Network Applications: LBN using Smart Switch; Application Overlay Networks

Y. Richard Yang

http://zoo.cs.yale.edu/classes/cs433/

3/2/2016

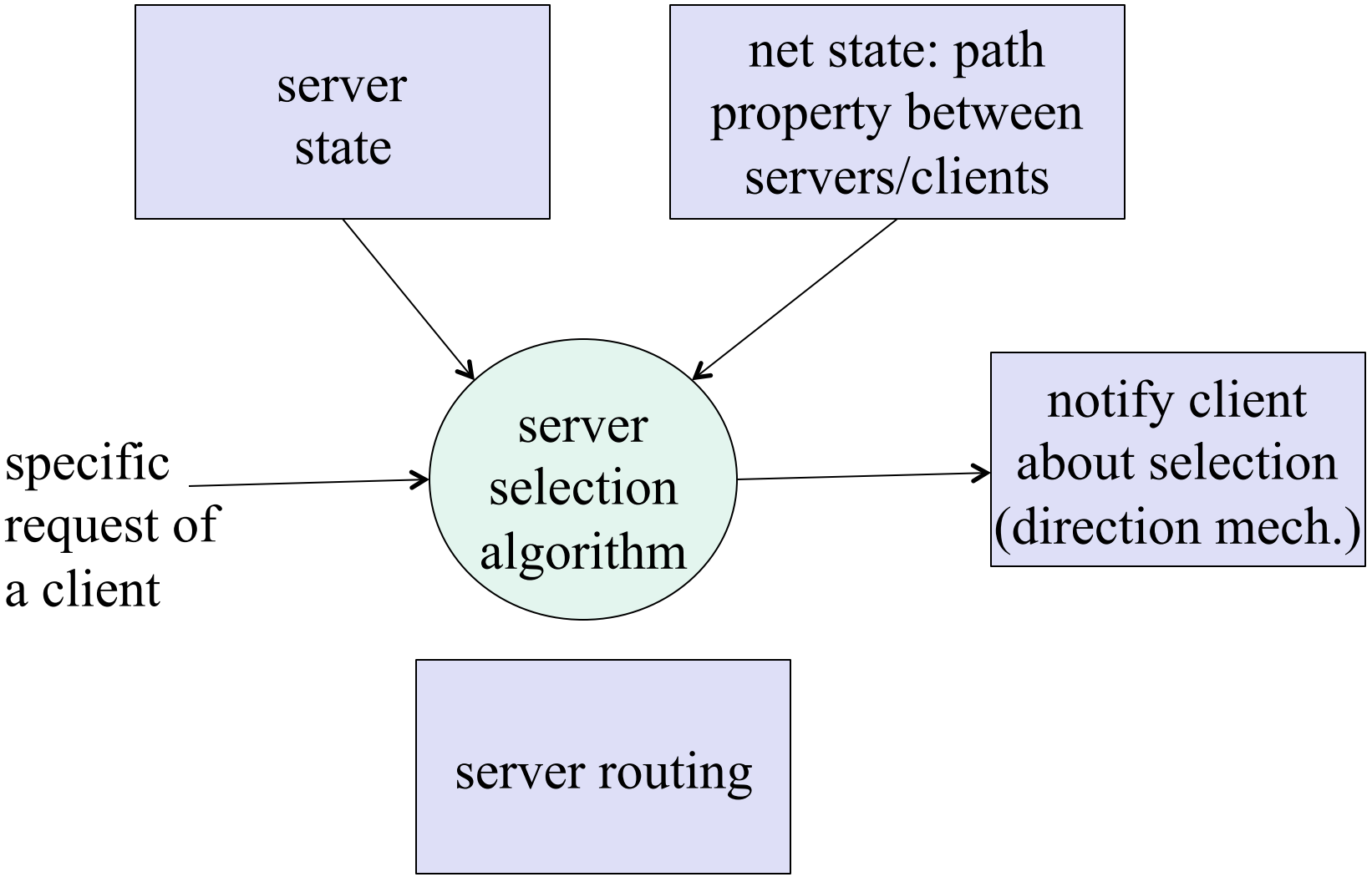
Outline

- Admin and recap
- Multiple servers
- Application overlays
  - Overlays for scalability
  - Overlays for anonymity
  - Overlays for distributed content hosting

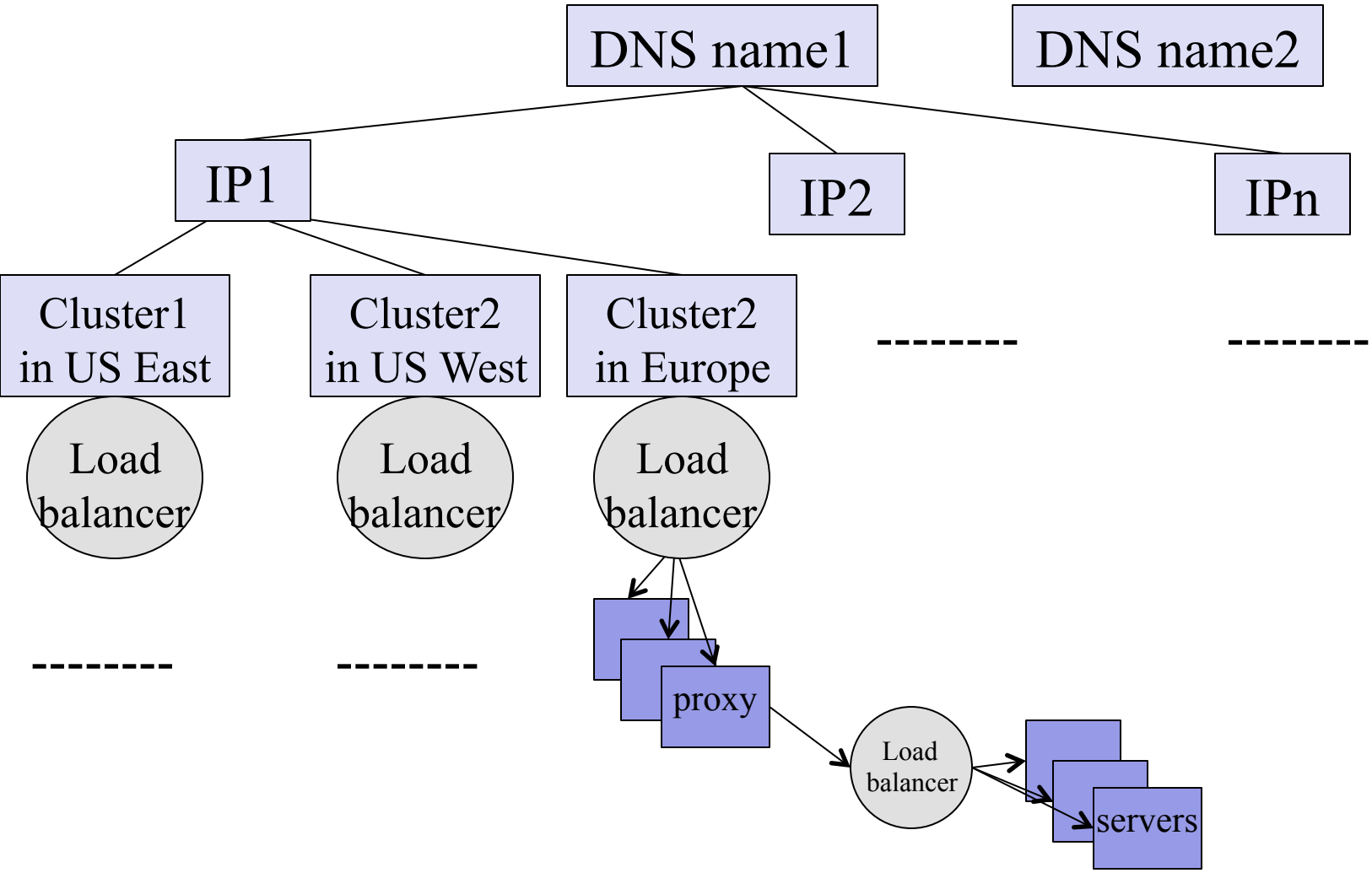
Admin

- Assignment three status and questions.

Recap: LBN Arch



Recap: LBN Mechanisms



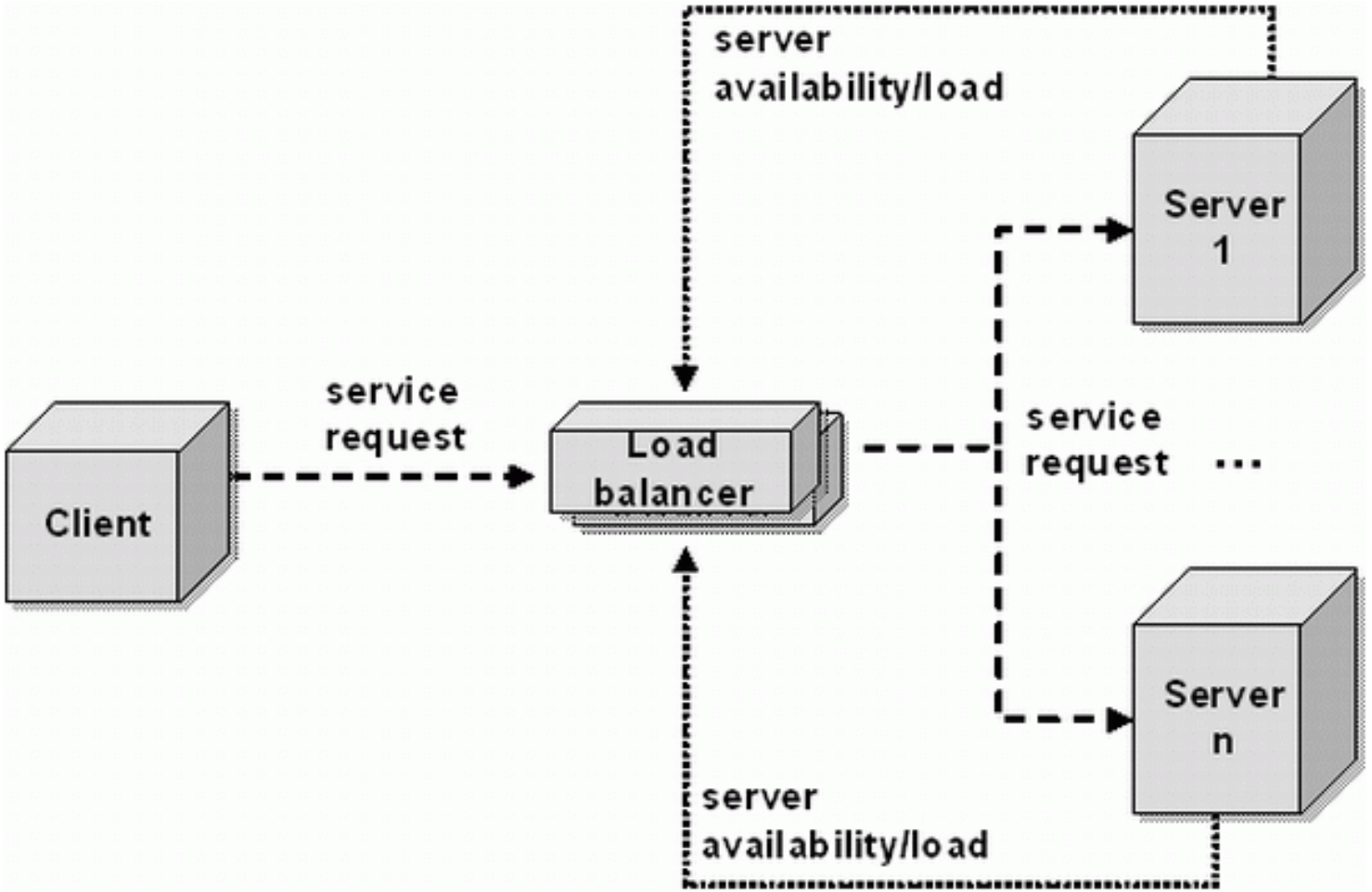
Problem

- Clients get a single service IP address, but we want to use a cluster of (physical) servers
  - The single IP is called a virtual IP address (VIP)

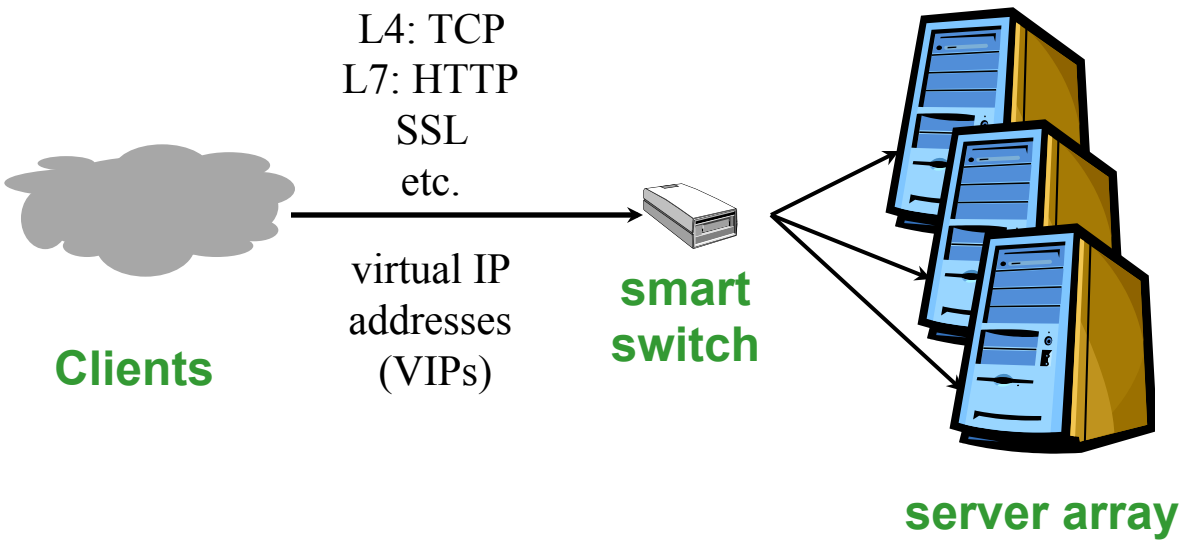
Outline

- Recap
- Multiple network servers
  - Basic issues
  - Load direction
    - DNS
    - Load balancer (smart switch)

Smart Switch: Big Picture



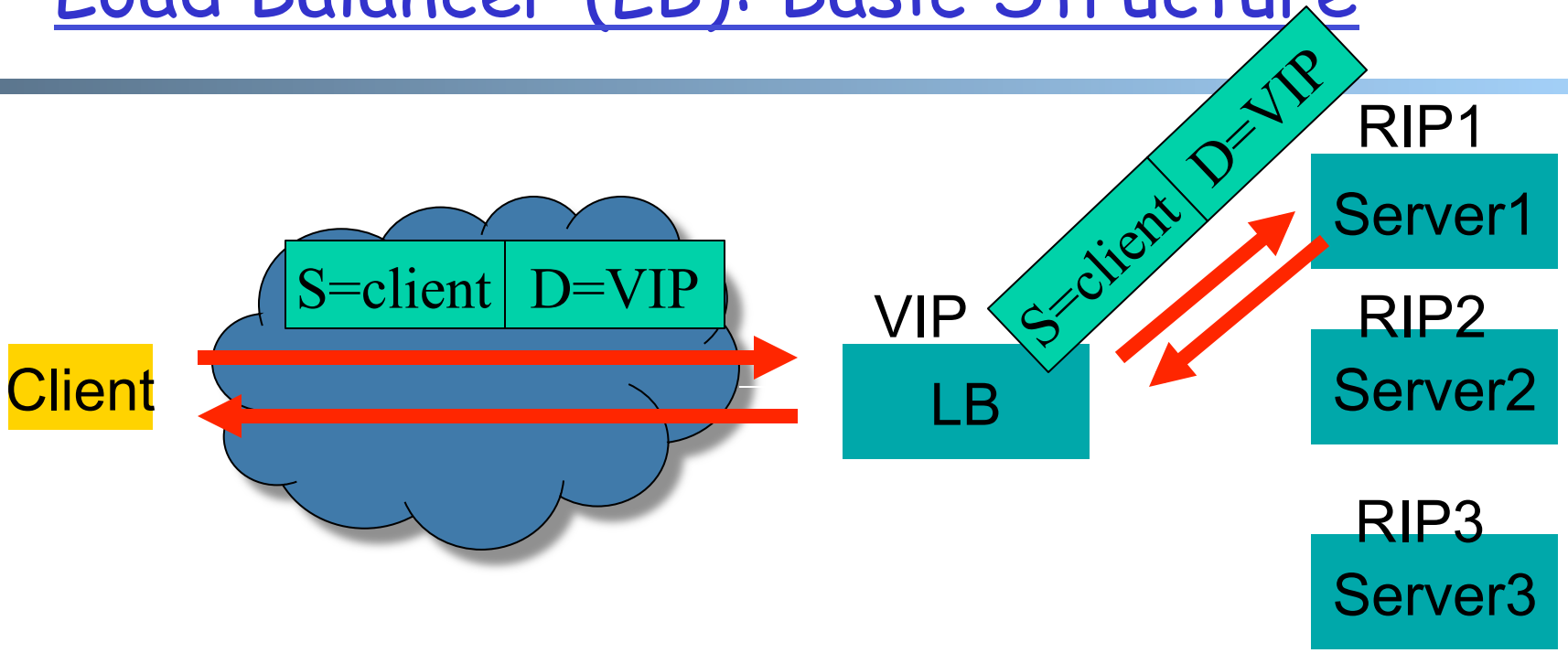
VIP Clustering



Goals
server load balancing
failure detection
access control filtering
priorities/QoS
request locality
transparent caching

What to switch/filter on?
L3 source IP and/or VIP
L4 (TCP) ports etc.
L7 URLs and/or cookies
L7 SSL session IDs

Load Balancer (LB): Basic Structure

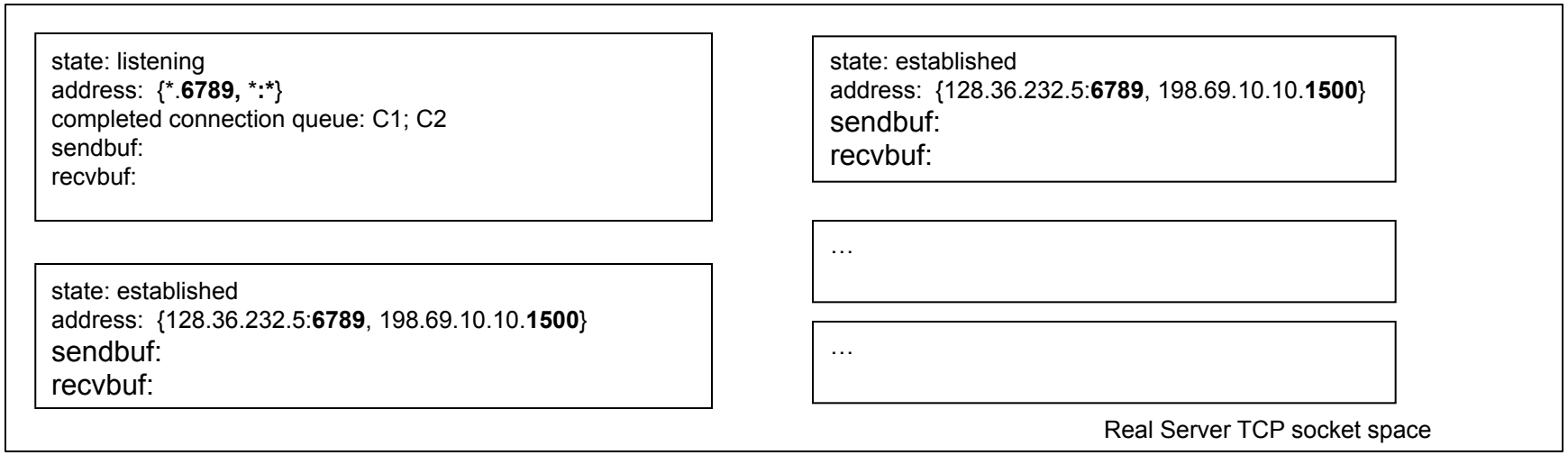


Problem of the basic structure?

Problem

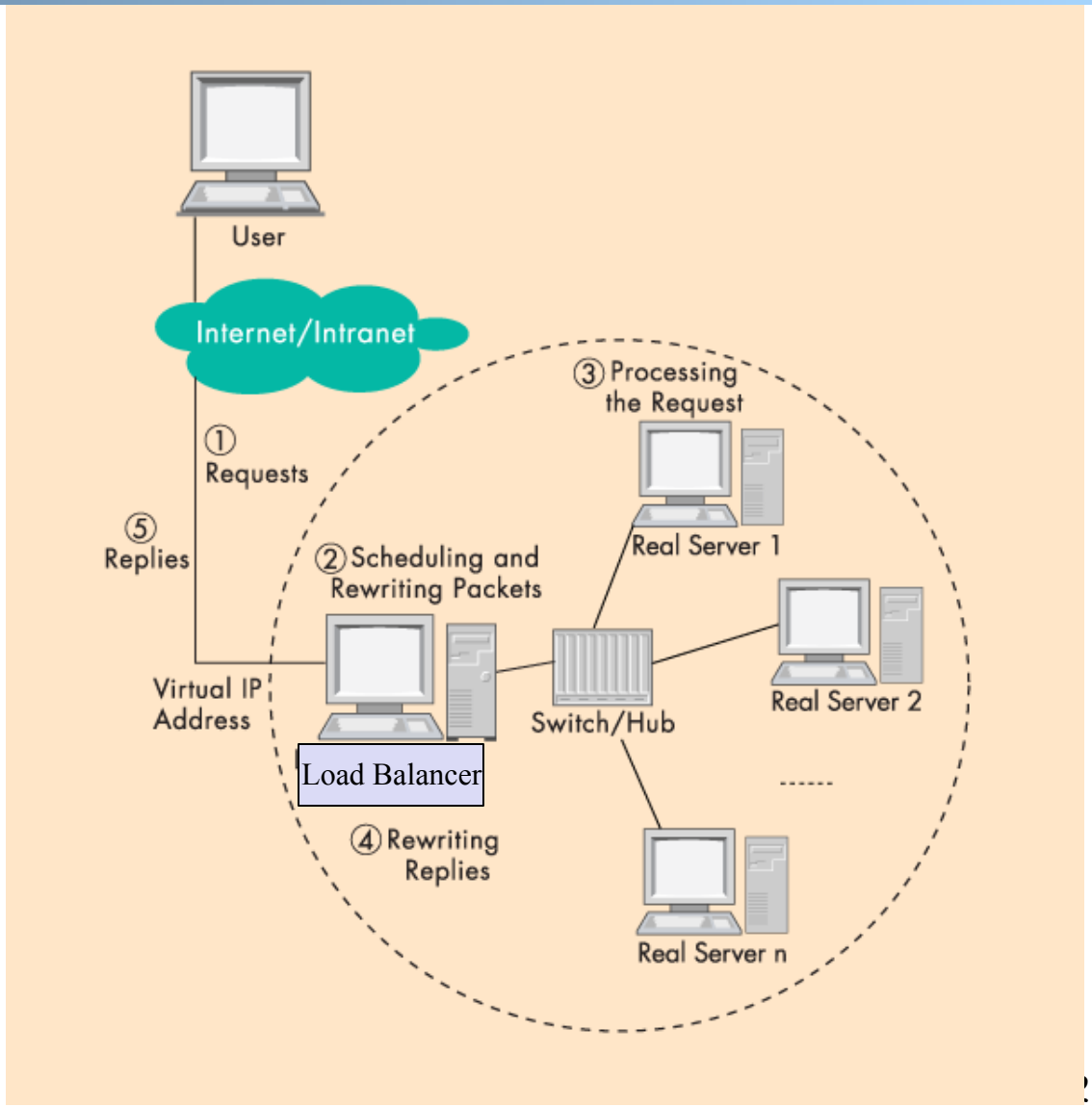
S=client	D=VIP

- Client to server packet has VIP as destination address, but real servers use RIPs
  - if LB just forwards the packet from client to a real server, the real server drops the packet
  - reply from real server to client has real server IP as source -> client will drop the packet



Solution 1: Network Address Translation (NAT)

- LB does rewriting/ translation
- Thus, the LB is similar to a typical NAT gateway with an additional scheduling function



Example Virtual Server via NAT

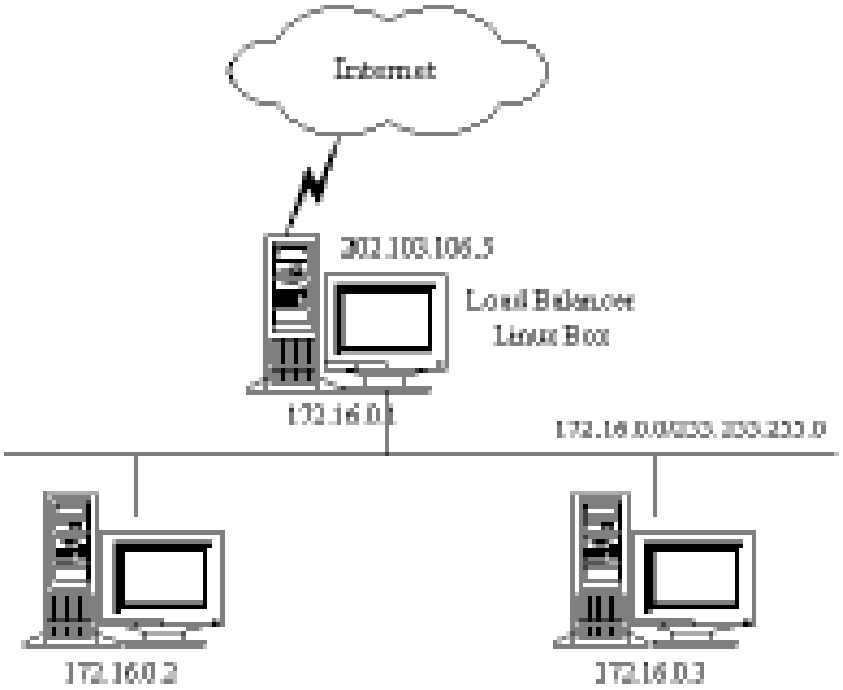
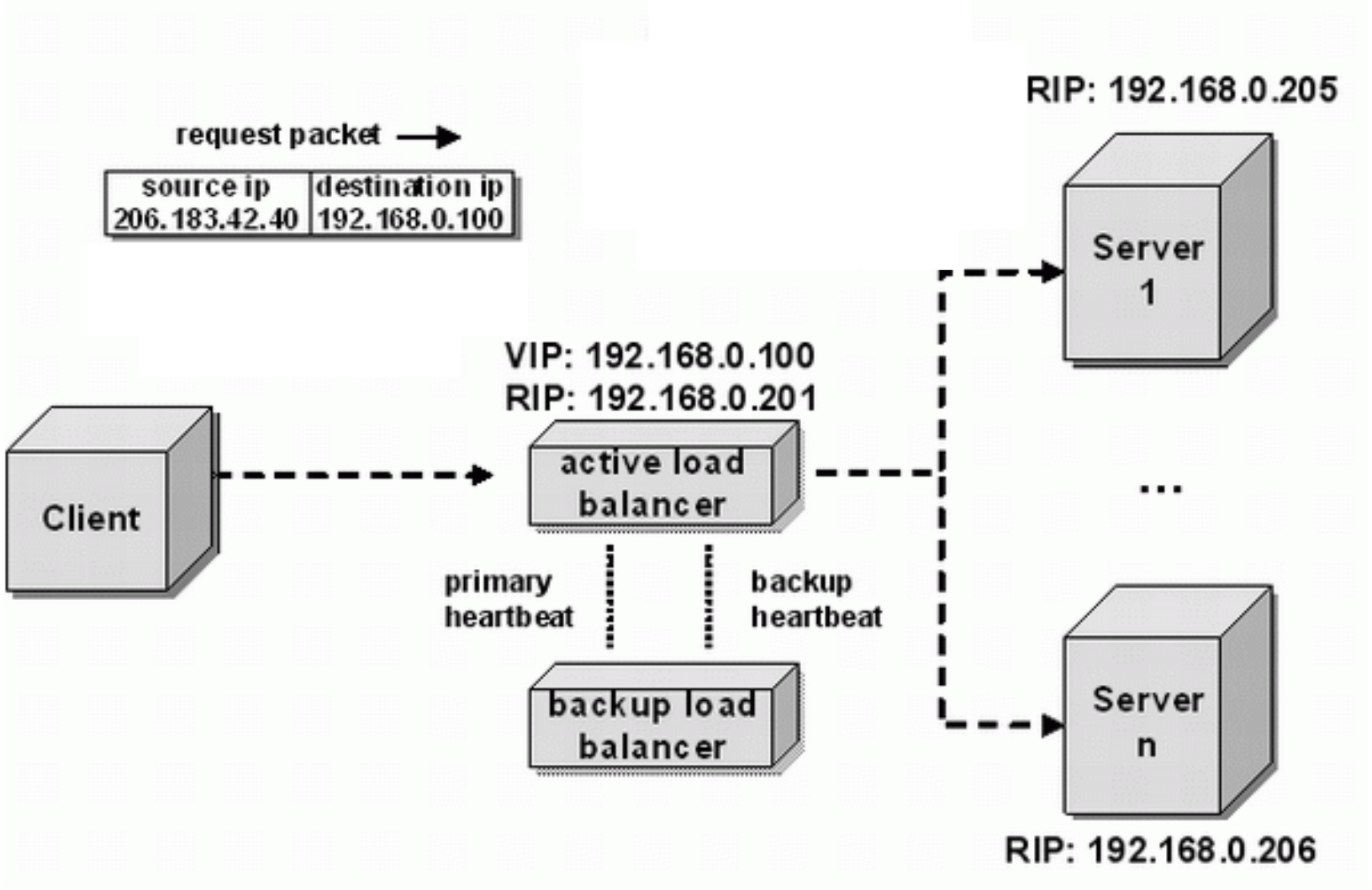
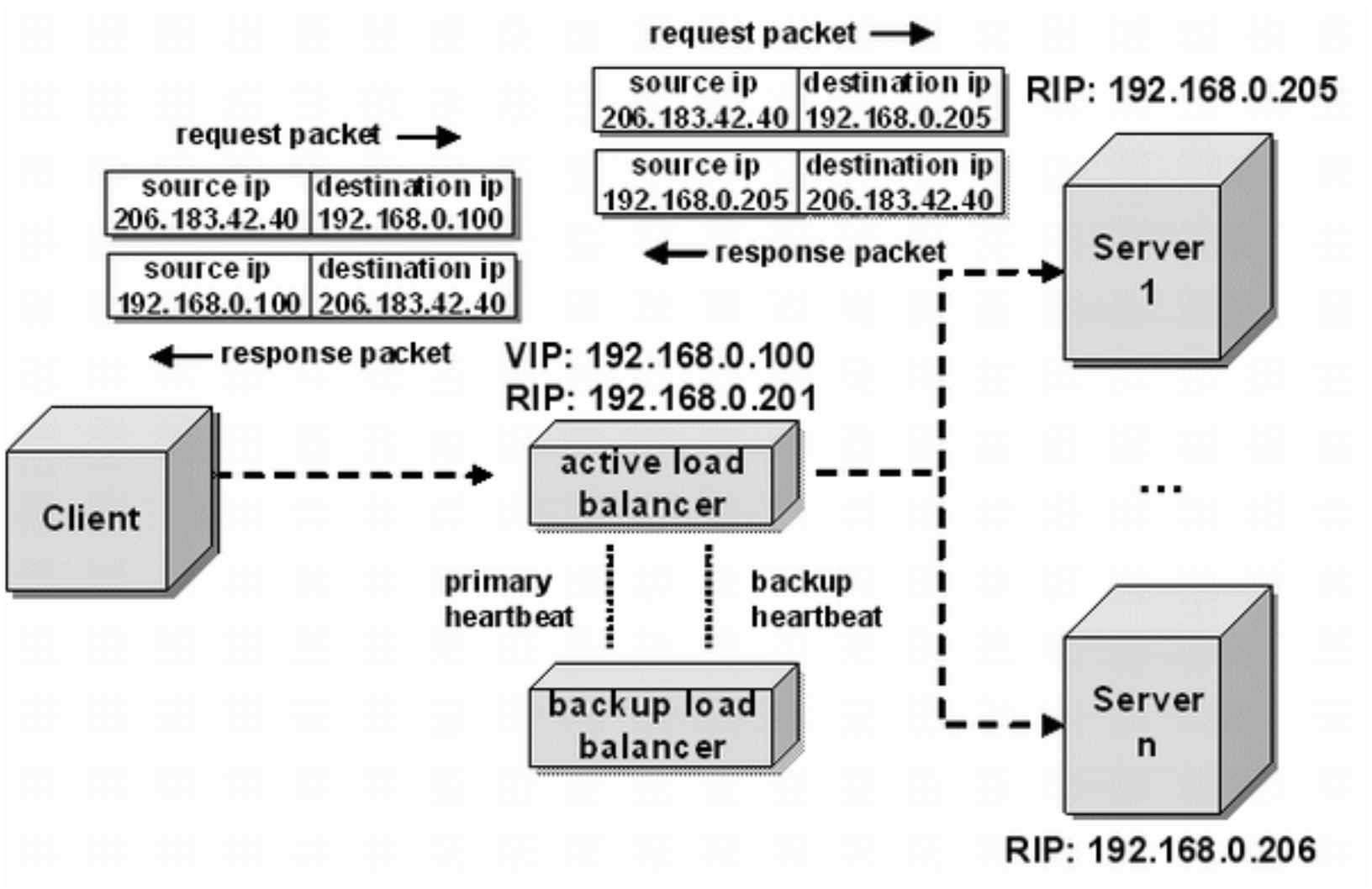


Table 1: an example of virtual server rules					
Protocol	Virtual IP Address	Port	Real IP Address	Port	Weight
TCP	202.103.106.5	80	172.16.0.2	80	1
			172.16.0.3	8000	2
TCP	202.103.106.5	21	172.16.0.3	21	1

LB/NAT Flow


RIP: 192.168.0.205
request packet
source ip
206.183.42.40
destination ip
192.168.0.100
Server
1
VIP: 192.168.0.100
RIP: 192.168.0.201
Client
active load
balancer
...
primary
heartbeat
backup
heartbeat
backup load
balancer
Server
n
RIP: 192.168.0.206

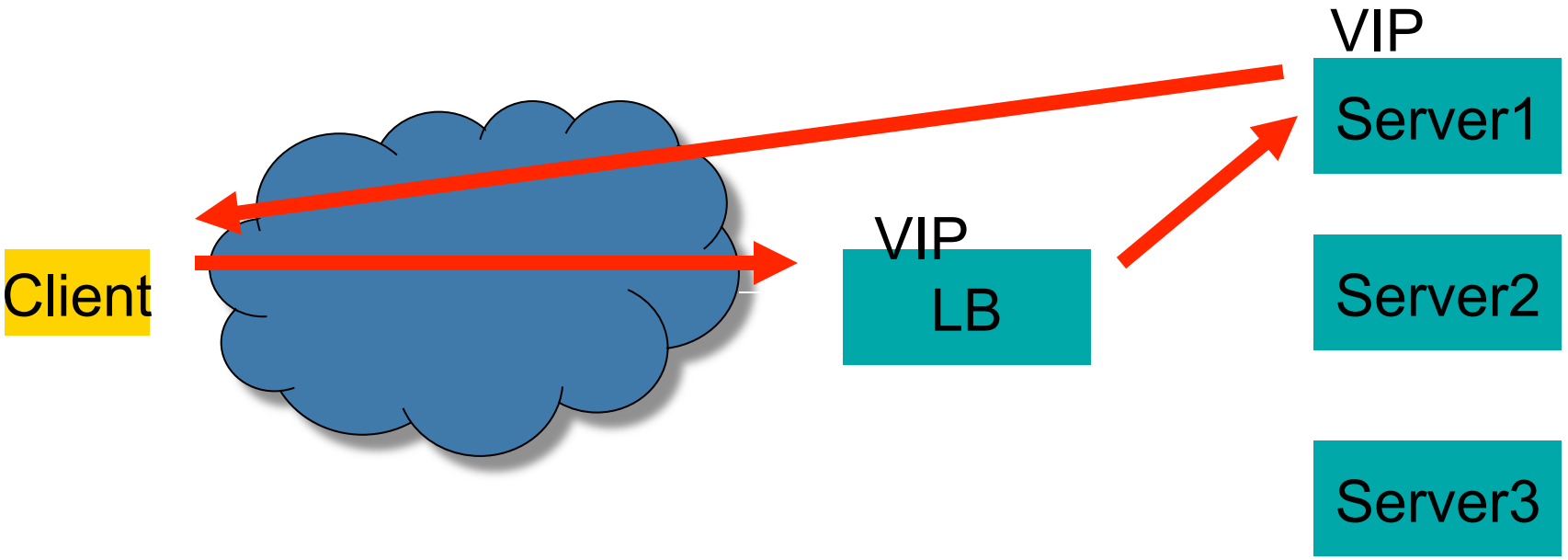
LB/NAT Flow


request packet
source ip 206.183.42.40
destination ip 192.168.0.205
RIP: 192.168.0.205
request packet
source ip 206.183.42.40
destination ip 192.168.0.100
source ip 192.168.0.205
destination ip 206.183.42.40
response packet
Server 1
source ip 192.168.0.100
destination ip 206.183.42.40
response packet
VIP: 192.168.0.100
RIP: 192.168.0.201
Client
active load balancer
...
primary heartbeat
backup heartbeat
backup load balancer
Server n
RIP: 192.168.0.206

LB/NAT Advantages and Disadvantages

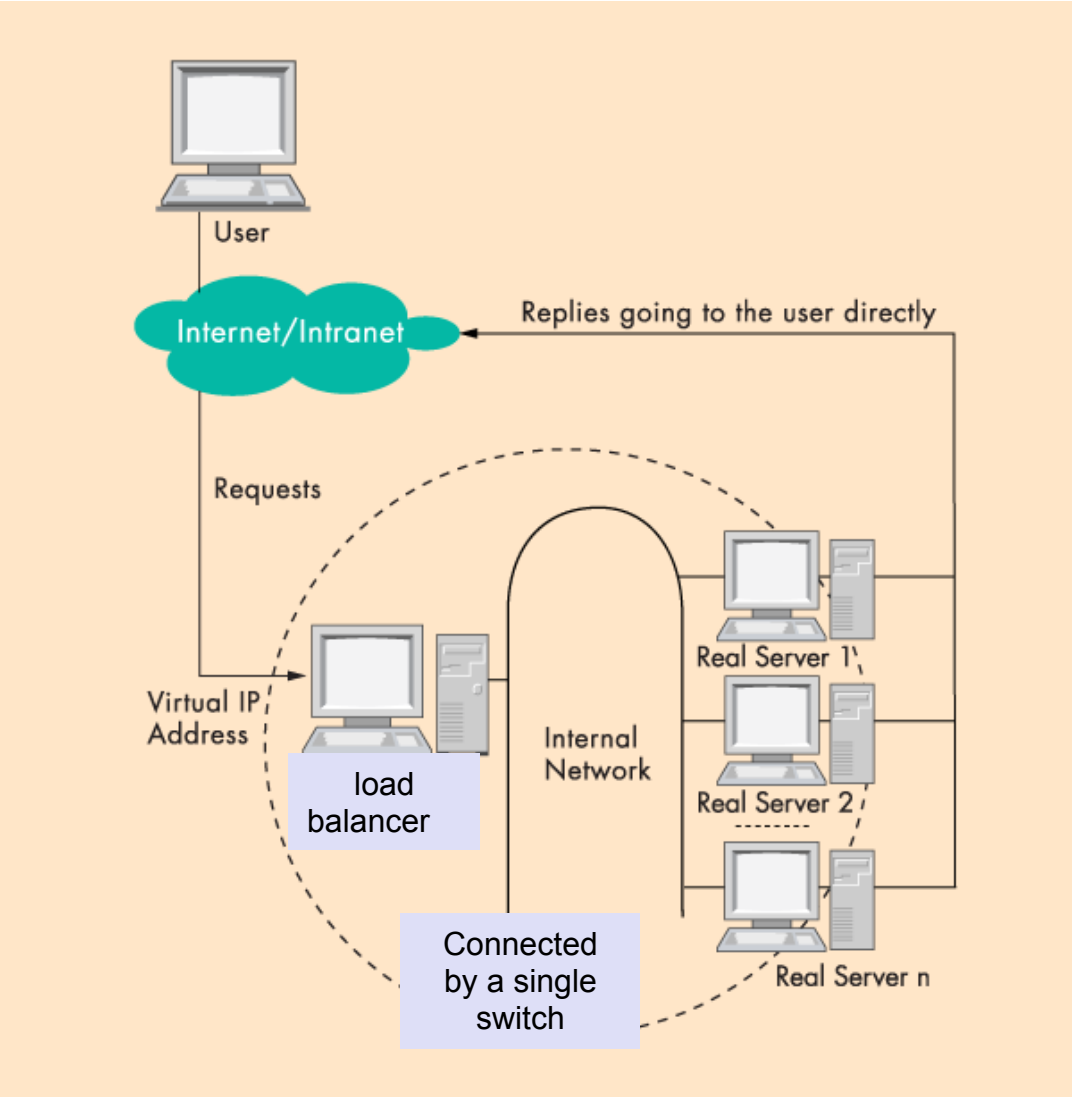
- Advantages:
  - Only one public IP address is needed for the load balancer; real servers can use private IP addresses
  - Real servers need no change and are not aware of load balancing
- Problem
  - The load balancer must be on the critical path and hence may become the bottleneck due to load to rewrite request and response packets
    - Typically, rewriting responses has more load because there are more response packets

LB with Direct Reply

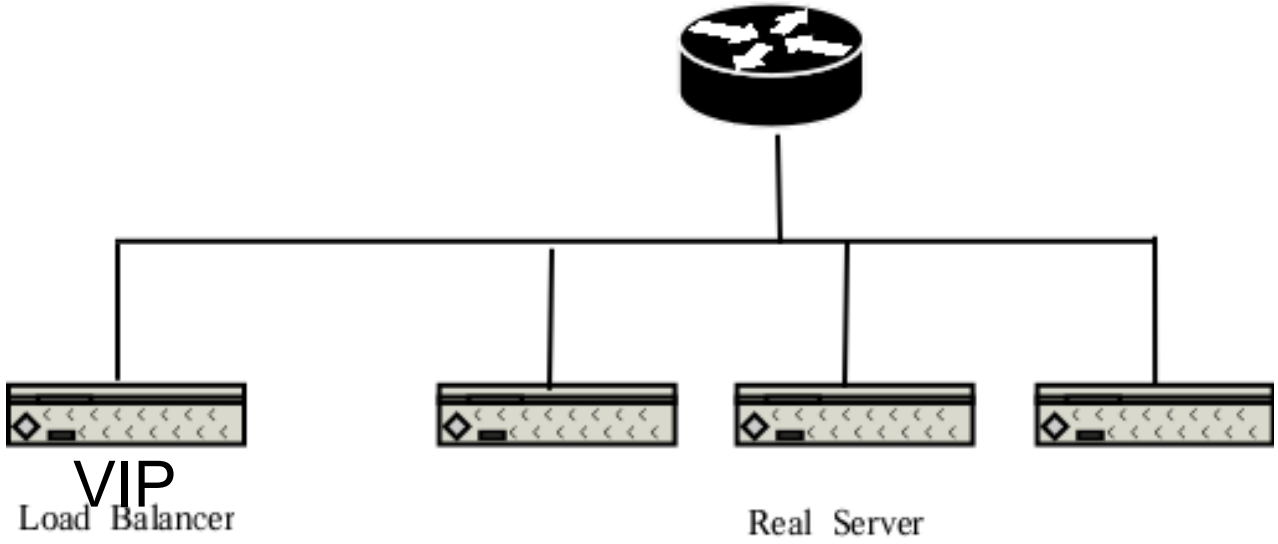


Direct reply → Each real server uses VIP as its IP address

LB/DR Architecture



Why IP Address Matters?



