Peer to Peer Systems in Cloud Computing

Dr. Rajiv Misra
Associate Professor
Dept. of Computer Science & Engg.
Indian Institute of Technology Patna
rajivm@iitp.ac.in

Preface

Content of this Lecture:

- In this lecture, we will discuss the Peer to Peer (P2P) techniques in cloud computing systems.

- We will study some of the widely-deployed P2P systems such as: **Napster, Gnutella, Fasttrack and BitTorrent** and P2P Systems with provable properties such as: **Chord, Pastry and Kelips.**

Need of Peer to Peer Systems

- First distributed systems that seriously focused on **scalability with respect to number of nodes**
- P2P techniques be abundant in cloud computing systems
  - Key-value stores (e.g., Cassandra) use Chord p2p hashing

P2P Systems

Widely-deployed P2P Systems:

1. Napster
2. Gnutella
3. Fasttrack
4. BitTorrent

P2P Systems with Provable Properties:

1. Chord
2. Pastry
3. Kelips

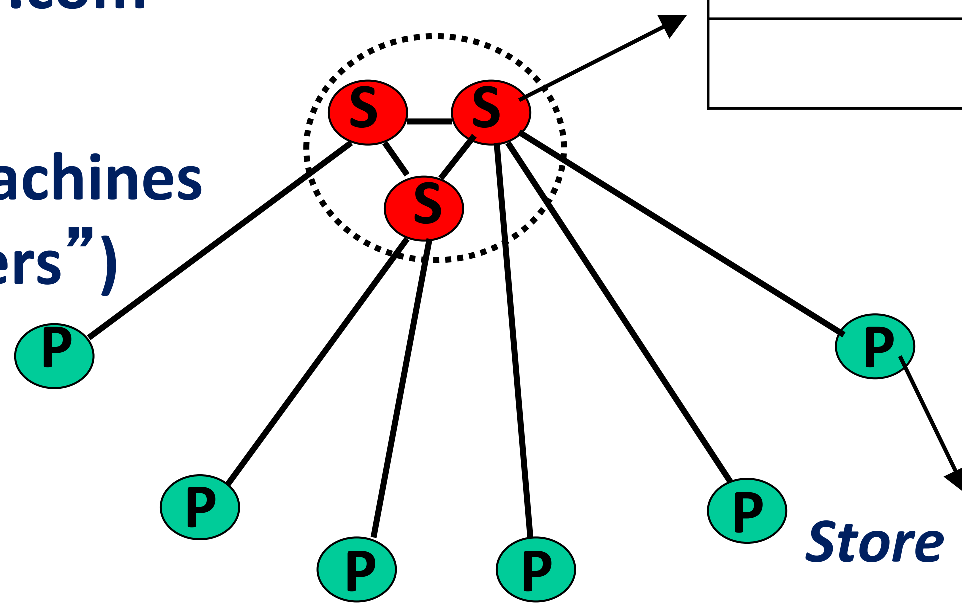
Napster Structure

Store a directory, i.e., filenames with peer pointers

Filename	Info about
Public enemy.mp3	Beatles, @123.34.12.32: 1003

napster.com
Servers

Client machines
("Peers")



Store their own files

Napster Structure

Client

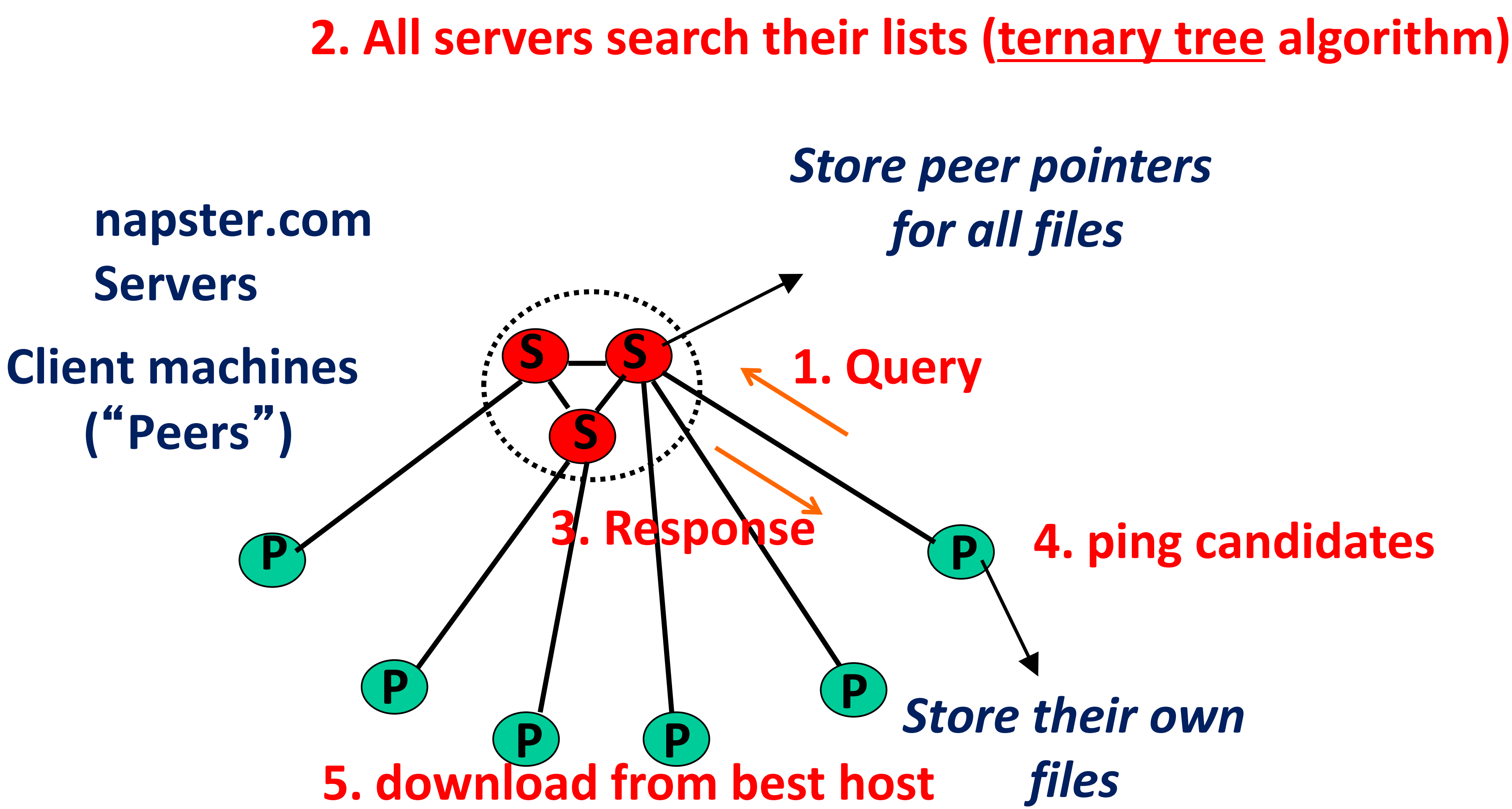
- Connect to a Napster server:
  - Upload list of music files that you want to share
  - Server maintains list of <filename, ip_address, portnum> tuples. **Server stores no files.**

Napster Operations

Client (contd.)

- **Search**
  - Send server keywords to search with
  - (Server searches its list with the keywords)
  - Server returns a list of hosts - <ip_address, portnum> tuples - to client
  - Client pings each host in the list to find transfer rates
  - Client fetches file from best host
- **All communication uses TCP (Transmission Control Protocol)**
  - Reliable and ordered networking protocol

Napster Search



Nodes Joining a P2P system

- **Can be used for any p2p system**
  - Send an http request to well-known url for that P2P service.
  - Message routed (after lookup in DNS=Domain Name system) to introducer, a well known server that keeps track of some recently joined nodes in p2p system
  - Introducer initializes new peers' neighbor table

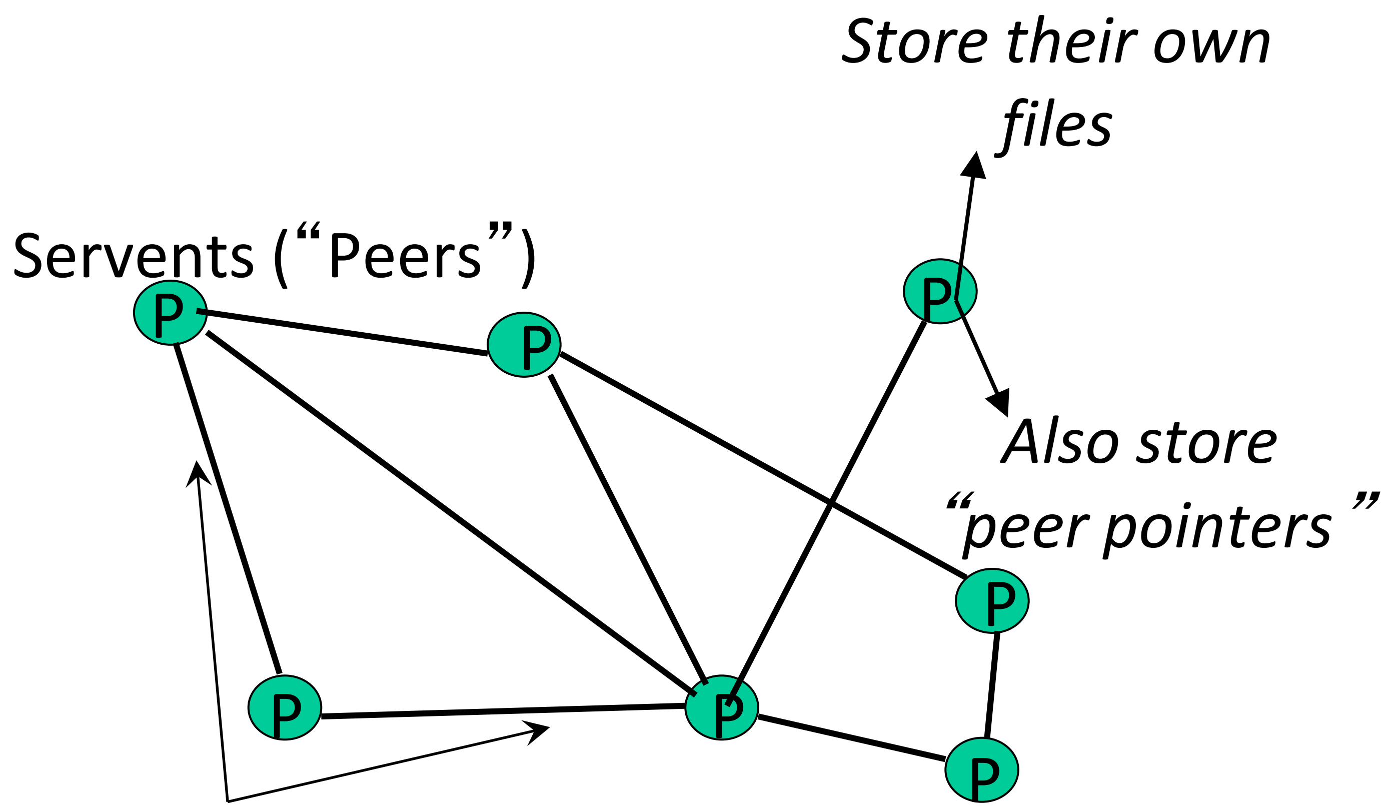
Issues with Napster

- Centralized server a source of congestion
- Centralized server single point of failure
- No security: plaintext messages and passwords
- **napster.com** declared to be responsible for users' copyright violation
  - "Indirect infringement"
  - **Next P2P system: Gnutella**

Gnutella

- Eliminate the servers
- Client machines search and retrieve amongst themselves
- Clients act as servers too, called **servents**
- **Gnutella** (possibly by analogy with the GNU Project) is a large peer-to-peer network. It was the first decentralized peer-to-peer network of its kind.
- [Mar 2000] release by AOL, immediately withdrawn, but 88K users by [Mar 2003]
- Original design underwent several modifications

Gnutella



Connected in an **overlay graph**

(== each link is an implicit Internet path)

How do I search for a particular file?

