ECEN 757
P2P Systems

Chapter 10

P2P for Data Centers

- Goal: Build a P2P system for storing all Netflix movies
- Q1: Is it a good idea to build a Napster-like system?
- Q2: Is it a good idea to build a Gnutella-like system?
- Q3: Consider Q1 and Q2, but this time for the application of storing all Facebook posts

Features of P2P Systems in Data Centers

- No selfish behavior
- Systems are less dynamics (failures still happen, though)
- Demand much better efficiency and load balance

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))*	

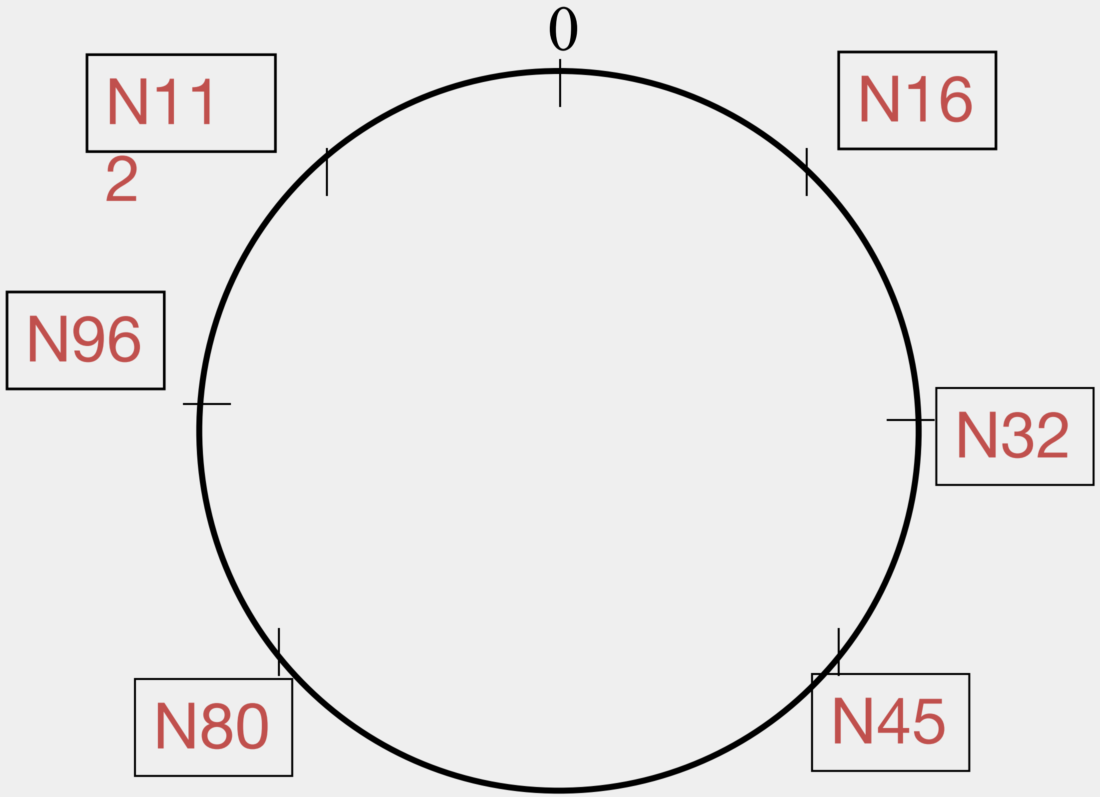
Chord

- 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

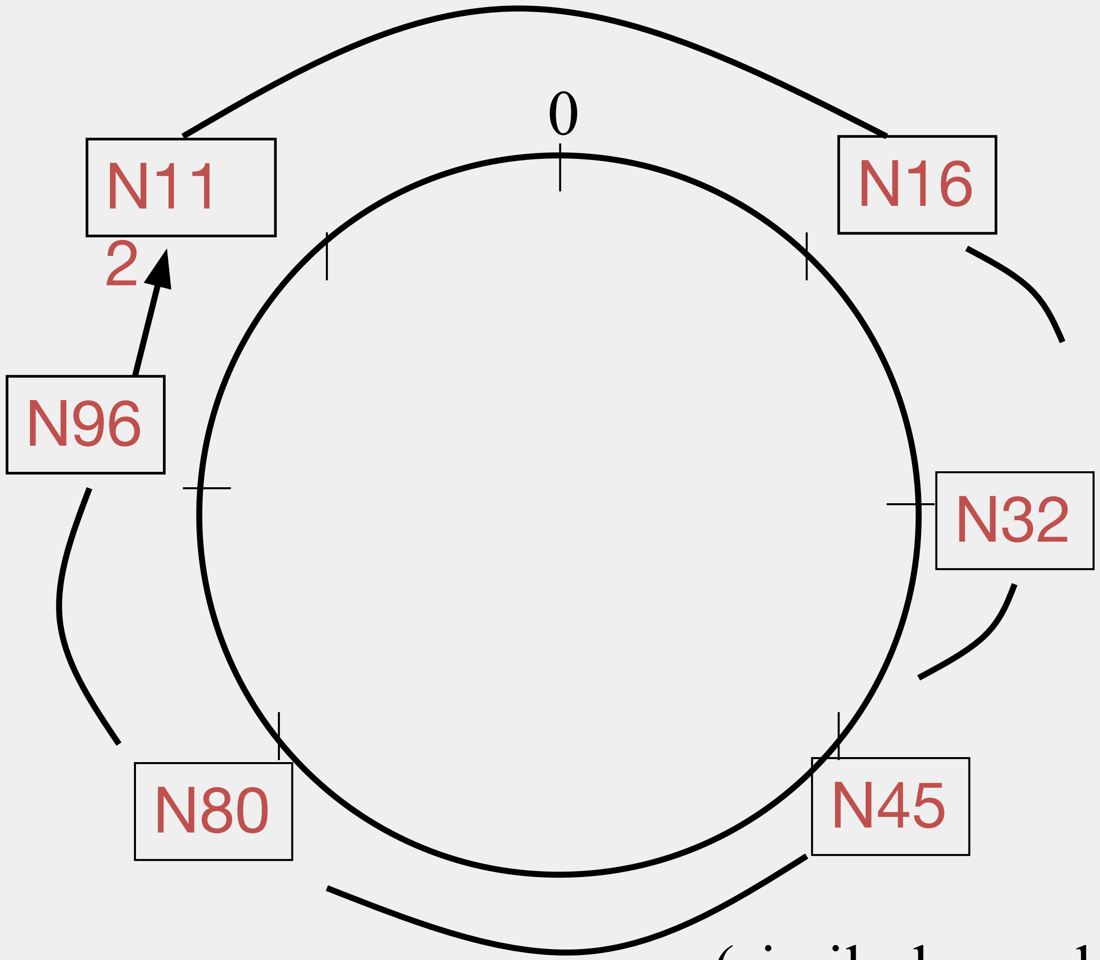
Say $m=7$

6 nodes



Peer pointers (1): successors

Say m=7



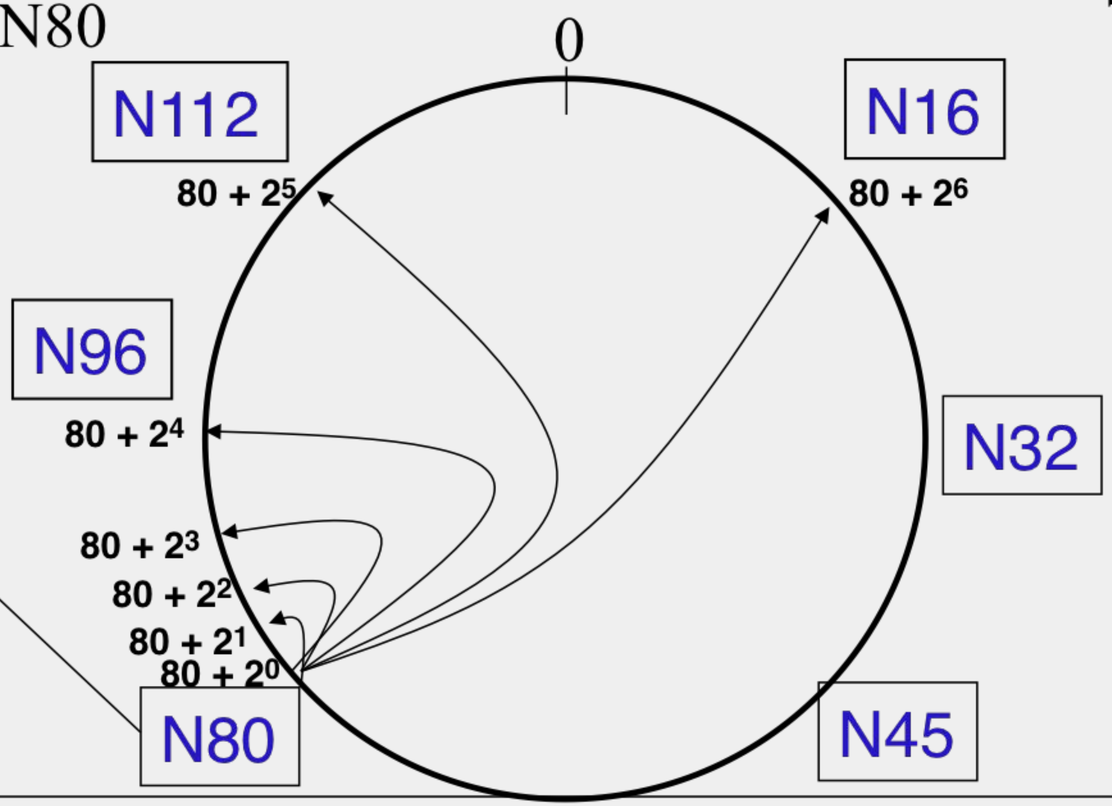
(similarly predecessors)