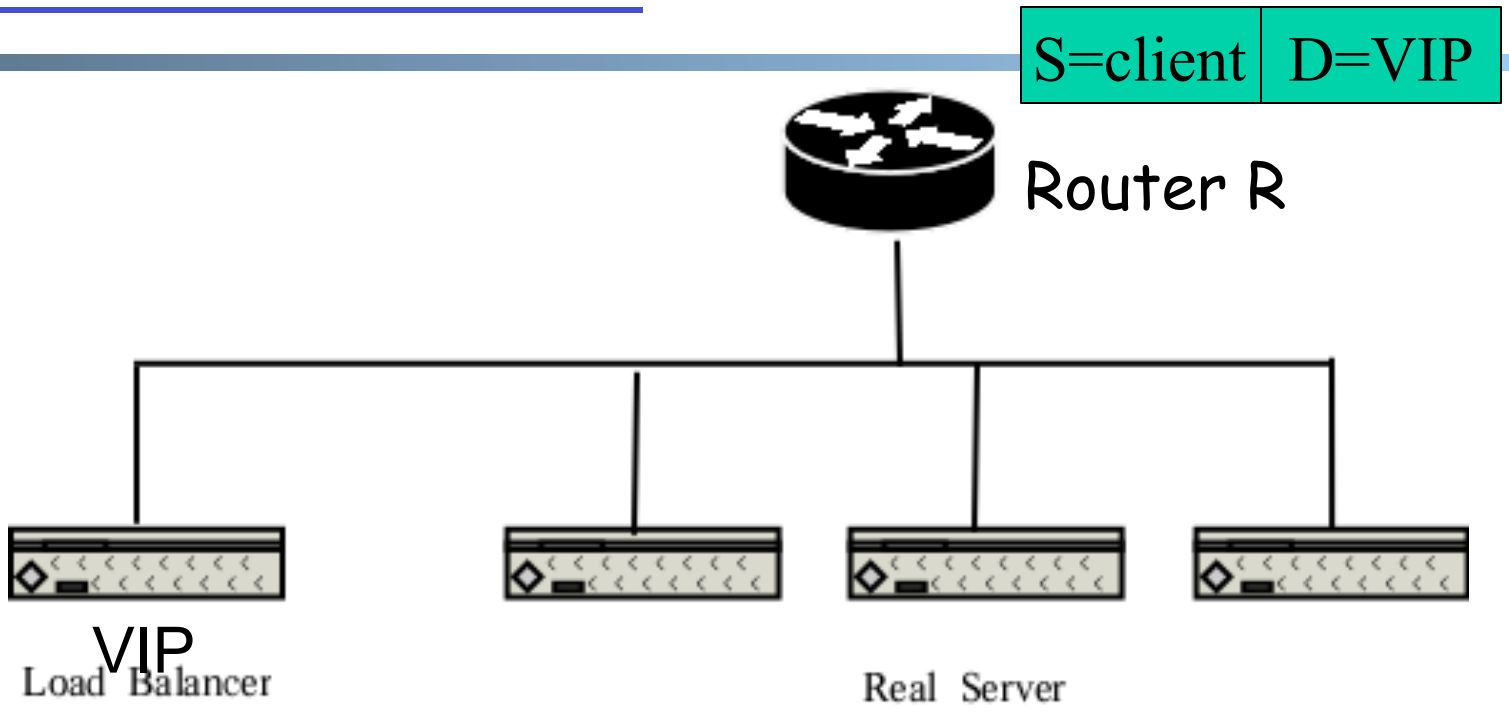
- Each network interface card listens to an assigned MAC address
- A router is configured with the range of IP addresses connected to each interface (NIC)
- To send to a device with a given IP, the router needs to translate IP to MAC (device) address
- The translation is done by the Address Resolution Protocol (ARP)

ARP Protocol

- ARP is "plug-and-play":
  - nodes create their ARP tables without intervention from net administrator

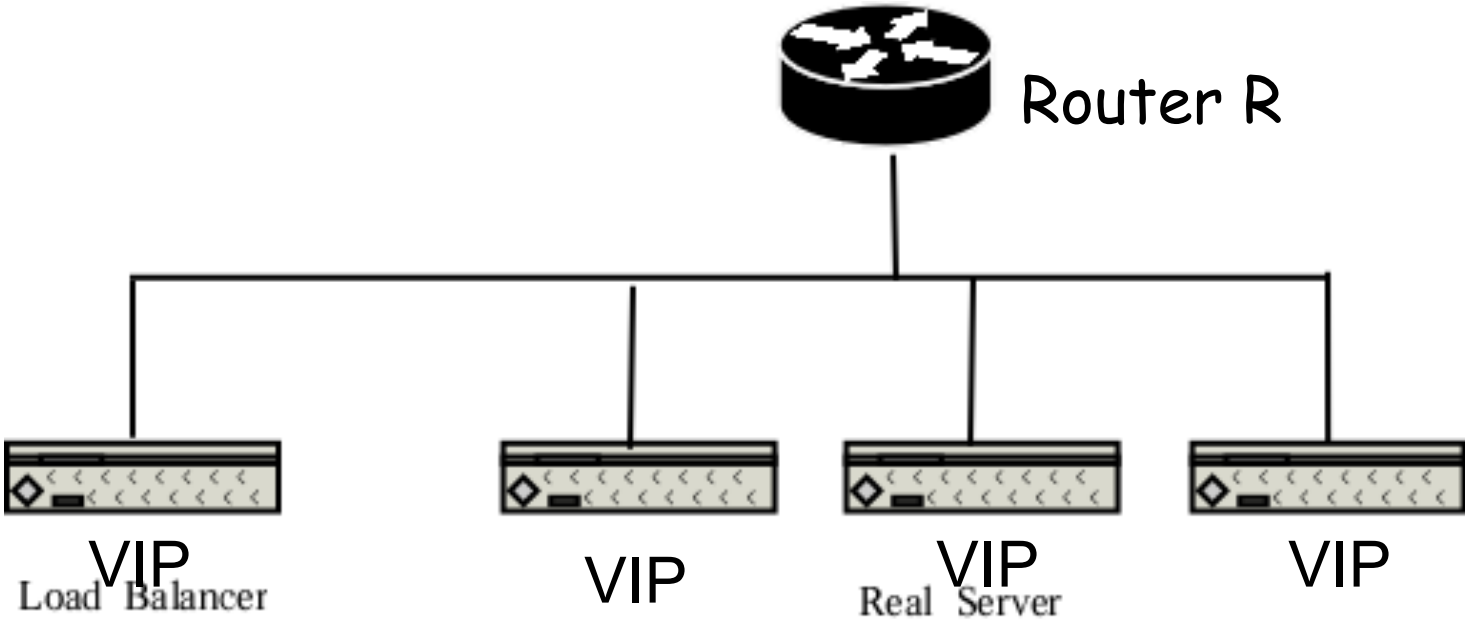
- A broadcast protocol:
  - Router broadcasts query frame, containing queried IP address
    - all machines on LAN receive ARP query
  - Node with queried IP receives ARP frame, replies its MAC address

ARP in Action



- Router broadcasts ARP broadcast query: who has VIP?
- ARP reply from LB: I have VIP; my MAC is MAC_{LB}
- Data packet from R to LB: destination MAC = MAC_{LB}

LB/DR Problem



ARP and race condition:

- When router R gets a packet with dest. address VIP, it broadcasts an Address Resolution Protocol (ARP) request: who has VIP?
- One of the real servers may reply before load balancer

Solution: configure real servers to not respond to ARP request

LB via Direct Routing

- The virtual IP address is shared by real servers and the load balancer.
- Each real server has a non-ARPing, loopback alias interface configured with the virtual IP address, and the load balancer has an interface configured with the virtual IP address to accept incoming packets.
- The workflow of LB/DR is similar to that of LB/NAT:
  - the load balancer directly routes a packet to the selected server
    - the load balancer simply changes the MAC address of the data frame to that of the server and retransmits it on the LAN (how to know the real server's MAC?)
  - When the server receives the forwarded packet, the server determines that the packet is for the address on its loopback alias interface, processes the request, and finally returns the result directly to the user

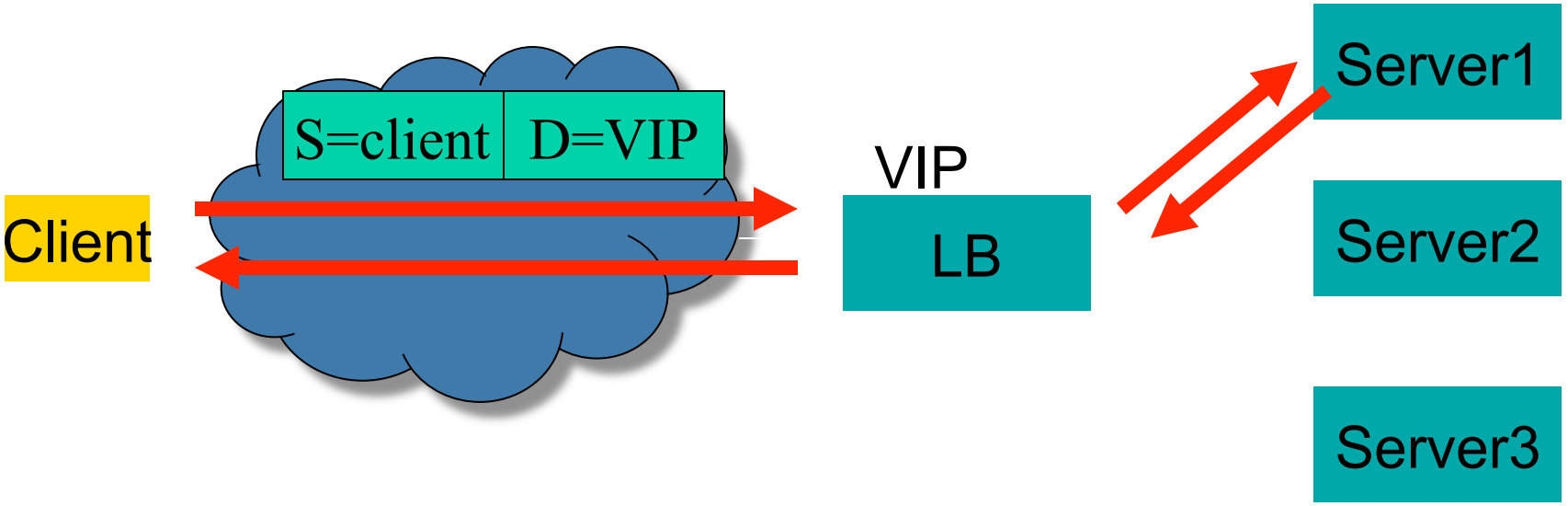
LB/DR Advantages and Disadvantages

- Advantages:
  - Real servers send response packets to clients directly, avoiding LB as bottleneck
- Disadvantages:
  - Servers must have non-arp alias interface
  - The load balancer and server must have one of their interfaces in the same LAN segment

Example Implementation of LB

- An example open source implementation is Linux virtual server (linux-vs.org)
  - Used by
    - www.linux.com
    - sourceforge.net
    - wikipedia.org
  - More details on ARP problem: http://www.austintek.com/LVS/LVS-HOWTO/HOWTO/LVS-HOWTO.arp_problem.html
  - Many commercial LB servers from F5, Cisco, …
- More details please read chapter 2 of Load Balancing Servers, Firewalls, and Caches

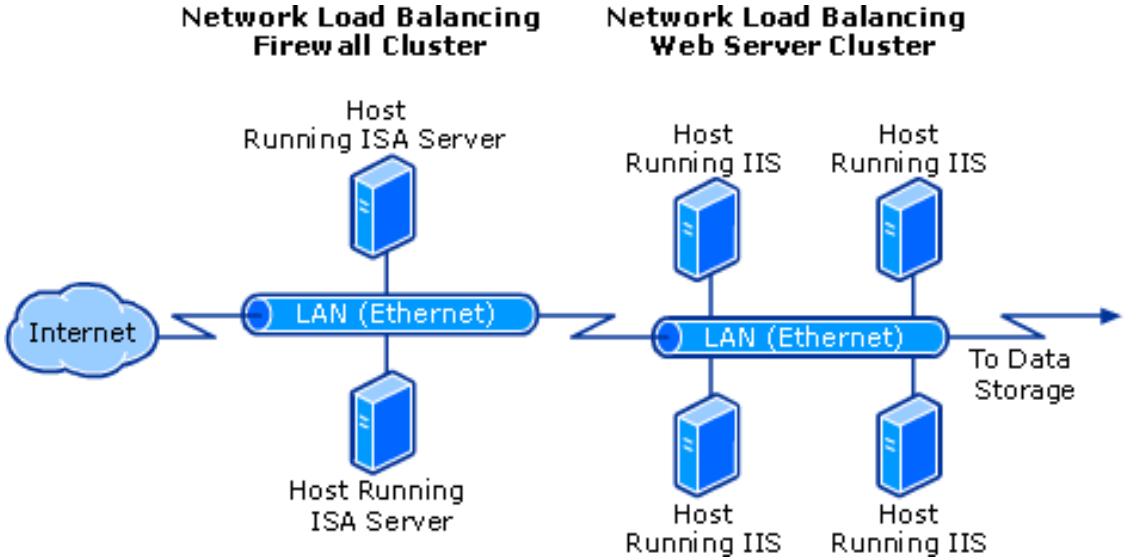
Discussion: Problem of the Load Balancer Architecture



A major remaining problem is that the LB becomes a single point of failure (SPOF).

Solutions

- Redundant load balancers
  - E.g., two load balancers
- Fully distributed load balancing
  - e.g., Microsoft Network Load Balancing (NLB)



Microsoft NLB

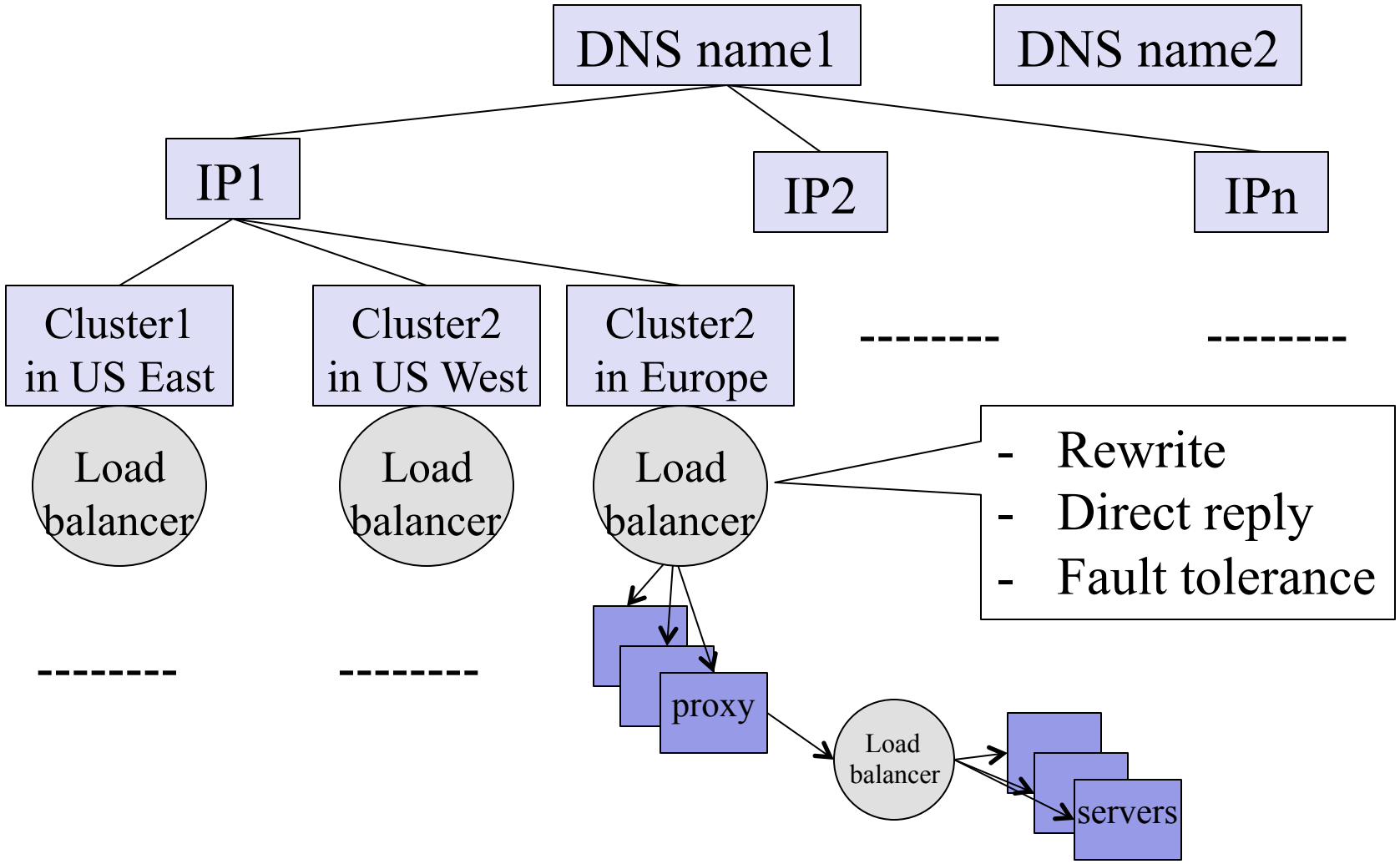
- No dedicated load balancer
- All servers in the cluster receive all packets
- All servers within the cluster simultaneously run a mapping algorithm to determine which server should handle the packet. Those servers not required to service the packet simply discard it.
  - Mapping (ranking) algorithm: computing the "winning" server according to host priorities, multicast or unicast mode, port rules, affinity, load percentage distribution, client IP address, client port number, other internal load information

http://technet.microsoft.com/en-us/library/cc739506%28WS.10%29.aspx

Discussion

- Compare the design of using Load Balancer vs Microsoft NLB

Summary: Example Direction Mechanisms



Outline

- Admin and recap
- Multiple servers
  - Overview
  - Basic mechanisms
  - Example: YouTube (offline read)

http://video.google.com/videoplay?docid=-6304964351441328559#

You Tube

- 02/2005: Founded by Chad Hurley, Steve Chen and Jawed Karim, who were all early employees of PayPal.
- 10/2005: First round of funding ($11.5 M)
- 03/2006: 30 M video views/day
- 07/2006: 100 M video views/day
- 11/2006: acquired by Google
- 10/2009: Chad Hurley announced in a blog that YouTube serving well over 1 B video views/day (avg = 11,574 video views /sec)

Pre-Google Team Size

- 2 Sysadmins
- 2 Scalability software architects
- 2 feature developers
- 2 network engineers
- 1 DBA
- 0 chefs

YouTube Design Alg.

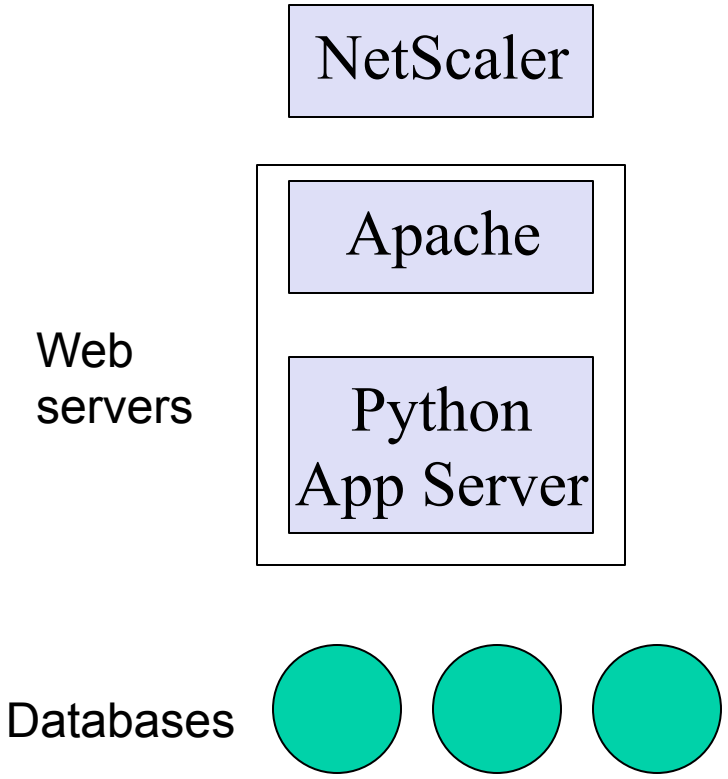
```
while (true)
{
  identify_and_fix_bottlenecks();
  drink();
  sleep();
  notice_new_bottleneck();
}
```

YouTube Major Components

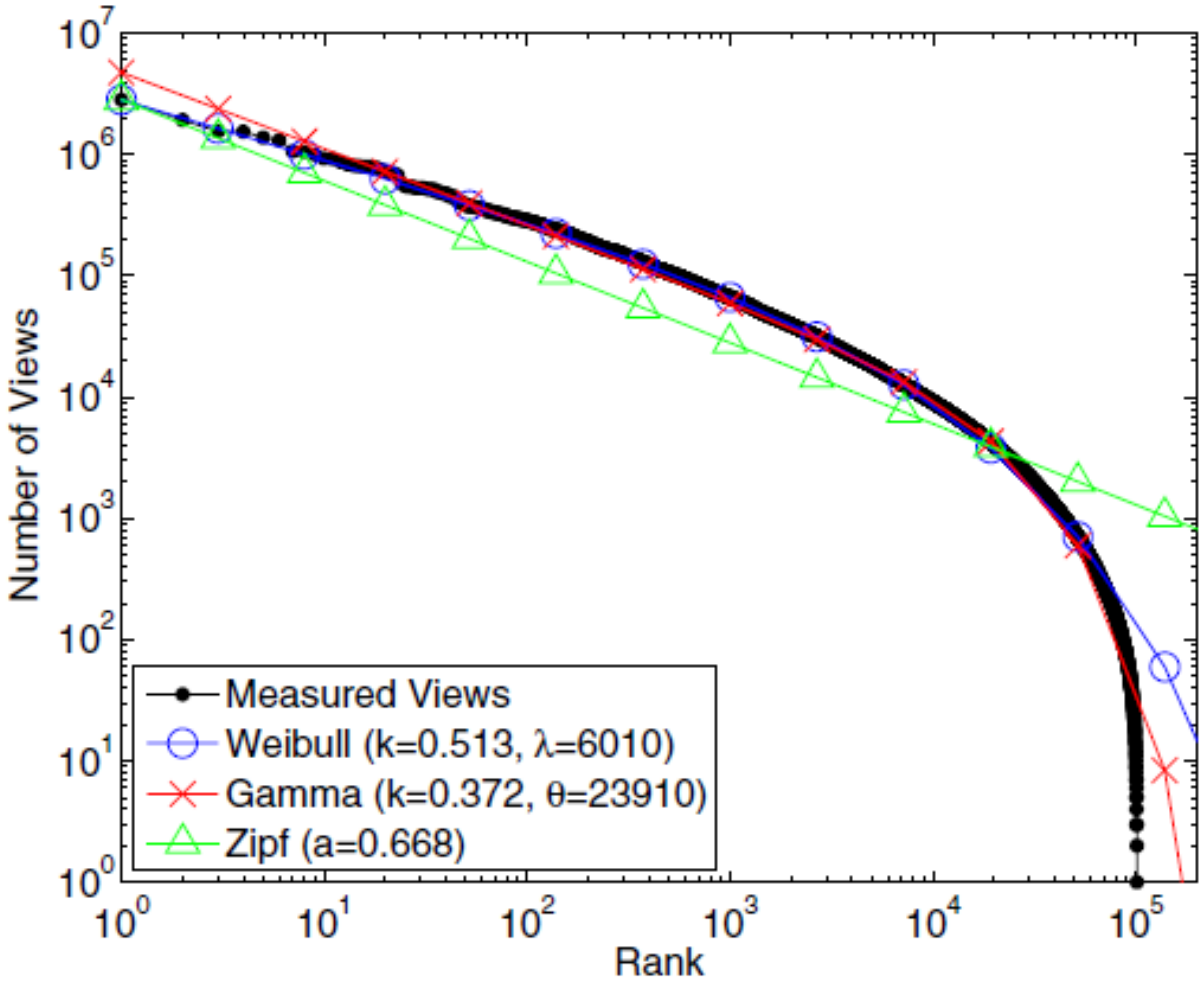
- Web servers
- Video servers
- Thumbnail servers
- Database servers

YouTube: Web Servers

- Components
  - Netscaler load balancer; Apache; Python App Servers; Databases
- Python
  - Web code (CPU) is not bottleneck
    - JIT to C to speedup
    - C extensions
    - Pre-generate HTML responses
  - Development speed more important



YouTube: Video Popularity



See "Statistics and Social Network of YouTube Videos", 2008.

YouTube: Video Popularity

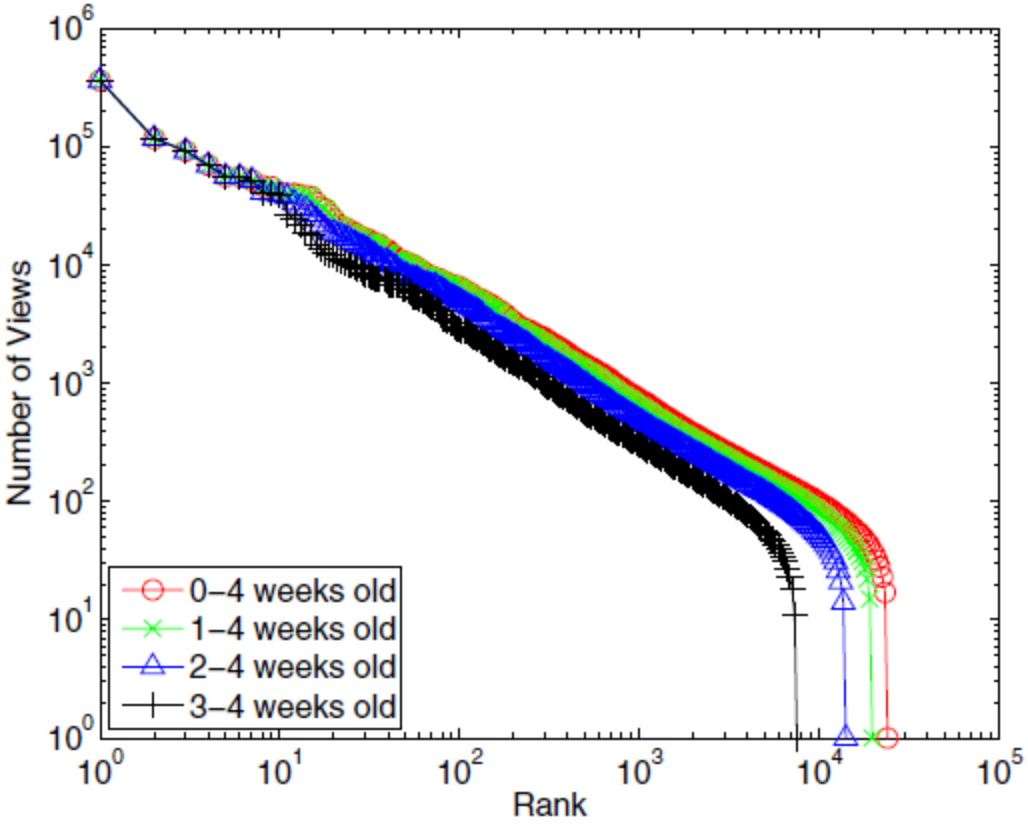


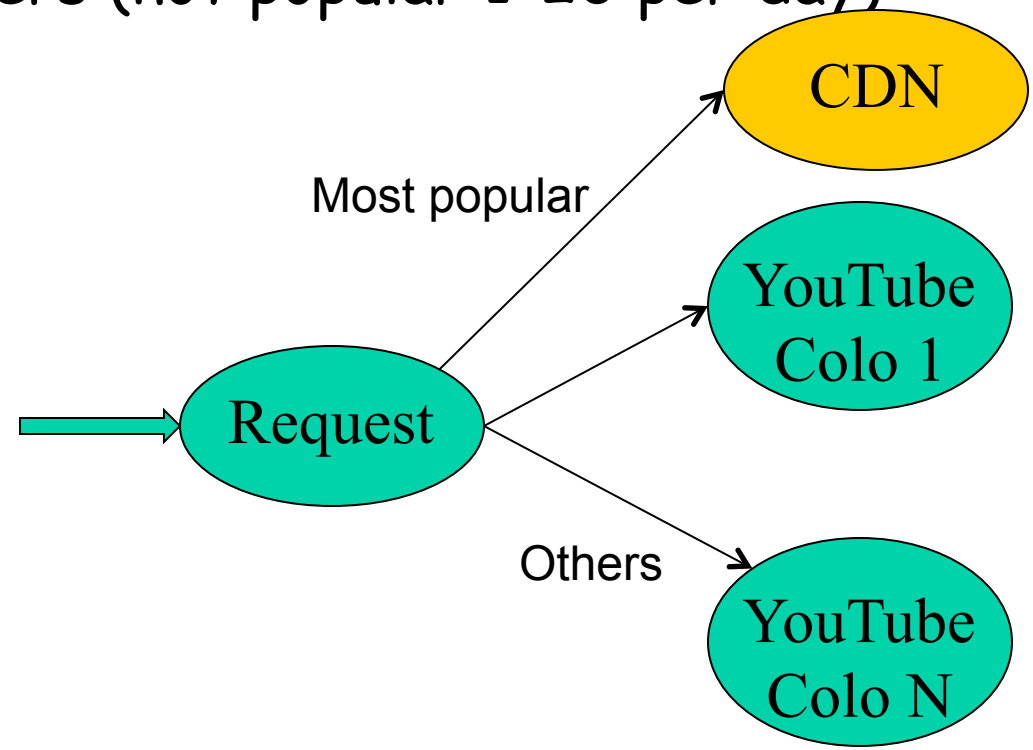
Fig. 8. Recently added YouTube videos rank by popularity

How to design a system to handle highly skewed distribution?

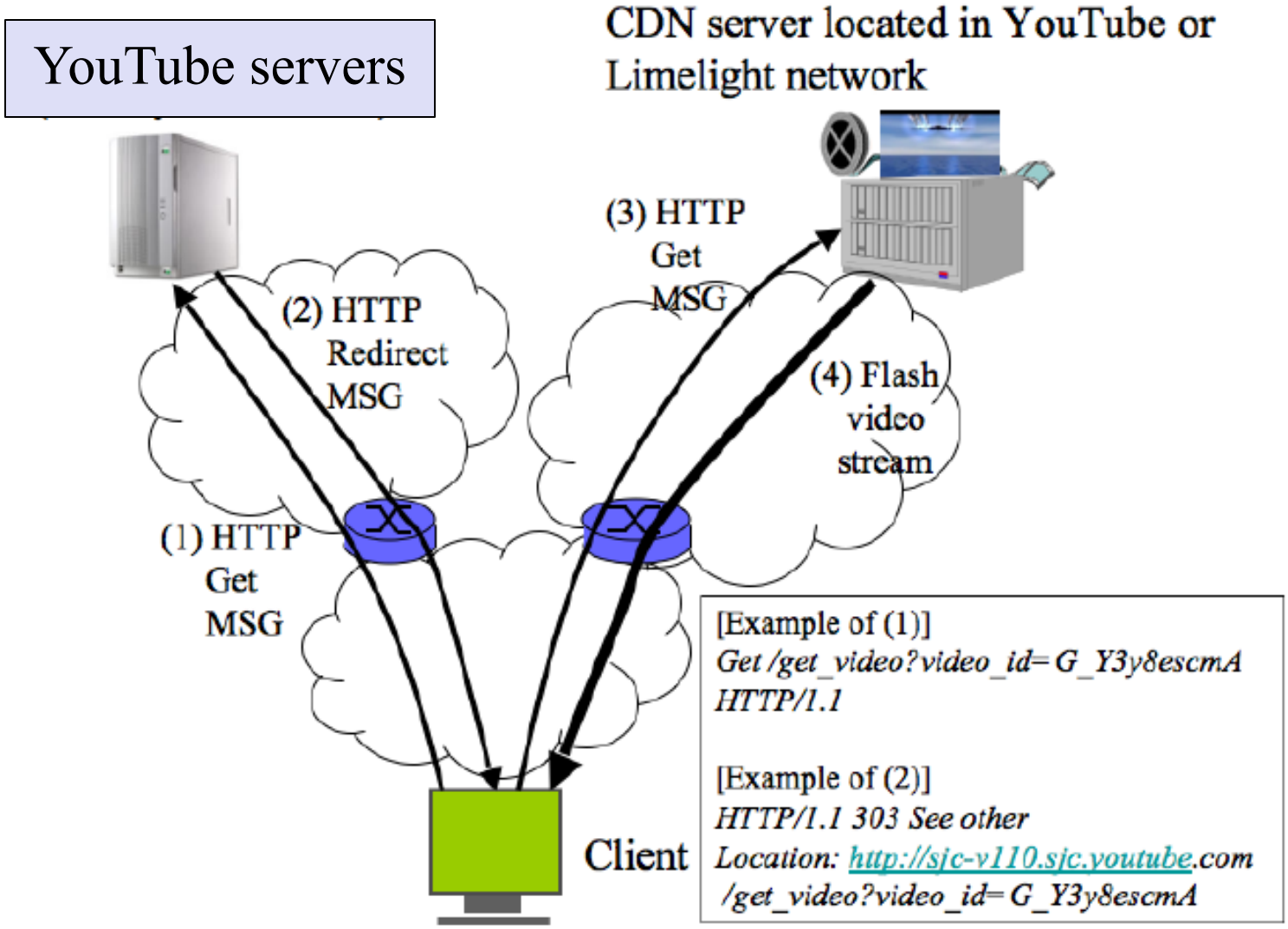
See "Statistics and Social Network of YouTube Videos", 2008.

YouTube: Video Server Architecture

- Tiered architecture
  - CDN servers (for popular videos)
    - Low delay; mostly in-memory operation
  - YouTube servers (not popular 1-20 per day)



YouTube Redirection Architecture



YouTube Video Servers

- Each video hosted by a mini-cluster consisting of multiple machines
- Video servers use the lighttpd web server for video transmission:
  - Apache had too much overhead (used in the first few months and then dropped)
  - Async io: uses epoll to wait on multiple fds
  - Switched from single process to multiple process configuration to handle more connections

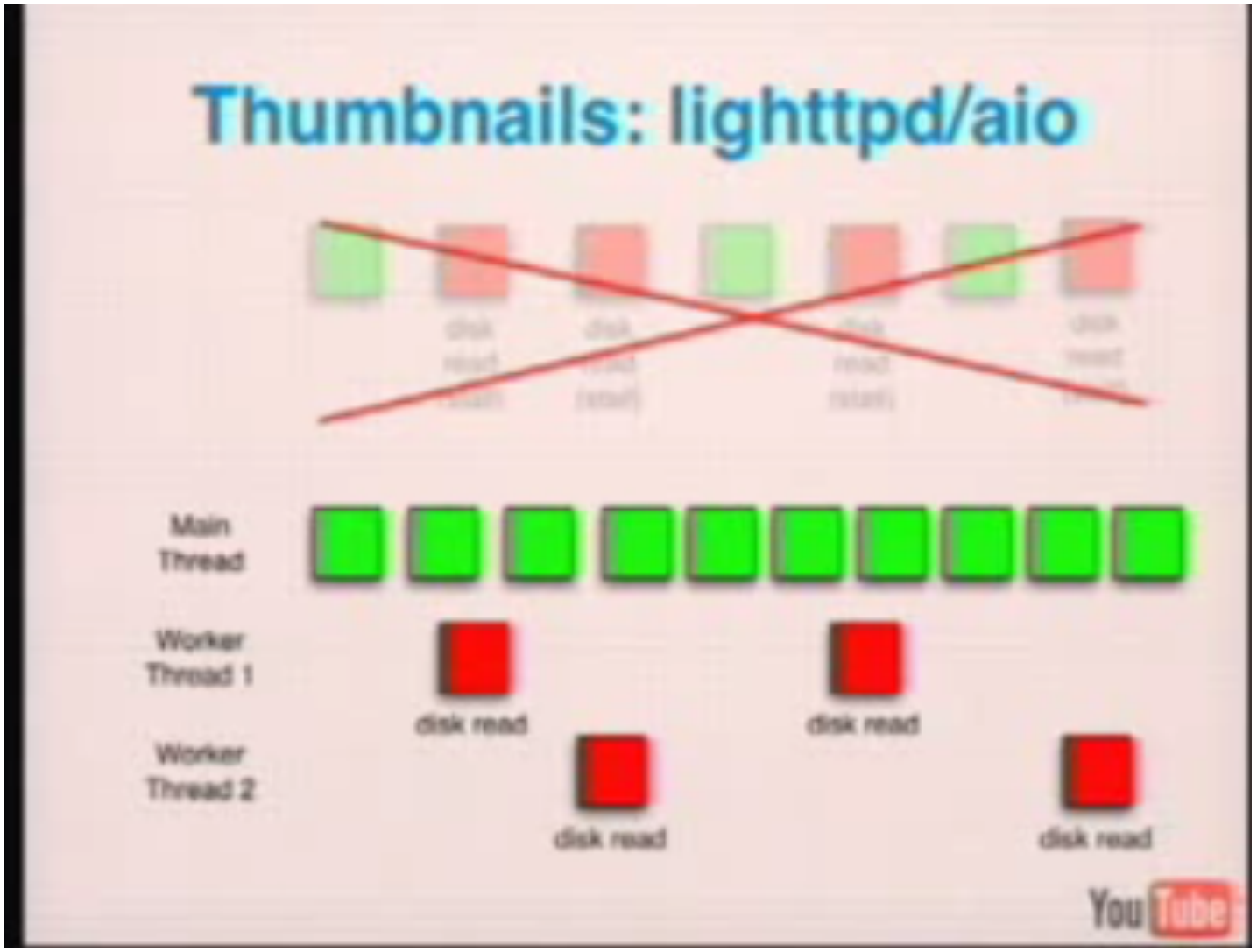
Thumbnail Servers

- Thumbnails are served by a few machines
- Problems running thumbnail servers
  - A high number of requests/sec as web pages can display 60 thumbnails on page
  - Serving a lot of small objects implies
    - lots of disk seeks and problems with file systems inode and page caches
    - may ran into per directory file limit
    - Solution: storage switched to Google BigTable

Thumbnail Server Software Architecture

- Design 1: Squid in front of Apache
  - Problems
    - Squid worked for a while, but as load increased performance eventually decreased: Went from 300 requests/second to 20
    - under high loads Apache performed badly, changed to lighttpd
- Design 2: lighttpd default: By default lighttpd uses a single thread
  - Problem: often stalled due to I/O
- Design 3: switched to multiple processes contending on shared accept
  - Problems: high contention overhead/individual caches

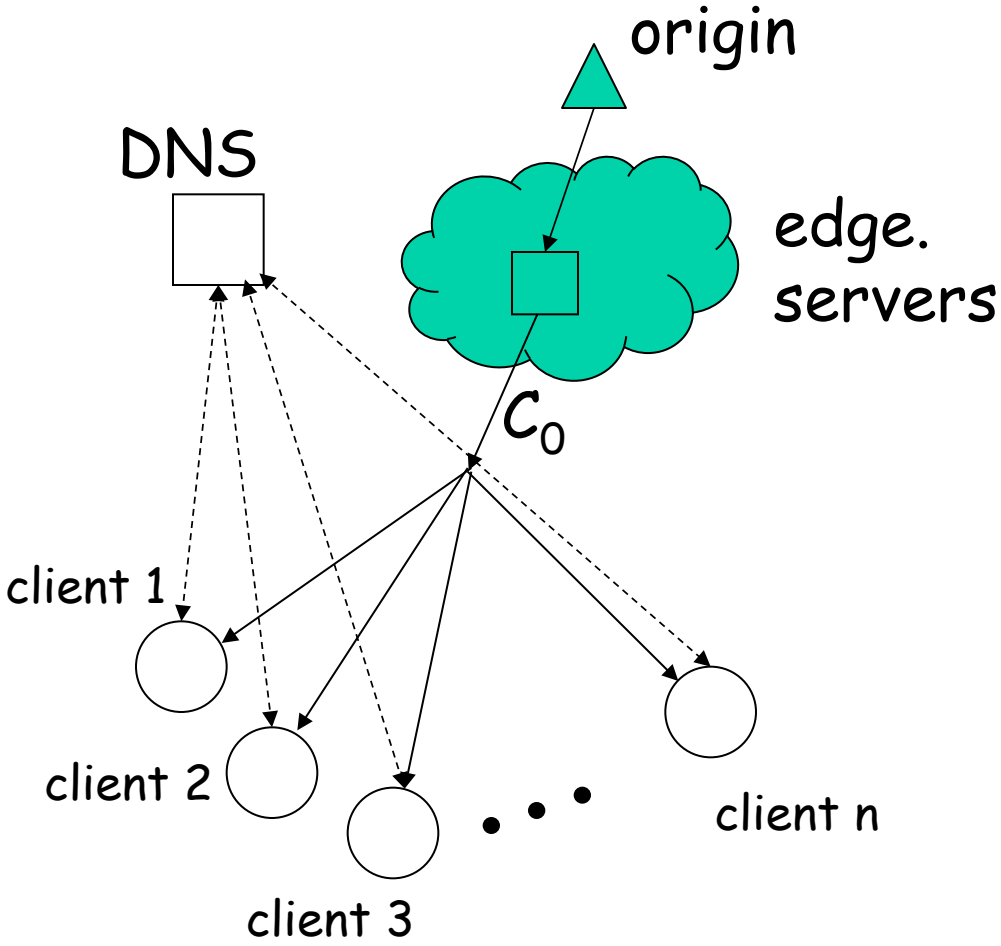
Thumbnails Server: lighttpd/aio



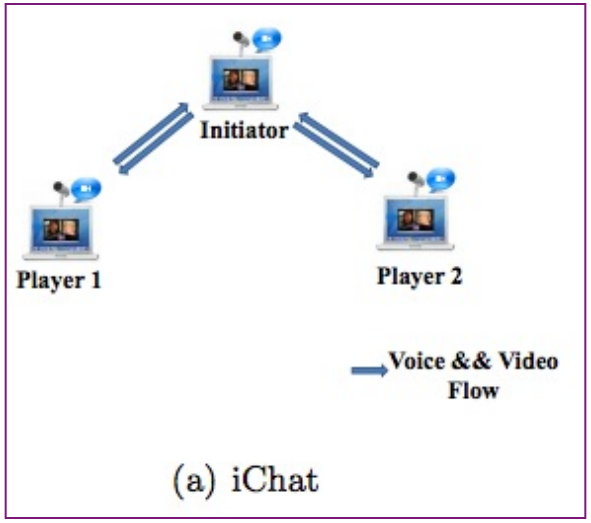
Outline

- Admin and recap
- Multiple servers
- Application overlays

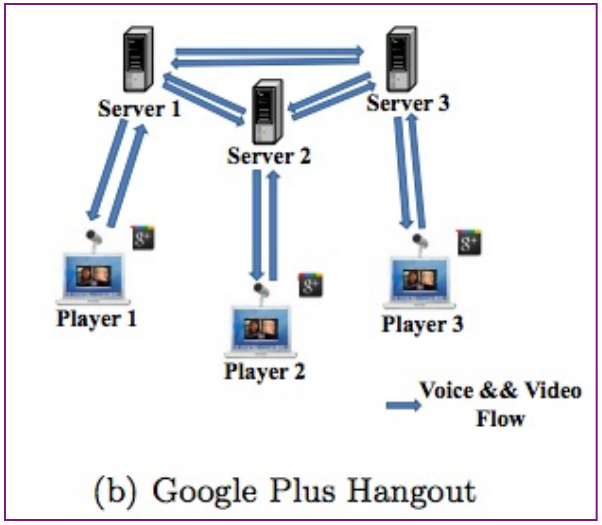
Scalability of Server-Only Approaches



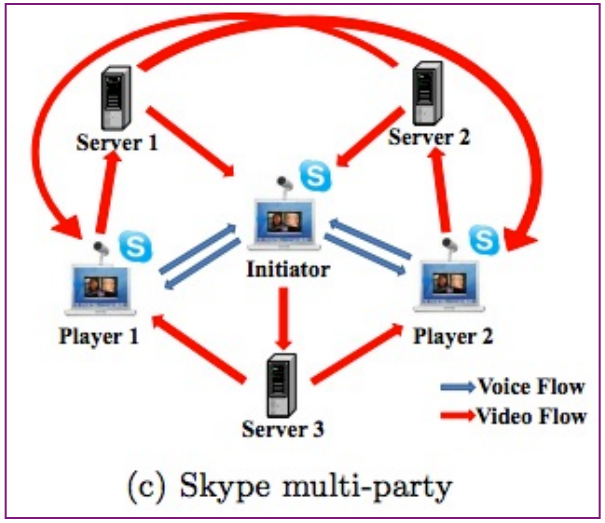
Server + Hosts Systems



(a) iChat



(b) Google Plus Hangout

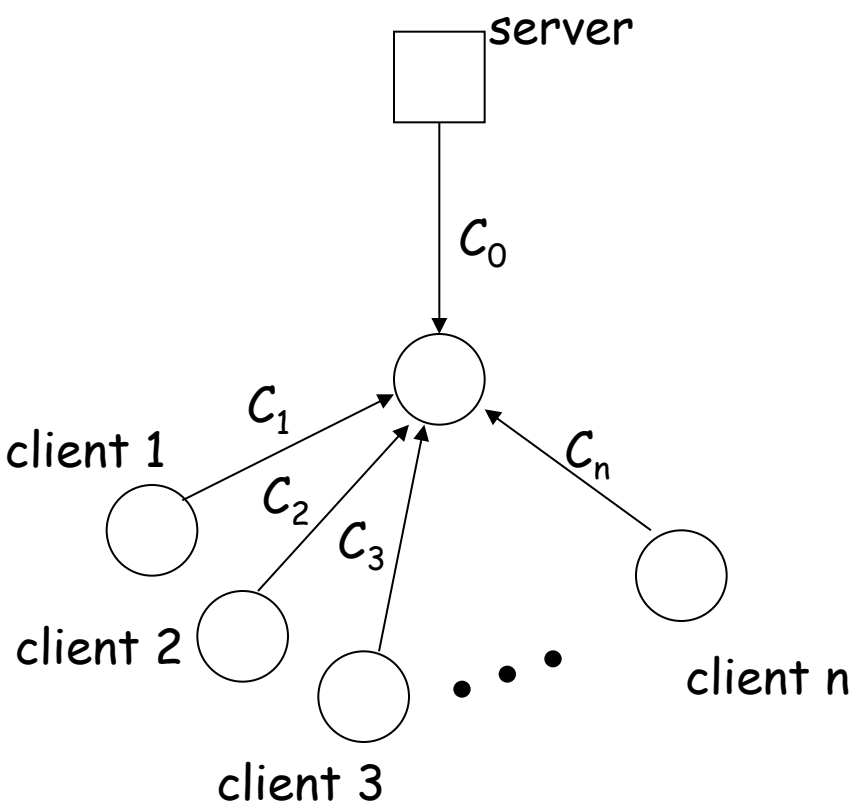


(c) Skype multi-party

http://eeweb.poly.edu/faculty/yongliu/docs/imc12.pdf

An Upper Bound on Scalability

- Assume
  - need to achieve same rate to all clients
  - only uplinks can be bottlenecks
- What is an upper bound on scalability?