- Gnutella ***routes*** different messages within the overlay graph
- Gnutella protocol has 5 main message types
  1. **Query** **(search)**
  2. **QueryHit** **(response to query)**
  3. **Ping** **(to probe network for other peers)**
  4. **Pong** **(reply to ping, contains address of another peer)**
  5. **Push** **(used to initiate file transfer)**
- Into the message structure and protocol
  - All fields except IP address are in little-endian format
  - 0x12345678 stored as 0x78 in lowest address byte, then 0x56 in next higher address, and so on.

How do I search for a particular file?

Descriptor Header | **Payload**

Descriptor ID	Payload descriptor	TTL	Hops	Payload length
0 – 15	16	17	18	22
ID of this search transaction	*Type of payload* *0x00 Ping* *0x01 Pong* *0x40 Push* *0x80 Query* *0x81 Queryhit*	*Decremented at each hop, Message dropped when ttl=0* *ttl_initial usually 7 to 10*	*Incremented at each hop*	*Number of bytes of message following this header*

Gnutella Message Header Format

How do I search for a particular file?

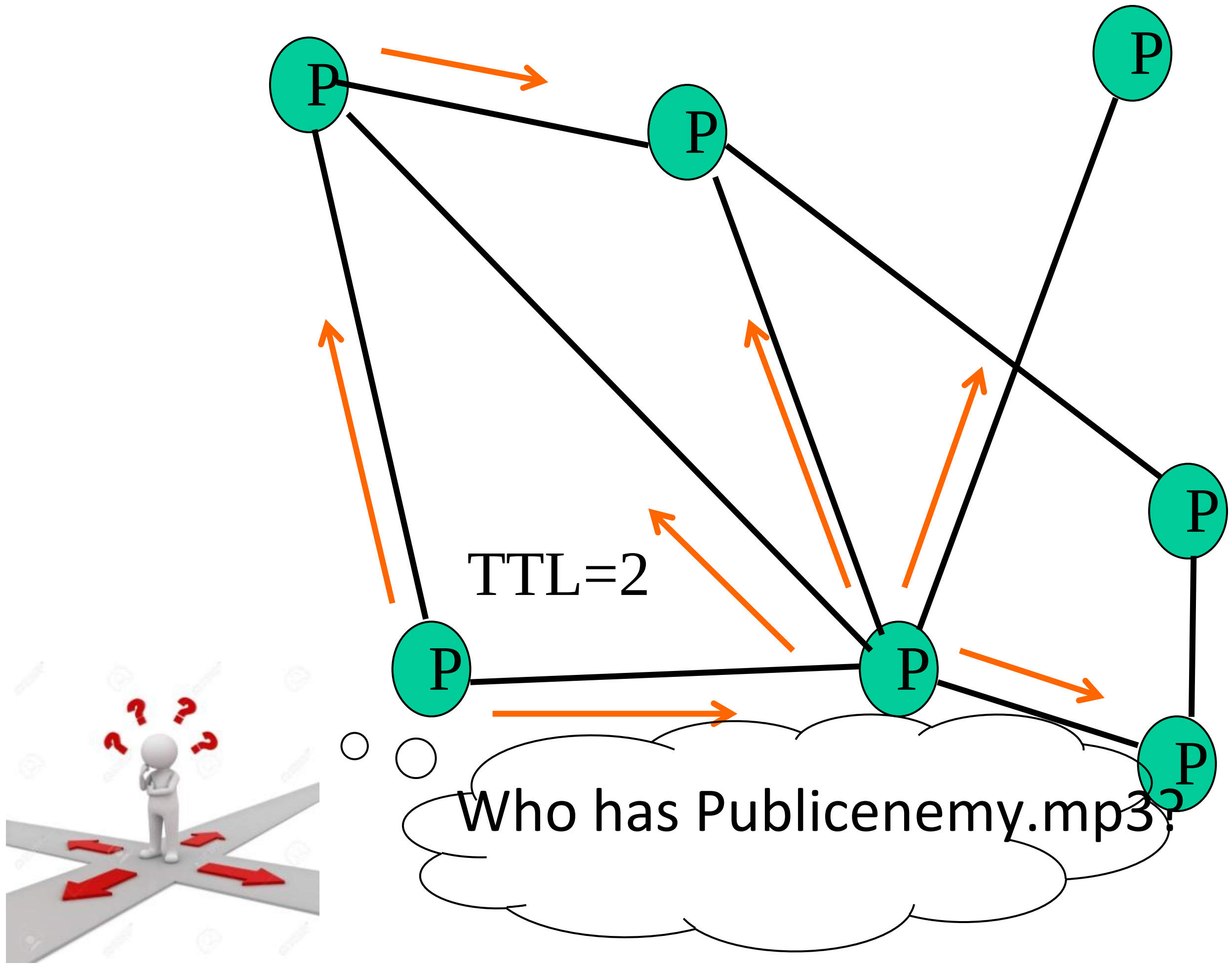
Query (0x80)

Minimum Speed	Search criteria (keywords)
0	1 .....

Payload Format in Gnutella Query Message

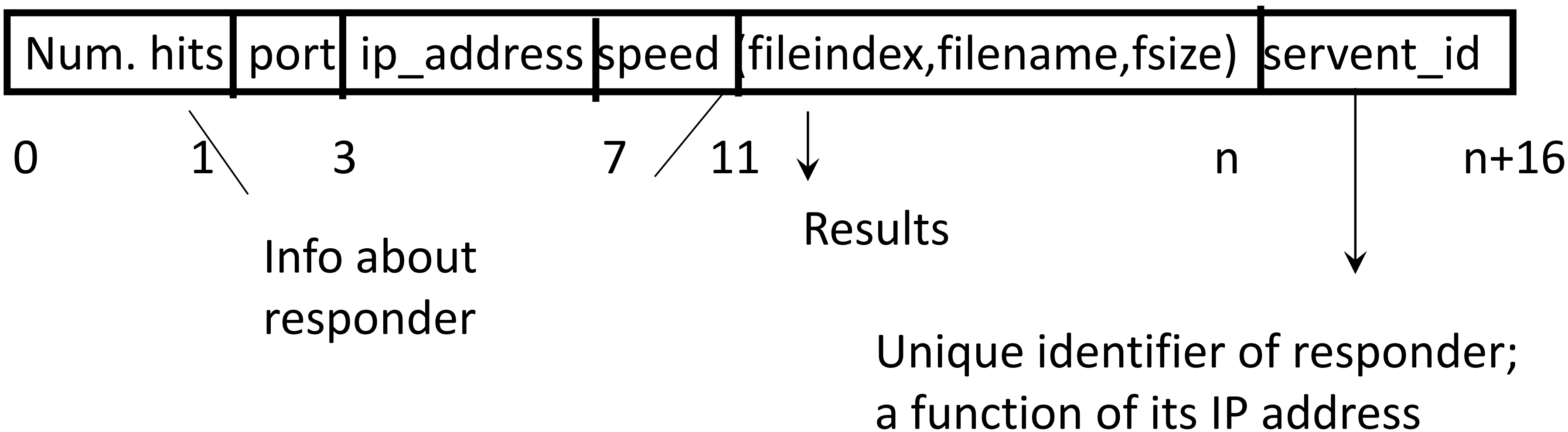
Gnutella Search

Query's flooded out, ttl-restricted, forwarded only once



Gnutella Search

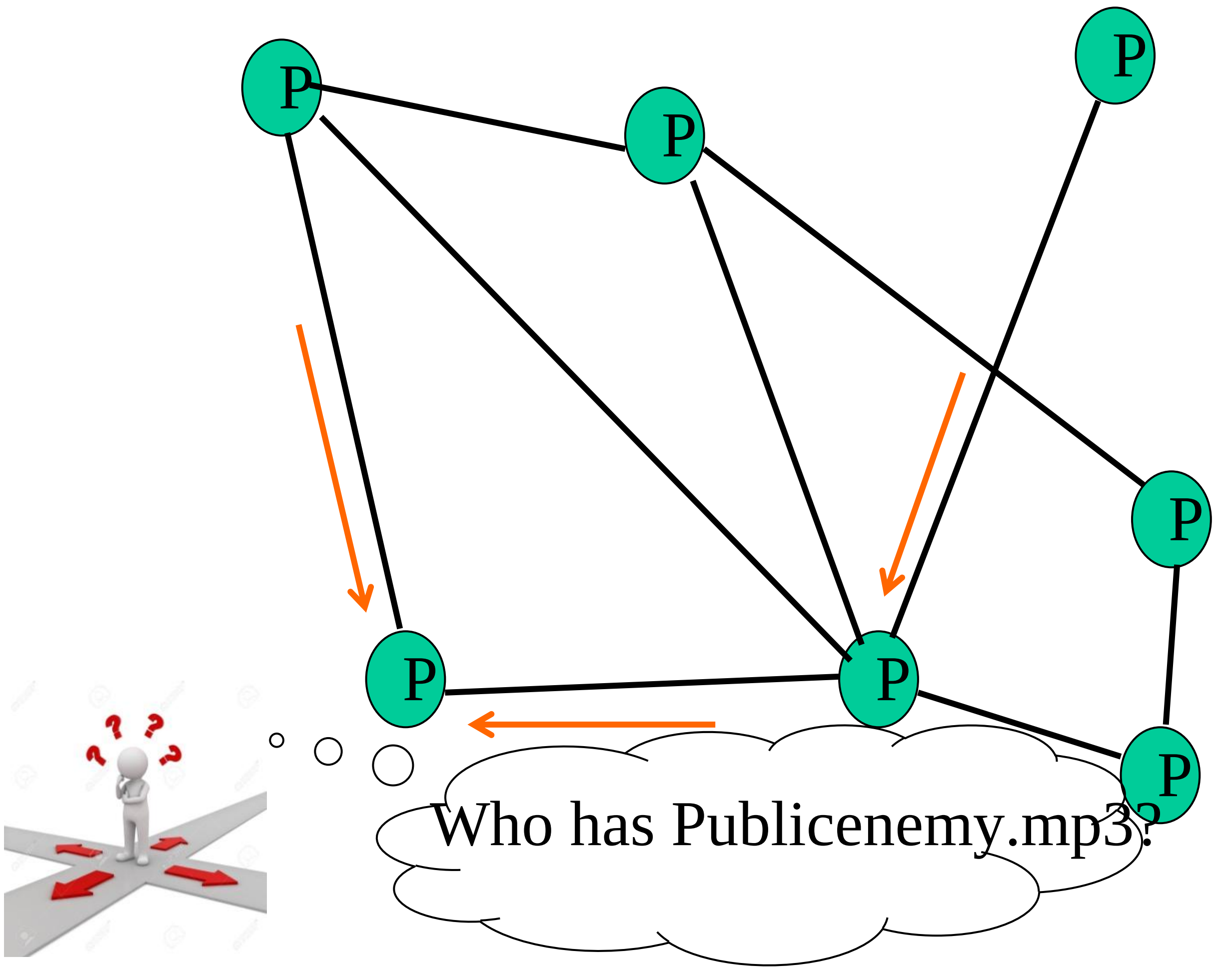
QueryHit **(0x81)** : successful result to a query



Payload Format in Gnutella QueryHit Message

Gnutella Search

Successful results QueryHit's routed on reverse path



Avoiding excessive traffic

- To avoid duplicate transmissions, each peer maintains a list of recently received messages
- Query forwarded to all neighbors except peer from which received
- Each Query **(identified by DescriptorID)** forwarded only once
- **QueryHit** routed back only to peer from which Query received with same DescriptorID
- Duplicates with same DescriptorID and Payload descriptor (msg type, e.g., Query) are dropped
- QueryHit with DescriptorID for which Query not seen is dropped

After receiving QueryHit messages

- **Requestor chooses “best” QueryHit responder**
  - Initiates HTTP request directly to responder’s ip+port