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



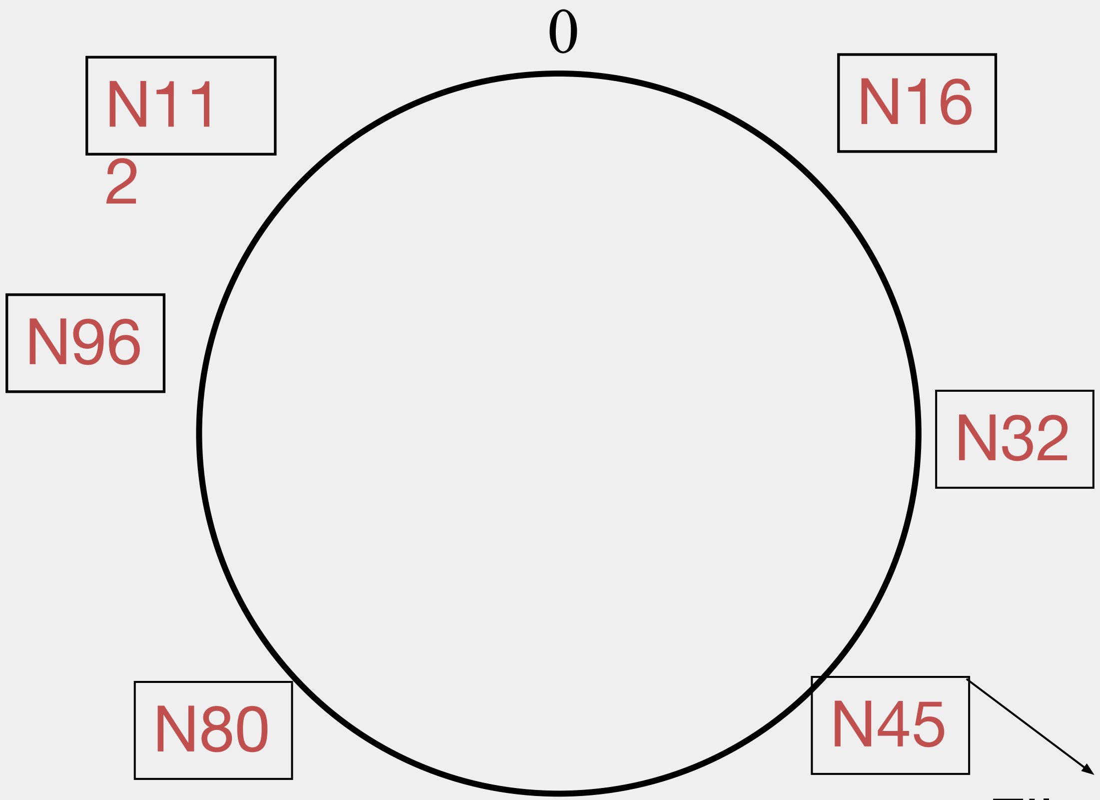
ith entry at peer with id n is first peer with id >= $n + 2^i (\mathrm{mod}\, 2^m)$

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 or equal to 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

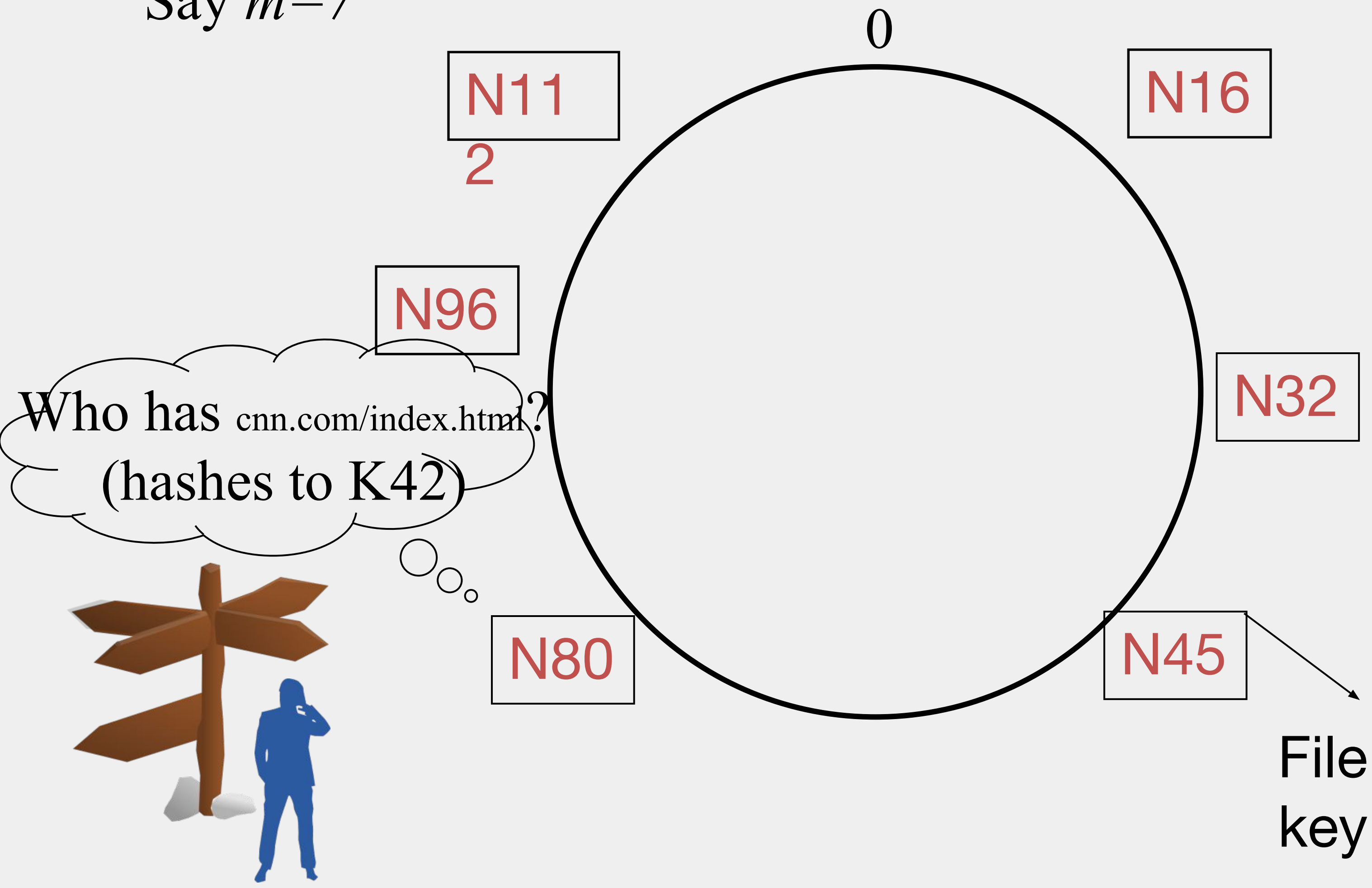
Say m=7



File with key K42
stored here

Search

Say $m=7$



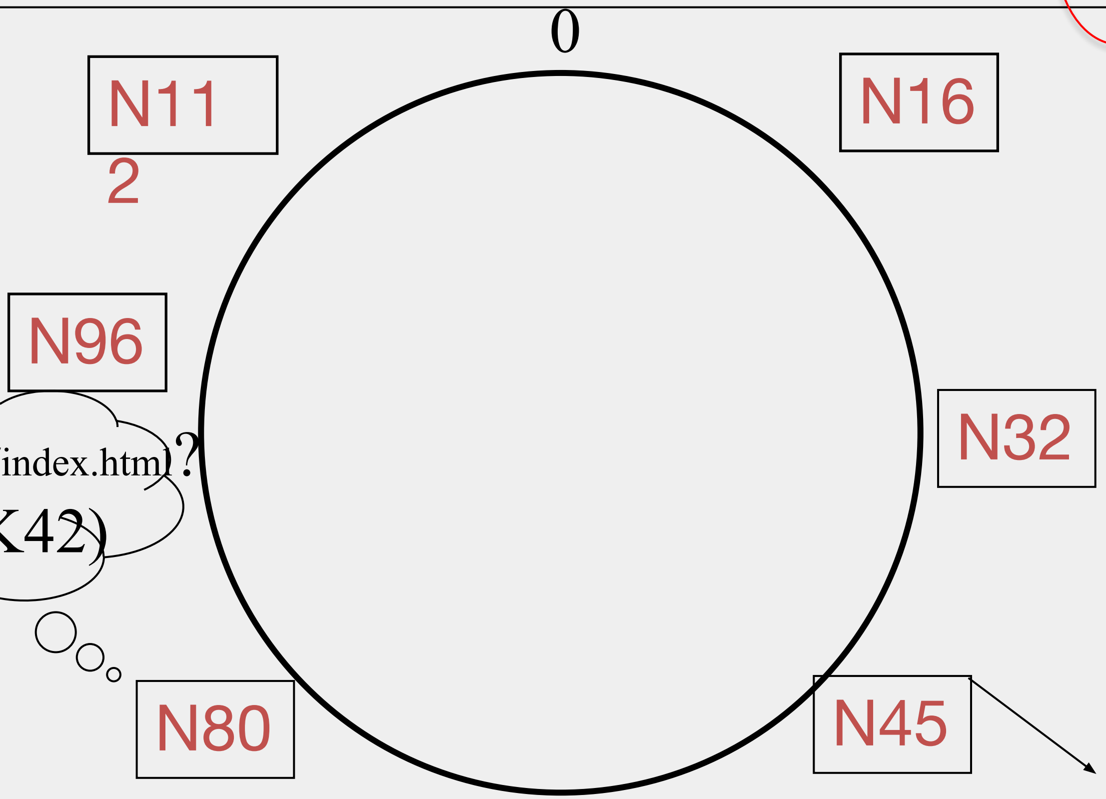
File cnn.com/index.html with key K42 stored here