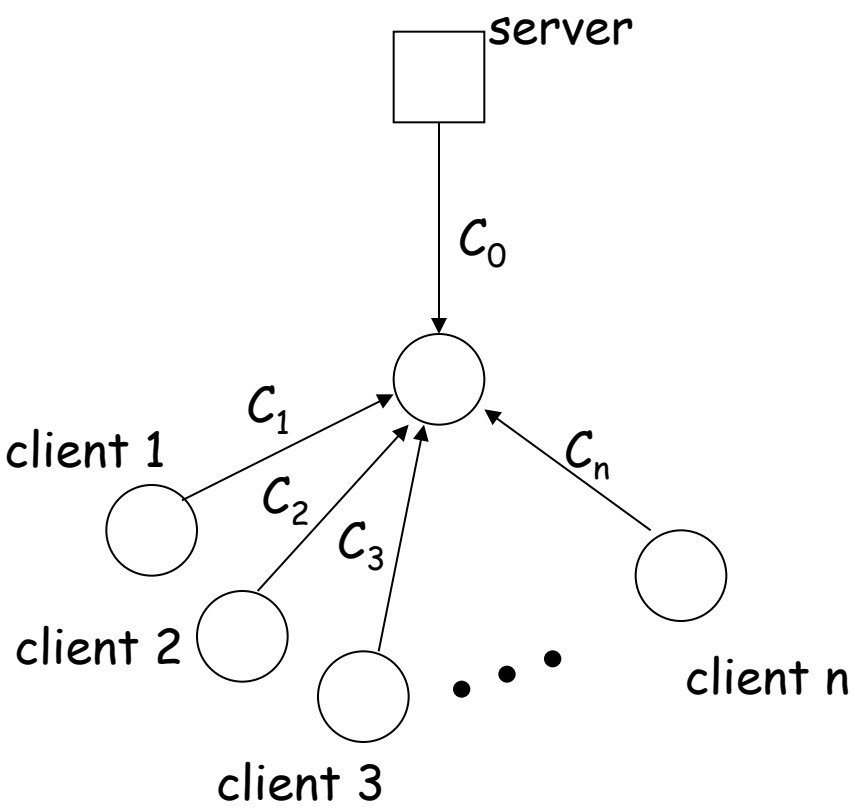


The Scalability Problem

- Maximum throughput

  $\mathbf{R = \min\{C_0, (C_0+\Sigma C_i)/n\}}$

- The bound is theoretically approachable



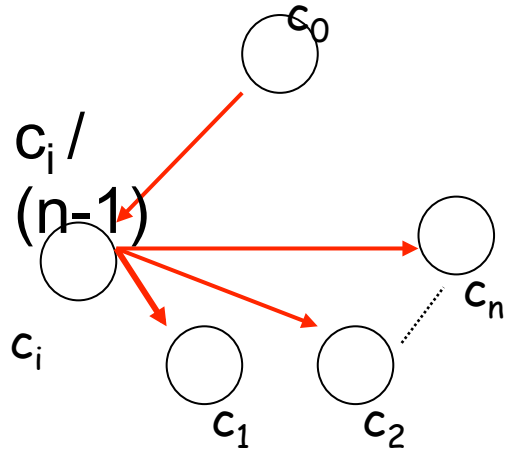
Theoretical Capacity: upload is bottleneck

$$R = \min\{C_0, (C_0+\Sigma C_i)/n\}$$

- Assume $c_0 > (C_0+\Sigma C_i)/n$
- Tree i:

  server → client i: $c_i/(n-1)$

  client i → other n-1 clients

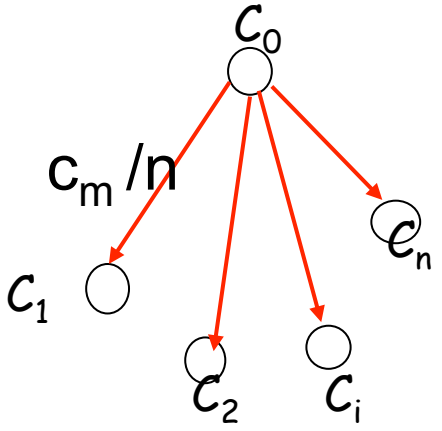


- Tree 0:

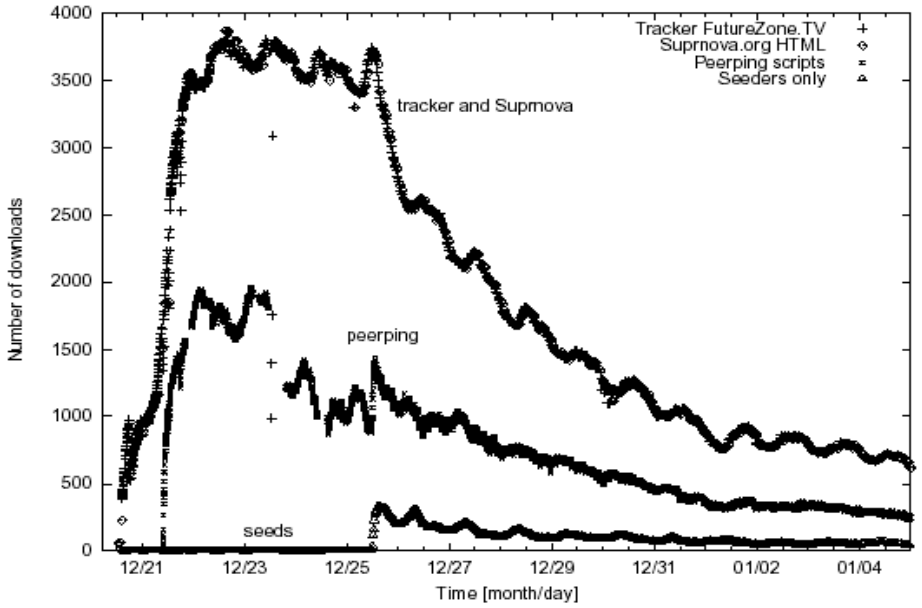
  server has remaining

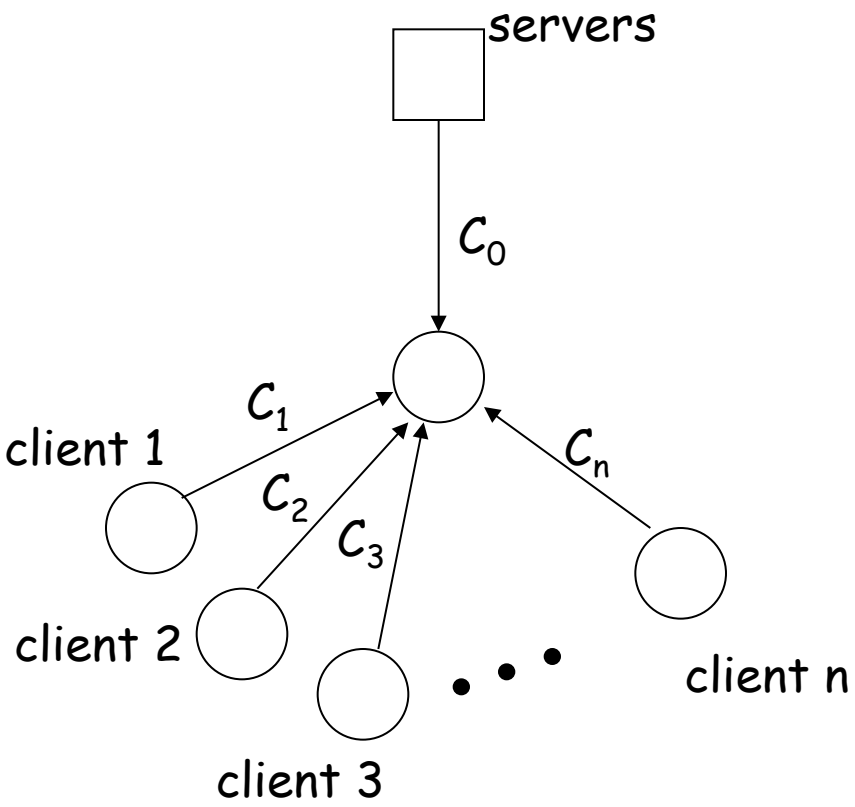
  $c_m = c0 - (c1 + c2 + \ldots cn)/(n-1)$

  send to client i: cm/n



Why not Building the Trees?

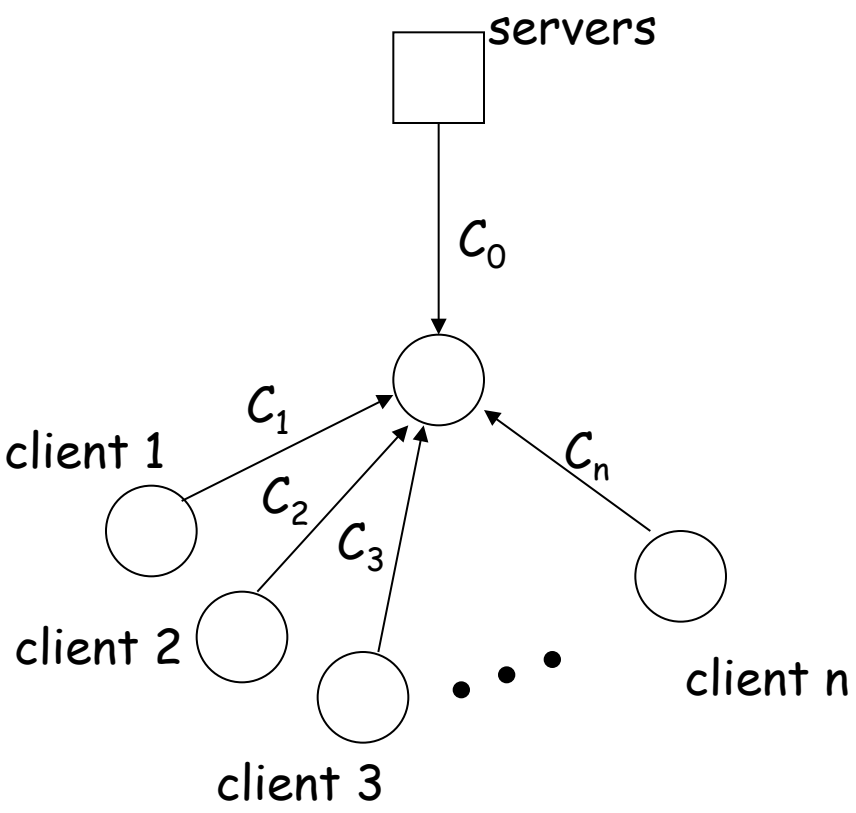




- Clients come and go (churns): maintaining the trees is too expensive
- Each client needs N connections

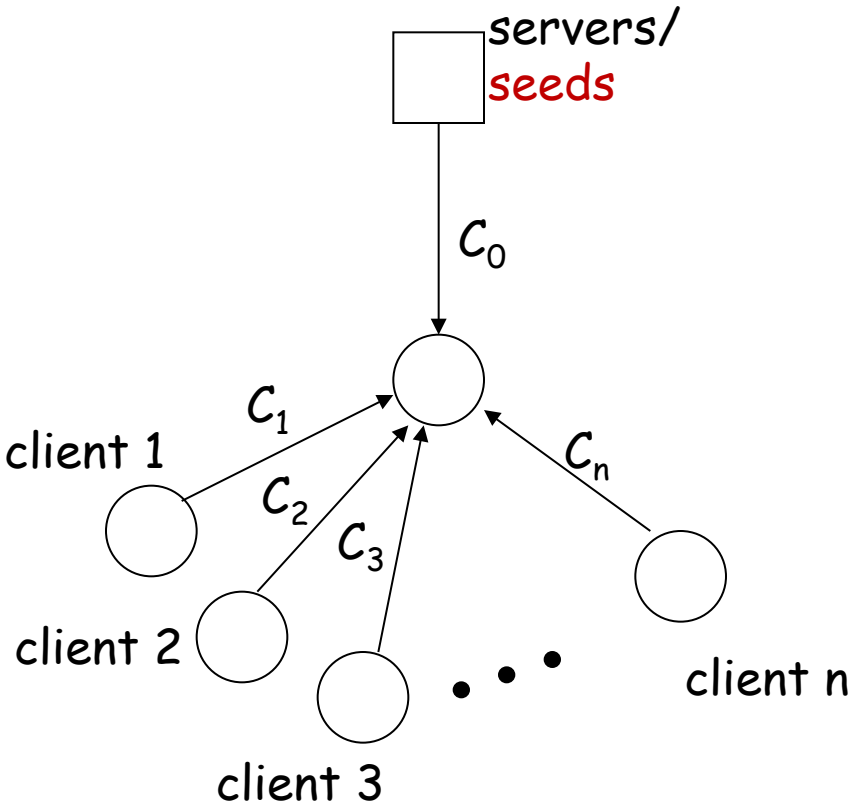
Server+Host (P2P) Content Distribution: Key Design Issues

- Robustness
  - Resistant to churns and failures
- Efficiency
  - A client has content that others need; otherwise, its upload capacity may not be utilized
- Incentive: clients are willing to upload
  - Some real systems nearly 50% of all responses are returned by the top 1% of sharing hosts



Discussion: How to handle the issues?

- Robustness
- Efficiency
- Incentive



Example: BitTorrent

- A P2P file sharing protocol
- Created by Bram Cohen in 2004
  - Spec at bep_0003: http://www.bittorrent.org/beps/bep_0003.html

BitTorrent: Lookup

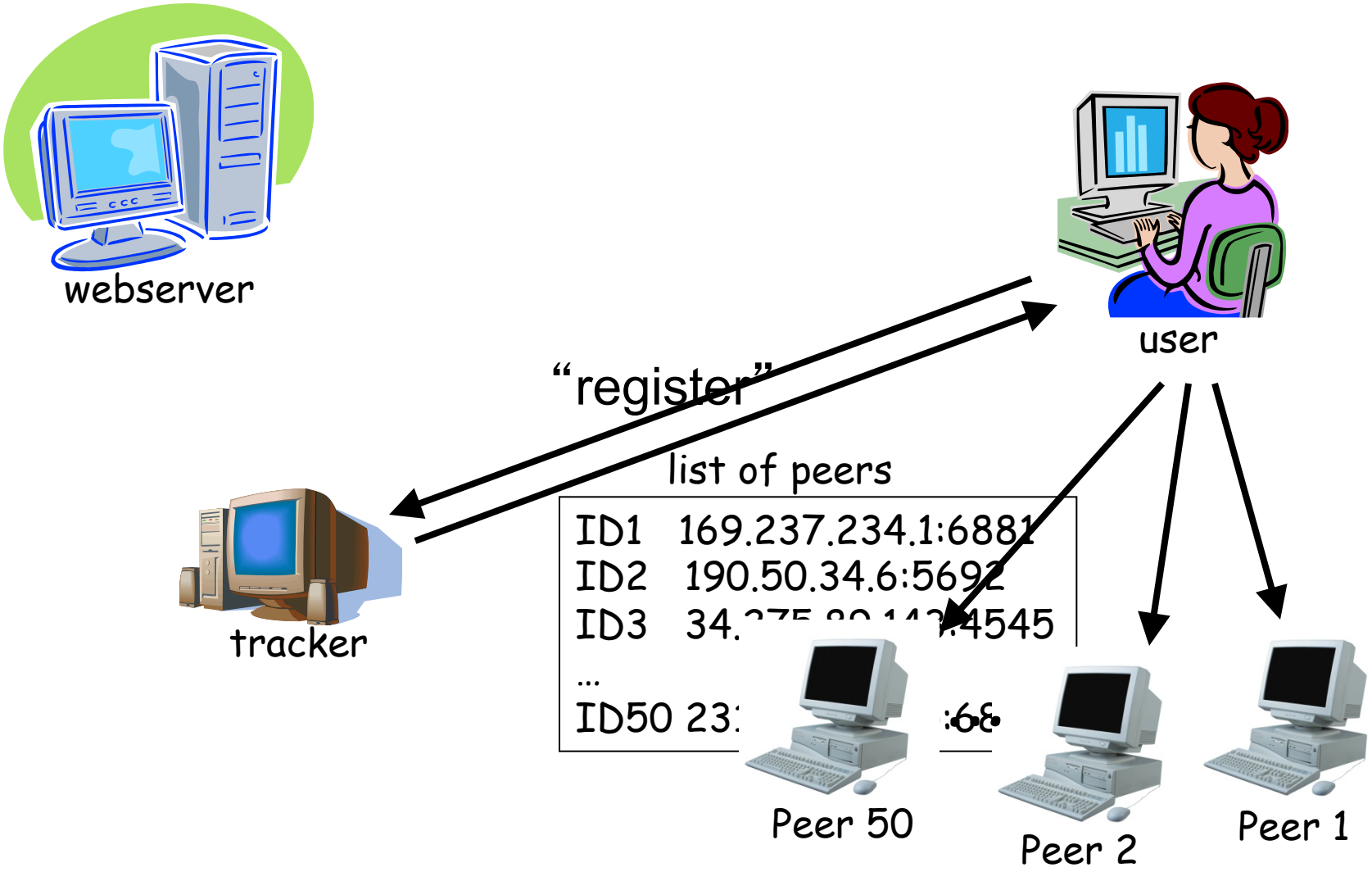
Metadata (.torrent) File Structure

- Meta info contains information necessary to contact the tracker and describes the files in the torrent
  - URL of tracker
  - file name
  - file length
  - piece length (typically 256KB)
  - SHA-1 hashes of pieces for verification
  - also creation date, comment, creator, …

Tracker Protocol

- Communicates with clients via HTTP/HTTPS
- Client GET request
  - info_hash: uniquely identifies the file
  - peer_id: chosen by and uniquely identifies the client
  - client IP and port
  - numwant: how many peers to return (defaults to 50)
  - stats: *e.g.*, bytes uploaded, downloaded
- Tracker GET response
  - interval: how often to contact the tracker
  - list of peers, containing peer id, IP and port
  - stats

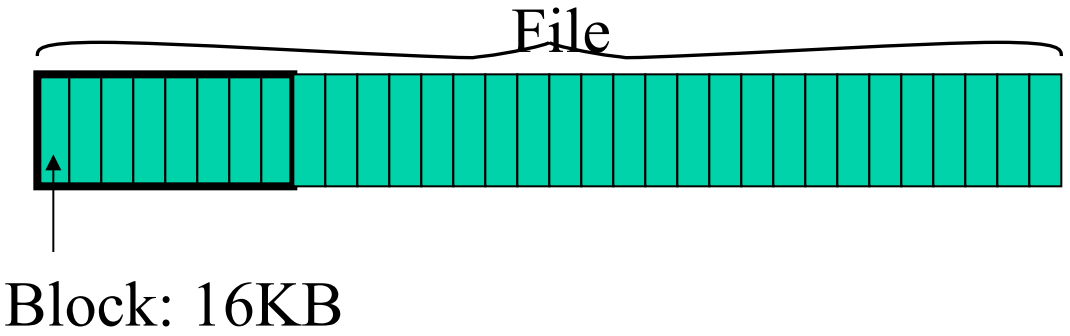
Tracker Protocol



Robustness and efficiency: Piece-based Swarming

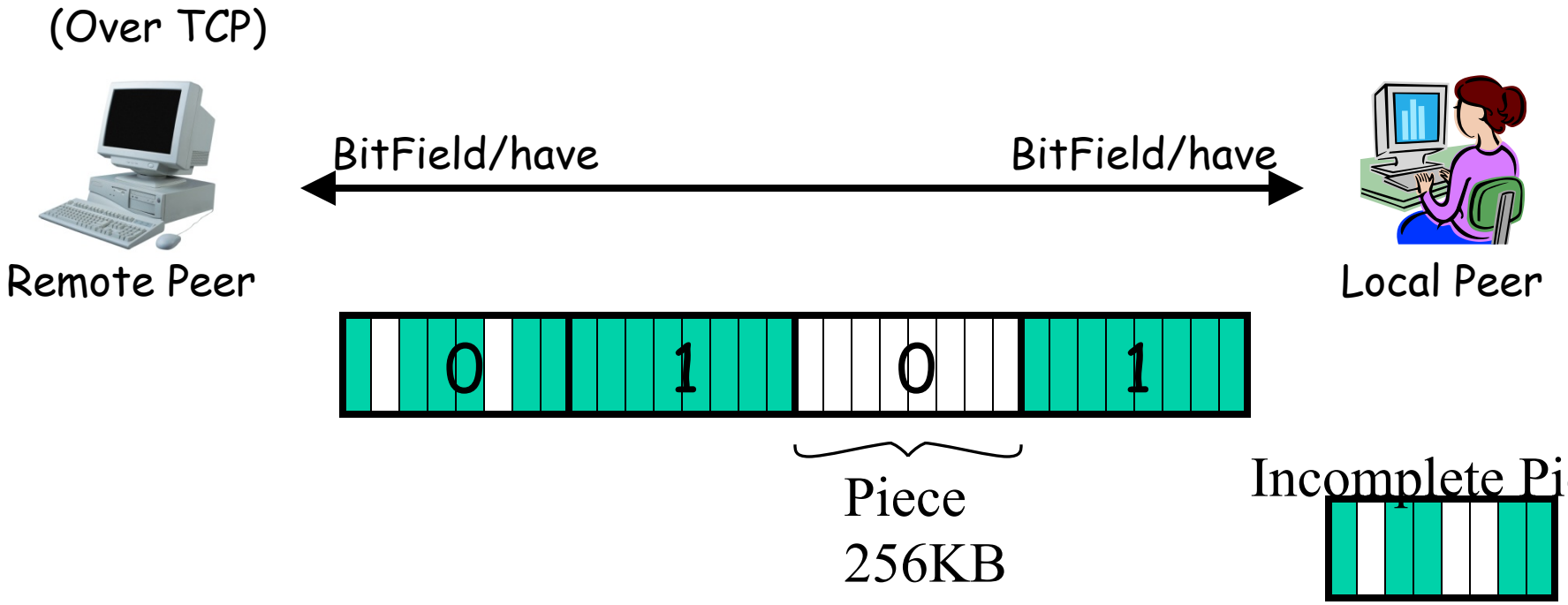
- Divide a large file into small blocks and request block-size content from different peers

Block: unit of download



- If do not finish downloading a block from one peer within timeout (say due to churns), switch to requesting the block from another peer

Detail: Peer Protocol

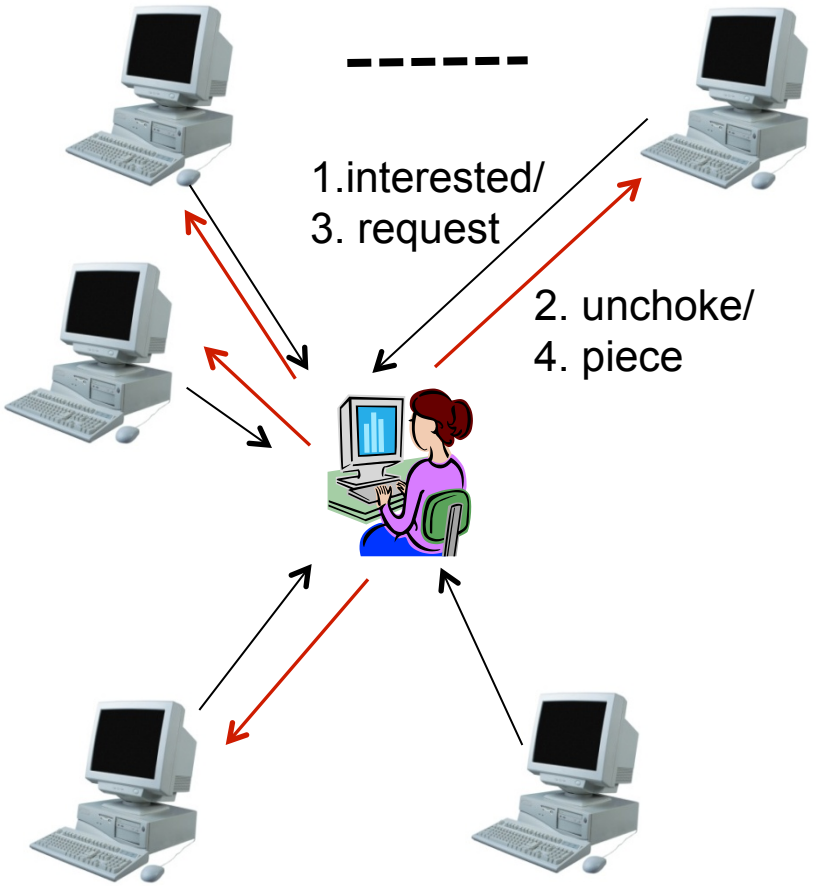


- Peers exchange bitmap representing content availability
  - `bitfield` msg during initial connection
  - `have` msg to notify updates to bitmap
  - to reduce bitmap size, aggregate multiple blocks as a piece

Peer Request

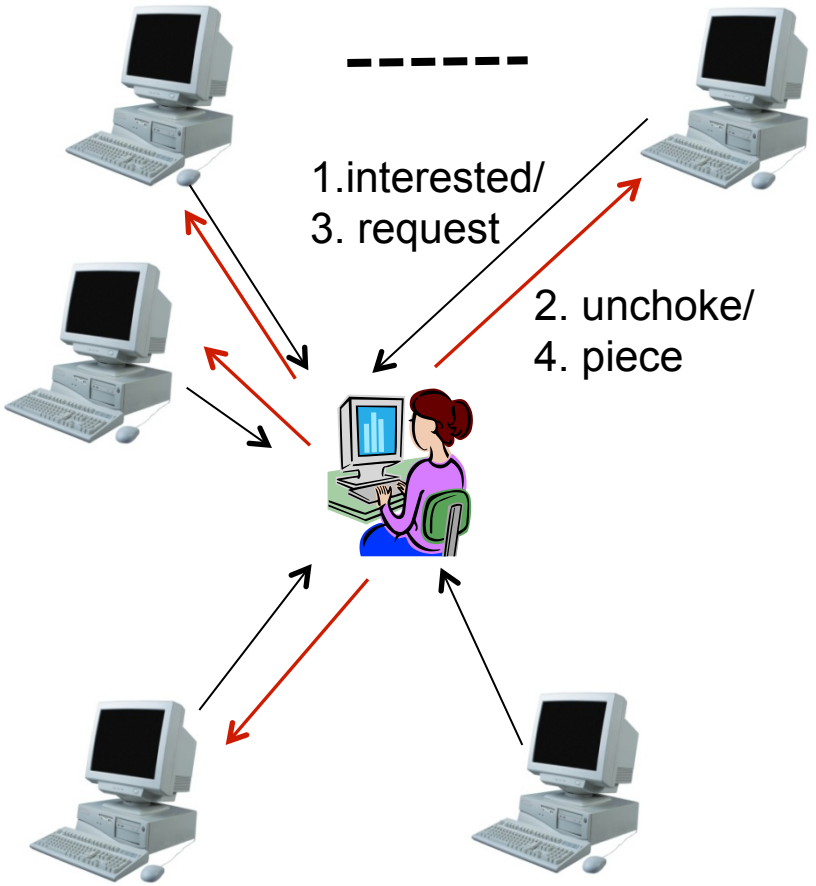
http://www.bittorrent.org/beps/bep_0003.html

- If peer A has a piece that peer B needs, peer B sends `interested` to A
- `unchoke`: indicate that A allows B to request
- `request`: B requests a specific block from A
- `piece`: specific data



Key Design Points

- request:
  - which data blocks to request?
- unchoke:
  - which peers to serve?



Request: Block Availability

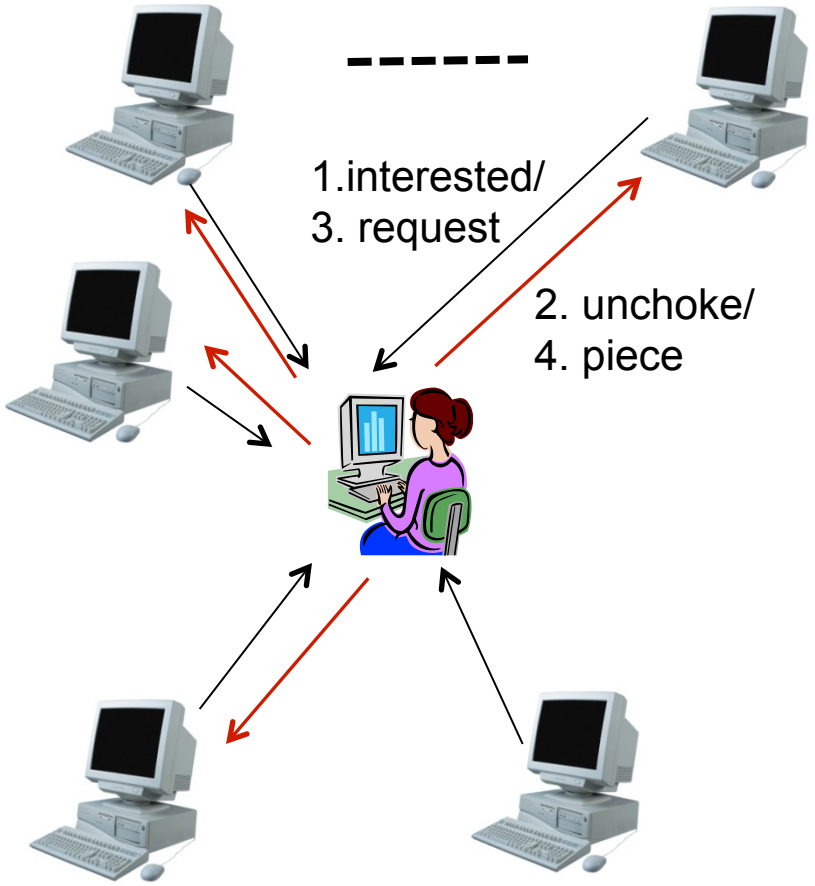
- Request (local) rarest first
  - achieves the fastest replication of rare pieces
  - obtain something of value

Block Availability: Revisions

- When downloading starts (first 4 pieces): choose at random and request them from the peers
  - get pieces as quickly as possible
  - obtain something to offer to others
- Endgame mode
  - defense against the "last-block problem": cannot finish because missing a few last pieces
  - send requests for missing pieces to all peers in our peer list
  - send `cancel` messages upon receipt of a piece

BitTorrent: Unchoke

- Periodically (typically every 10 seconds) calculate data-receiving rates from all peers
- Upload to (*unchoke*) the fastest
  - constant number (4) of unchoking slots
  - partition upload bw equally among unchoked



commonly referred to as "tit-for-tat" strategy

Optimistic Unchoking

- Periodically select a peer at random and upload to it
  - typically every 3 unchoking rounds (30 seconds)
- Multi-purpose mechanism
  - allow bootstrapping of new clients
  - continuously look for the fastest peers (exploitation vs exploration)

BitTorrent Fluid Analysis

- Normalize file size to 1
- x(t): number of downloaders (also known as leechers) who do not have all pieces at time t.
- y(t): number of seeds in the system at time t.
- λ: the arrival rate of new requests.
- μ: the uploading bandwidth of a given peer.
- c: the downloading bandwidth of a given peer, assume $c \geq \mu$.
- θ: the rate at which downloaders abort download.
- γ: the rate at which seeds leave the system.
- η: indicates the effectiveness of downloader sharing, η takes values in [0, 1].

