5 Lectures on Peer-peer and Application-level Networking Advanced System Topics

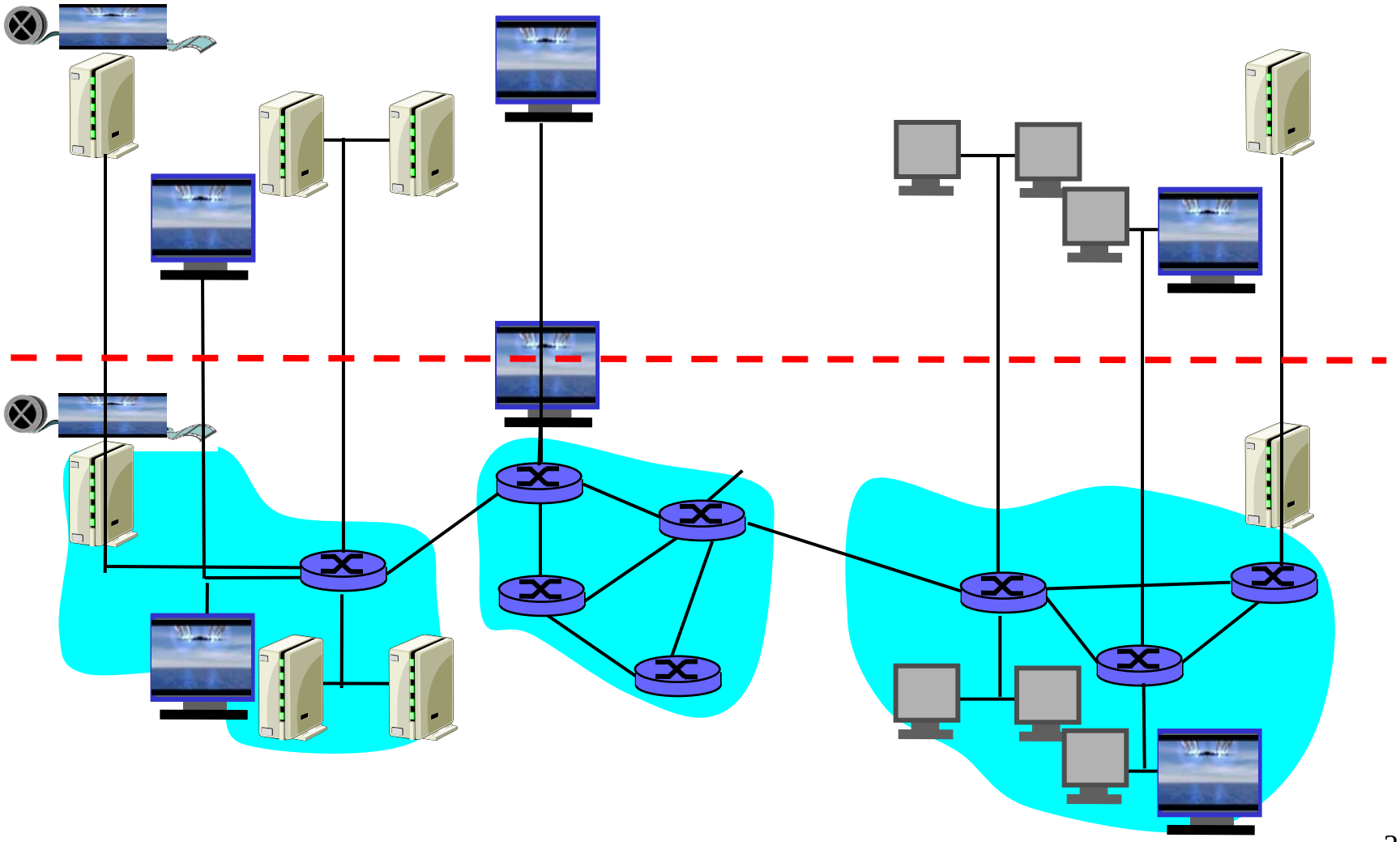
Presented by Jon Crowcroft

Ack for material by
Jim Kurose,Brian Levine,Don Towsley,Umass, + Marcello Pias (CL) & Ant Rowstron (MSR)

0.Introduction

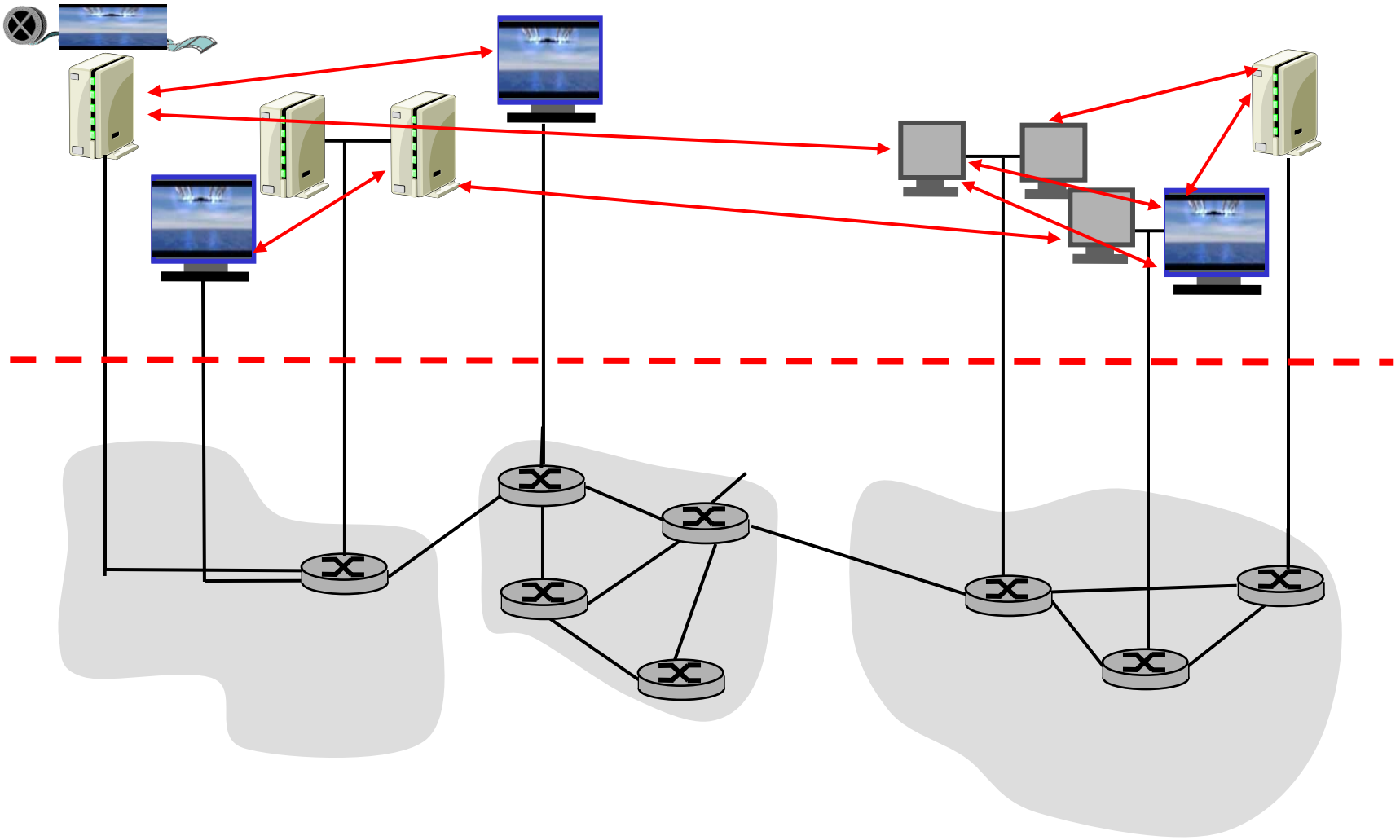
- Background
- Motivation
- outline of the lectures

Peer-peer networking



Peer-peer networking

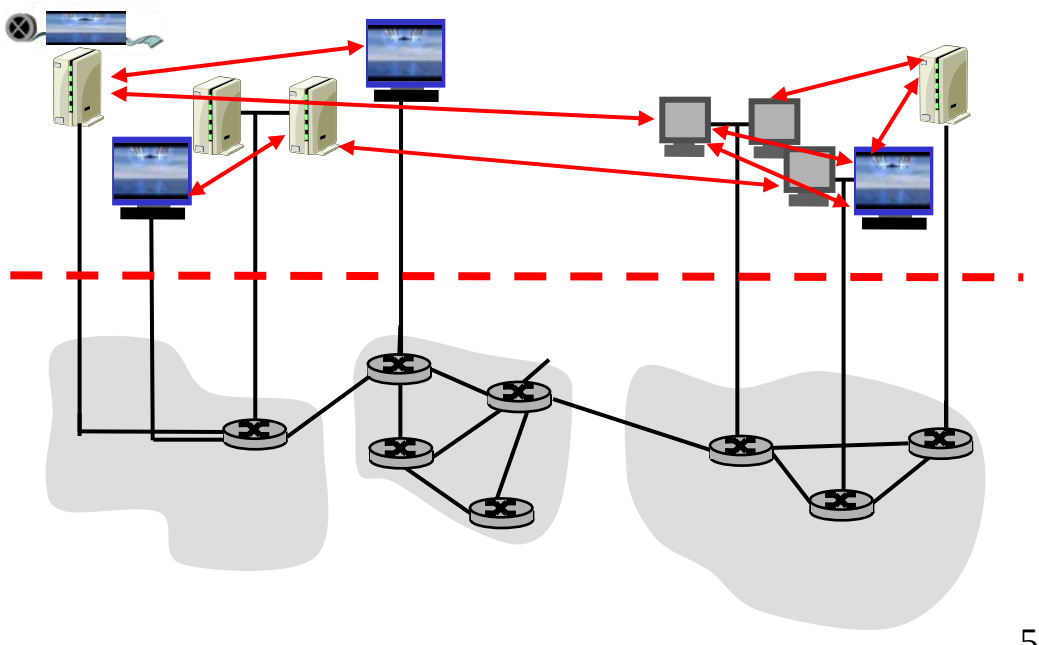
Focus at the application level



Peer-peer networking

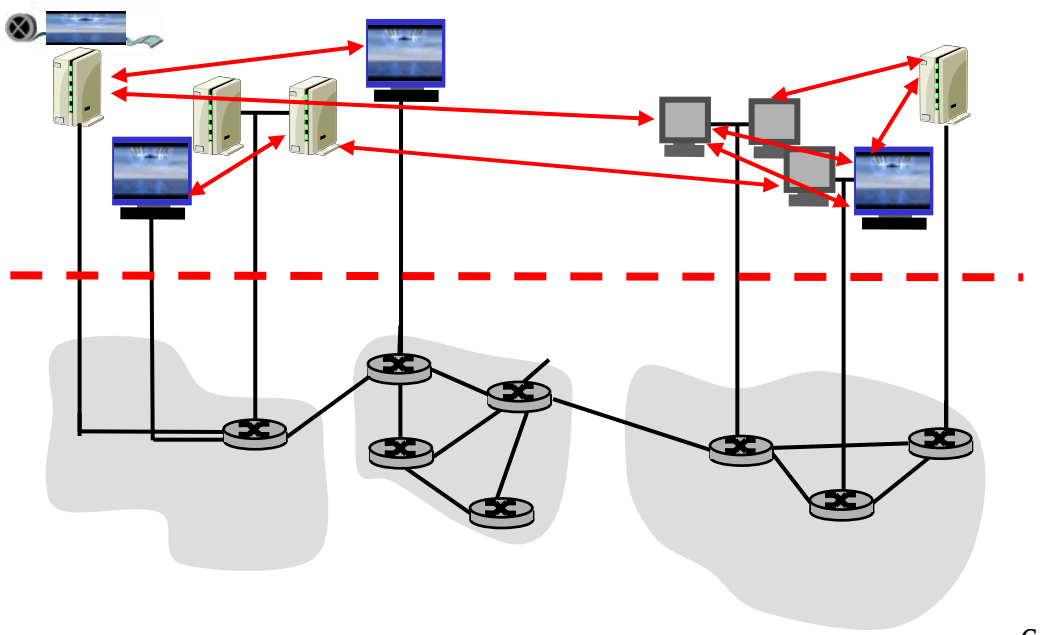
Peer-peer applications

- Napster, Gnutella, Freenet: file sharing
- ad hoc networks
- multicast overlays (e.g., video distribution)



Peer-peer networking

- Q: What are the new technical challenges?
- Q: What new services/applications enabled?
- Q: Is it just "networking at the application-level"?
  - "There is nothing new under the Sun" (William Shakespeare)



Course Lecture Contents

- Introduction
  - Client-Server v. P2P
  - File Sharing/Finding
  - Napster
  - Gnutella
- Overlays and Query Routing
  - RON
  - Freenet
  - Publius
- Distributed Hash Tables
  - Chord
  - CAN
  - Tapestry
- Middleware
  - JXTA
  - ESM
  - Overcast
- Applications
  - Storage
  - Conferencing

Client Server v. Peer to Peer(1)

- RPC/RMI
- Synchronous
- Assymmetric
- Emphasis on language integration and binding models (stub *IDL/XDR* compilers etc)
- Kerberos style security – access control, crypto

- Messages
- Asynchronous
- Symmetric
- Emphasis on service location, content addressing, application layer routing.
- Anonymity, high availability, integrity.
- Harder to get right☺

Client Server v. Peer to Peer(2) RPC

```
Cli_call(args)

Srv_main_loop()
{
  while(true) {
   deque(call)
   switch(call.procid)
   case 0:
   call.ret=proc1(call.args)
   case 1:
   call.ret=proc2(call.args)
   … … …
   default:
   call.ret = exception
   }
}
```

Client Server v. Peer to Peer(3) P2P

```
Peer_main_loop()
{
   while(true) {
   await(event)
   switch(event.type) {
   case timer_expire: do some p2p work()
        randomize timers
        break;
   case inbound message: handle it
        respond
        break;
   default: do some book keeping
        break;
   }
}
```

Peer to peer systems actually old

- IP routers are peer to peer.
- Routers discover topology, and maintain it
- Routers are neither client nor server
- Routers continually chatter to each other
- Routers are fault tolerant, inherently
- Routers are autonomous

Peer to peer systems

- Have no distinguished role
- So no single point of bottleneck or failure.
- However, this means they need distributed algorithms for
  - Service discovery (name, address, route, metric, etc)
  - Neighbour status tracking
  - Application layer routing (based possibly on content, interest, etc)
  - Resilience, handing link and node failures
  - Etc etc etc

Ad hoc networks and peer2peer

- Wireless ad hoc networks have many similarities to peer to peer systems
- No *a priori knowledge*
- No given infrastructure
- Have to construct it from "thin air"!
- Note for later – wireless☺

Overlays and peer 2 peer systems

- P2p technology is often used to create overlays which offer services that could be offered in the IP level
- Useful **deployment** strategy
- Often economically a way around other barriers to deployment
- IP Itself was an overlay (on telephone core infrastructure)
- Evolutionary path!!!

Rest of lectures oriented 14 case studies from literature

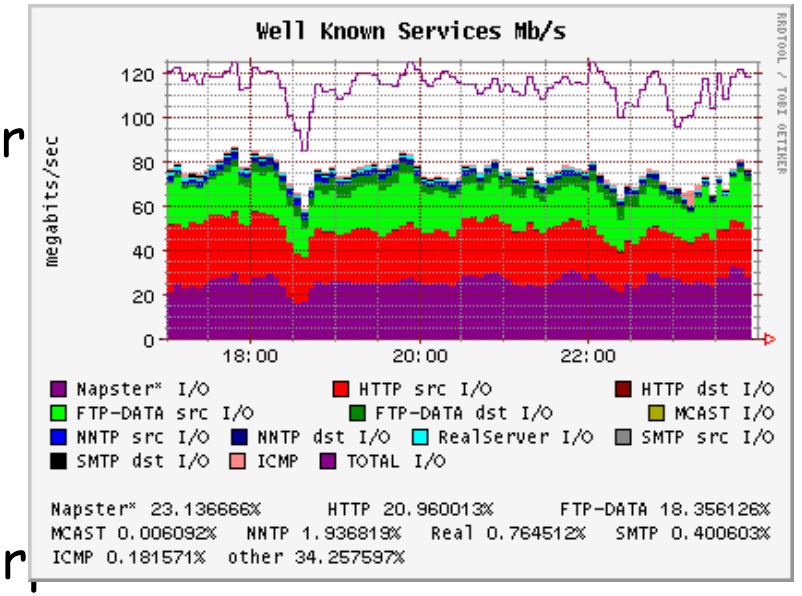
- Piracy^H^H^H^H^H^content sharing ☺
- Napster
- Gnutella
- Freenet
- Publius
- etc

1. NAPSTER

- The most (in)famous
- Not the first (c.f. probably Eternity, from Ross Anderson in Cambridge)
- But instructive for what it gets right, and
- Also wrong…
- Also has a political message…and economic and legal…etc etc etc

Napster

- program for sharing files over the Internet
- a "disruptive" application/technology?
- history:
  - 5/99: Shawn Fanning (freshman, Northeasten U.) founds Napster Online music service
  - 12/99: first lawsuit
  - 3/00: 25% UWisc traffic Napster
  - 2000: est. 60M users
  - 2/01: US Circuit Court of Appeals: Napster knew users violating copyright laws
  - 7/01: # simultaneous online users: Napster 160K, Gnutella: 40K, Mor



Napster: how does it work

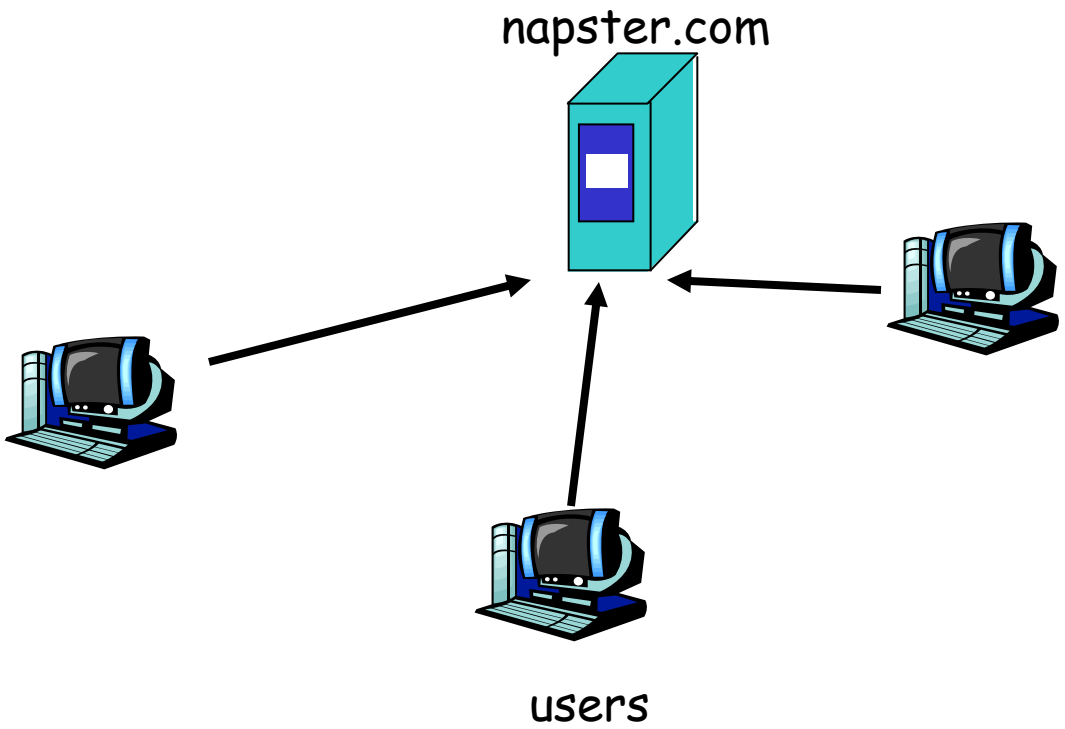
Application-level, client-server protocol over point-to-point TCP

Four steps:

- Connect to Napster server
- Upload your list of files (push) to server.
- Give server keywords to search the full list with.
- Select "best" of correct answers. (pings)

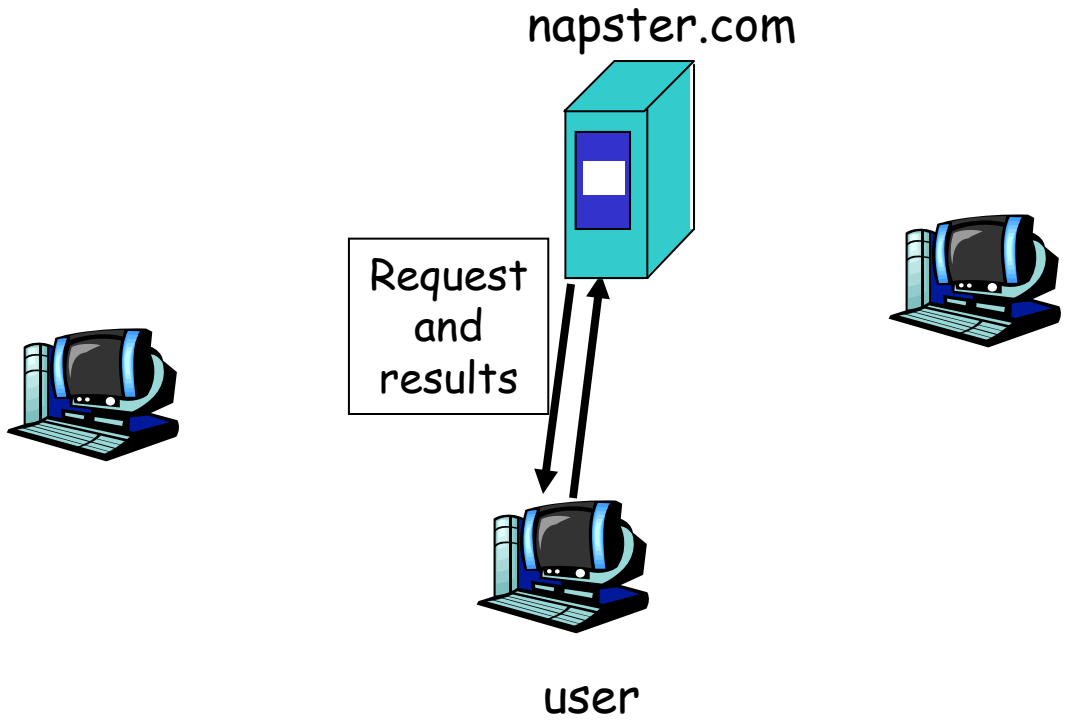
Napster

1. File list is uploaded



Napster

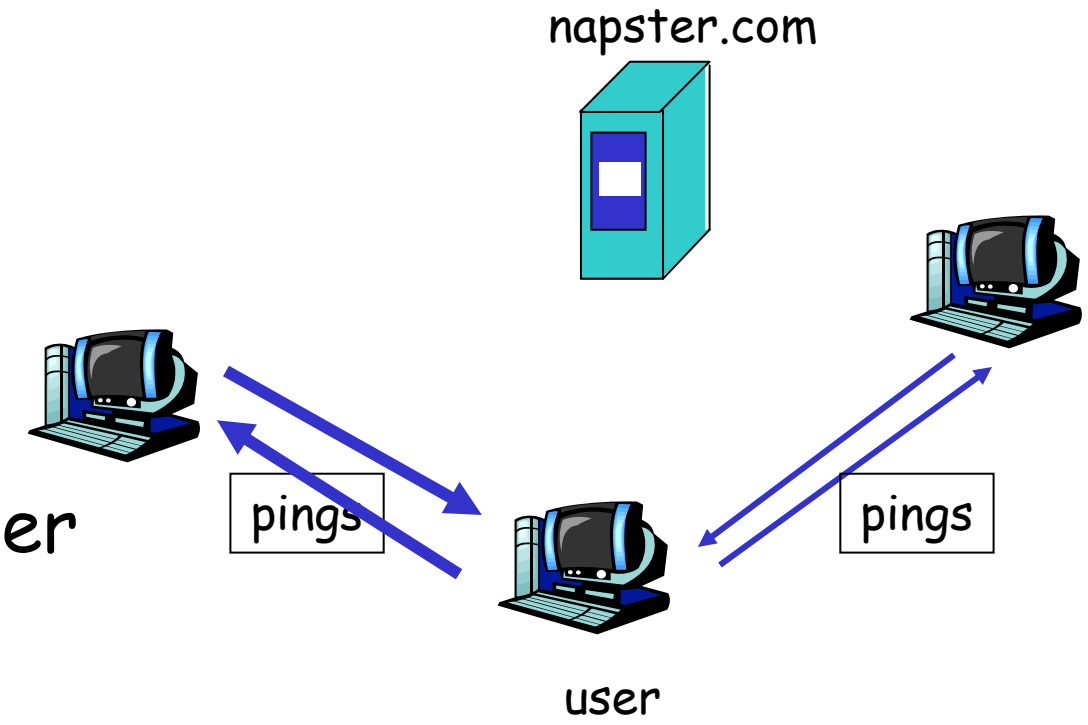
2. User requests search at server.



