

CS 525

Advanced Distributed Systems

Spring 2017

Indranil Gupta (Indy)

Lecture 4, 5

Peer to Peer Systems

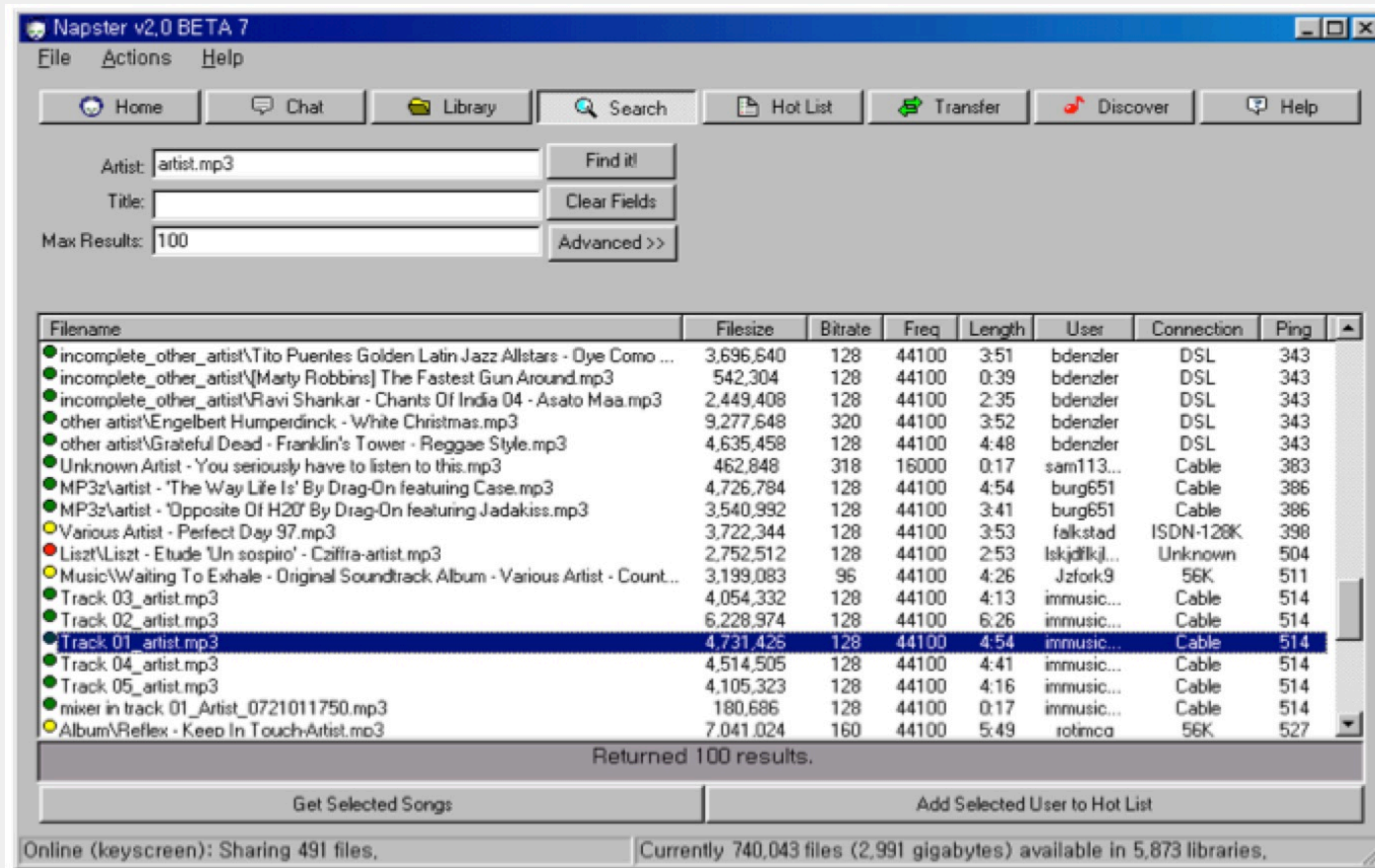
January 26-31, 2017

All slides © IG 1

Why Study Peer to Peer Systems?

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

Why Study Peer to Peer Systems?



A Brief History

- [6/99] Shawn Fanning (freshman Northeastern U.) releases Napster online music service
- [12/99] RIAA sues Napster, asking \$100K per download
- [3/00] 25% UWisc traffic Napster, many universities ban it
- [00] 60M users
- [2/01] US Federal Appeals Court: users violating copyright laws, Napster is abetting this
- [9/01] Napster decides to run paid service, pay % to songwriters and music companies
- [Today] Napster protocol is open, people free to develop opennap clients and servers <http://opennap.sourceforge.net>
 - Gnutella: <http://www.limewire.com> (deprecated)
 - Peer to peer working groups: <http://p2p.internet2.edu>

What We Will Study

- Widely-deployed P2P Systems
 1. Napster
 2. Gnutella
 3. Fasttrack (Kazaa, Kazaalite, Grokster)
 4. BitTorrent
- P2P Systems with Provable Properties
 1. Chord
 2. Pastry
 3. Kelips

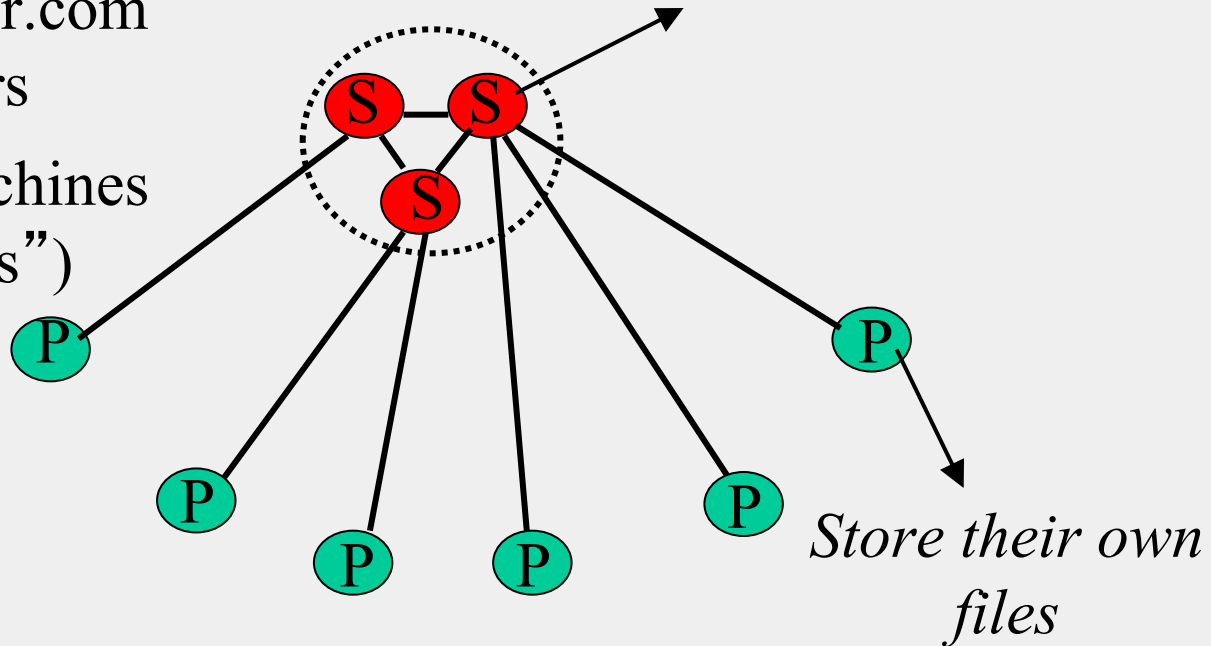
Napster Structure

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

Filename	Info about
PennyLane.mp3	Beatles, @ 128.84.92.23:1006

napster.com
Servers

Client machines
("Peers")



Napster Operations

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.**

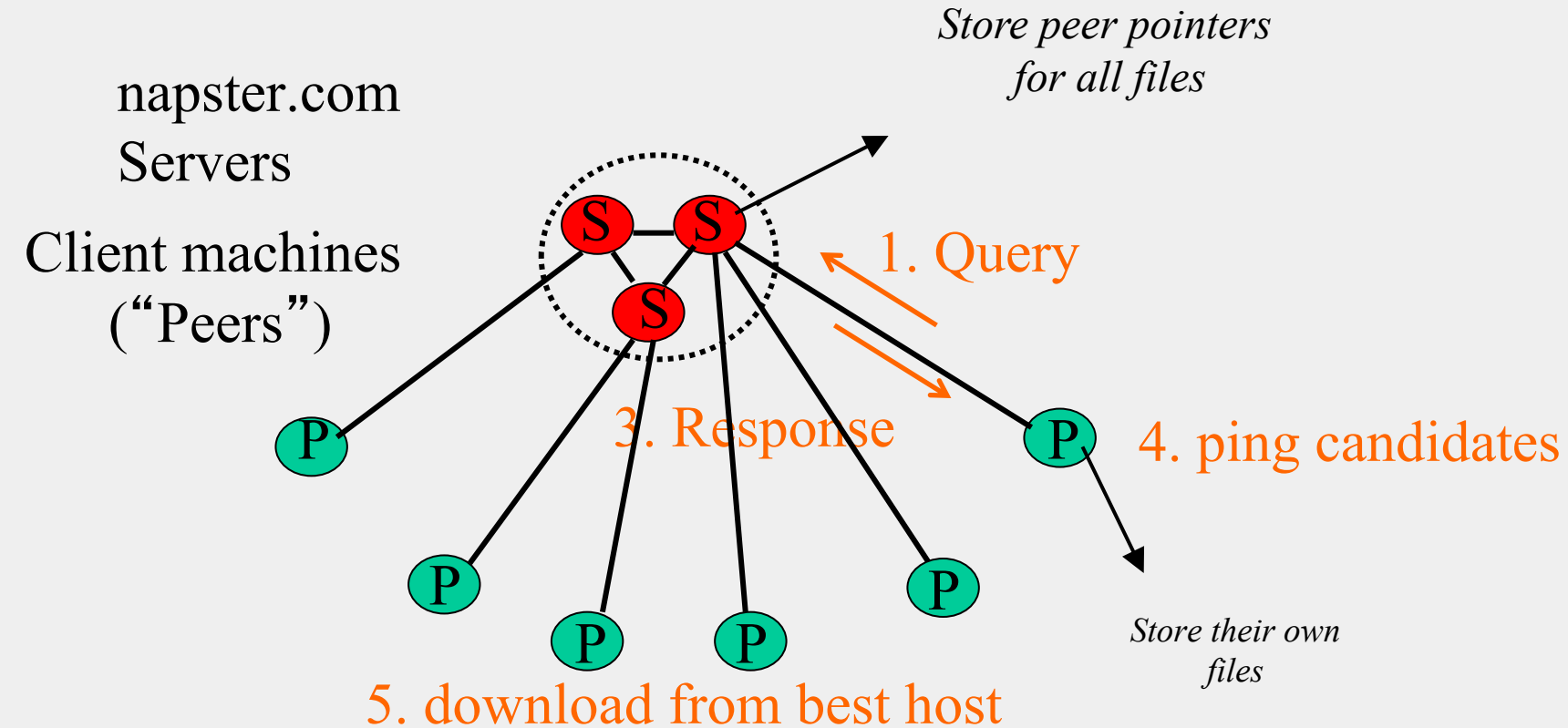
Napster Operations

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

2. All servers search their lists (ternary tree algorithm)



Joining a P2P system

- Can be used for any p2p system
 - Send an http request to well-known url for that P2P service - `http://www.myp2pservice.com`
 - 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

