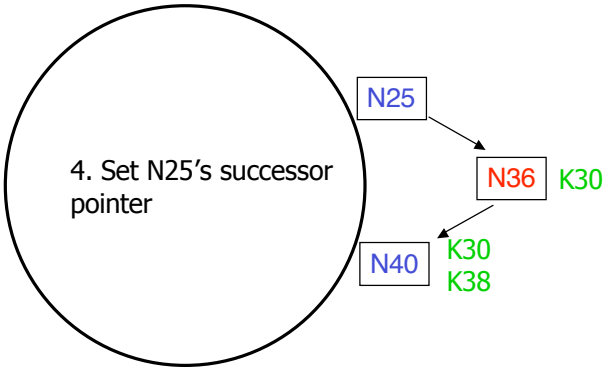
Join (2)


2. N36 sets its own successor pointer
N25
N36
N40
K30
K38

Join (3)

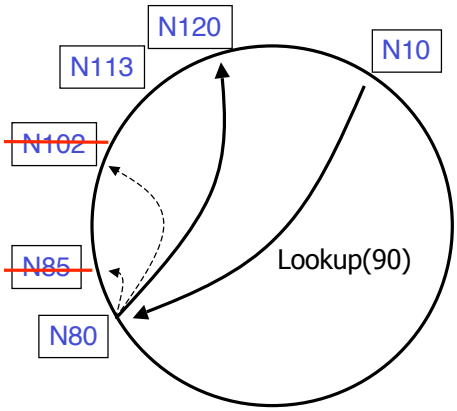

3. Copy keys 26..36 from N40 to N36
N25
N36
K30
N40
K30
K38

Join (4)



Update finger pointers in the background
Correct successors produce correct lookups

Failures might cause incorrect lookup



N80 doesn't know correct successor, so incorrect lookup

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

Choosing the successor list length

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

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)
```


Chord status

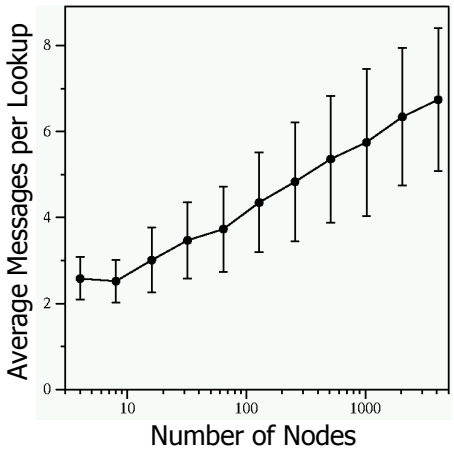
- Working implementation as part of CFS
- Chord library: 3,000 lines of C++
- Deployed in small Internet testbed
- Includes:
  - Correct concurrent join/fail