    GET /get/<File Index>/<File Name>/HTTP/1.0\r\n

    Connection: Keep-Alive\r\n

    Range: bytes=0-\r\n

    User-Agent: Gnutella\r\n

    \r\n

- **Responder then replies with file packets after this message:**

    HTTP 200 OK\r\n

    Server: Gnutella\r\n

    Content-type:application/binary\r\n

    Content-length: 1024 \r\n

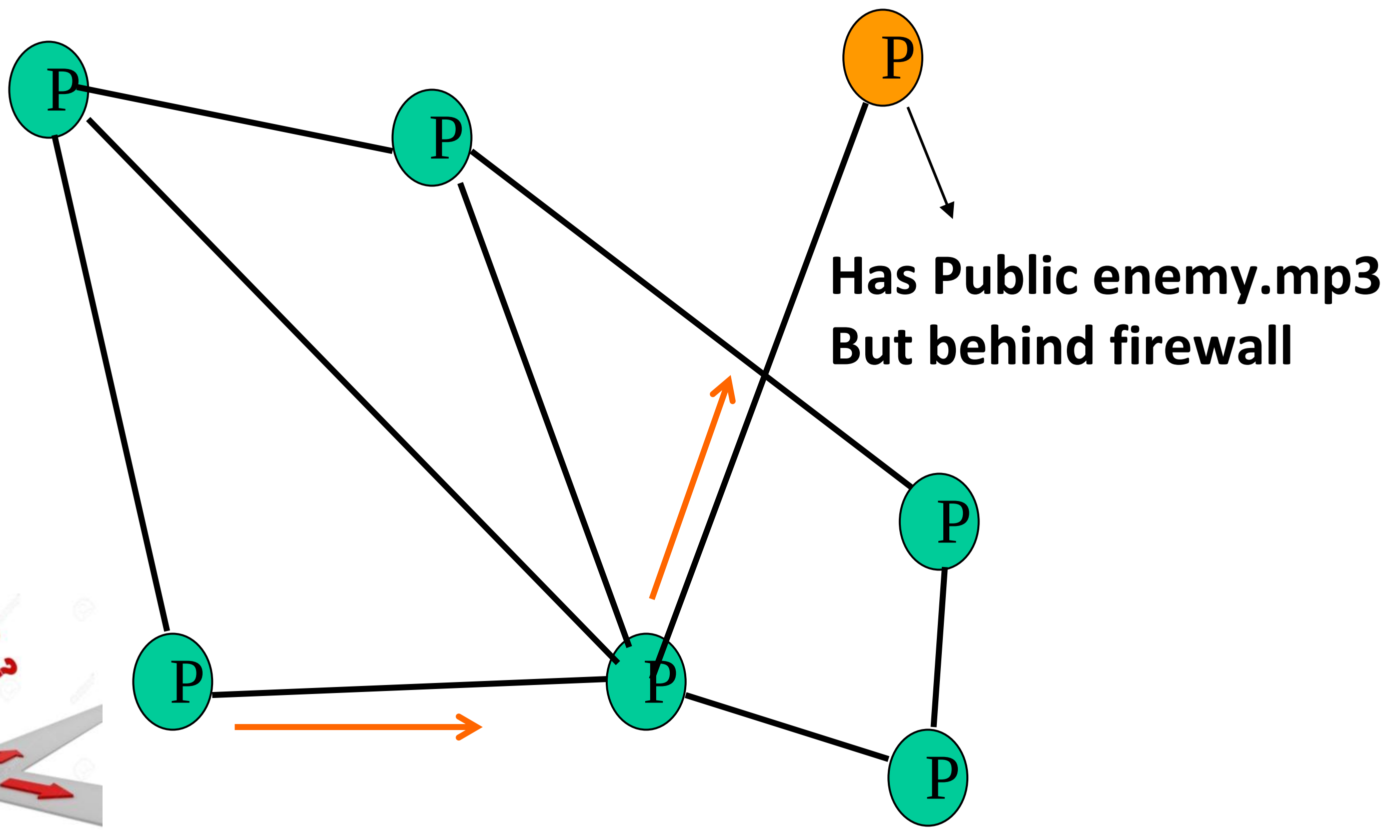
    \r\n

After receiving QueryHit messages (2)

- HTTP is the file transfer protocol. Why?
  - Because it's standard, well-debugged, and widely used.
- Why the "range" field in the GET request?
  - To support partial file transfers.
- What if responder is behind firewall that disallows incoming connections?

Dealing with Firewalls

Requestor sends Push to responder asking for file transfer



Dealing with Firewalls

Push (0x40)

servent_id	fileindex	ip_address	port

same as in
received QueryHit

Address at which
<u>requestor</u> can accept
incoming connections

Dealing with Firewalls

- Responder establishes a TCP connection at ip_address, port specified. Sends

  GIV <File Index>:<Servent Identifier>/<File Name>\n\n

- Requestor then sends GET to responder (as before) and file is transferred as explained earlier

- What if requestor is behind firewall too?
  - Gnutella gives up
  - Can you think of an alternative solution?

Ping-Pong

Ping (0x00)

no payload

Pong (0x01)

Port	ip_address	Num. files shared	Num. KB shared

- Peers initiate Ping's periodically
- Pings flooded out like Querys, Pongs routed along reverse path like QueryHits
- Pong replies used to update set of neighboring peers
  - to keep neighbor lists fresh in spite of peers joining, leaving and failing

Summary: Gnutella

- No servers
- Peers/servents maintain "neighbors", this forms an overlay graph
- Peers store their own files
- Queries flooded out, ttl restricted
- QueryHit (replies) reverse path routed
- Supports file transfer through firewalls
- Periodic Ping-pong to continuously refresh neighbor lists
  - List size specified by user at peer : heterogeneity means some peers may have more neighbors
  - Gnutella found to follow **power law** distribution:

P(#links = L) ~ L^{-k} (k is a constant)

Problems

- Ping/Pong constituted 50% traffic
  - **Solution:** Multiplex, *cache* and reduce frequency of pings/pongs
- Repeated searches with same keywords
  - **Solution:** *Cache* Query, QueryHit messages
- Modem-connected hosts do not have enough bandwidth for passing Gnutella traffic
  - **Solution:** use a central server to act as proxy for such peers
  - **Another solution:**

    ➔FastTrack System

Problems (Contd...)

- Large number of ***freeloaders***
  - 70% of users in 2000 were freeloaders
  - Only download files, never upload own files

- Flooding causes excessive traffic
  - Is there some way of maintaining meta-information about peers that leads to more intelligent routing?

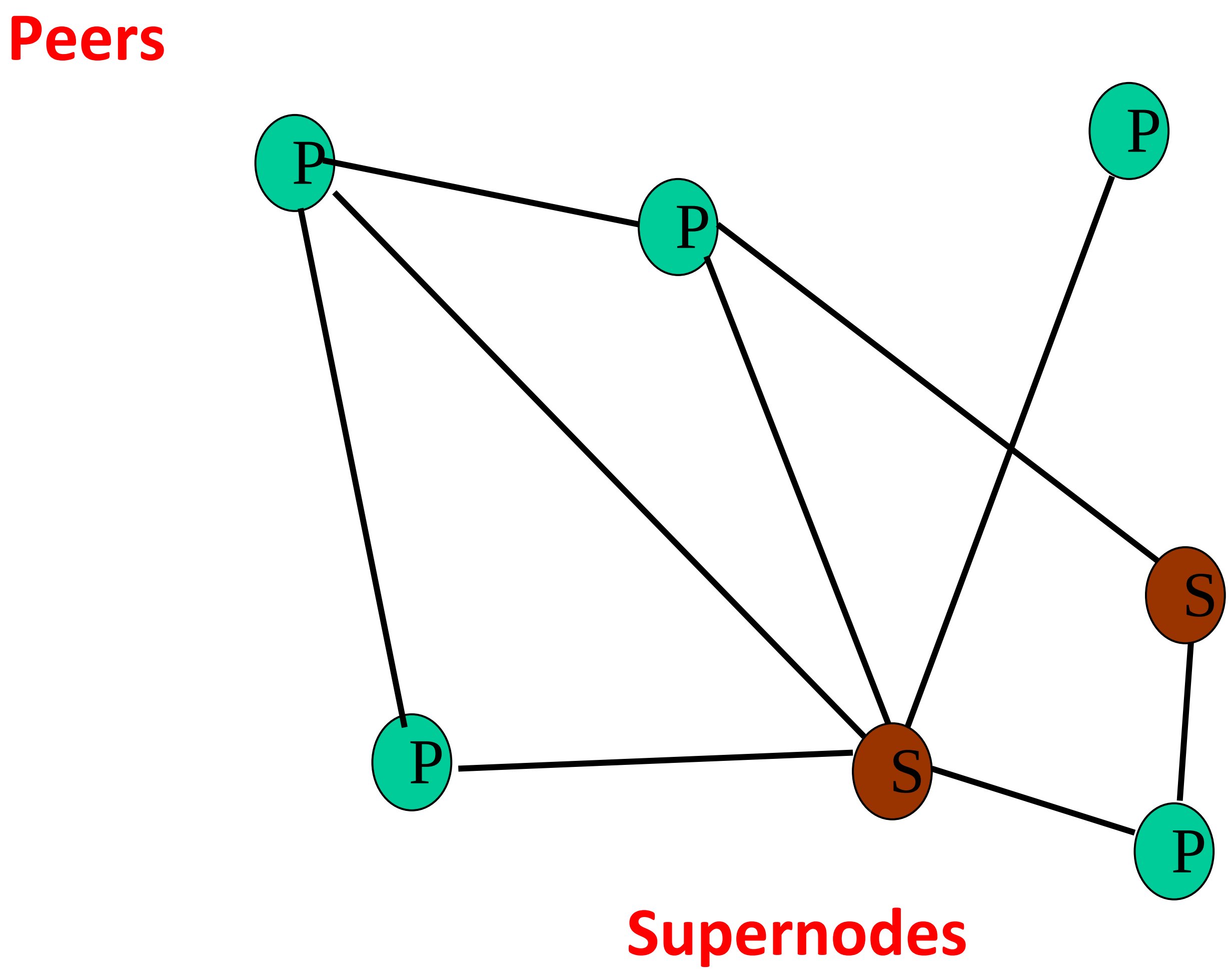
    ➔ Structured Peer-to-peer systems

    Example: Chord System

FastTrack

- Hybrid between Gnutella and Napster
- Takes advantage of "healthier" participants in the system
- Underlying technology in Kazaa, KazaaLite, Grokster
- Proprietary protocol, but some details available
- Like Gnutella, but with some peers designated as ***supernodes***