Search

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

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

Say $m=7$



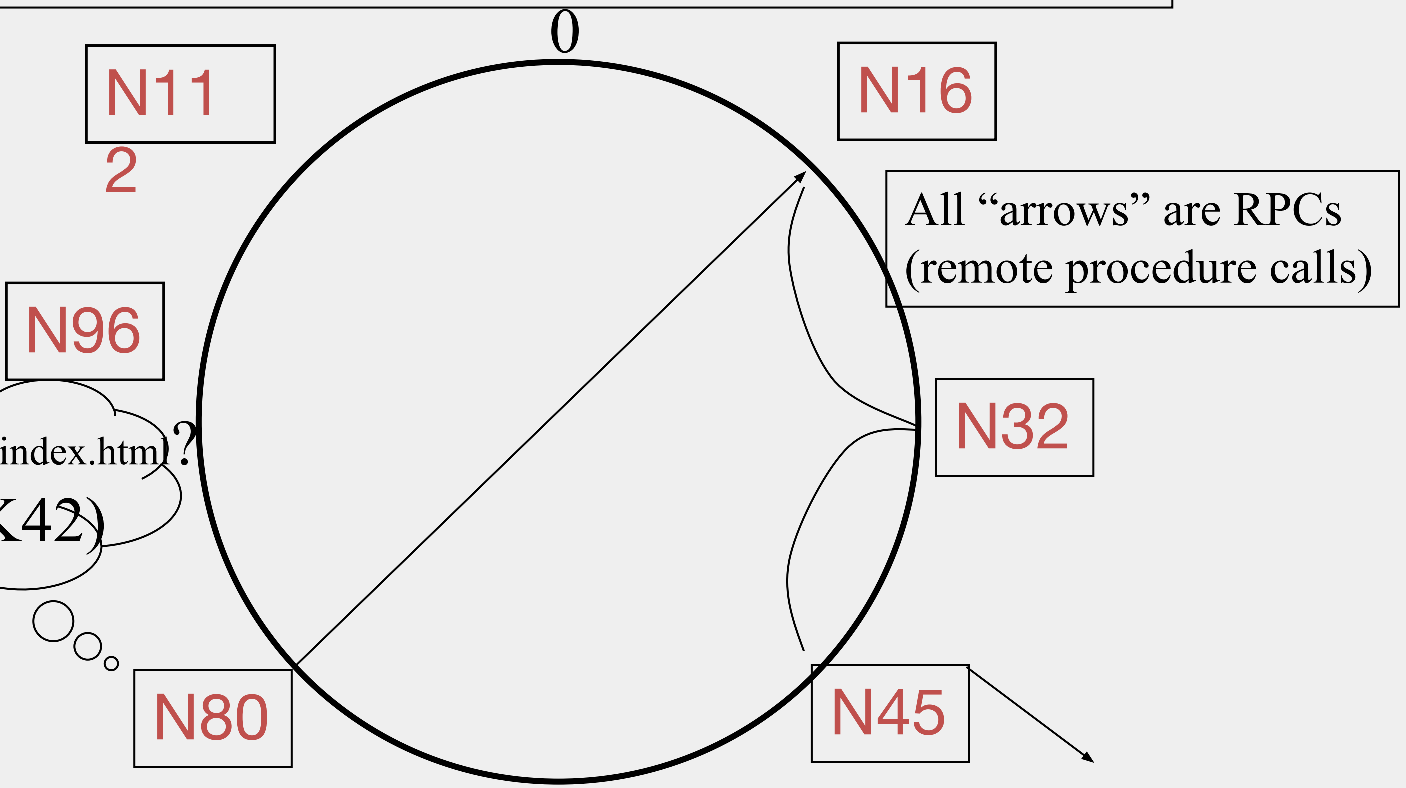
Who has cnn.com/index.html?
(hashes to K42)

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 $m=7$

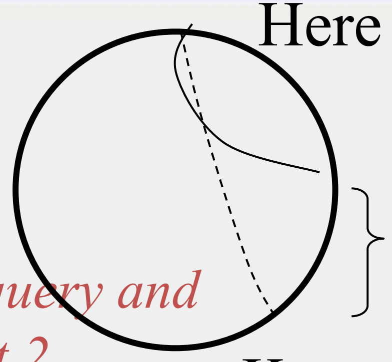


Who has cnn.com/index.html?
(hashes to K42)

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

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

Analysis



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

Proof

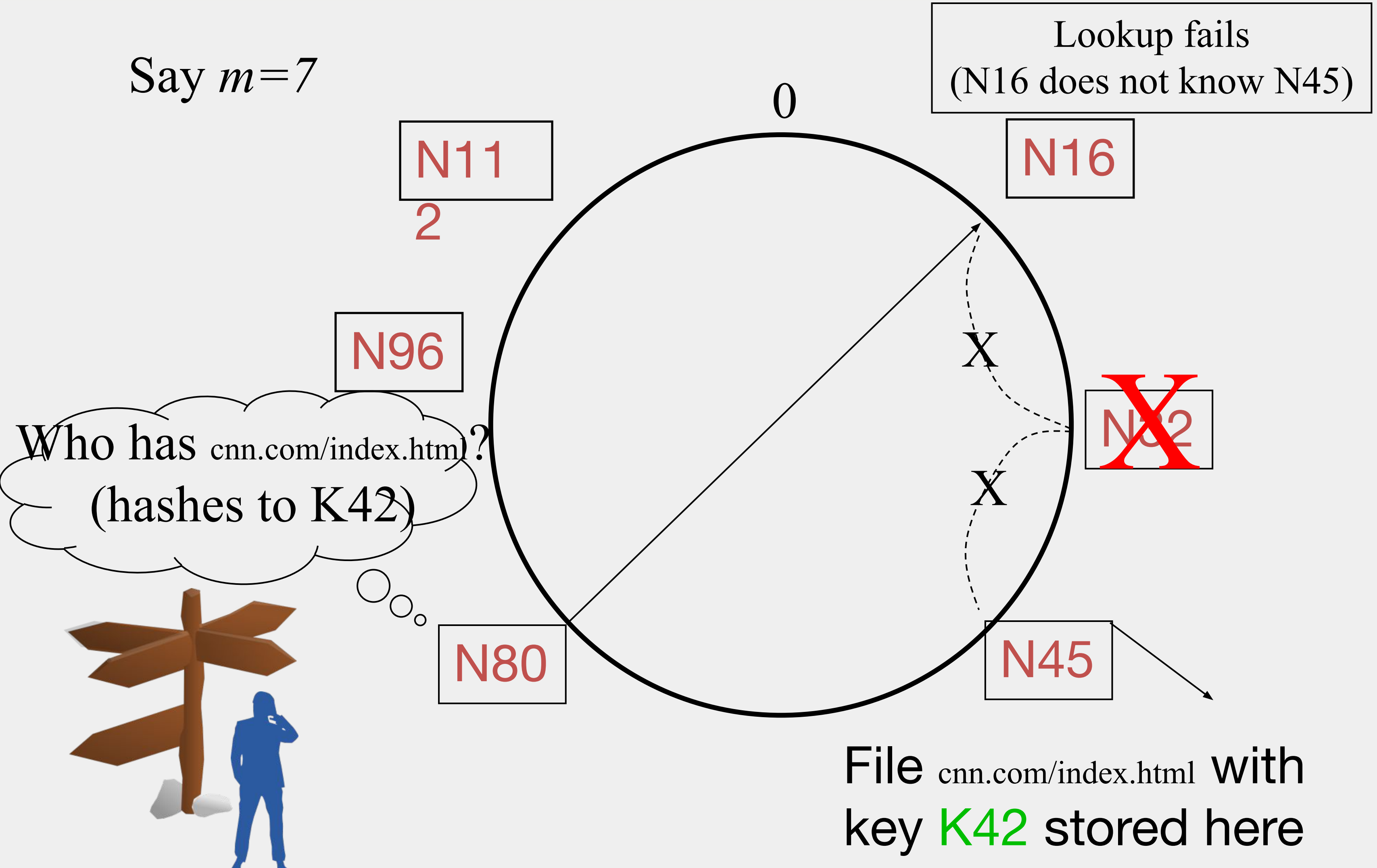
- (intuition): *at each step, distance between query and peer-with-file reduces by a factor of at least 2*
- (intuition): after *log(N)* forwardings, distance to key is at most $2^m / 2^{\log(N)} = 2^m / N$
- Number of node identifiers in a finite range is *O(log(N))* with high probability (why? SHA-1!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