  - Proximity-based routing for low delay
  - Load control for heterogeneous nodes
  - Resistance to spoofed node IDs

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

Chord lookup cost is O(log N)

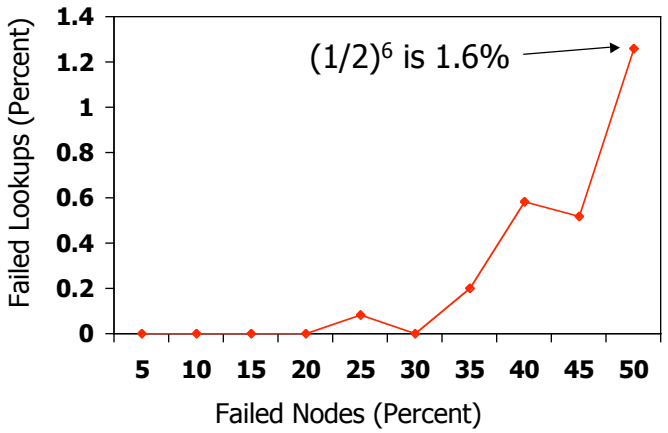


Constant is 1/2

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

Massive failures have little impact



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

Chord Evaluation

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

A Scalable, Content-Addressable Network

Sylvia Ratnasamy[1,2], Paul Francis[3], Mark Handley[1],
Richard Karp[1,2], Scott Shenker[1]

[1] ACIRI [2] U.C.Berkeley [3] Tahoe Networks

Outline

- Introduction
- Design
- Evalution
- Ongoing Work

Internet-scale hash tables

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