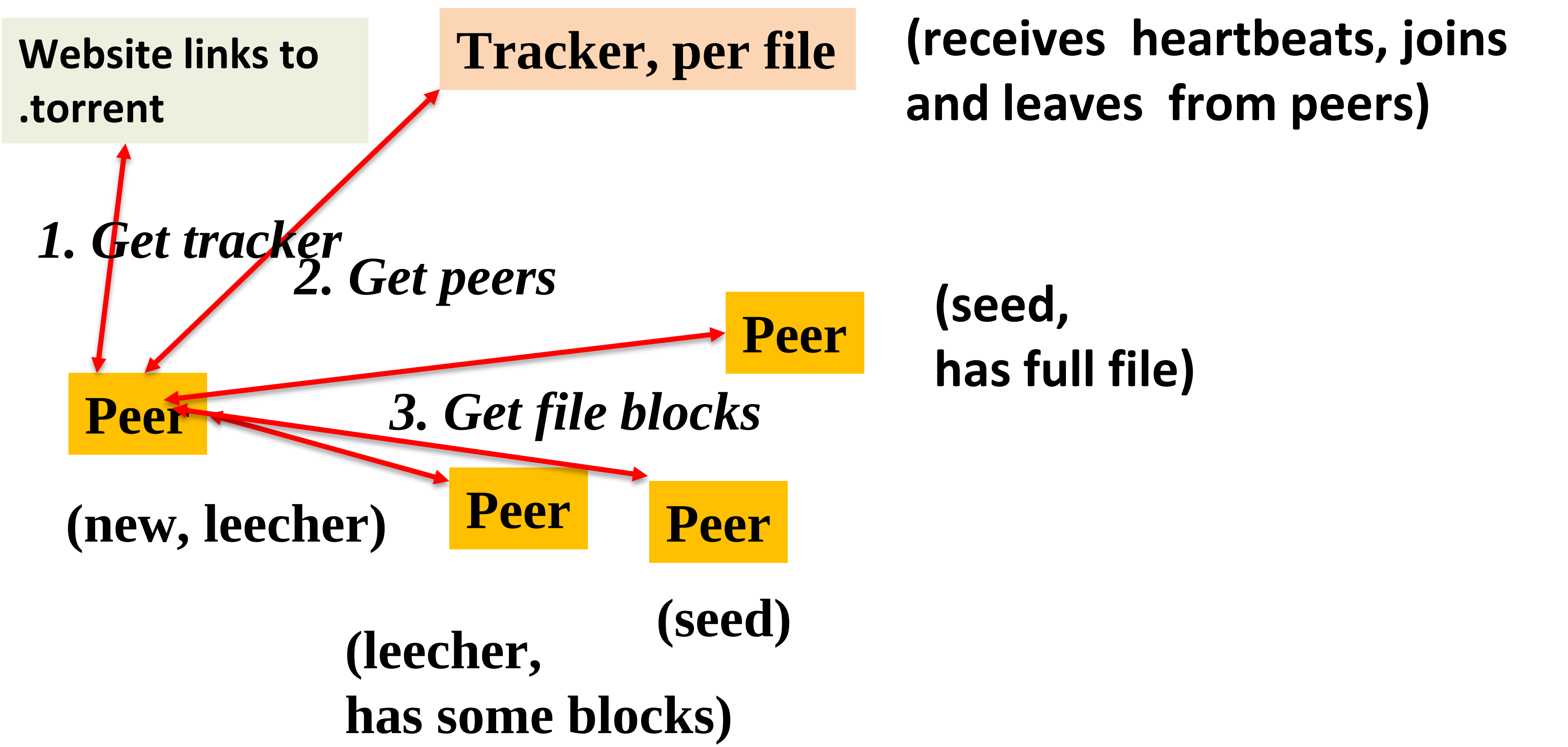
A FastTrack-like System



FastTrack (Contd...)

- A supernode stores a directory listing a subset of nearby (<filename,peer pointer>), similar to Napster servers
- Supernode membership changes over time
- Any peer can become (and stay) a supernode, provided it has earned enough ***reputation***
  - **Kazaalite:** participation level (=reputation) of a user between 0 and 1000, initially 10, then affected by length of periods of connectivity and total number of uploads
  - More sophisticated Reputation schemes invented, especially based on economics
- A peer searches by contacting a nearby supernode

BitTorrent



Website links to .torrent
Tracker, per file
(receives heartbeats, joins and leaves from peers)
1. Get tracker
2. Get peers
3. Get file blocks
Peer
(new, leecher)
Peer
(seed, has full file)
Peer
(leecher, has some blocks)
Peer
(seed)

BitTorrent (2)

- File split into blocks (32 KB – 256 KB)
- Download **Local Rarest First** block policy: prefer early download of blocks that are least replicated among neighbors
  - Exception: New node allowed to pick one random neighbor: helps in bootstrapping
- **Tit for tat** bandwidth usage: Provide blocks to neighbors that provided it the best download rates
  - Incentive for nodes to provide good download rates
  - Seeds do the same too
- **Choking:** Limit number of neighbors to which concurrent uploads <= a number (5), i.e., the "best" neighbors
  - Everyone else choked
  - Periodically re-evaluate this set (e.g., every 10 s)
  - **Optimistic unchoke:** periodically (e.g., ~30 s), unchoke a random neigbhor – helps keep unchoked set fresh

DHT (Distributed Hash Table)

- A **hash table** allows you to insert, lookup and delete objects with keys
- A ***distributed* hash table** allows you to do the same in a distributed setting (objects=files)
- **Performance Concerns:**
  - Load balancing
  - Fault-tolerance
  - Efficiency of lookups and inserts
  - Locality
- **Napster, Gnutella, FastTrack are all DHTs** (sort of)
- So is **Chord, a structured peer to peer system**

Comparative Performance

	Memory	Lookup Latency	#Messages for a lookup
Napster	O(1) (O(N)@server)	O(1)	O(1)
Gnutella	O(N)	O(N)	O(N)
Chord	**O(log(N))**	**O(log(N))**	**O(log(N))**

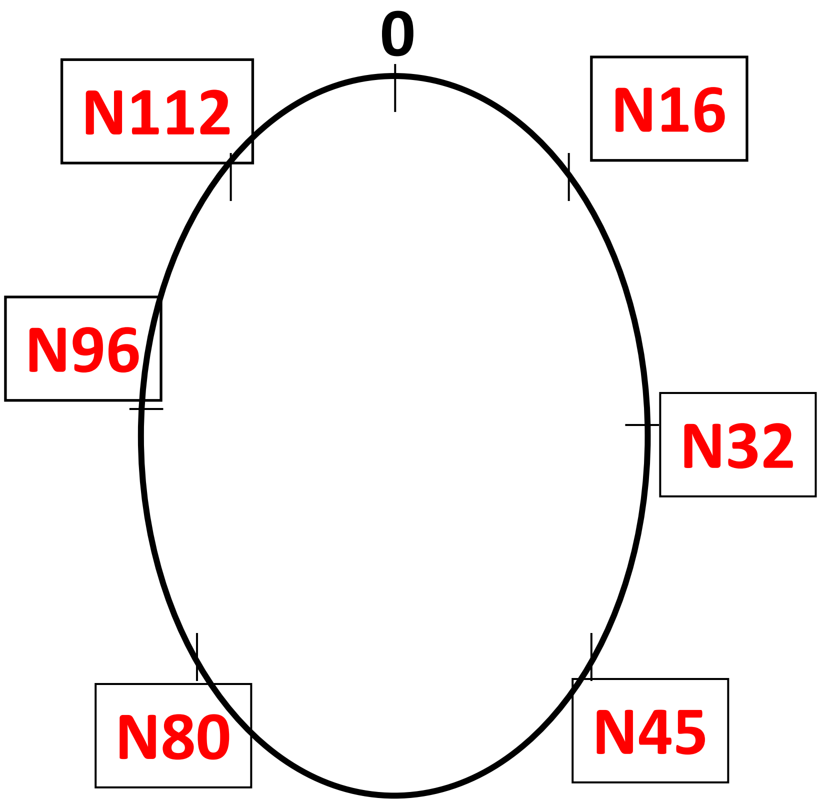
Chord

- **Developers:** I. Stoica, D. Karger, F. Kaashoek, H. Balakrishnan, R. Morris, Berkeley and MIT
- Intelligent choice of neighbors to reduce latency and message cost of routing (lookups/inserts)
- Uses ***Consistent Hashing*** on node's (peer's) address
  - **SHA-1**(ip_address,port) →160 bit string
  - Truncated to *m* bits
  - Called peer *id* (number between 0 and $2^m - 1$)
  - Not unique but id conflicts very unlikely
  - Can then map peers to one of 2^m logical points on a circle

Ring of peers

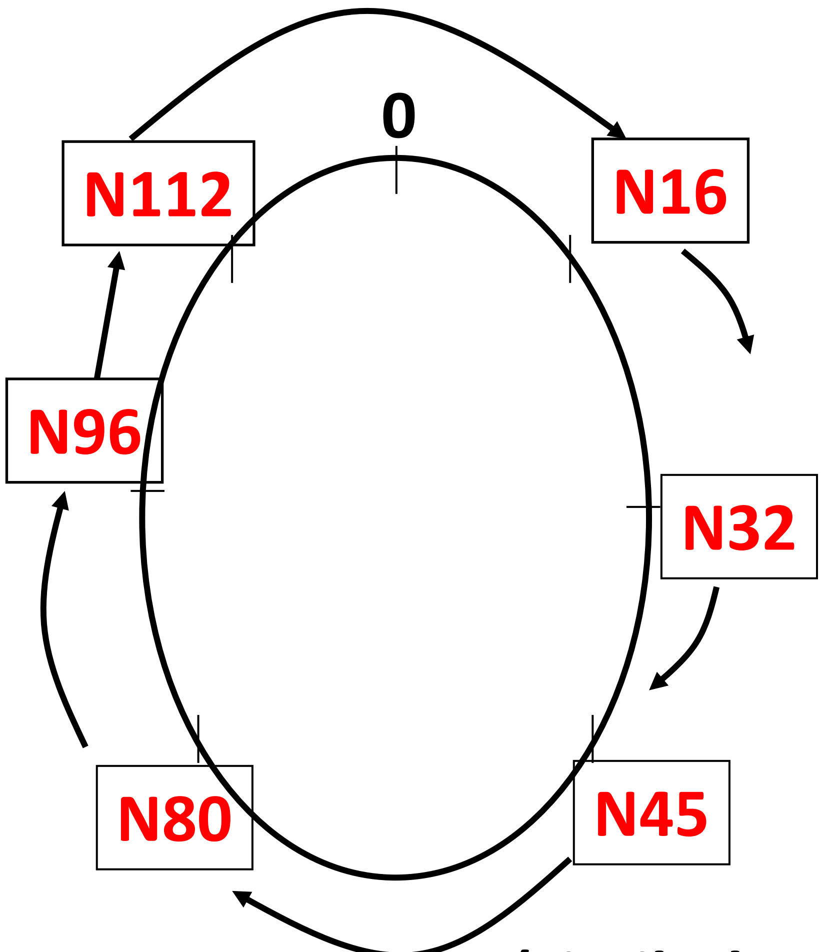
Say *m=7*

6 nodes



Peer Pointers (1): Successors

Say *m=7*



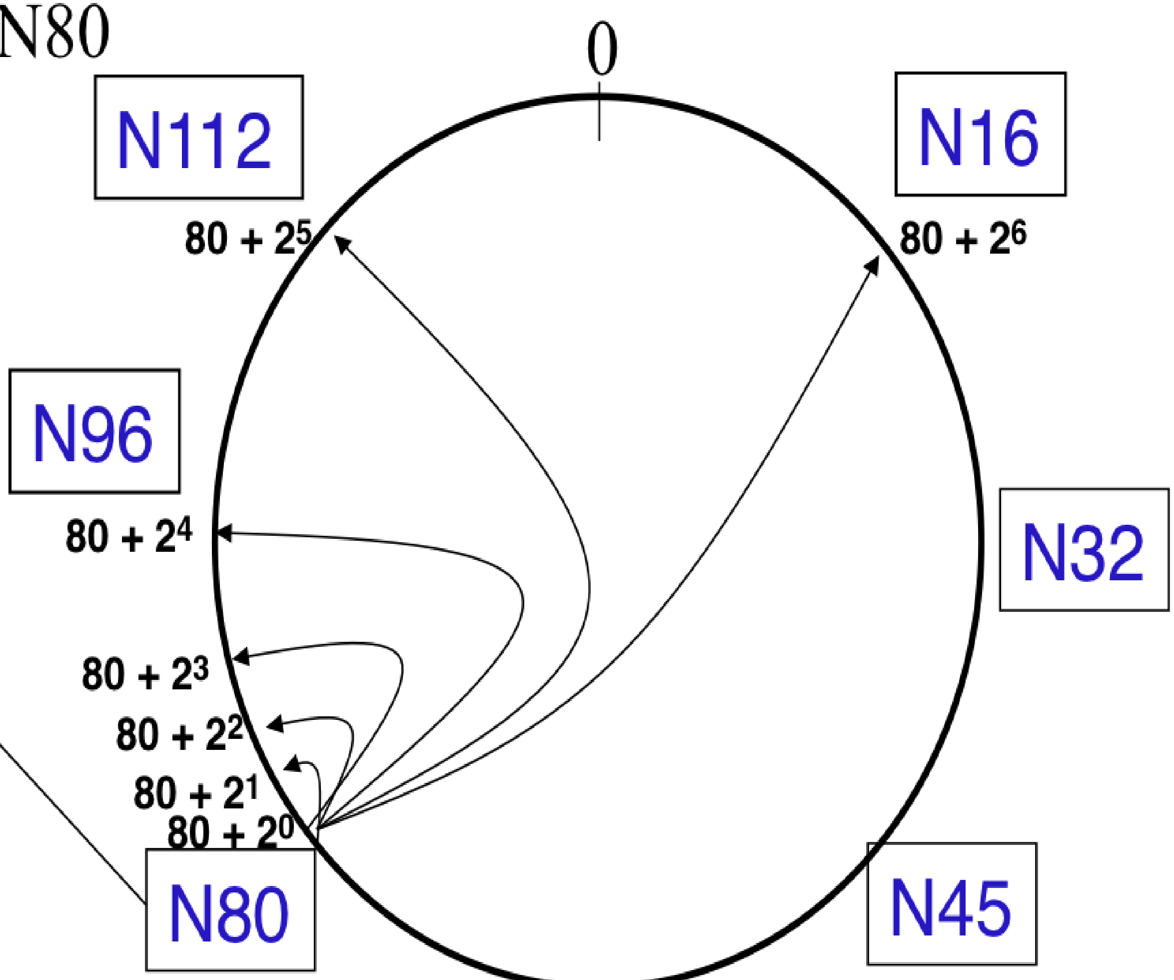
(similarly predecessors)