  So using *successor*s 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

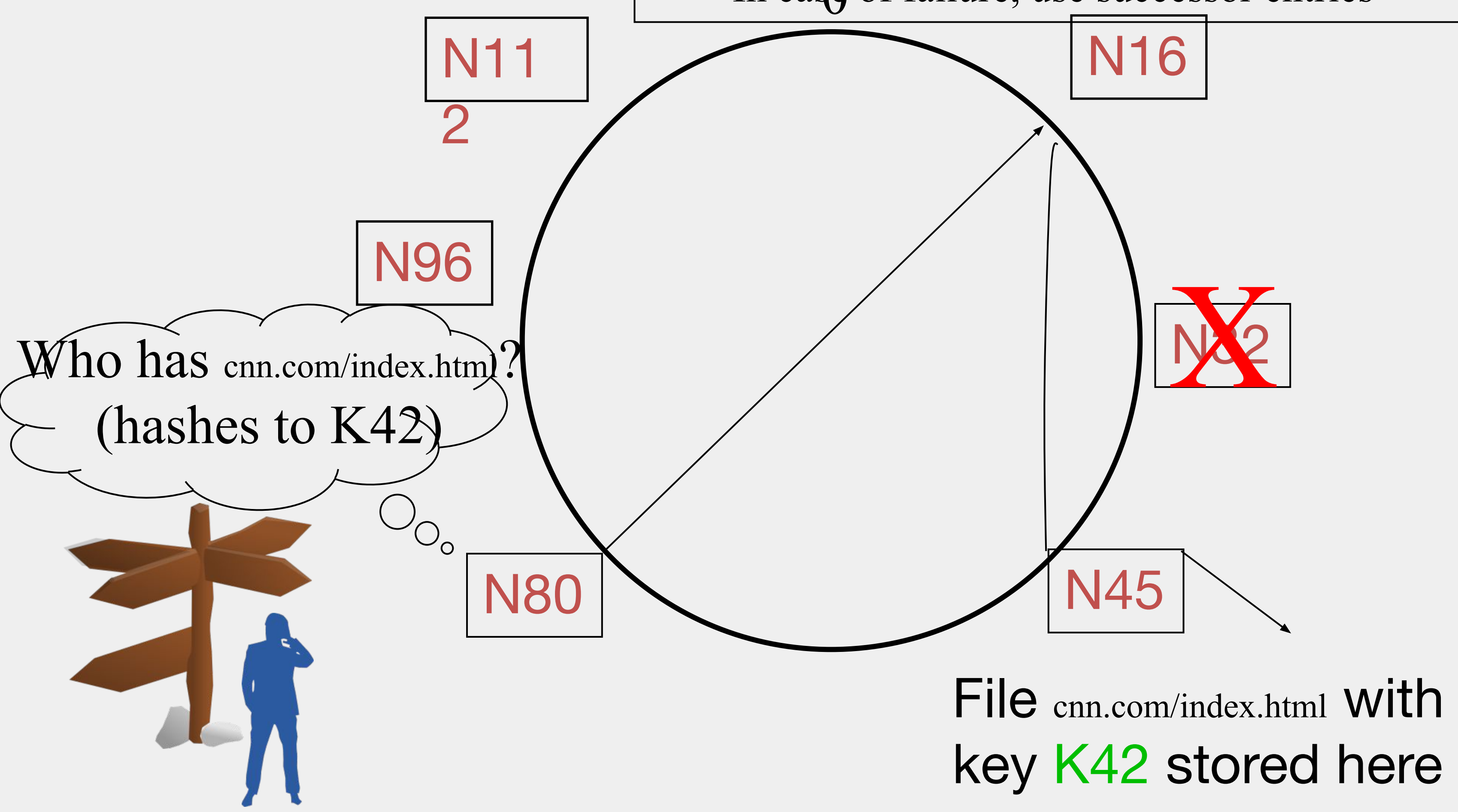
Search under peer failures



Search under peer failures

Say m=7

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



Search under peer failures

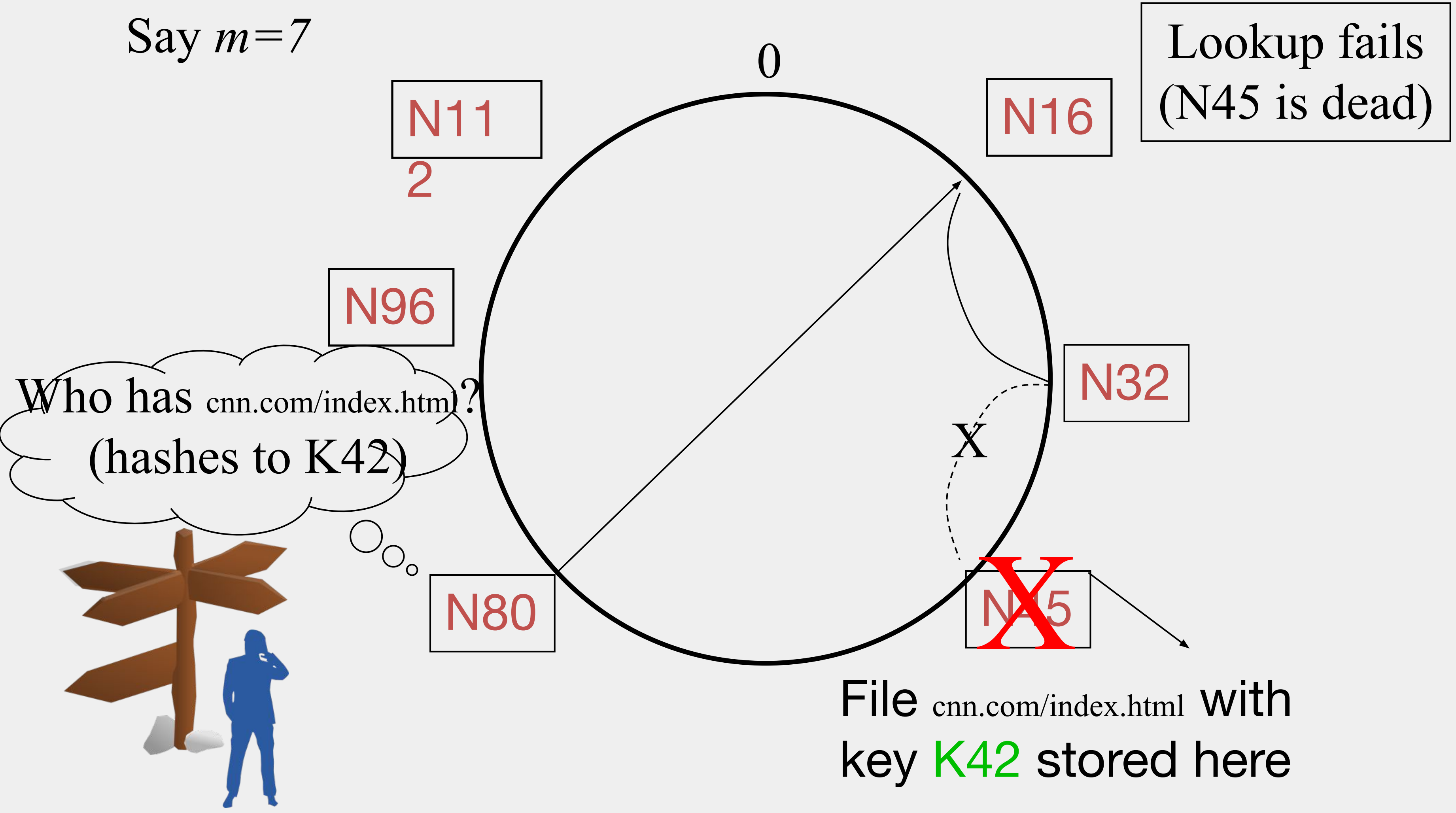
- Choosing *r=2log(N)* suffices to maintain *lookup correctness* w.h.p.(i.e., ring connected)
  - Say 50% of nodes fail
  - Pr(at given node, at least one successor alive)=