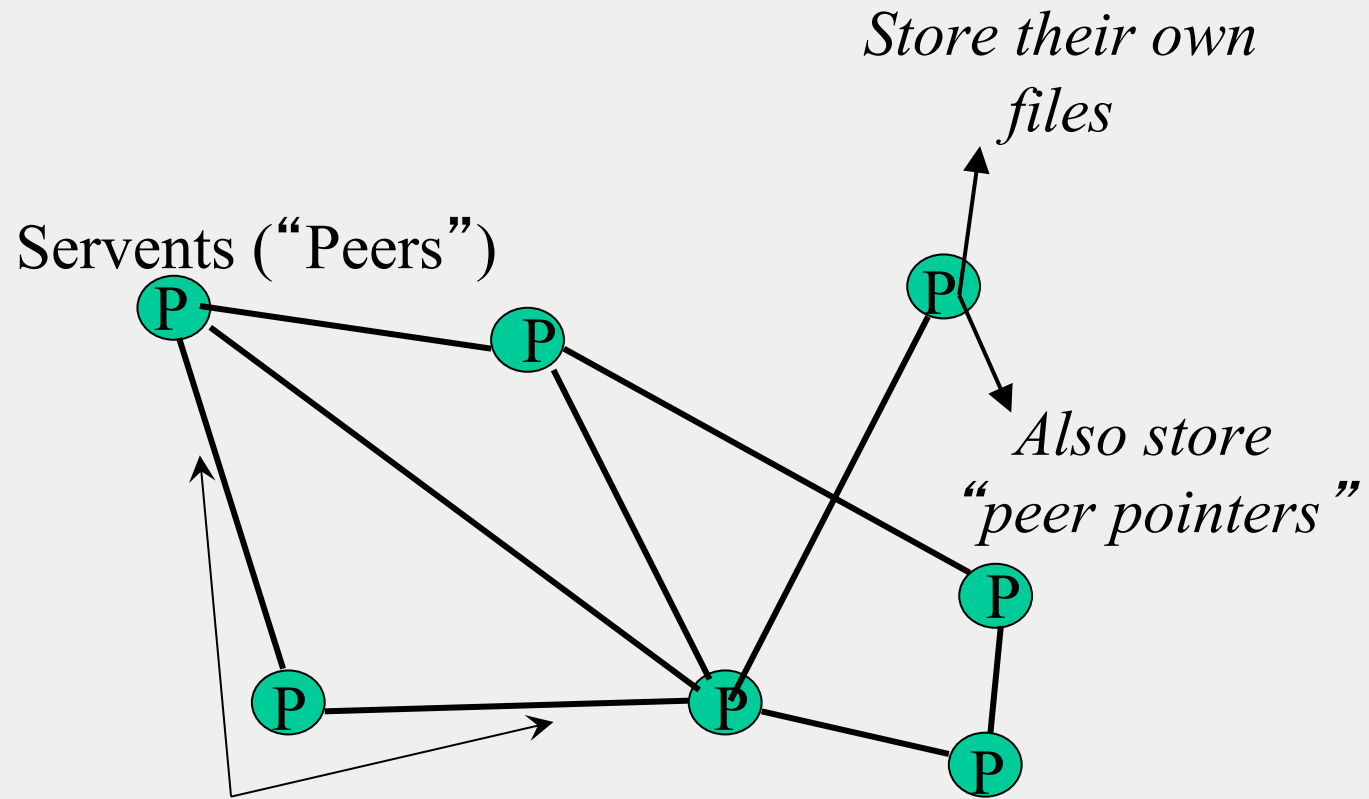
Problems

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

Gnutella

- Eliminate the servers
- Client machines search and retrieve amongst themselves
- Clients act as servers too, called **servents**
- [3/00] release by AOL, immediately withdrawn, but 88K users by 3/03
- Original design underwent several modifications

Gnutella

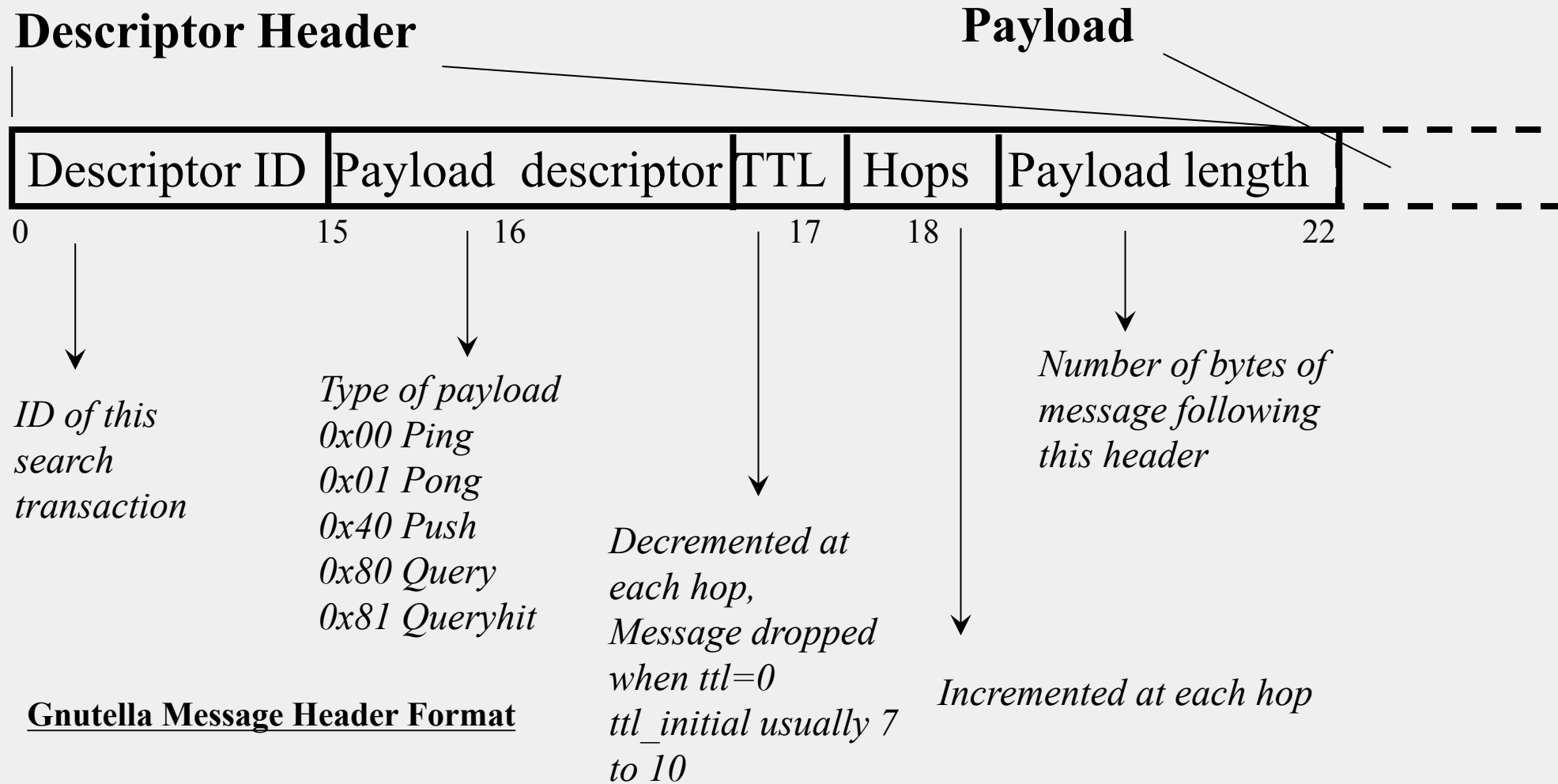


Connected in an **overlay graph**
(== each link is an implicit Internet path)

How do I search for my Beatles file?

- Gnutella *routes* different messages within the overlay graph
- Gnutella protocol has 5 main message types
 - **Query** (search)
 - **QueryHit** (response to query)
 - **Ping** (to probe network for other peers)
 - **Pong** (reply to ping, contains address of another peer)
 - Push (used to initiate file transfer)
- We'll go into the message structure and protocol now
 - 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 my Beatles file?



How do I search for my Beatles file?

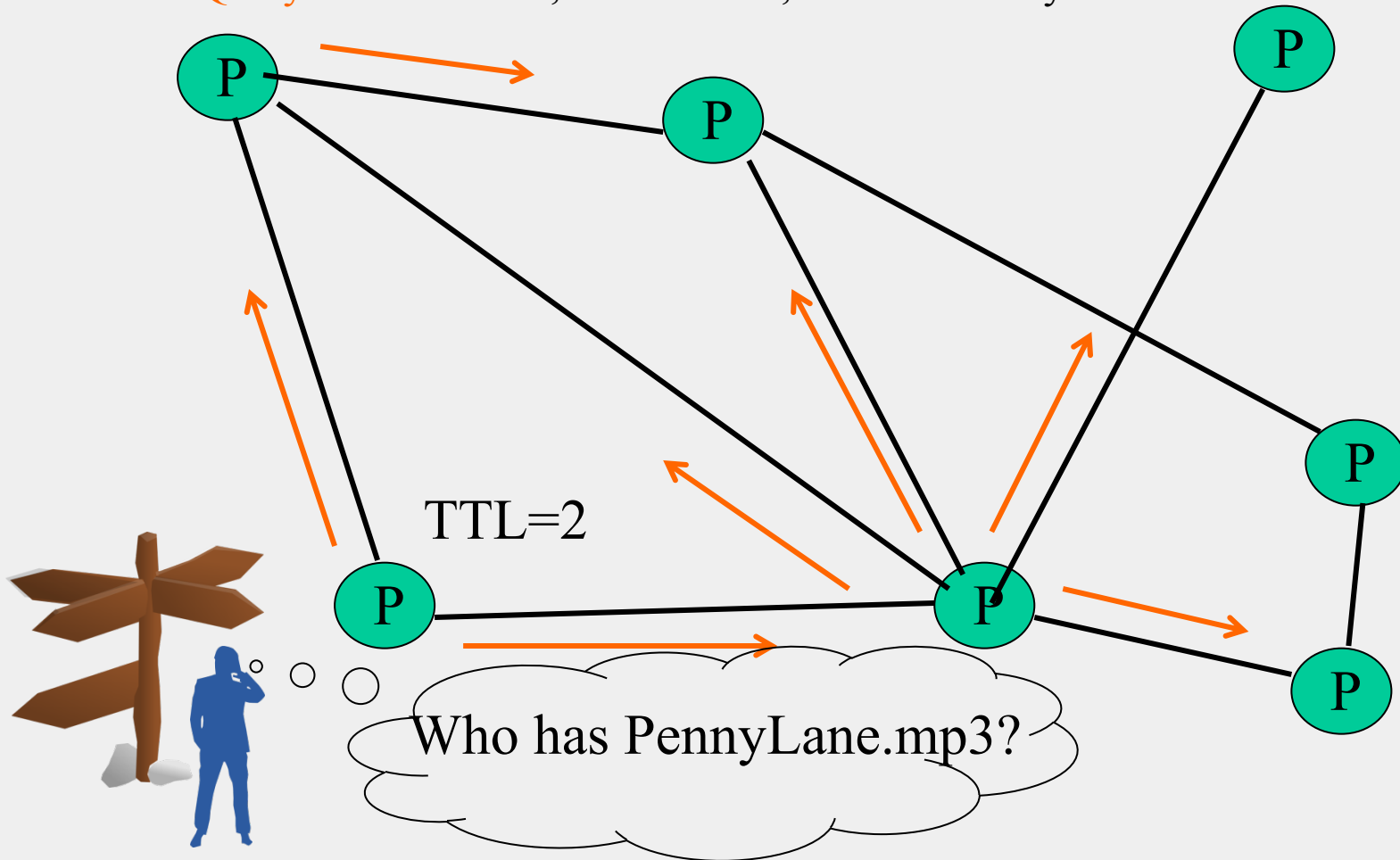
Query (0x80)