System Evolution

$$\begin{aligned}\frac{\mathrm{d}x}{\mathrm{d}t} &= \lambda - \theta x(t) - \min\{cx(t), \mu(\eta x(t) + y(t))\},\\ \frac{\mathrm{d}y}{\mathrm{d}t} &= \min\{cx(t), \mu(\eta x(t) + y(t))\} - \gamma y(t),\end{aligned}$$

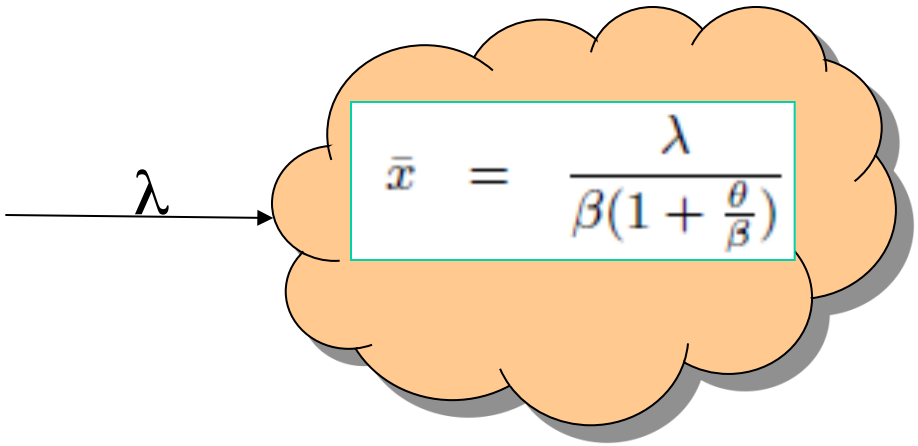
Solving steady state: $\frac{\mathrm{d}x(t)}{\mathrm{d}t} = \frac{\mathrm{d}y(t)}{\mathrm{d}t} = 0$

Define $\frac{1}{\beta} = \max\{\frac{1}{c}, \frac{1}{\eta}(\frac{1}{\mu} - \frac{1}{\gamma})\}$

$$\begin{aligned}\bar{x} &= \frac{\lambda}{\beta(1+\frac{\theta}{\beta})}\\ \bar{y} &= \frac{\lambda}{\gamma(1+\frac{\theta}{\beta})}.\end{aligned}$$

System State

Q: How long does each downloader stay as a downloader?



$$T = \frac{1}{\theta + \beta}$$

$$\frac{1}{\beta} = \max\{\frac{1}{c}, \frac{1}{\eta}(\frac{1}{\mu} - \frac{1}{\gamma})\}$$

Outline

- Admin and recap
- Multiple servers
- Application overlays
  - Overlays for scalability
  - Overlays for anonymity

Problem

- Internet surveillance like traffic analysis reveals users privacy
- Encryption does not hide packet headers, which still reveal a great deal about users
- End-to-end anonymity is needed

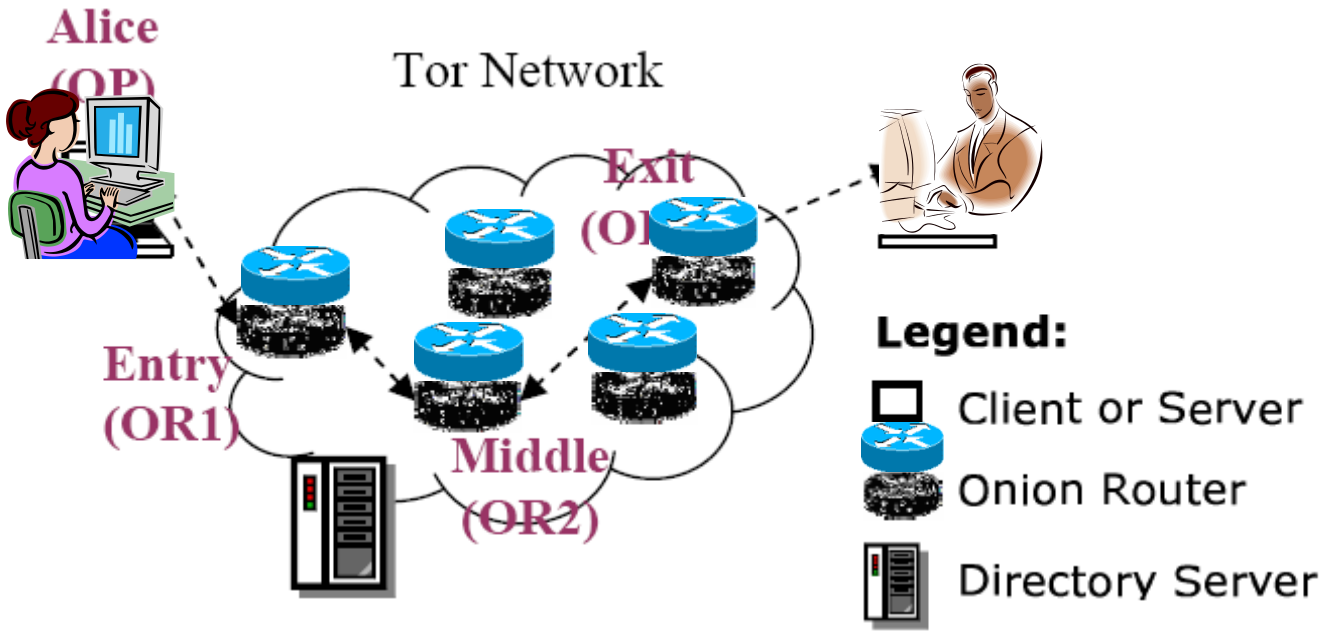
- Solution: distributed, anonymous networks

Example: Tor Networks

- Basic idea
  - Overlay network using Onion Routers (OR)
  - Onion routing so that each onion router knows only the previous and next hop

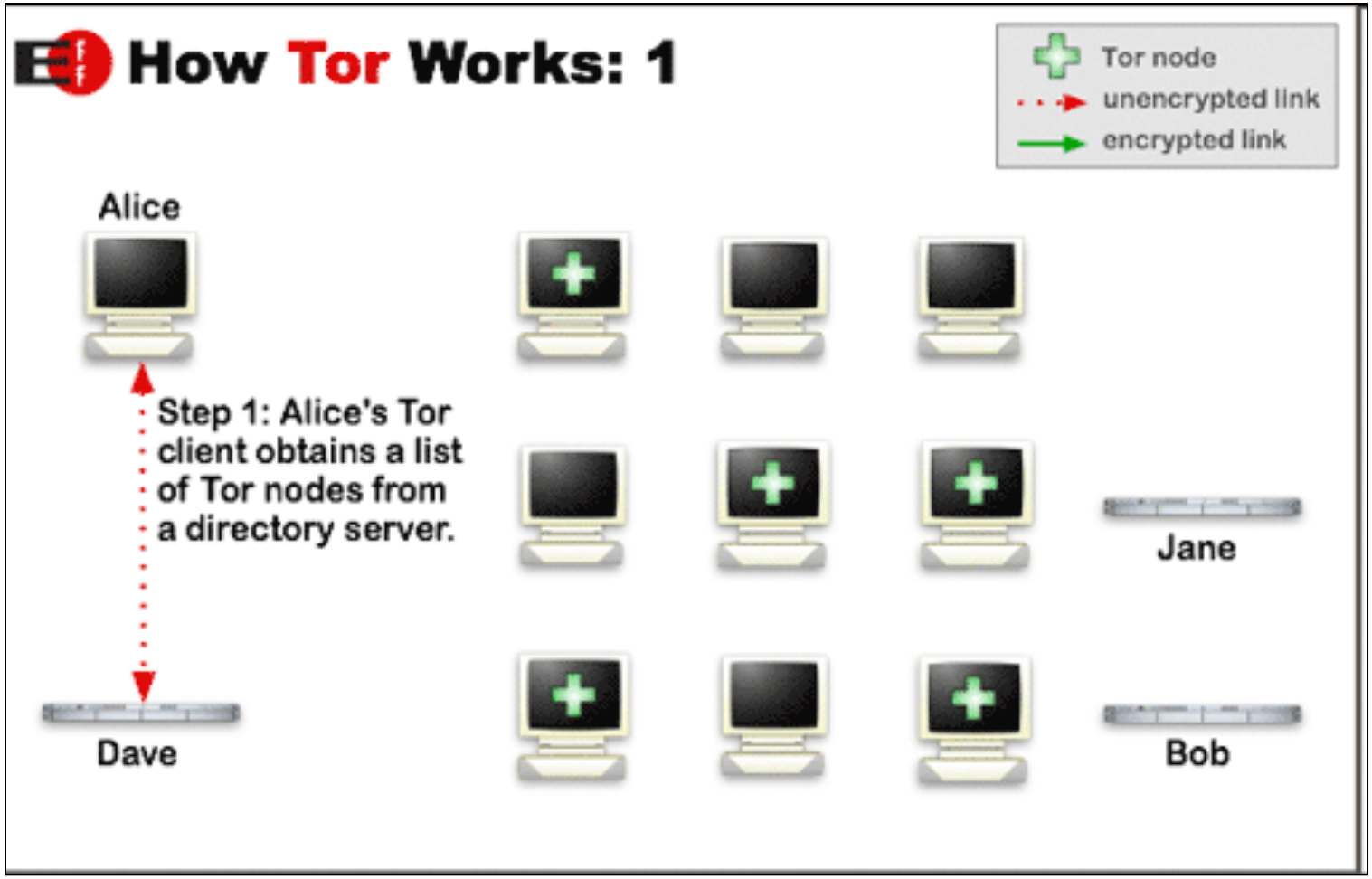
https://torstatus.blutmagie.de/

Tor Components

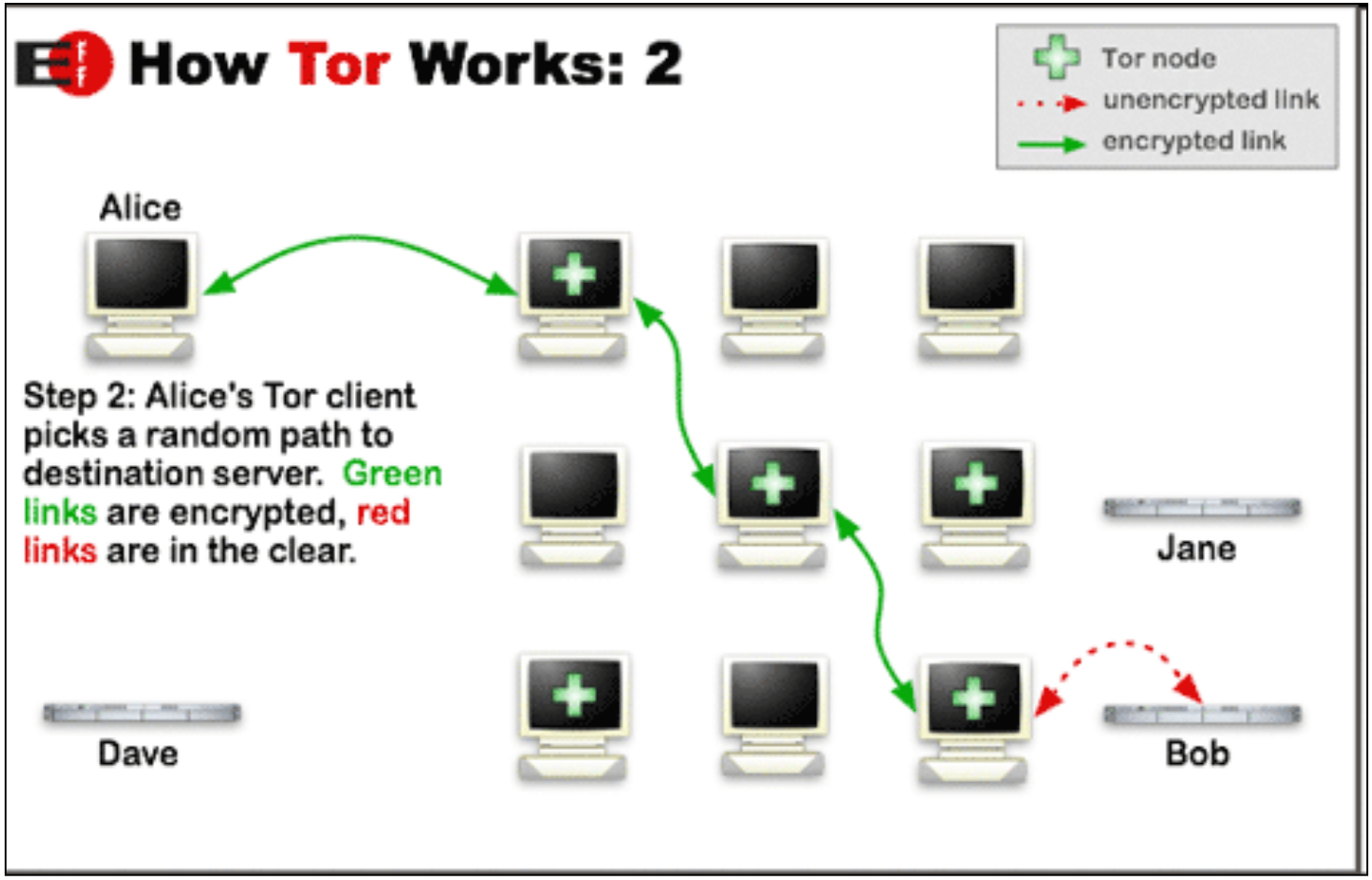


- **Client**: the user of the Tor network
- **Server**: the target TCP applications such as web servers
- **Tor (onion) router**: the special proxy relays the application data
- **Directory server**: servers holding Tor router information

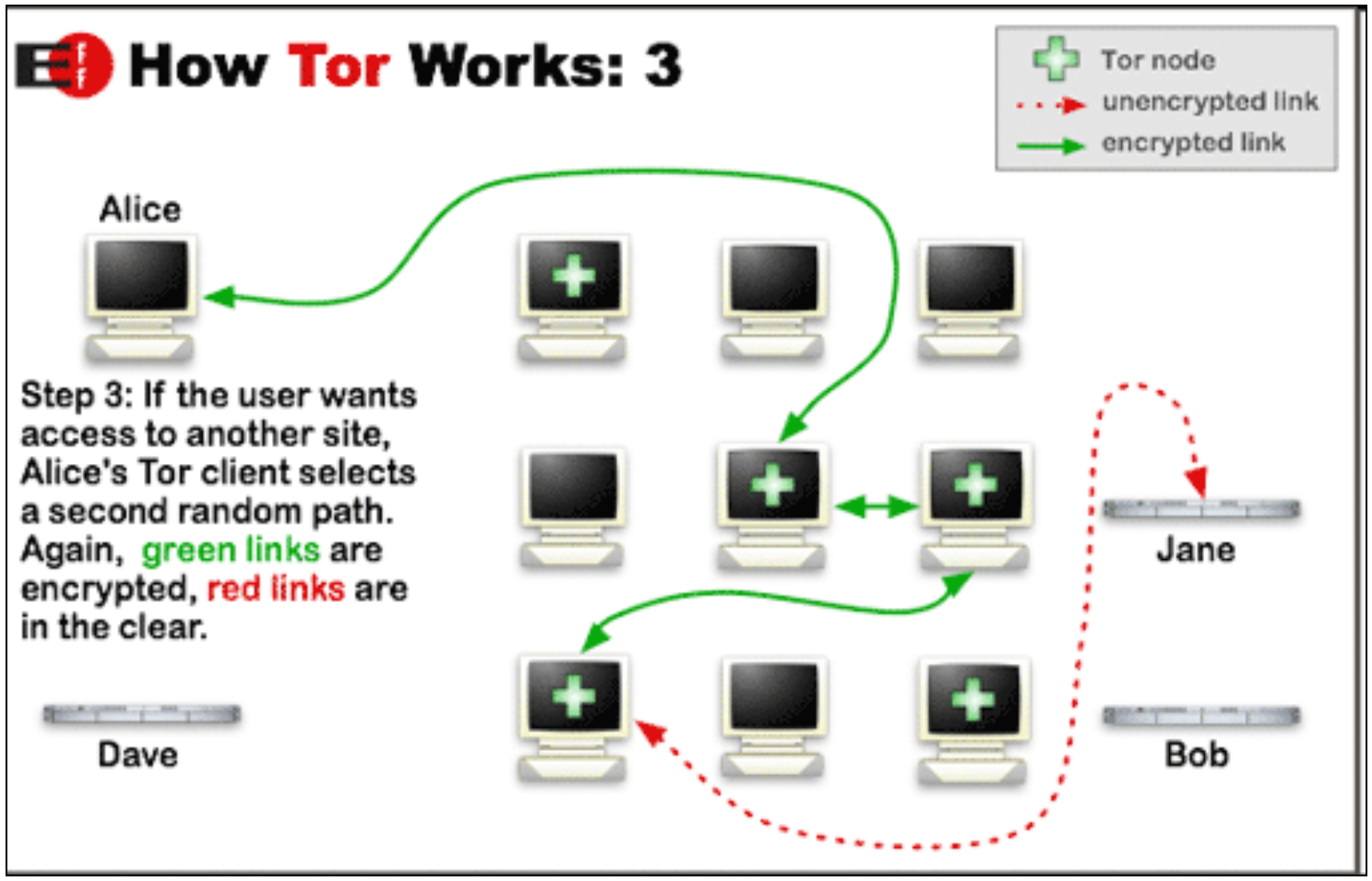
Tor Steps


How Tor Works: 1
Tor node
unencrypted link
encrypted link
Alice
Step 1: Alice's Tor client obtains a list of Tor nodes from a directory server.
Dave
Jane
Bob

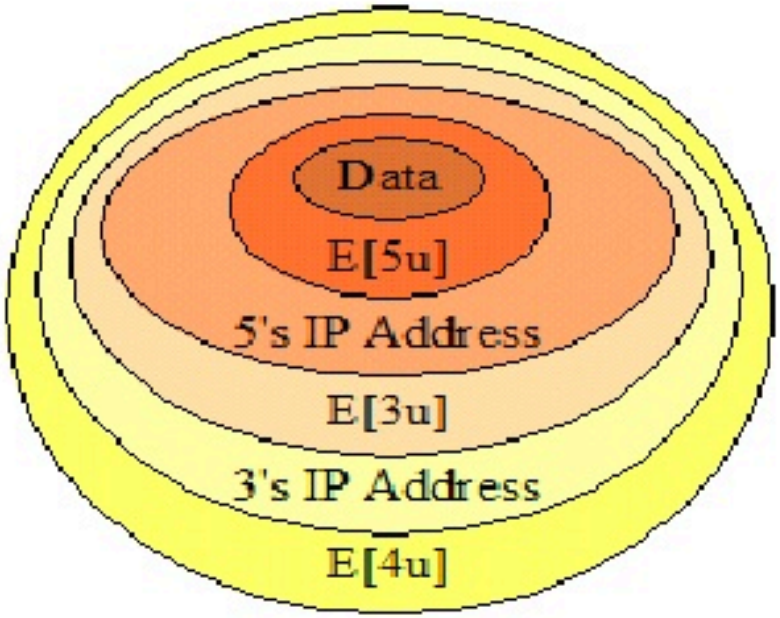
Tor Steps


How Tor Works: 2
Tor node
unencrypted link
encrypted link
Alice
Step 2: Alice's Tor client picks a random path to destination server. Green links are encrypted, red links are in the clear.
Jane
Dave
Bob

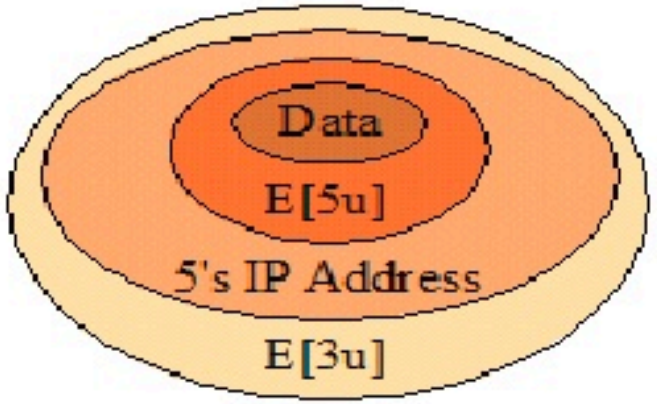
Tor Steps


How Tor Works: 3
Tor node
unencrypted link
encrypted link
Alice
Step 3: If the user wants access to another site, Alice's Tor client selects a second random path. Again, green links are encrypted, red links are in the clear.
Jane
Dave
Bob

Onions using public key



Onion Sent by Client to 4
Router 4 will decrypt the E[4u] layer using its private key, to find the next router's IP address, and encrypted data.

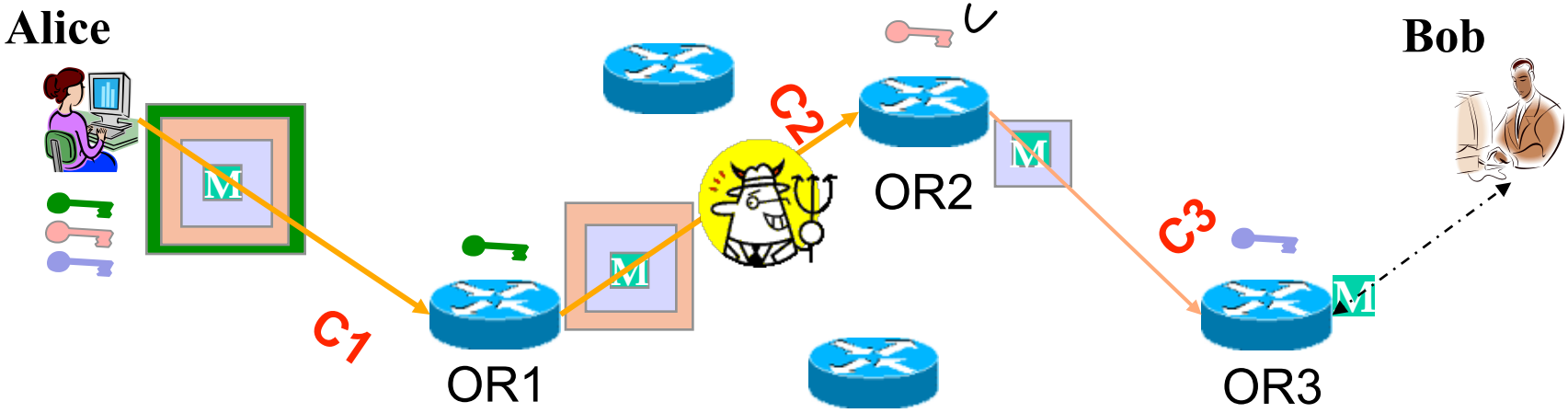


Onion Sent by 4 to 3
Router 3 will decrypt the E[3u] layer using its private key, to find the next router's IP address, and encrypted data.

Onion Sent by 3 to 5
Router 5 will decrypt the E[5u] layer using its private key, to find just the unencrypted data packet.

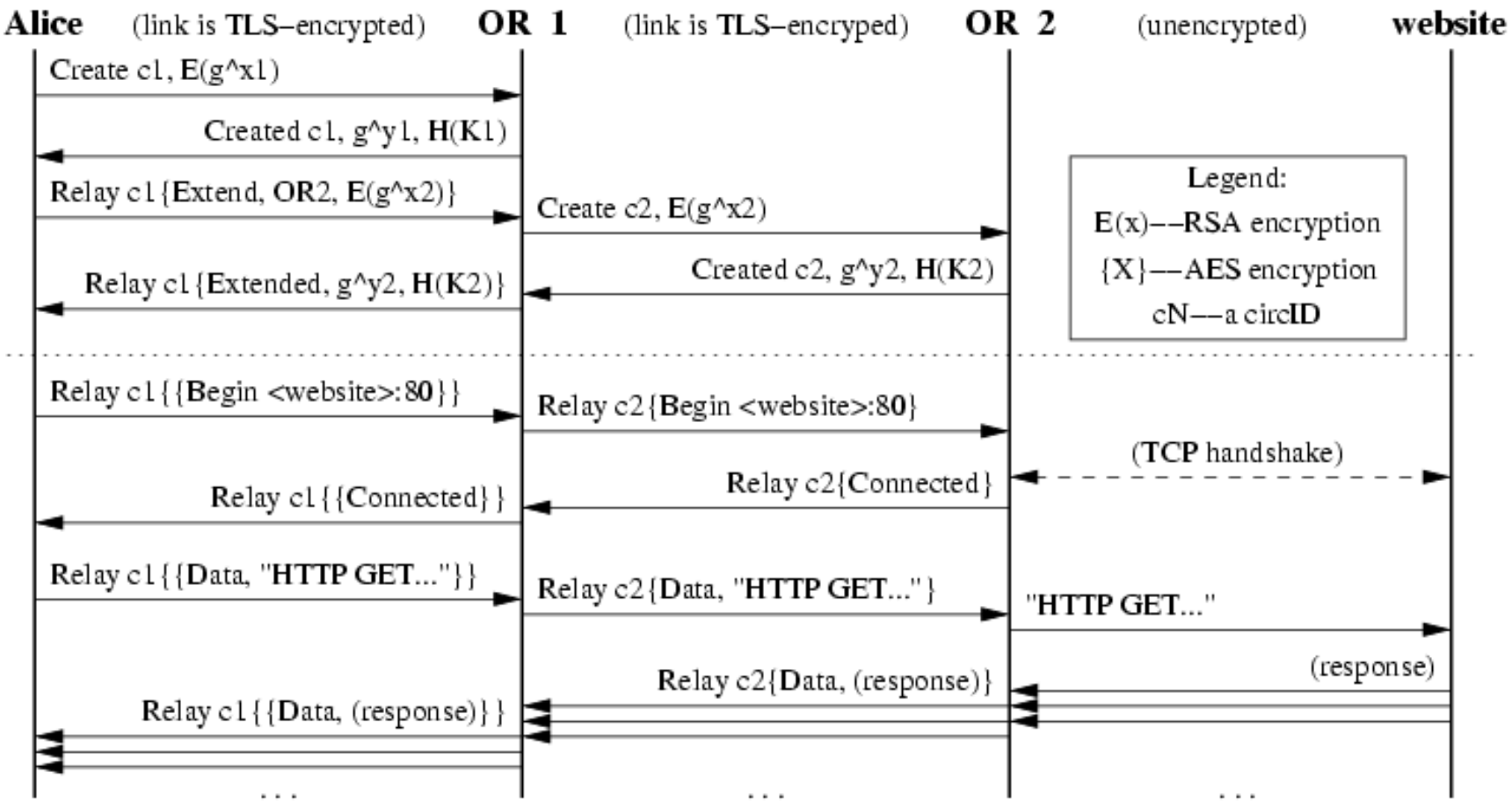
Data Sent by 5 to Target

Tor Optimization: Symmetric Key



- A circuit is built incrementally one hop by one hop
- Onion-like encryption
  - Alice negotiates an AES key with each router
  - Messages are divided into equal sized cells
  - Each router knows only its predecessor and successor
  - Only the Exit router (OR3) can see the message

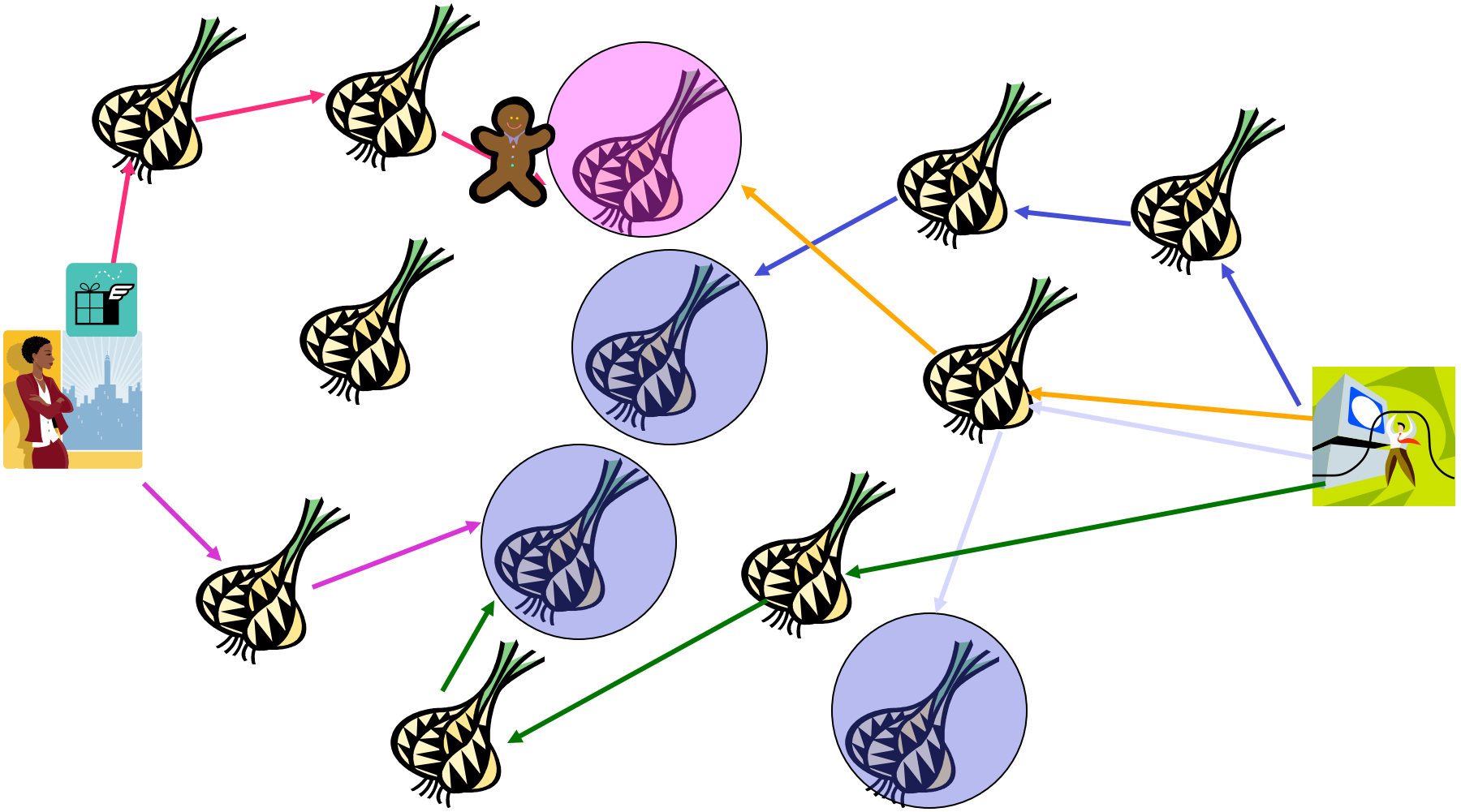
Example Tor Commands



Tor Location-Hidden Service

- Goal: allow one to offer a TCP service without revealing his IP address
  - Access Control: filtering incoming requests
  - Robustness: maintain a long-term pseudonymous identity
  - Smear-resistance: against socially disapproved acts

Tor Location-Hidden Service: Creating and connecting to a Location hidden service

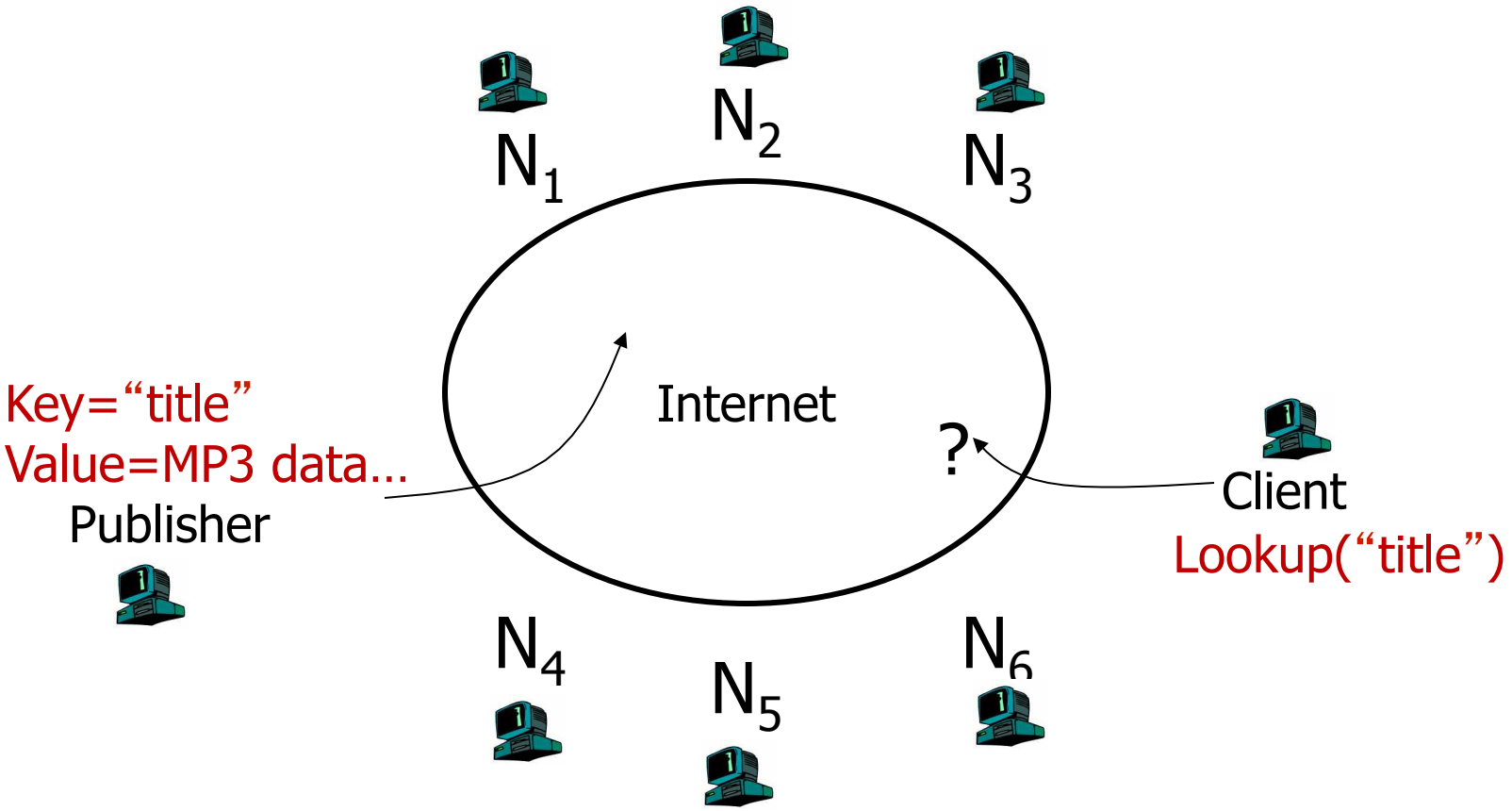


Optional Slides

Outline

- Admin and recap
- Multiple servers
- Application overlays
  - Overlays for scalability
  - Overlays for anonymity
  - Overlays for distributed content hosting

The Content Lookup Problem

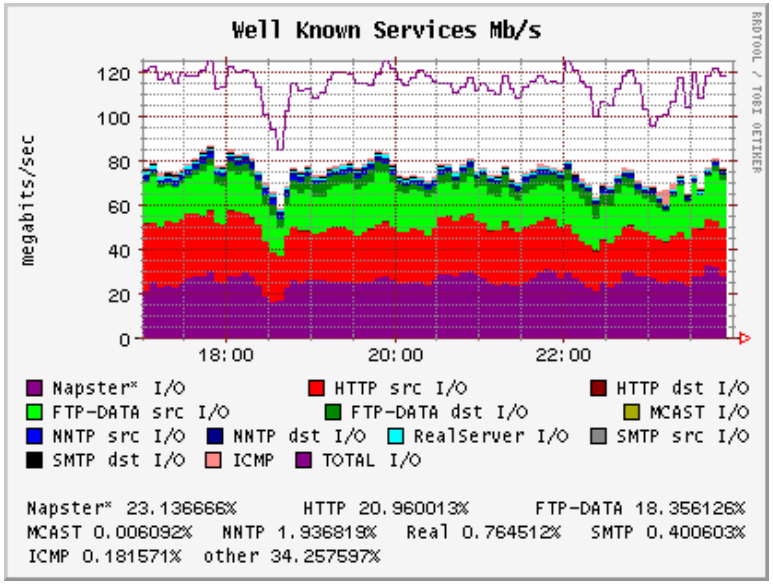


find where a particular file is stored

pay particular attention to see its equivalence of DNS

Napster

- Program for sharing music over the Internet
- History:
  - 5/99: Shawn Fanning (freshman, Northeasten U.) founded Napster Online music service, wrote the program in 60 hours
  - 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
  - 9/02: bankruptcy



03/2000

We are referring to the Napster before closure.

Napster: How Does it Work?

Application-level, client-server protocol over TCP

A centralized index system that maps files (songs) to machines that are alive and with files

Steps:

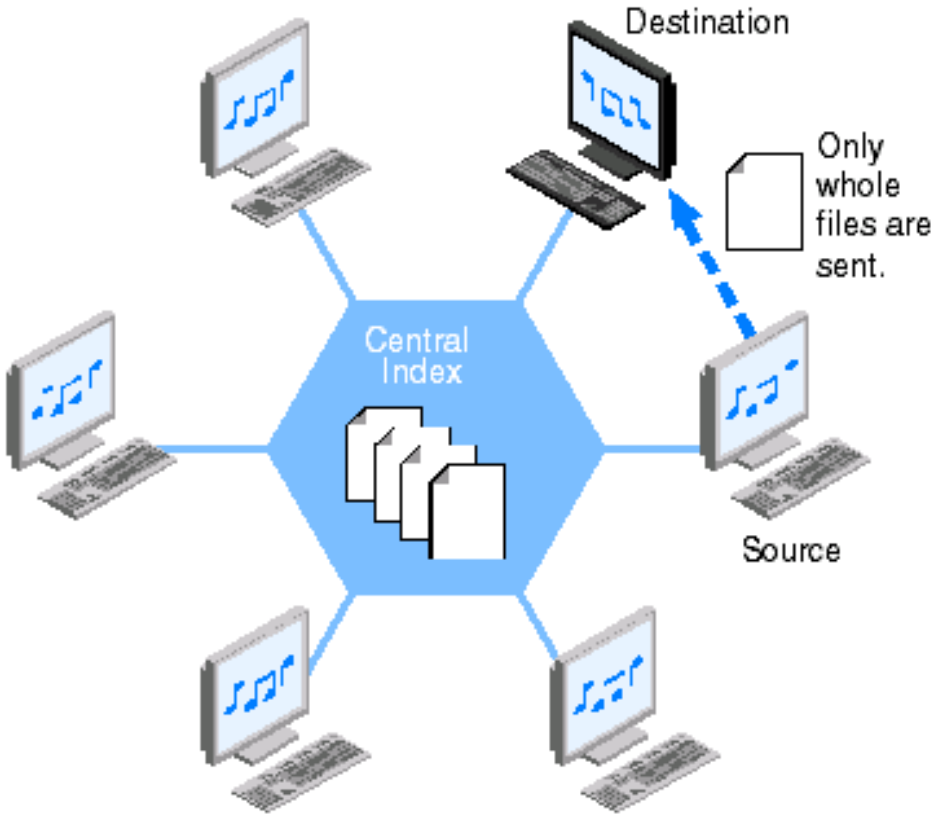
- Connect to Napster server
- Upload your list of files (push) to server
- Give server keywords to search the full list
- Select "best" of hosts with answers

Napster Architecture

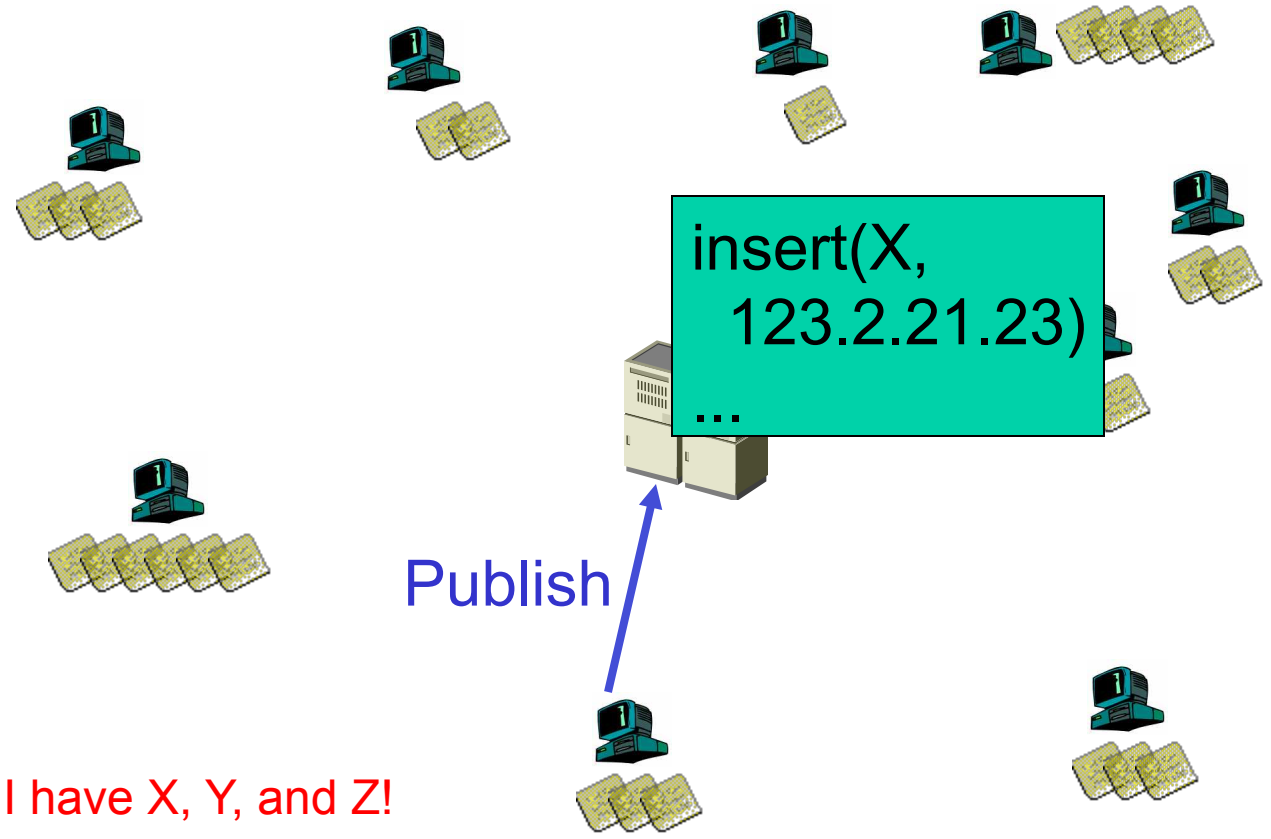
From Computer Desktop Encyclopedia
© 2004 The Computer Language Co. Inc.

THE ORIGINAL NAPSTER

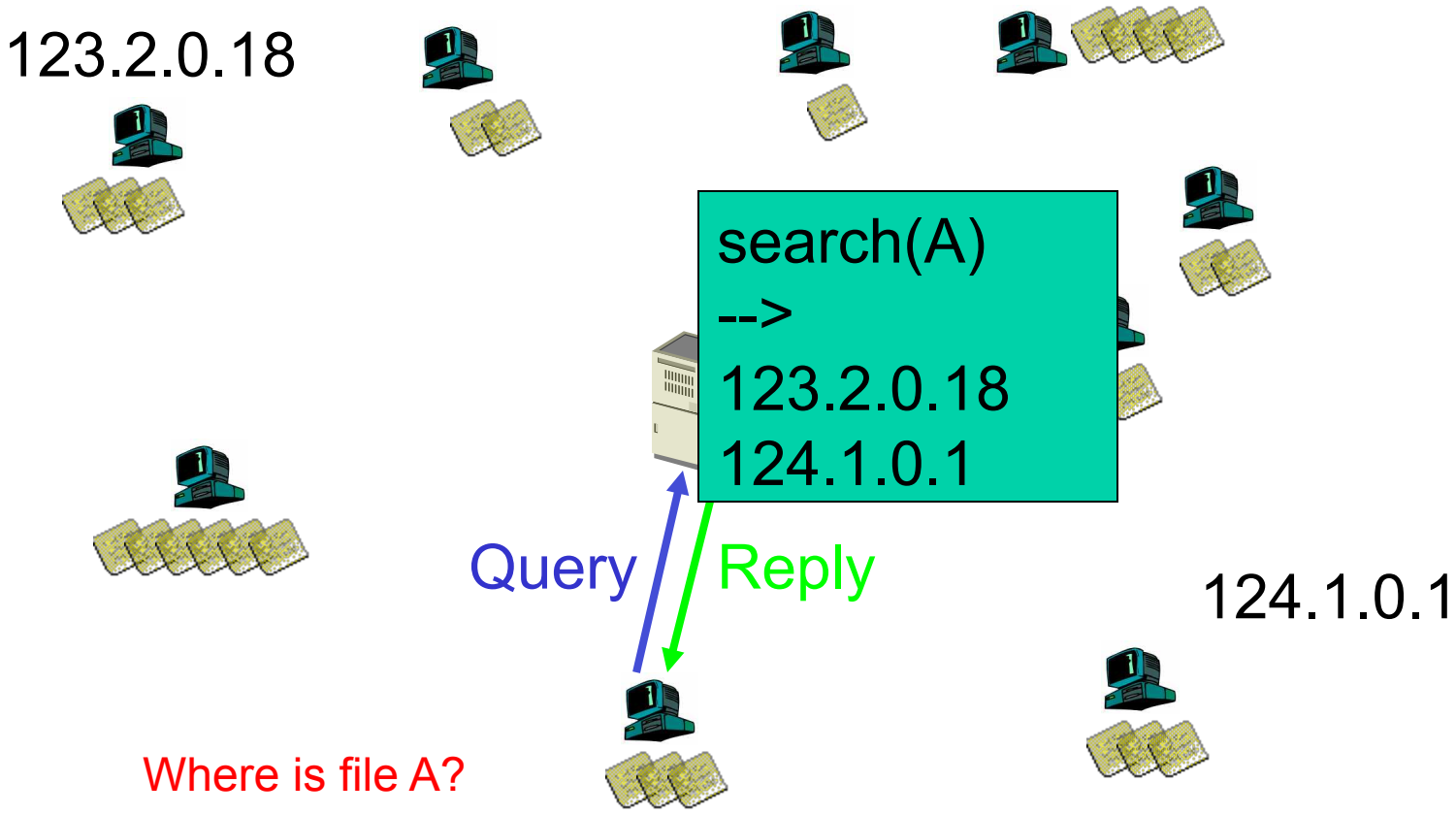
Napster provided a central directory of users who had files to share.