Napster

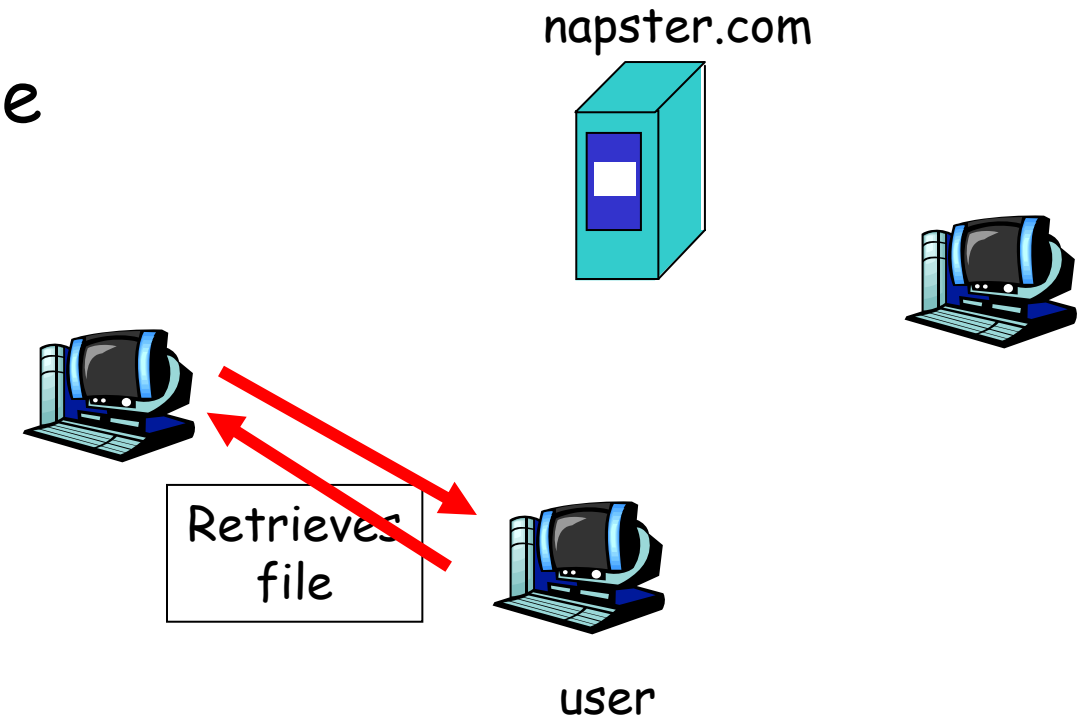
3. User pings hosts that apparently have data.

   Looks for ***best*** transfer rate.



Napster

4. User retrieves file



Napster messages

General Packet Format

[chunksize] [chunkinfo] [data...]

CHUNKSIZE:
Intel-endian 16-bit integer
size of [data...] in bytes

CHUNKINFO: (hex)
Intel-endian 16-bit integer.

00 - login rejected
02 - login requested
03 - login accepted
0D - challenge? (nuprin1715)
2D - added to hotlist
2E - browse error (user isn't online!)
2F - user offline

5B - whois query
5C - whois result
5D - whois: user is offline!
69 - list all channels
6A - channel info
90 - join channel
91 - leave channel
.....

From: http://david.weekly.org/code/napster.php3

Napster: requesting a file

SENT to server (after logging in to server)
    2A 00 CB 00 username
        "C:\MP3\REM - Everybody Hurts.mp3"
RECEIVED
    5D 00 CC 00 username
        2965119704 (IP-address backward-form = A.B.C.D)
        6699 (port)
        "C:\MP3\REM - Everybody Hurts.mp3" (song)
        (32-byte checksum)
        (line speed)
[connect to A.B.C.D:6699]
RECEIVED from client
    31 00 00 00 00 00
SENT to client
    GET
RECEIVED from client
    00 00 00 00 00 00

SENT to client
    Myusername
    "C:\MP3\REM - Everybody Hurts.mp3"
    0 (port to connect to)
RECEIVED from client
    (size in bytes)
SENT to server
    00 00 DD 00 (give go-ahead thru server)
RECEIVED from client
    [DATA]

Napster: architecture notes

- centralized server:
  - single logical point of failure
  - can load balance among servers using DNS rotation
  - potential for congestion
  - Napster "in control" (freedom is an illusion)
- no security:
  - passwords in plain text
  - no authentication
  - no anonymity

2 Gnutella

- Napster fixed
- Open Source
- Distributed
- Still very political…

Gnutella

- peer-to-peer networking: applications connect to peer applications
- focus: decentralized method of searching for files
- each application instance serves to:
  - store selected files
  - route queries (file searches) from and to its neighboring peers
  - respond to queries (serve file) if file stored locally
- Gnutella history:
  - 3/14/00: release by AOL, almost immediately withdrawn
  - too late: 23K users on Gnutella at 8 am this AM
  - many iterations to fix poor initial design (poor design turned many people off)

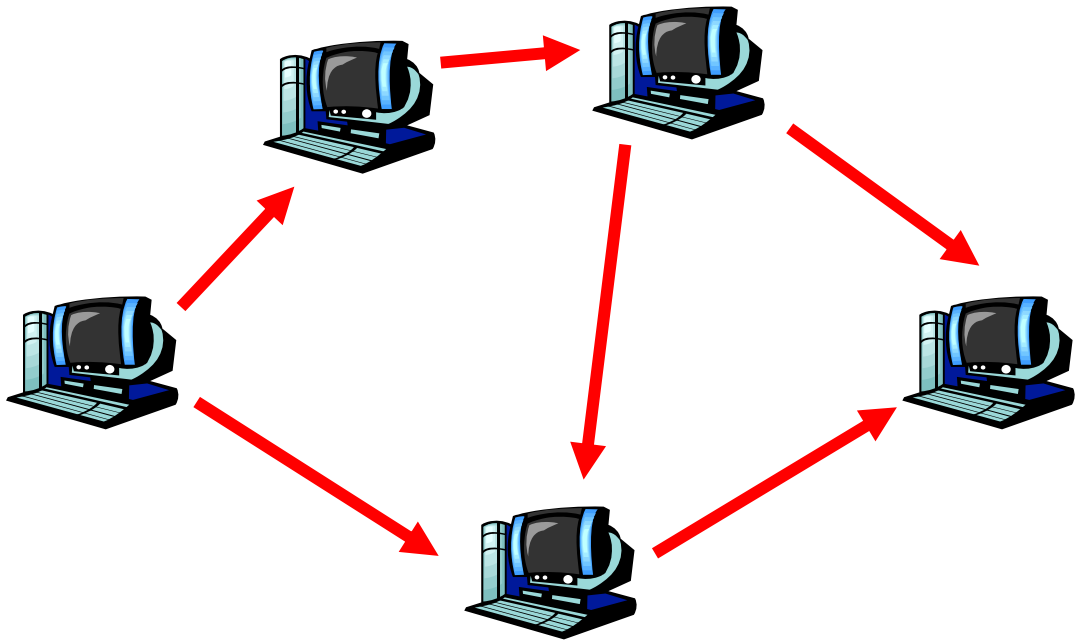
Gnutella: how it works

Searching by flooding:

- If you don't have the file you want, query 7 of your partners.
- If they don't have it, they contact 7 of their partners, for a maximum hop count of 10.
- Requests are flooded, but there is no tree structure.
- No looping but packets may be received twice.
- Reverse path forwarding(?)

Note: Play gnutella animation at:
http://www.limewire.com/index.jsp/p2p

Flooding in Gnutella: loop prevention



Seen already list: "A"

Gnutella message format

- Message ID: 16 bytes (yes bytes)
- FunctionID: 1 byte indicating
  - 00 ping: used to probe gnutella network for hosts
  - 01 pong: used to reply to ping, return # files shared
  - 80 query: search string, and desired minimum bandwidth
  - 81: query hit: indicating matches to 80:query, my IP address/port, available bandwidth
- RemainingTTL: decremented at each peer to prevent TTL-scoped flooding
- HopsTaken: number of peer visited so far by this message
- DataLength: length of data field

Gnutella: initial problems and fixes

- Freeloading: WWW sites offering search/retrieval from Gnutella network without providing file sharing or query routing.
  - Block file-serving to browser-based non-file-sharing users
- Prematurely terminated downloads:
  - long download times over modems
  - modem users run gnutella peer only briefly (Napster problem also!) or any users becomes overloaded
  - fix: peer can reply "I have it, but I am busy. Try again later"
  - late 2000: only 10% of downloads succeed
  - 2001: more than 25% downloads successful (is this success or failure?)

Gnutella: initial problems and fixes (more)

- 2000: avg size of reachable network ony 400-800 hosts. Why so smalll?
  - modem users: not enough bandwidth to provide search routing capabilities: routing black holes
- Fix: create peer hierarchy based on capabilities
  - previously: all peers identical, most modem blackholes
  - connection preferencing:
    - favors routing to well-connected peers
    - favors reply to clients that themselves serve large number of files: prevent freeloading
  - Limewire gateway functions as Napster-like central server on behalf of other peers (for searching purposes)

Anonymous?

- Not anymore than it's scalable.
- The person you are getting the file from knows who you are. That's not anonymous.

- Other protocols exist where the owner of the files doesn't know the requester.

- Peer-to-peer anonymity exists.
- See Eternity and, next, Freenet!

Gnutella Discussion:

- Architectural lessons learned?
- Do Gnutella's goals seem familiar? Does it work better than say squid or summary cache? Or multicast with carousel?
- anonymity and security?
- Other?
- Good source for technical info/open questions:

  http://www.limewire.com/index.jsp/tech_papers

Lecture 3: Distributed Hash Tables

- Can we go from content to location in one go?
- Can we still retain locality?
- Can we keep any anonimity
- Look at Chord, Tapestry, CAN (pastry is similar… … …)
- Notice how networking people like silly names☺

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

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

MIT and Berkeley

Now we see some CS in strength – Hash and Content based….for more scaleble (distributed) directory lookup

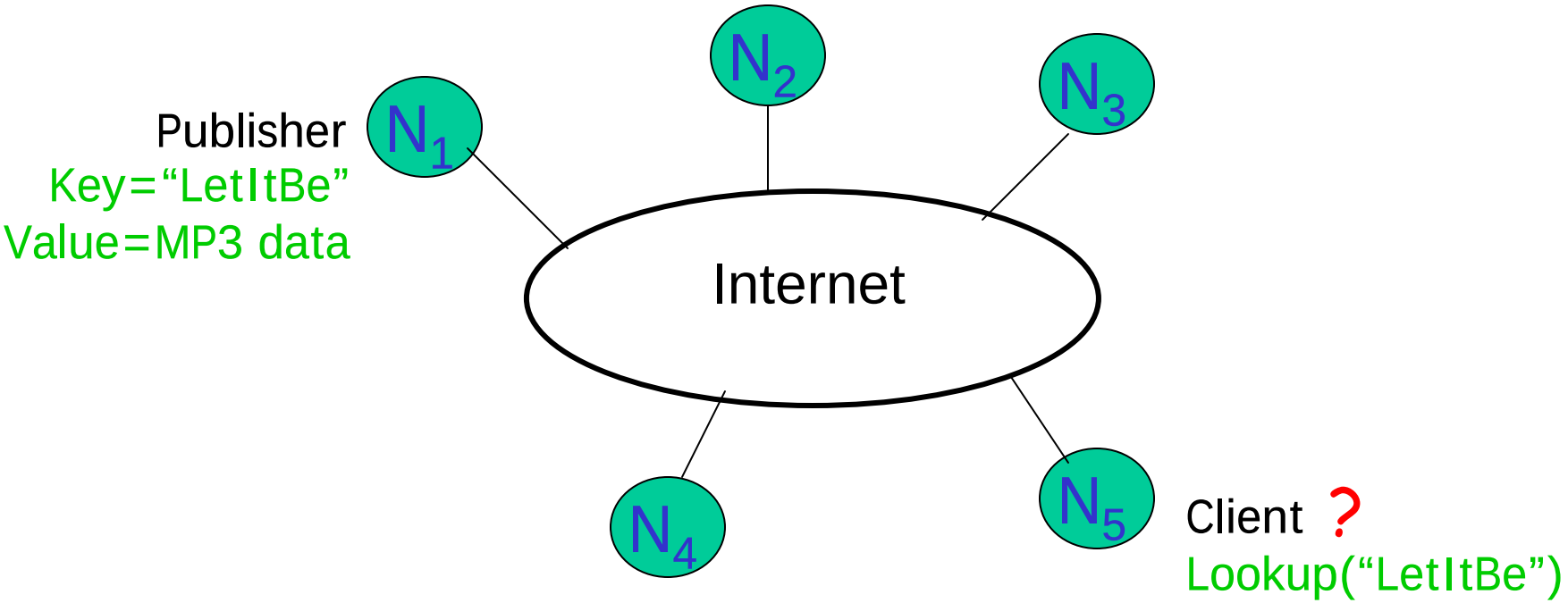
- presentation based on slides by Robert Morris (SIGCOMM'01)

Outline

- Motivation and background
- Consistency caching
- Chord
- Performance evaluation
- Conclusion and discussion

Motivation

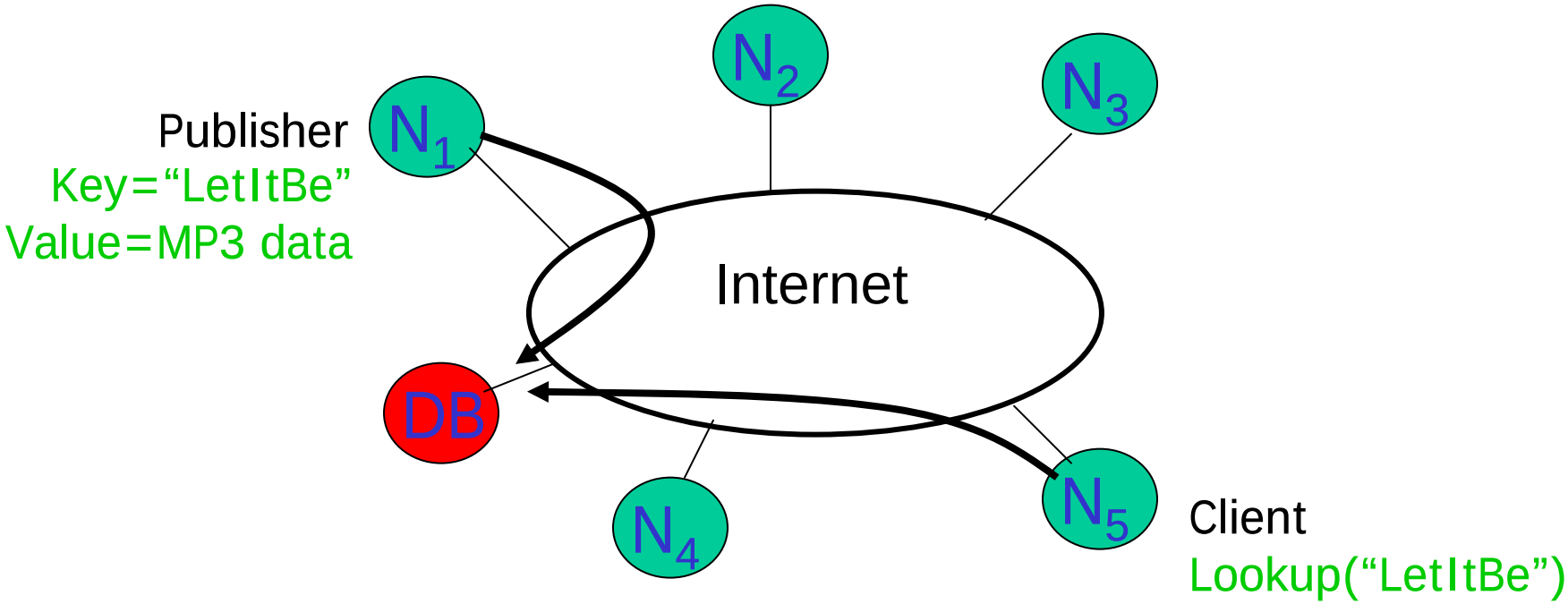
How to find data in a distributed file sharing system?



- Lookup is the key problem

Centralized Solution

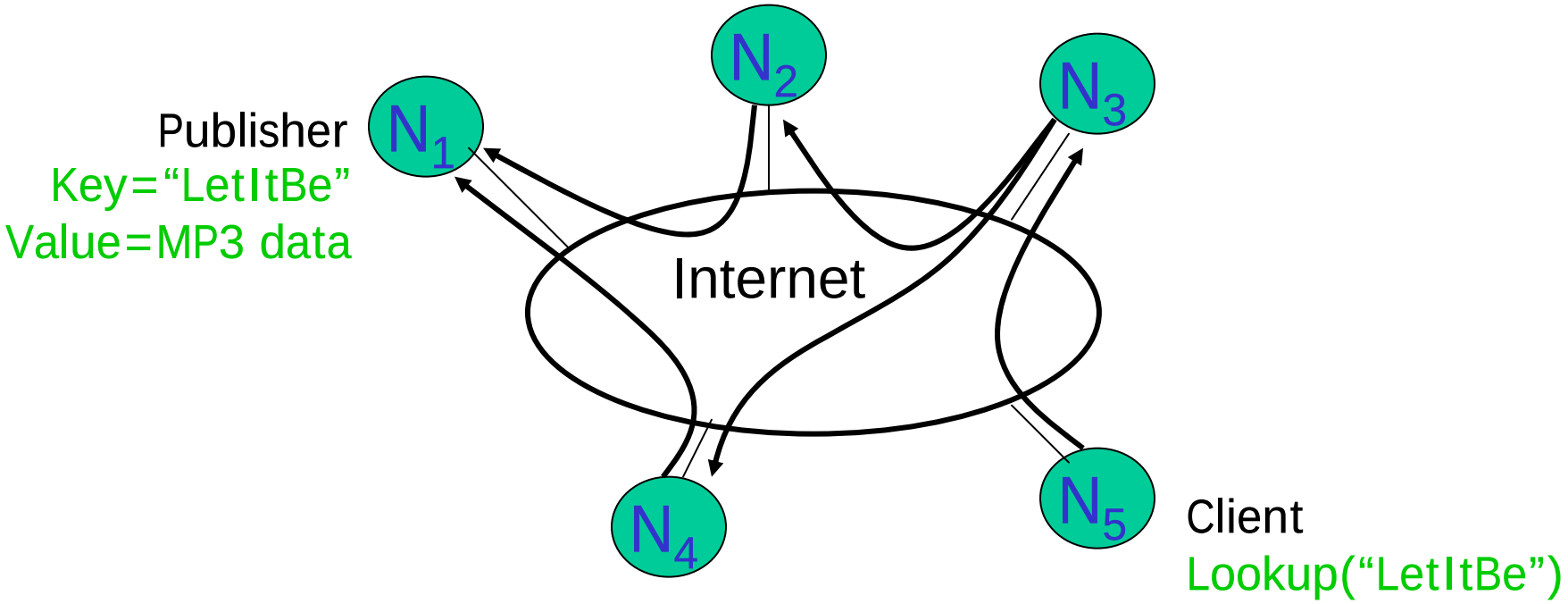
- Central server (Napster)



- Requires O(M) state
- Single point of failure

Distributed Solution (1)

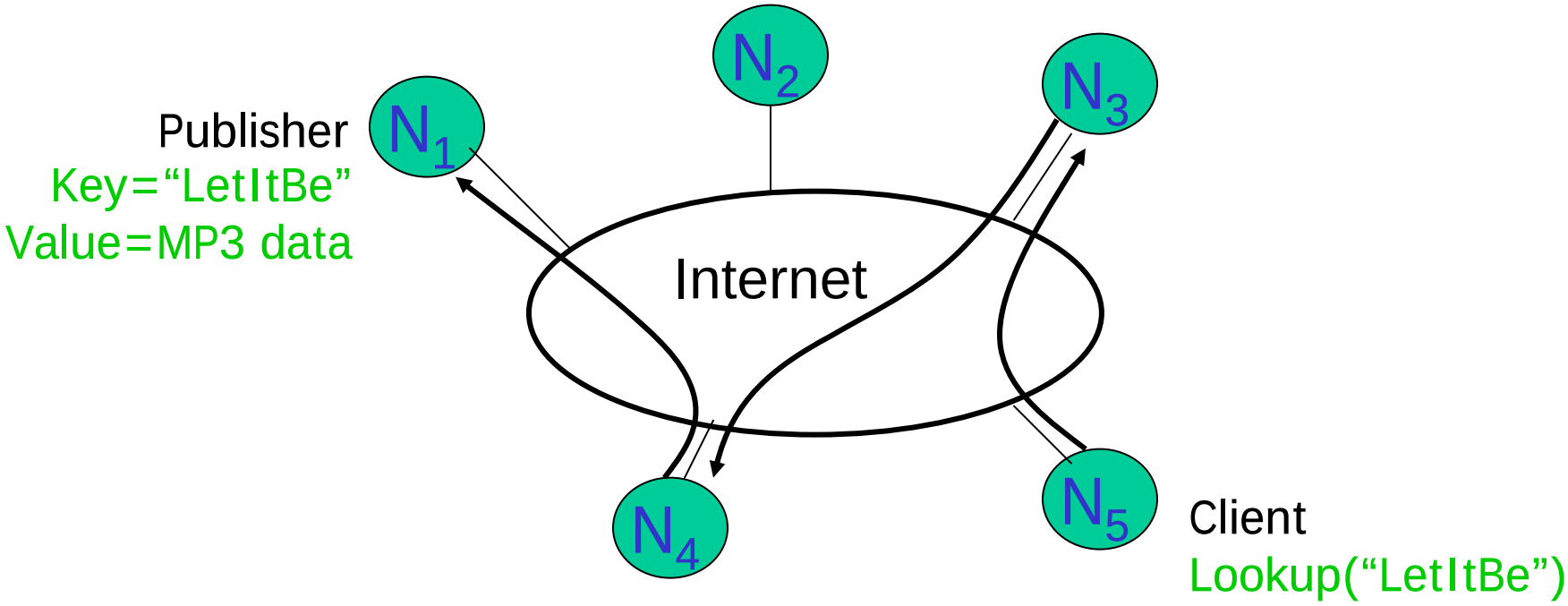
- Flooding (Gnutella, Morpheus, etc.)