Payload Format in Gnutella **Query** Message

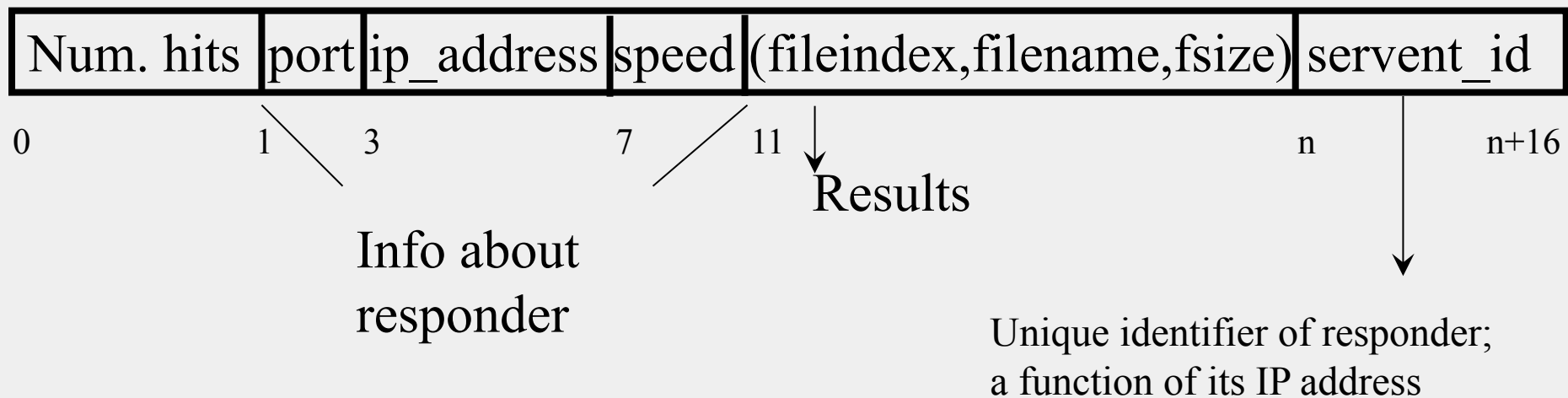
Gnutella Search

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



Gnutella Search

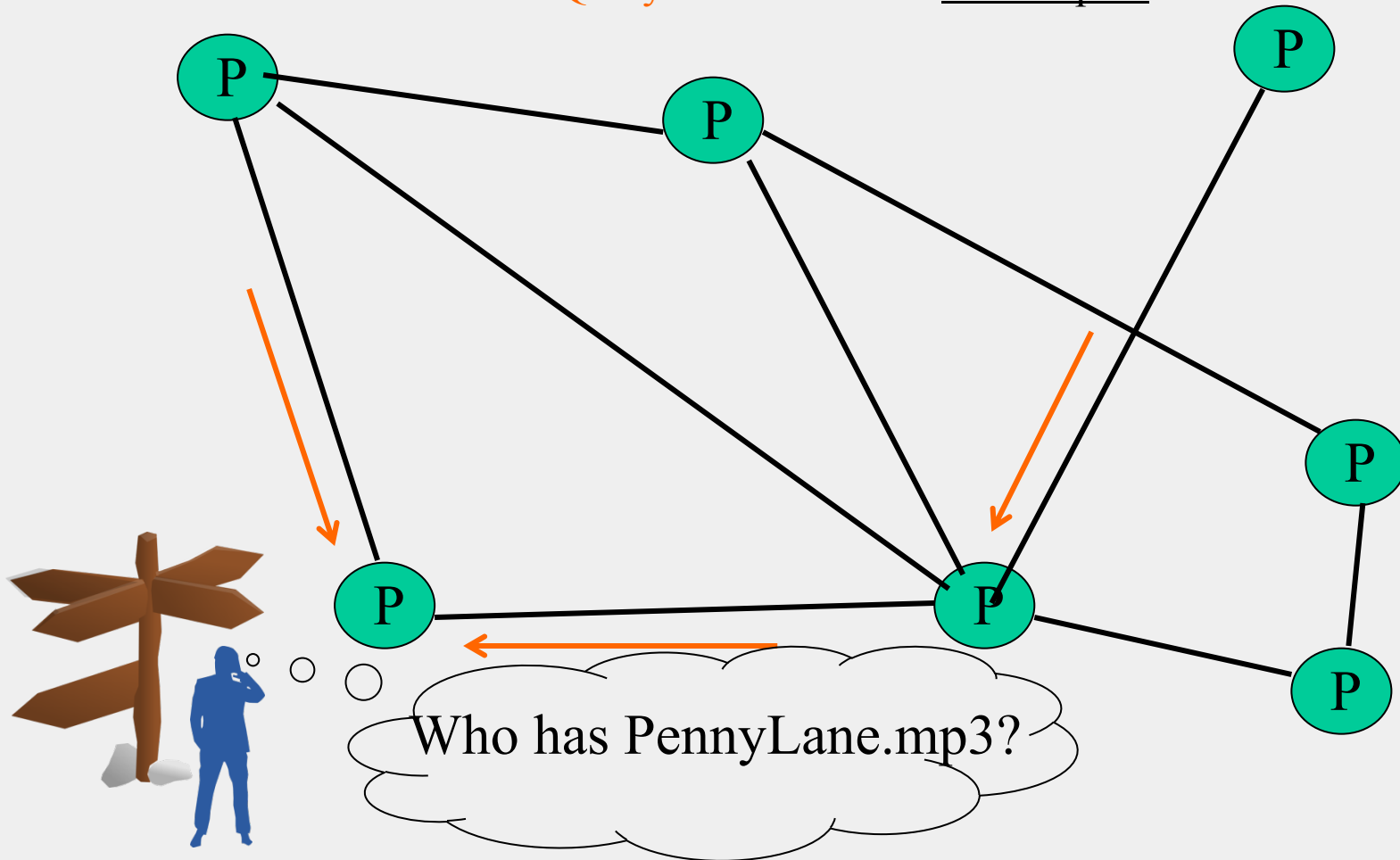
QueryHit (0x81) : successful result to a query