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

Content-Addressable Network
(CAN)

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

Content-Addressable Network
(CAN)

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

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 …

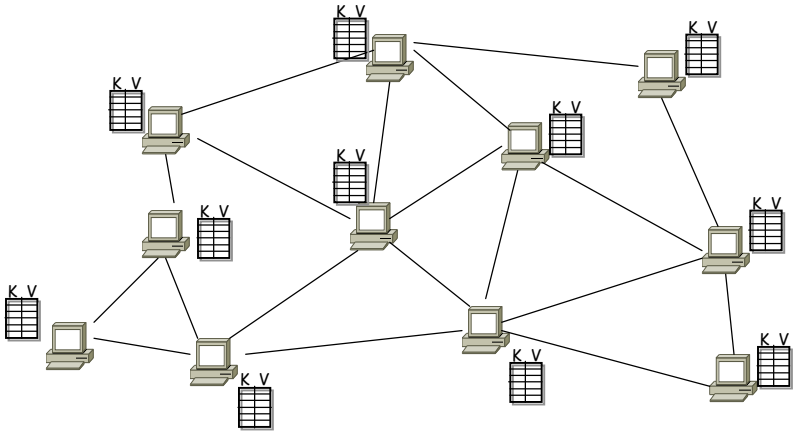
Problem Scope

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

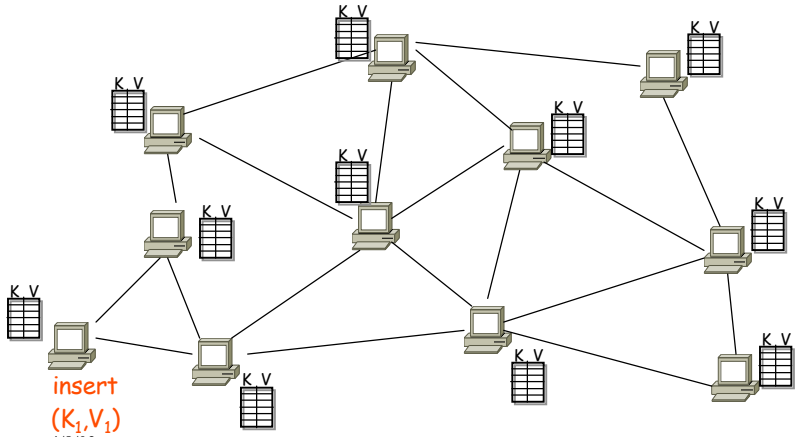
Outline

- Introduction
- Design
- Evalution
- Ongoing Work

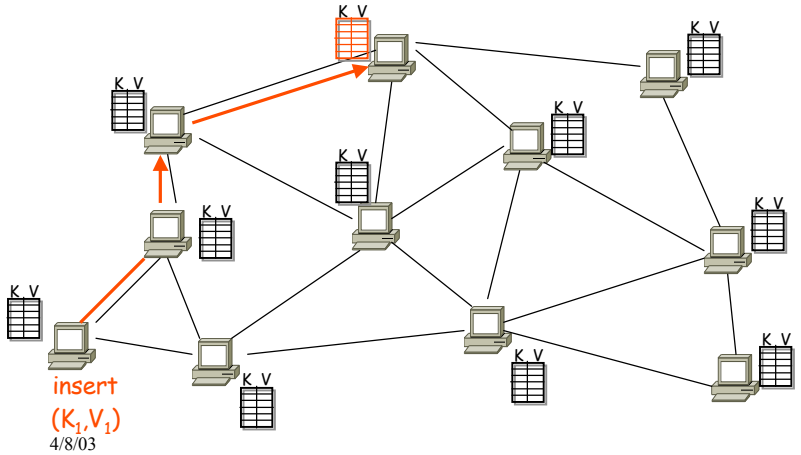
CAN: basic idea



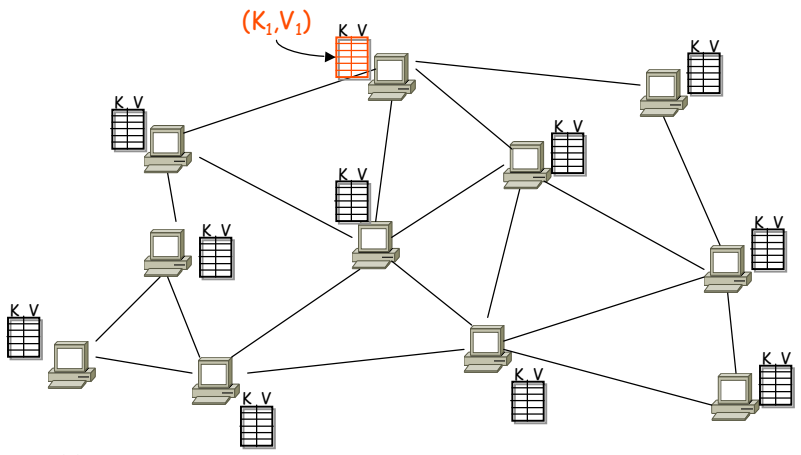
CAN: basic idea



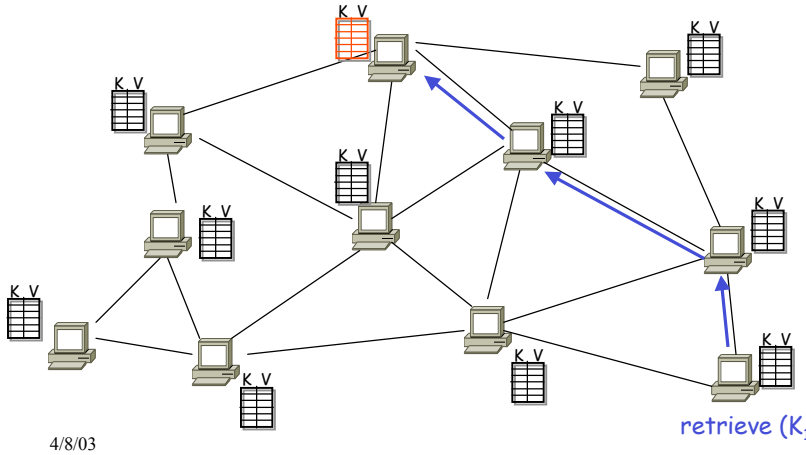
CAN: basic idea



CAN: basic idea



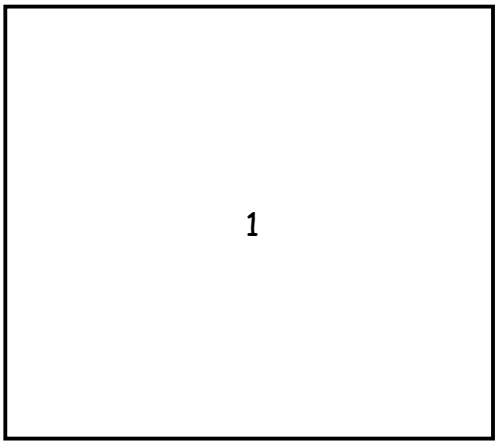
CAN: basic idea



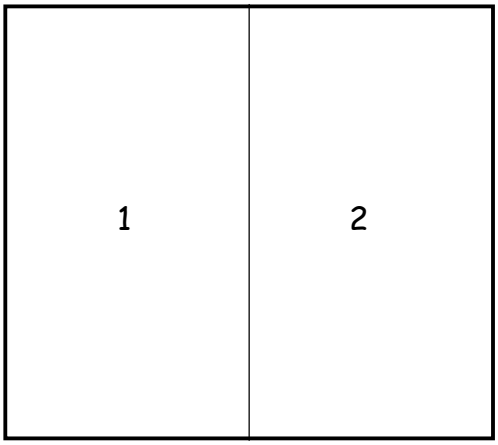
CAN: solution

- 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