- Worst case O(N) messages per lookup

Distributed Solution (2)

- Routed messages (Freenet, Tapestry, Chord, CAN, etc.)



- Only exact matches

Routing Challenges

- Define a useful key nearness metric
- Keep the hop count small
- Keep the routing tables "right size"
- Stay robust despite rapid changes in membership

Authors claim:

- Chord: emphasizes efficiency and simplicity

Chord Overview

- Provides peer-to-peer hash lookup service:
  - Lookup(key) → IP address
  - Chord does not store the data
- How does Chord locate a node?
- How does Chord maintain routing tables?
- How does Chord cope with changes in membership?

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 changes in membership

- Proofs are in paper / tech report
  - Assuming no malicious participants

Chord IDs

- *m* bit identifier space for both keys and nodes
- Key identifier = SHA-1(key)

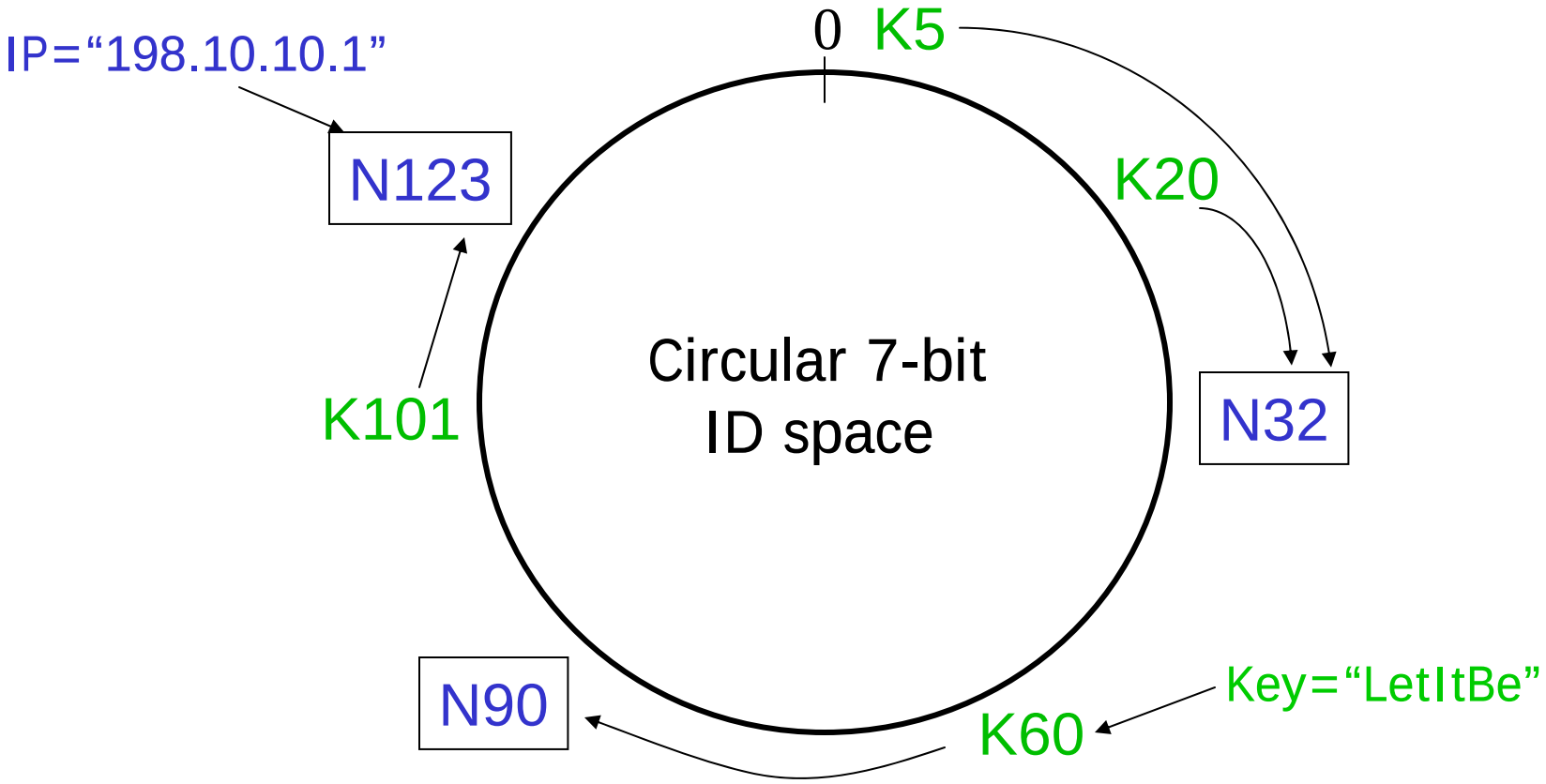
Key="LetItBe" —SHA-1→ ID=60

- Node identifier = SHA-1(IP address)

IP="198.10.10.1" —SHA-1→ ID=123

- Both are uniformly distributed
- How to map key IDs to node IDs?

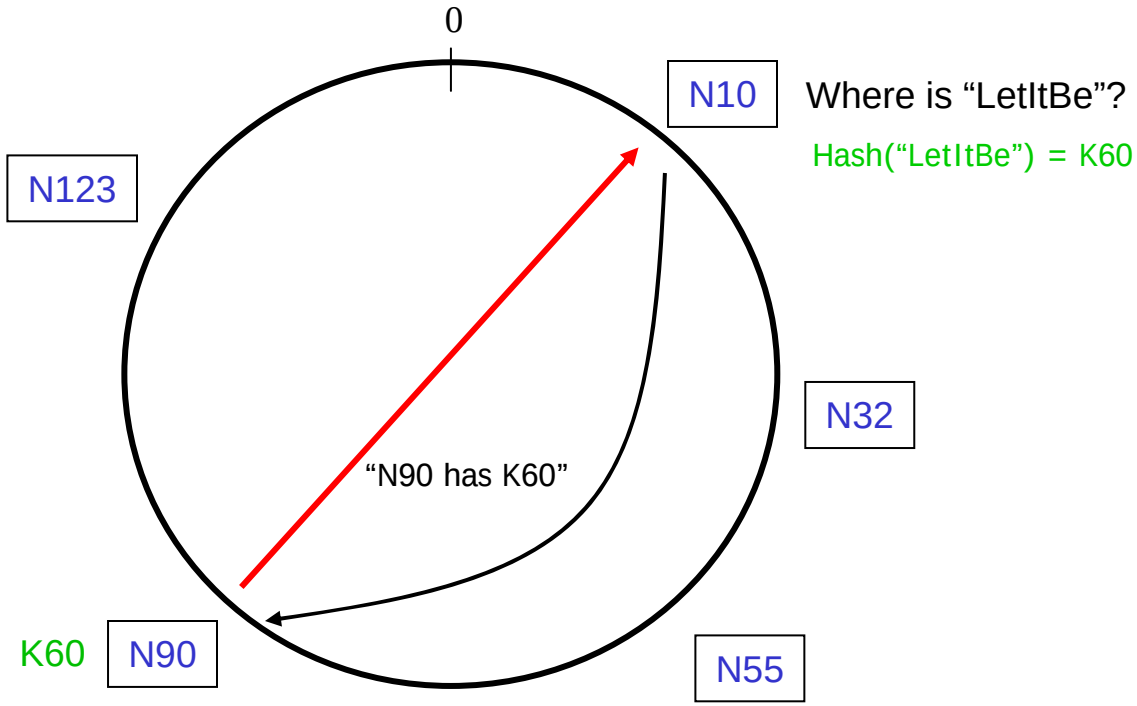
Consistent Hashing [Karger 97]



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

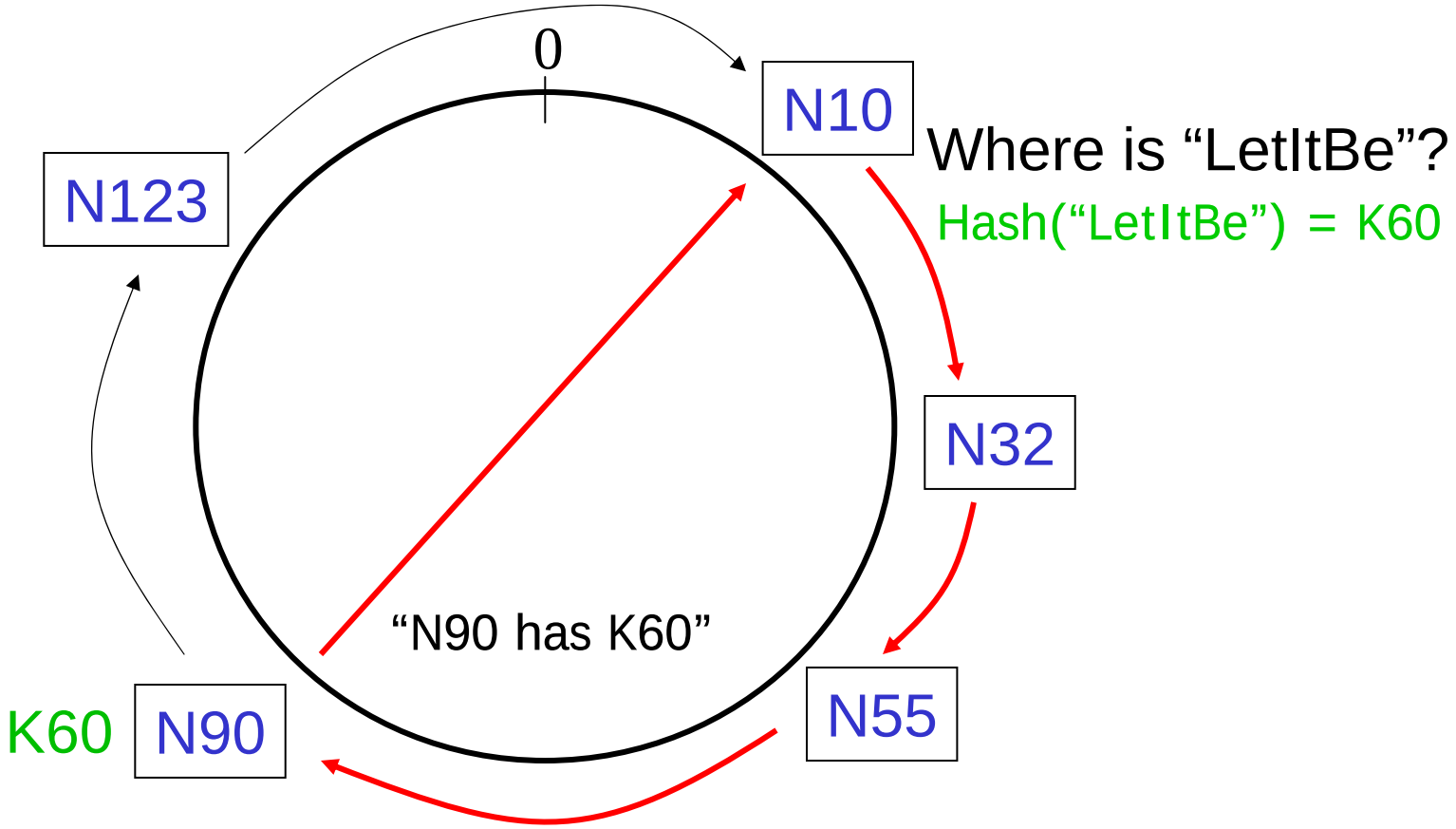
Consistent Hashing

- Every node knows of every other node
  - requires global information
- Routing tables are large O(N)
- Lookups are fast O(1)



Chord: Basic Lookup

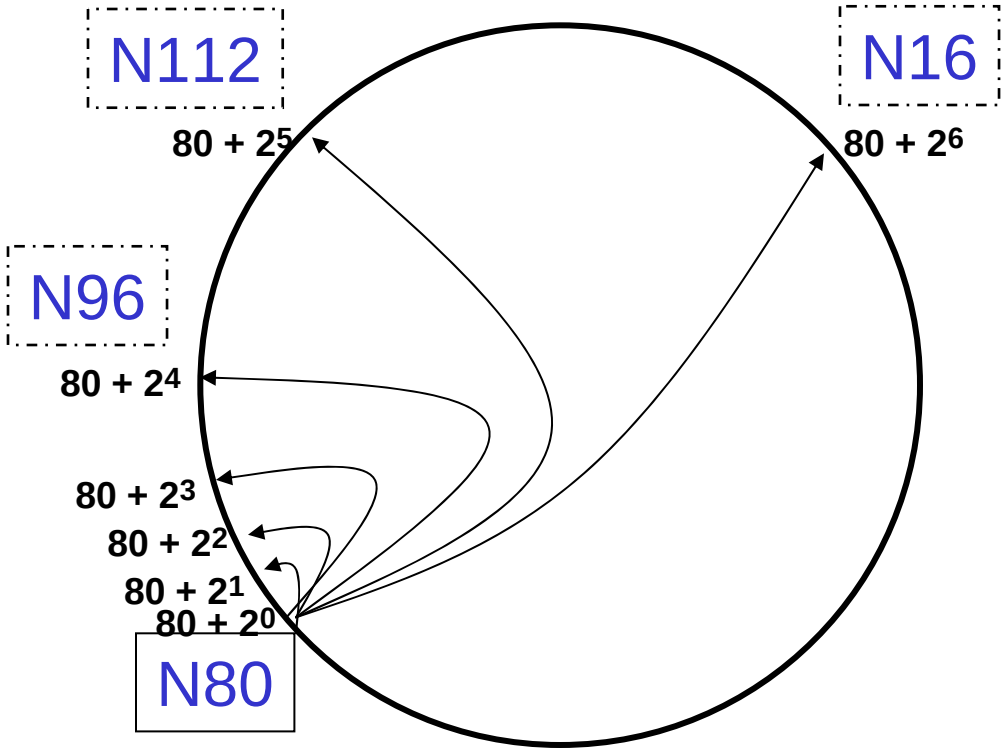
- Every node knows its successor in the ring



- requires O(N) time

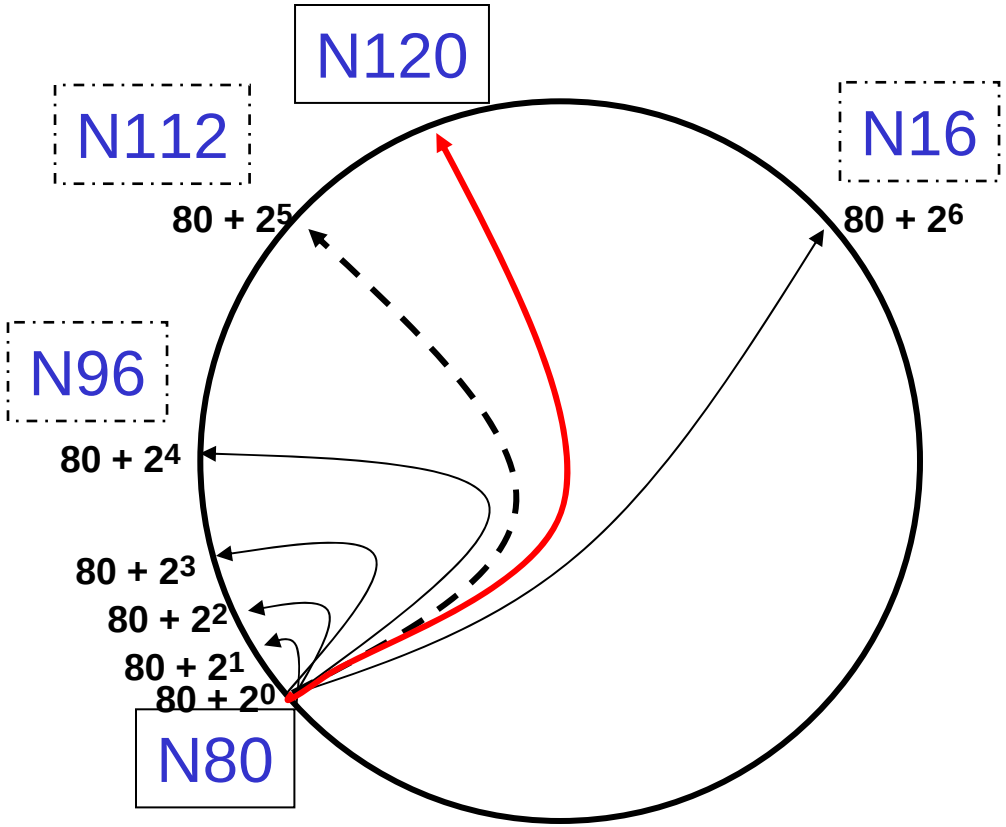
"Finger Tables"

- Every node knows *m* other nodes in the ring
- Increase distance exponentially



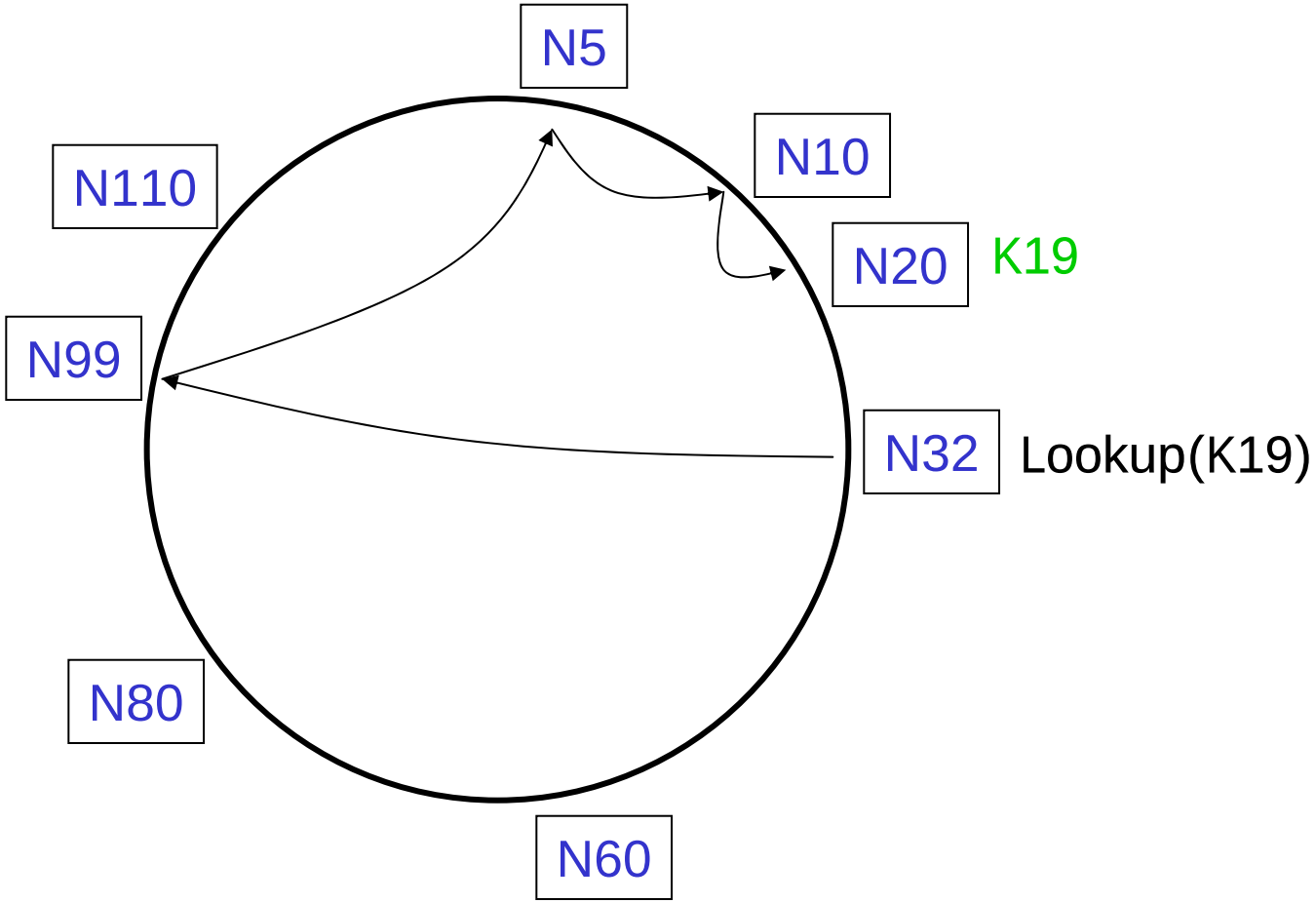
"Finger Tables"