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
```

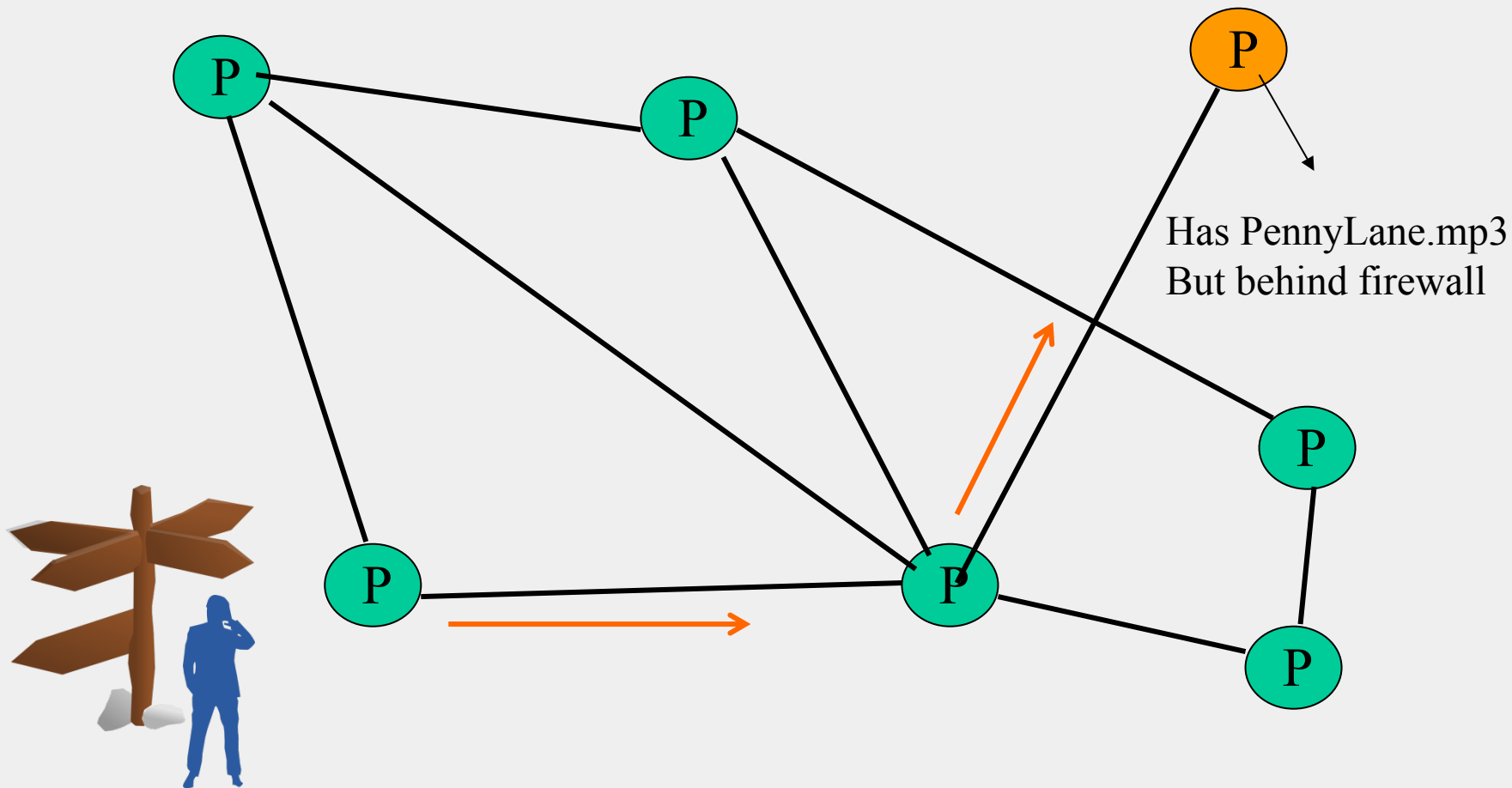
```
\r\n
```

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)



same as in
received QueryHit

Address at which
requestor 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

Gnutella Summary

- 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:
$$P(\text{\#links} = L) \sim L^{-k} \quad (k \text{ is a constant})$$

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 (soon)

Problems (contd.)

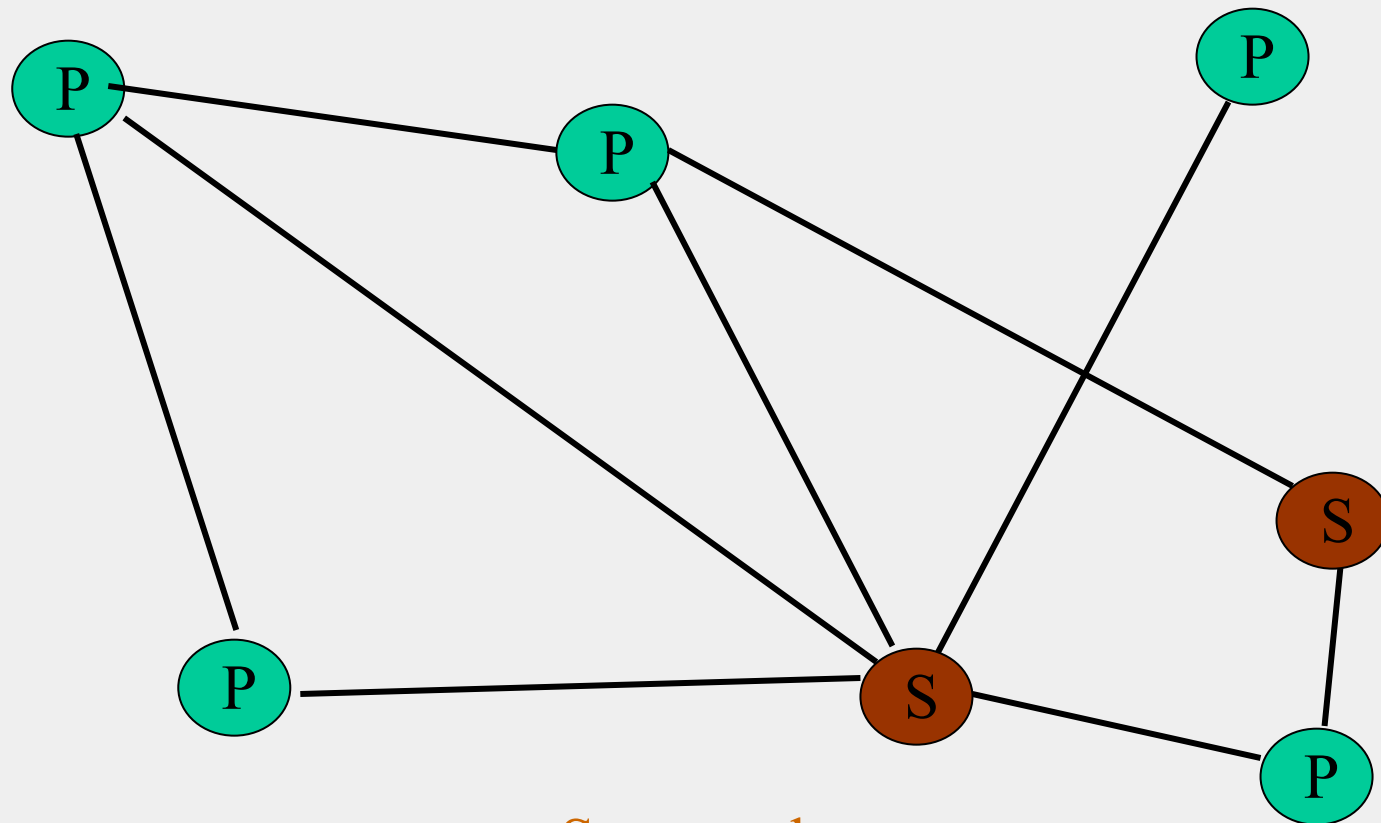
- Large number of *freeloaders*
 - 70% of users in 2000 were freeloaders
 - Only download files, never upload own files
- Flooding causes excessive traffic
 - Is there some way of maintaining meta-information about peers that leads to more intelligent routing?
 - ➔ Structured Peer-to-peer systems
 - e.g., Chord System (coming up next lecture)

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

Peers

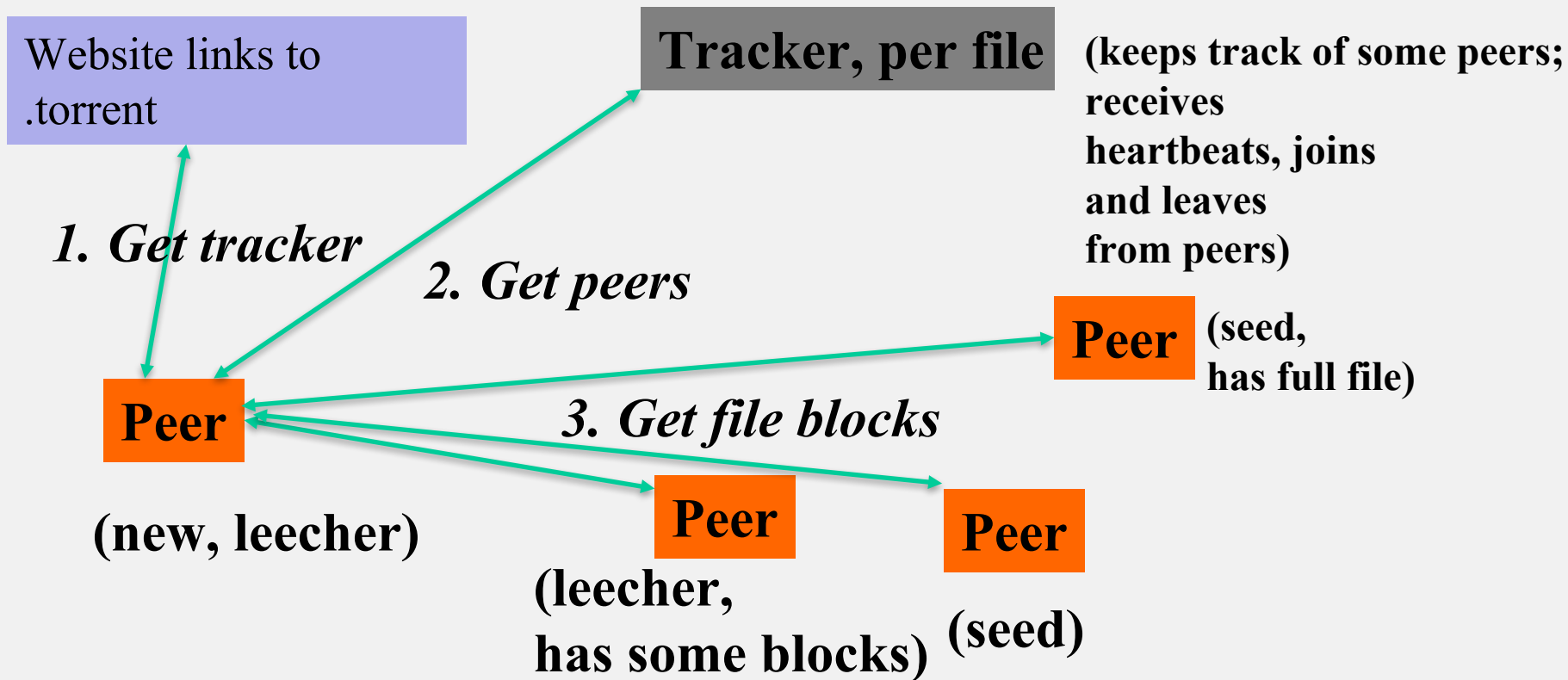


Supernodes

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 (See P2PEcon workshop)
- A peer searches by contacting a nearby supernode

BitTorrent



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 \leq 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 neighbor – 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 that we study next