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

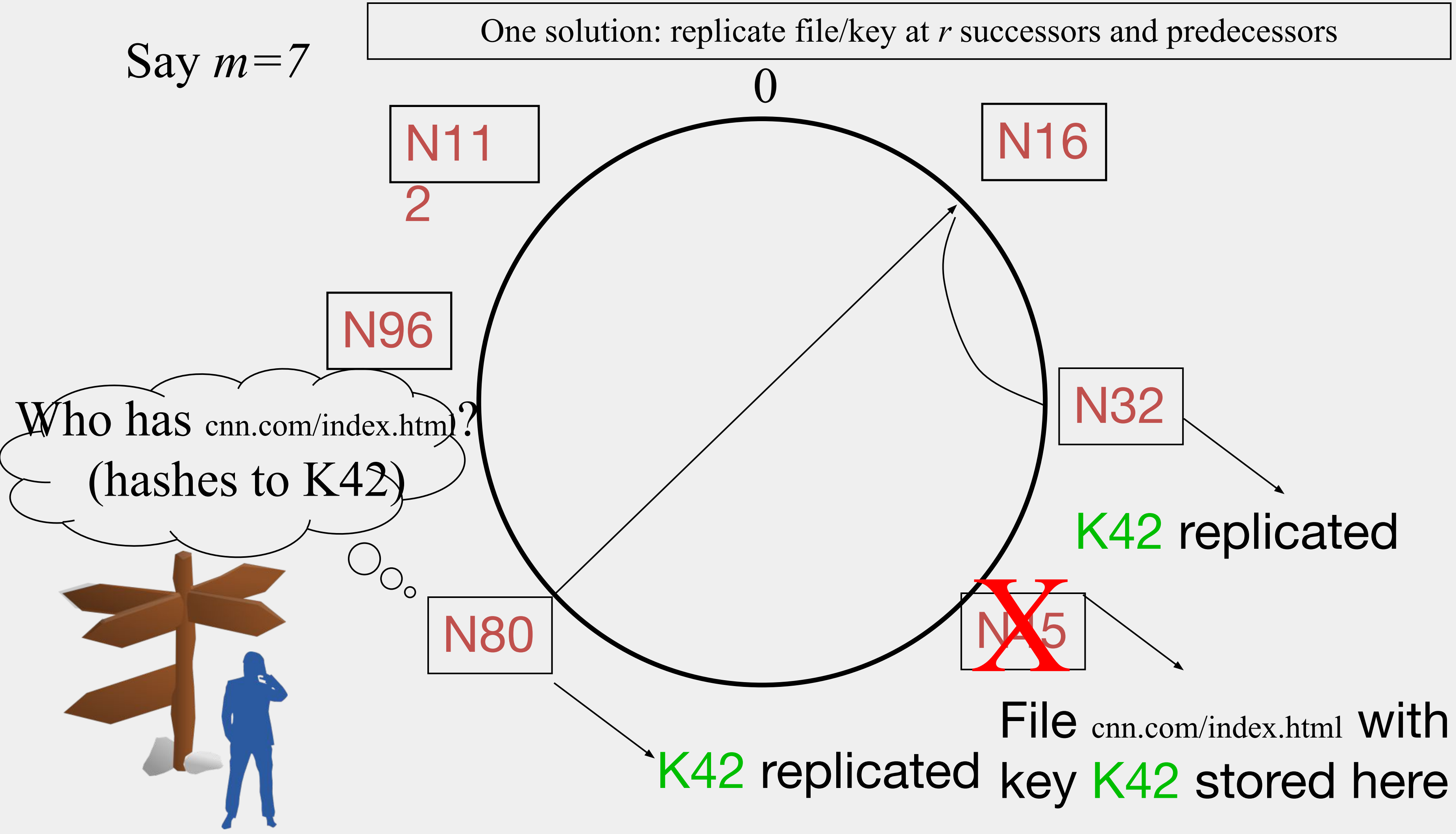
  - Pr(above is true at all alive nodes)=

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

Search under peer failures (2)



Search under peer failures (2)



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.

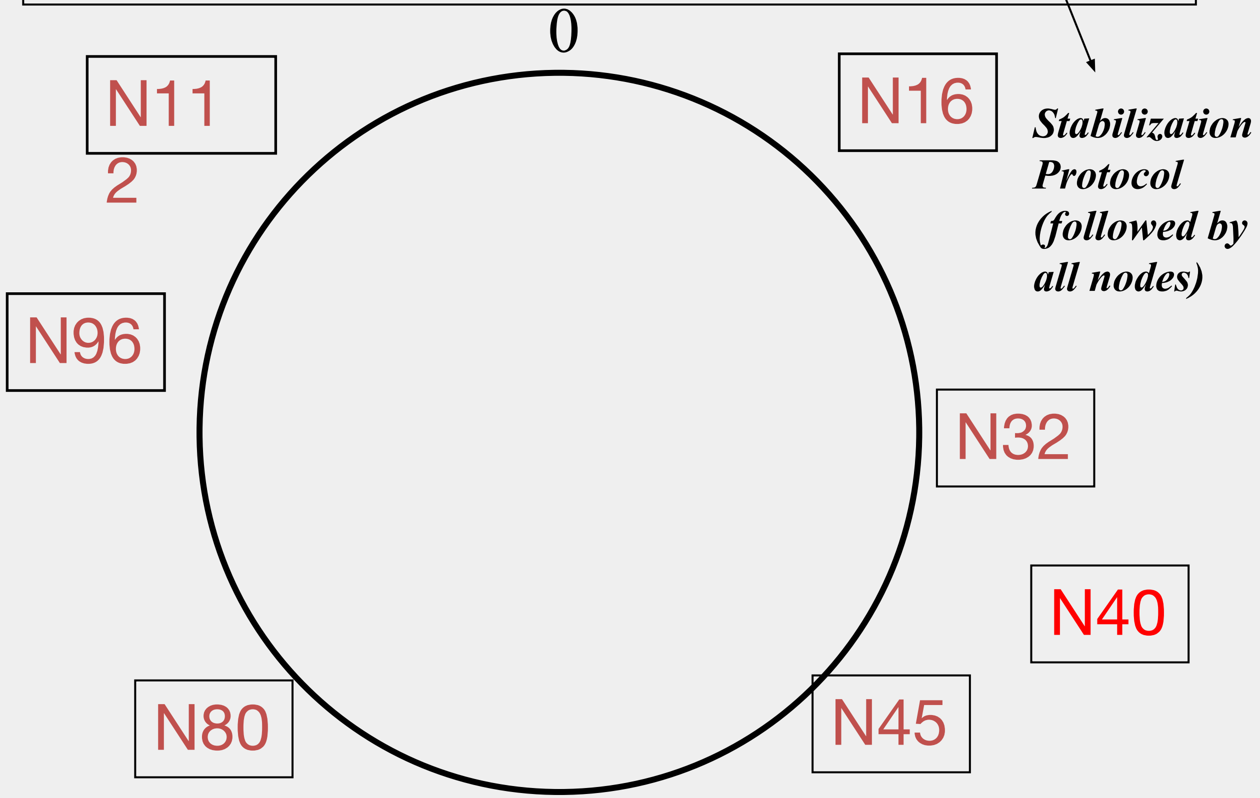
So, all the time, need to:

 Need to update *successor*s and *finger*s, 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

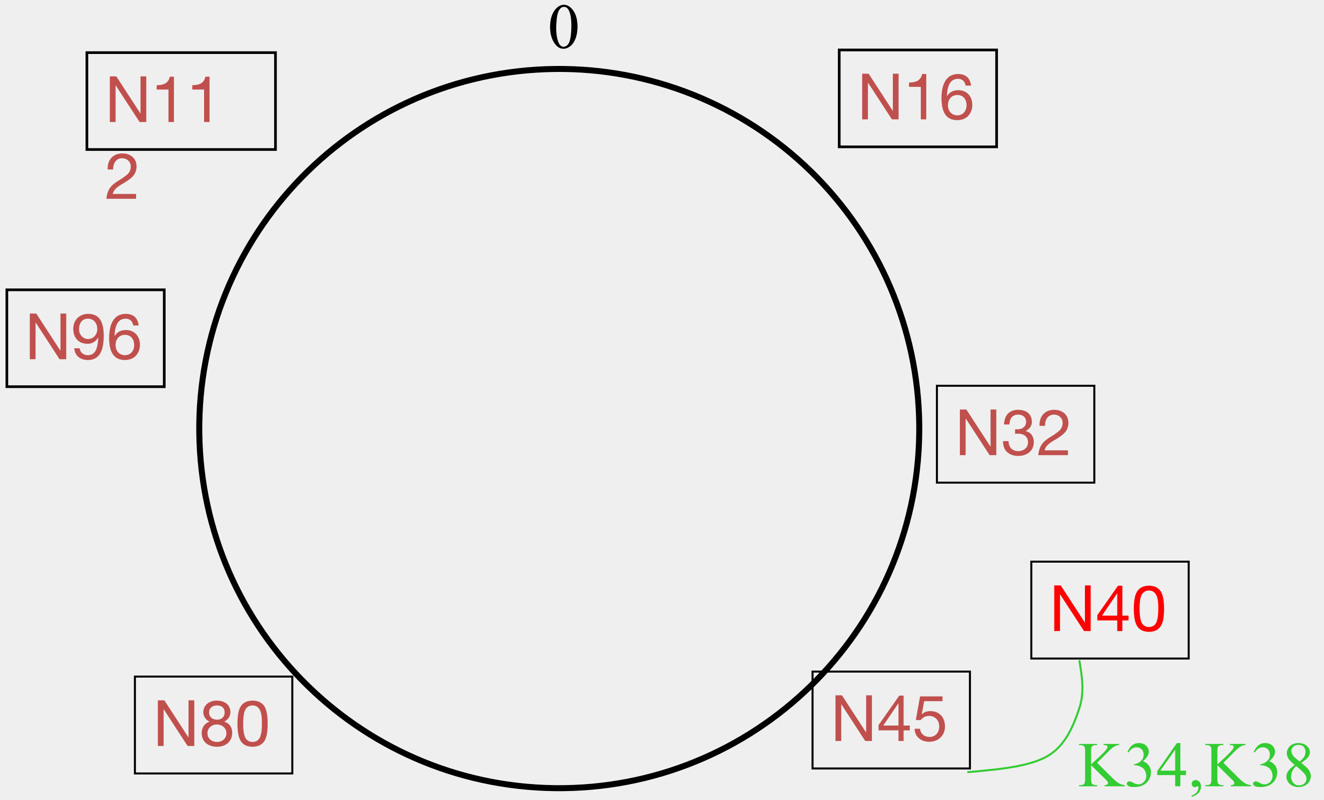
Say m=7



New peers joining (2)