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



Lookups are Faster

- Lookups take O(Log N) hops

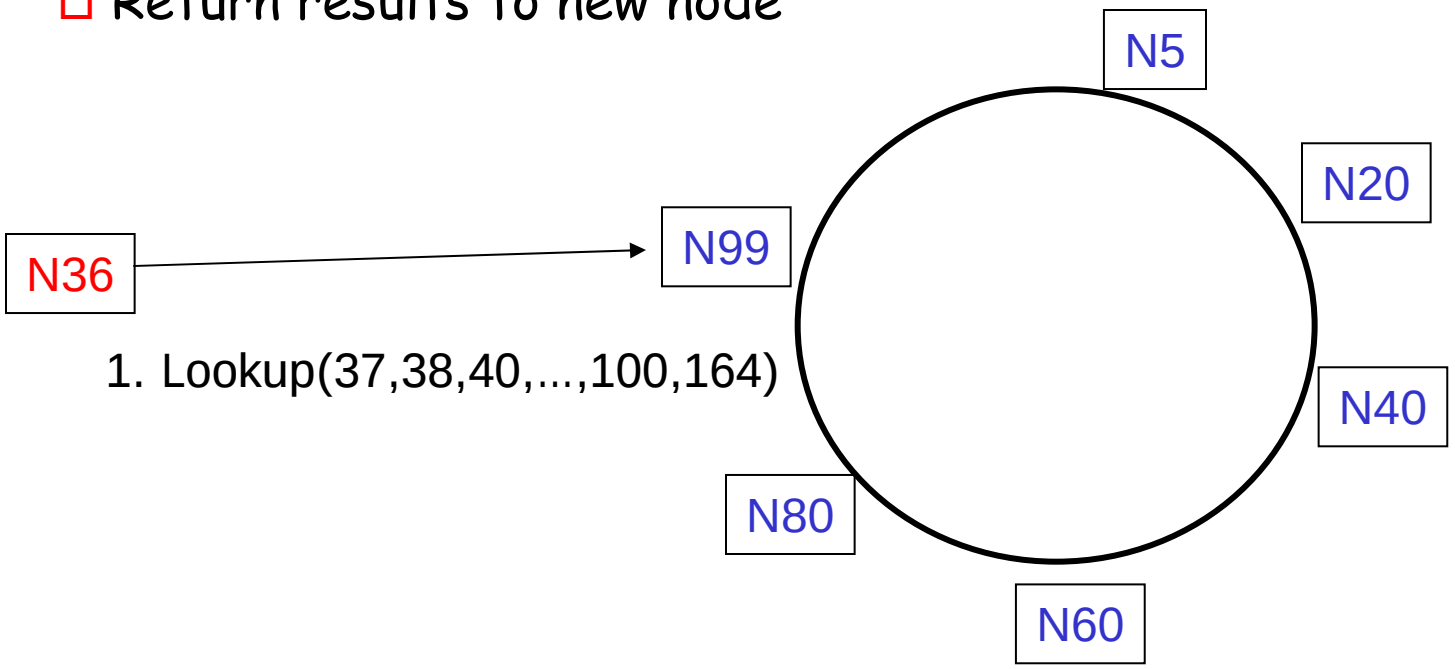


Joining the Ring

- Three step process:
  - Initialize all fingers of new node
  - Update fingers of existing nodes
  - Transfer keys from successor to new node
- Less aggressive mechanism (lazy finger update):
  - Initialize only the finger to successor node
  - Periodically verify immediate successor, predecessor
  - Periodically refresh finger table entries

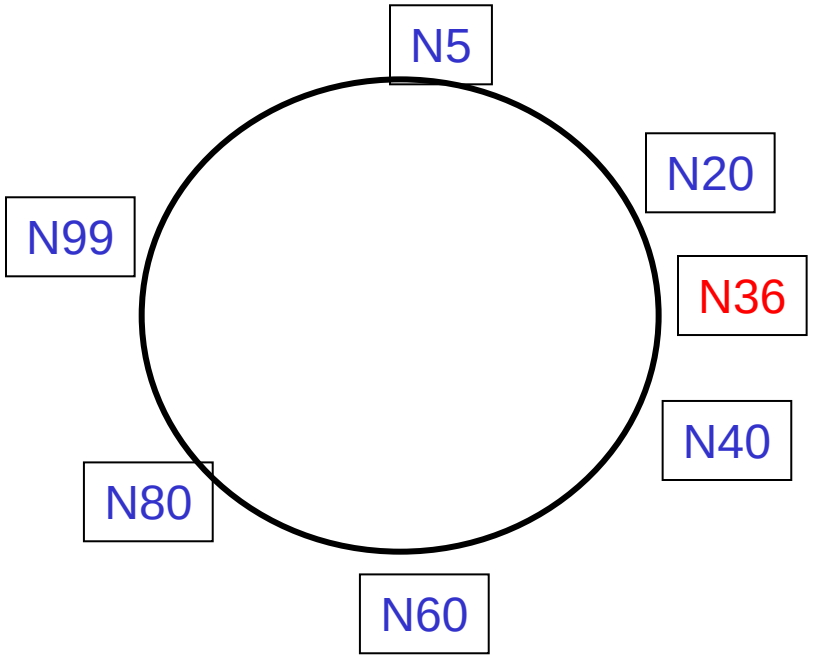
Joining the Ring - Step 1

- Initialize the new node finger table
  - Locate any node *p* in the ring
  - Ask node *p* to lookup fingers of new node N36
  - Return results to new node



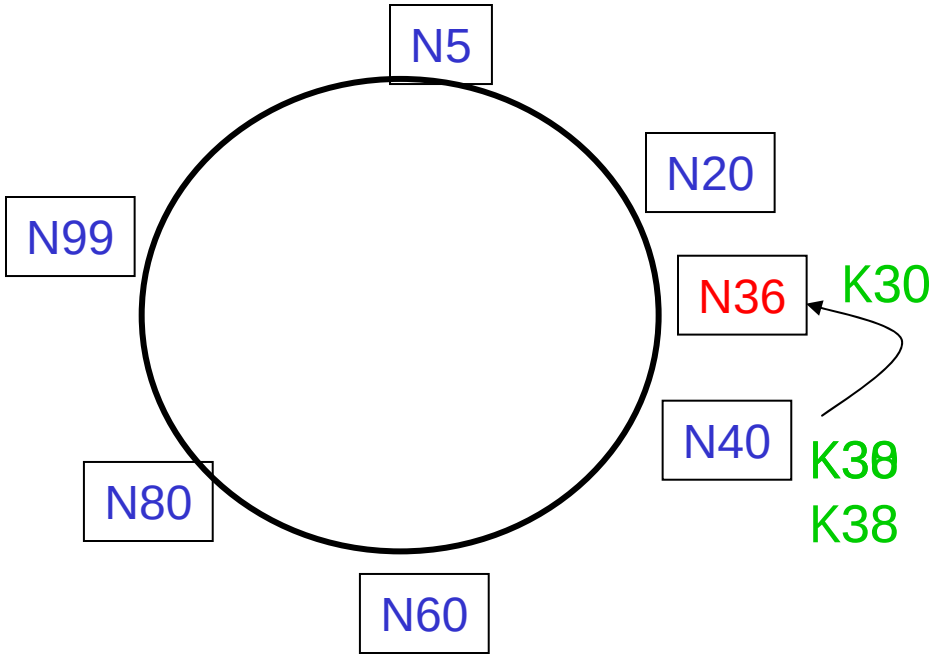
Joining the Ring - Step 2

- Updating fingers of existing nodes
  - new node calls update function on existing nodes
  - existing nodes can recursively update fingers of other nodes



Joining the Ring - Step 3

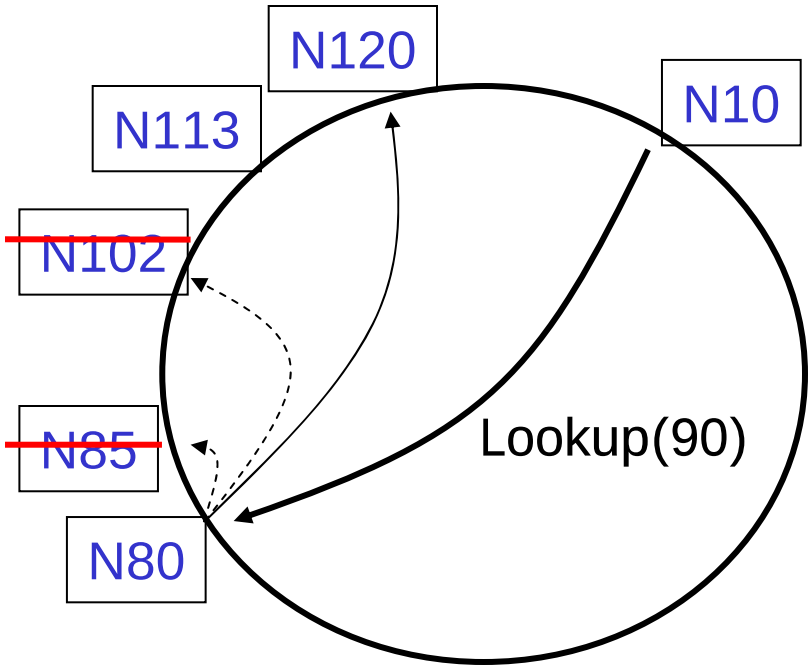
- Transfer keys from successor node to new node
  - only keys in the range are transferred



Copy keys 21..36
from N40 to N36

Handing Failures

- Failure of nodes might cause incorrect lookup



- N80 doesn't know correct successor, so lookup fails
- Successor fingers are enough for correctness

Handling Failures

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

- Guarantee is with some probability
  - Can choose *r* to make probability of lookup failure arbitrarily small

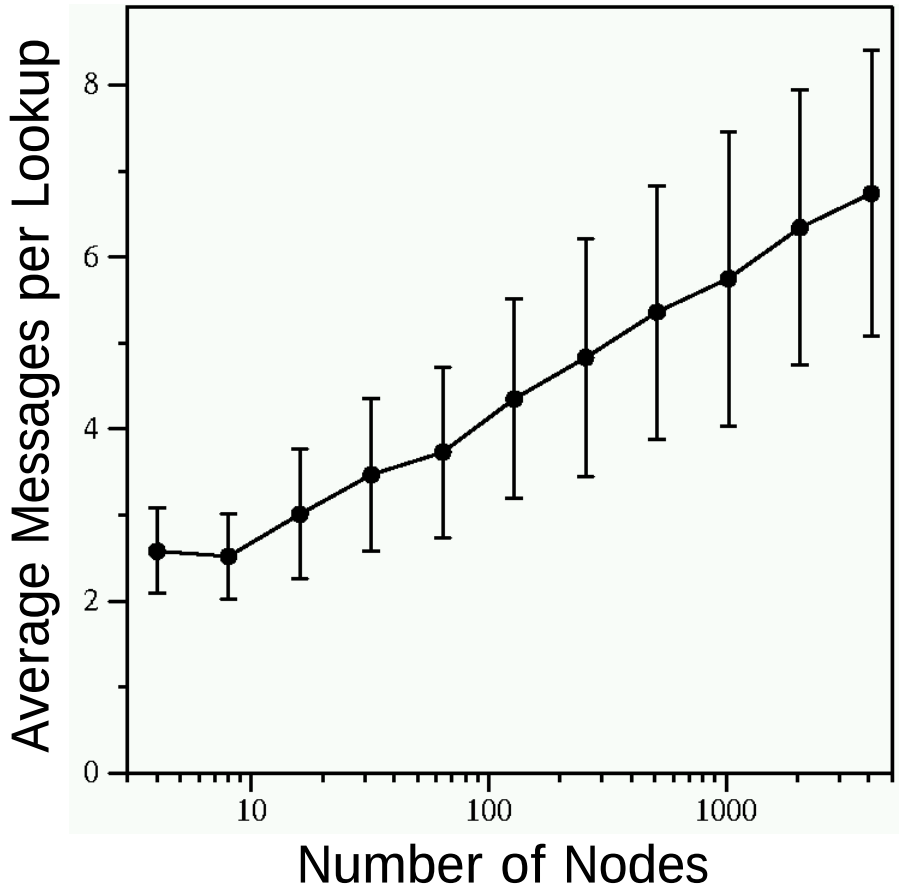
Evaluation Overview

- Quick lookup in large systems
- Low variation in lookup costs
- Robust despite massive failure

- Experiments confirm theoretical results

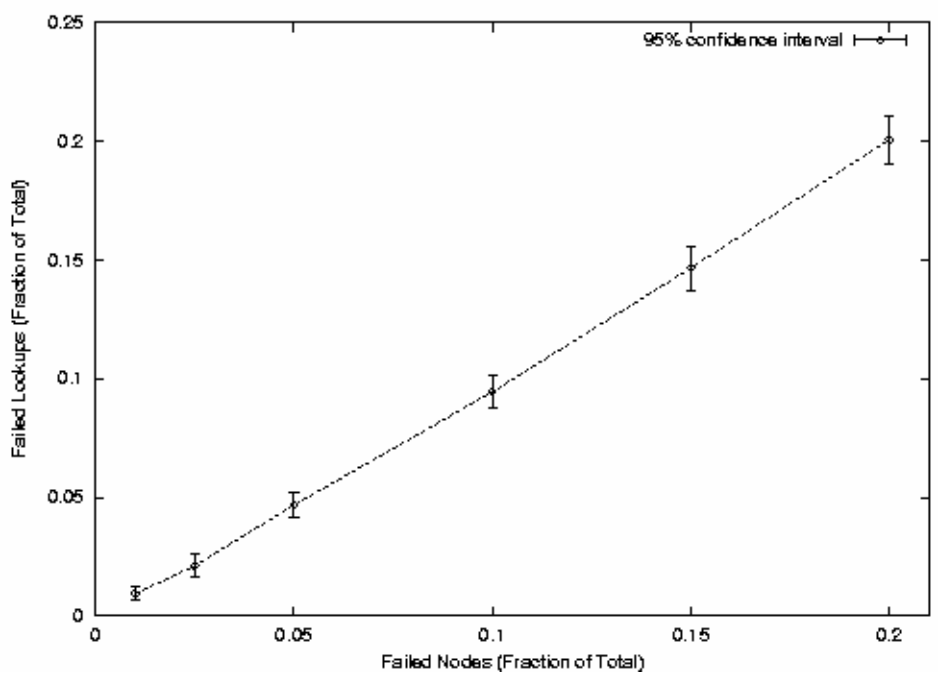
Cost of lookup

- Cost is O(Log N) as predicted by theory
- constant is 1/2



Robustness

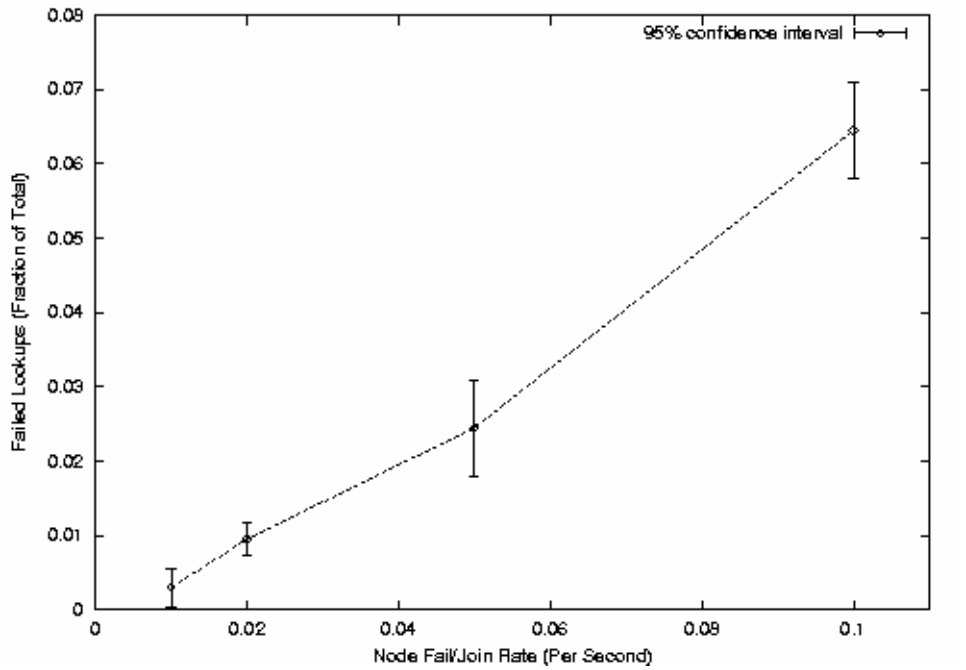
- Simulation results: static scenario
- Failed lookup means original node with key failed (no replica of keys)



- Result implies good balance of keys among nodes!

Robustness

- Simulation results: dynamic scenario
- Failed lookup means finger path has a failed node



- 500 nodes initially
- average *stabilize( )* call 30s
- 1 lookup per second (Poisson)
- x join/fail per second (Poisson)

Current implementation

- 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 (?)

Strengths

- Based on theoretical work (consistent hashing)
- Proven performance in many different aspects
  - "with high probability" proofs
- Robust (Is it?)

Weakness

- **NOT** that simple (compared to CAN)
- Member joining is complicated
  - aggressive mechanisms requires too many messages and updates
  - no analysis of convergence in lazy finger mechanism
- Key management mechanism mixed between layers
  - upper layer does insertion and handle node failures
  - Chord transfer keys when node joins (no leave mechanism!)
- Routing table grows with # of members in group
- Worst case lookup can be slow

Discussions

- Network proximity (consider latency?)
- Protocol security
  - Malicious data insertion
  - Malicious Chord table information
- Keyword search and indexing
- …

8. A Scalable, Content-Addressable Network

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

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

Outline

- Introduction
- Design
- Evaluation
- Strengths & Weaknesses
- 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,, FreeNet, MojoNation…
    - large-scale storage management systems
      - Publius, OceanStore,, 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 (w/ improvement)

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 ...