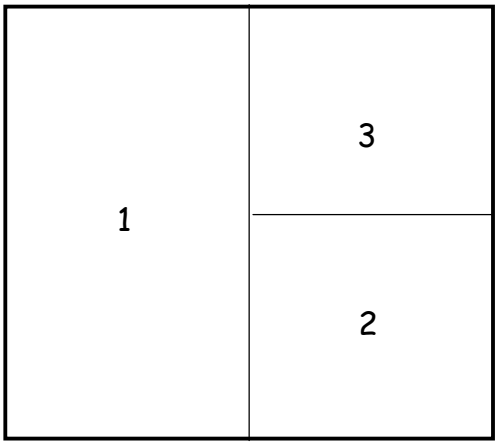
CAN: simple example



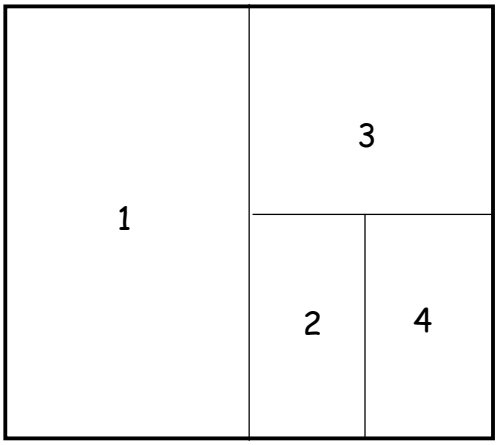
CAN: simple example



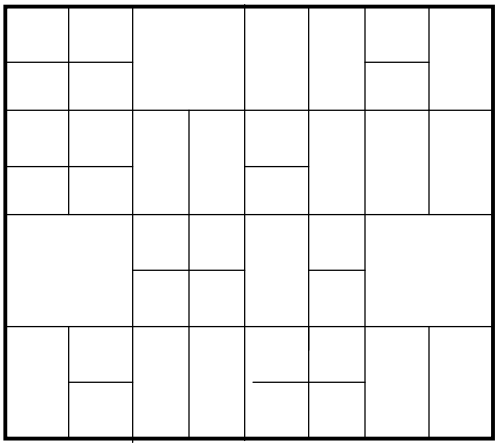
CAN: simple example



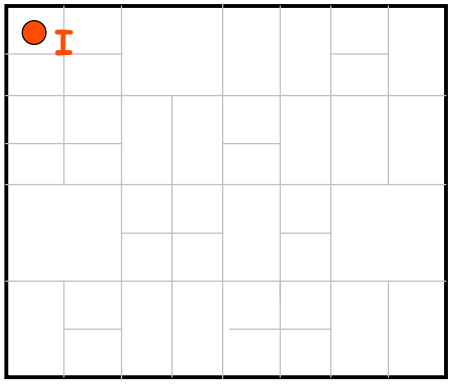
CAN: simple example



CAN: simple example

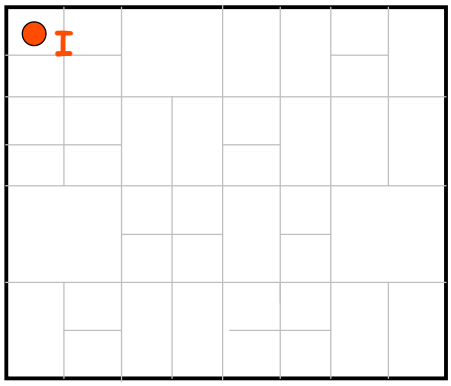


CAN: simple example



CAN: simple example

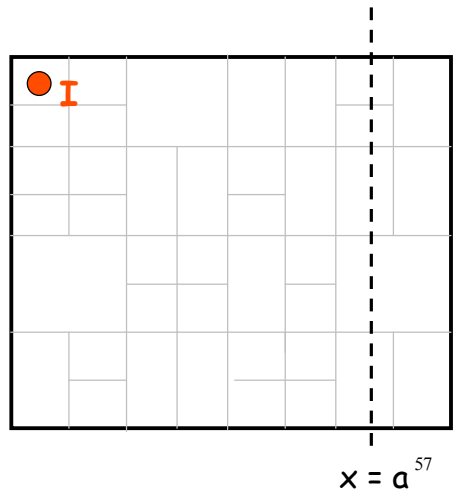
node I::insert(K,V)



CAN: simple example

node I::insert(K,V)

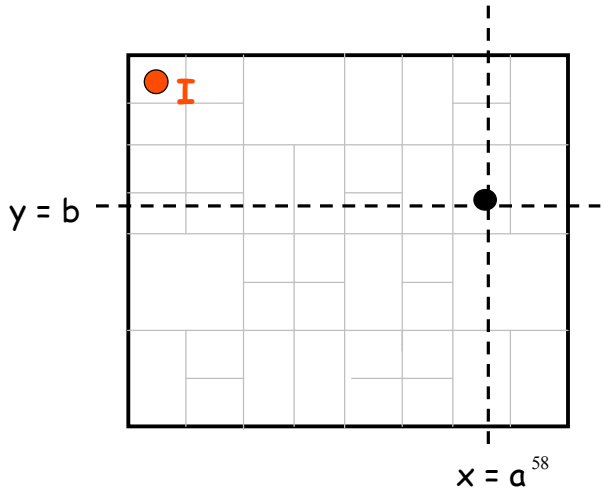
(1) $a = h_x(K)$



CAN: simple example

node I::insert(K,V)

(1) $a = h_x(K)$
$b = h_y(K)$

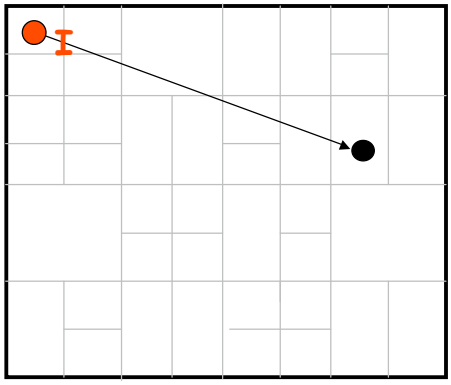


CAN: simple example

node I::insert(K,V)

(1) $a = h_x(K)$
$b = h_y(K)$

(2) route(K,V) -> (a,b)



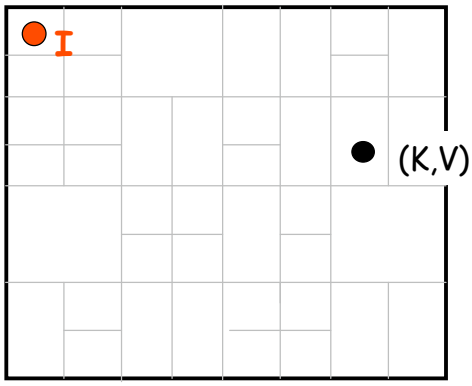
CAN: simple example

node I::insert(K,V)