Comparative Performance

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

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	$O(\log(N))$	$O(\log(N))$	$O(\log(N))$	

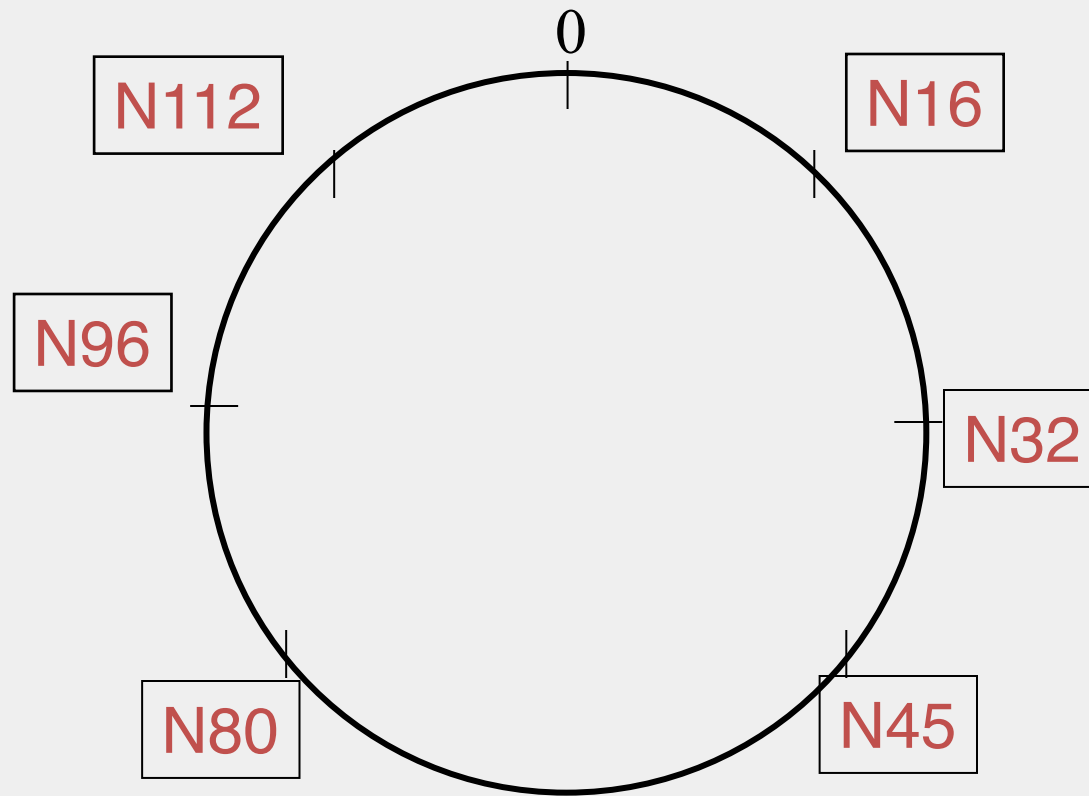
Chord

- 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

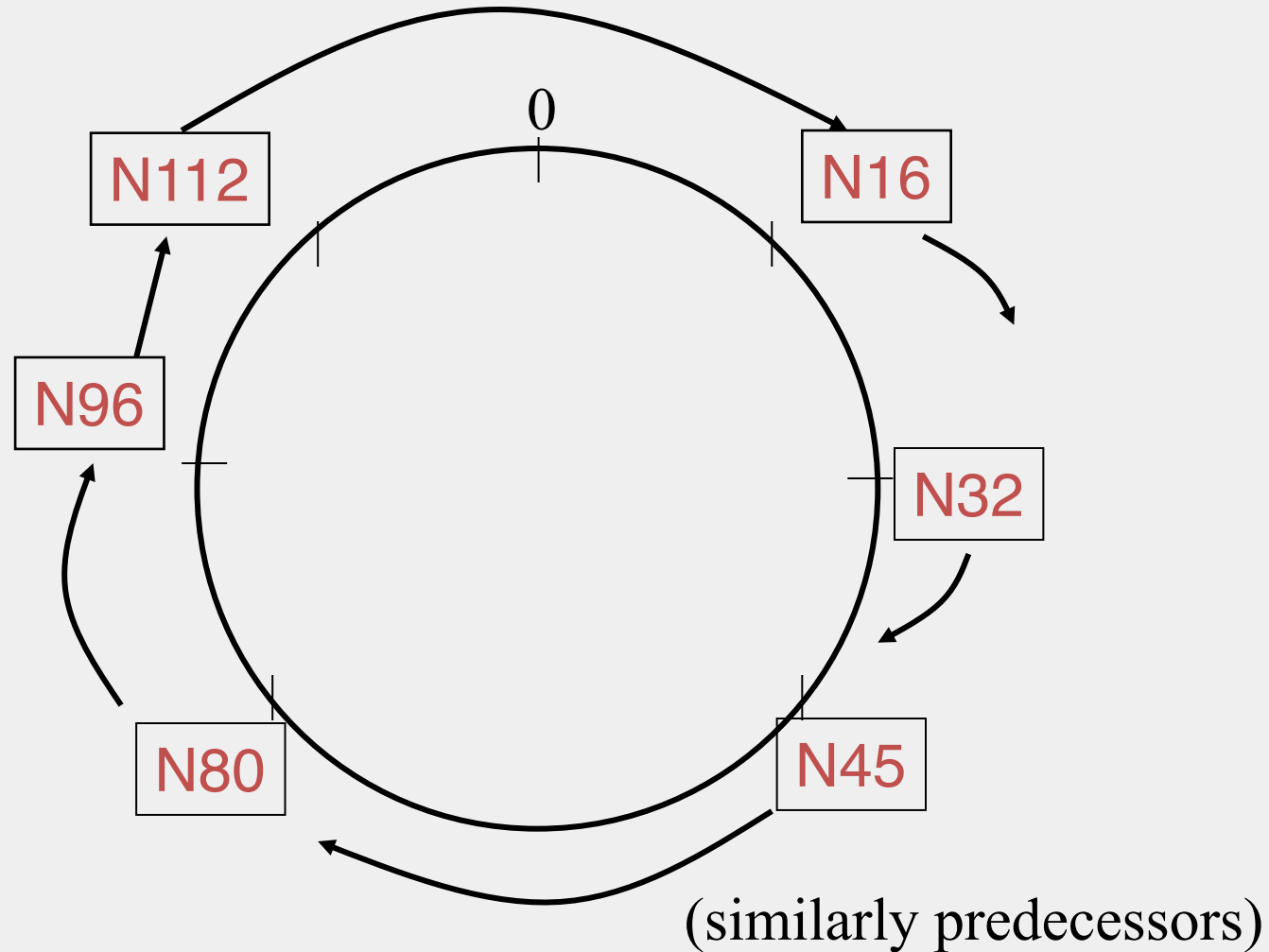
Say $m=7$

6 nodes

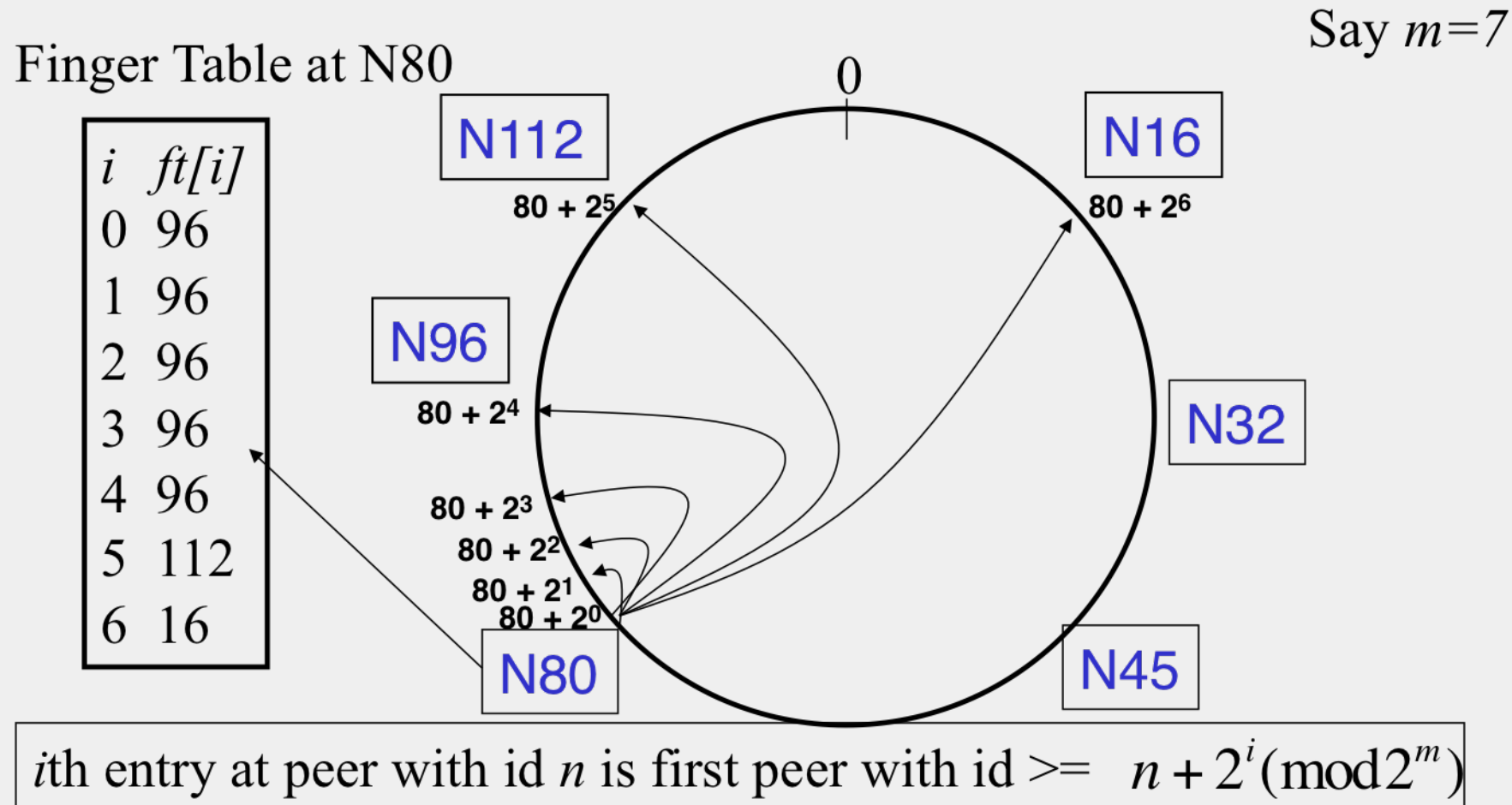


Peer pointers (1): successors

Say $m=7$



Peer pointers (2): finger tables

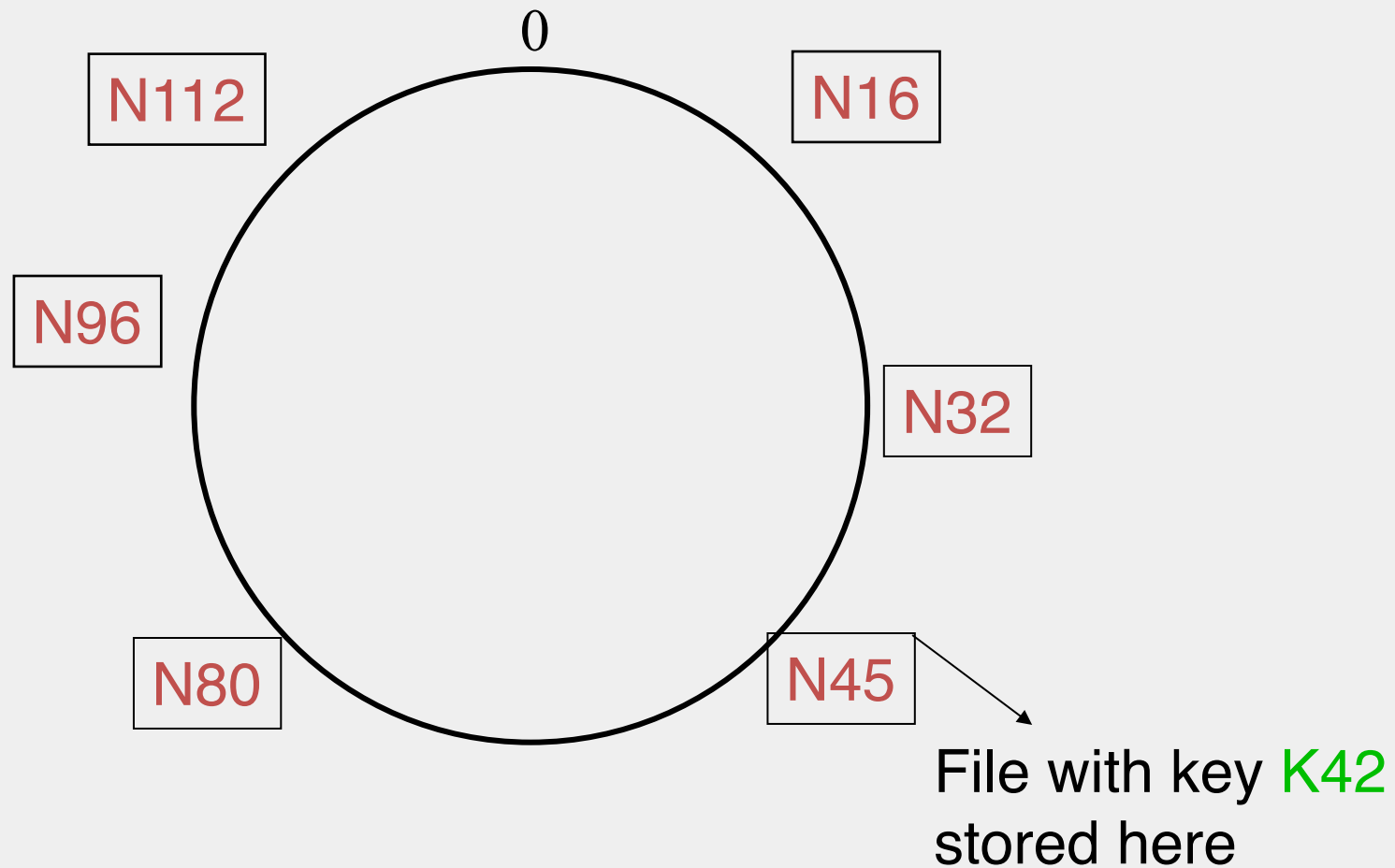


What about the files?

- Filenames also mapped using same consistent hash function
 - $\text{SHA-1}(\text{filename}) \rightarrow 160 \text{ bit string } (key)$
 - File is stored at first peer with id greater than or equal to its $key \pmod{2^m}$
- 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 \Rightarrow with K keys and N peers, each peer stores $O(K/N)$ keys. (i.e., $< c.K/N$, for some constant c)

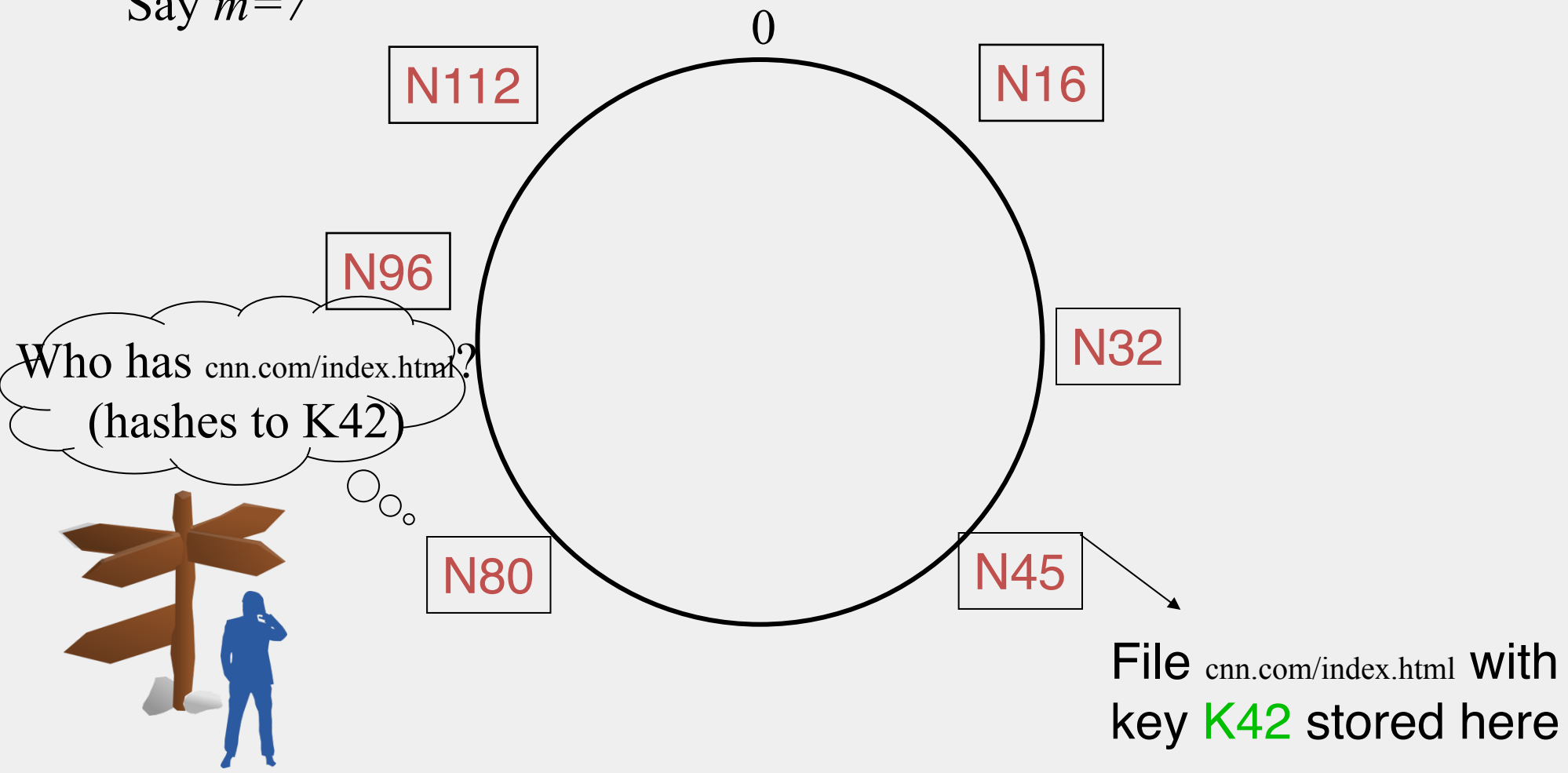
Mapping Files

Say $m=7$



Search

Say $m=7$

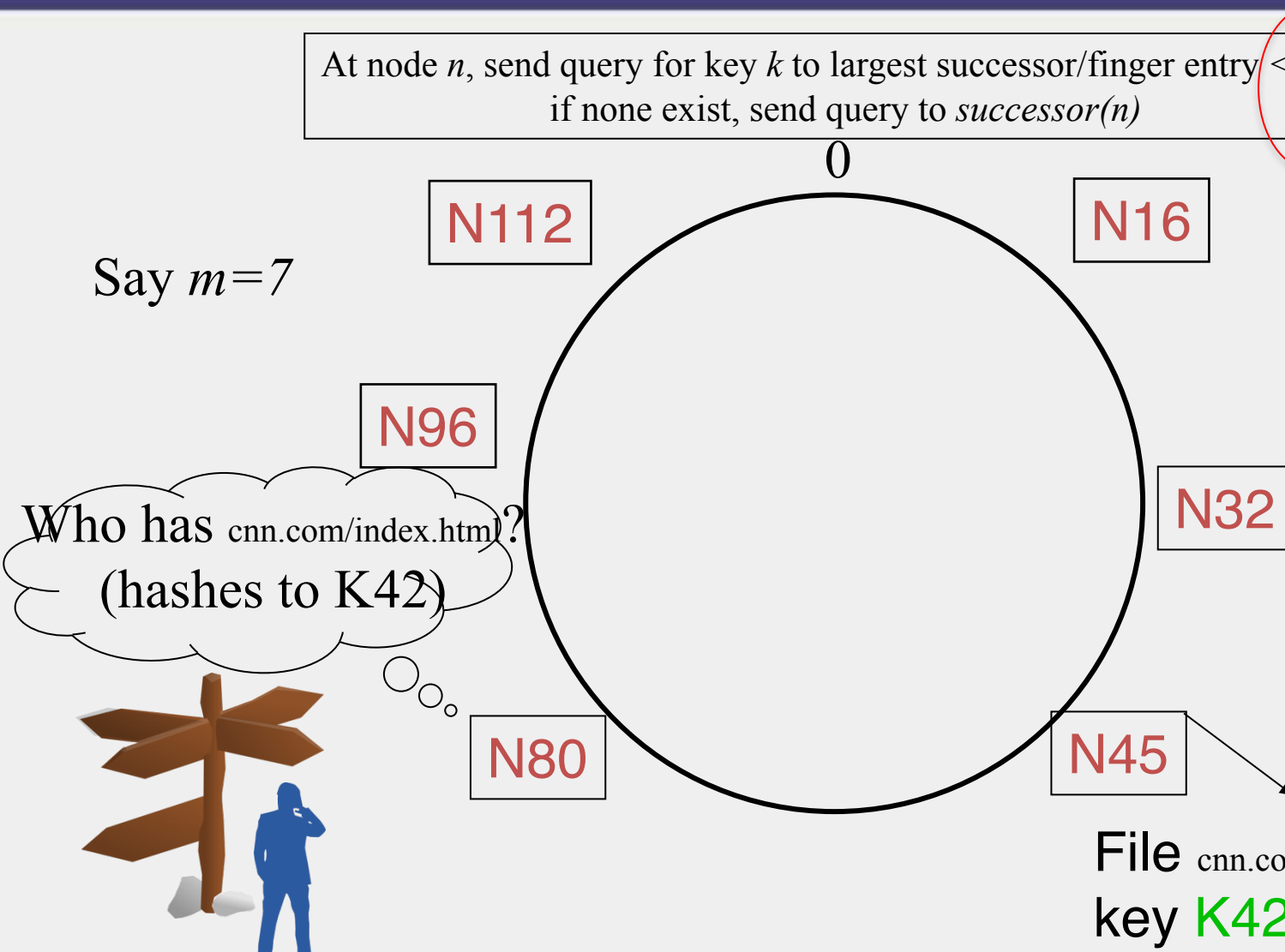


Search

At node n , send query for key k to largest successor/finger entry $\leq k$
if none exist, send query to $successor(n)$

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

Say $m=7$

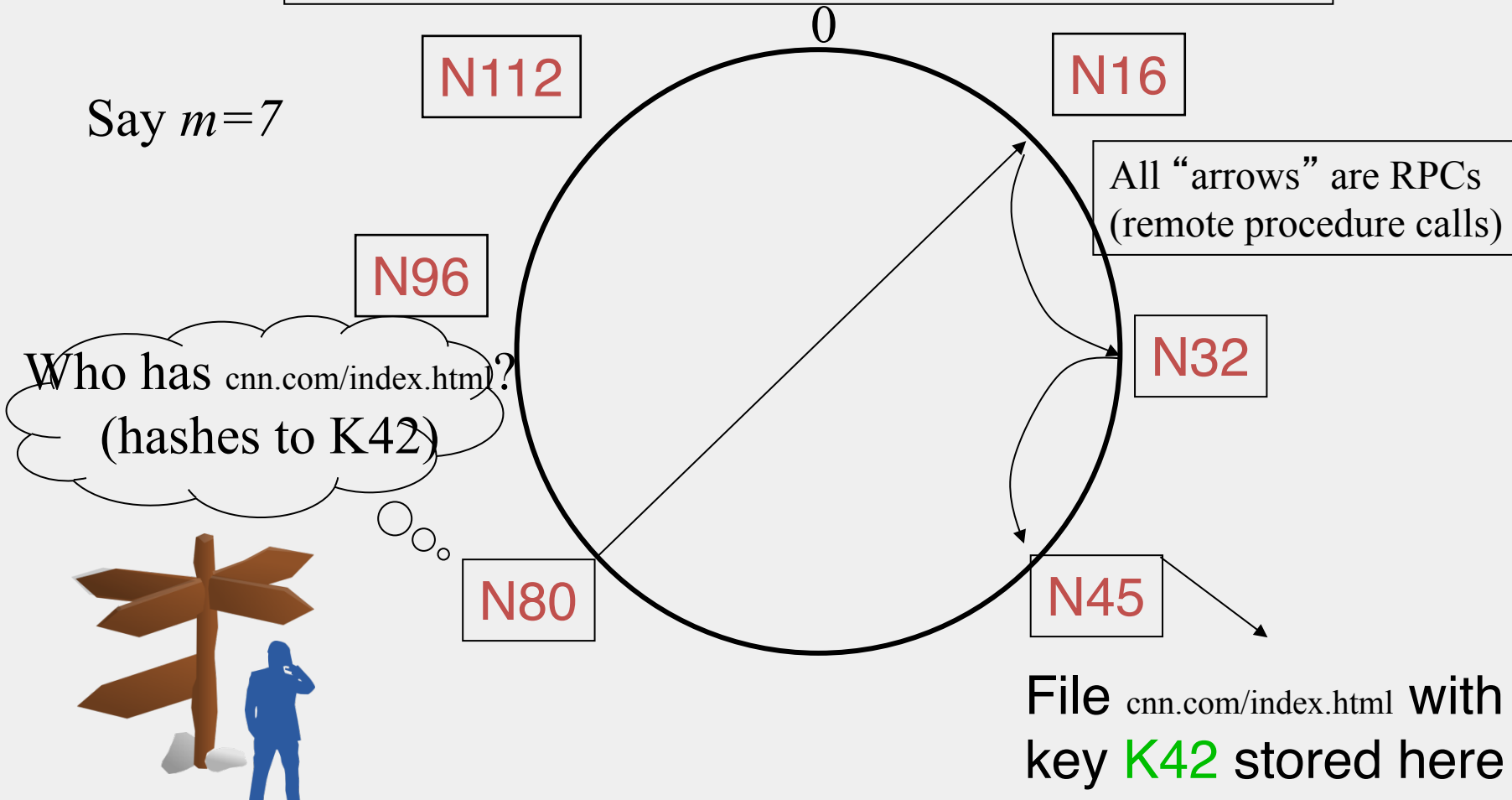