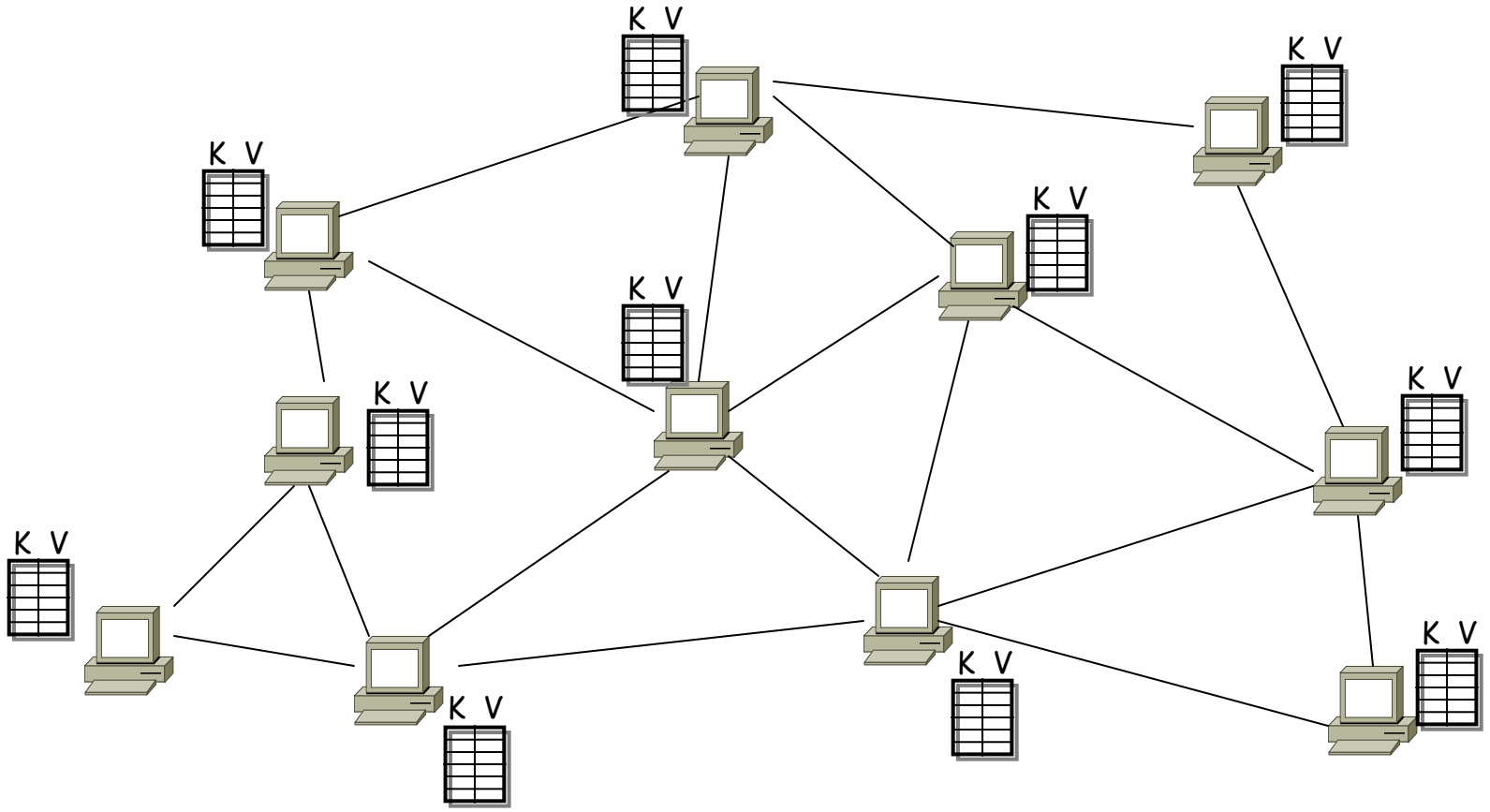
Problem Scope

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

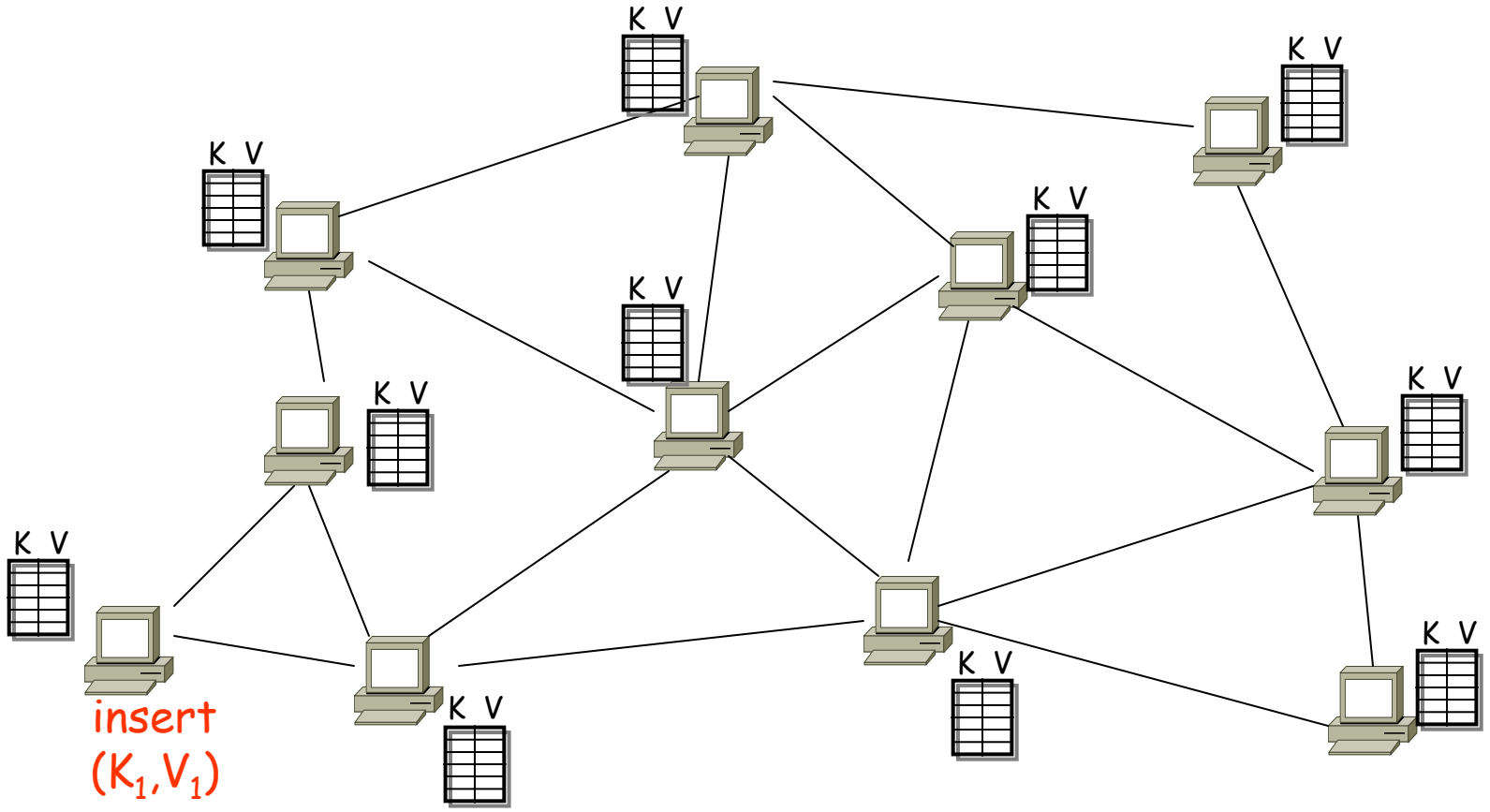
Outline

- Introduction
- Design
- Evaluation
- Strengths & Weaknesses
- Ongoing Work

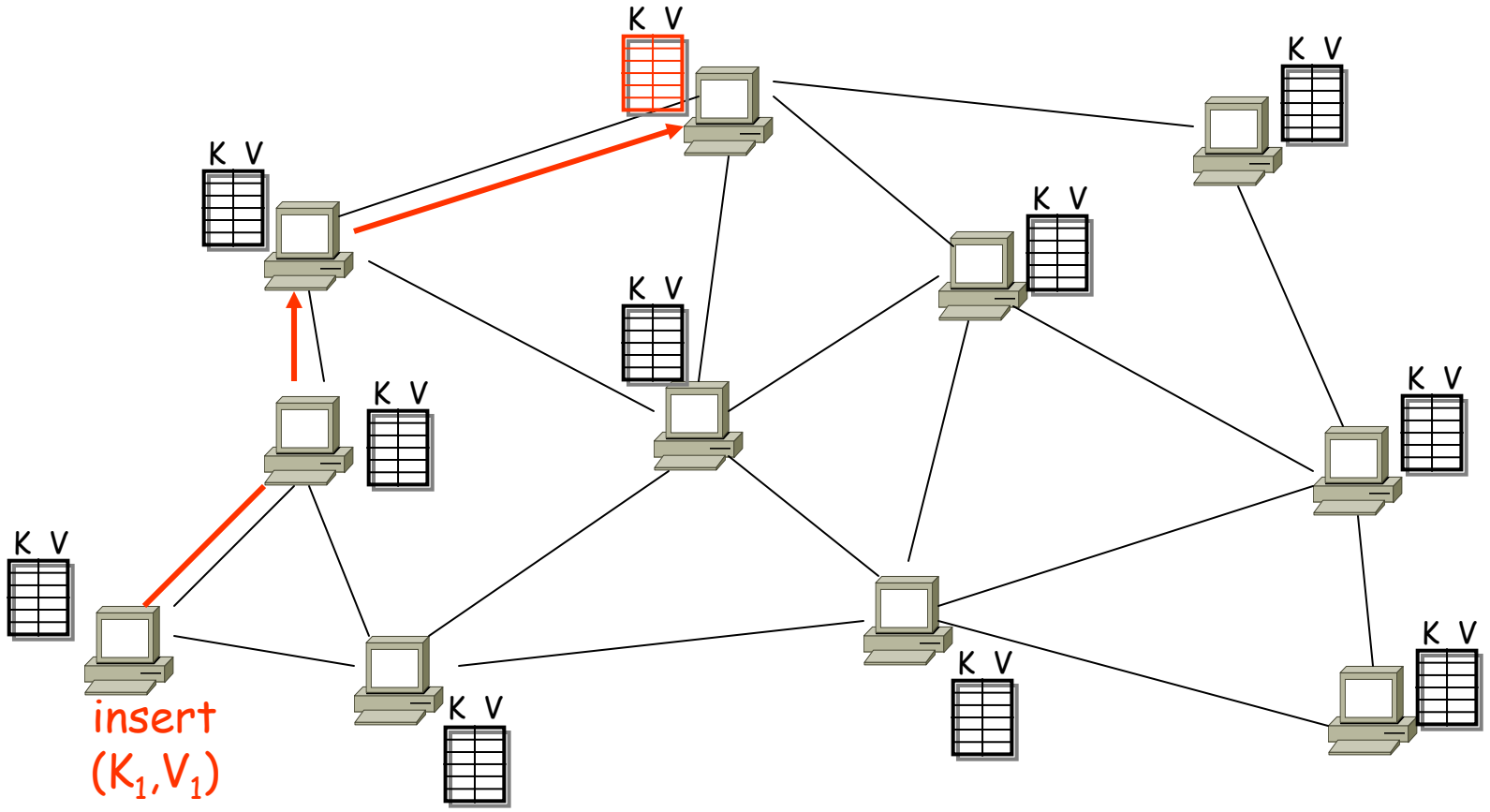
CAN: basic idea



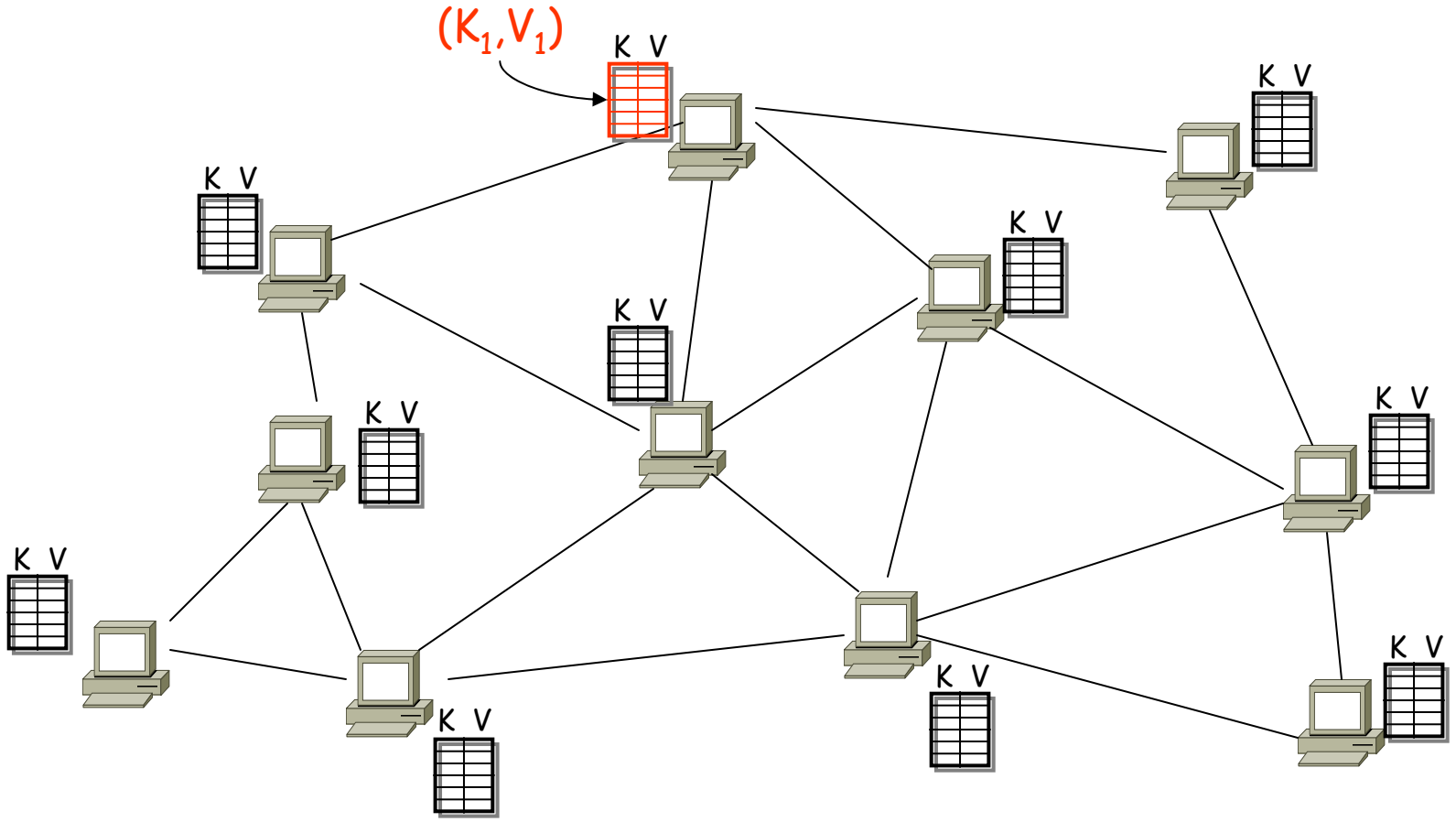
CAN: basic idea



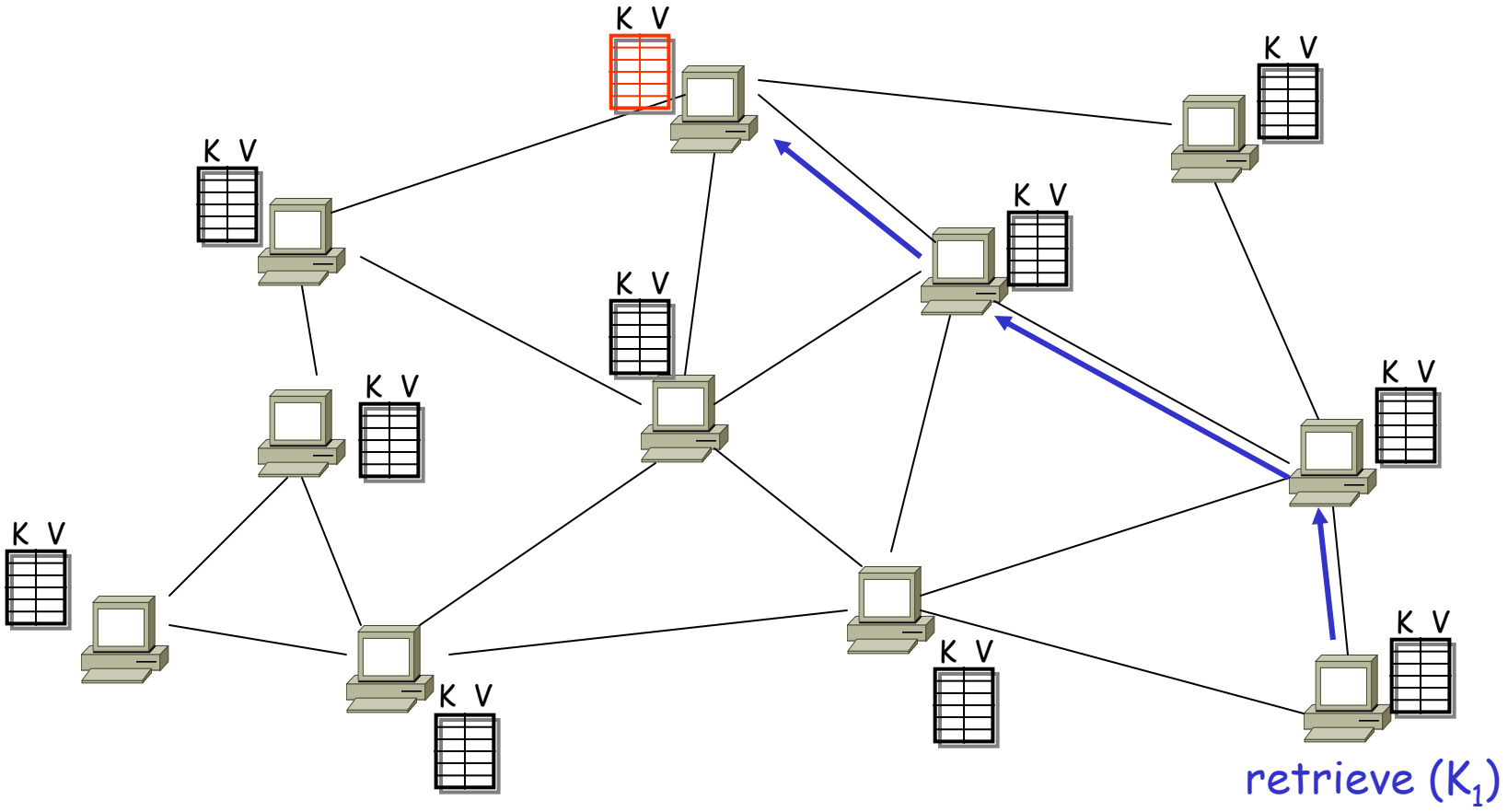
CAN: basic idea



CAN: basic idea



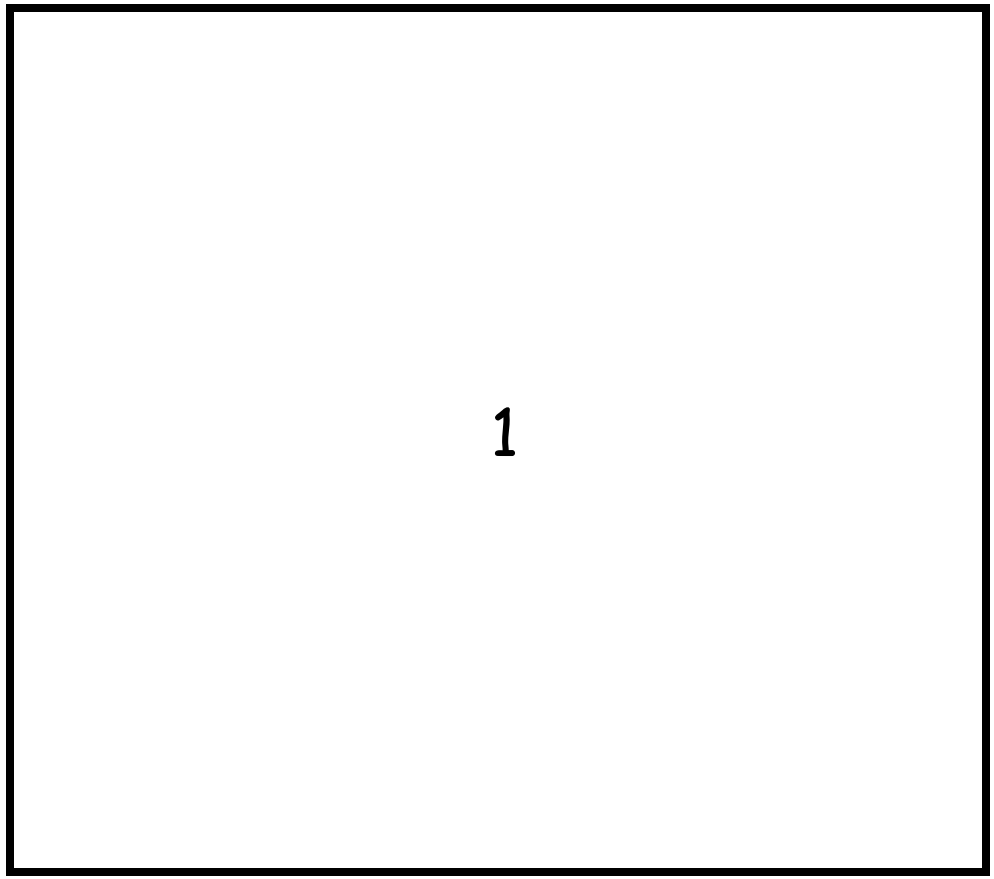
CAN: basic idea



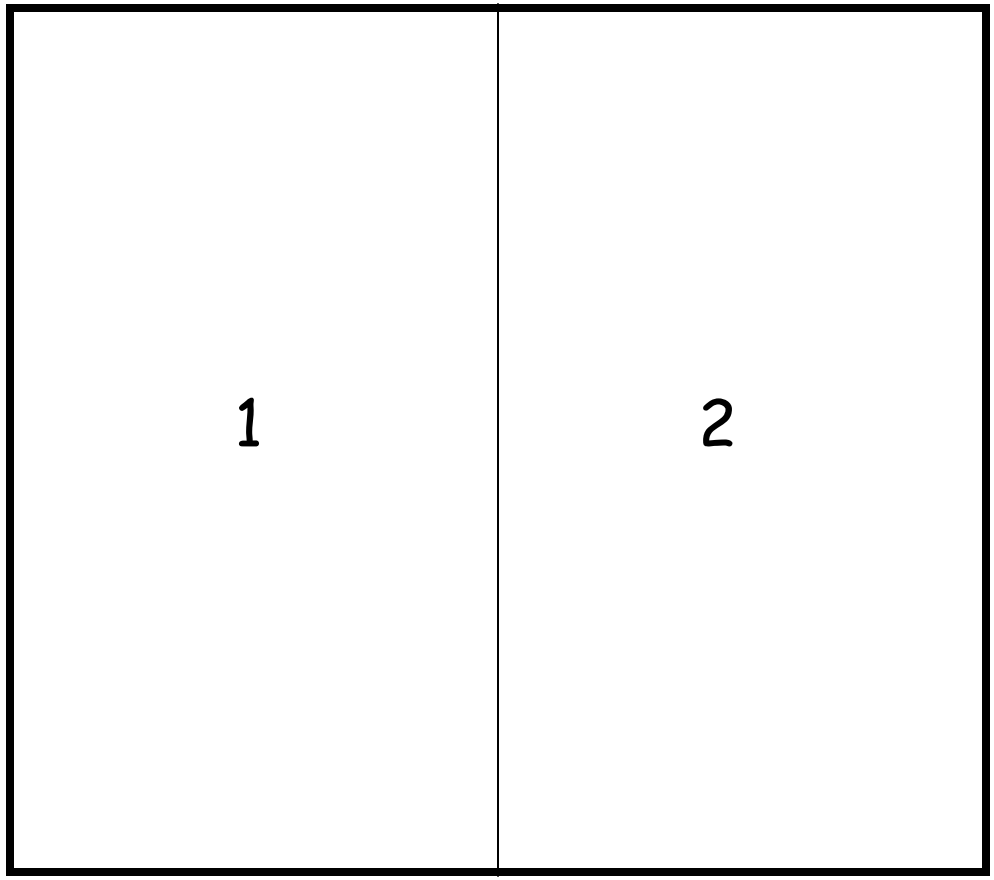
CAN: solution

- 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

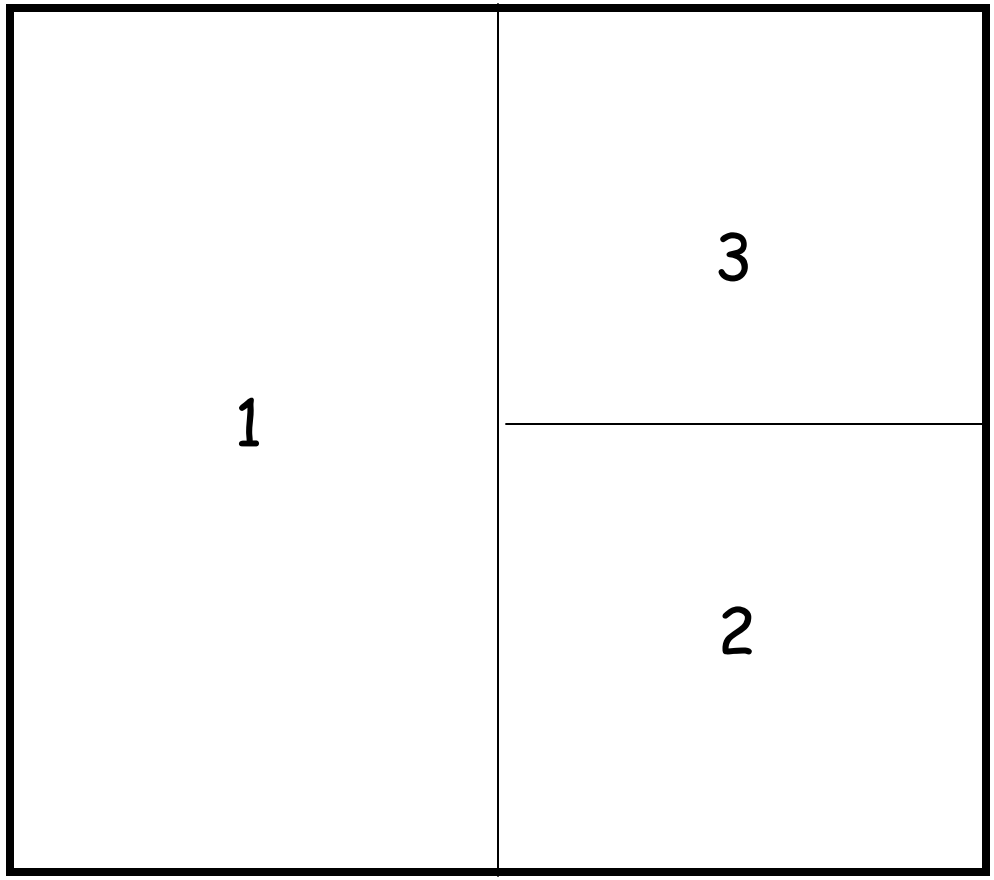
CAN: simple example



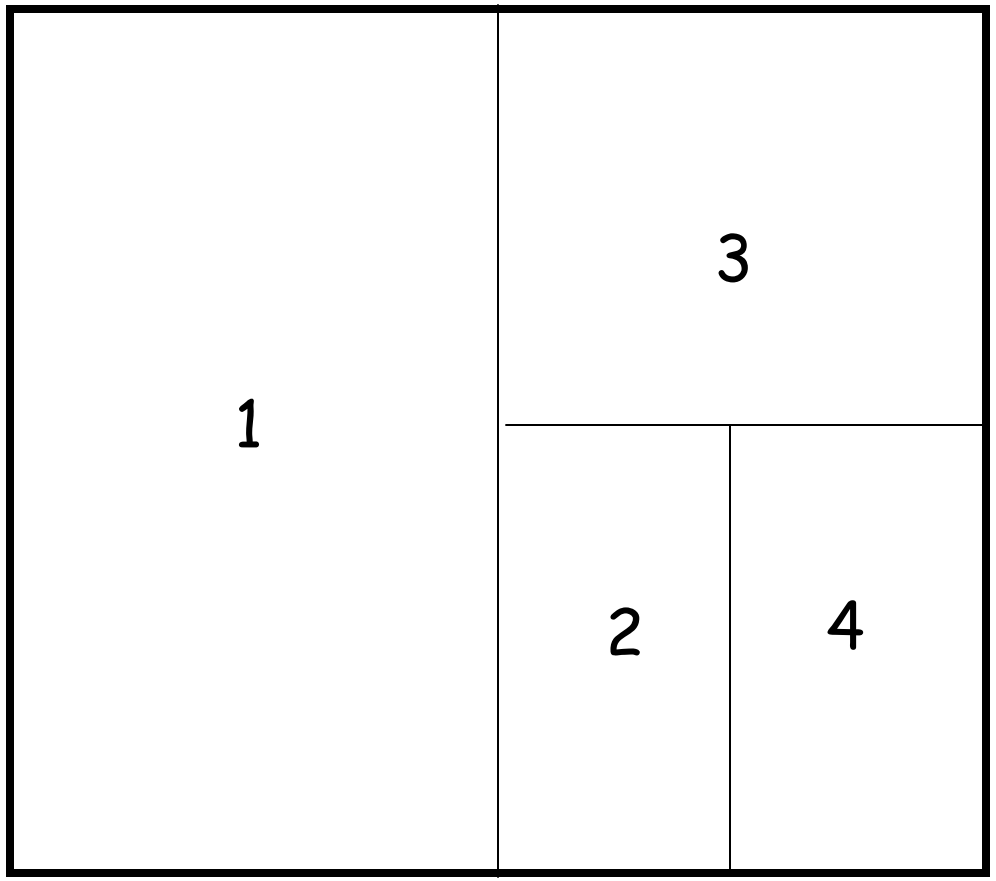
CAN: simple example



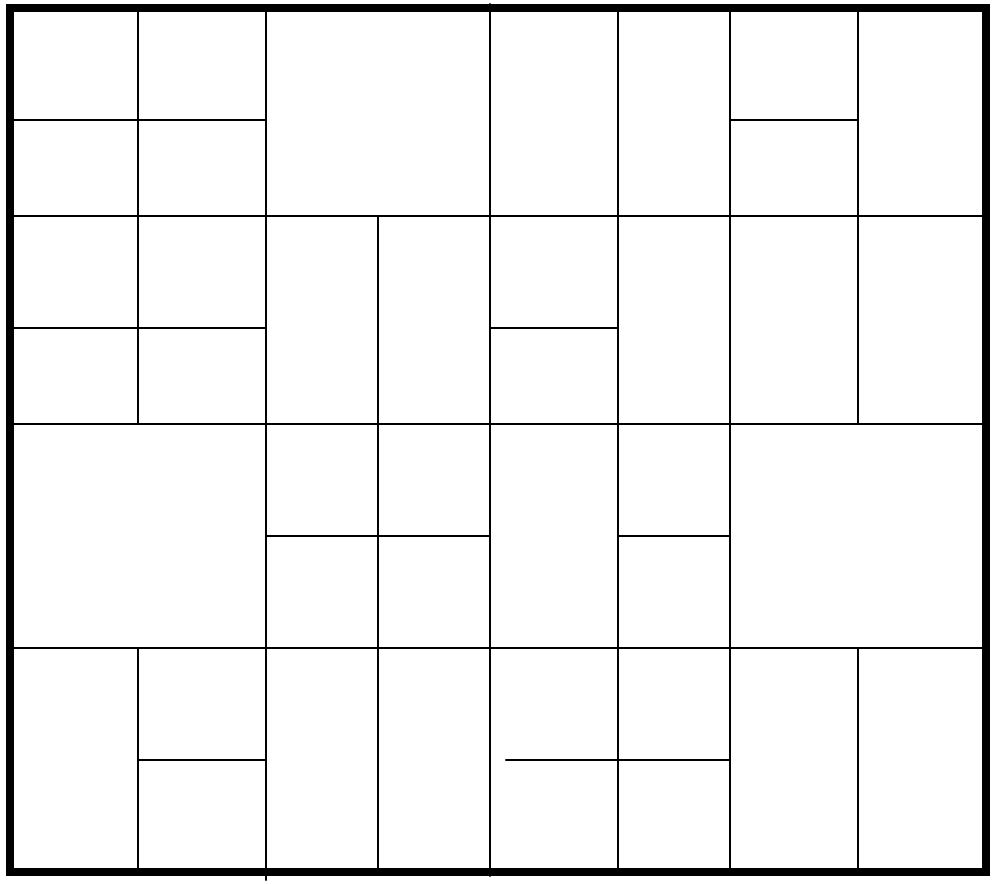