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

Say m=7



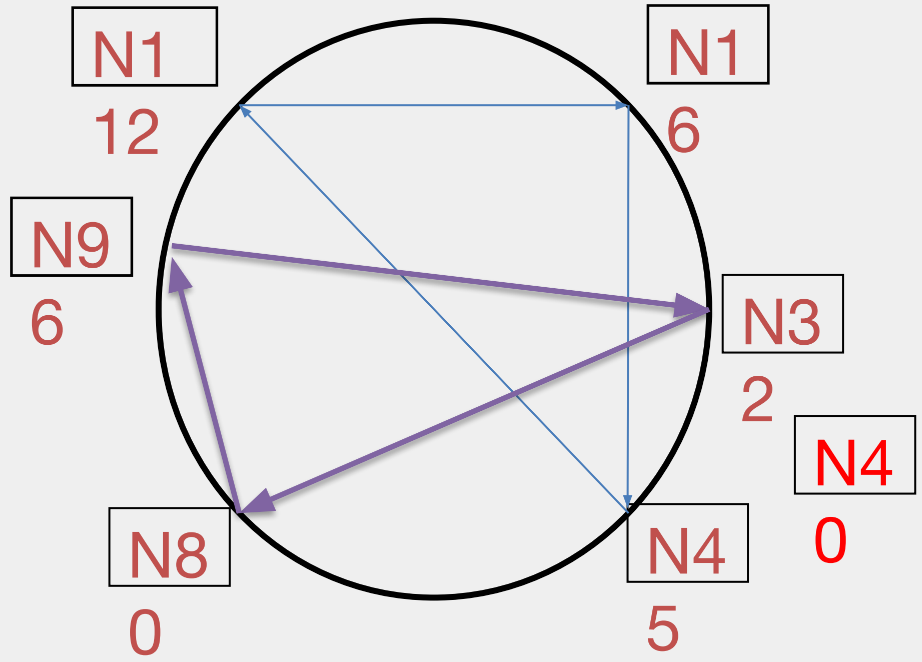
New peers joining (3)

- 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*

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

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

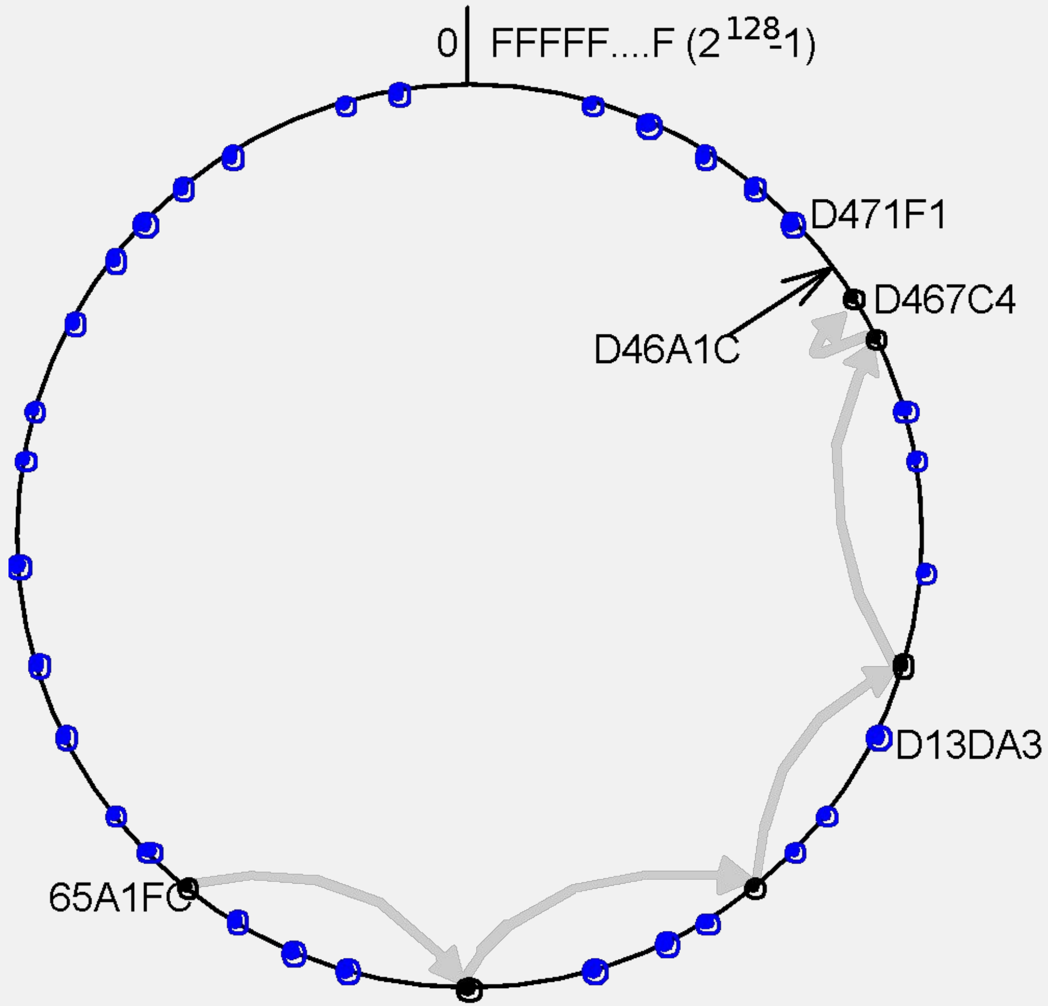
http://www.pdos.lcs.mit.edu/chord/

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), a total of $2l$ of them

Use Leaf Set for Routing

- An example of $l = 4$
- Number of messages is about $\frac{N}{2l}$
- Performance is poor



Pastry Neighbors

- **Routing tables** based **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

- Address is specified using hexadecimal number
  - Each digit has 16 possibilities
- Consider a peer with id 65A1FC. It maintains a neighbor peer with an id matching each of the following prefixes:
  - *
  - 6*
  - 65*
  - …
  - * = any number
- When it needs to route to a peer, say 65A2BC, it starts by forwarding to a neighbor with the largest matching prefix, i.e., 65A2XX

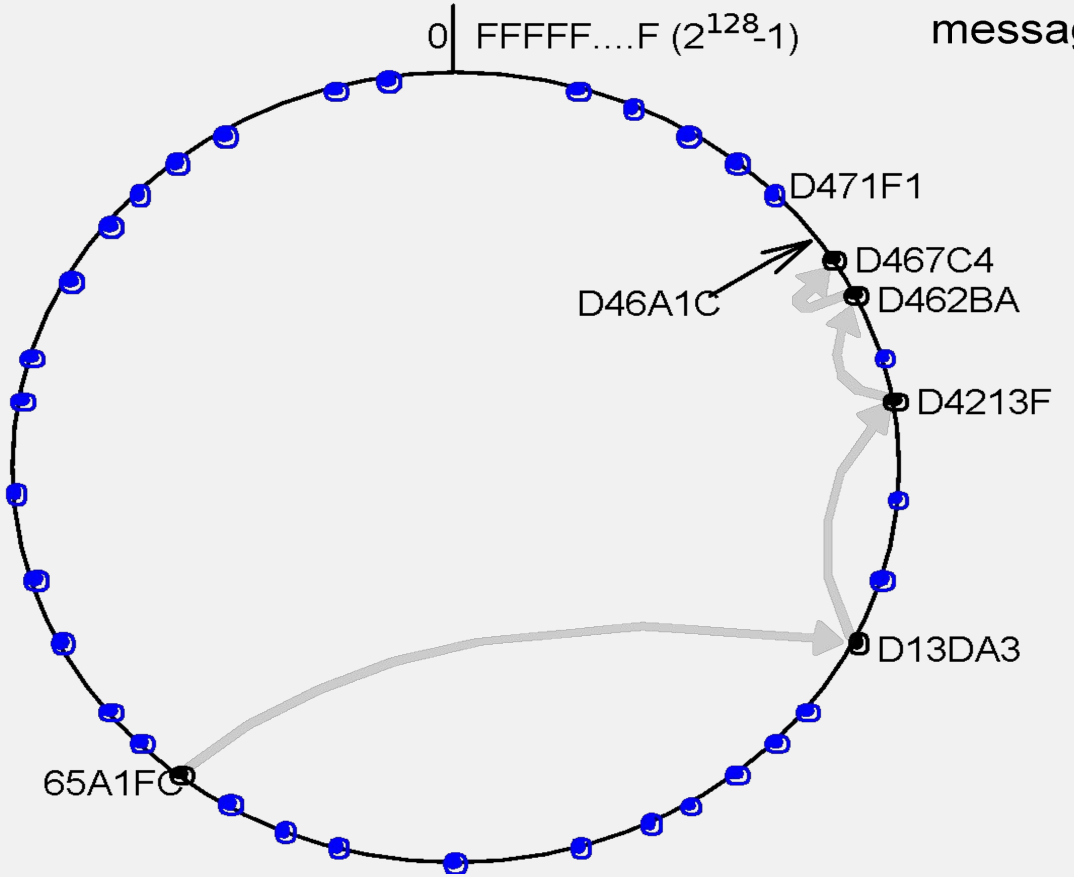
Routing Table of 65A1FC

p =	*GUID prefixes and corresponding nodehandles n*															
0	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*A*	*B*	*C*	*D*	*E*	*F*
	n	*n*	*n*	*n*	*n*	*n*		*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*
1	*60*	*61*	*62*	*63*	*64*	*65*	*66*	*67*	*68*	*69*	*6A*	*6B*	*6C*	*6F*	*6E*	*6F*
	n	*n*	*n*	*n*	*n*		*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*
2	*650*	*651*	*652*	*653*	*654*	*655*	*656*	*657*	*658*	*659*	*65A*	*65B*	*65C*	*65D*	*65E*	*65F*
	n	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*		*n*	*n*	*n*	*n*	*n*
3	*65A0*	*65A1*	*65A2*	*65A3*	*65A4*	*65A5*	*65A6*	*65A7*	*65A8*	*65A9*	*65AA*	*65AB*	*65AC*	*65AD*	*65AE*	*65AF*
	n		*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*

A node whose id is 65Dxxxx

The routing table is located at a node whose GUID begins 65A1. Digits are in hexadecimal. The *n*'s represent [GUID, IP address] pairs specifying the next hop to be taken by messages addressed to GUIDs that match each given prefix. Grey- shaded entries indicate that the prefix matches the current GUID up to the given value of *p*: the next row down or the leaf set should be examined to find a route. Although there are a maximum of 128 rows in the table, only log16 *N* rows will be populated on average in a network with *N* active nodes.

