Distributed Hash Tables (DHTs)
Chord & CAN

CS 699/IT 818

Sanjeev Setia

Acknowledgements

The followings slides are borrowed or adapted from talks by Robert Morris (MIT) and Sylvia Ratnasamy (ICSI)

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

SIGCOMM Proceedings, 2001

Chord Simplicity

- Resolution entails participation by O(*log(N)*) nodes
- Resolution is efficient when each node enjoys accurate information about O(*log(N)*) other nodes
- Resolution is possible when each node enjoys accurate information about 1 other node

"Degrades gracefully"

Chord Algorithms

- Basic Lookup
- Node Joins
- Stabilization
- Failures and Replication

5

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

6

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?

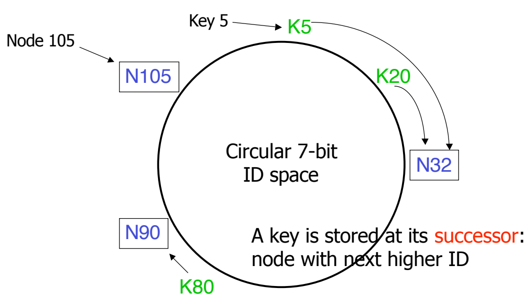
7

Consistent Hashing[Karger 97]

- Target: web page caching
- Like normal hashing, assigns items to buckets so that each bucket receives roughly the same number of items
- Unlike normal hashing, a small change in the bucket set does not induce a total remapping of items to buckets

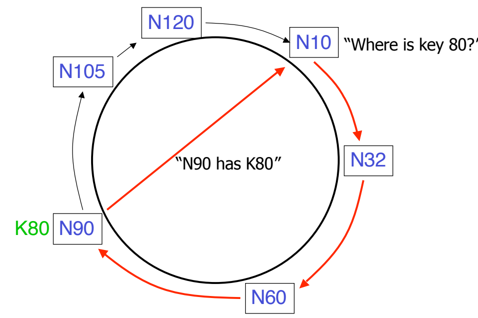
8

Consistent Hashing [Karger 97]



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

Basic lookup

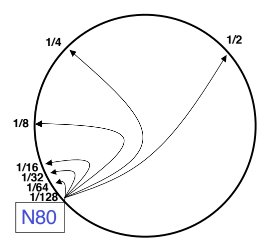


Simple lookup algorithm

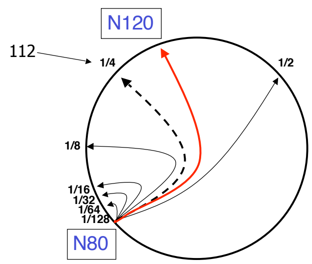
```
Lookup(my-id, key-id)
   n = my successor
   if my-id < n < key-id
        call Lookup(id) on node n    // next hop
   else
        return my successor                  // done
```

- Correctness depends only on successors

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



Finger i points to successor of $n+2^{i-1}$



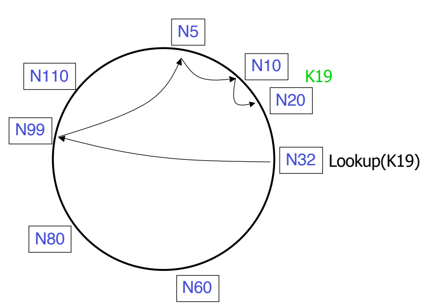
13

Lookup with fingers

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

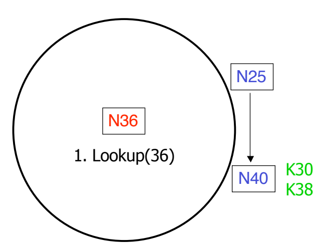
14

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



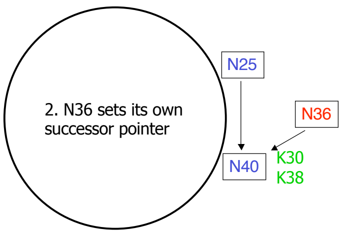
15

Node Join - Linked List Insert



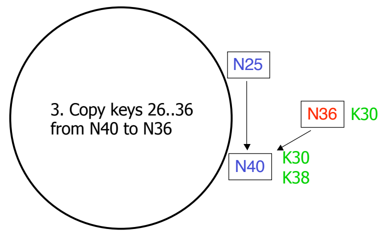
16

Node Join (2)



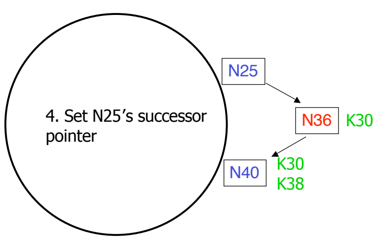
17

Node Join (3)