CAN: simple example



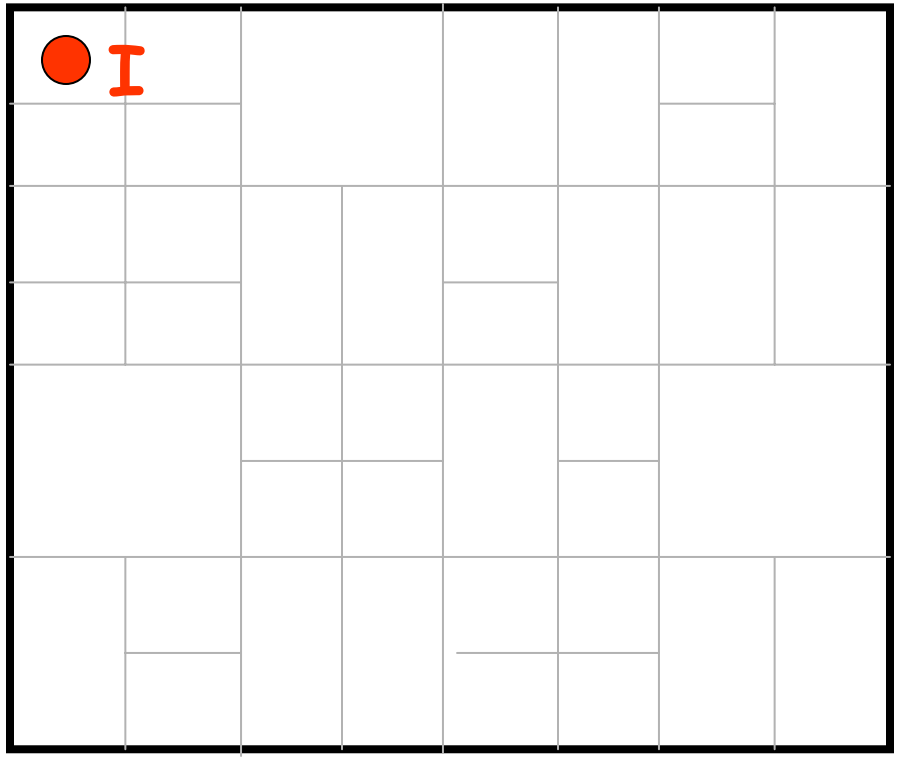
CAN: simple example



CAN: simple example

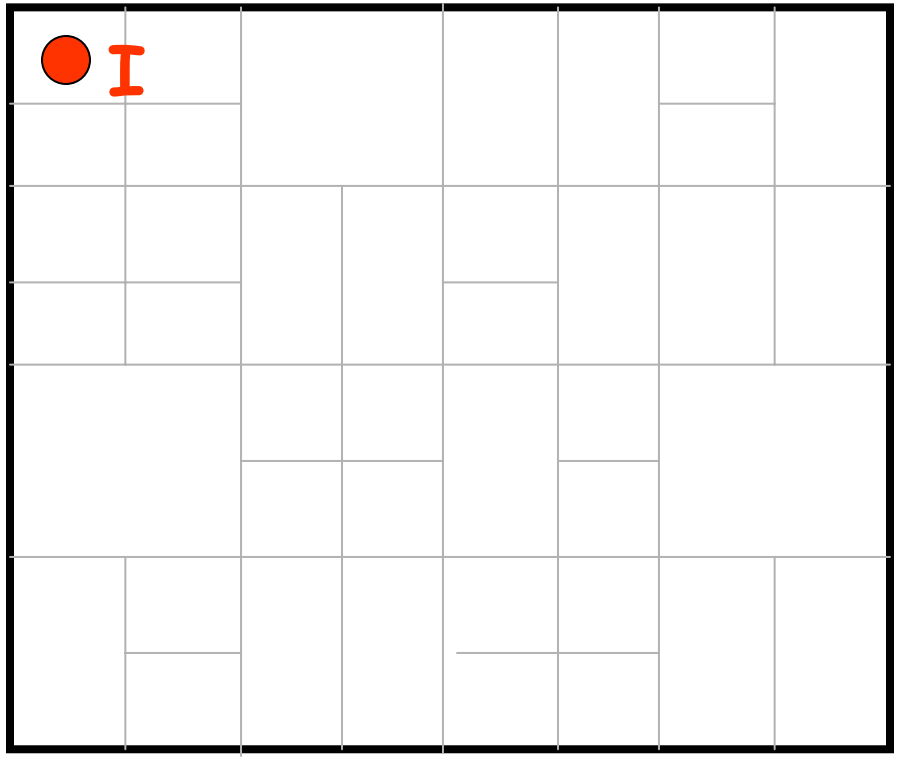


CAN: simple example



CAN: simple example

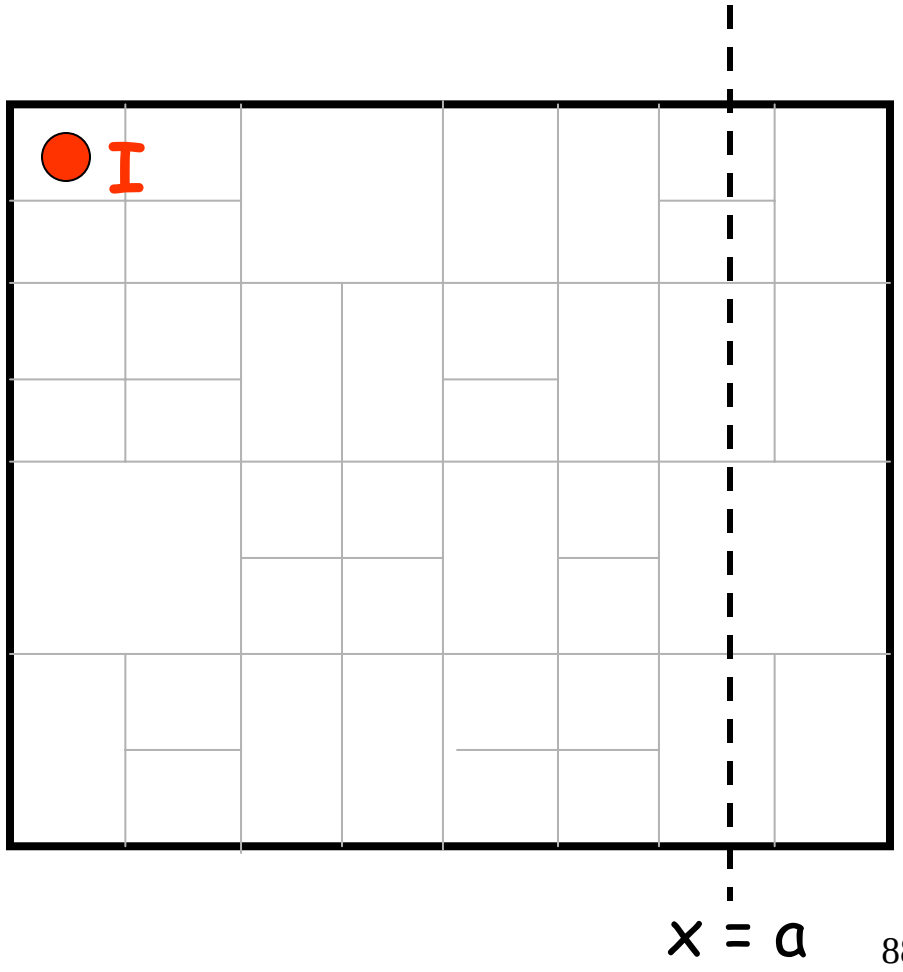
node I::insert(K,V)



CAN: simple example

node I::insert(K,V)

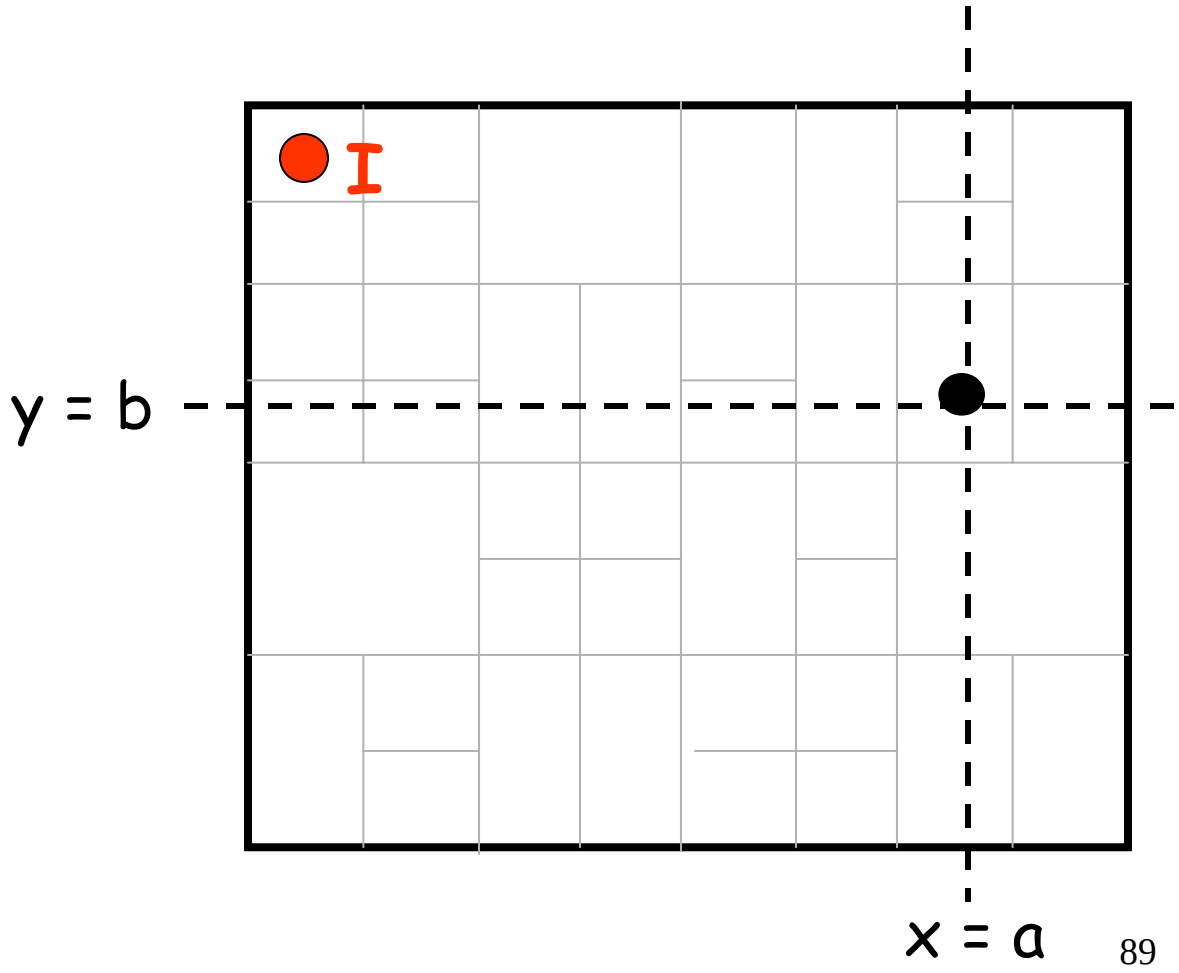
(1) a = h_x(K)



CAN: simple example

node I::insert(K,V)

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

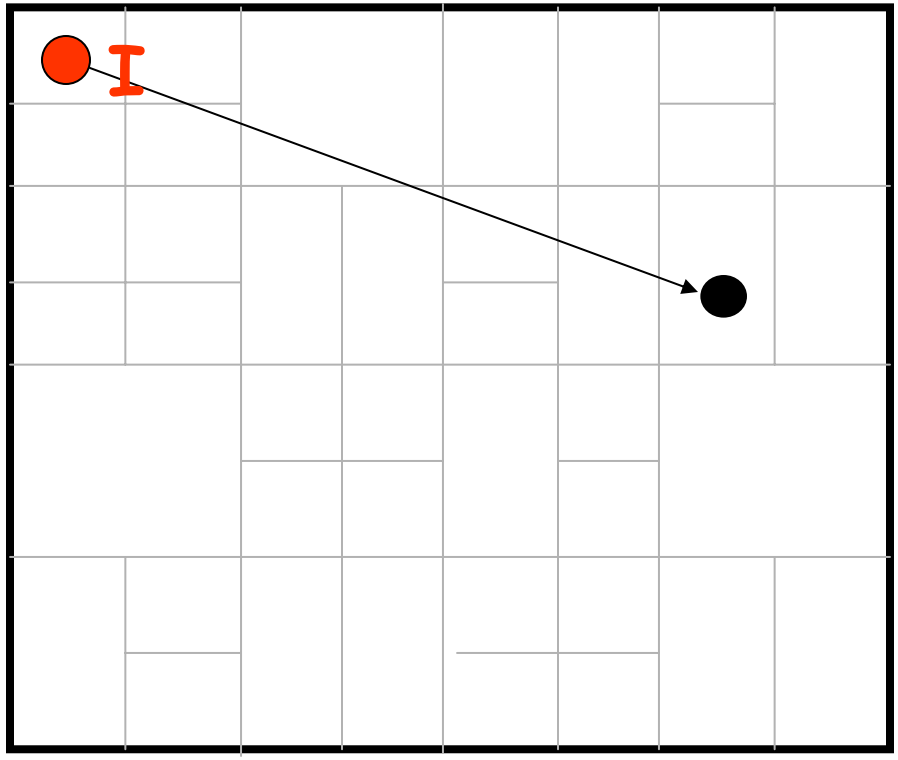


CAN: simple example

node I::insert(K,V)

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

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



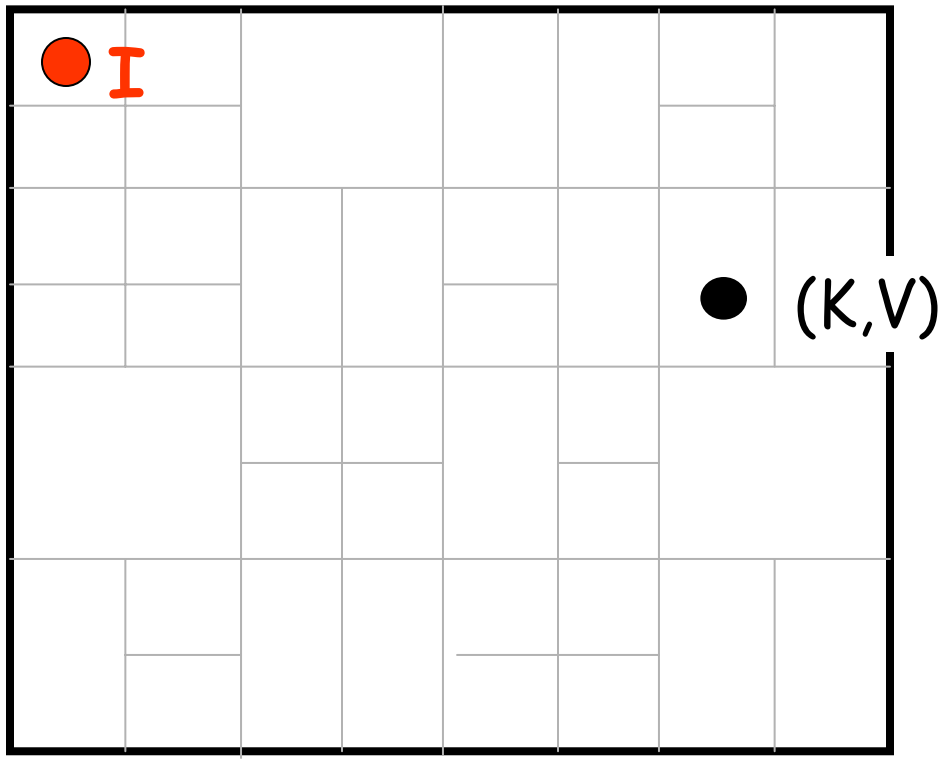
CAN: simple example

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)