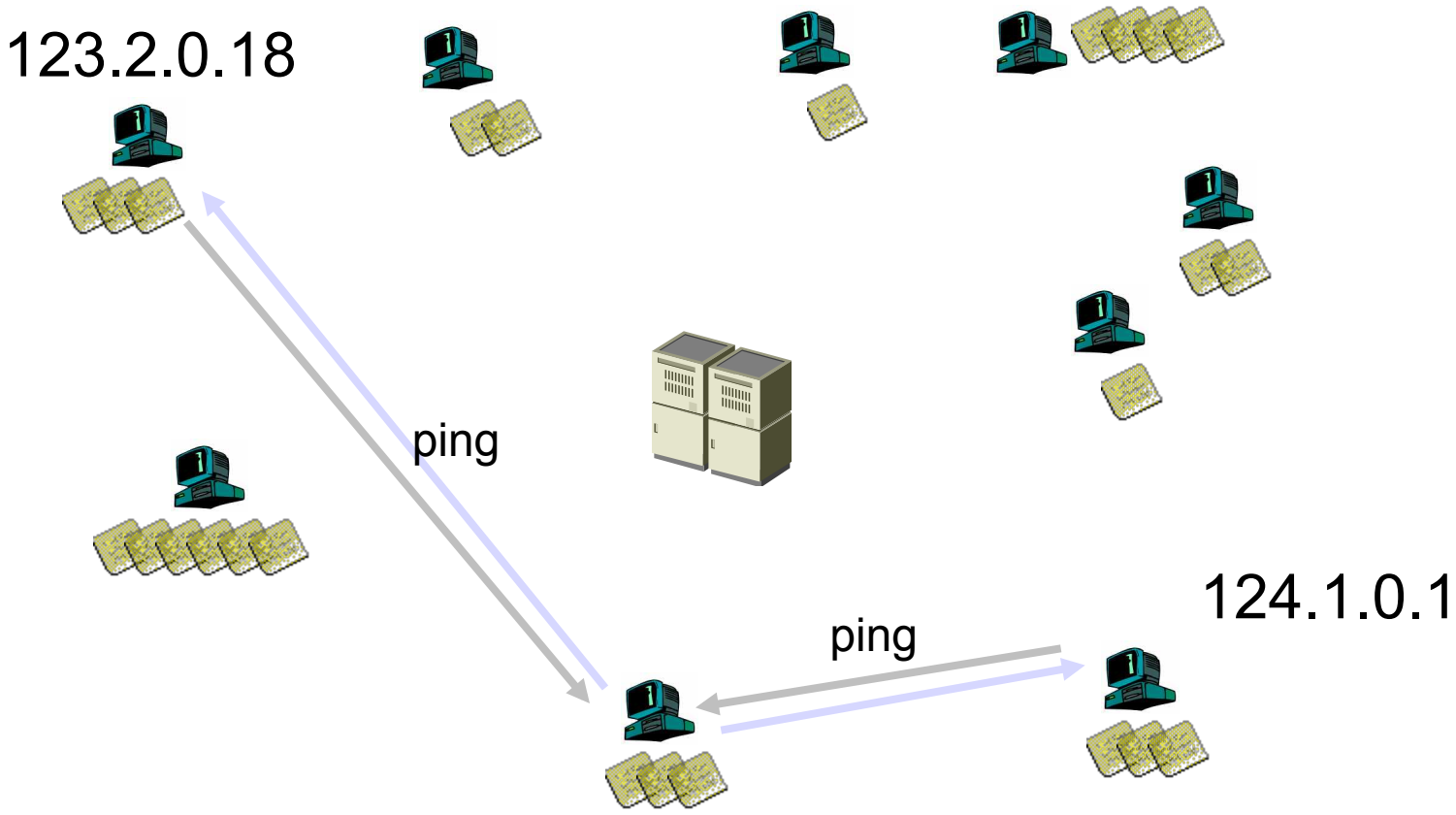
Napster: Publish



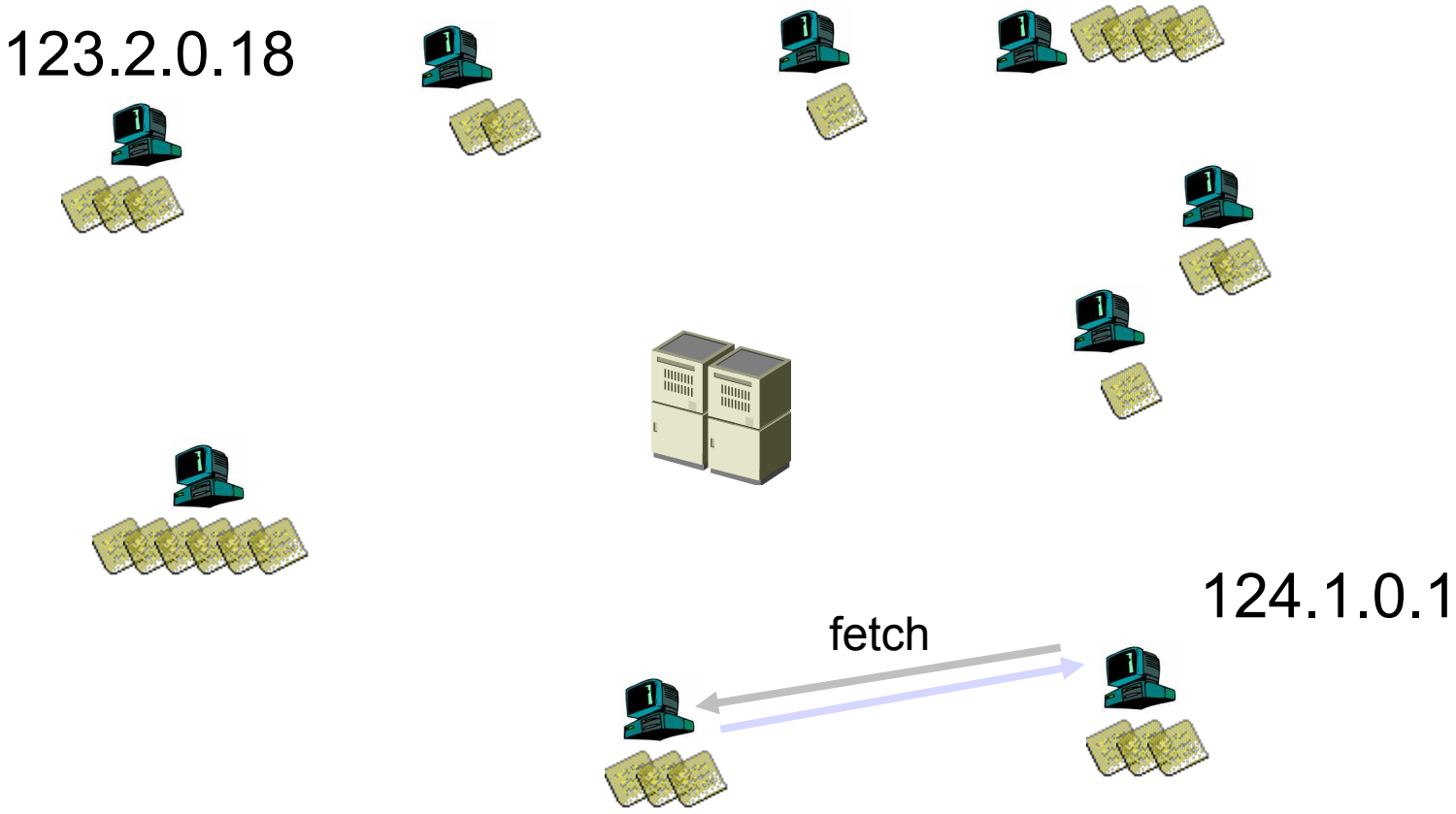
Napster: Search



Napster: Ping



Napster: Fetch



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
.....

Centralized Database: Napster

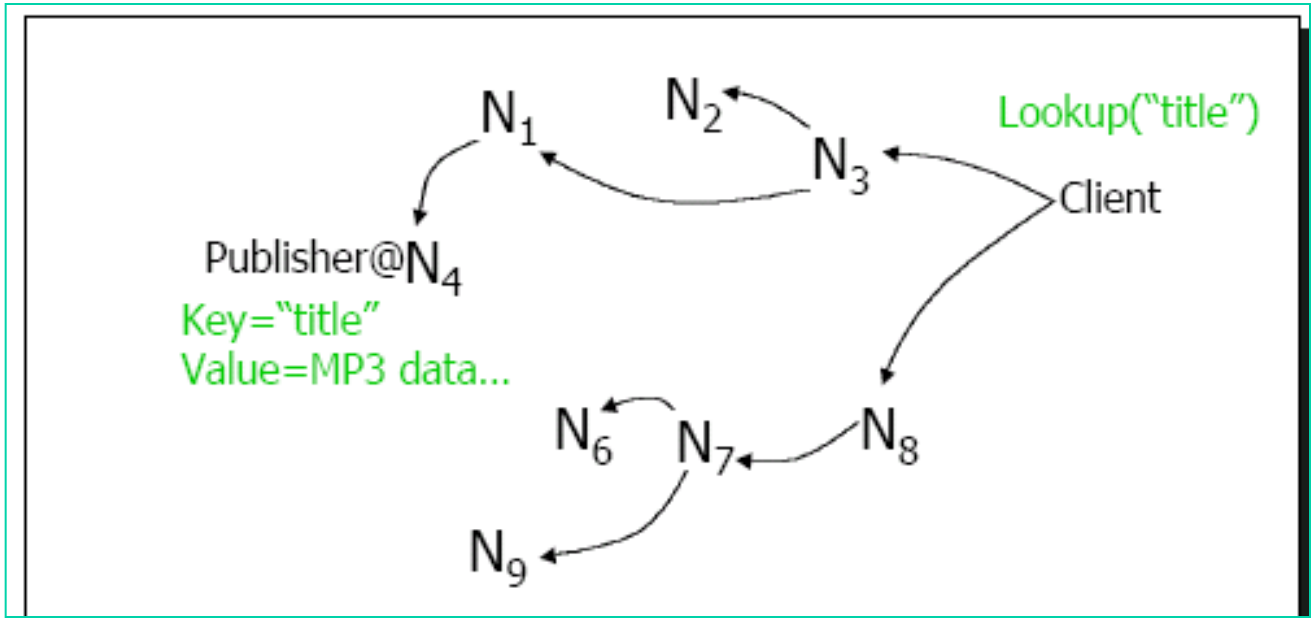
- Summary of features: a hybrid design
  - control: client-server (aka special DNS) for files
  - data: peer to peer
- Advantages
  - simplicity, easy to implement sophisticated search engines (boolean exp) on top of the index system
- Disadvantages
  - application specific (compared with DNS)
  - lack of robustness, scalability: central search server single point of bottleneck/failure
  - easy to take down !

Gnutella

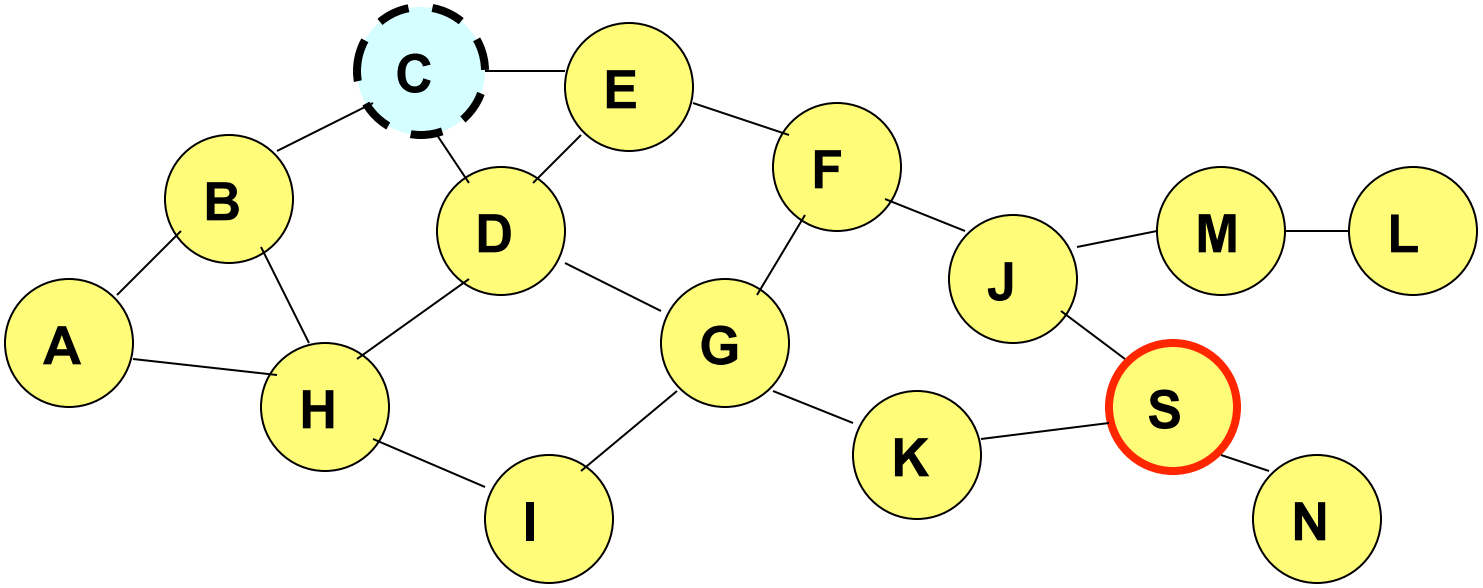
- On March 14th 2000, J. Frankel and T. Pepper from AOL's Nullsoft division (also the developers of the popular Winamp mp3 player) released Gnutella
- Within hours, AOL pulled the plug on it
- Quickly reverse-engineered and soon many other clients became available: Bearshare, Morpheus, LimeWire, etc.

Decentralized Flooding: Gnutella

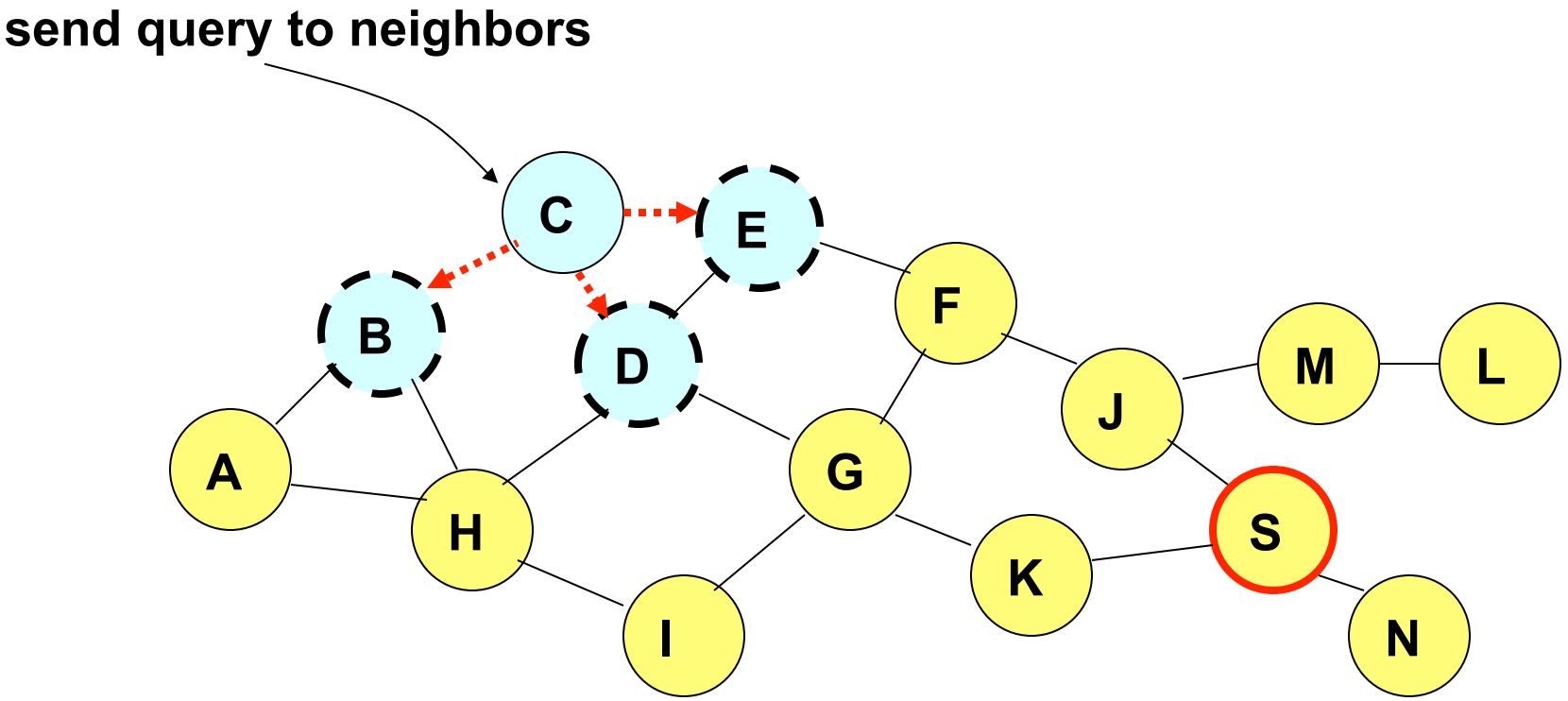
- On startup, client contacts other servents (**server** + cli**ent**) in network to form interconnection/peering relationships
  - servent interconnection used to forward control (queries, hits, etc)
- How to find a resource record: decentralized flooding
  - send requests to neighbors
  - neighbors recursively forward the requests



Decentralized Flooding

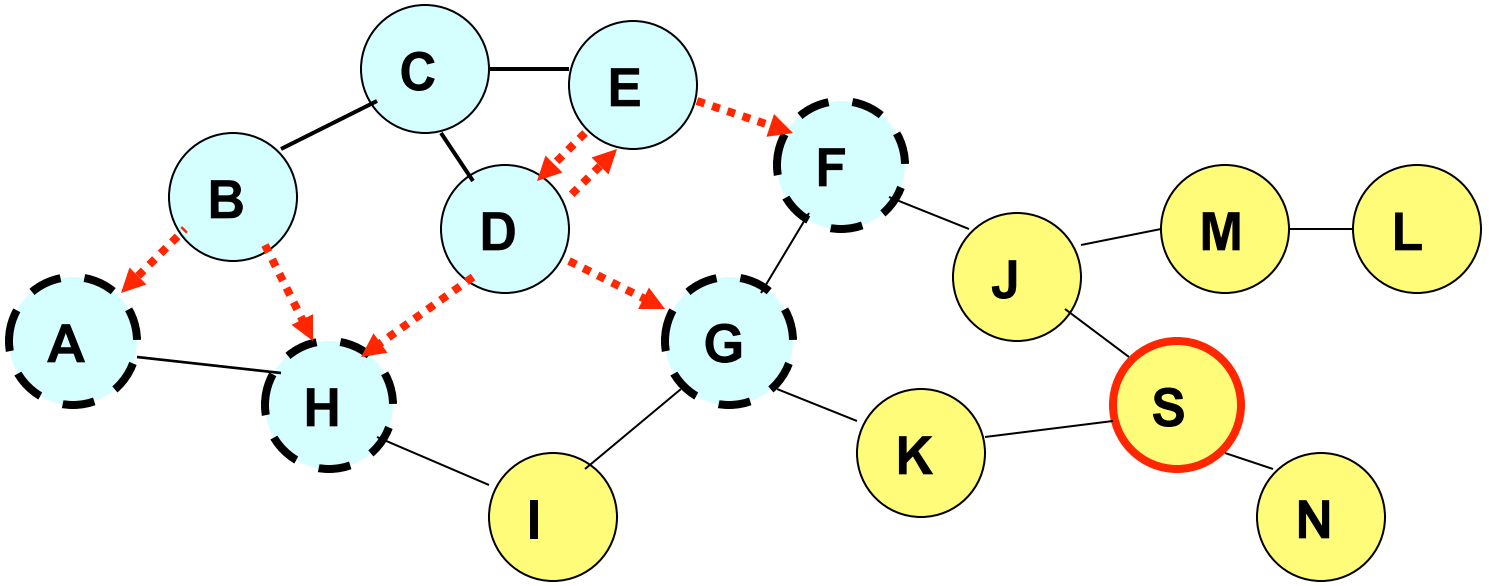


Decentralized Flooding



- Each node forwards the query to its neighbors other than the one who forwards it the query

Background: Decentralized Flooding



- Each node should keep track of forwarded queries to avoid loop !
  - node state: nodes keep state (which will time out---soft state)
  - packet state: carry the state in the query, i.e. carry a list of visited nodes

Decentralized Flooding: Gnutella

- Basic message header
  - Unique ID, TTL, Hops
- Message types
  - Ping – probes network for other servents
  - Pong – response to ping, contains IP addr, # of files, etc.
  - Query – search criteria + speed requirement of servent
  - QueryHit – successful response to Query, contains addr + port to transfer from, speed of servent, etc.
  - Ping, Queries are flooded
  - QueryHit, Pong: reverse path of previous message

Advantages and Disadvantages of Gnutella

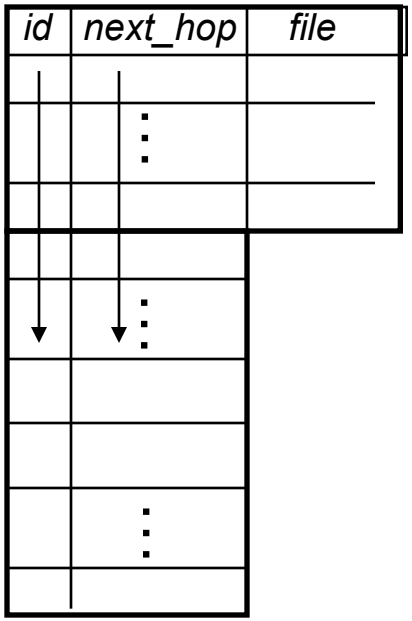
- Advantages:
  - totally decentralized, highly robust
- Disadvantages:
  - not scalable; the entire network can be swamped with flood requests
    - especially hard on slow clients; at some point broadcast traffic on Gnutella exceeded 56 kbps
  - to alleviate this problem, each request has a TTL to limit the scope
    - each query has an initial TTL, and each node forwarding it reduces it by one; if TTL reaches 0, the query is dropped (consequence?)

Freenet

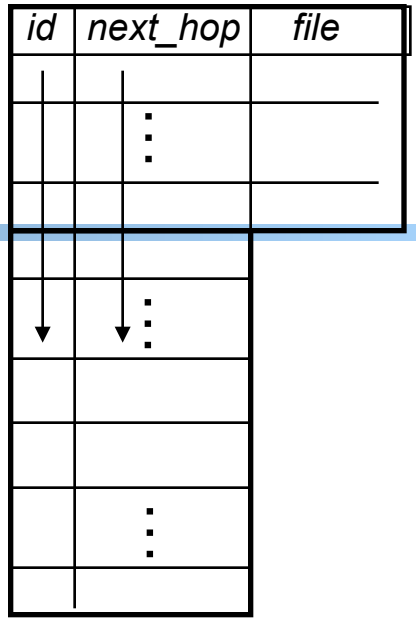
- History
  - final year project Ian Clarke , Edinburgh University, Scotland, June, 1999
- Goals:
  - totally distributed system without using centralized index or broadcast (flooding), instead search by routing
  - routing/storing system responds adaptively to usage patterns, transparently moving, replicating files as necessary to provide efficient service
  - provide publisher anonymity, security
  - free speech : resistant to attacks – a third party shouldn't be able to deny (e.g., deleting) the access to a particular file (data item, object)

Basic Structure of Freenet

- Each machine stores a set of files; each file is identified by a unique identifier (called key or id)
- Each node maintains a "routing table"
  - *id* – file id, key
  - *next_hop node* – where to search for a file with id' that is similar to id
  - *file* – local copy, if exists, of file with id

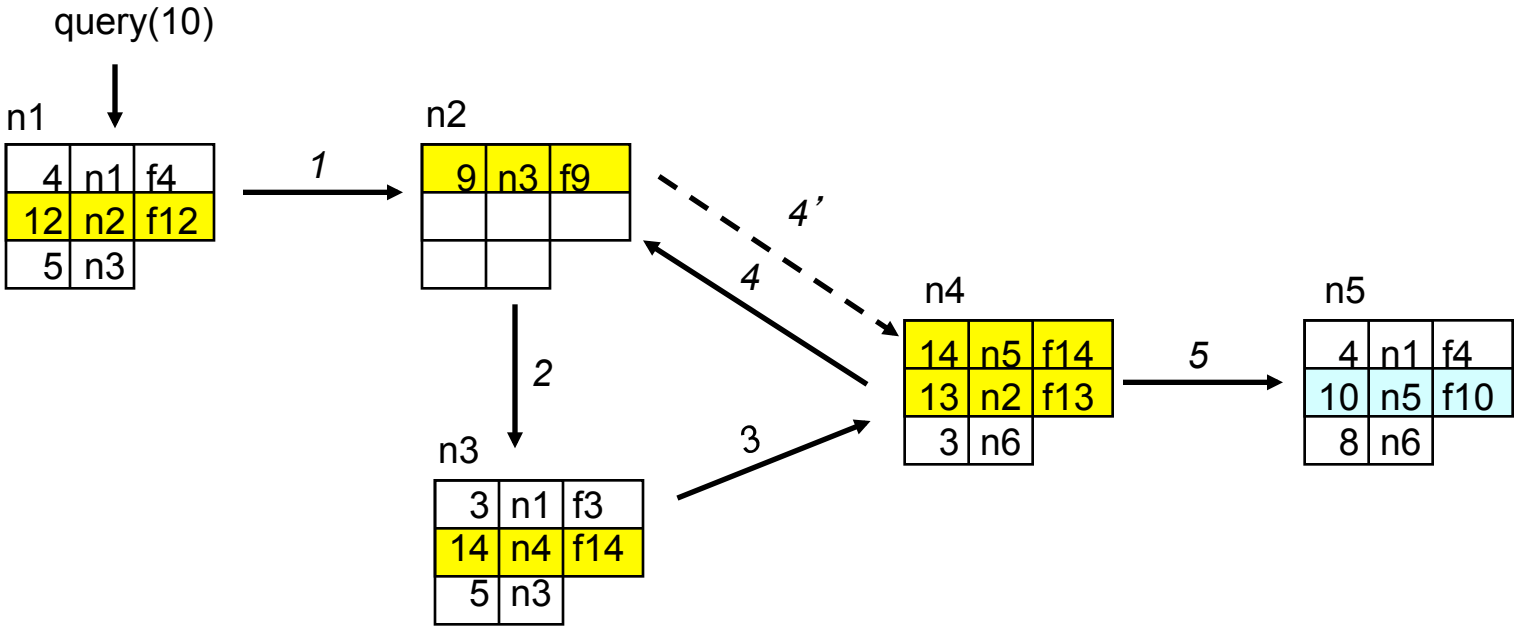


Freenet Query



- API: *file* = query(*id*);
- Upon receiving a query for file *id*
  - check whether the queried file is stored locally
  - check TTL to limit the search scope
    - each query is associated a TTL that is decremented each time the query message is forwarded
    - when TTL=1, the query is forwarded with a probability
    - TTL can be initiated to a random value (why random value?)
  - look for the "**closest**" *id* in the table with an unvisited next_hop node
    - if found one, forward the query to the corresponding *next_hop*
    - otherwise, backtrack
      - ends up performing a Depth First Search (DFS)-like traversal
      - search direction ordered by closeness to target
- When file is returned it is cached along the reverse path (any advantage?)

Query Example



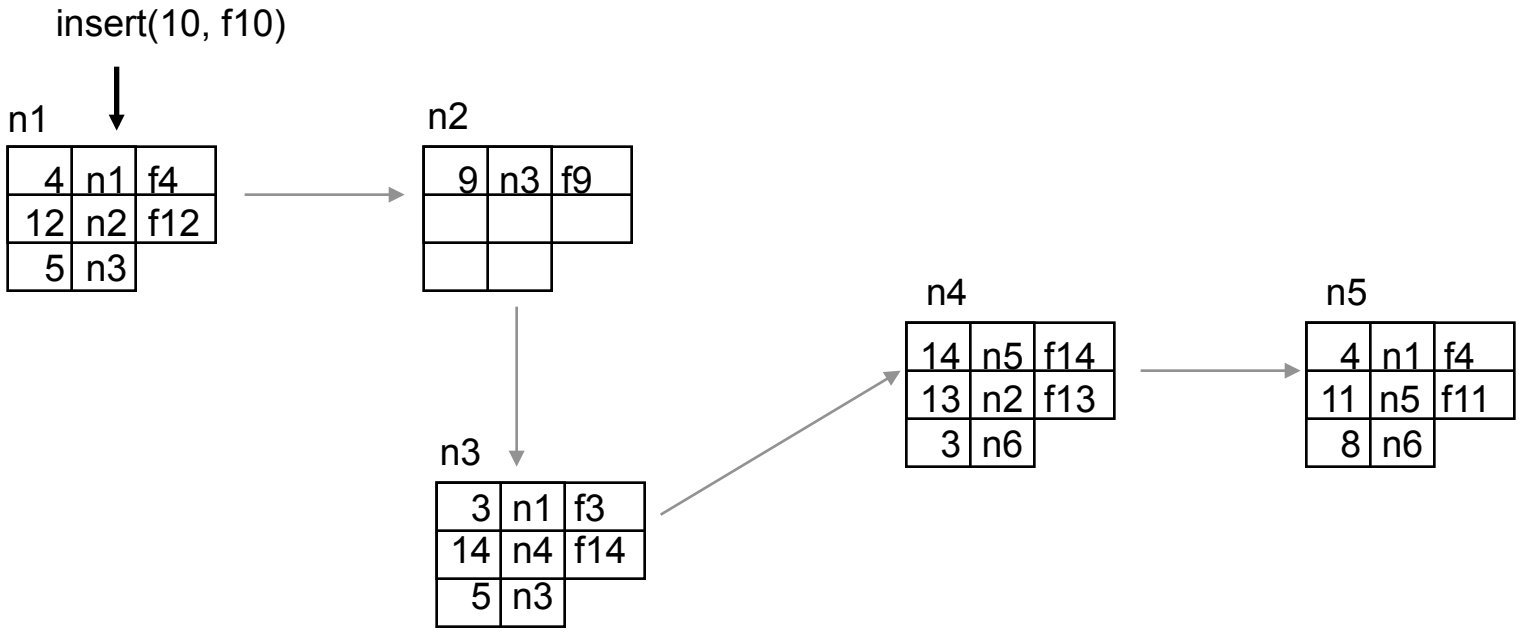
Beside the routing table, each node also maintains a query table containing the state of all outstanding queries that have traversed it → to backtrack

Insert

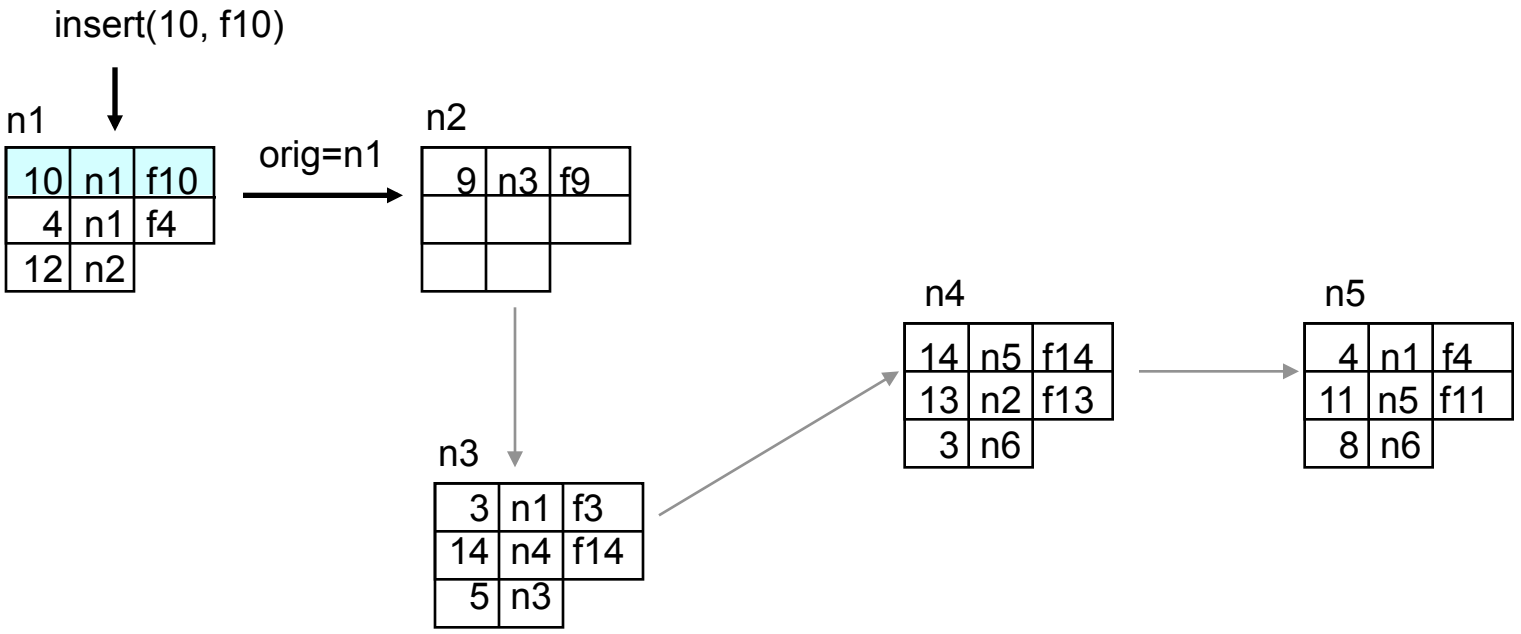
- API: insert(*id*, *file*);
- Two steps
  - first attempt a "search" for the file to be inserted
  - if found, report collision
  - if not found, insert the file by sending it along the query path (why?)
    - a node probabilistically replaces the originator with itself (why?)

Insert Example

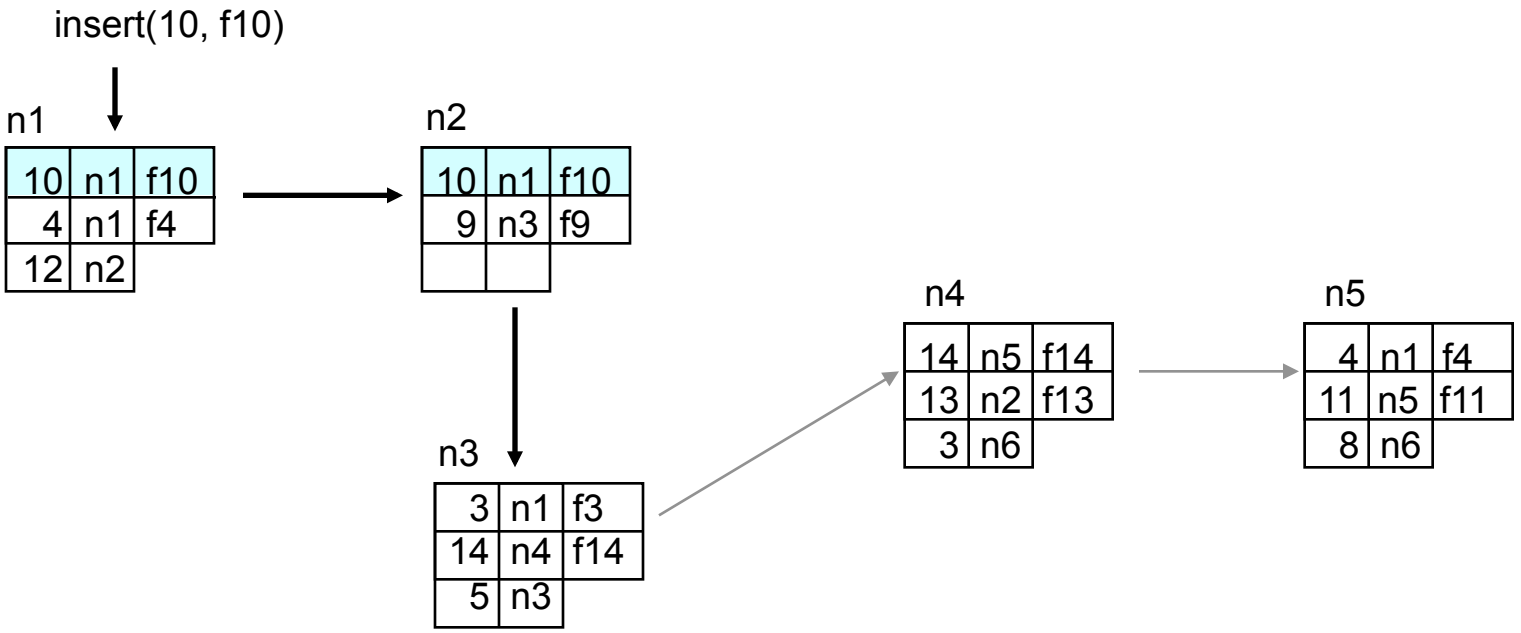
- Assume query returned failure along the shown path (backtrack slightly complicates things); insert f10



Insert Example

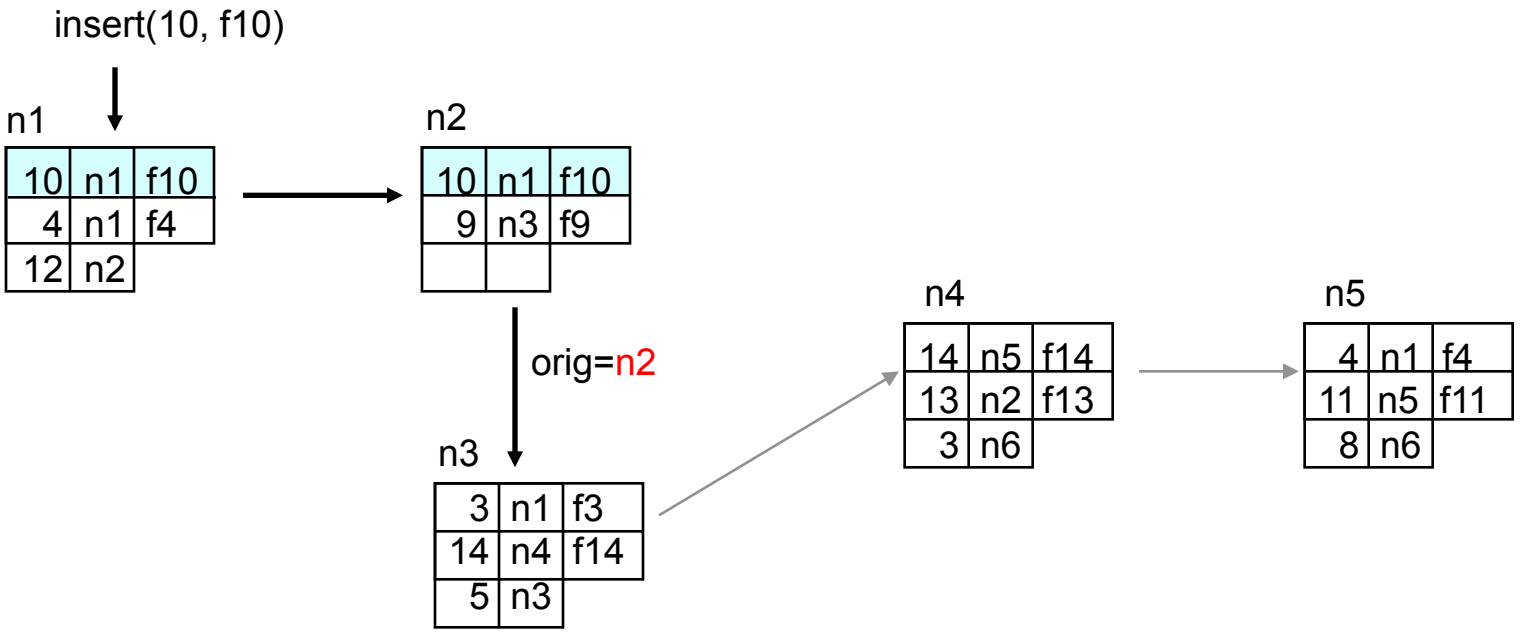


Insert Example

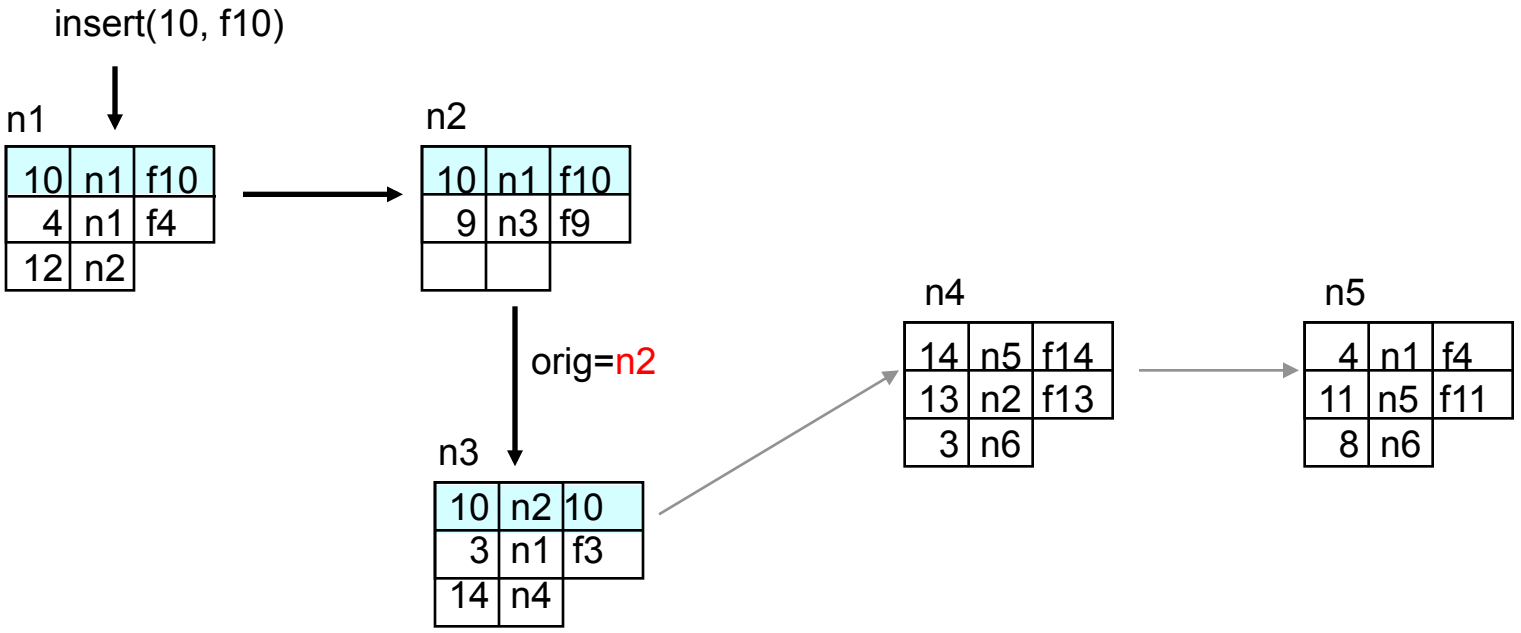


Insert Example

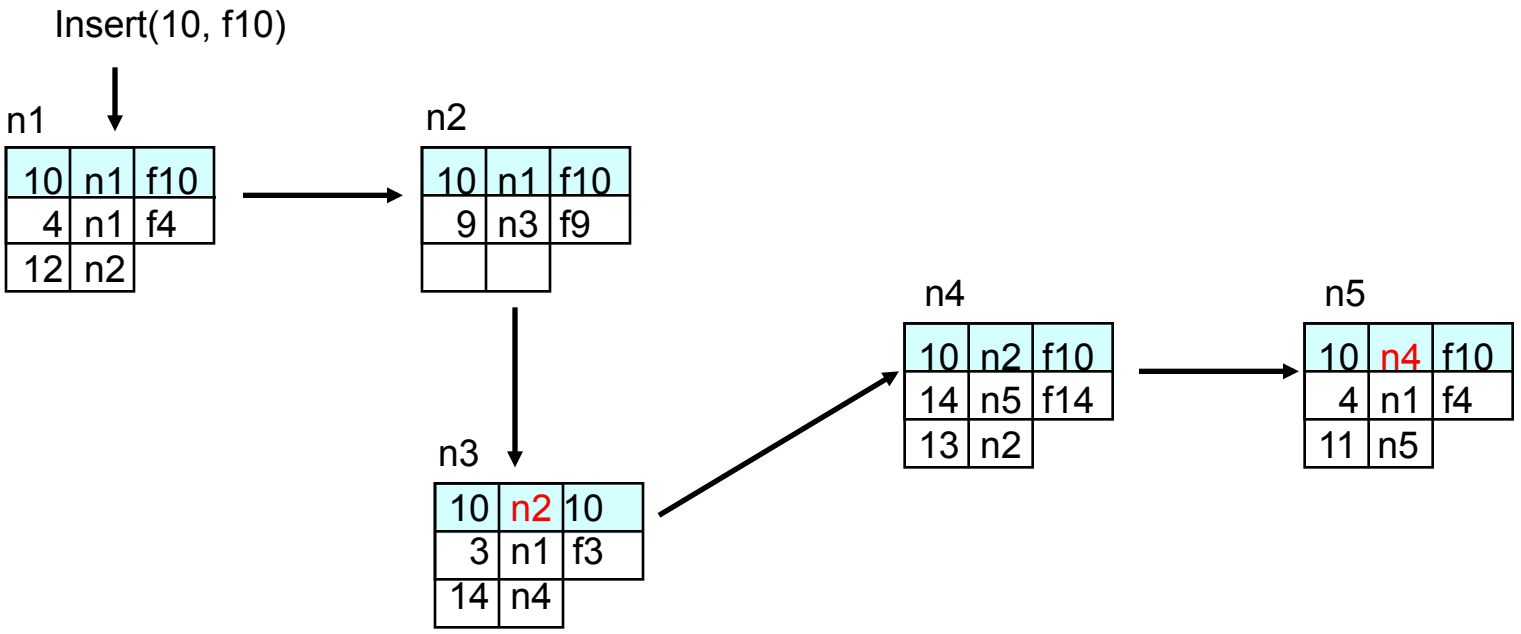
- n2 replaces the originator (n1) with itself



