Computer Science 425
Distributed Systems

CS 425 / ECE 428
Fall 2013

Indranil Gupta (Indy)

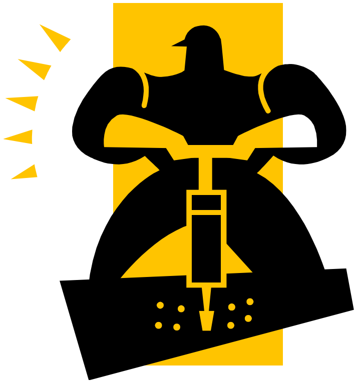
October 1, 2013

Lecture 11

Peer-to-peer Systems II

Reading: Chord paper on website (Sec 1-4, 6-7)

Two types of P2P Systems



Systems that work well in practice but with no big/famous names

• ***Non-academic P2P systems***

e.g., Napster, Gnutella, BitTorrent (previous lecture)

Systems with big/famous names from academia, with varied uses

• ***Academic P2P systems***

e.g., Chord (this lecture)

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)
- DHTs are inspiration for key-value store in a cloud
- Performance Concerns:
  - Load balancing
  - Fault-tolerance
  - Efficiency of lookups and inserts
- 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))*

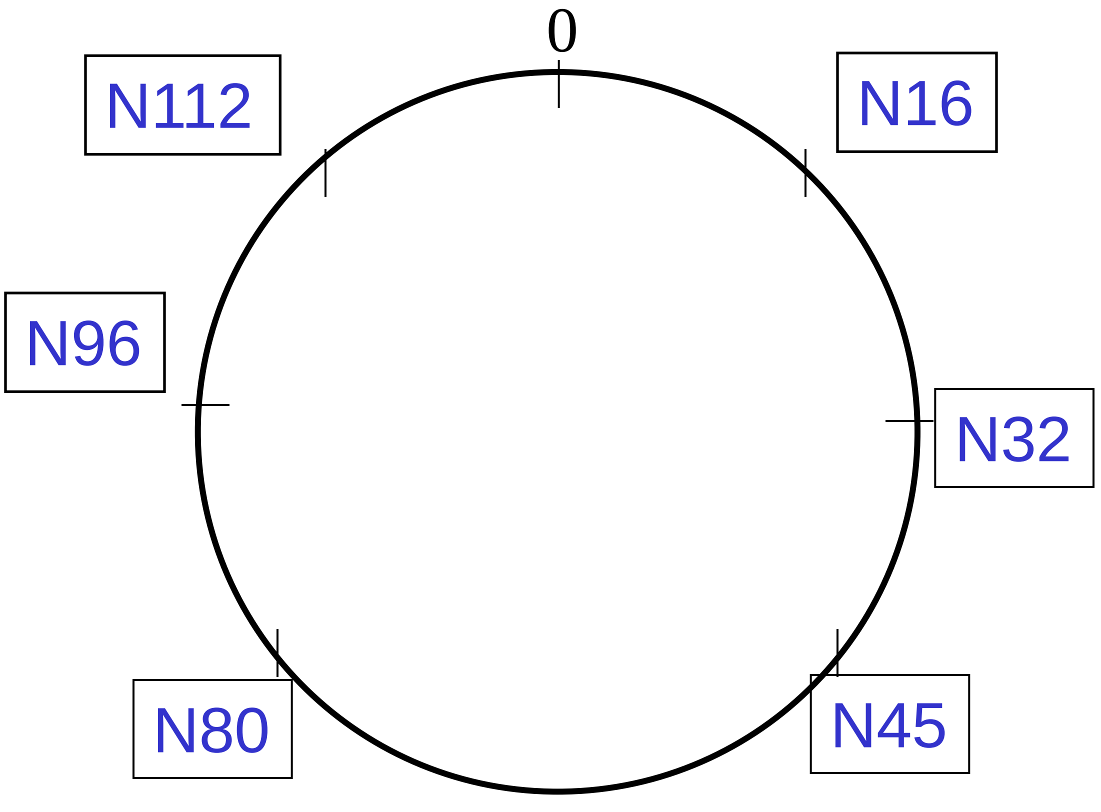
Chord

- 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 (m ~ 128)
  - Can then map peers to one of 2^m logical points on a circle