(1) $a = h_x(K)$
$b = h_y(K)$

(2) route(K,V) -> (a,b)

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

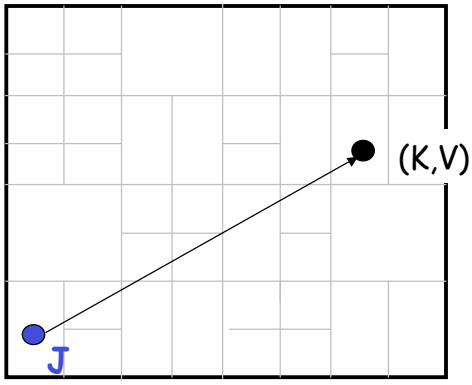


CAN: simple example

node J::retrieve(K)

(1) $a = h_x(K)$
$b = h_y(K)$

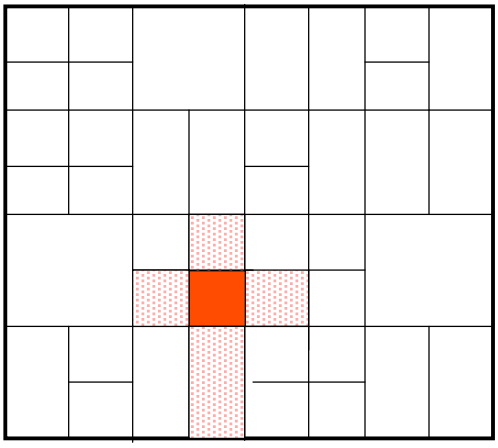
(2) route "retrieve(K)" to (a,b)



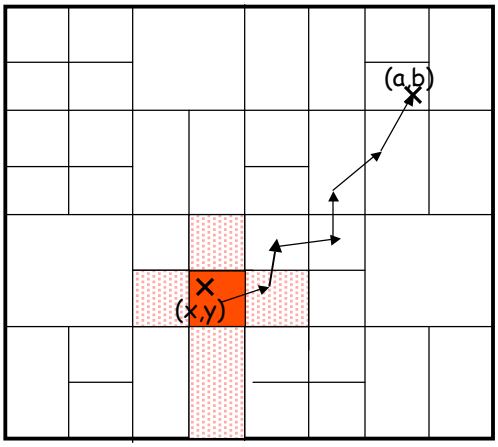
CAN

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

CAN: routing table



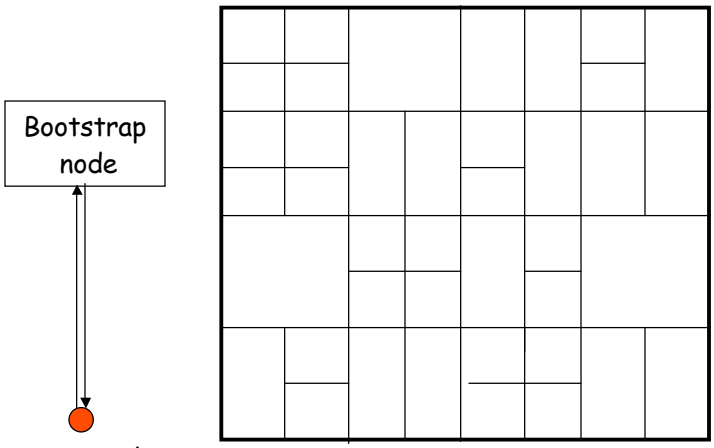
CAN: routing



CAN: routing

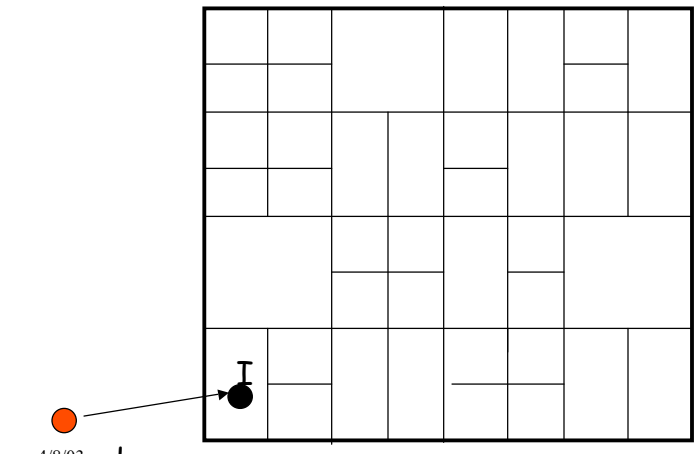
A node only maintains state for its immediate neighboring nodes

CAN: node insertion



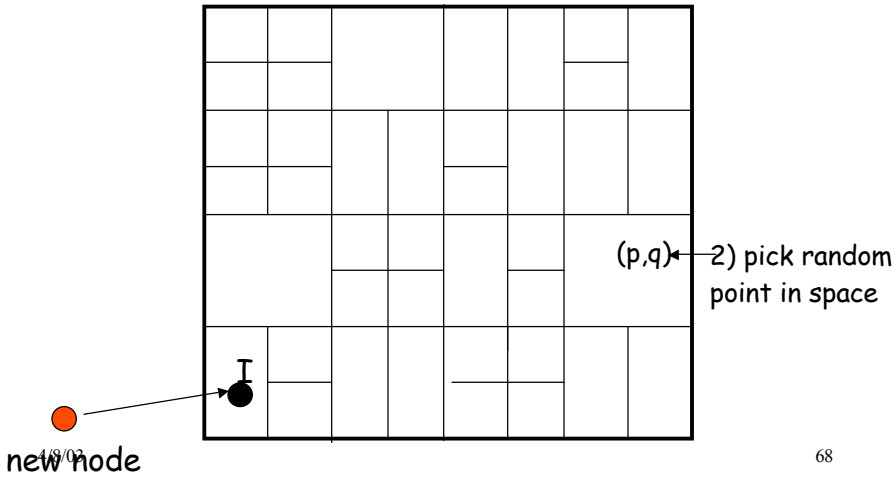
1) Discover some node "I" already in CAN

CAN: node insertion

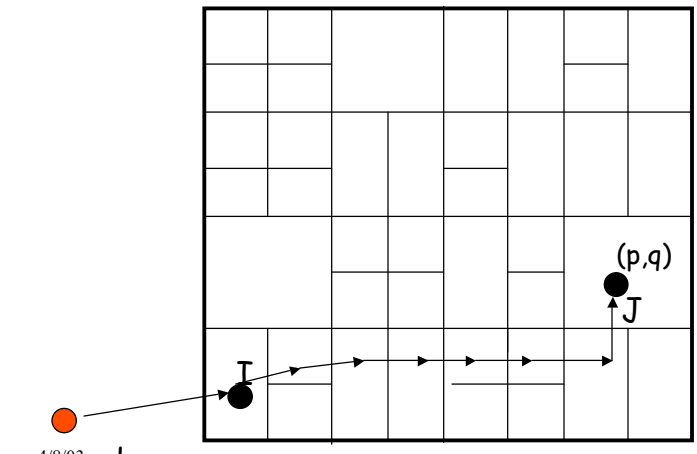


1) discover some node "I" already in CAN

CAN: node insertion