Insert Example



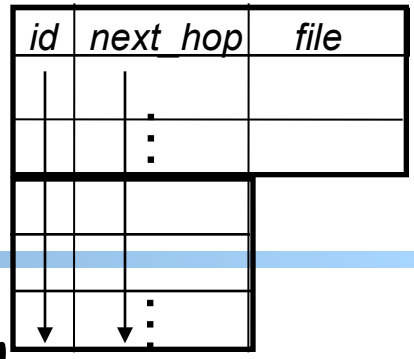
Insert Example



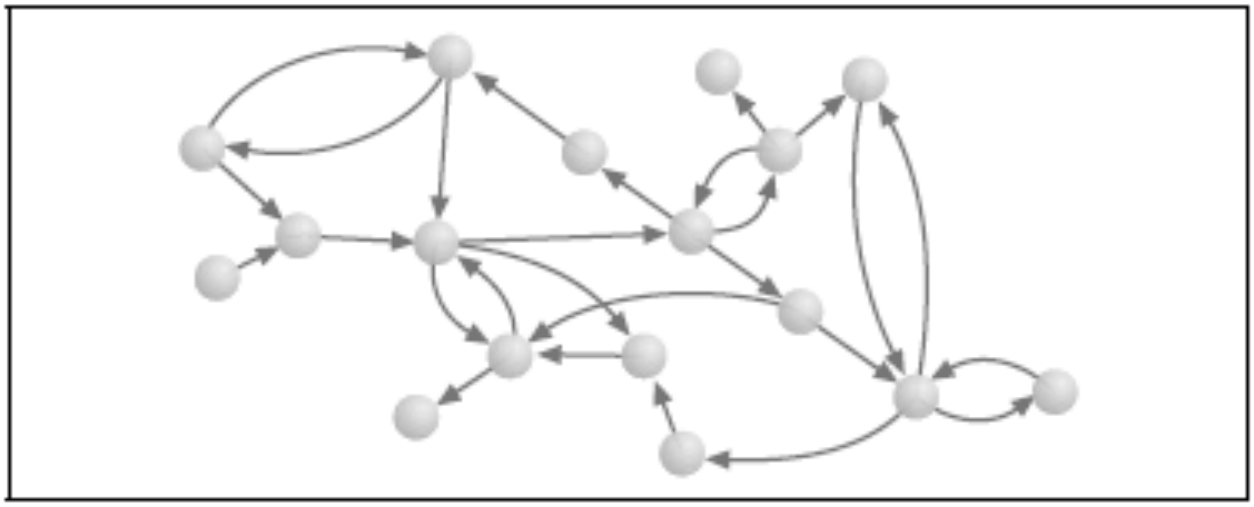
Freenet Analysis

- Authors claim the following effects:
  - nodes eventually specialize in locating similar keys
    - if a node is listed in a routing table, it will get queries for related keys
    - thus will gain "experience" answering those queries
  - popular data will be transparently replicated and will exist closer to requestors
  - as nodes process queries, connectivity increases
    - nodes will discover other nodes in the network
- Caveat: lexigraphic closeness of file names/keys may not imply content similarity

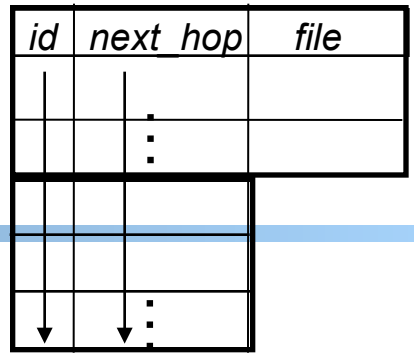
Understanding Freenet Self-Organization: Freenet Graph

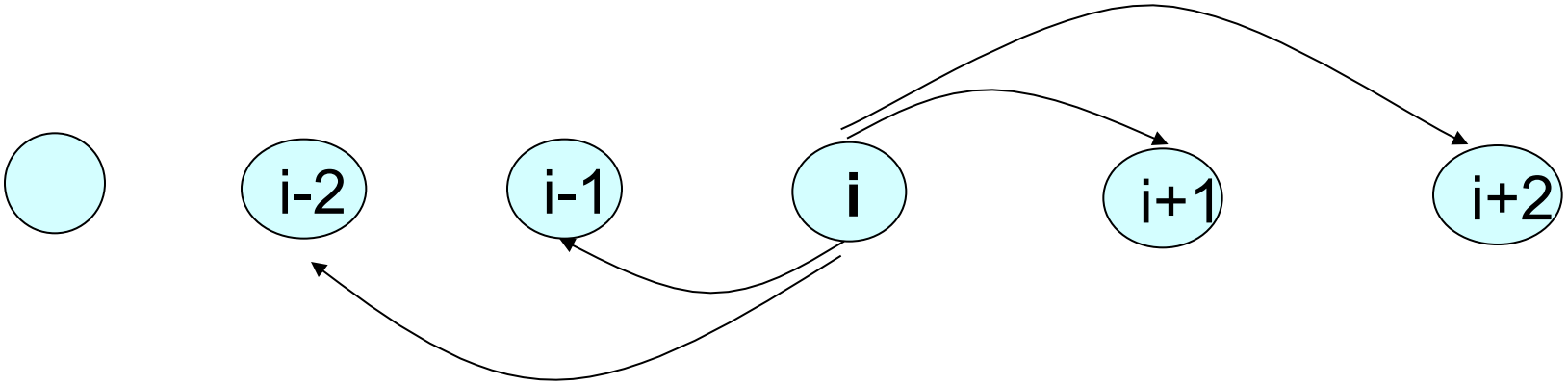


- We create a Freenet reference graph
  - creating a vertex for each Freenet node
  - adding a directed link from A to B if A refers to an item stored at B



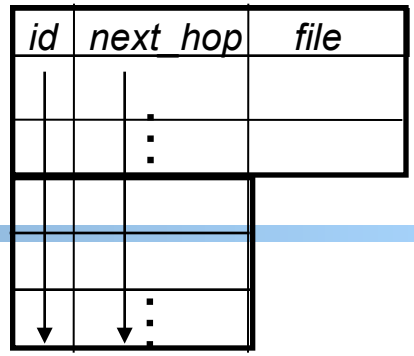
Experiment: Freenet Graph: Init

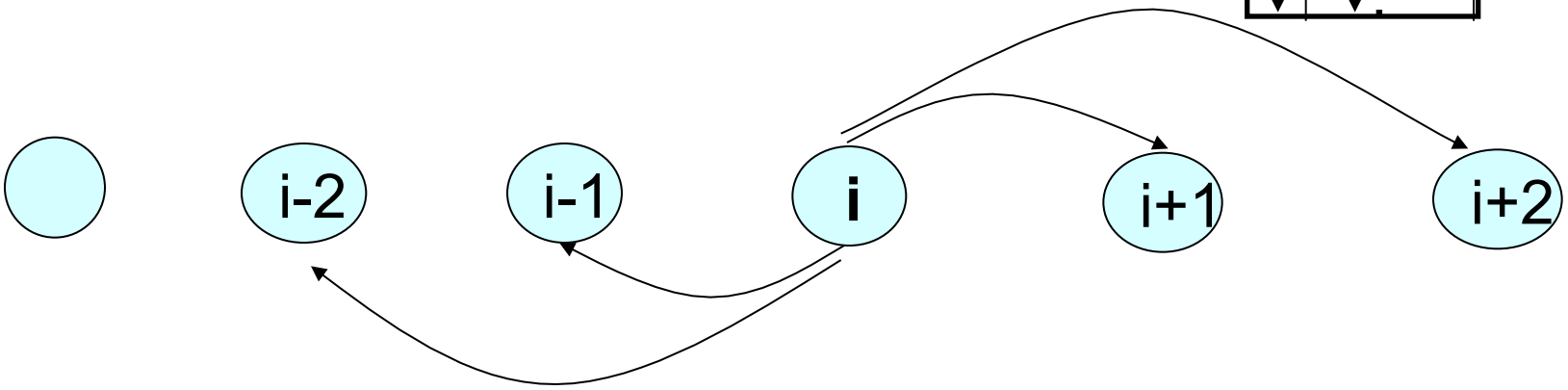




- Assume a network of 1000 nodes, with node id 0 to 999
- Each node can store 50 data items, and 200 references
- Assume initially each node i has i, and knows the storage of i – 2, -1, i + 1, i + 2 (all mod 1000)
- thus a regular, locally-clustered graph with avg path length ~ 1000 / 8 = 125

Example: Search Effect

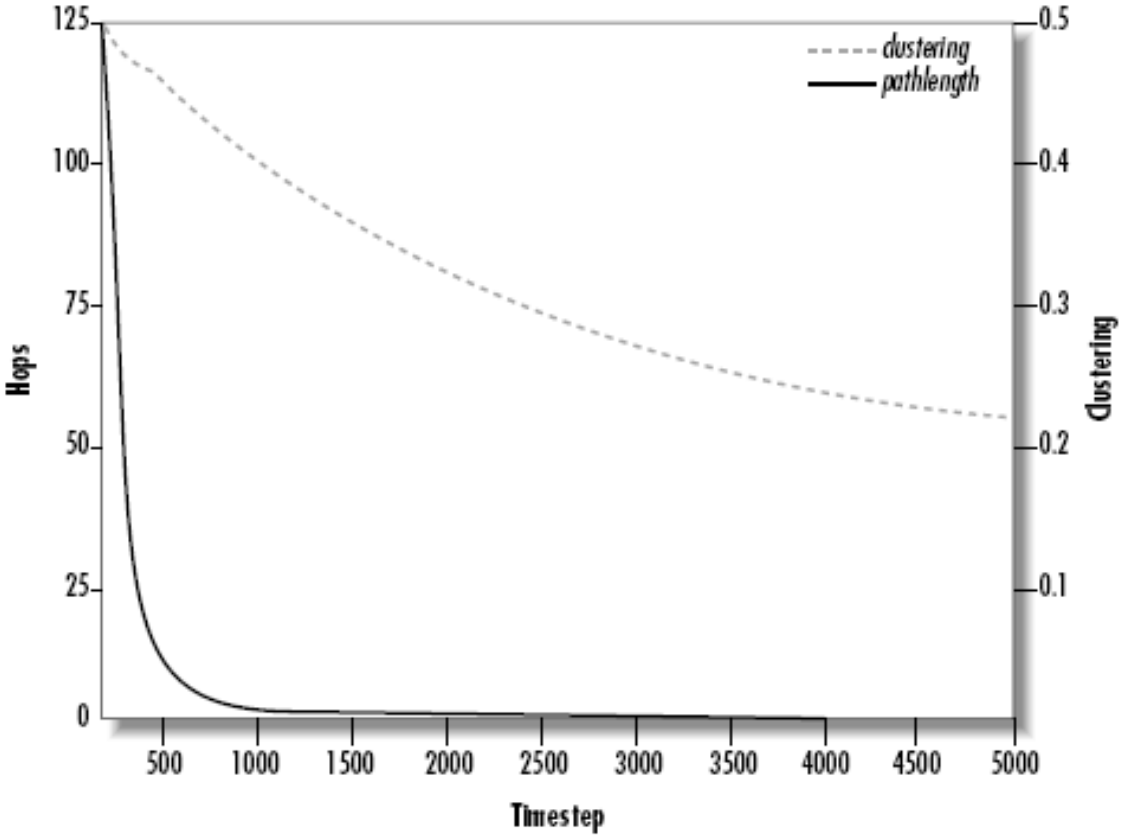




- What is the effect that if the first search is node 0 searching for item 490 (assume no probabilistic replacement to hide origin)?
  - Nodes 0, 2, 4, 6, …, 488 all cache item 490, and has a pointer to node 490
  - The search forms many long-distance links

Experiment: Evolution of Freenet Graph

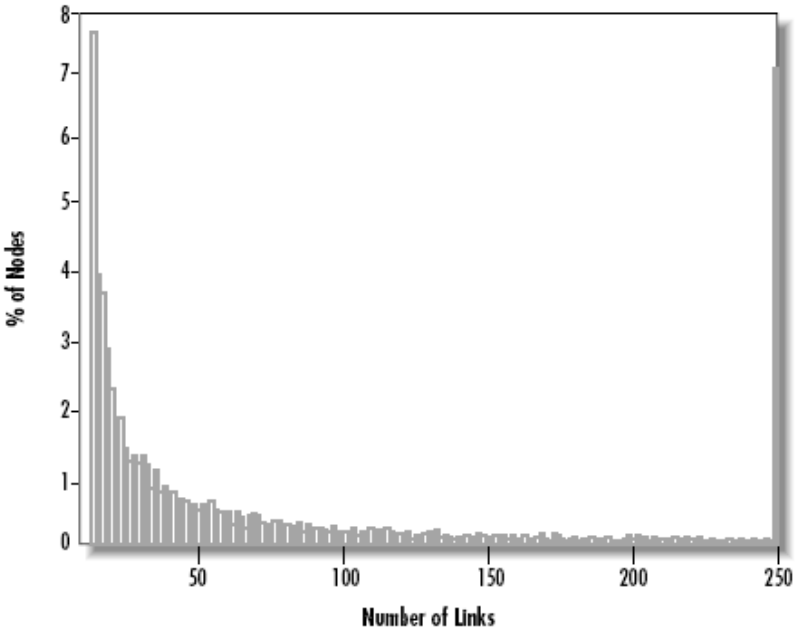
- At each step
  - pick a node randomly
  - flip a coin to determine search or insert
    - if search, randomly pick a key in the network
    - if insert, pick a random key

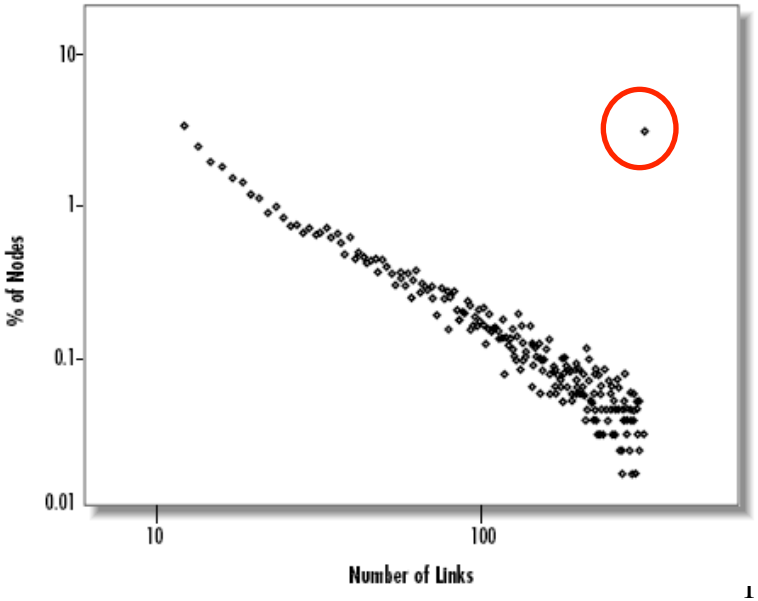


Evolution of path length and clustering; Clustering is defined as percentage of local links

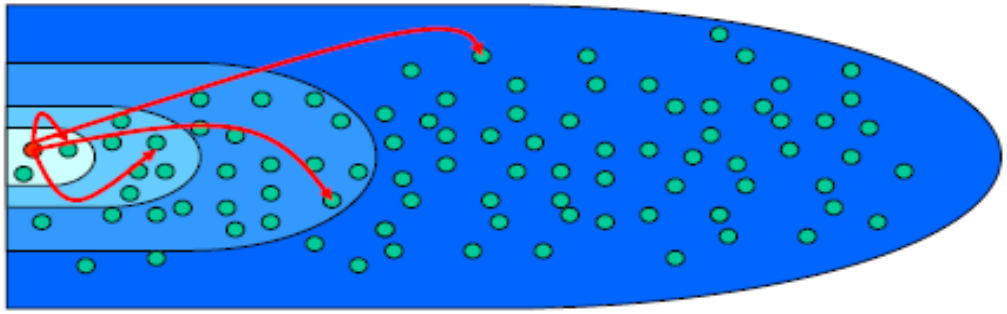
Freenet Evolves to Small-World Network

- With usage, the regular, highly localized Freenet network evolved into one irregular graph
- High percentage of highly connected nodes provide shortcuts/bridges
  - make the world a "small world"
  - most queries only traverse a small number of hops to find the file





Small-World



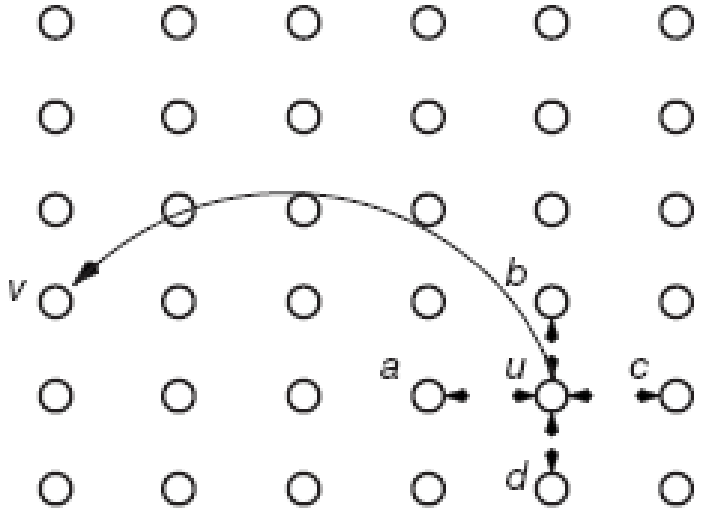
- First discovered by Milgrom
  - in 1967, Milgram mailed 160 letters to a set of randomly chosen people in Omaha, Nebraska
  - goal: pass the letters to a given person in Boston
    - each person can only pass the letter to an intermediary known on a first-name basis
    - pick the person who may make the best progress
  - result: 42 letters made it through !
  - median intermediaries was 5.5---thus six degree of separation
  - a potential explanation: highly connected people with non-local links in mostly locally connected communities improve search performance !

The Computer Networks Distributed Search Question

- Question: what kind of long distance links to maintain so that distributed network search is effective?
- Assume that each node has
  - a fixed # (say p distance away) local links
  - a small # (say a total of q) long-distance links s.t. the probability of a link between nodes x and y is some (α) inverse-power of the distance d(x, y) of x and y

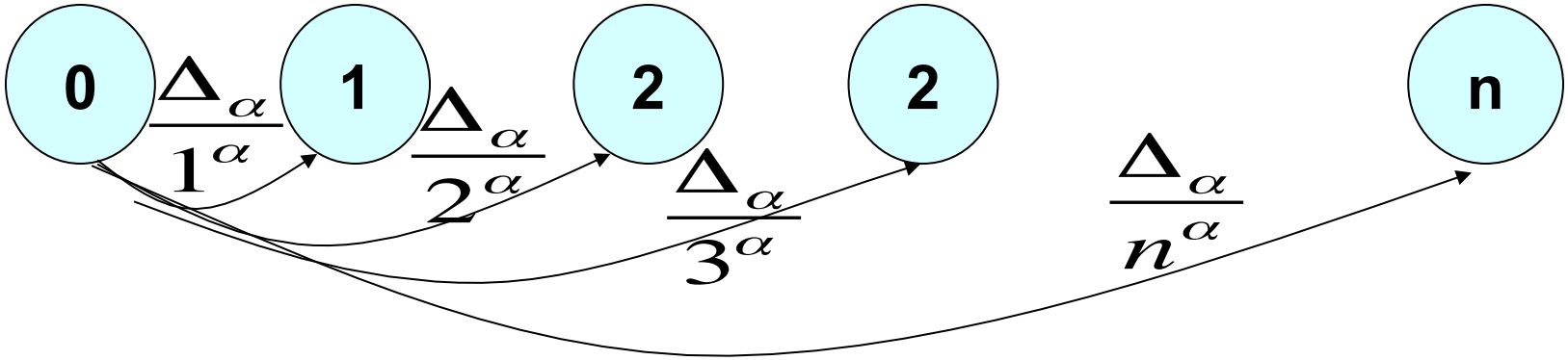
$$\frac{d(x,y)^{-\alpha}}{\Delta_\alpha}$$

  - Different alpha's give diff types of links.
  - Q: what is a good alpha?



What Should the Long-Distance Links Look?

- Consider the simple case of one dimensional space.



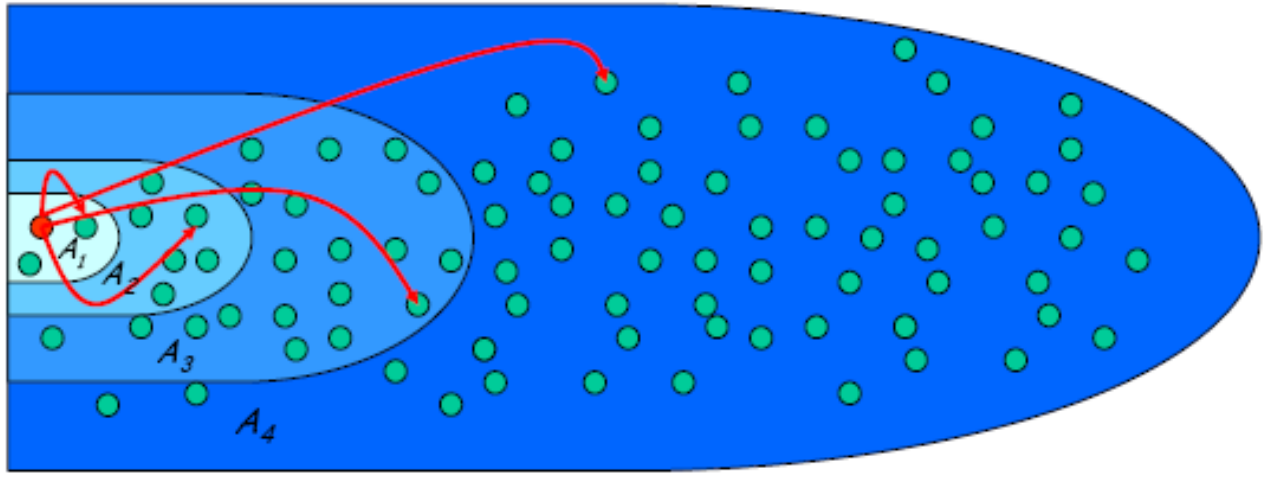
- Which alpha leads to best performing distributed search alg?

What Should the Long-Distance Links Look?

- For 1-d space, for any distributed algorithm, the expected # of search steps, for different α's:
  - $0 \leq \alpha < 1$: $\geq k_1 n^{(1-\alpha)/2}$
  - $\alpha > 1$: $\geq k_1 n^{(\alpha-1)/\alpha}$
  - $\alpha = 1$: $O(\log^2 n)$ greedy search

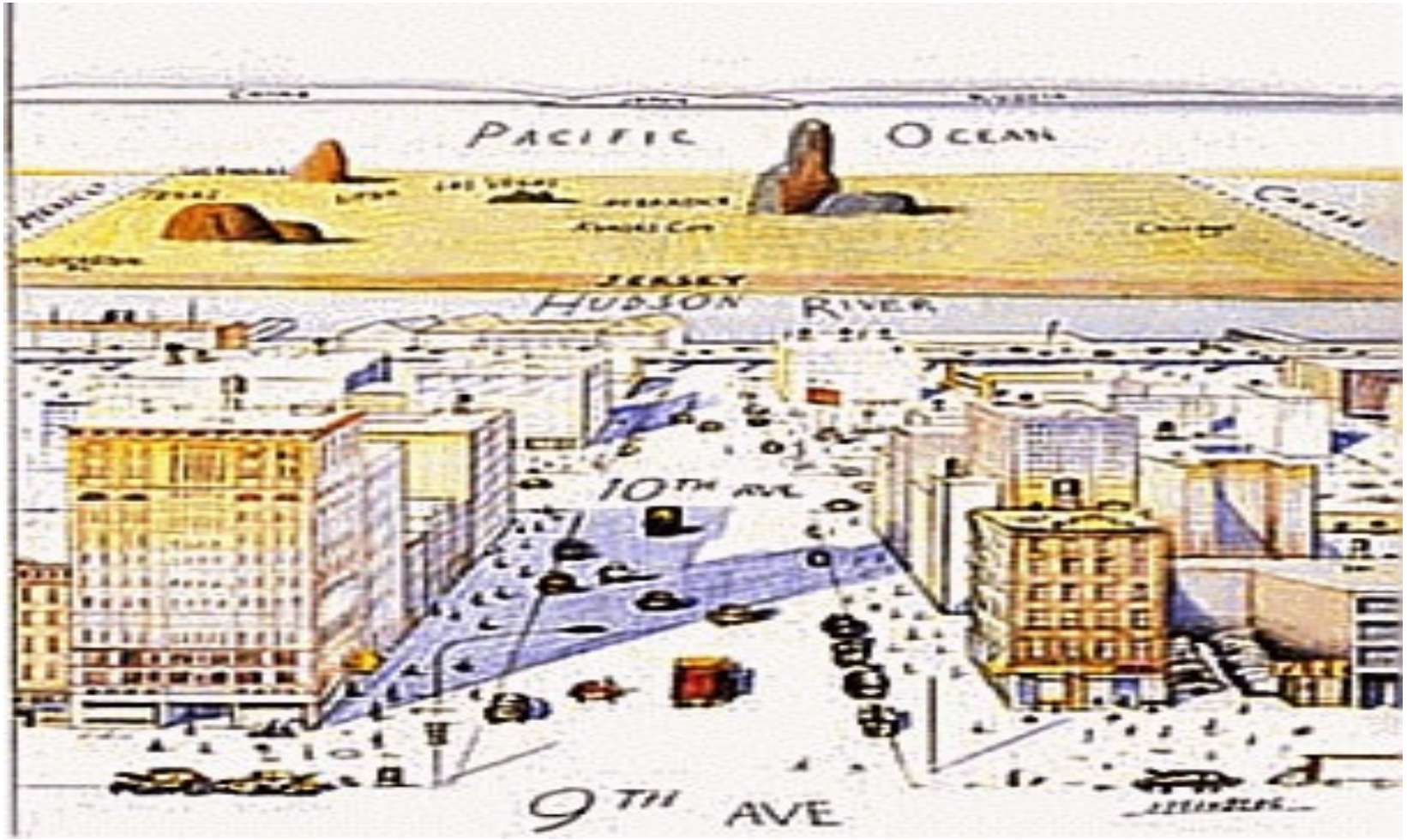
Distributed Search

- In the general case, α should be the dimension
- In other words, a guideline on long-distance links: roughly the same number of nodes in each region A1, A2, A4, A8, where A1 is the set of nodes who are one lattice step away, A2 is those two steps away, A4 is four steps away…



probability is proportional to (lattice steps)$^{-d}$

Small World



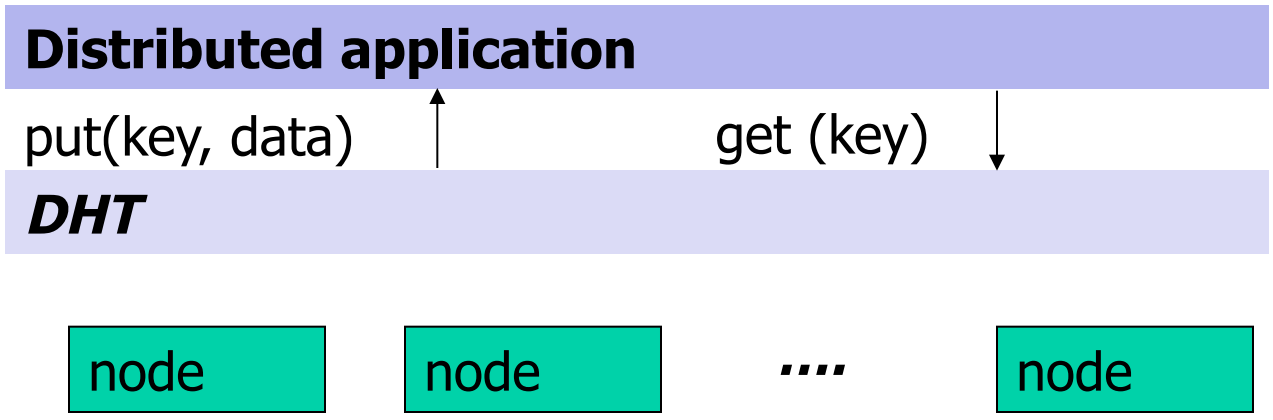
Saul Steinberg; View of World from 9th Ave

Freenet: Issues

- Does not always guarantee that a file is found, even if the file is in the network
- Good average-case performance, but a potentially long search path in a large network
  - approaching small-world…

DHT: Overview

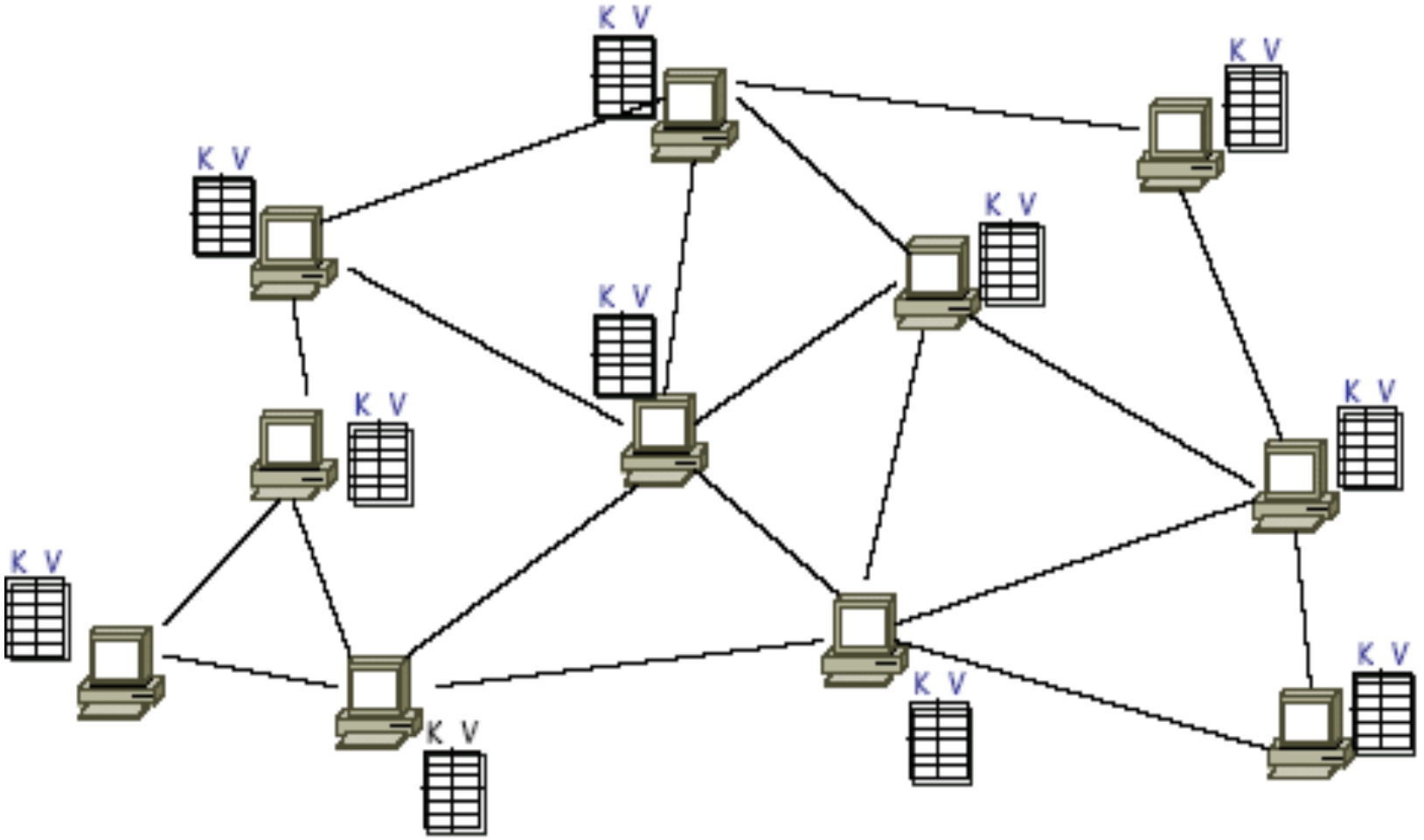
- Abstraction: a distributed "hash-table" (DHT) data structure
  - put(key, value) and get(key) → value
  - DHT imposes no structure/meaning on keys
  - one can build complex data structures using DHT
- Implementation:
  - nodes in system form an interconnection network: ring, zone, tree, hypercube, butterfly network, ...



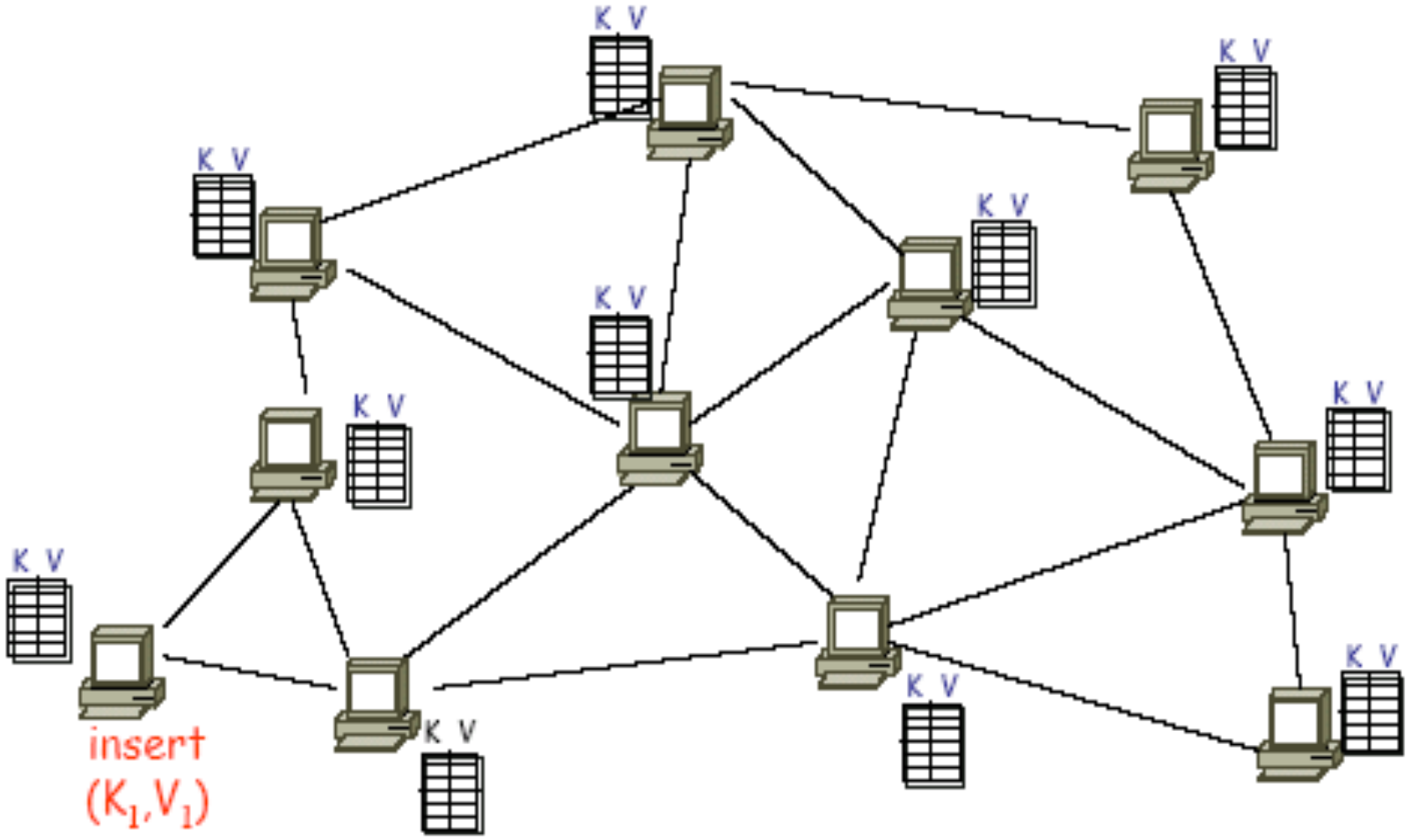
DHT Applications

- File sharing and backup [CFS, Ivy, OceanStore, PAST, Pastiche …]
- Web cache and replica [Squirrel, Croquet Media Player]
- Censor-resistant stores [Eternity]
- DB query and indexing [PIER, Place Lab, VPN Index]
- Event notification [Scribe]
- Naming systems [ChordDNS, Twine, INS, HIP]
- Communication primitives [I3, …]
- Host mobility [DTN Tetherless Architecture]

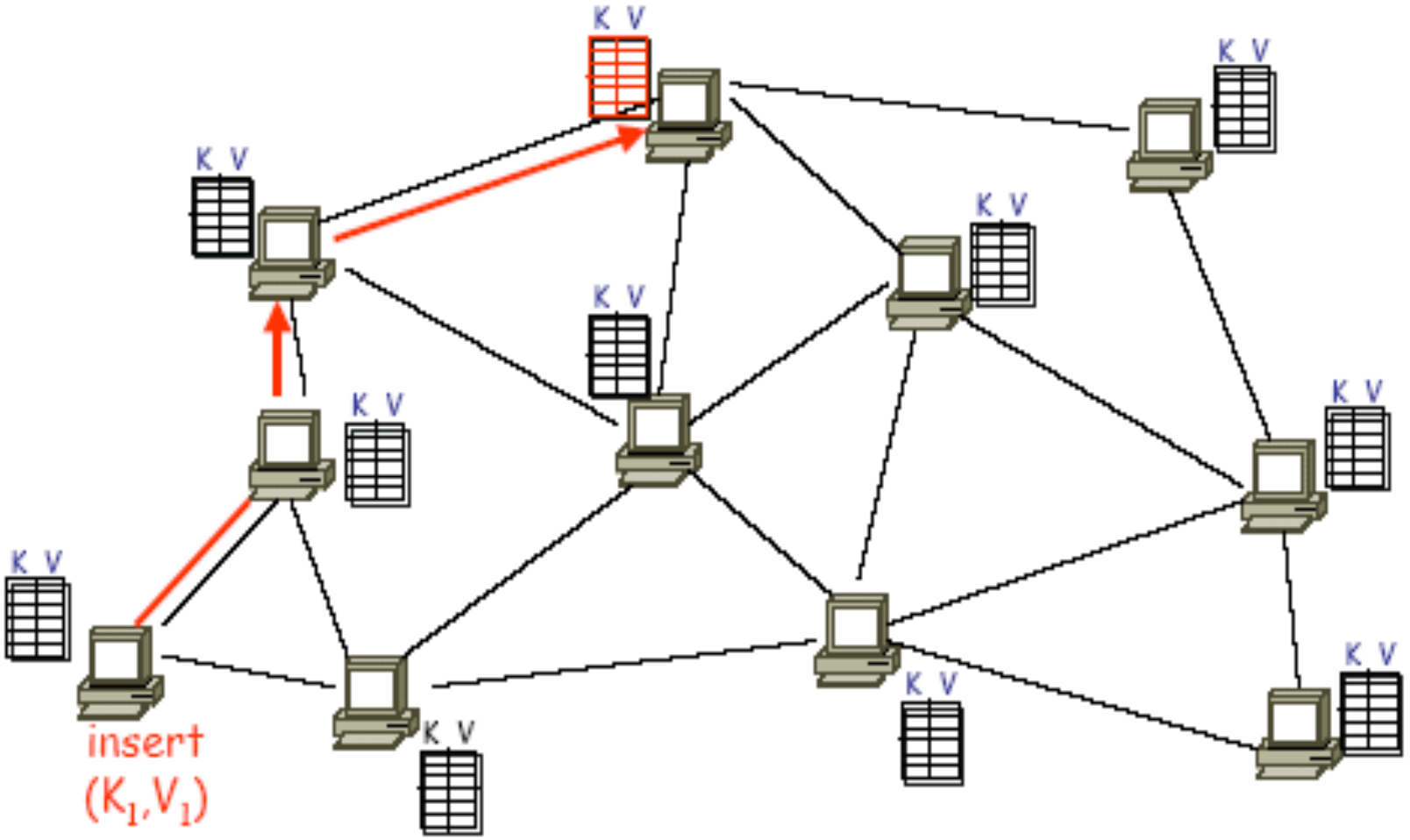
DHT: Basic Idea



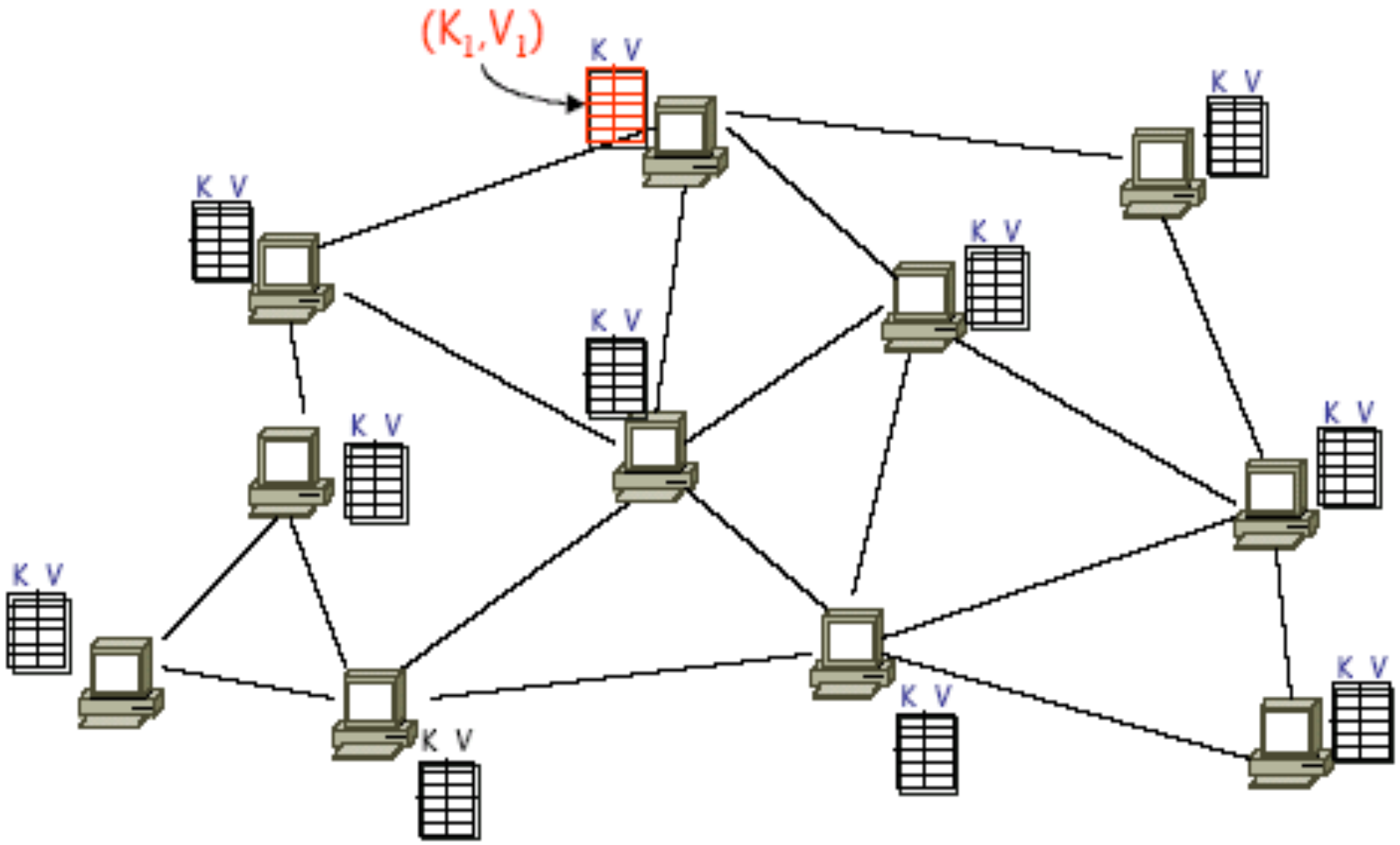
DHT: Basic Idea (2)