18

Node Join (4)



Update finger pointers in the background
Correct successors produce correct lookups

19

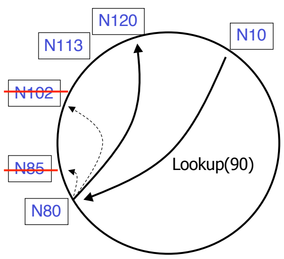
Stabilization

- Case 1: finger tables are reasonably fresh
- Case 2: successor pointers are correct; fingers are inaccurate
- Case 3: successor pointers are inaccurate or key migration is incomplete

- Stabilization algorithm periodically verifies and refreshes node knowledge
  - Successor pointers
  - Predecessor pointers
  - Finger tables

20

Failures and Replication



N80 doesn't know correct successor, so incorrect lookup

21

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

22

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. = $1 - 1/N$

23

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

24

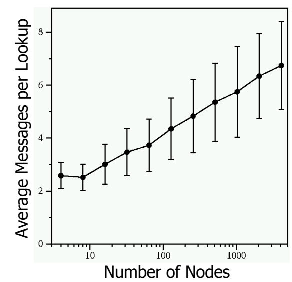
Experimental overview

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

Experiments confirm theoretical results

25

Chord lookup cost is O(log N)



Constant is 1/2

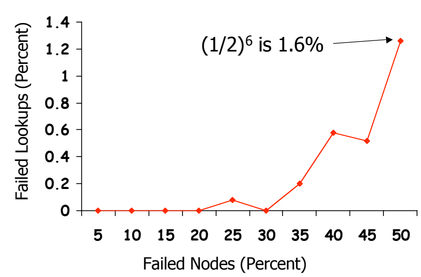
26

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

27

Massive failures have little impact



28

Latency Measurements

- 180 Nodes at 10 sites in US testbed
- 16 queries per physical site (*sic*) for random keys
- Maximum < 600 ms
- Minimum > 40 ms
- Median = 285 ms
- Expected value = 300 ms (5 round trips)

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

A Scalable, Content-Addressable Network

Sylvia Ratnasamy, Paul Francis, Mark Handley,

Richard Karp, Scott Shenker

Content-Addressable Network (CAN)

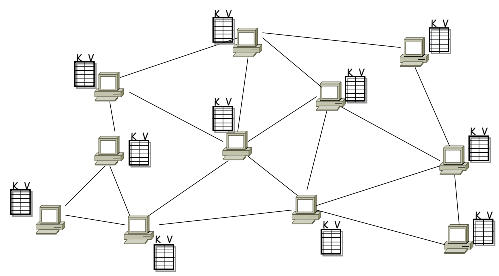
- CAN: Internet-scale hash table
- Interface
  - insert(key,value)
  - value = retrieve(key)
- Properties
  - scalable
  - operationally simple
  - good performance
- Related systems: Chord/Pastry/Tapestry/Buzz/Plaxton ...