File `cnn.com/index.html` with
key **K42** stored here

Search

At node n , send query for key k to largest successor/finger entry $\leq k$
if none exist, send query to $successor(n)$

Say $m=7$



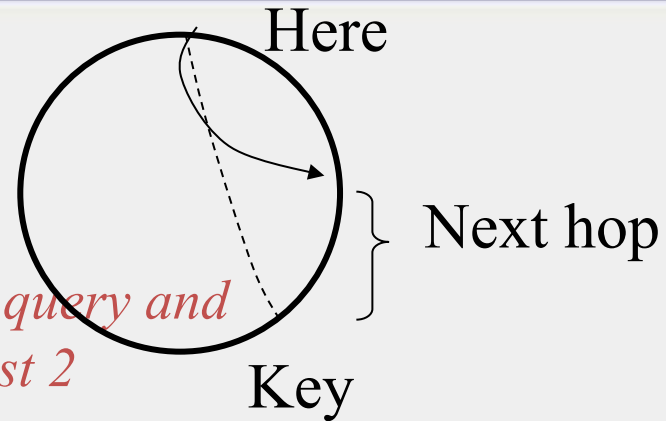
Analysis

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 $2^m / 2^{\log(N)} = 2^m / N$
- Number of node identifiers in a range of 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

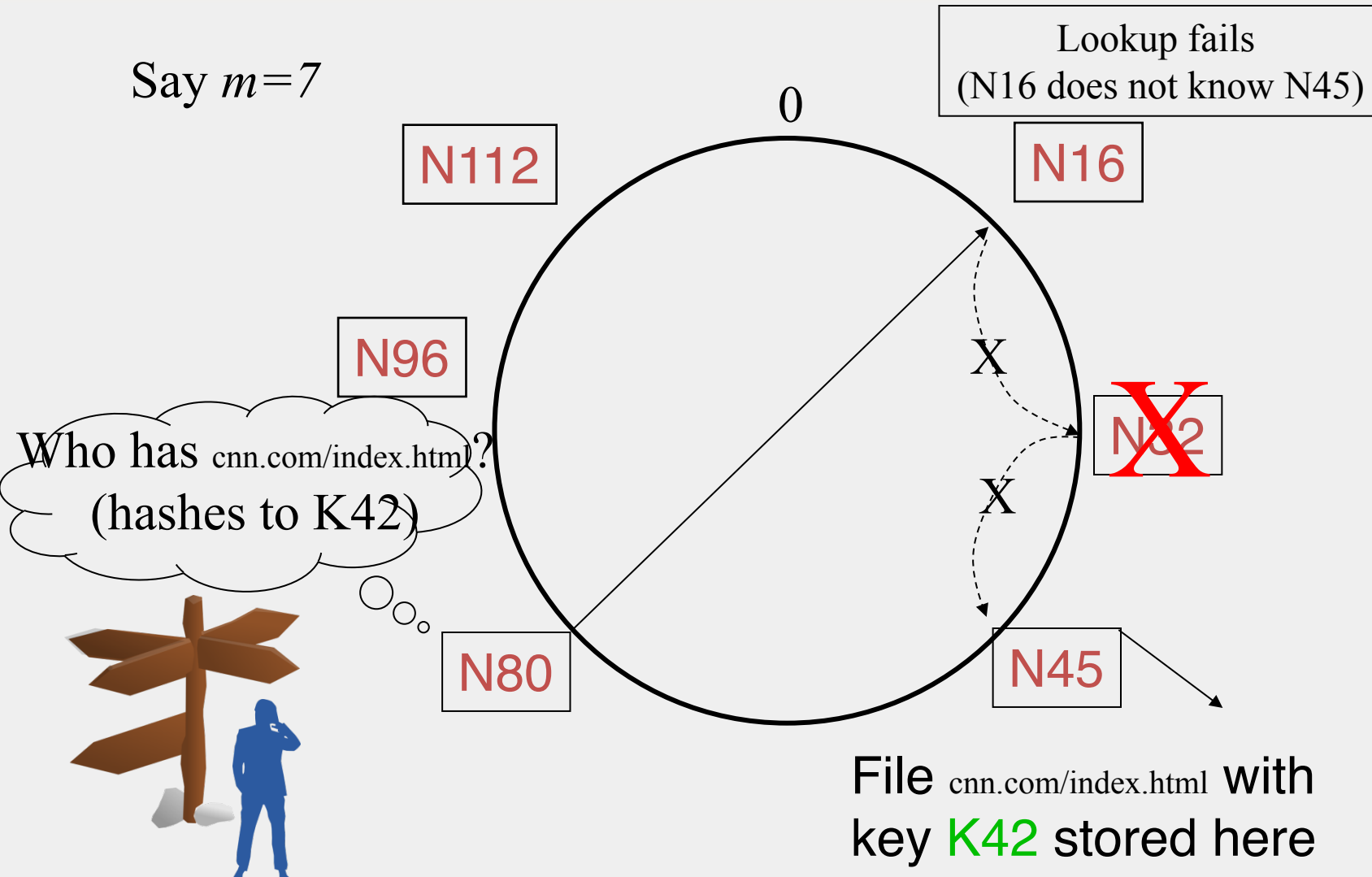


Analysis (contd.)

- $O(\log(N))$ 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

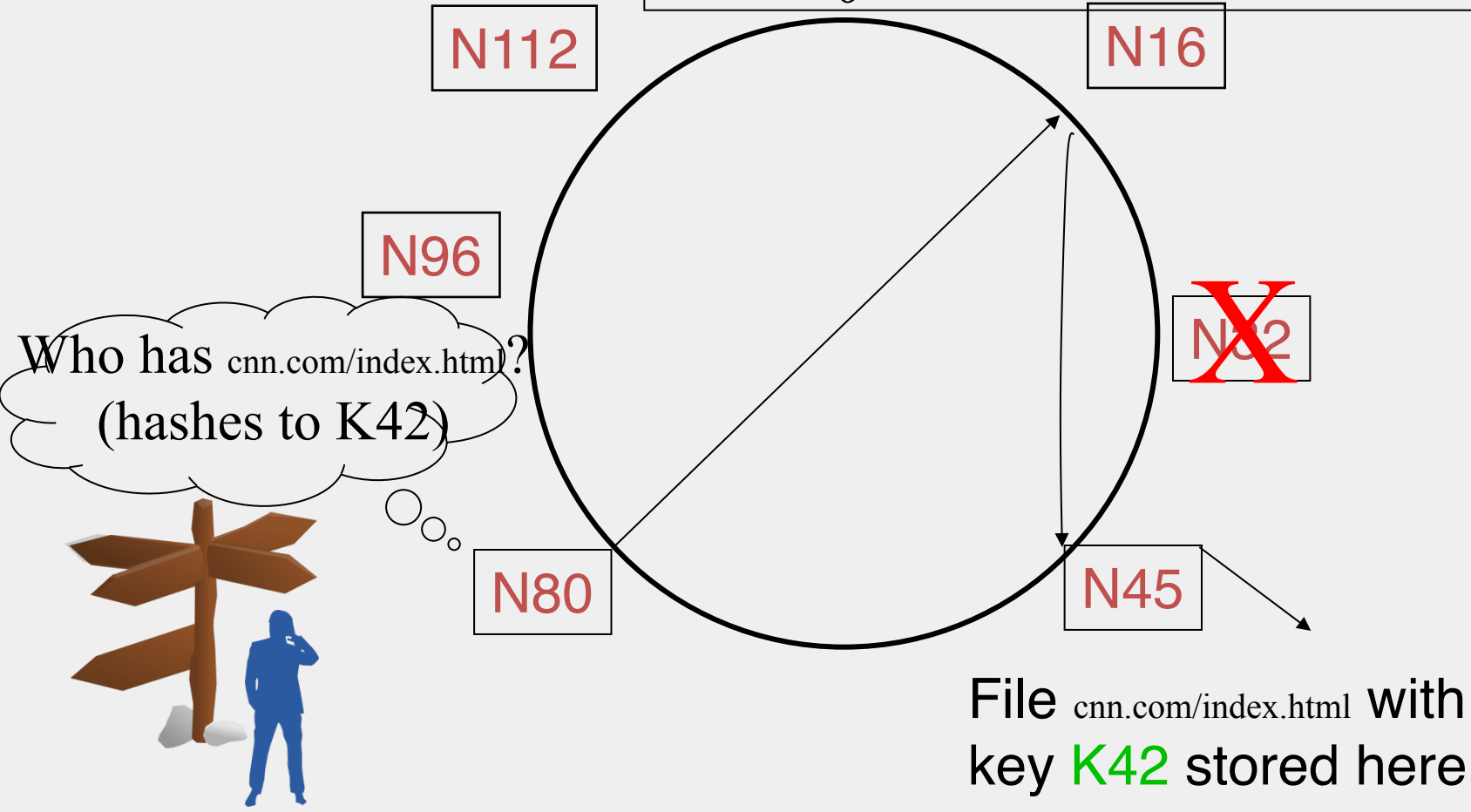
Say $m=7$



Search under peer failures

Say $m=7$

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



Search under peer failures

- Choosing $r=2\log(N)$ suffices to maintain *lookup correctness* w.h.p.(i.e., ring connected)
 - Say 50% of nodes fail
 - $\Pr(\text{at given node, at least one successor alive})=$

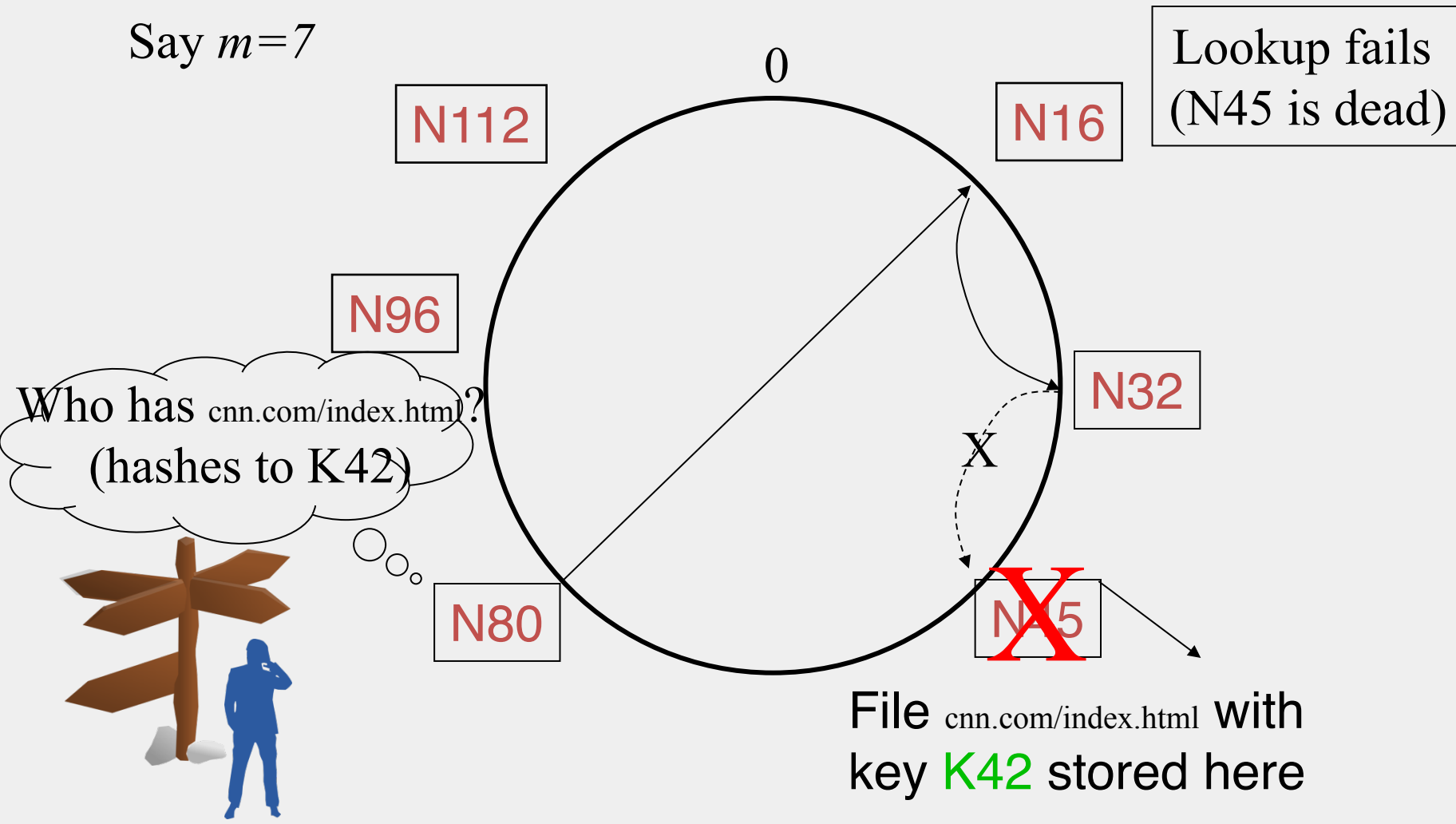
$$1 - \left(\frac{1}{2}\right)^{2\log N} = 1 - \frac{1}{N^2}$$

- $\Pr(\text{above is true at all alive nodes})=$

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

Search under peer failures (2)

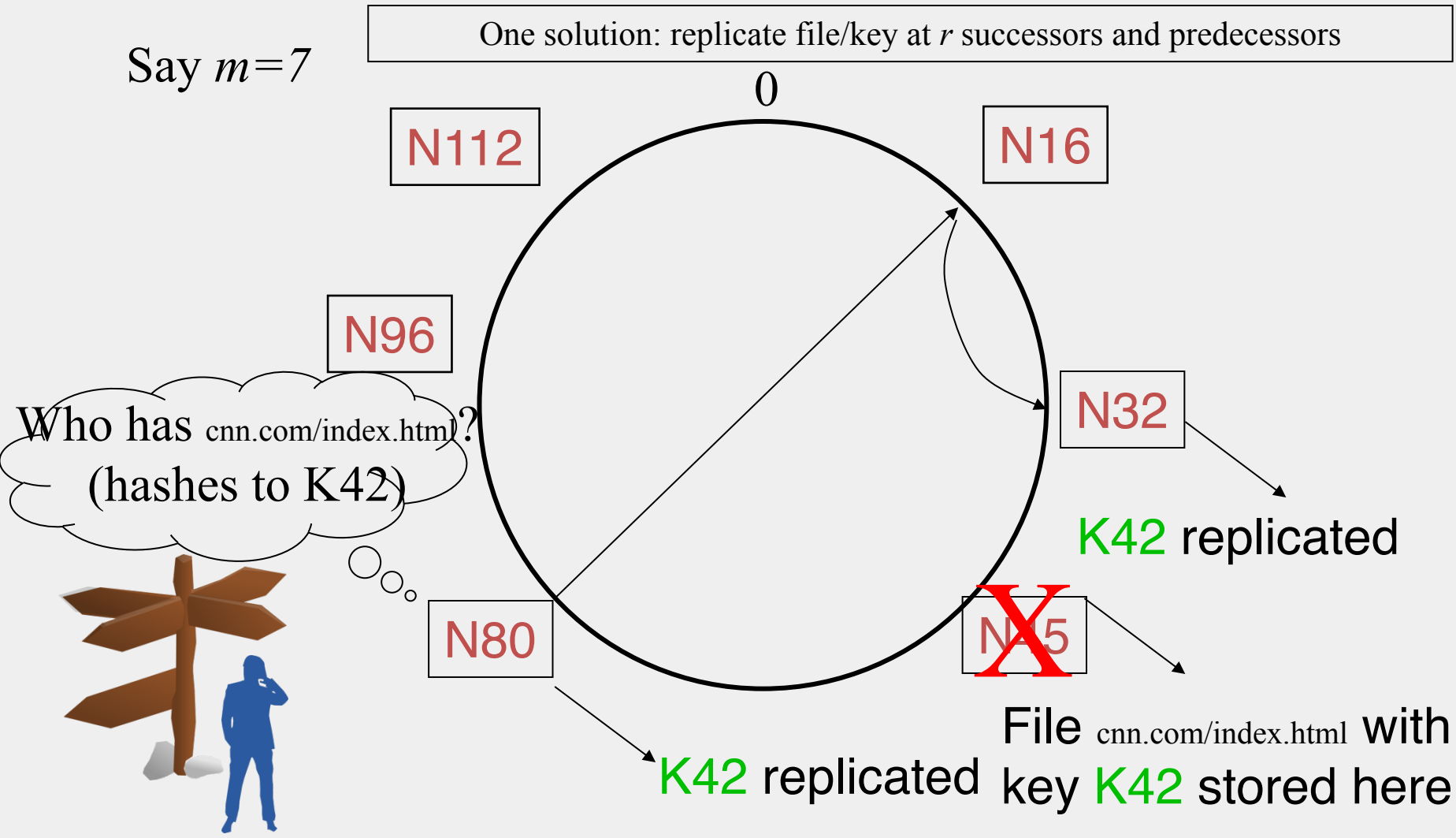
Say $m=7$



Search under peer failures (2)

Say $m=7$

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