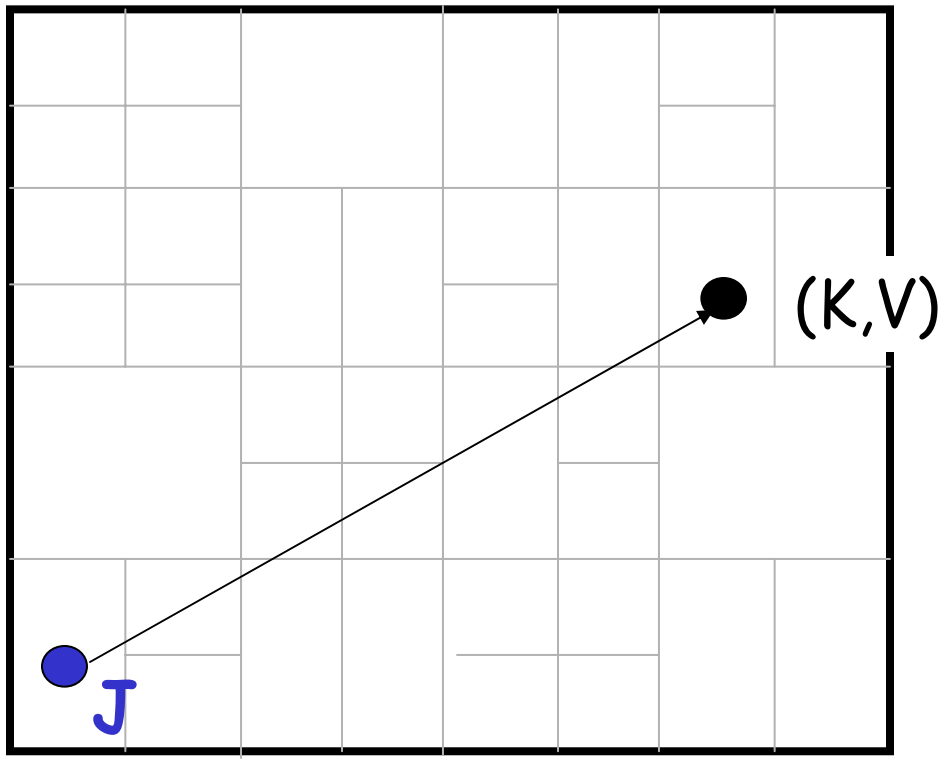


CAN: simple example

node J::retrieve(K)

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

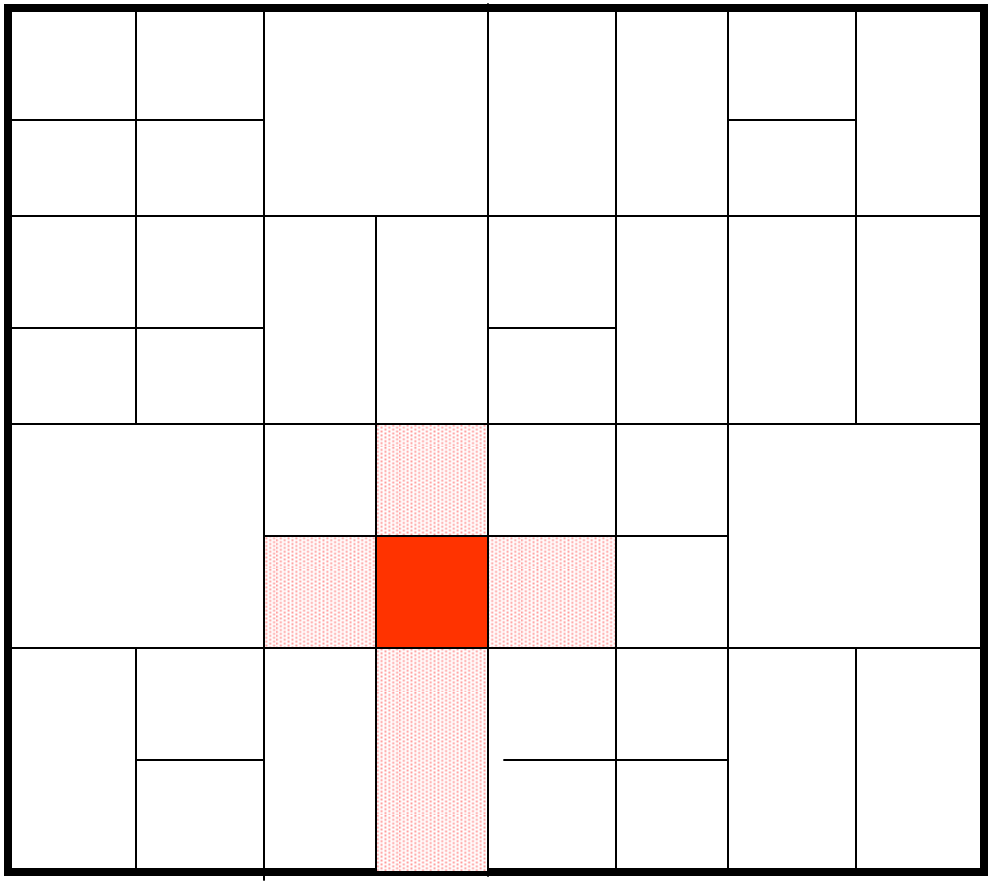
(2) route "retrieve(K)" to (a,b)



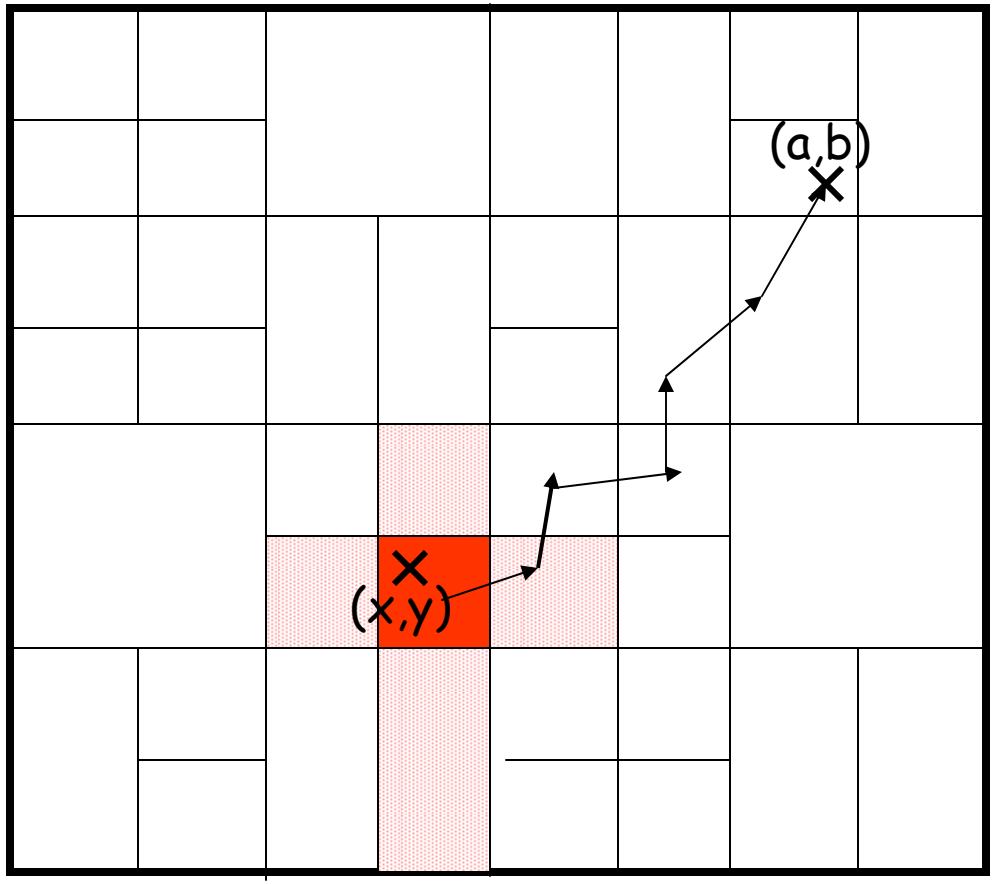
CAN

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

CAN: routing table



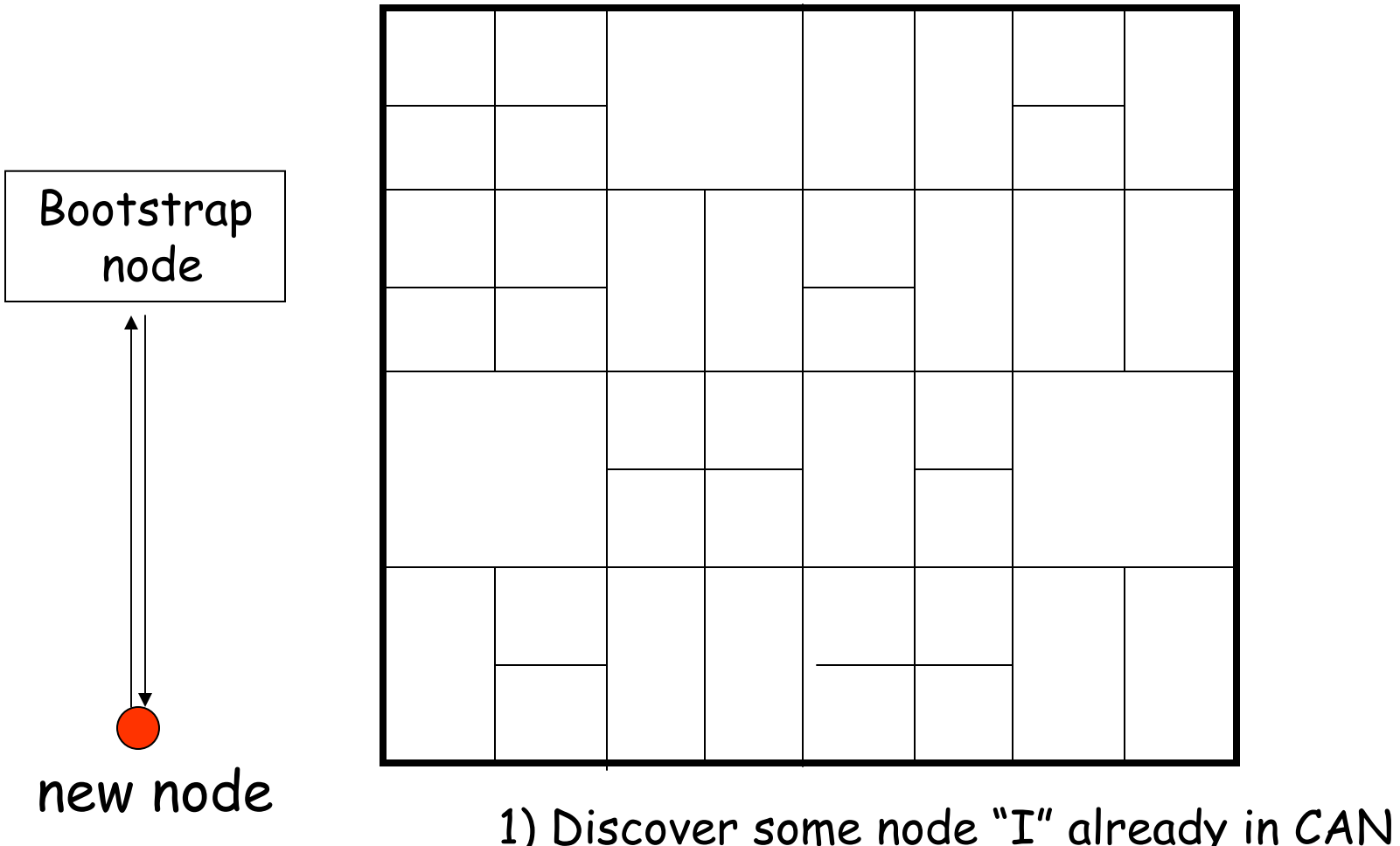
CAN: routing



CAN: routing

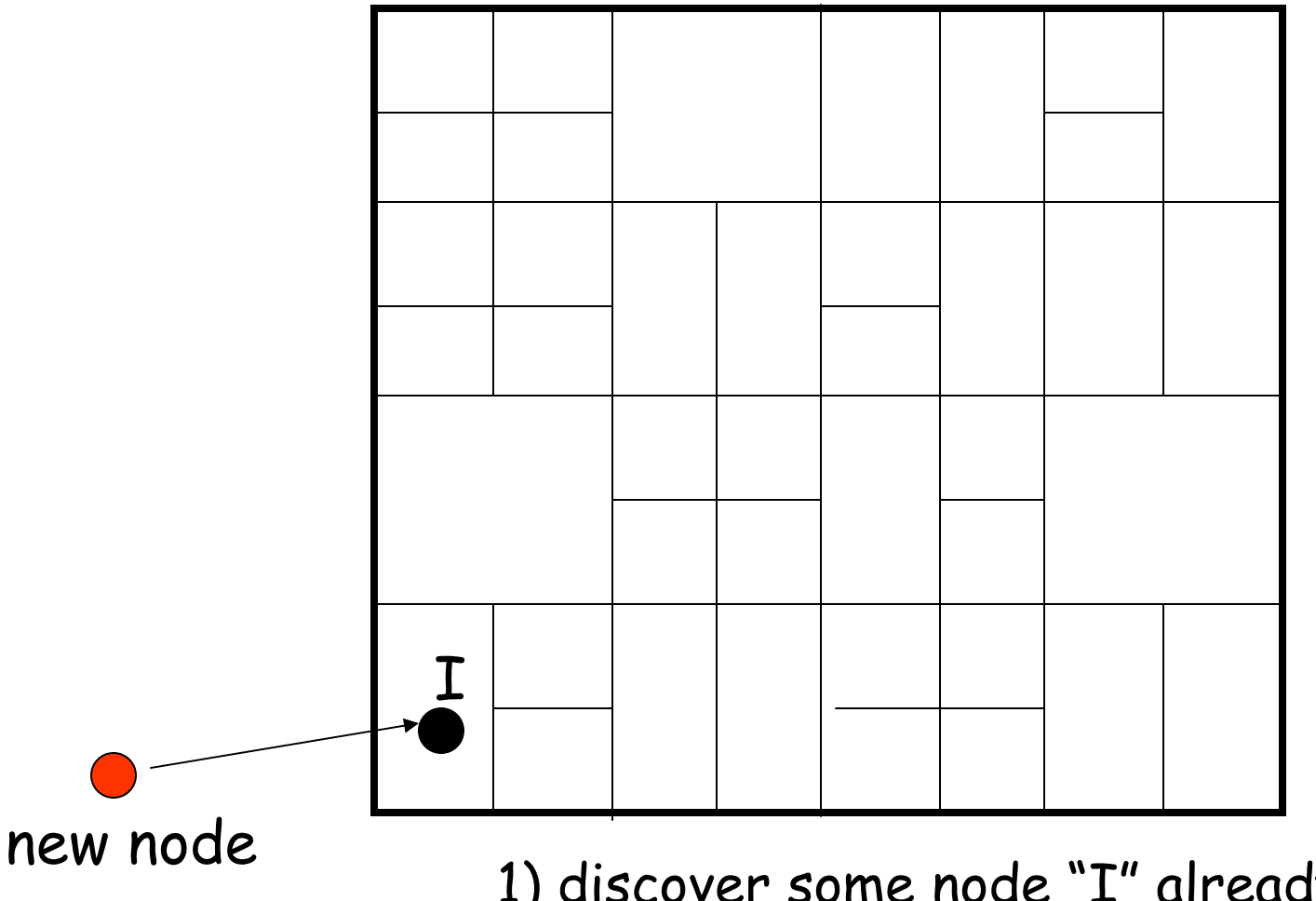
A node only maintains state for its immediate neighboring nodes

CAN: node insertion



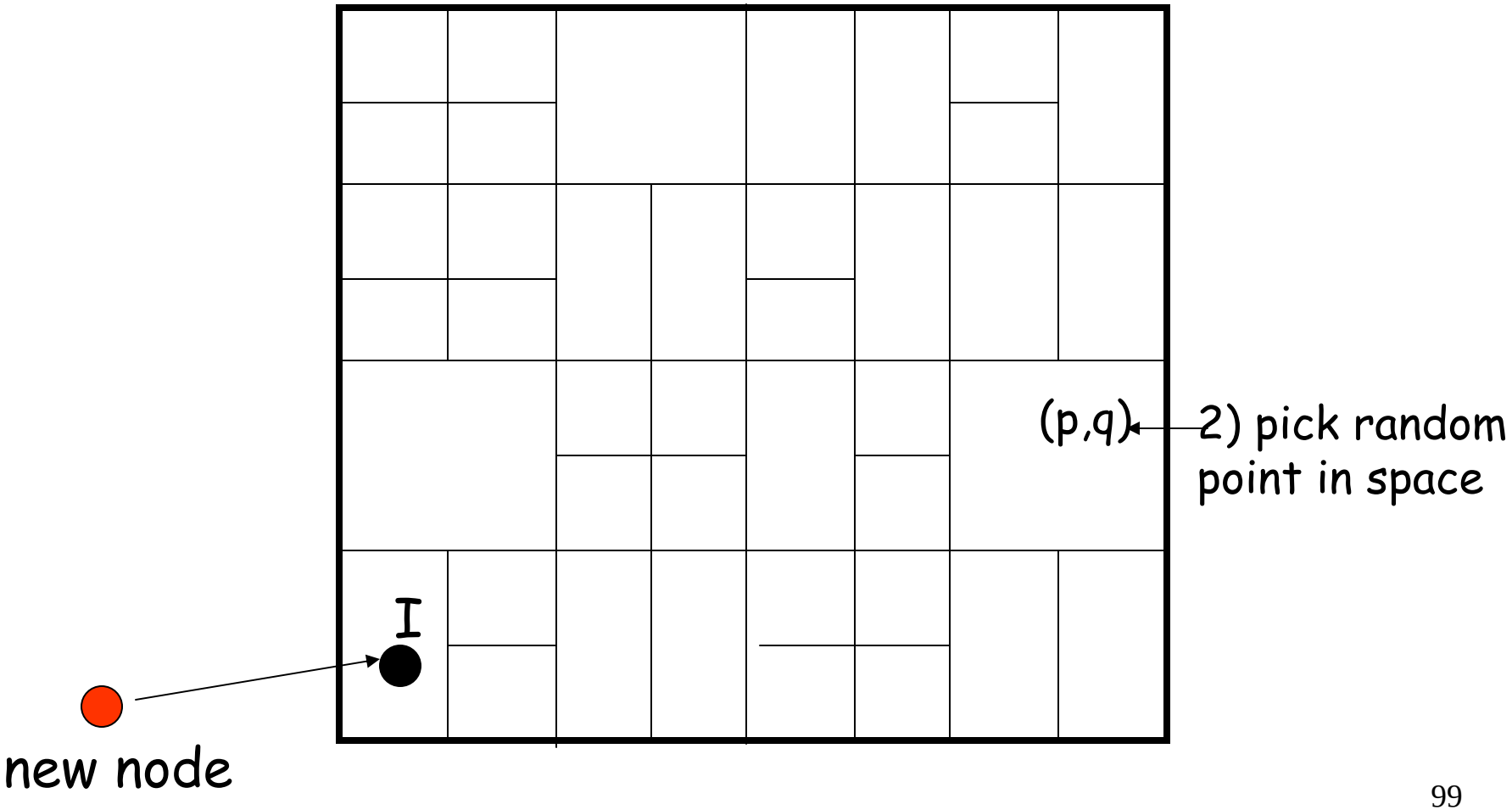
1) Discover some node "I" already in CAN