Problem Scope

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

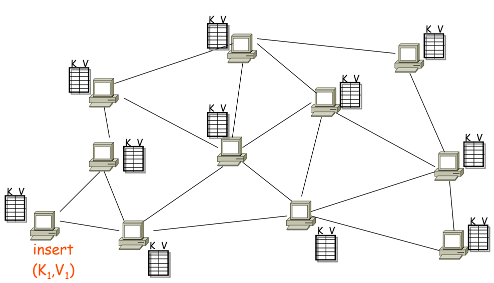
33

CAN: basic idea



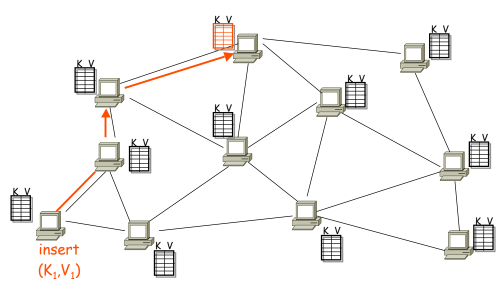
34

CAN: basic idea



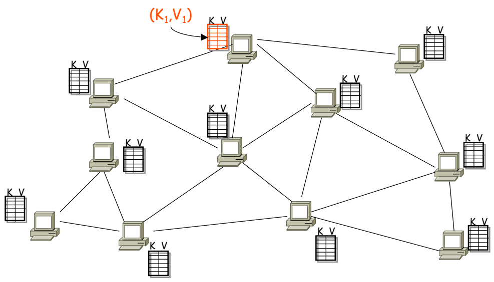
35

CAN: basic idea

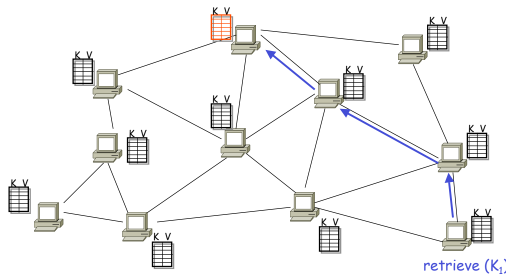


36

CAN: basic idea



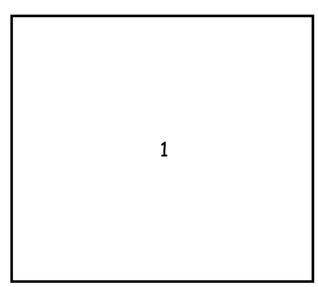
CAN: basic idea



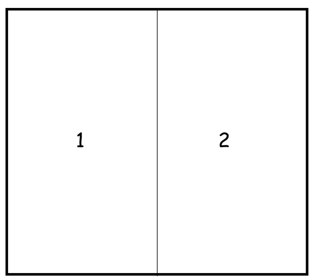
CAN: solution

- 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

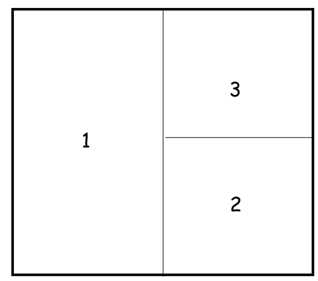
CAN: simple example



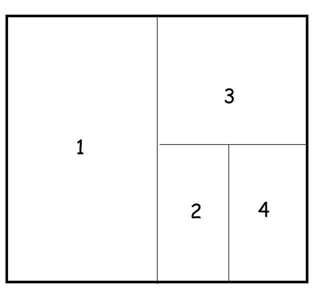
CAN: simple example



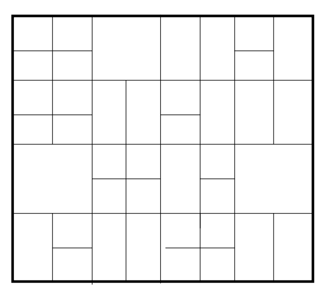
CAN: simple example



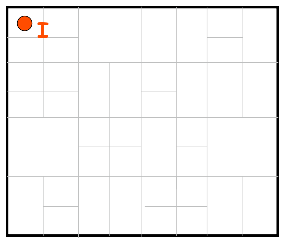
CAN: simple example



CAN: simple example

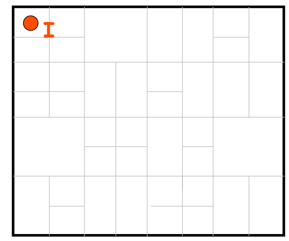


CAN: simple example



CAN: simple example

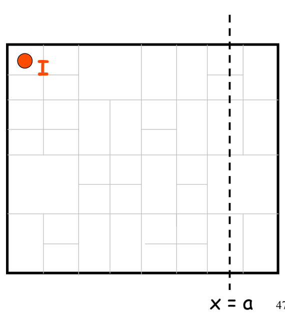
node I::insert(K,V)



CAN: simple example

node I::insert(K,V)

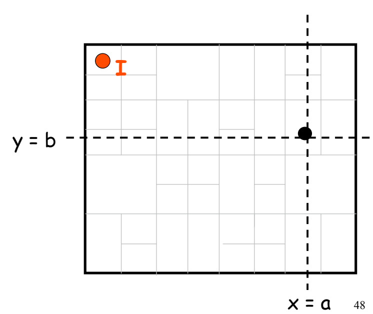
(1) $a = h_x(K)$



CAN: simple example

node I::insert(K,V)

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

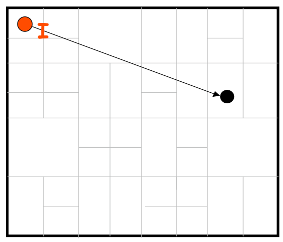


CAN: simple example

node I::insert(K,V)

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

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



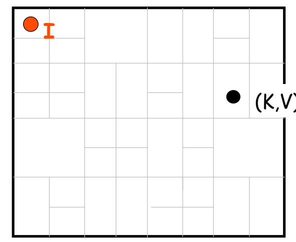
CAN: simple example

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)

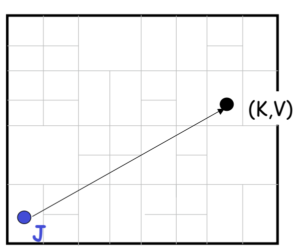


CAN: simple example

node J::retrieve(K)