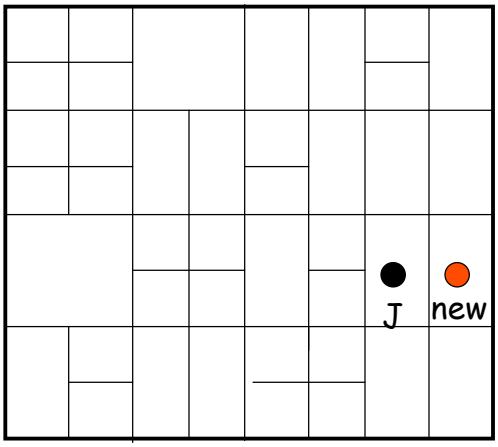


CAN: node insertion



3) I routes to (p,q), discovers node J

CAN: node insertion



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

CAN: node insertion

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

CAN: node failures

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

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

CAN: node failures

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

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

Outline

- Introduction
- Design
- Evalution
- Ongoing Work

Evaluation

- Scalability
- Low-latency
- Load balancing
- Robustness

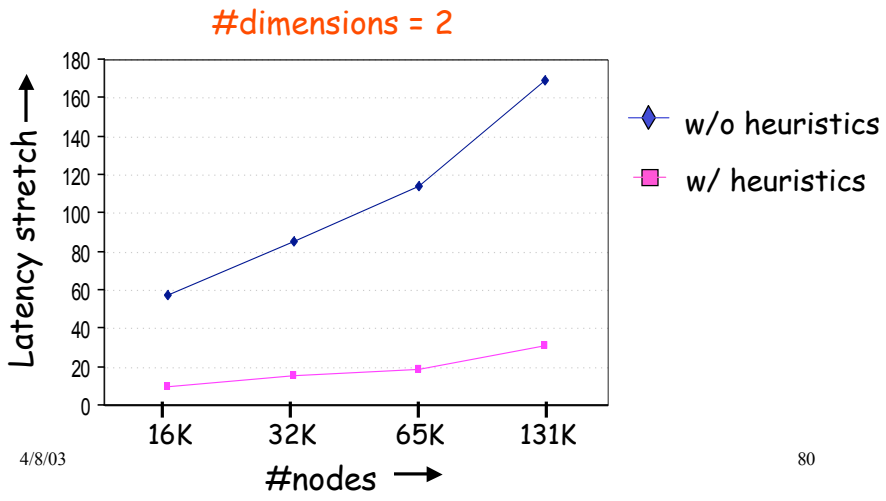
CAN: scalability

- 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

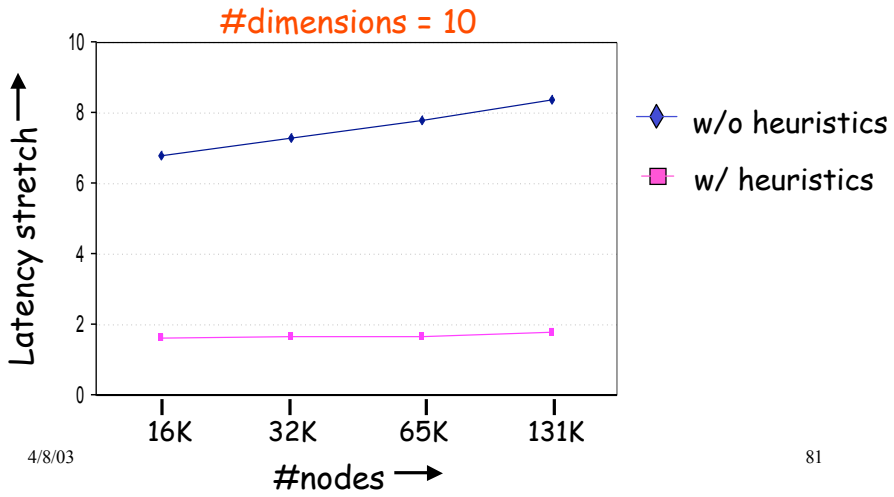
CAN: low-latency