Peer Pointers (2): finger tables

Say m=7

Finger Table at N80

i	$ft[i]$
0	96
1	96
2	96
3	96
4	96
5	112
6	16



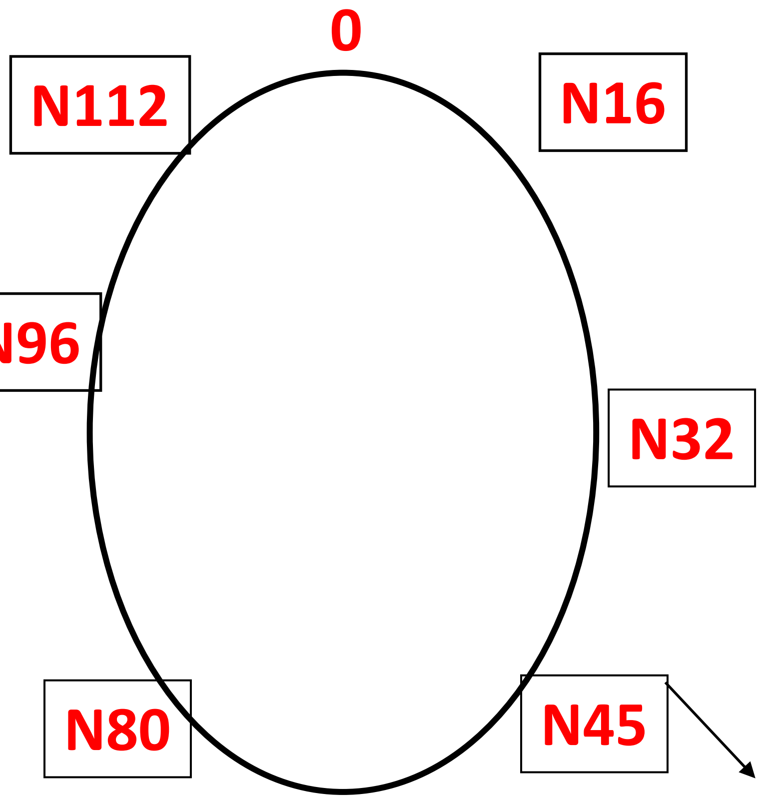
ith entry at peer with id n is first peer with id >= $n + 2^i (\bmod 2^m)$

What about the files?

- Filenames also mapped using same consistent hash function
  - SHA-1(filename) →160 bit string (*key*)
  - File is stored at **first peer with id greater than or equal to its key (mod 2^m)**
- File *cnn.com/index.html* that maps to key K42 is stored at first peer with id greater than 42
  - Note that we are considering a different file-sharing application here : *cooperative web caching*
  - The same discussion applies to any other file sharing application, including that of mp3 files.
- Consistent Hashing => with K keys and N peers, each peer stores *O(K/N)* keys. (i.e., < *c.K/N*, for some constant *c*)

Mapping Files

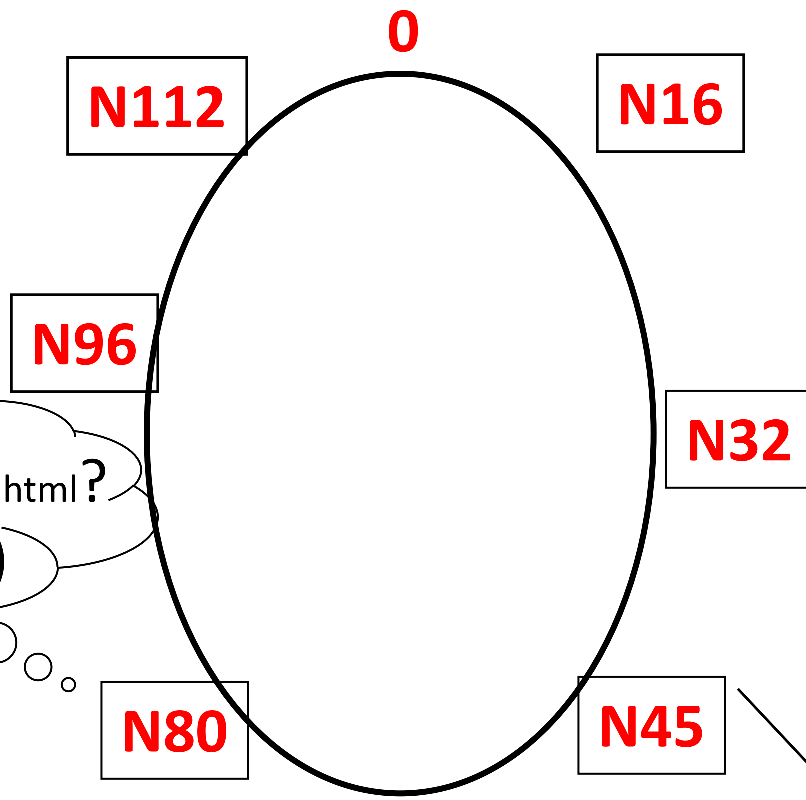
Say *m=7*



File with key K42 stored here

Search

Say *m=7*



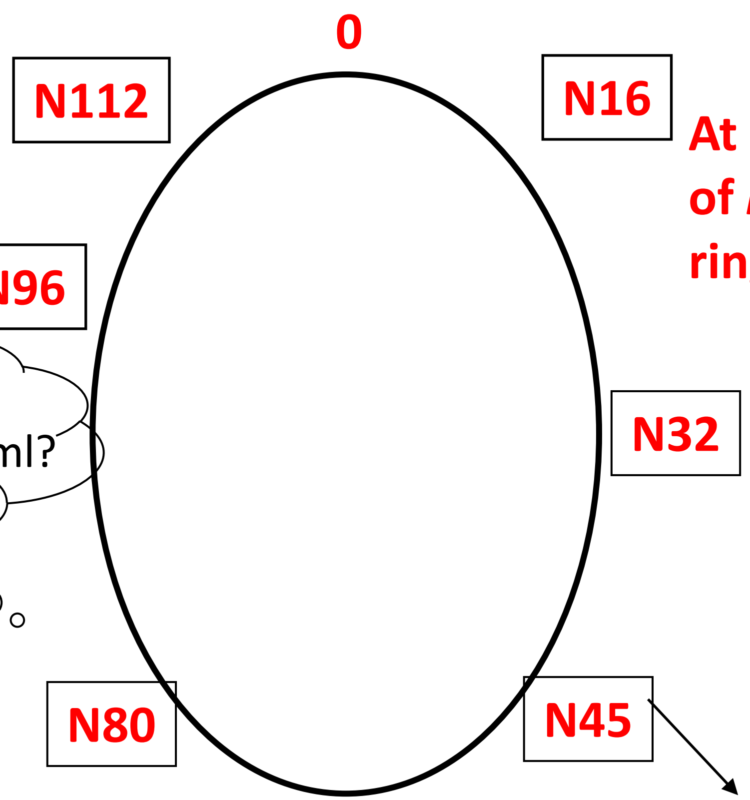
Who has cnn.com/index.html?
(hashes to K42)

File cnn.com/index.html with key **K42** stored here

Search

At node *n*, send query for key *k* to largest successor/finger entry <= *k*
if none exist, send query to *successor(n)*

Say *m=7*



At or to the anti-clockwise of *k* (it wraps around the ring)

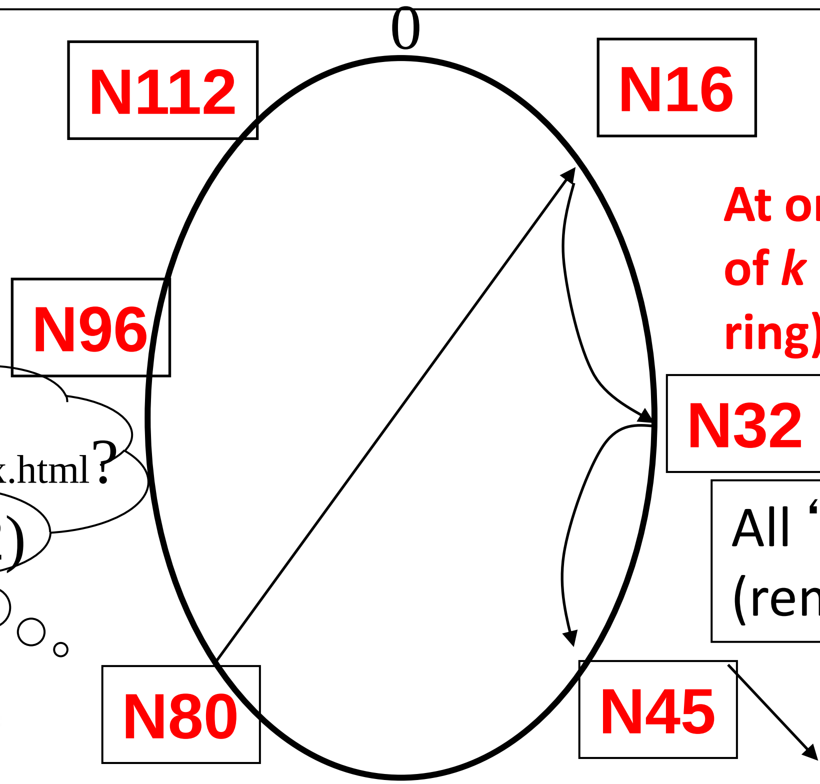
Who has cnn.com/index.html?
(hashes to K42)

File cnn.com/index.html with key **K42** stored here

Search

At node *n*, send query for key *k* to largest successor/finger entry <= *k*
if none exist, send query to *successor(n)*

Say *m=7*



At or to the anti-clockwise of *k* (it wraps around the ring)

All "arrows" are RPCs (remote procedure calls)

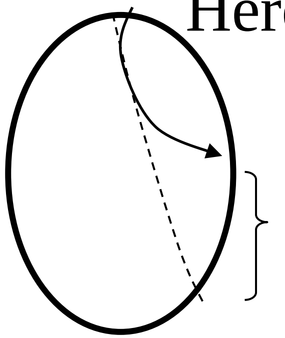
Who has cnn.com/index.html?
(hashes to K42)

File cnn.com/index.html with key K42 stored here

Analysis

Search takes *O(log(N))* time

Proof :