Routing: From 65A1FC to D46A1C



Routing a message from node 65A1FC to D46A1C. With the aid of a well-populated routing table the message can be delivered in ~ $\log_{16}(N)$ hops.

Pastry Locality

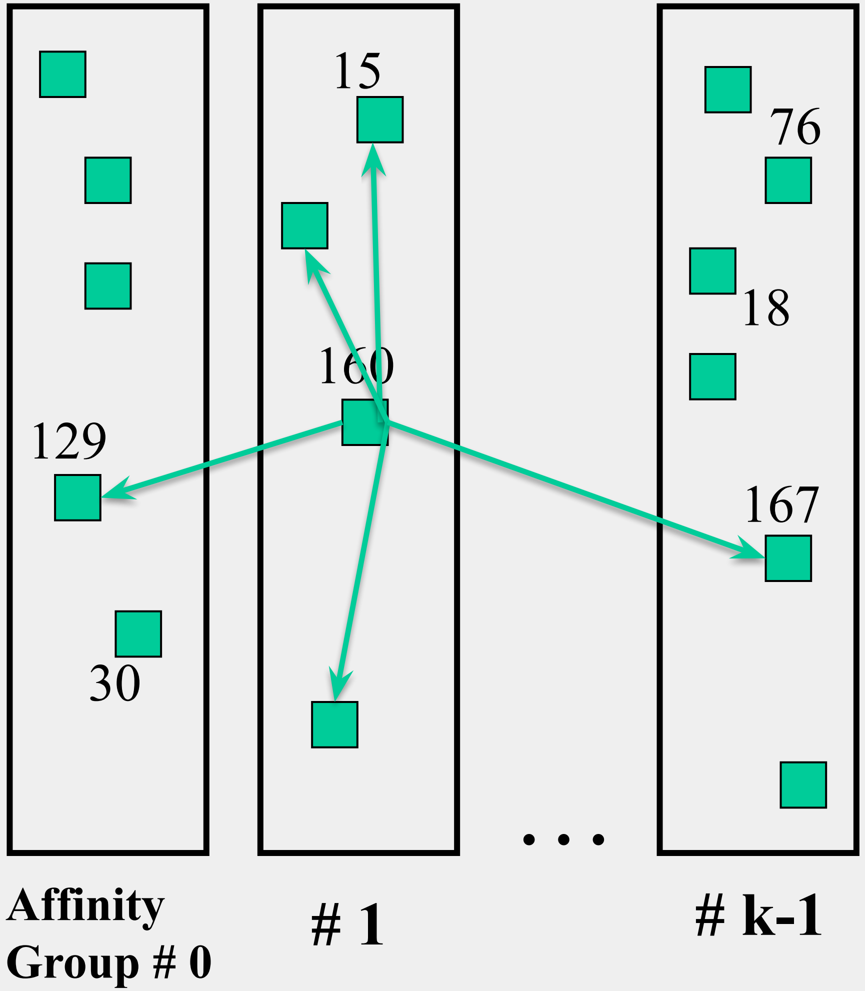
- For each prefix, say D46*, among all potential neighbors with the matching prefix, the neighbor with the shortest round-trip-time is selected
- Since shorter prefixes have many more candidates (spread out throughout the Internet), the neighbors for shorter prefixes are likely to be closer than the 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?

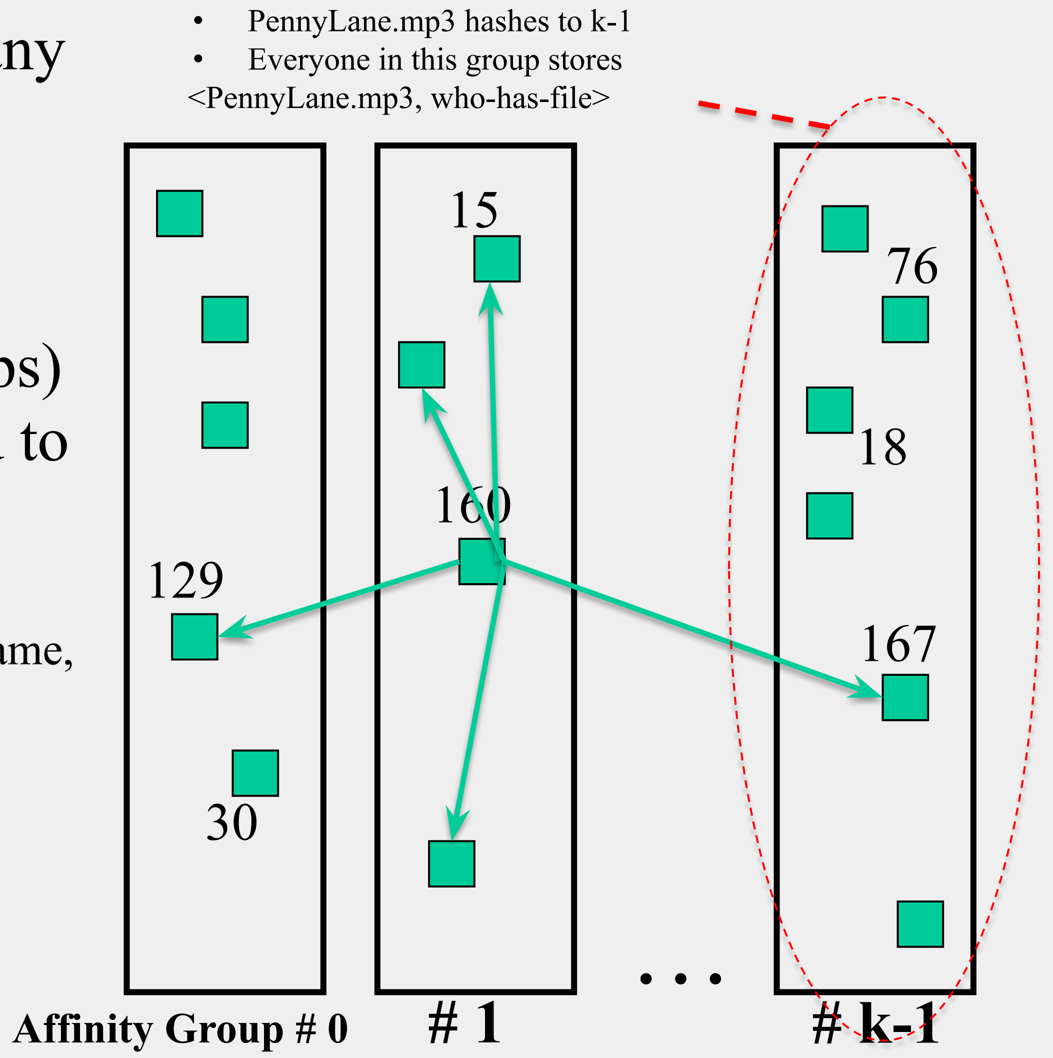
Kelips – A 1 hop Lookup DHT

- k "affinity groups"
  - k ~ √ 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