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

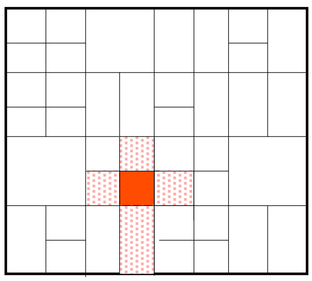
(2) route "retrieve(K)" to (a,b)



CAN

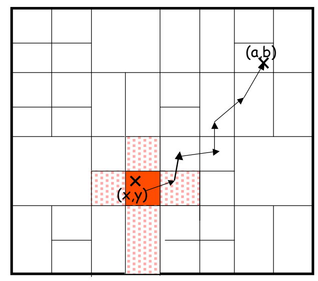
Data stored in the CAN is addressed by name (i.e. key), not location (i.e. IP address)

CAN: routing table



53

CAN: routing



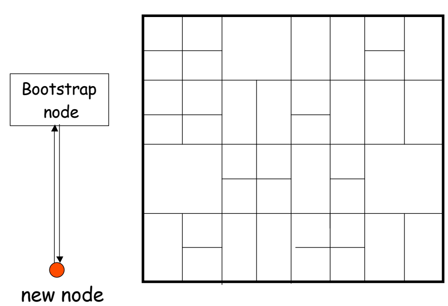
54

CAN: routing

A node only maintains state for its immediate neighboring nodes

55

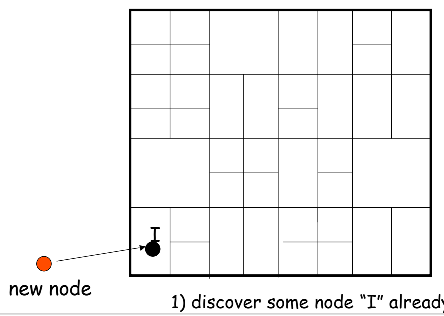
CAN: node insertion



1) Discover some node "I" already in CAN

56

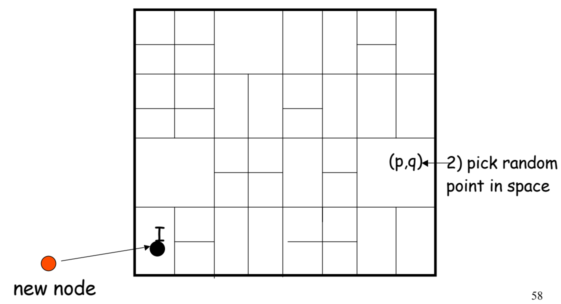
CAN: node insertion



1) discover some node "I" already in CAN

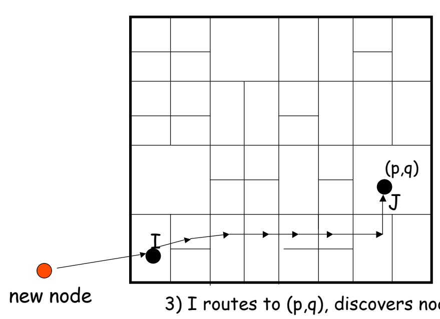
57

CAN: node insertion



58

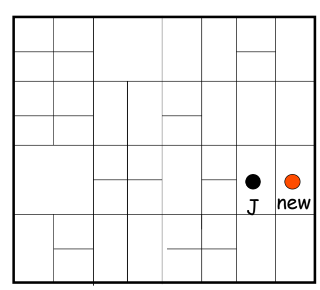
CAN: node insertion



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

59

CAN: node insertion



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

60

CAN: node insertion

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

61

CAN: node failures

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

62

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

63

CAN: node failures

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

64

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

Evaluation

- Scalability
- Low-latency
- Load balancing
- Robustness

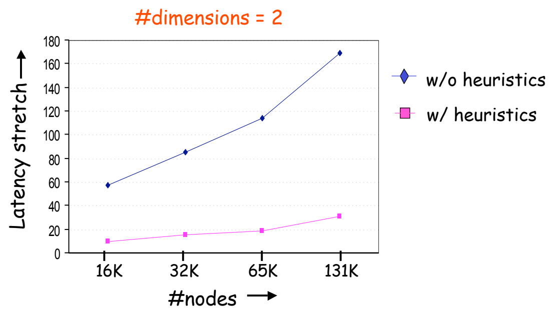
CAN: scalability