Ring of peers

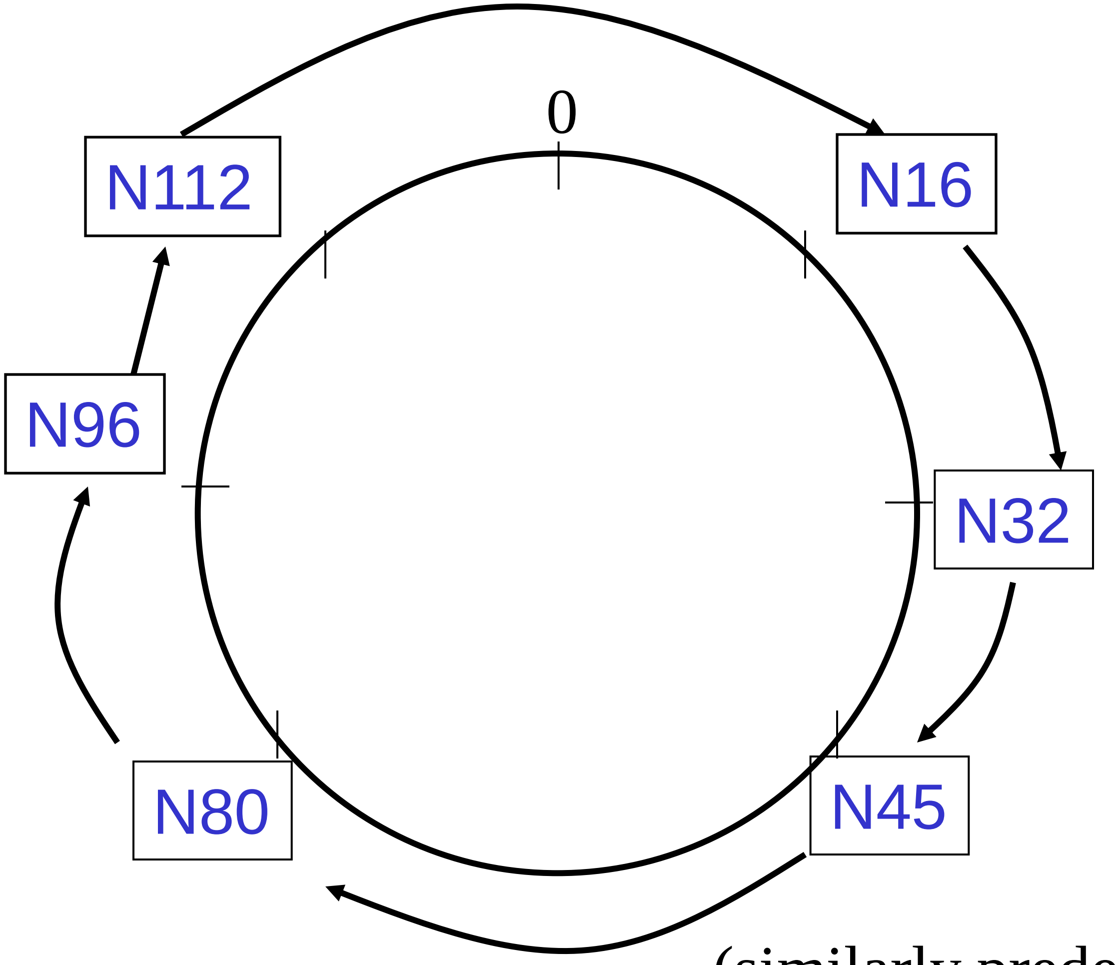
Say *m=7*

6 nodes



Peer pointers (1): *successors*

Say *m=7*



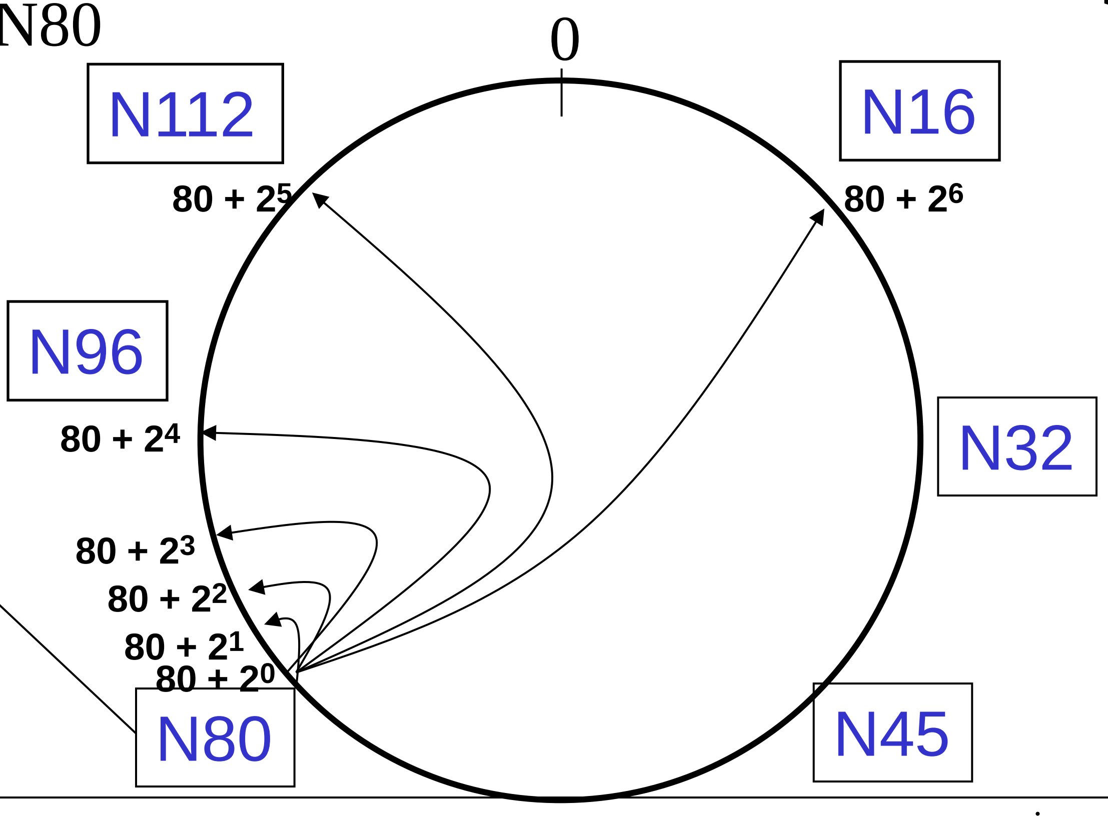
(similarly predecessors)