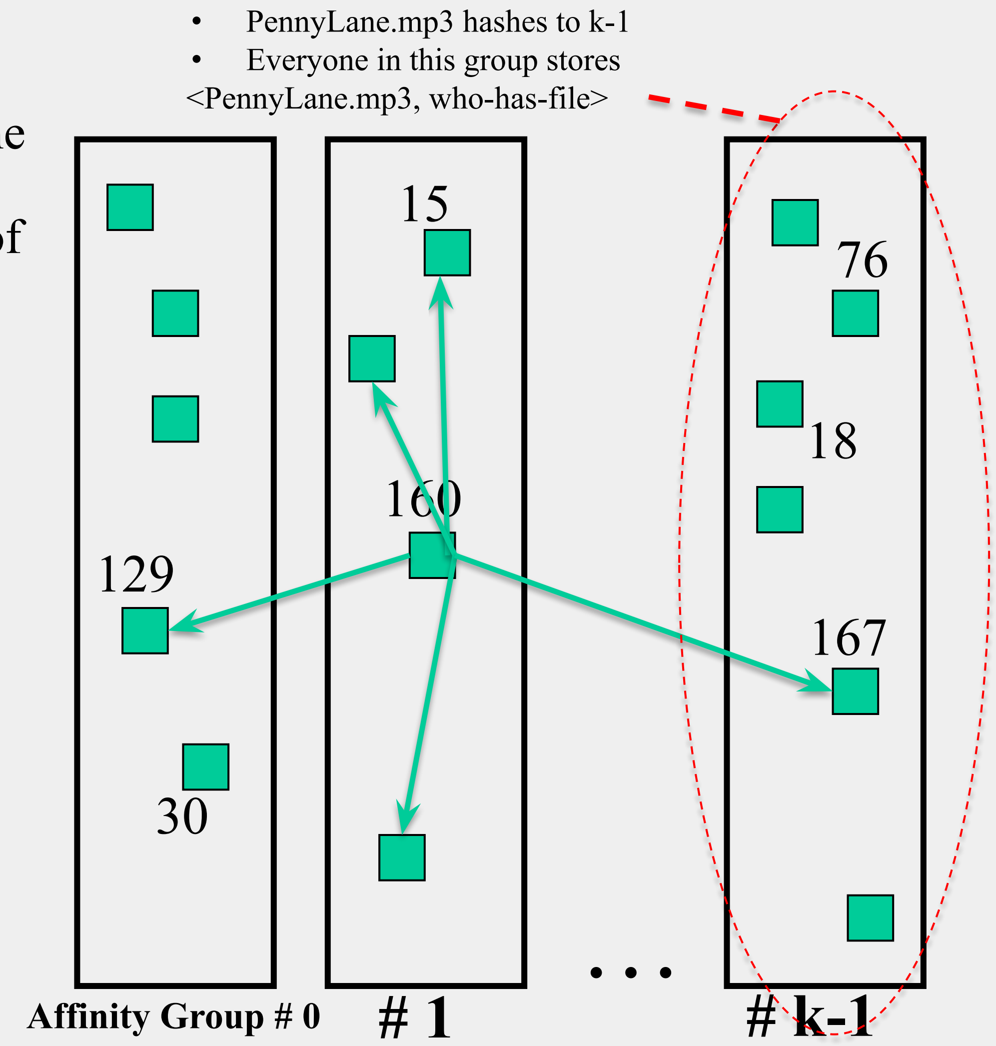
Kelips Files and Metadata

- 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>
  - Affinity group <u>does not</u> store files



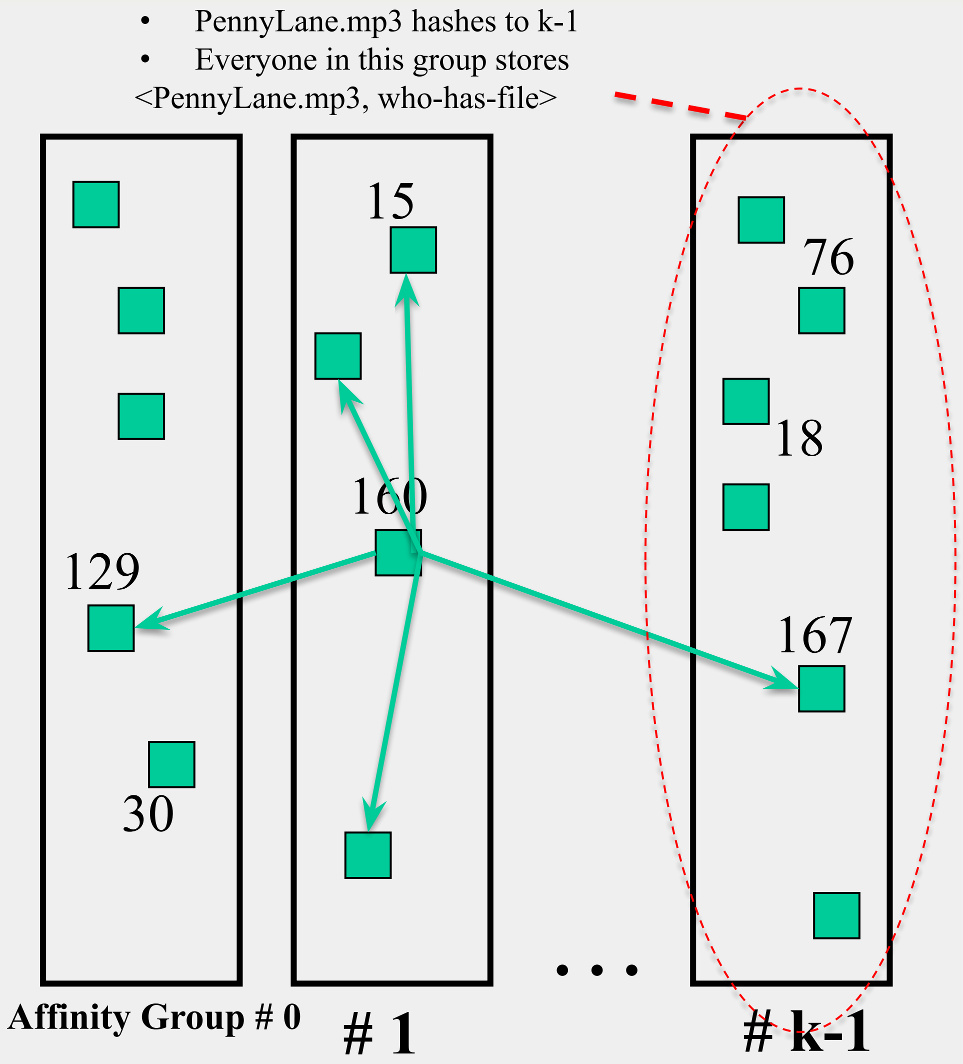
Kelips Lookup

- 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(√ N)
  - 1.93 MB for 100K nodes, 10M files
  - Fits in RAM of most workstations/laptops today (COTS machines)



Kelips Soft State

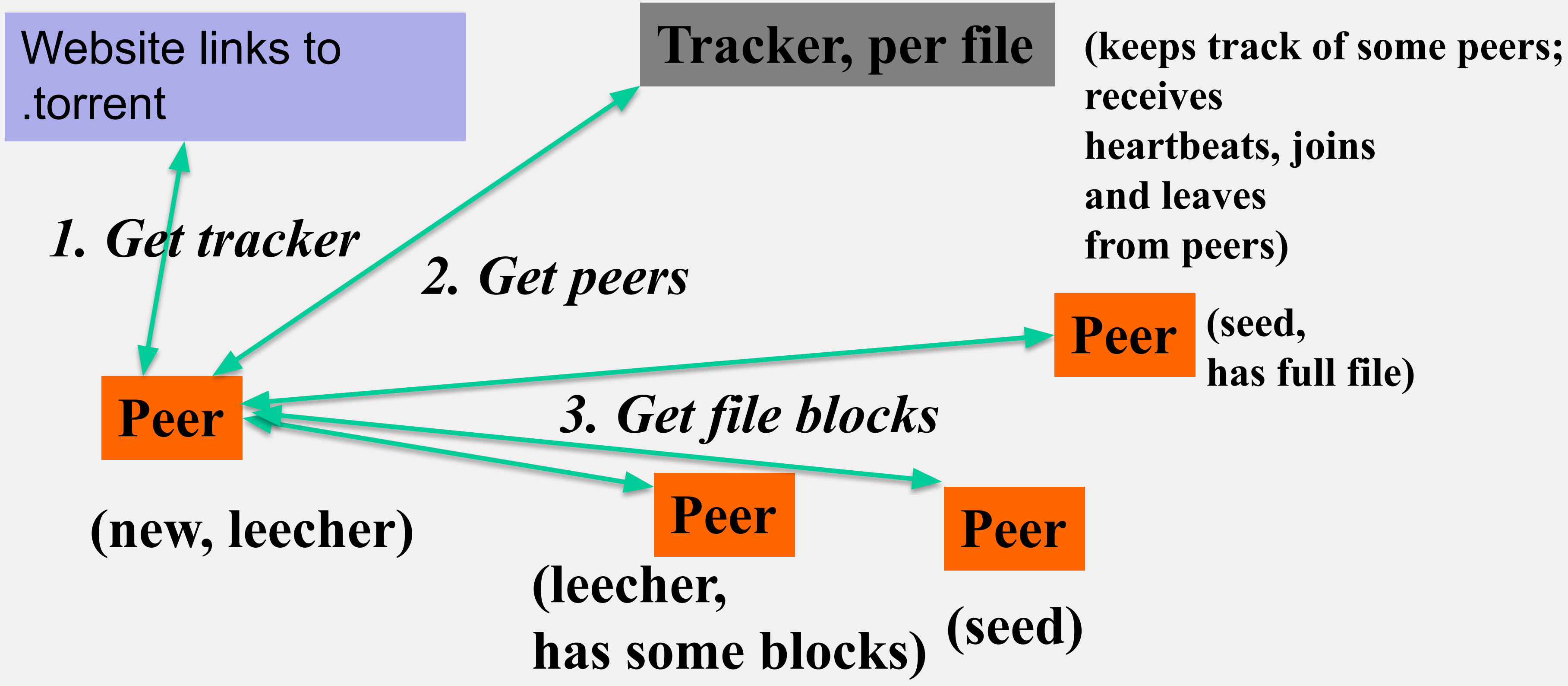
- Membership lists
  - Within each affinity group
  - And also across affinity groups
  - *O(log(N))* dissemination time
- File metadata
  - Needs to be periodically refreshed from source node
  - Times out



Chord vs. Pastry vs. Kelips

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

BitTorrent


Website links to .torrent
Tracker, per file
(keeps track of some peers; 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

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