- (intuition): ***at each step, distance between query and peer-with-file reduces by a factor of at least 2***
- (intuition): after *log(N)* forwardings, distance to key is at most
- Number of node identifiers in a range of $2^m / 2^{\log(N)} = 2^m / N$ is *O(log(N))* with high probability (why? SHA-1! and "Balls and Bins")

So using *successors* in that range will be ok, using another *O(log(N))* hops

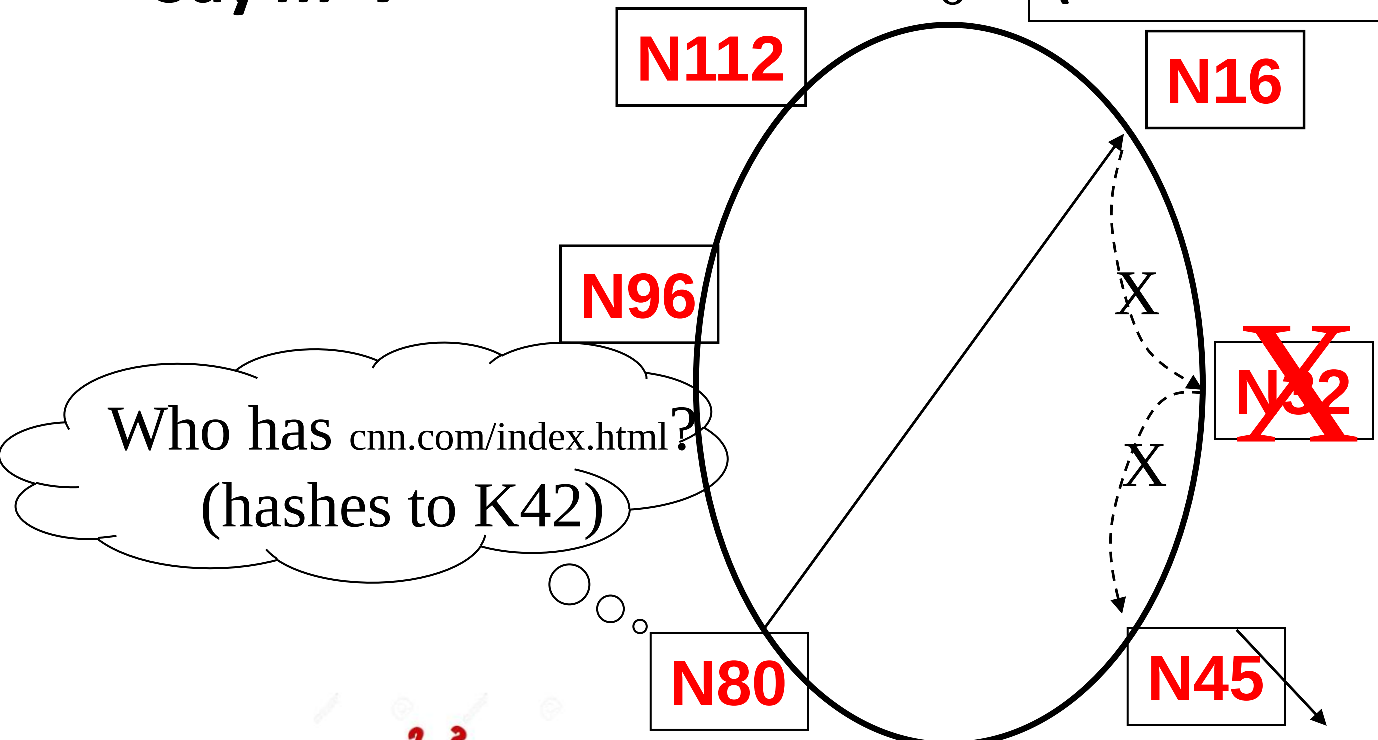
Analysis (Contd.)

- *O(log(N))* search time holds for file insertions too (in general for ***routing*** *to any key*)
  - "Routing" can thus be used as a **building block** for
    - All operations: insert, lookup, delete
- *O(log(N))* time true only if finger and successor entries correct
- When might these entries be wrong?
  - When you have failures

Search under peer failures

Lookup fails
(N16 does not know N45)

Say *m=7*

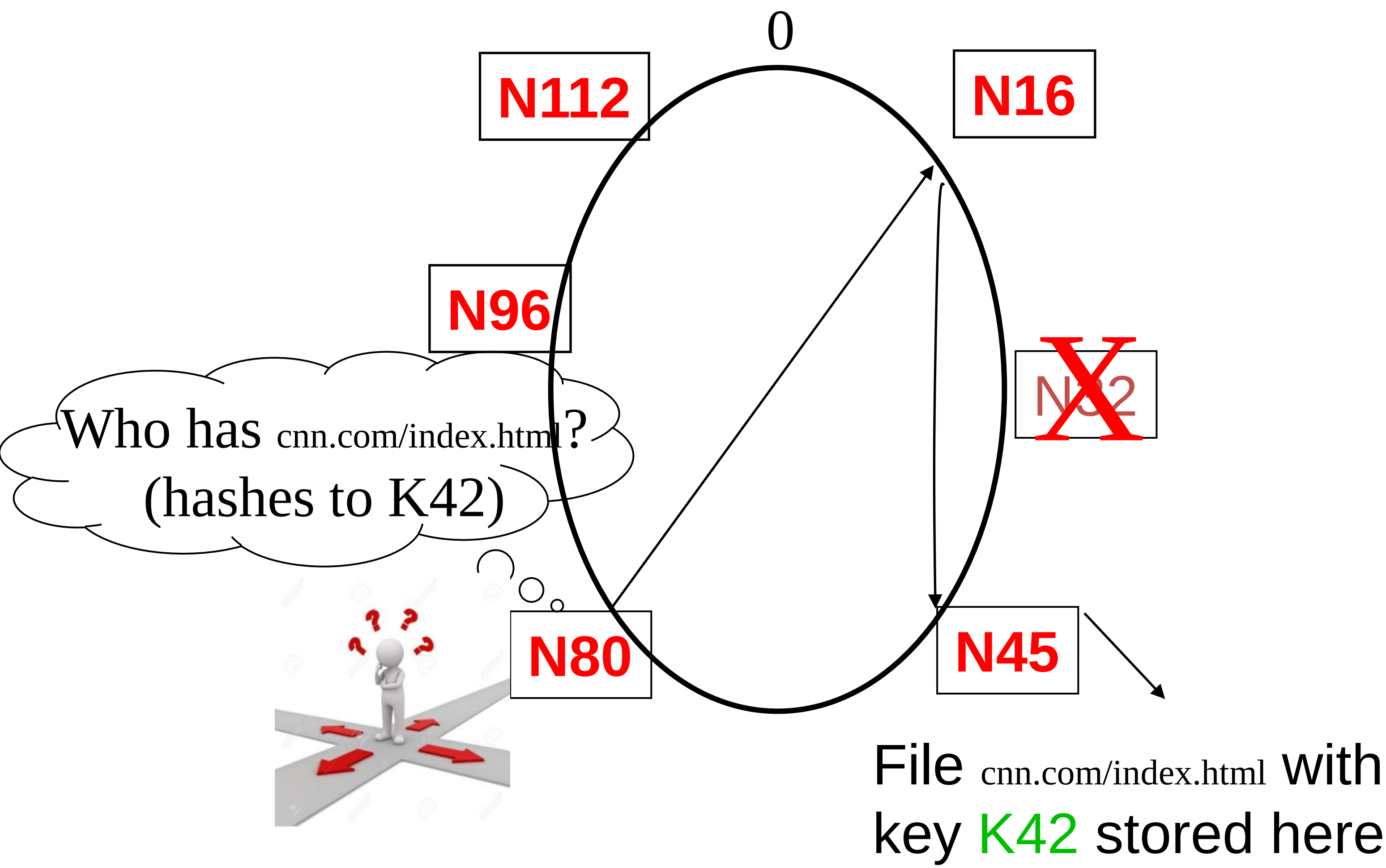


File cnn.com/index.html with
key K42 stored here

Search under peer failures

One solution: maintain *r* multiple *successor* entries
In case of failure, use successor entries

Say *m=7*



Search under peer failures

- Choosing *r=2log(N)* suffices to maintain *lookup correctness* with high probability (i.e., ring connected)
  - Say 50% of nodes fail
  - **Pr(at given node, at least one successor alive)=**

$$1-(\frac{1}{2})^{2\log N}=1-\frac{1}{N^2}$$

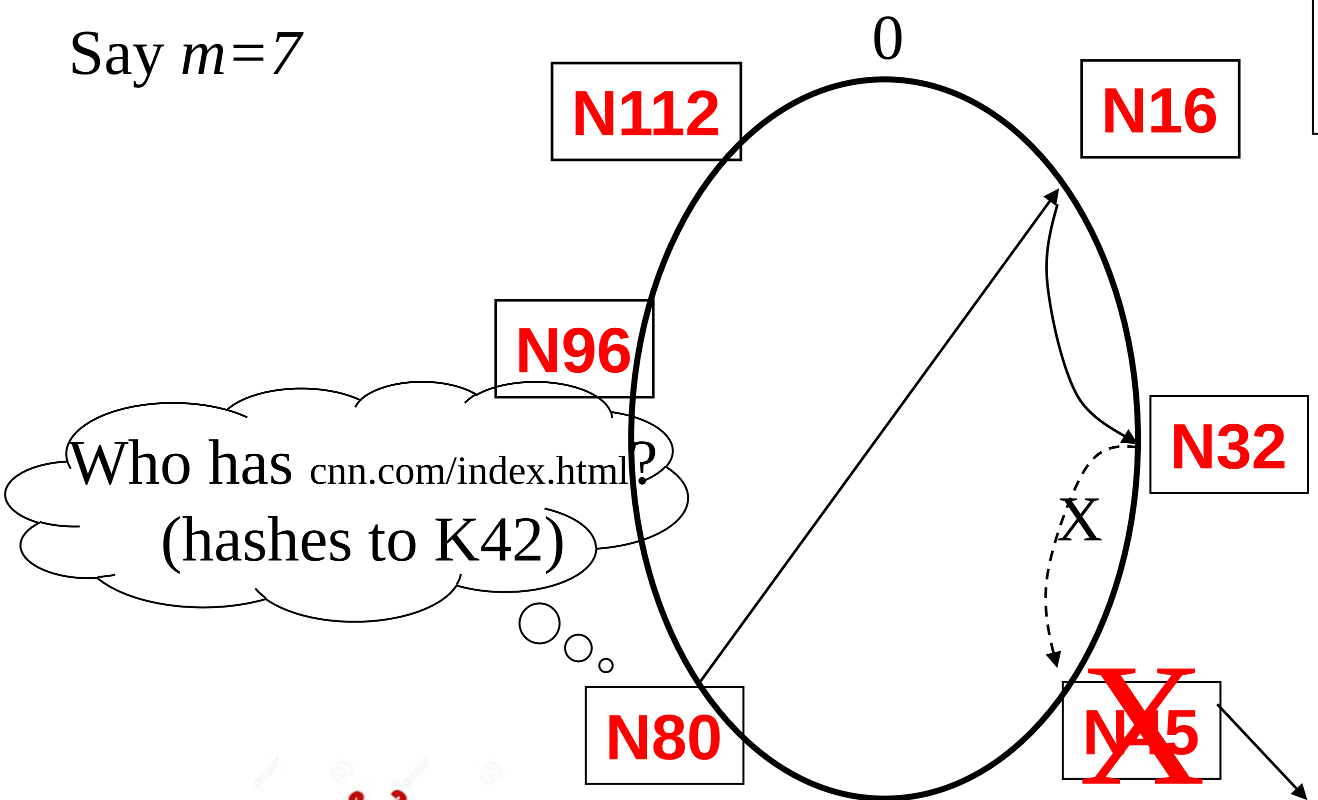
  - **Pr(above is true at all alive nodes)=**

$$(1-\frac{1}{N^2})^{N/2}=e^{-\frac{1}{2N}}\approx 1$$

Search under peer failures (2)