Need to deal with dynamic changes

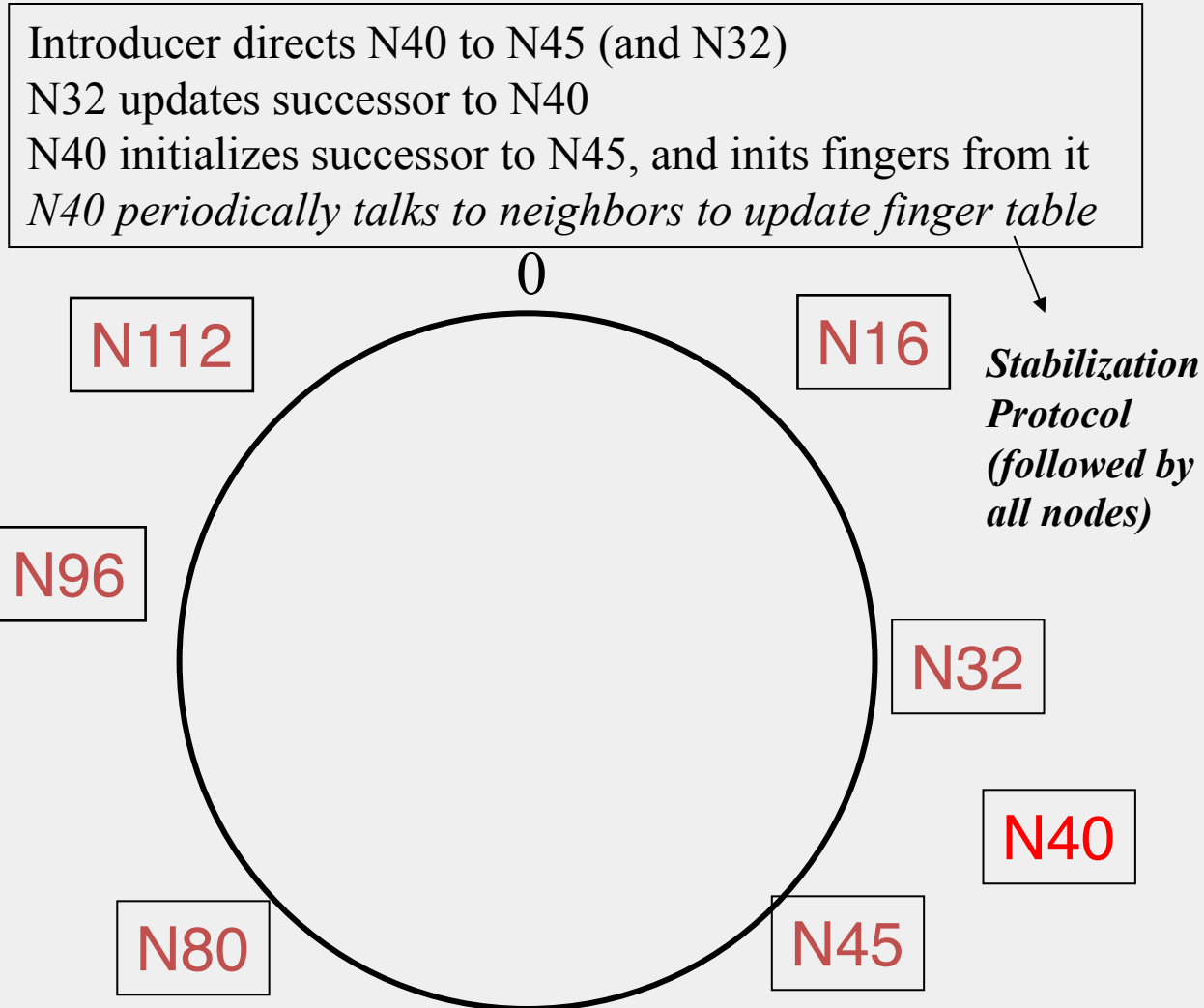
- ✓ Peers fail
 - New peers join
 - Peers leave
 - P2P systems have a high rate of *churn* (node join, leave and failure)
 - 25% per hour in Overnet (eDonkey)
 - 100% per hour in Gnutella
 - Lower in managed clusters
 - Common feature in all distributed systems, including wide-area (e.g., PlanetLab),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

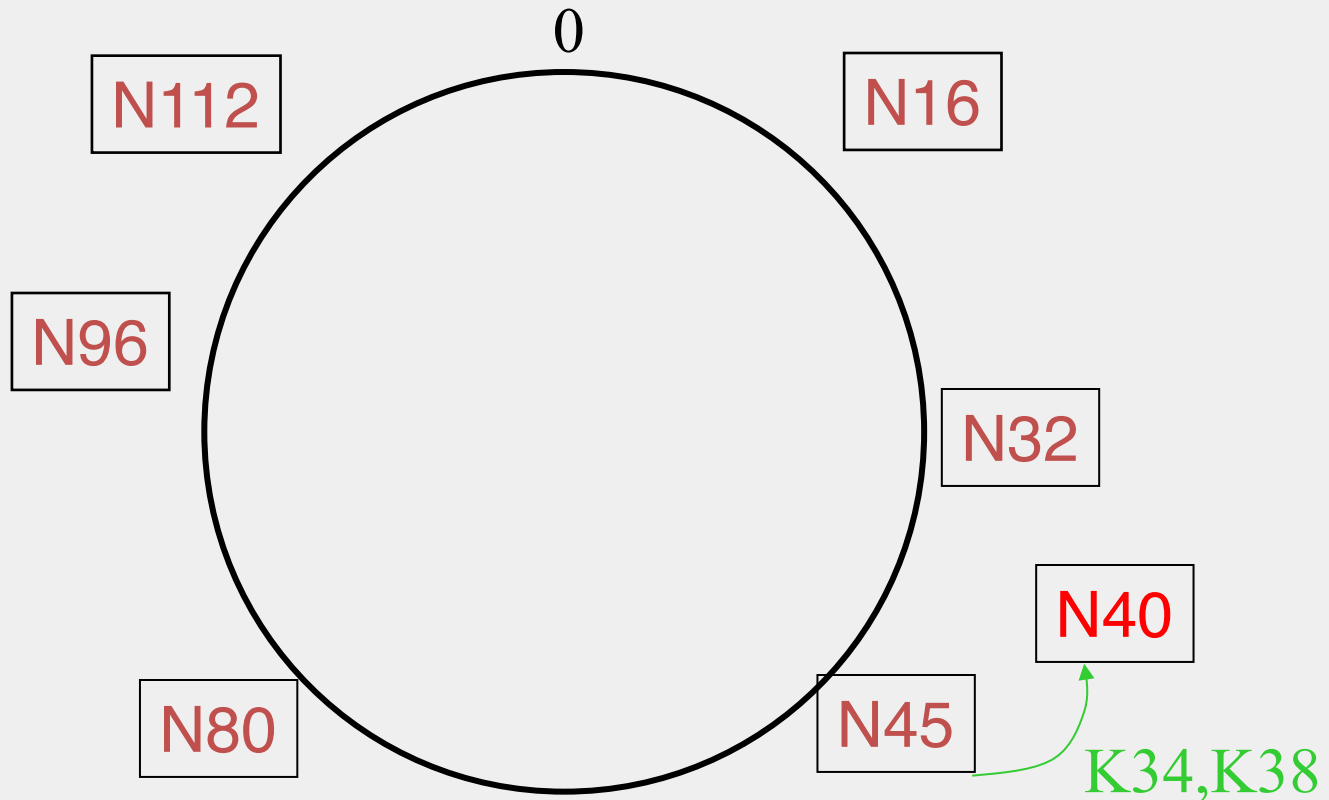
Say $m=7$



New peers joining (2)

N40 may need to copy some files/keys from N45
(files with fileid 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* (we'll see these later in the course!)

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 w.h.p.
 - Each stabilization round at a peer involves a constant number of messages
 - Strong stability takes $O(N^2)$ stabilization rounds
 - For more see [TechReport on Chord webpage]

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 (any p2p system)
 - Replicate metadata more, e.g., Kelips (later in this lecture)

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

Chord Wrap-up Notes

- 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

<http://www.pdos.lcs.mit.edu/chord/>

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 (* = bit different 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