CAN: node insertion

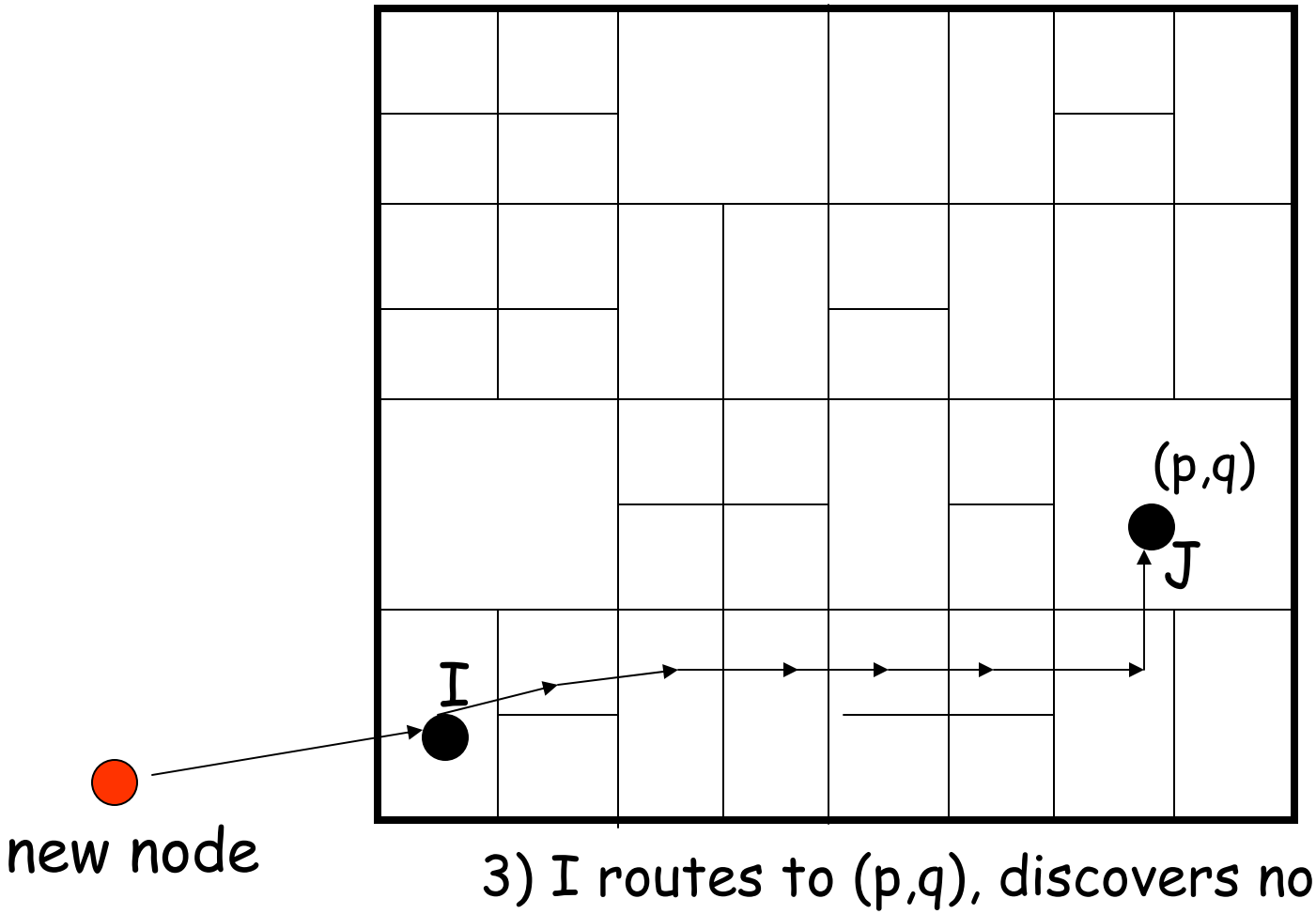


1) discover some node "I" already in CAN

CAN: node insertion

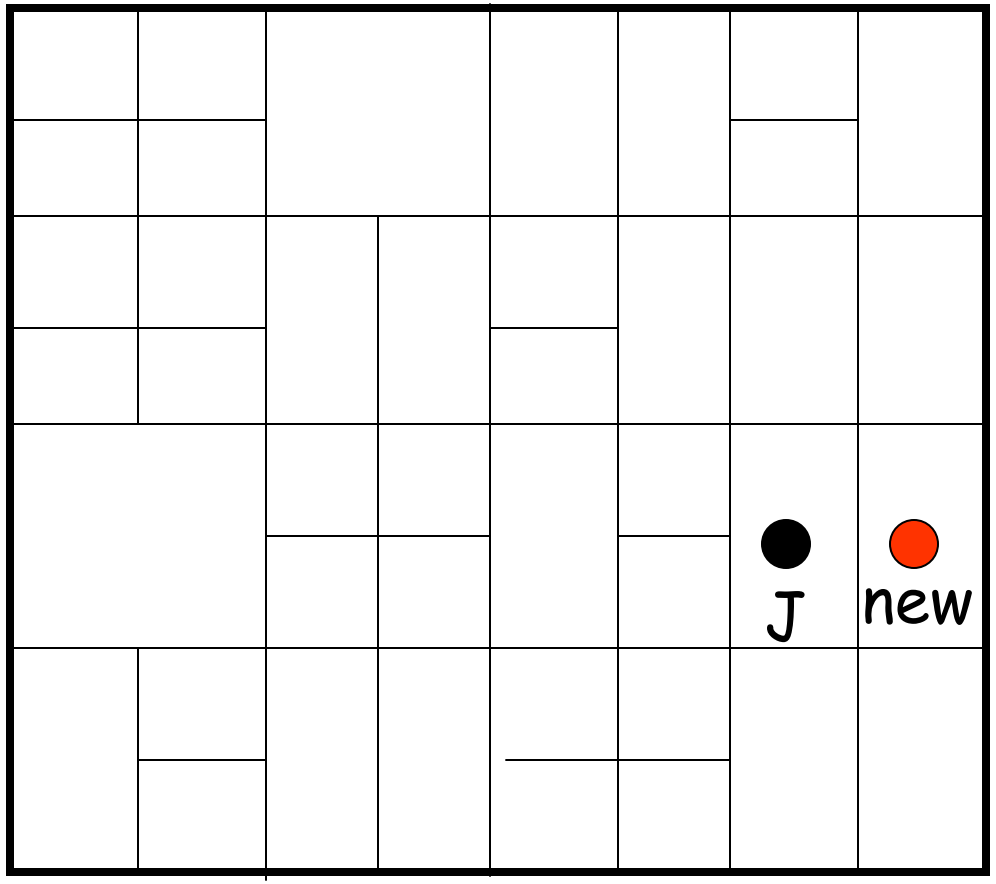


CAN: node insertion



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 (weak point)
    - 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
- Evaluation
- Strengths & Weaknesses
- Ongoing Work

Evaluation

- Scalability
- Low-latency
- Load balancing
- Robustness

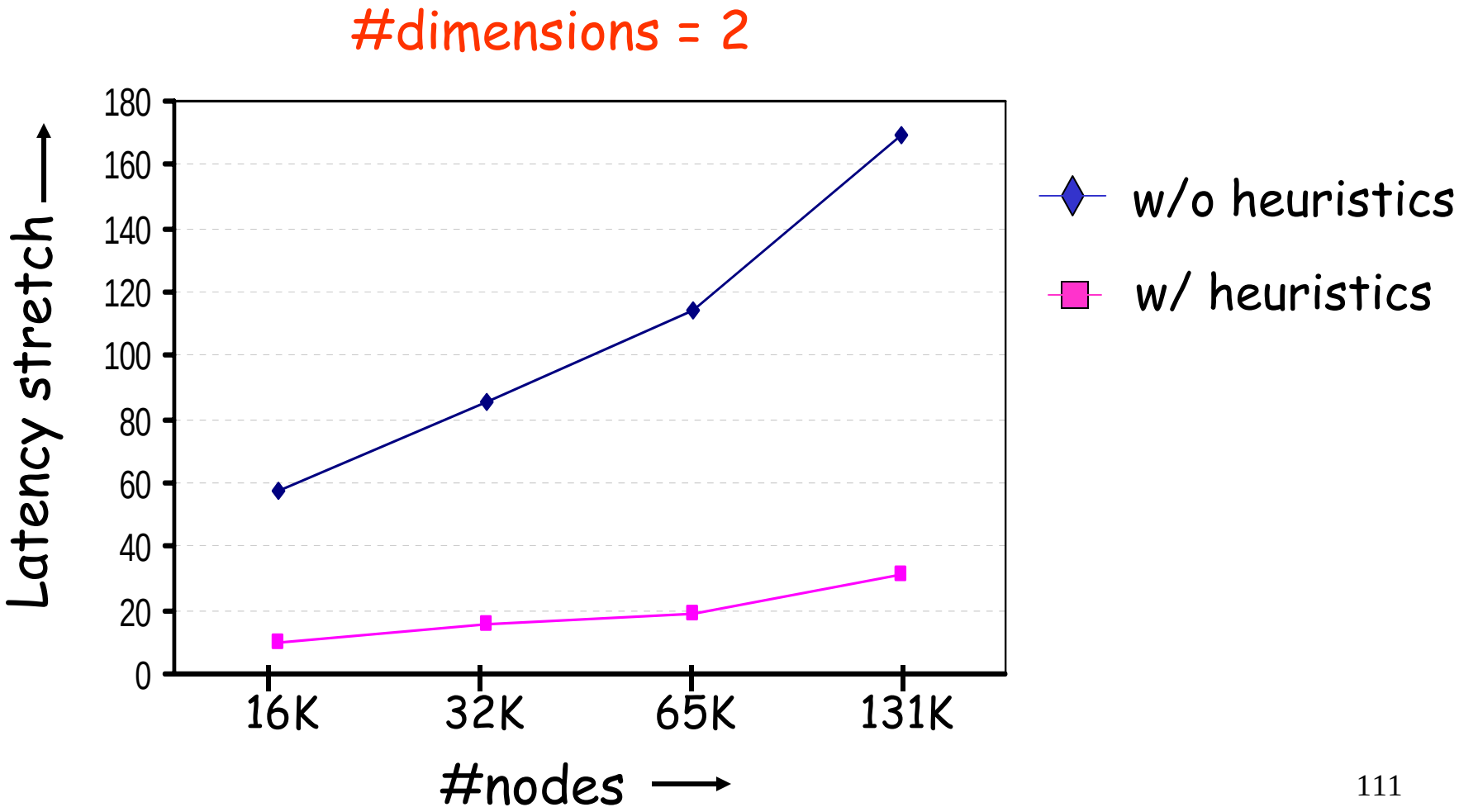
CAN: scalability

- 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

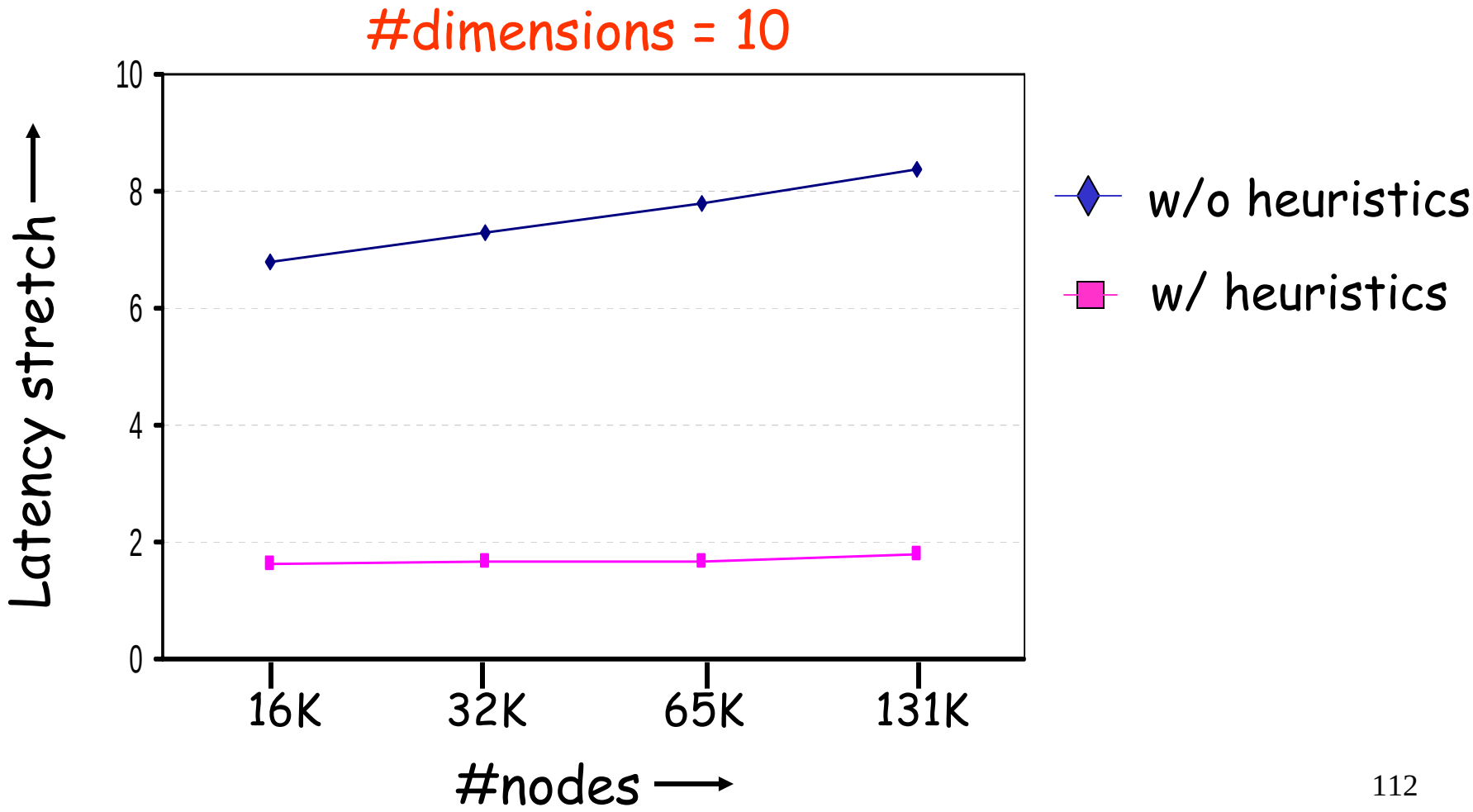
CAN: low-latency

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

CAN: low-latency



CAN: low-latency



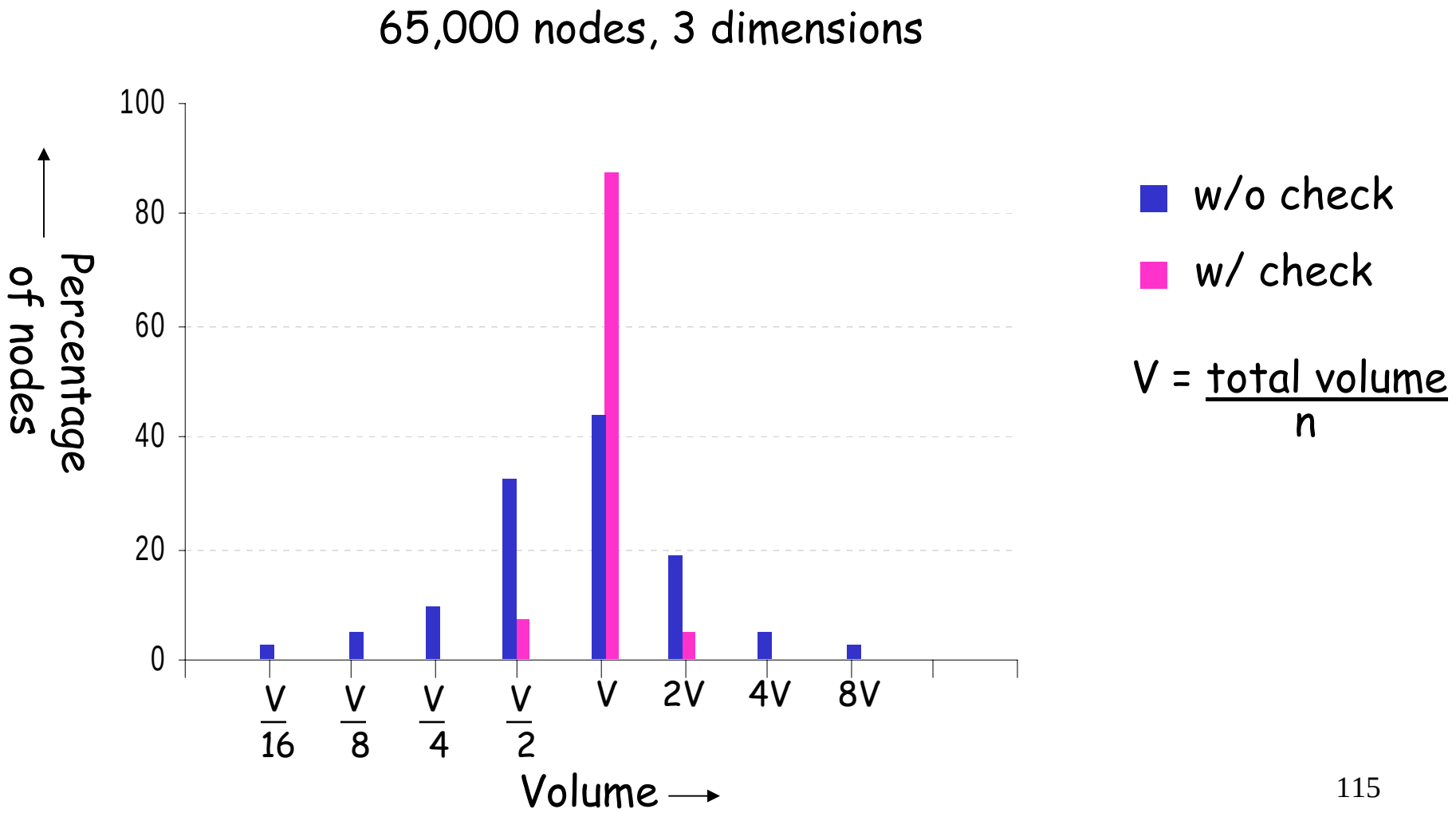
CAN: load balancing

- 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



CAN: Robustness

- Completely distributed
  - no single point of failure (not applicable to pieces of database when node failure happens)
- Not exploring database recovery (in case there are multiple copies of database)
- Resilience of routing
  - can route around trouble

Outline

- Introduction
- Design
- Evaluation
- Strengths & Weaknesses
- Ongoing Work

Strengths

- More resilient than flooding broadcast networks
- Efficient at locating information
- Fault tolerant routing
- Node & Data High Availability (w/ improvement)
- Manageable routing table size & network traffic

Weaknesses

- Impossible to perform a fuzzy search
- Susceptible to malicious activity
- Maintain coherence of all the indexed data (Network overhead, Efficient distribution)
- Still relatively higher routing latency
- Poor performance w/o improvement

Suggestions

- Catalog and Meta indexes to perform search function
- Extension to handle mutable content efficiently for web-hosting
- Security mechanism to defense against attacks

Outline

- Introduction
- Design
- Evaluation
- Strengths & Weaknesses
- Ongoing Work

Ongoing Work

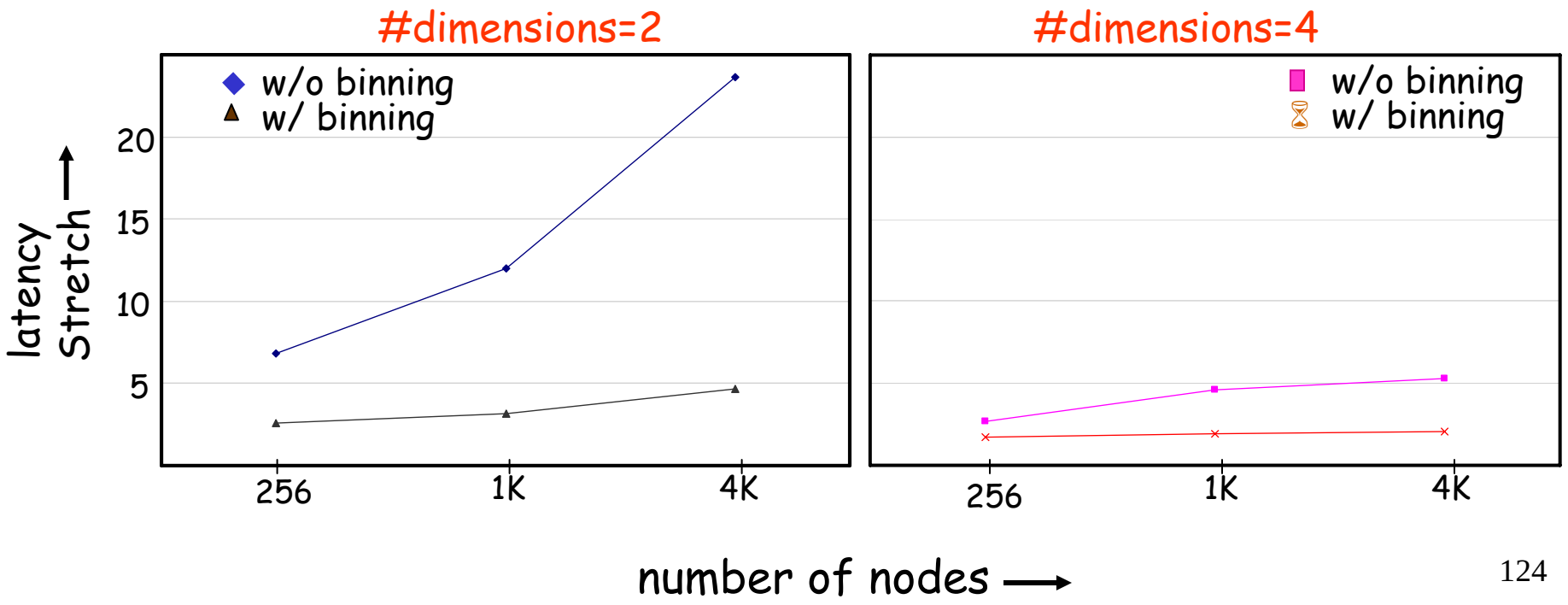
- Topologically-sensitive CAN construction
  - distributed binning

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

Wrapup discussion questions (1):

- What is a peer-peer network (what is not a peer-to-peer network?). Necessary:
  - *every* node is designed to (but may not by user choice) *provide some service* that helps other nodes in the network get service
  - no 1-N service providing
  - each node potentially has the same responsibility, functionality (maybe nodes can be polymorhpic)
    - corollary: by design, nothing (functionally) prevents two nodes from communicating directly
  - some applications (e.g., Napster) are a mix of peer-peer and centralized (lookup is centralized, file service is peer-peer) [recursive def. of peer-peer]
  - (logical connectivity rather than physical connectivity) routing will depend on service and data

Overlays?

- What is the relationship between peer-peer and application overlay networks?
  - Peer-peer and application overlays are different things. It is possible for an application level overlay to be built using peer-peer (or vice versa) but not always necessary
  - Overlay: in a wired net: if two nodes can communicate in the overlay using a path that is *not* the path the network level routing would define for them. Logical network on top of underlying network
    - source routing?
  - Wireless ad hoc nets – what commonality is there REALLY?

Wrapup discussion questions (2):

- What were the best p2p idea
- Vote now (and should it be a secret ballot usign Eternity☺

Wrapup discussion questions (3):

- Is ad hoc networking a peer-peer application?
  - Yes (30-1)
- Why peer-peer over client-server?
  - A well-deigned p2p provides better "scaability"
- Why client-server of peer-peer
  - peer-peer is harder to make reliable
  - availability different from client-server (p2p is more often at least partially "up")
  - more trust is required
- If all music were free in the future (and organized), would we have peer-peer.
  - Is there another app: ad hoc networking, any copyrighted data, peer-peer sensor data gathering and retrieval, simulation
- Evolution #101 – what can we learn about systems?