Say *m*=7

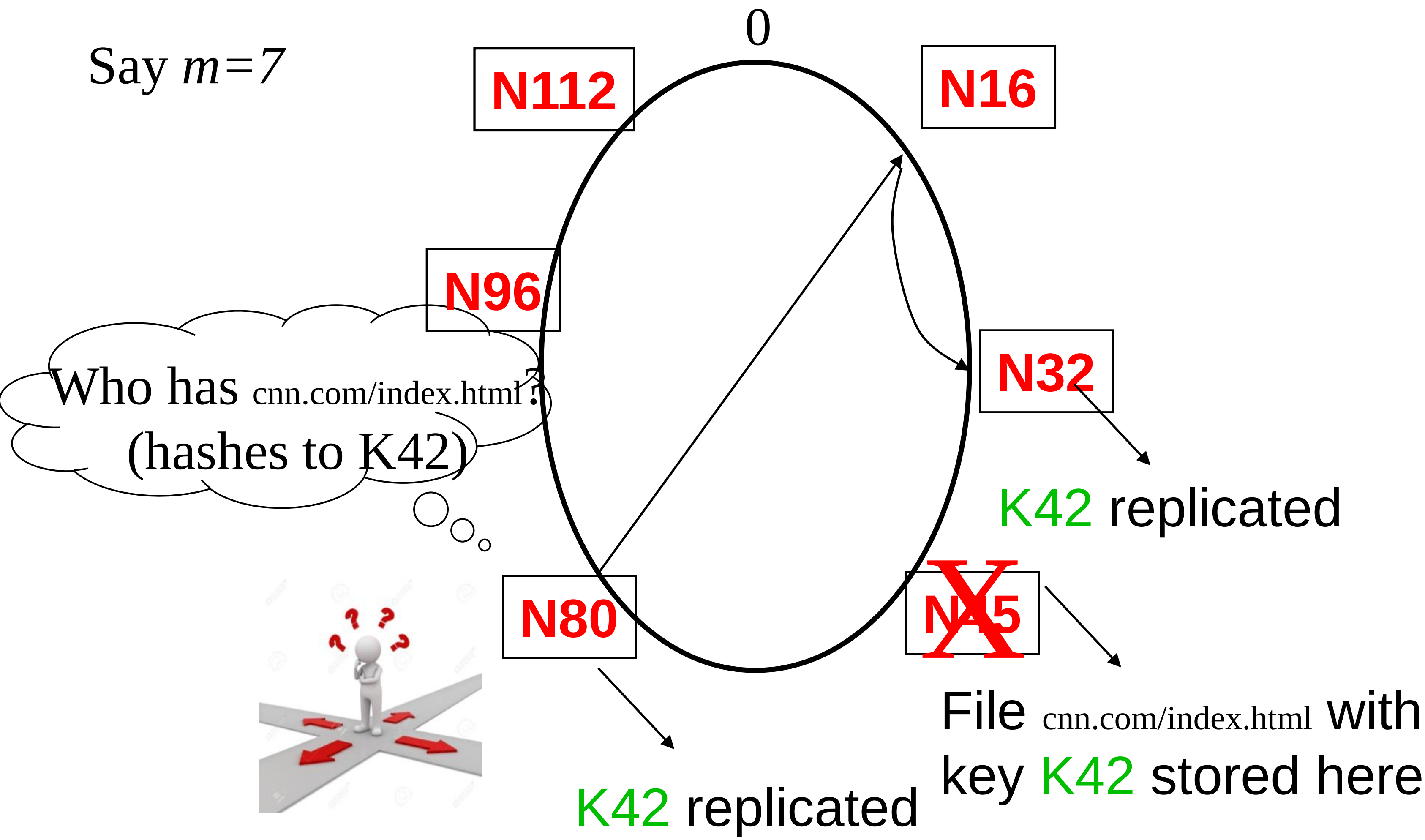
Lookup fails
(N45 is dead)



File cnn.com/index.html with
key K42 stored here

Search under peer failures (2)

One solution: replicate file/key at *r* successors and predecessors



Need to deal with dynamic changes

- ✓ Peers fail
- New peers join
- Peers leave
  - P2P systems have a high rate of ***churn*** (node join, leave and failure)
    - 25% per hour in Overnet (eDonkey)
    - 100% per hour in Gnutella
    - Lower in managed clusters
    - Common feature in all distributed systems, including wide-area (e.g., PlanetLab), clusters (e.g., Emulab), clouds (e.g., AWS), etc.

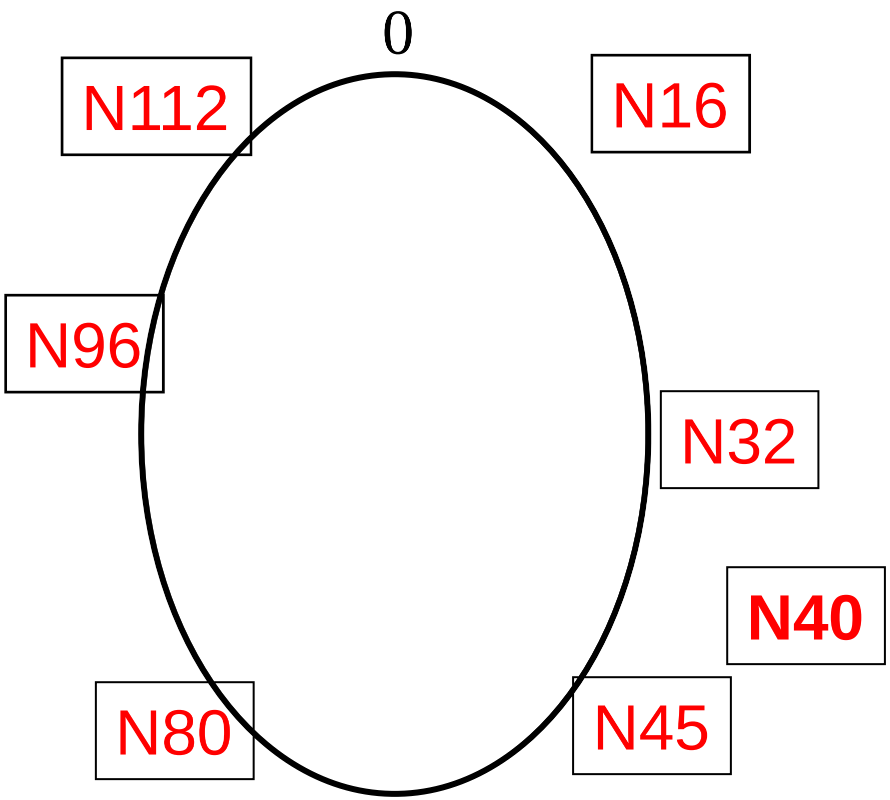
So, all the time, need to:

→ Need to update *successors* and *fingers*, and copy keys

New peers joining

Introducer directs N40 to N45 (and N32)
N32 updates successor to N40
N40 initializes successor to N45, and inits fingers from it
N40 periodically talks to neighbors to update finger table

Say *m=7*

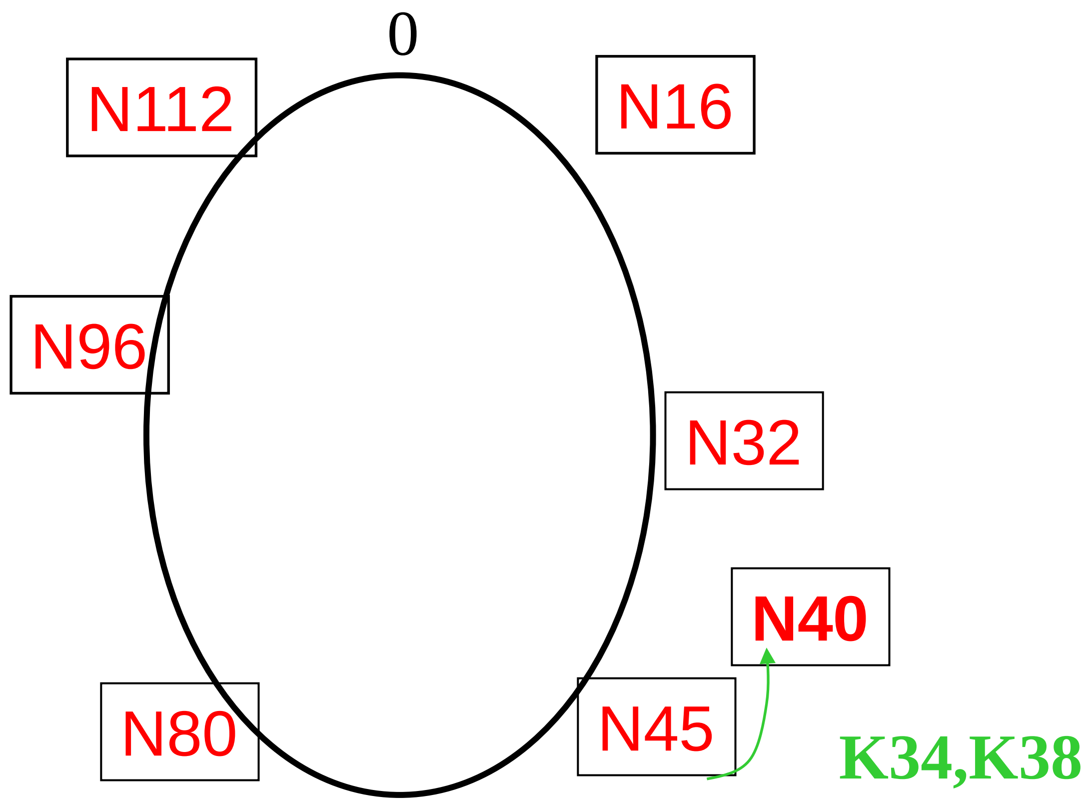


Stabilization Protocol (followed by all nodes)

New peers joining (2)

N40 may need to copy some files/keys from N45 (files with file id between 32 and 40)

Say *m=7*



New peers joining (3)

- A new peer affects *O(log(N))* other finger entries in the system, on average [Why?]

- Number of messages per peer join= *O(log(N)*log(N))*

- Similar set of operations for dealing with peers leaving
  - For dealing with failures, also need ***failure detectors***.

Stabilization Protocol

- Concurrent peer joins, leaves, failures might cause loopiness of pointers, and failure of lookups
  - Chord peers periodically run a *stabilization* algorithm that checks and updates pointers and keys
  - Ensures *non-loopiness* of fingers, eventual success of lookups and *O(log(N))* lookups with high probability
  - Each stabilization round at a peer involves a constant number of messages
  - Strong stability takes $O(N^2)$ stabilization rounds

Churn

- **When nodes are constantly joining, leaving, failing**
  - Significant effect to consider: traces from the Overnet system show *hourly* peer turnover rates (***churn***) could be *25-100%* of total number of nodes in system
  - Leads to excessive (unnecessary) key copying (remember that keys are replicated)
  - Stabilization algorithm may need to consume more bandwidth to keep up
  - Main issue is that files are replicated, while it might be sufficient to replicate only meta information about files
  - **Alternatives**
    - Introduce a level of indirection, i.e., store only pointers to files (any p2p system)
    - Replicate metadata more, e.g., Kelips

Virtual Nodes

- Hash can get non-uniform ➔ Bad load balancing
  - Treat each node as multiple virtual nodes behaving independently
  - Each joins the system
  - Reduces variance of load imbalance

Remarks

- Virtual Ring and Consistent Hashing used in Cassandra, Riak, Voldemort, DynamoDB, and other key-value stores

- **Current status of Chord project:**
  - File systems (CFS,Ivy) built on top of Chord
  - DNS lookup service built on top of Chord
  - Internet Indirection Infrastructure (I3) project at UCB
  - Spawned research on many interesting issues about p2p systems

https://github.com/sit/dht/wiki

Pastry

- **Designed by Anthony Rowstron (Microsoft Research) and Peter Druschel (Rice University)**

- Assigns ids to nodes, just like Chord
  (using a virtual ring)