- 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

CAN: low-latency



CAN: low-latency



CAN: load balancing

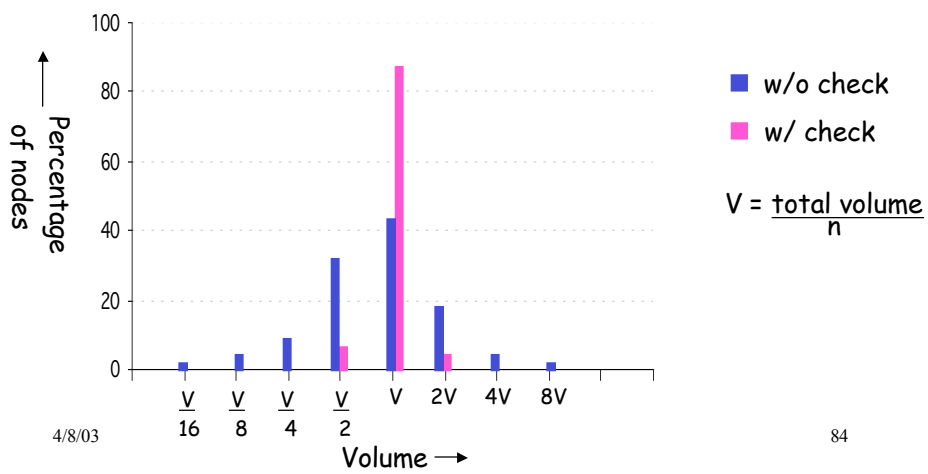
- Two pieces
  - Dealing with hot-spots
    - popular (key,value) pairs
    - nodes cache recently requested entries
    - overloaded node replicates popular entries at neighbors
  - Uniform coordinate space partitioning
    - uniformly spread (key,value) entries
    - uniformly spread out routing load

Uniform Partitioning

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

Uniform Partitioning

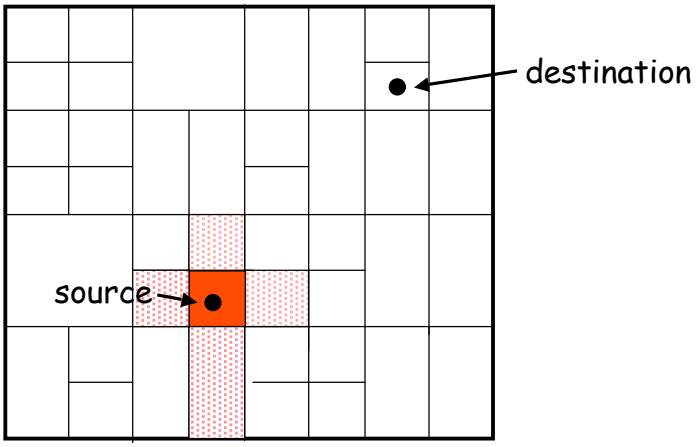
65,000 nodes, 3 dimensions



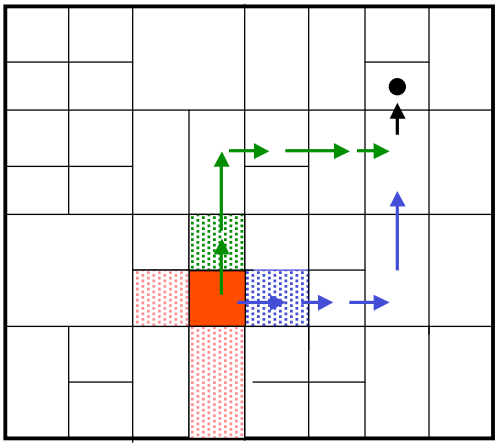
CAN: Robustness

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

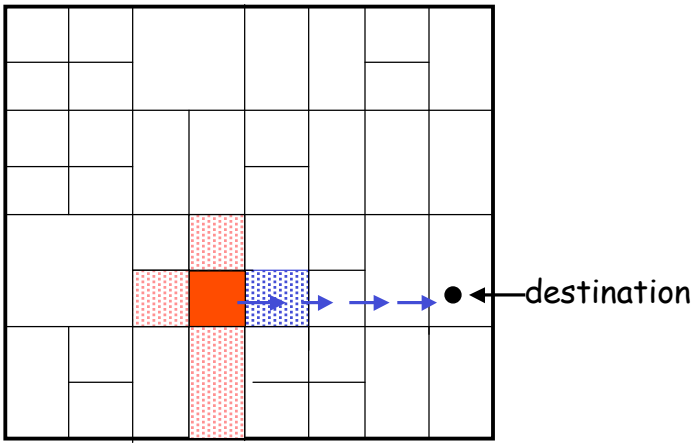
Routing resilience



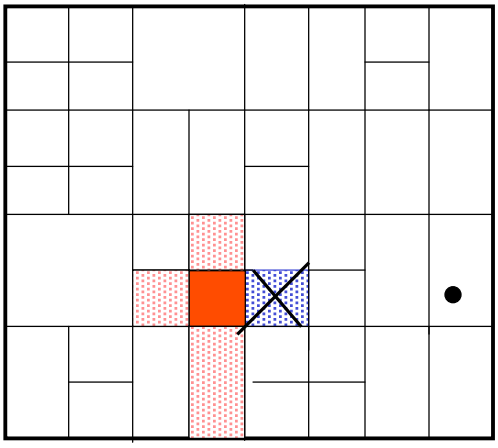