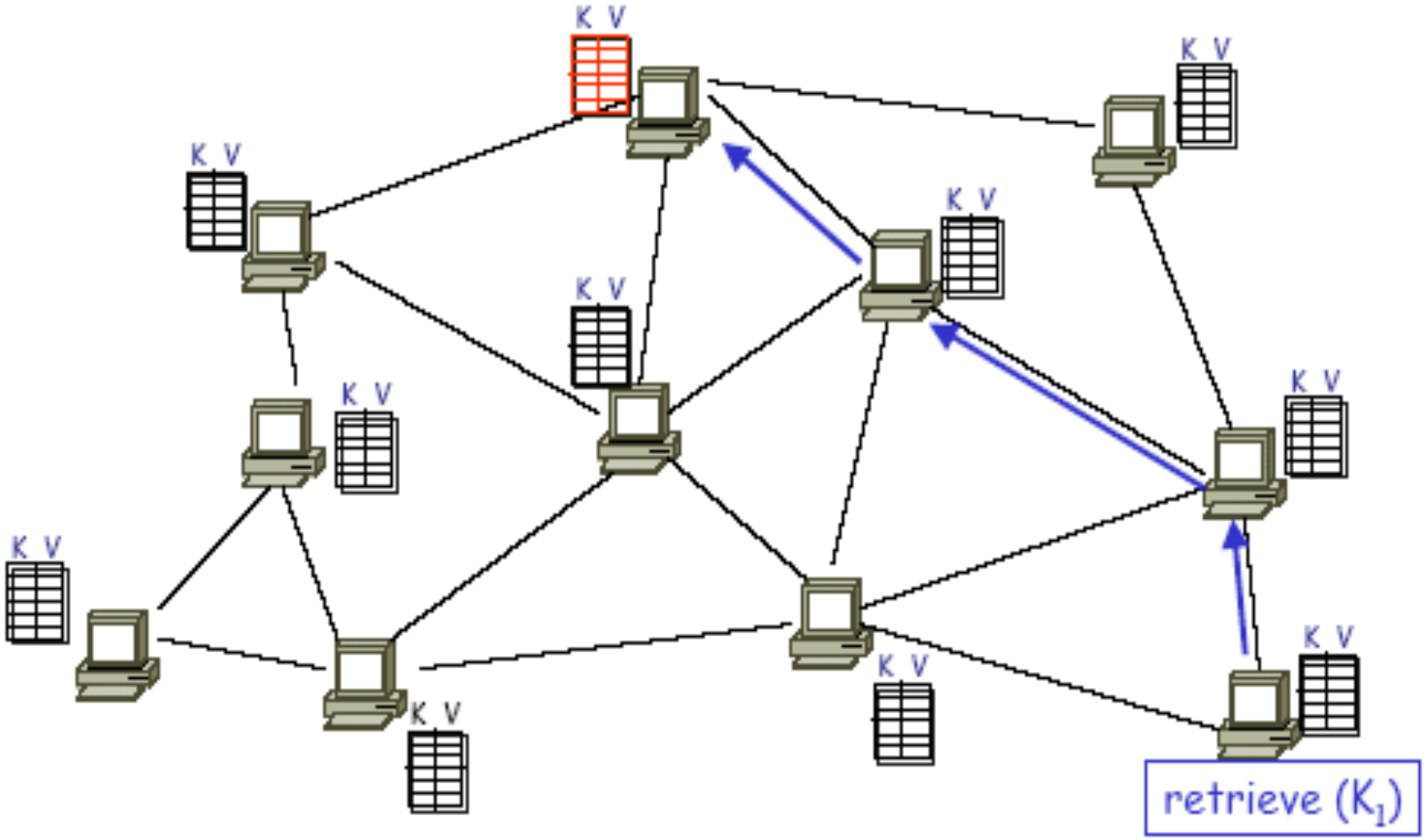
DHT: Basic Idea (3)



DHT: Basic Idea (4)



DHT: Basic Idea (5)

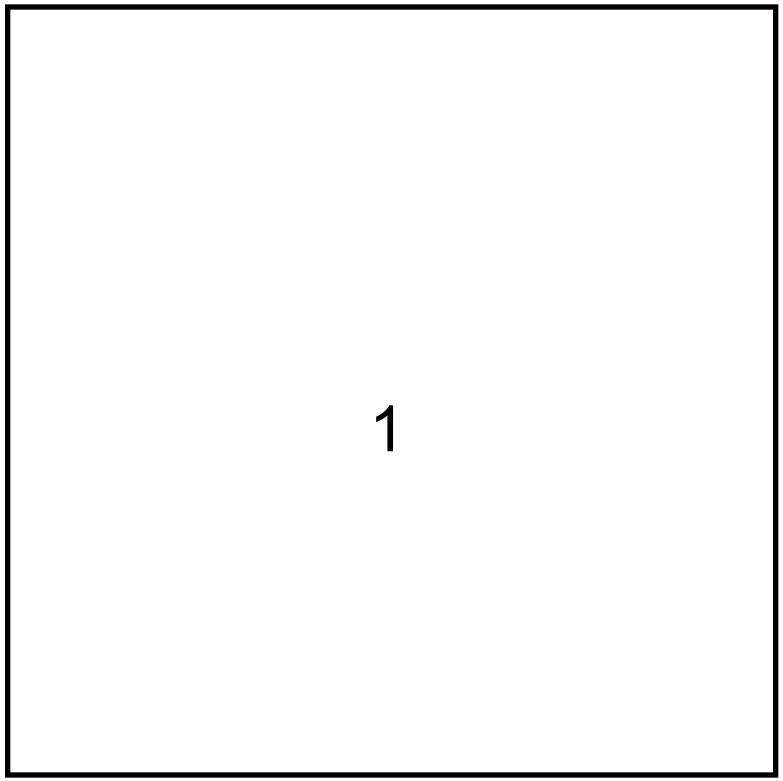


CAN

- Abstraction
  - map a key to a "point" in a multi-dimensional Cartesian space
  - a node "owns" a zone in the overall space
  - route from one "point" to another

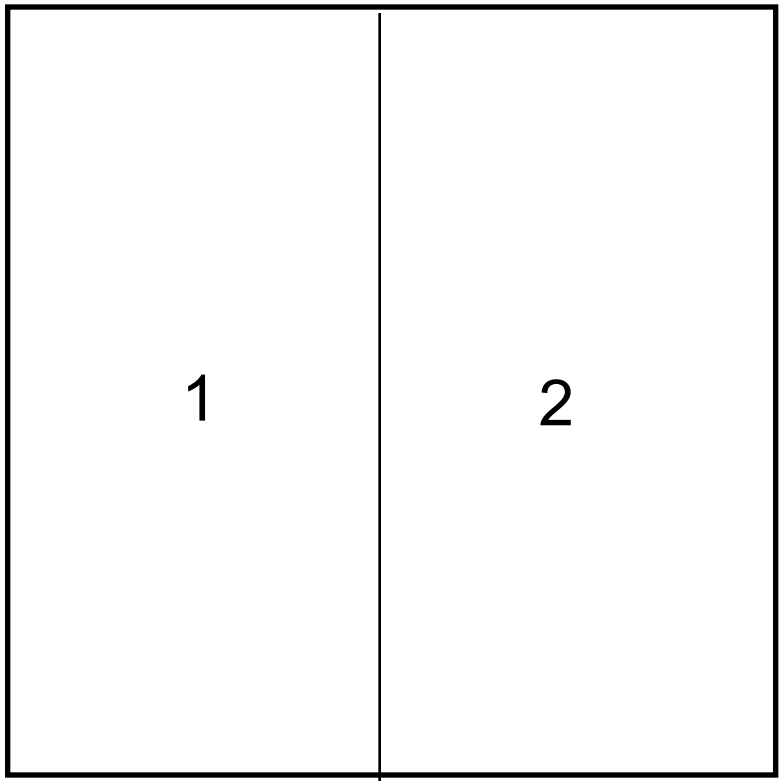
CAN Example: Two Dimensional Space

- Space divided among nodes
- Each node covers either a square or a rectangular area of ratios 1:2 or 2:1



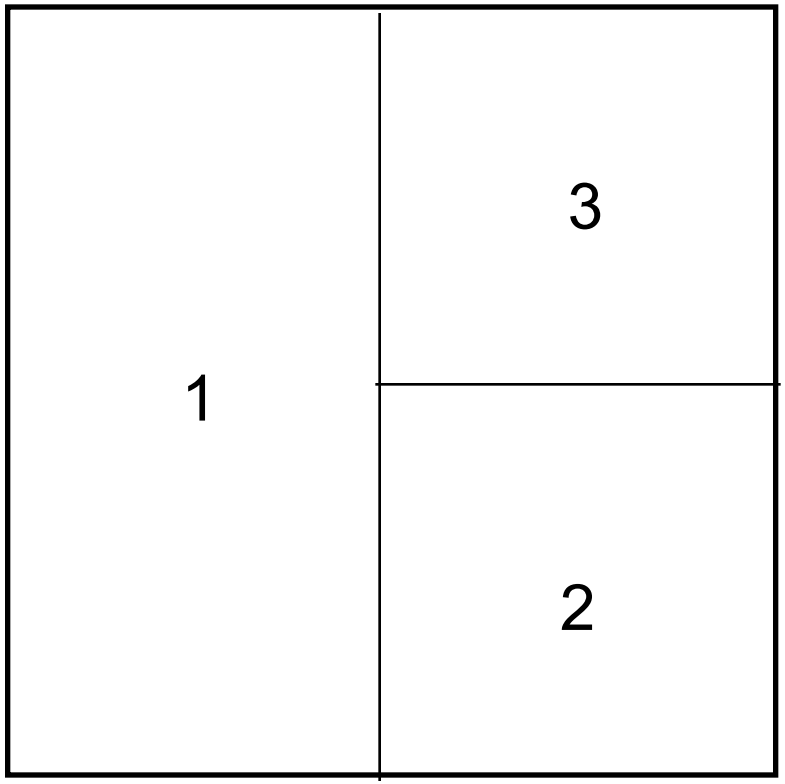
CAN Example: Two Dimensional Space

- Space divided among nodes
- Each node covers either a square or a rectangular area of ratios 1:2 or 2:1



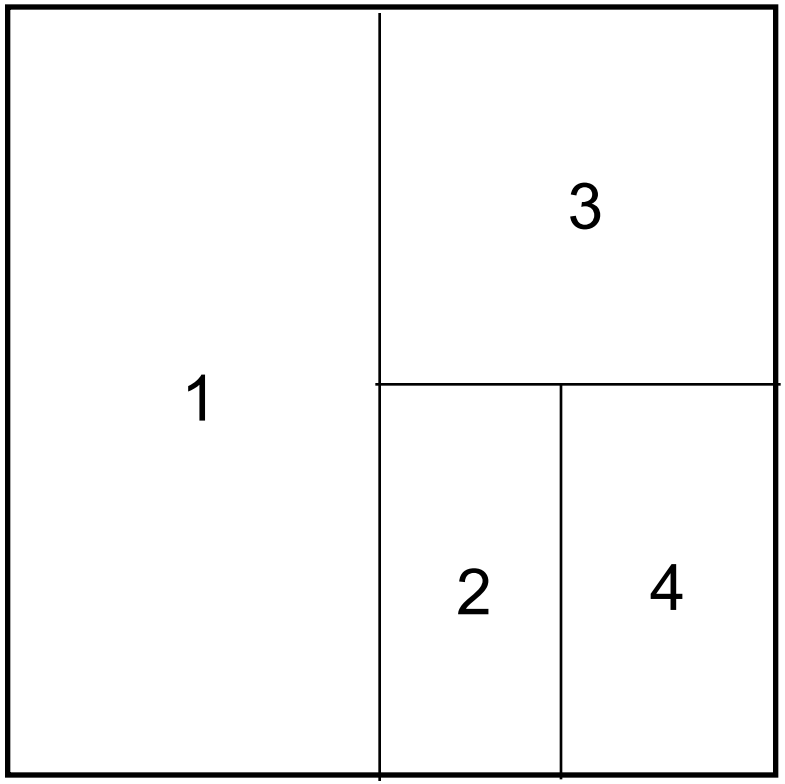
CAN Example: Two Dimensional Space

- Space divided among nodes
- Each node covers either a square or a rectangular area of ratios 1:2 or 2:1



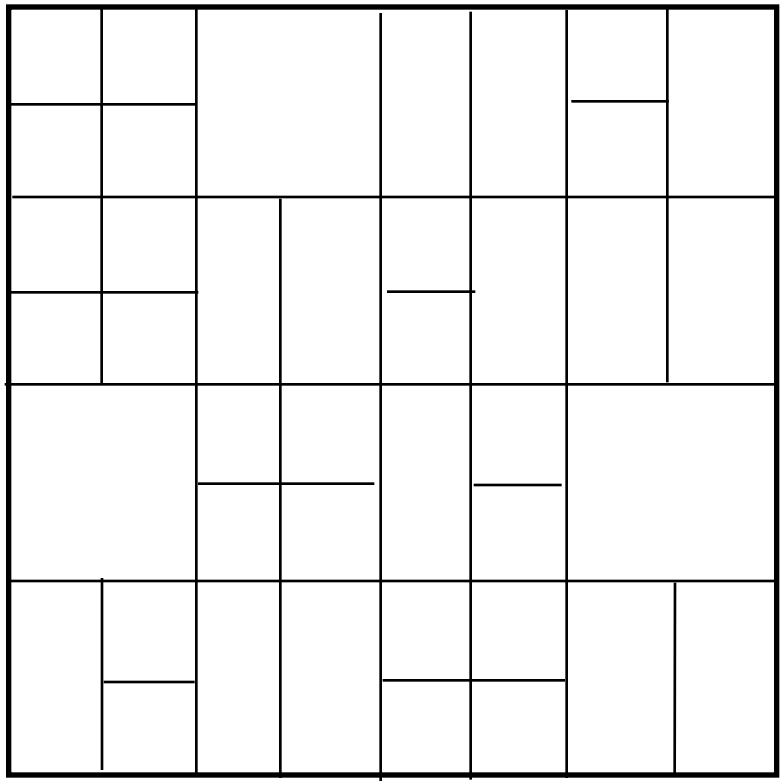
CAN Example: Two Dimensional Space

- Space divided among nodes
- Each node covers either a square or a rectangular area of ratios 1:2 or 2:1



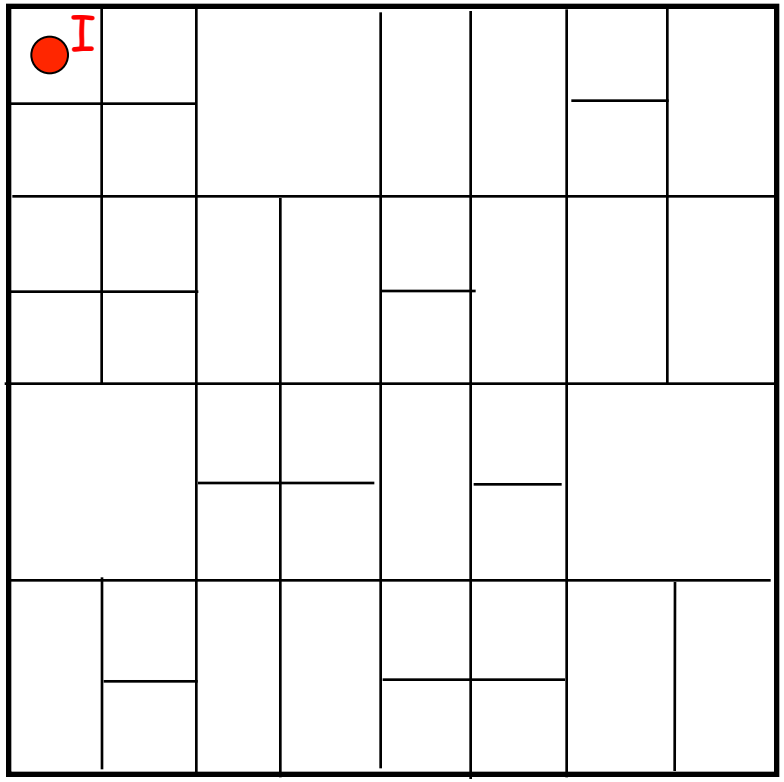
CAN Example: Two Dimensional Space

- Space divided among nodes
- Each node covers either a square or a rectangular area of ratios 1:2 or 2:1



CAN Insert: Example (1)

node I::insert(K,V)

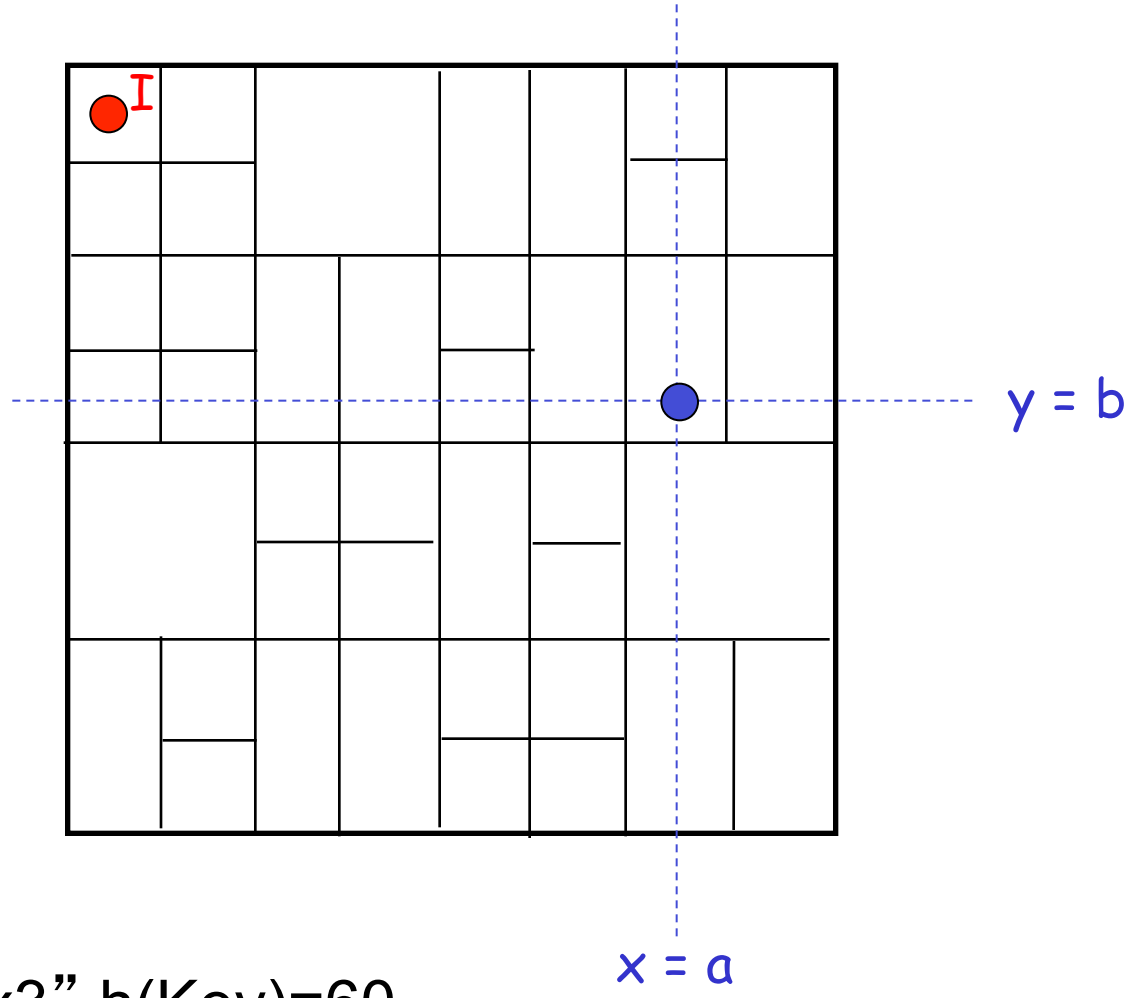


CAN Insert: Example (2)

node I::insert(K,V)

(1) a = h_x(K)

b = h_y(K)



Example: Key="Matrix3" h(Key)=60

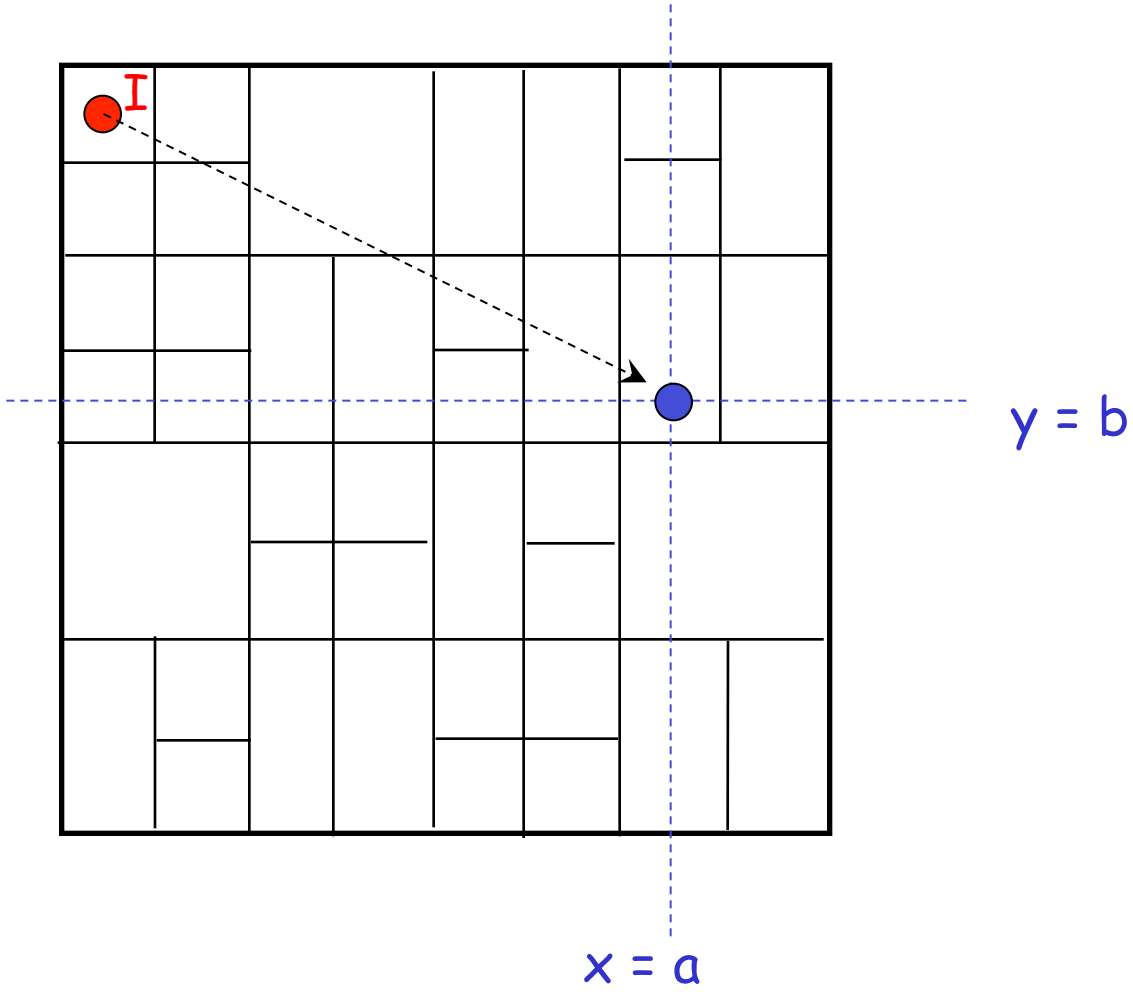
CAN Insert: Example (3)

node I::insert(K,V)

(1) a = h_x(K)

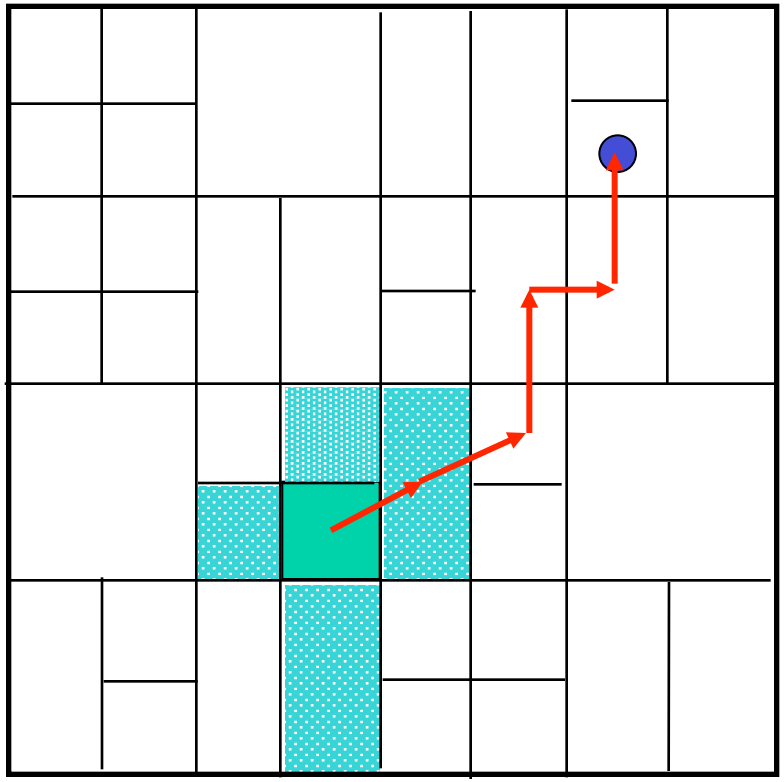
b = h_y(K)

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



CAN Insert: Routing

- A node maintains state only for its immediate neighboring nodes
- Forward to neighbor which is closest to the target point
  - a type of greedy, local routing scheme



CAN Insert: Example (4)

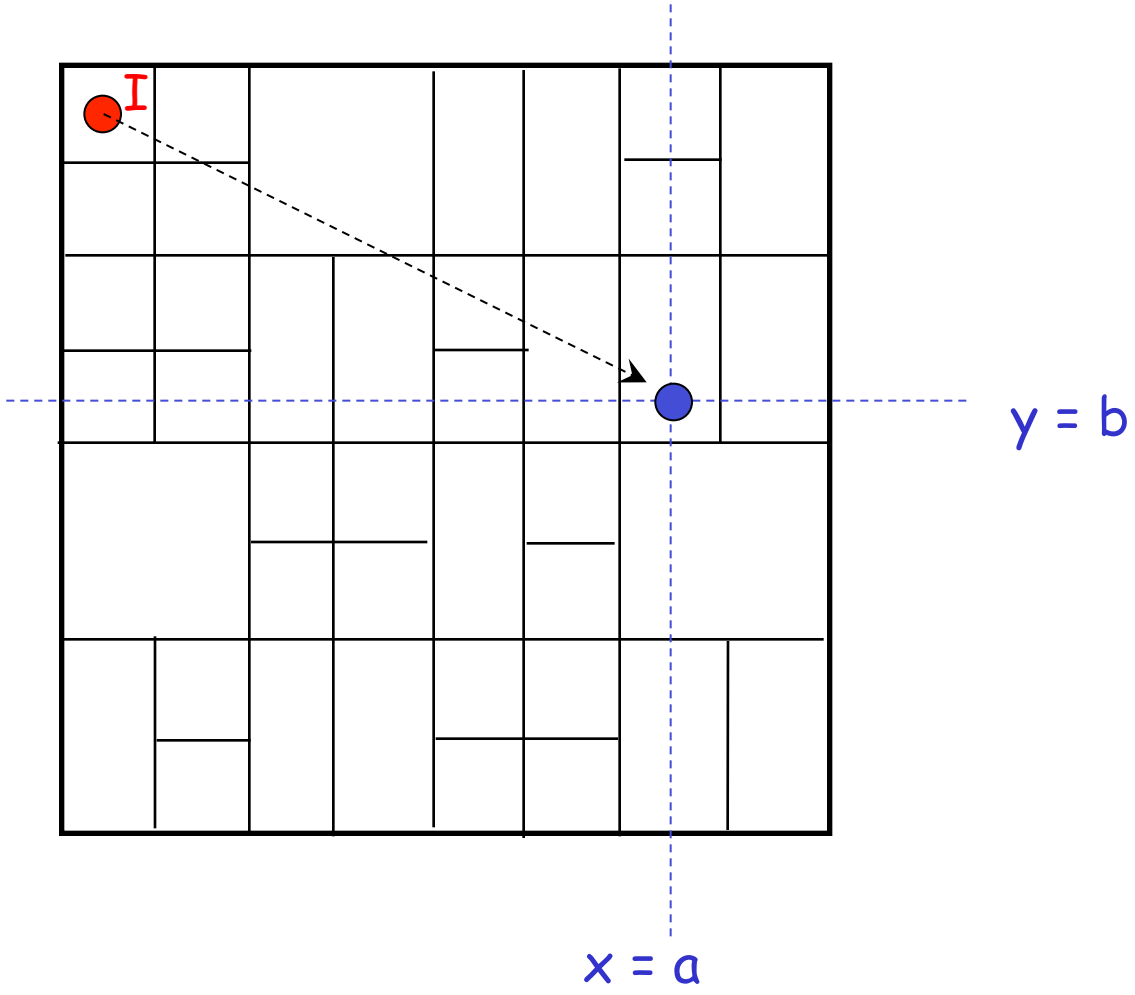
node I::insert(K,V)

(1) a = h_x(K)

b = h_y(K)

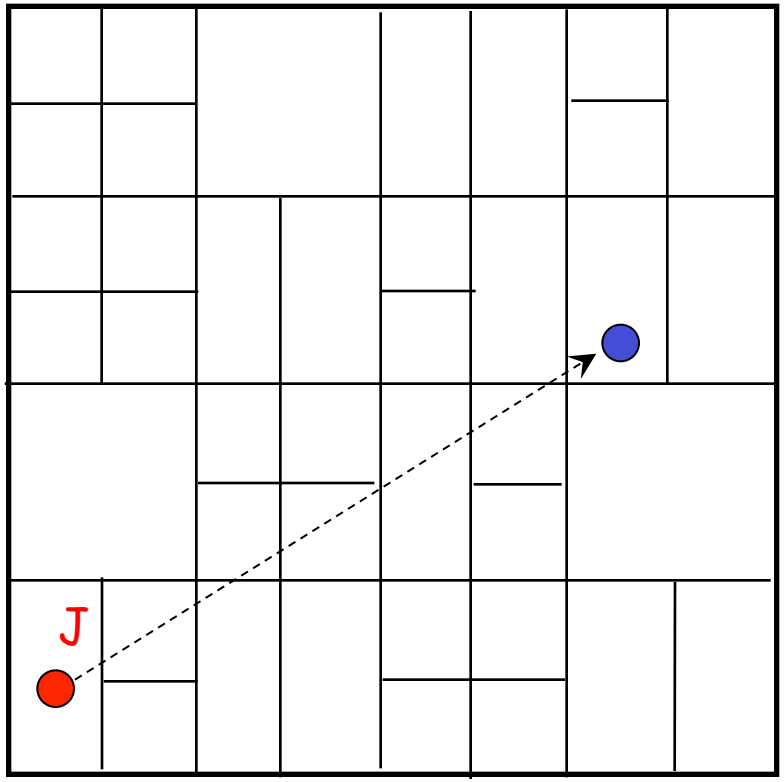
(2) route(K,V) -> (a,b)

(3) (K,V) is stored at (a,b)



CAN Retrieve: Example

node J::retrieve(K)



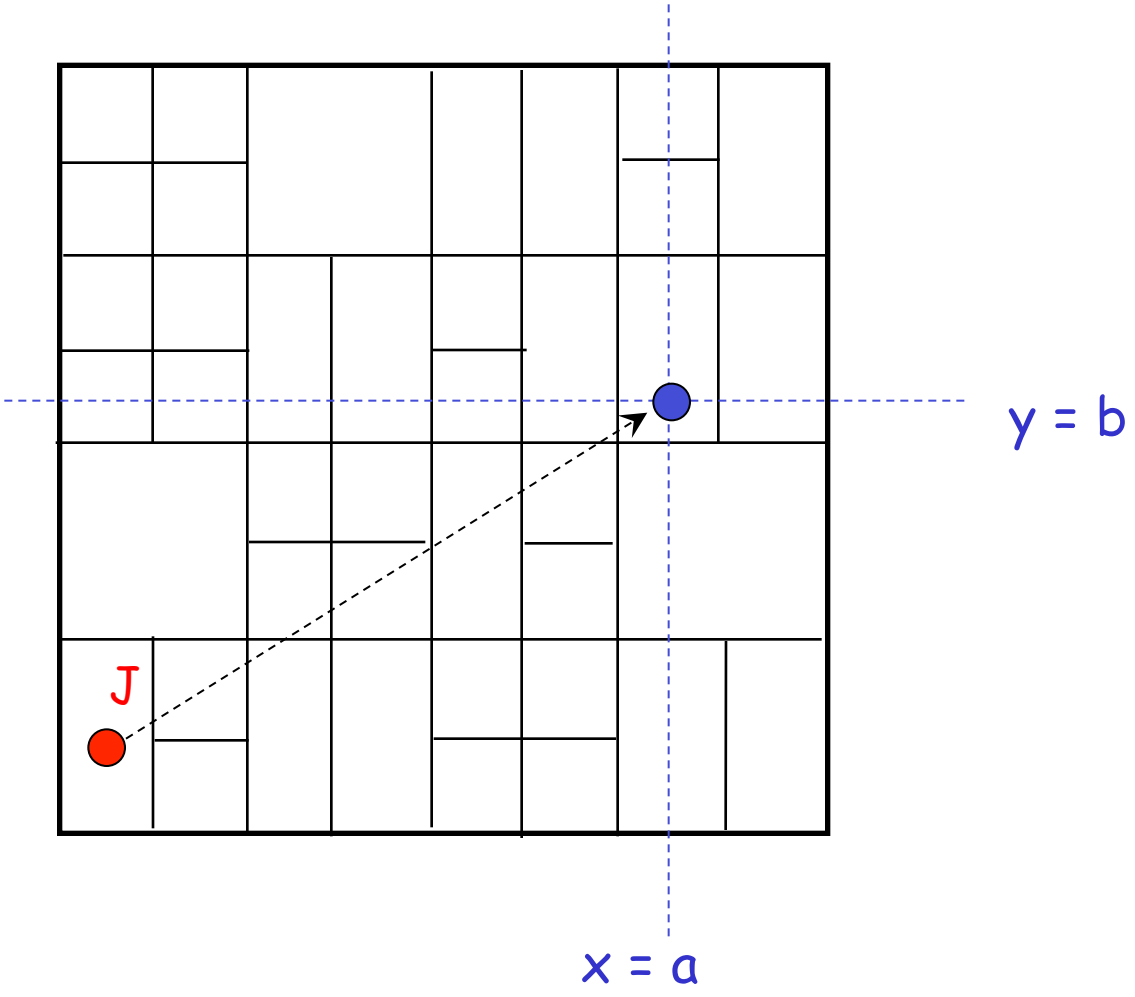
CAN Retrieve: Example

node J::retrieve(K)

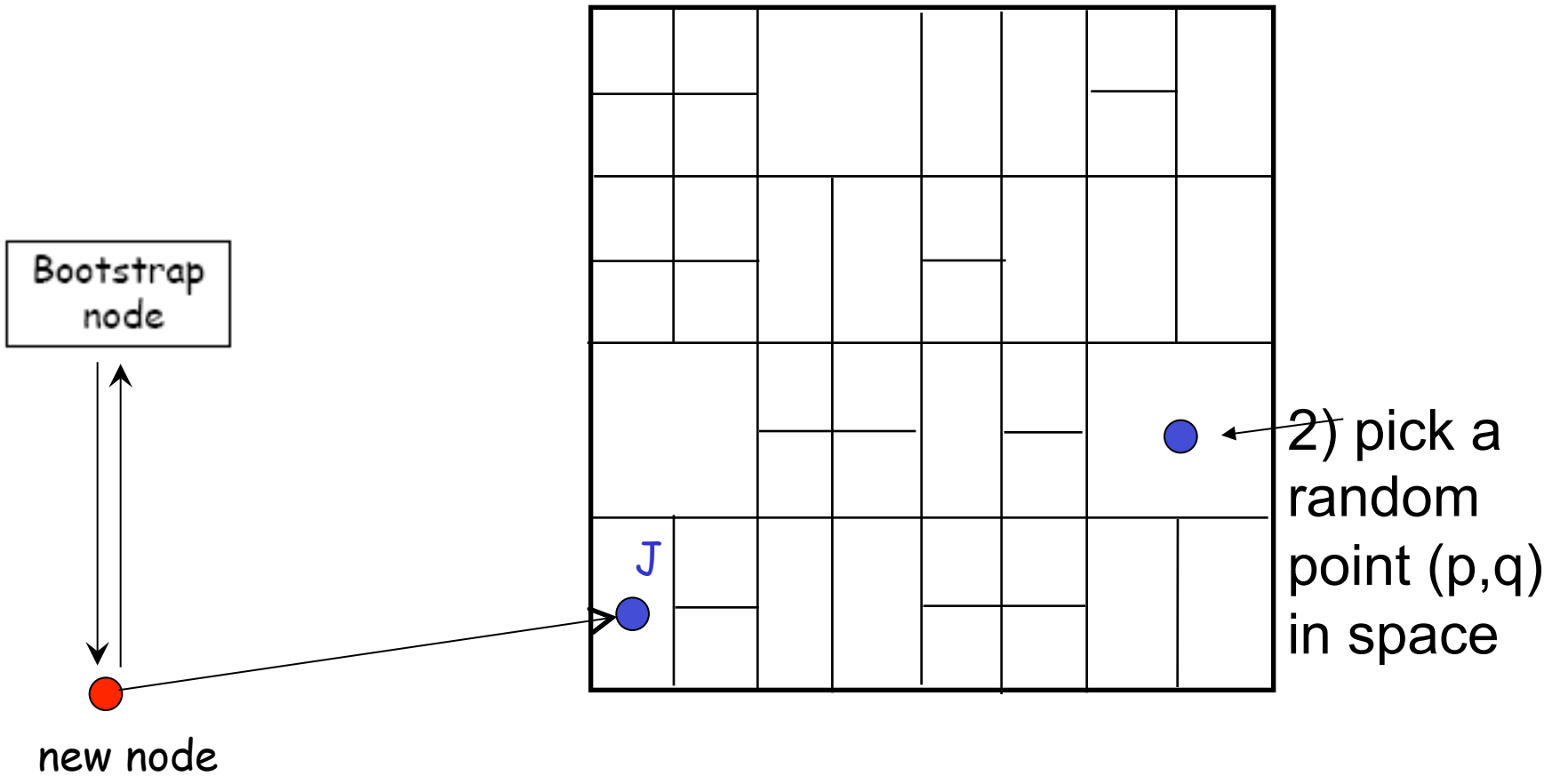
(1) a = h_x(K)

b = h_y(K)

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

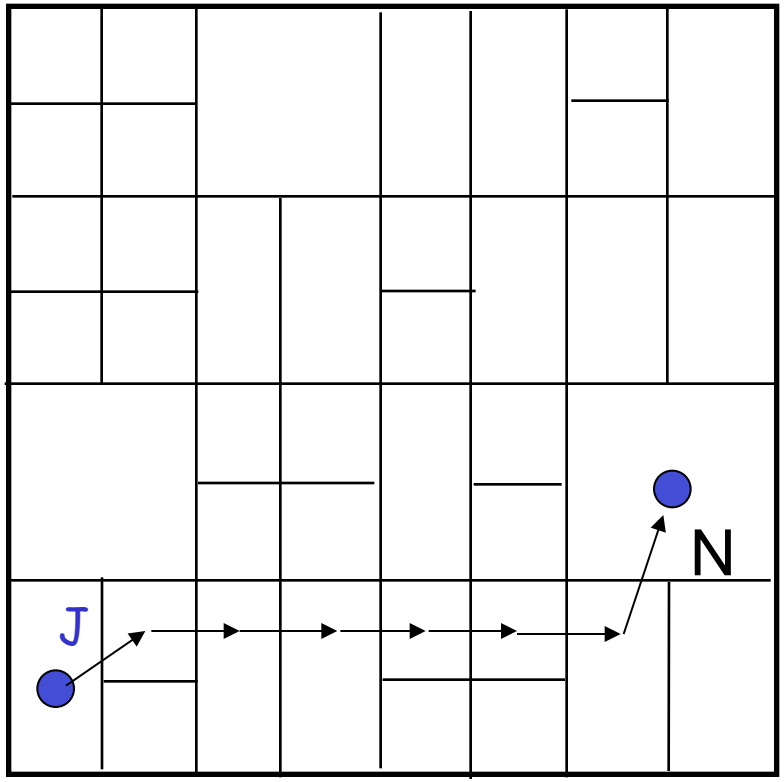


CAN Insert: Join (1)