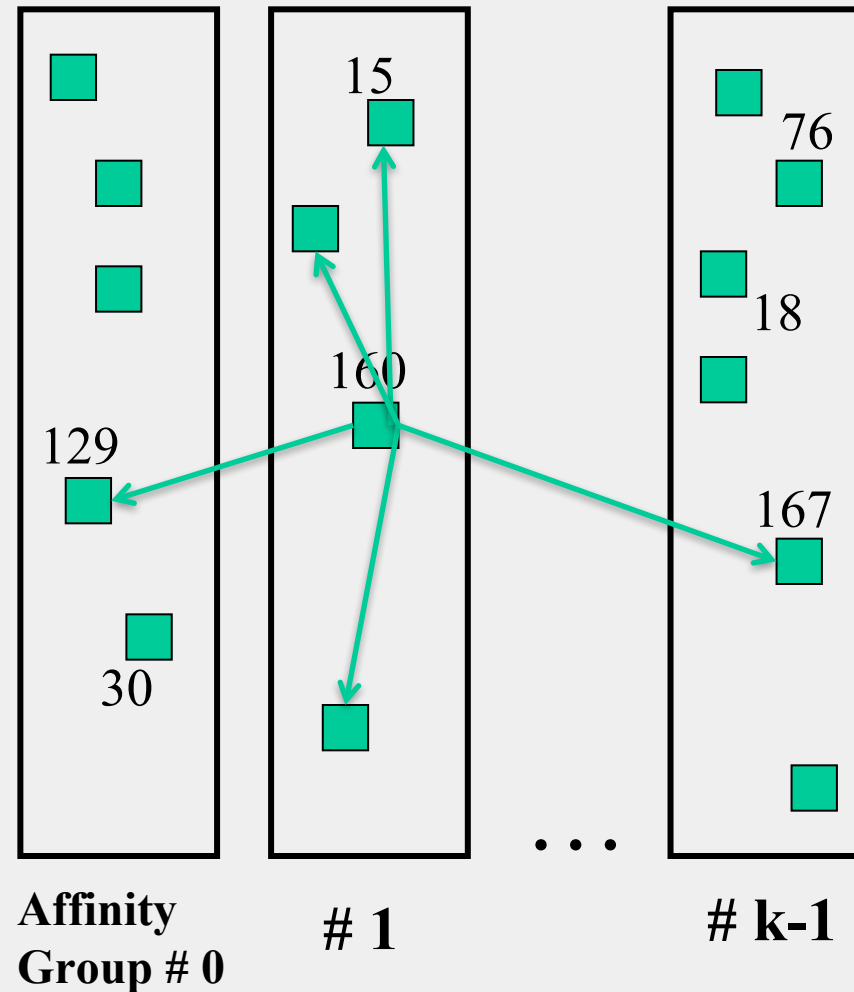
- For each prefix, say 011*, among all potential neighbors with the matching prefix, the neighbor with the shortest round-trip-time is selected
- Since shorter prefixes have many more candidates (spread out throughout the Internet), the neighbors for shorter prefixes are likely to be closer than the 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 of 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?

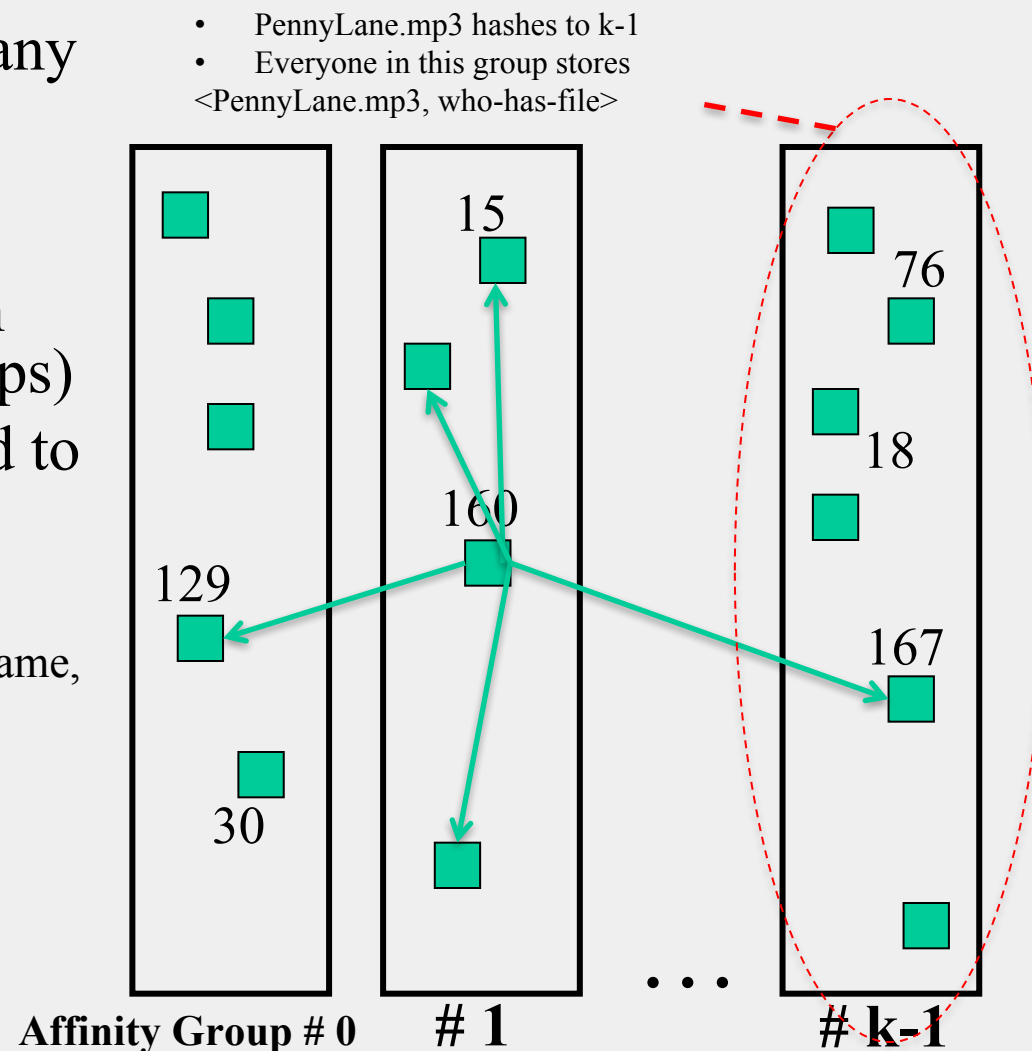
Kelips – A 1 hop Lookup DHT

- k “affinity groups”
 - $k \sim \sqrt{N}$
- 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



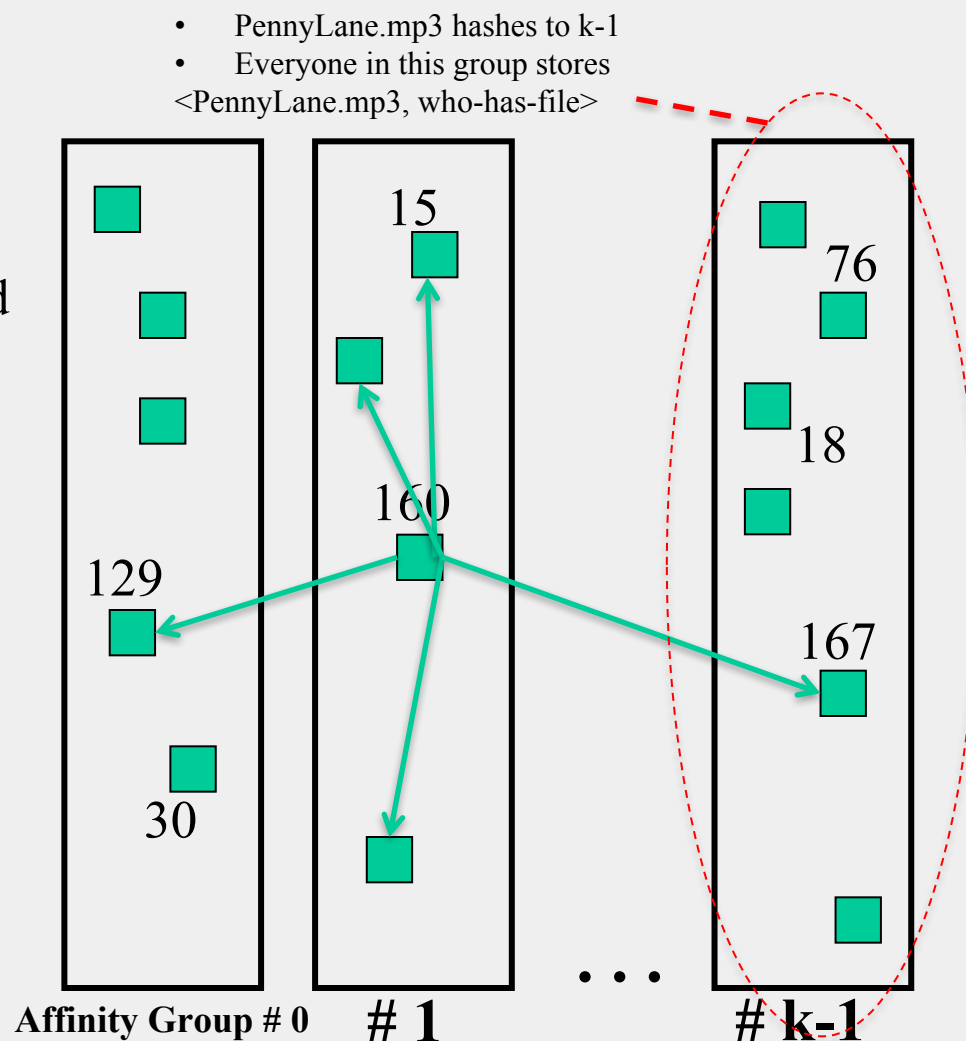
Kelips Files and Metadata

- 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



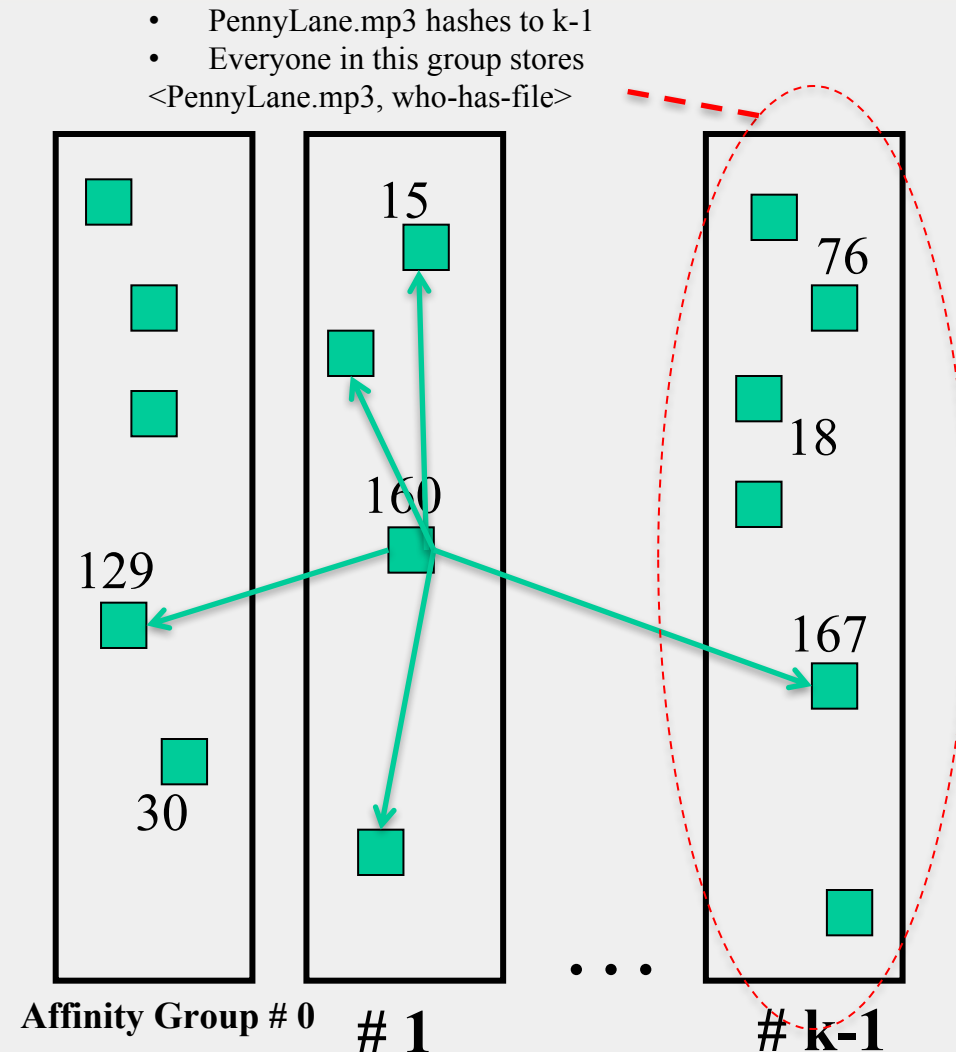
Kelips Lookup

- 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 $O(\sqrt{N})$
- 1.93 MB for 100K nodes, 10M files
- Fits in RAM of most workstations/laptops today (COTS machines)



Kelips Soft State

- 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)

What We Have Studied

- Widely-deployed P2P Systems
 1. Napster
 2. Gnutella
 3. Fasttrack (Kazaa, Kazaalite, Grokster)
 4. BitTorrent
- P2P Systems with Provable Properties
 1. Chord
 2. Pastry
 3. Kelips