Routing resilience



Routing resilience



Routing resilience

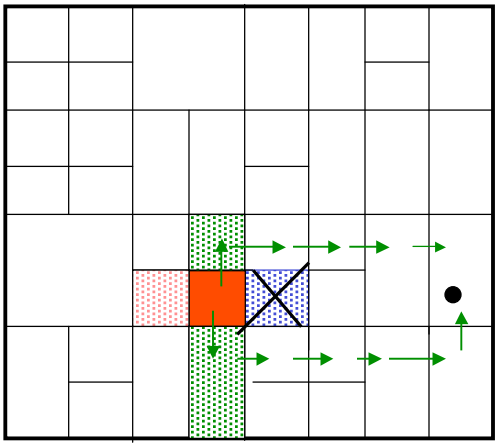


Routing resilience

- 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

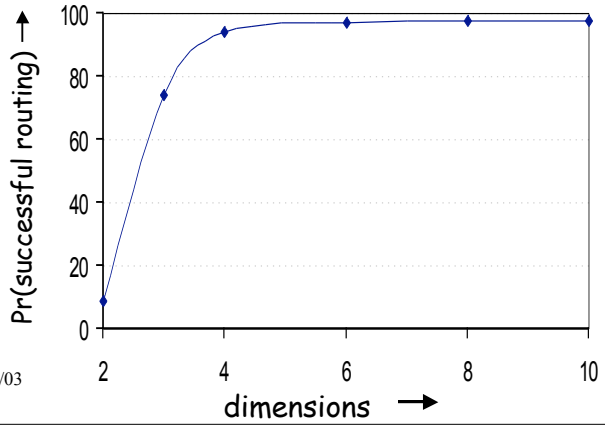
Routing resilience



Routing resilience

CAN size = 16K nodes

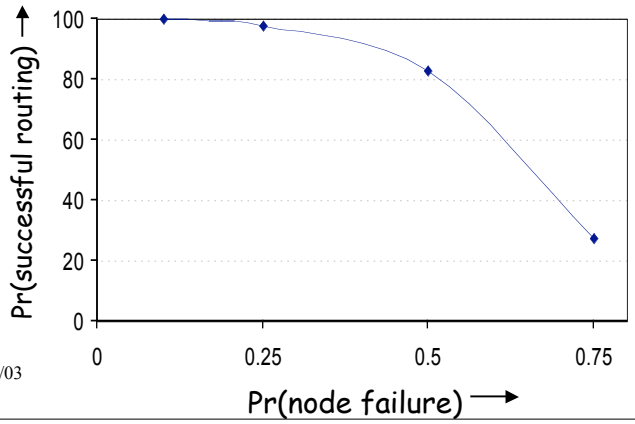
Pr(node failure) = 0.25



Routing resilience

CAN size = 16K nodes

#dimensions = 10



Outline

- Introduction
- Design
- Evalution
- Ongoing Work

Ongoing Work

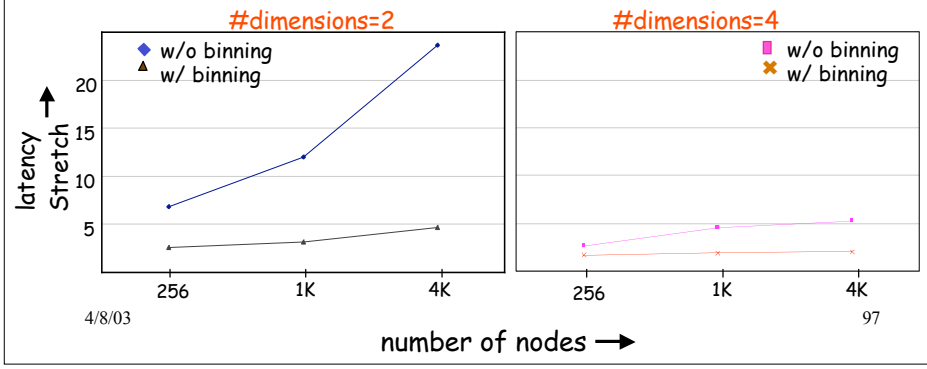
- Topologically-sensitive CAN construction
  - distributed binning

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

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

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)

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

CAN Evaluation