- 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

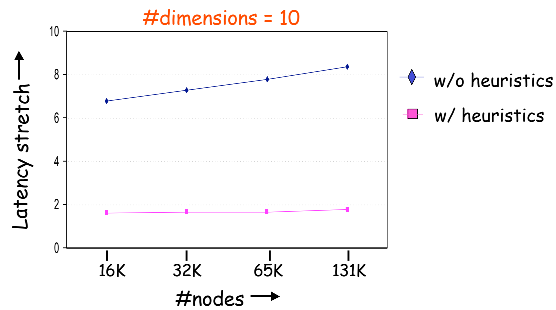
- Problem
  - latency stretch = $\frac{\text{(CAN routing delay)}}{\text{(IP routing delay)}}$
  - application-level routing may lead to high stretch
- Solution
  - increase dimensions
  - heuristics
    - RTT-weighted routing
    - multiple nodes per zone (peer nodes)
    - deterministically replicate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

CAN: low-latency



69

CAN: low-latency



70

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

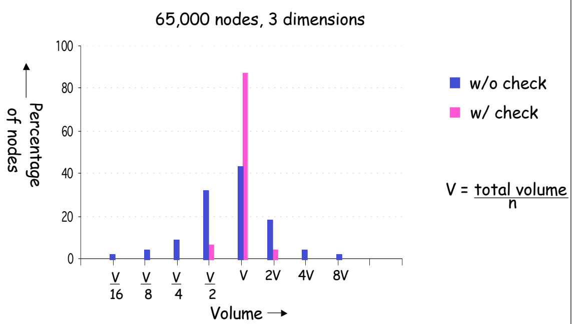
71

Uniform Partitioning

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

72

Uniform Partitioning



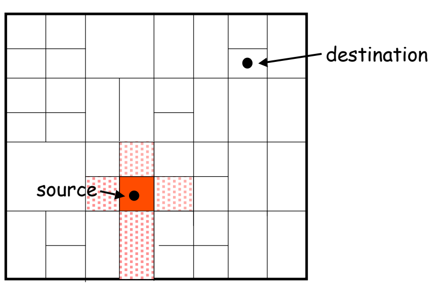
73

CAN: Robustness

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

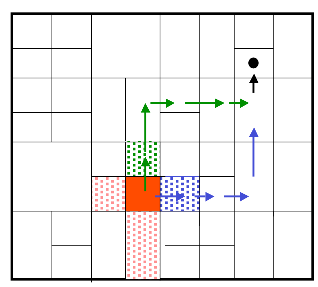
74

Routing resilience



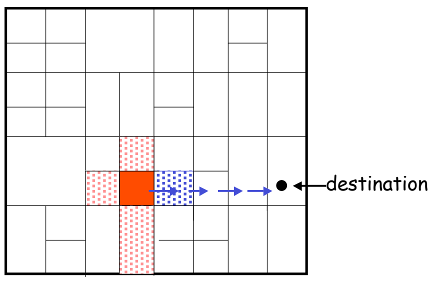
75

Routing resilience



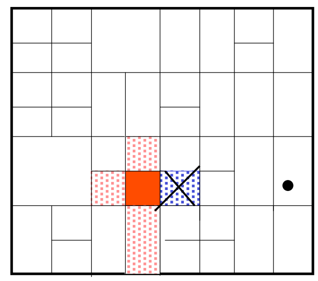
76

Routing resilience



77

Routing resilience



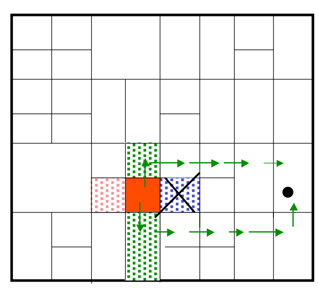
78

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

79

Routing resilience

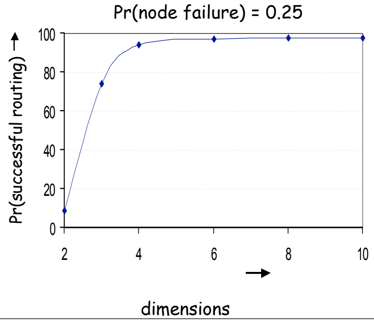


80

Routing resilience

CAN size = 16K nodes

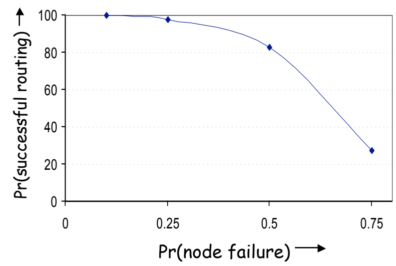
Pr(node failure) = 0.25



Routing resilience

CAN size = 16K nodes

#dimensions = 10



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