- **Leaf Set -** Each node knows its successor(s) and predecessor(s)

Pastry Neighbors

- **Routing tables** based on **prefix matching**
  - Think of a hypercube
- Routing is thus based on prefix matching, and is thus log(N)
  - And hops are short (in the underlying network)

Pastry Routing

- Consider a peer with id 01110100101. It maintains a neighbor peer with an id matching each of the following prefixes (* = starting bit differing from this peer's corresponding bit):
  - *
  - 0*
  - 01*
  - 011*
  - … 0111010010*
- When it needs to route to a peer, say 01110111001, it starts by forwarding to a neighbor with the largest matching prefix, i.e., 011101*

Pastry Locality

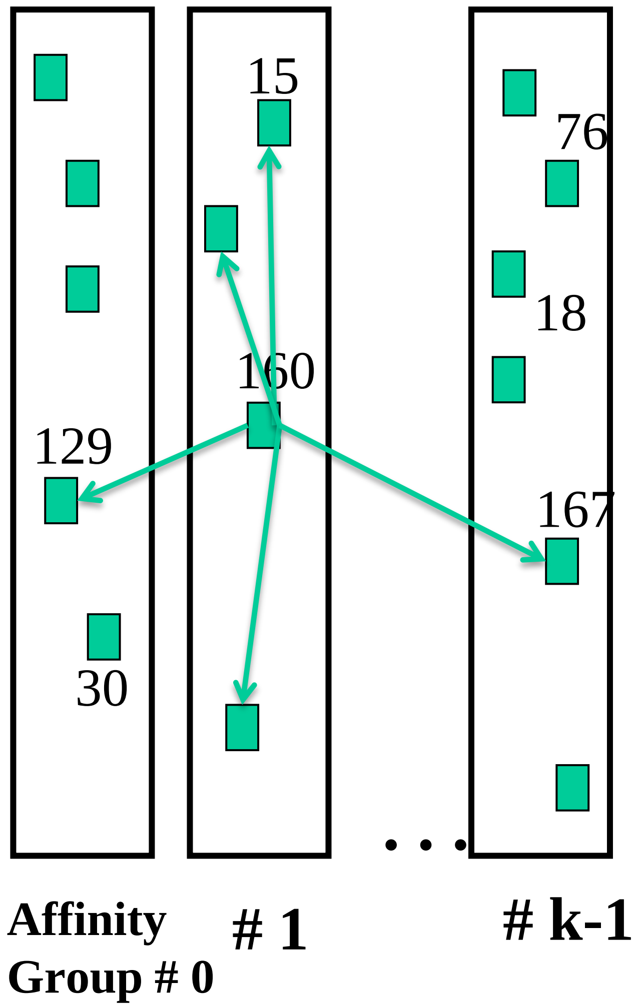
- For each prefix, say 011*, among all potential neighbors with the matching prefix, the neighbor with the shortest round-trip-time is selected
- Since shorter prefixes have many more candidates (spread out throughout the Internet), the neighbors for shorter prefixes are likely to be closer than the neighbors for longer prefixes
- Thus, in the prefix routing, early hops are short and later hops are longer
- Yet overall “stretch”, compared to direct Internet path, stays short

Summary: Chord and Pastry

- **Chord and Pastry protocols:**
  - More structured than Gnutella
  - Black box lookup algorithms
  - Churn handling can get complex
  - ***O(log(N))*** memory and lookup cost
    - ***O(log(N))*** lookup hops may be high
    - Can we reduce the number of hops?

Kelips : A 1 hop Lookup DHT

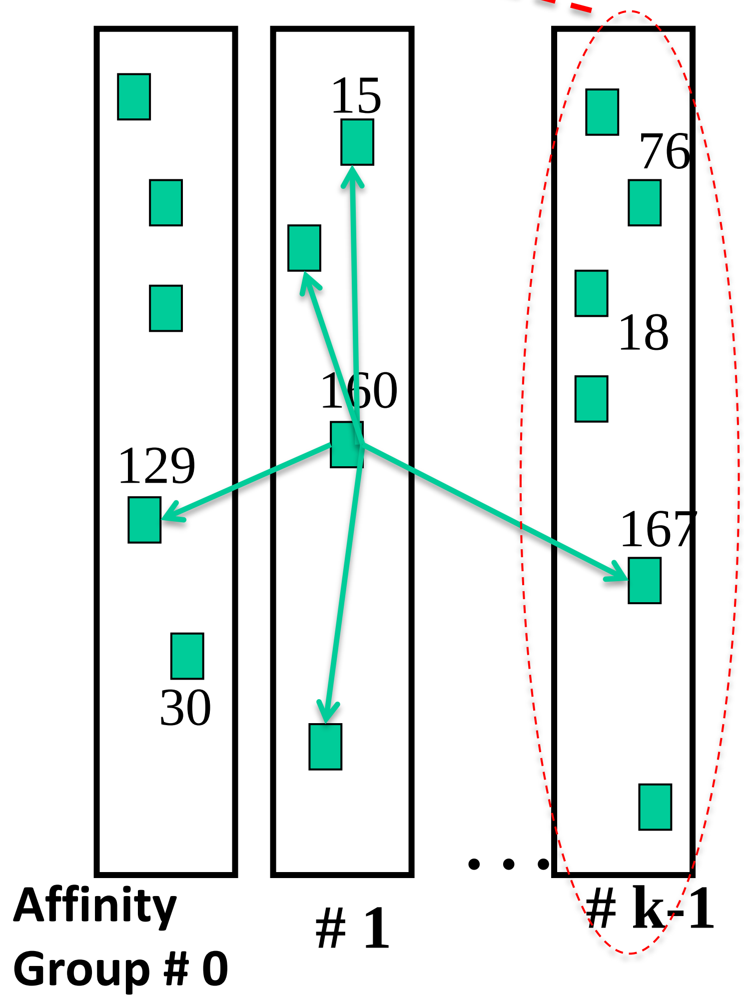
- k “affinity groups”
  - k ~ √ N
- Each node hashed to a group (hash mod k)
- Node’s neighbors
  - (Almost) all other nodes in its own affinity group
  - One contact node per foreign affinity group



Kelips Files and Metadata

- File can be stored at any (few) node(s)
- Decouple file replication/location (outside Kelips) from file querying (in Kelips)
- Each filename hashed to a group
  - All nodes in the group replicate pointer information, i.e., <filename, file location>
  - Spread using gossip
  - Affinity group **does not** store files

- Publicenemy.mp3 hashes to k-1
- Everyone in this group stores <Publicenemy.mp3, who-has-file>

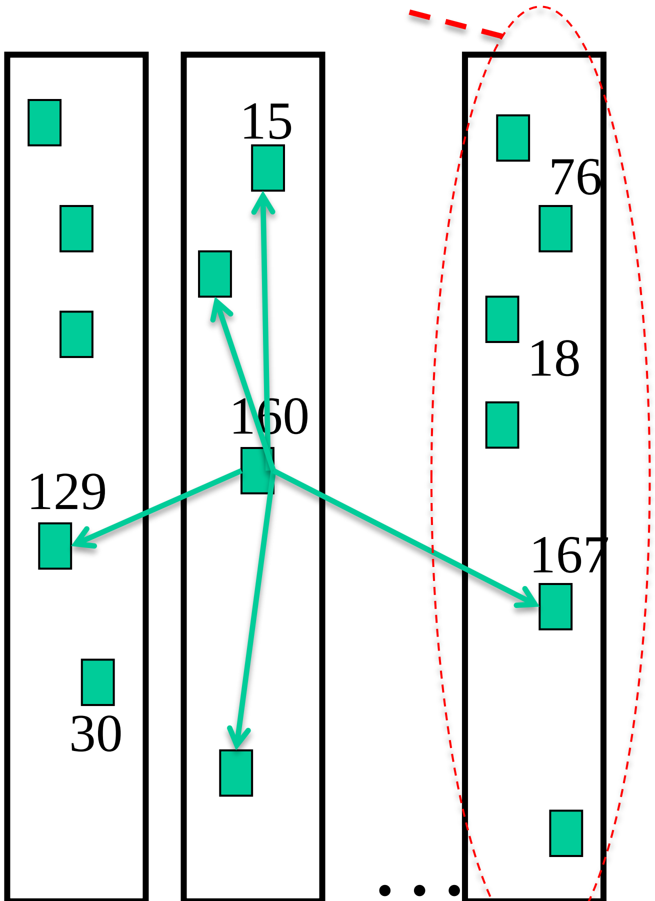


Kelips Lookup

- **Lookup**
  - Find file affinity group
  - Go to your contact for the file affinity group
  - Failing that try another of your neighbors to find a contact
- **Lookup = 1 hop (or a few)**
  - Memory cost O(√ N)
  - 1.93 MB for 100K nodes, 10M files
  - Fits in RAM of most workstations/laptops today (COTS machines)

- Publicenemy.mp3 hashes to k-1
- Everyone in this group stores

<Publicenemy.mp3, who-has-file>

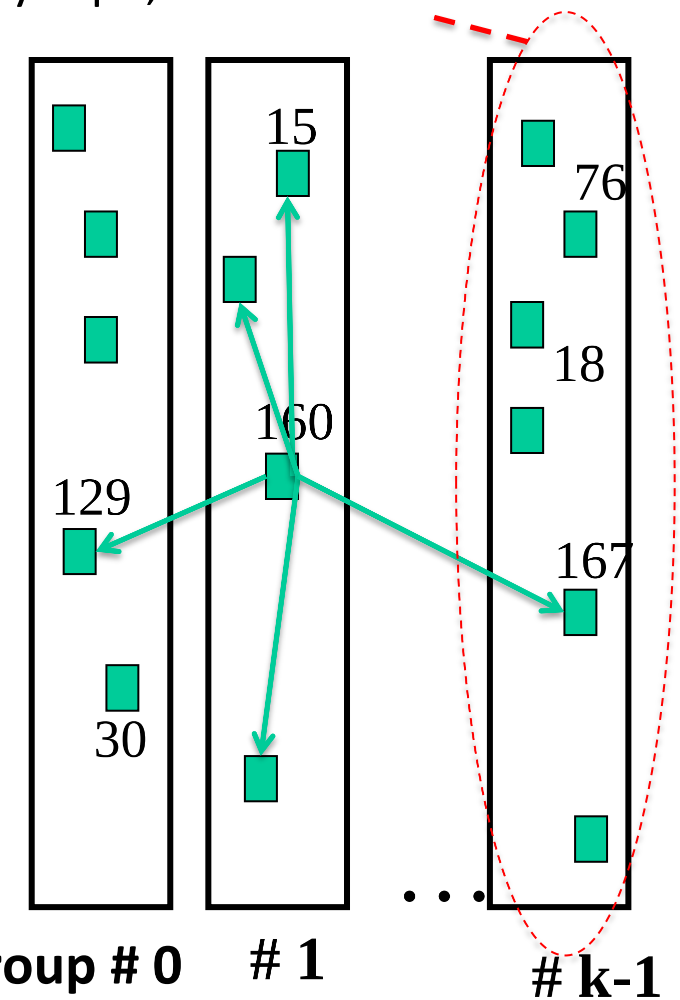


Kelips Soft State

- Publicenemy.mp3 hashes to k-1
- Everyone in this group stores

<Publicenemy.mp3, who-has-file>

- **Membership lists**
  - Gossip-based membership
  - Within each affinity group
  - And also across affinity groups
  - *O(log(N))* dissemination time
- **File metadata**
  - Needs to be periodically refreshed from source node
  - Times out



Chord vs. Pastry vs. Kelips

- **Range of tradeoffs available:**
  - Memory vs. lookup cost vs. background bandwidth (to keep neighbors fresh)

Conclusion

- In this lecture, we have studied some of the **widely-deployed P2P Systems such as:**
  1. Napster
  2. Gnutella
  3. Fasttrack
  4. BitTorrent

- We have also discussed some of the **P2P Systems with Provable Properties such as:**
  1. Chord
  2. Pastry
  3. Kelips