1) Discover some node "J" already in CAN

CAN Insert: Join (2)

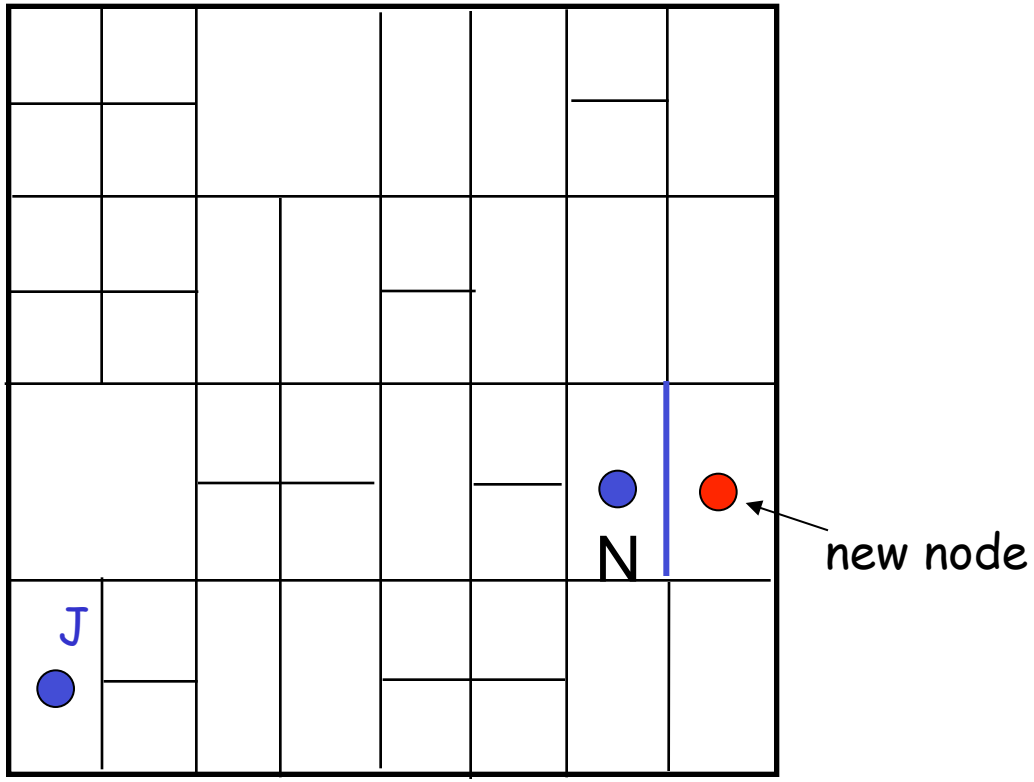


new node

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

CAN Insert: Join (3)

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



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

CAN Evaluations

- Guarantee to find an item if in the network
- Load balancing
  - hashing achieves some load balancing
  - overloaded node replicates popular entries at neighbors
- Scalability
  - for a uniform (regularly) partitioned space with **n** nodes and **d** dimensions
  - storage:
    - per node, number of neighbors is 2d
  - routing
    - average routing path is $(dn^{1/d})/3$ hops (due to Manhattan distance routing, expected hops in each dimension is dimension length * 1/3)
    - a fixed d can scale the network without increasing per-node state

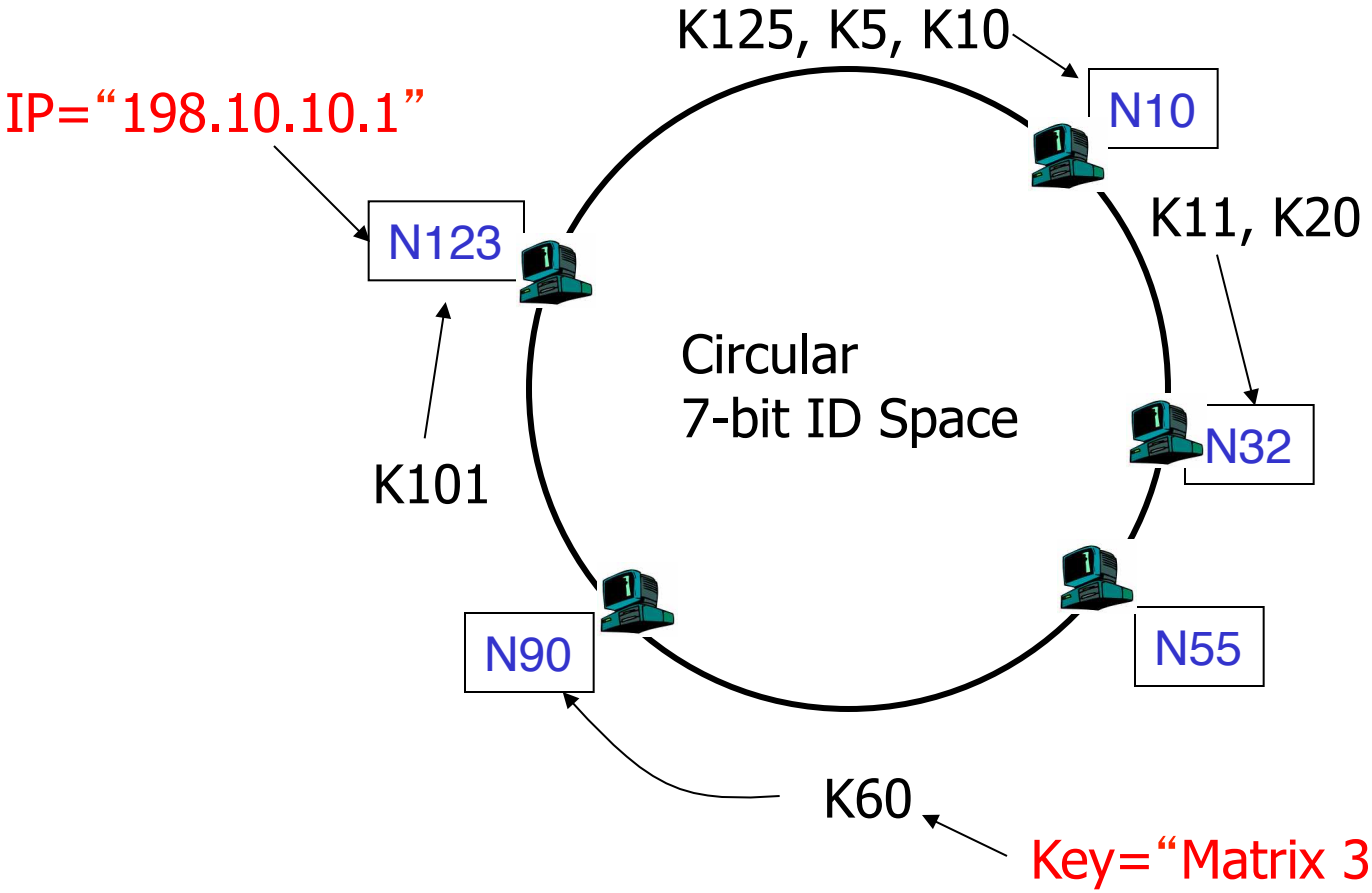
Chord

- Space is a ring

- Consistent hashing: m bit identifier space for both keys and nodes
  - key identifier = SHA-1(key), where SHA-1() is a popular hash function,

    Key="Matrix3" → ID=60
  - node identifier = SHA-1(IP address)
    - IP="198.10.10.1" → ID=123

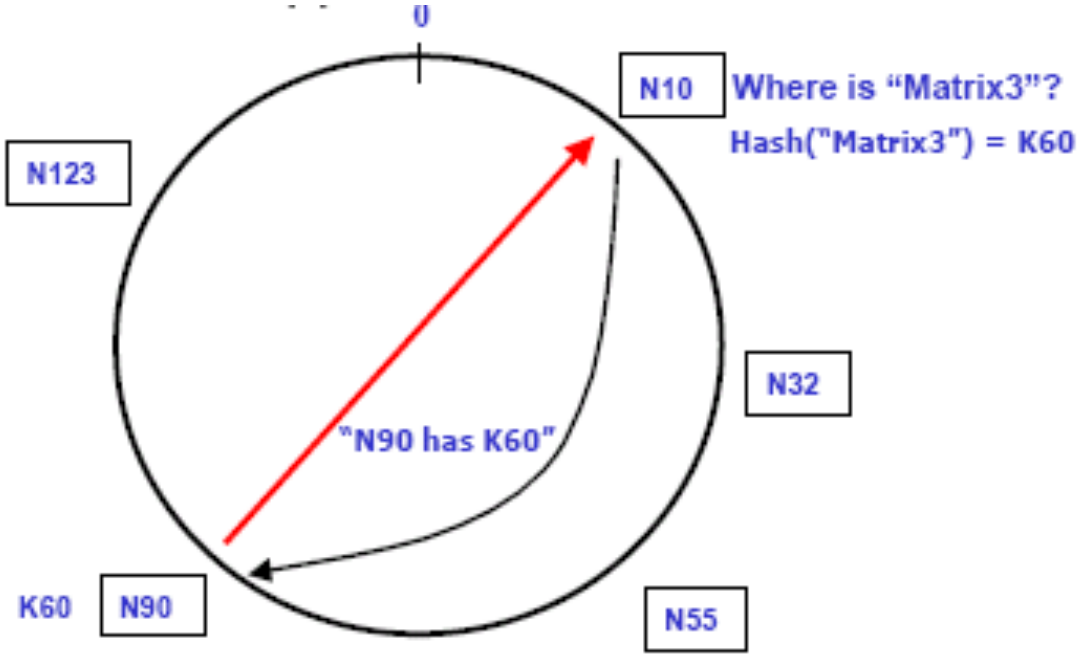
Chord: Storage using a Ring



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

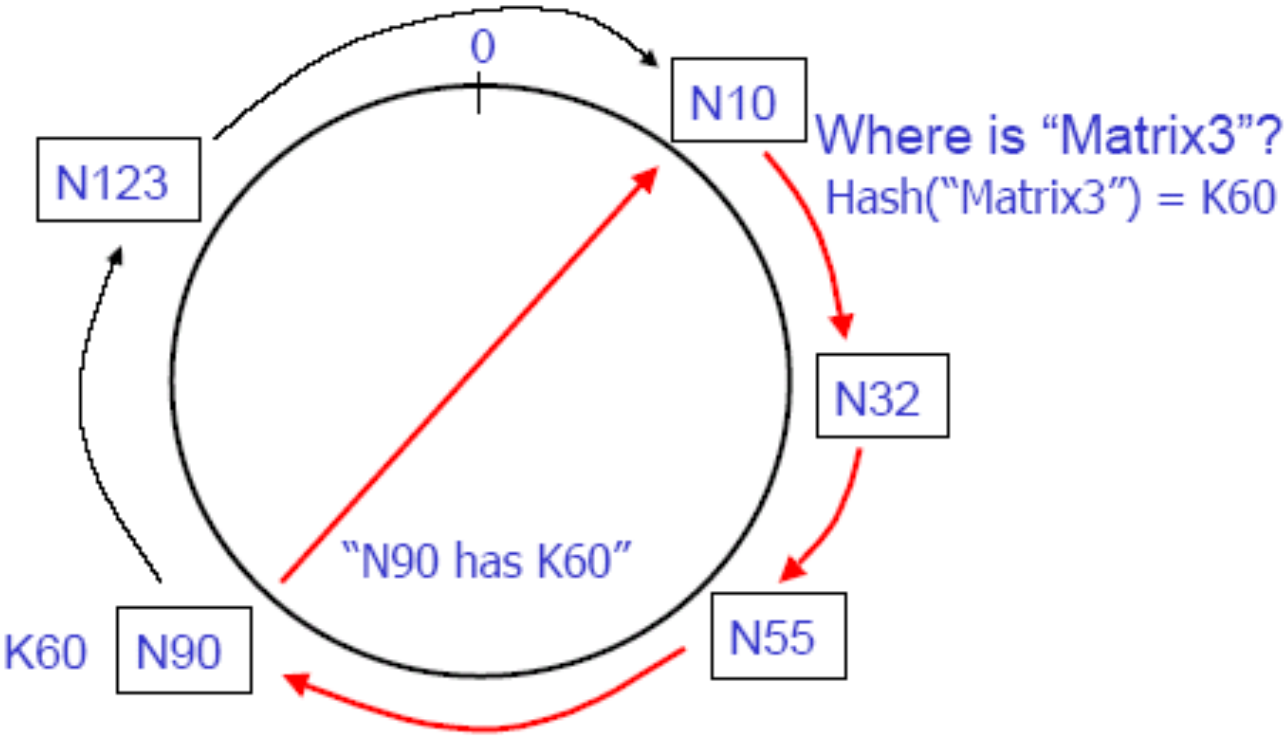
How to Search: One Extreme

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



How to Search: the Other Extreme

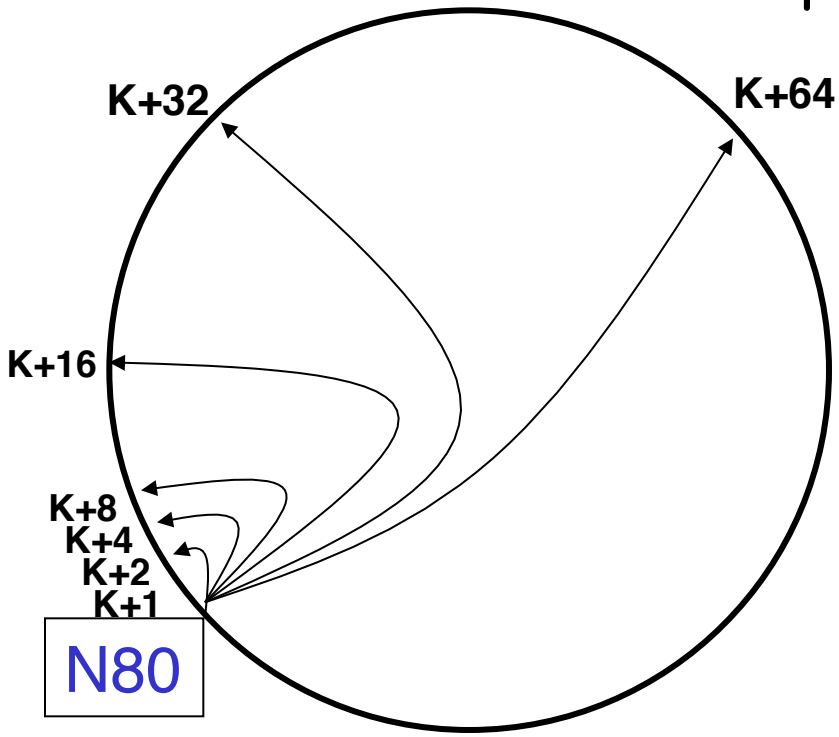
- Every node knows its successor in the ring

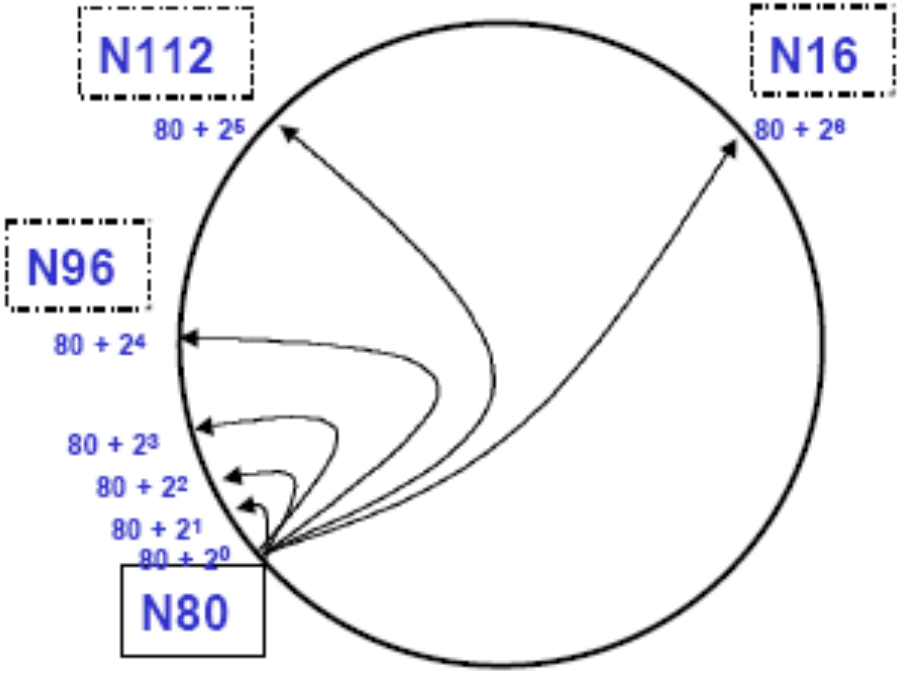


- Routing tables are small O(1)
- Lookups are slow O(N)

Chord Solution: "finger tables"

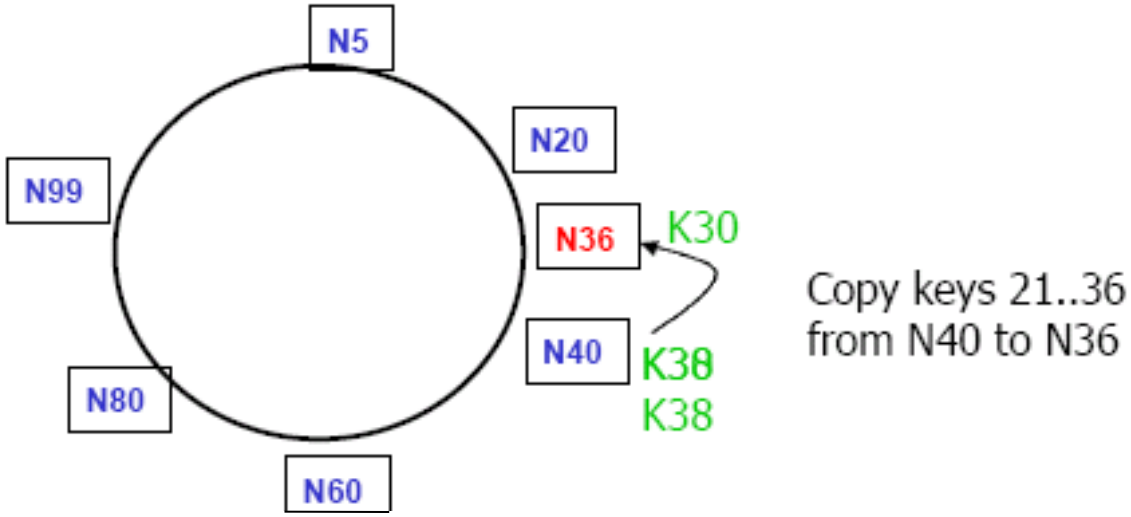
- Node K knows the node that is maintaining $K + 2^i$, where K is mapped id of current node
  - increase distance exponentially





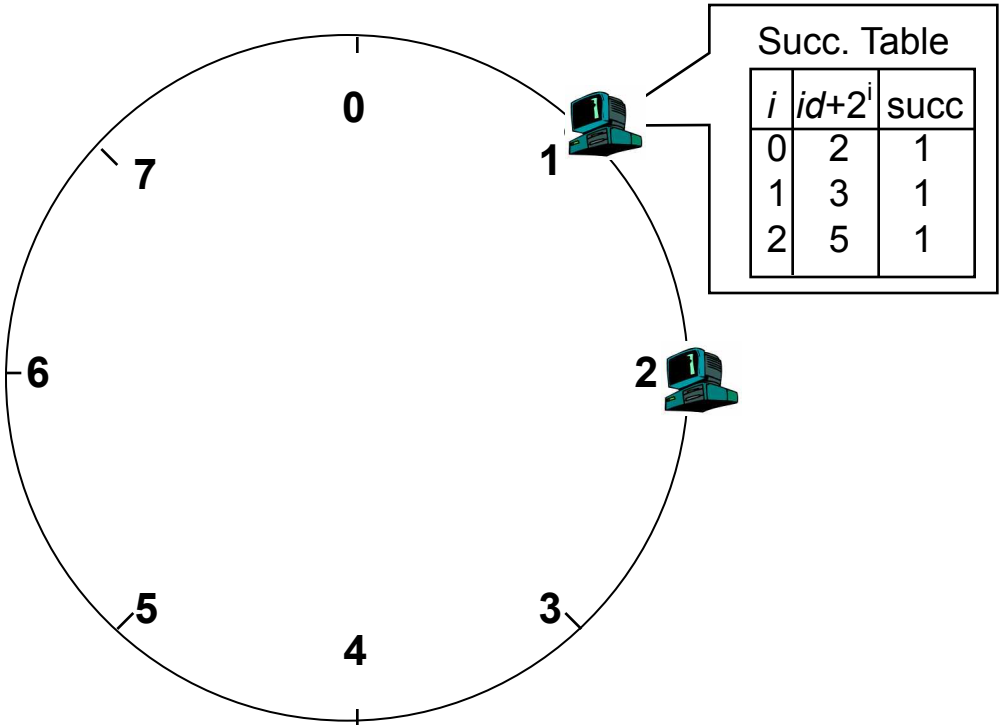
Joining the Ring

- use a contact node to obtain info
- transfer keys from successor node to new node
- updating fingers of existing nodes



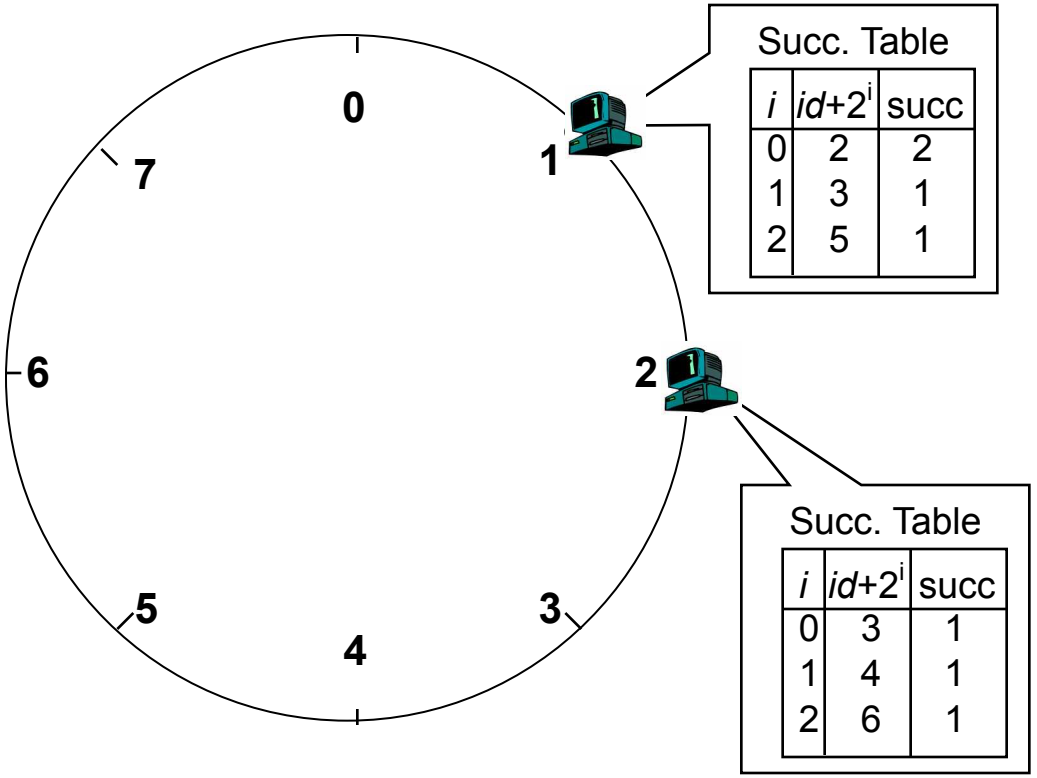
DHT: Chord Node Join

- Assume an identifier space [0..8]
- Node n1 joins



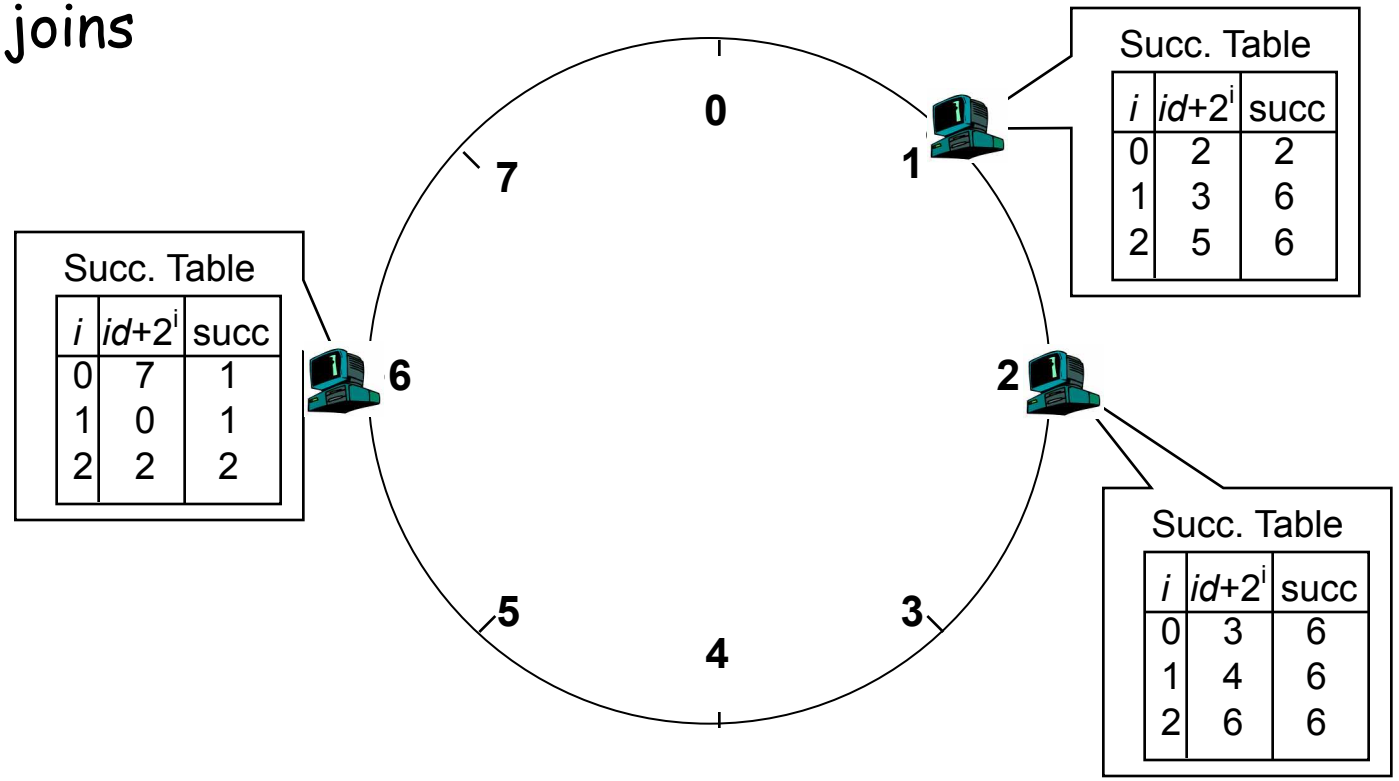
DHT: Chord Node Join

- Node n2 joins



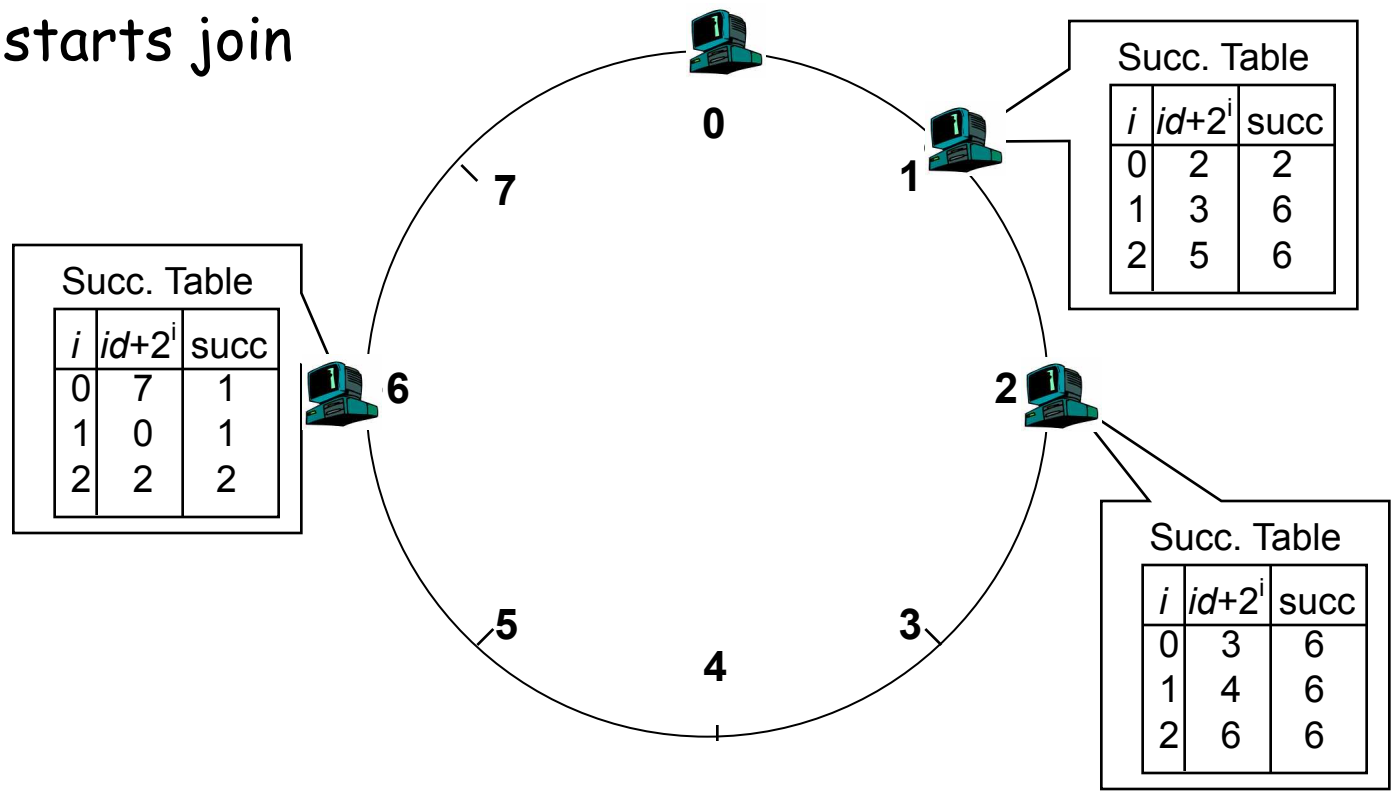
DHT: Chord Node Join

- Node n6 joins



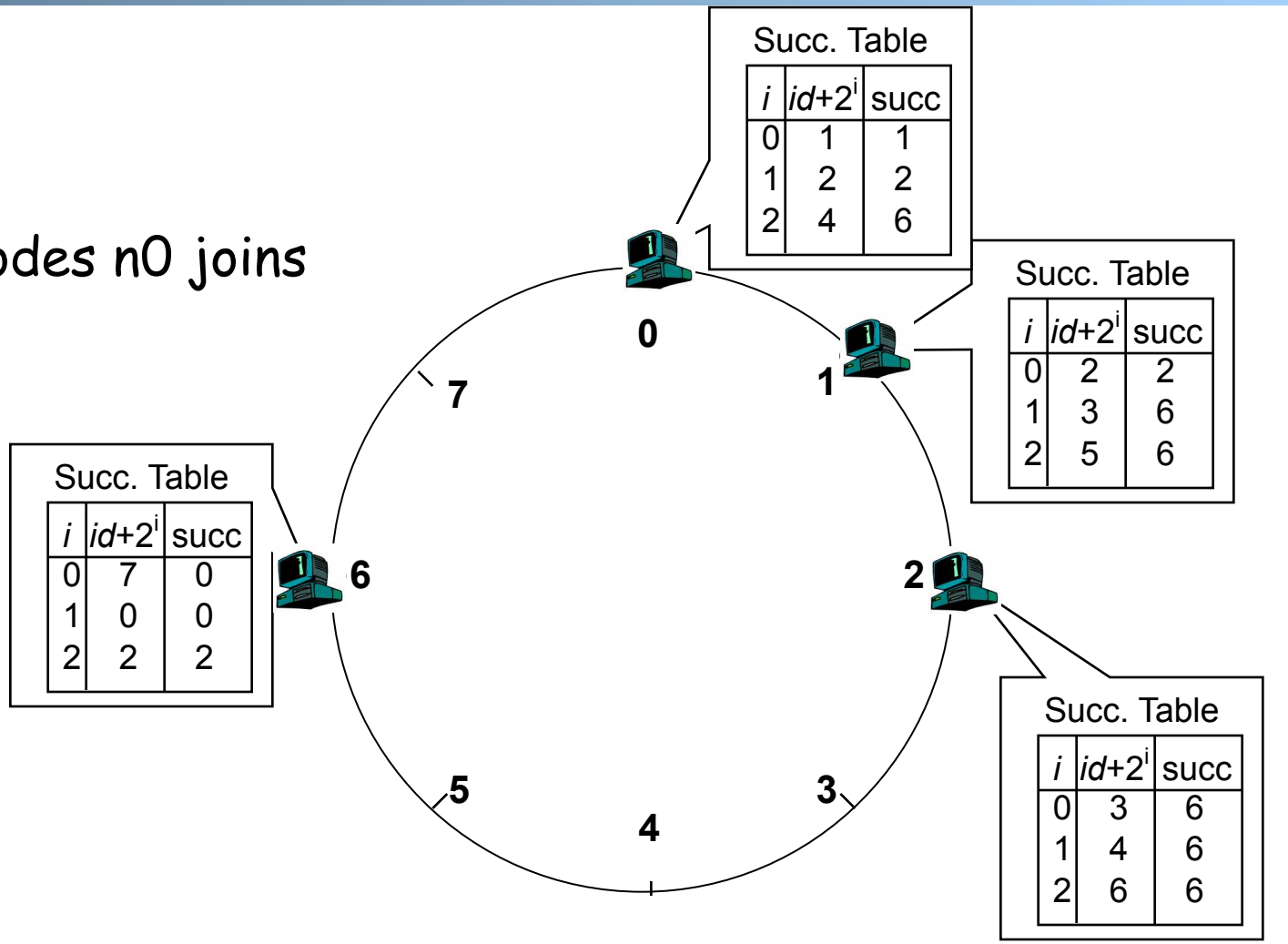
DHT: Chord Node Join

- Node n0 starts join



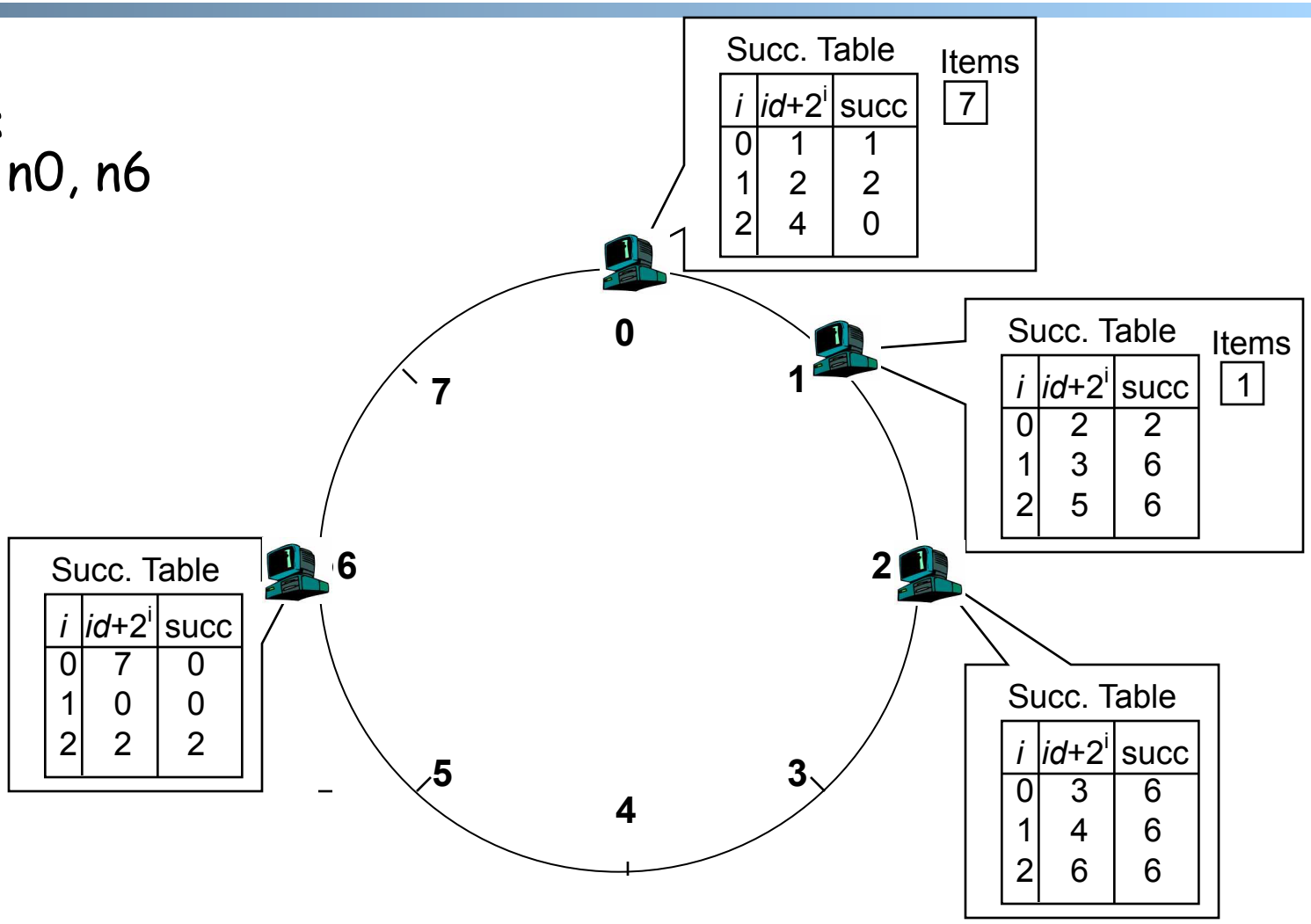
DHT: Chord Node Join

- After Nodes n0 joins



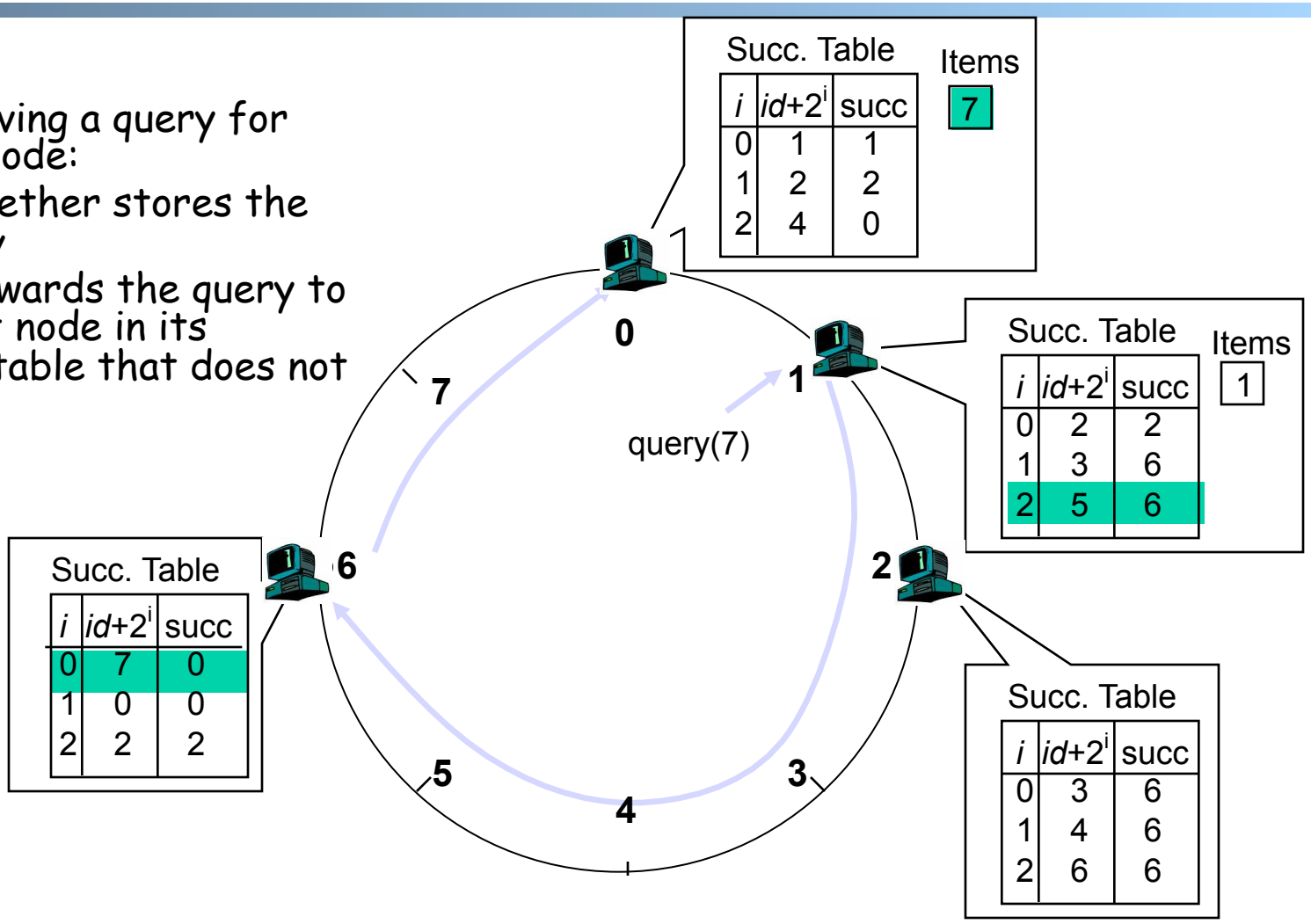
DHT: Chord Insert Items

- Nodes:
  n1, n2, n0, n6
- Items:
  f7, f1



DHT: Chord Routing

- Upon receiving a query for item *id*, a node:
- checks whether stores the item locally
- if not, forwards the query to the largest node in its successor table that does not exceed *id*



Chord/CAN Summary

- Each node "owns" some portion of the key-space
  - in CAN, it is a multi-dimensional "zone"
  - in Chord, it is the key-id-space between two nodes in 1-D ring
- Files and nodes are assigned random locations in key-space
  - provides some load balancing
    - probabilistically equal division of keys to nodes
- Routing/search is local (distributed) and greedy
  - node X does not know of a path to a key Z
  - but if it appears that node Y is the closest to Z among all of the nodes known to X
  - so route to Y