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



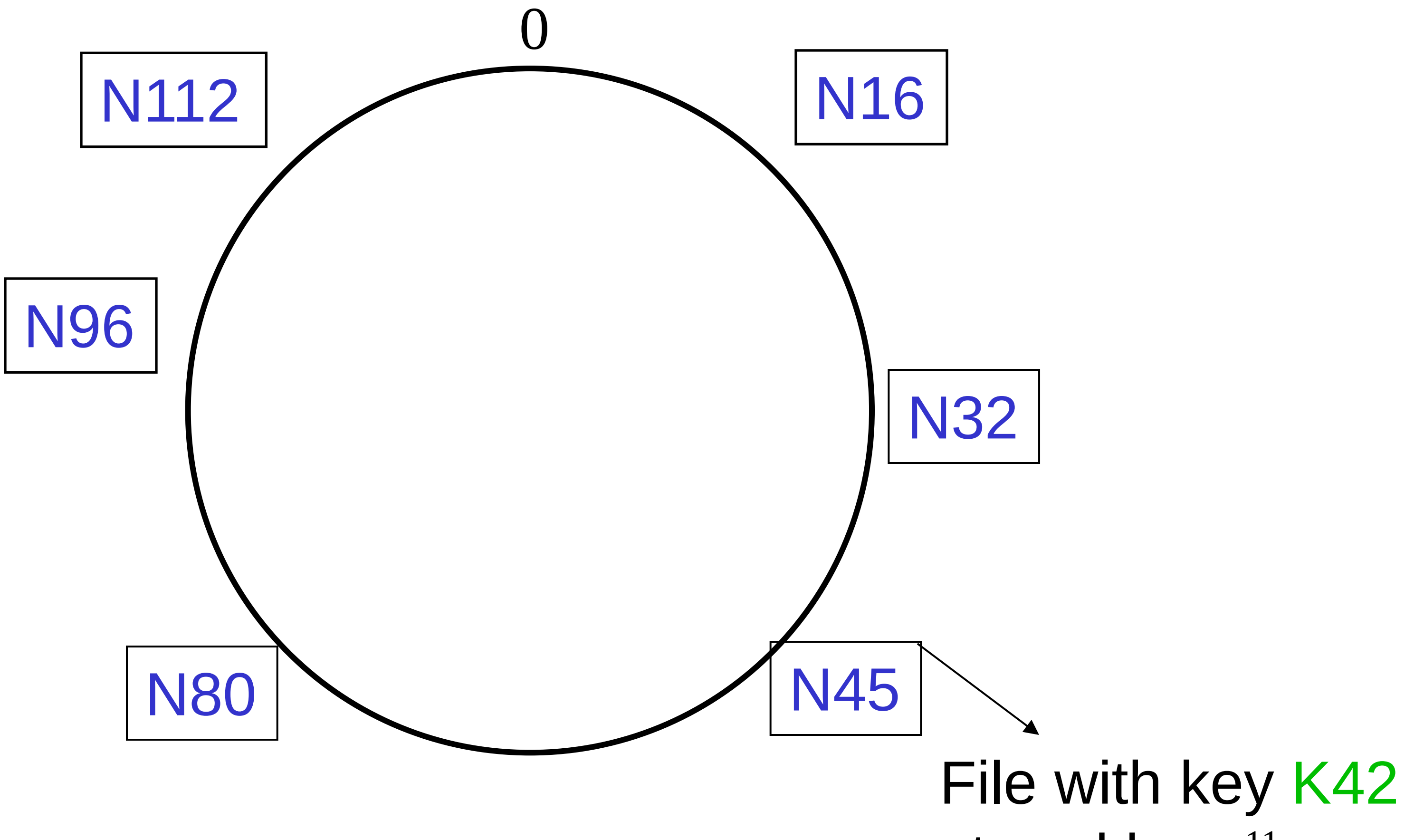
*i*th entry at peer with id *n* is first peer with id >= $n + 2^i (\mathrm{mod}\, 2^m)$

What about the files?

- Filenames also mapped using same consistent hash function
  - SHA-1(filename) →160 bit string (*key*), truncate to m
  - File is stored at first peer with id greater than 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
  - If you store webpages this way, it's called *cooperative web caching* (~ Memcached architecture)
  - Generic though

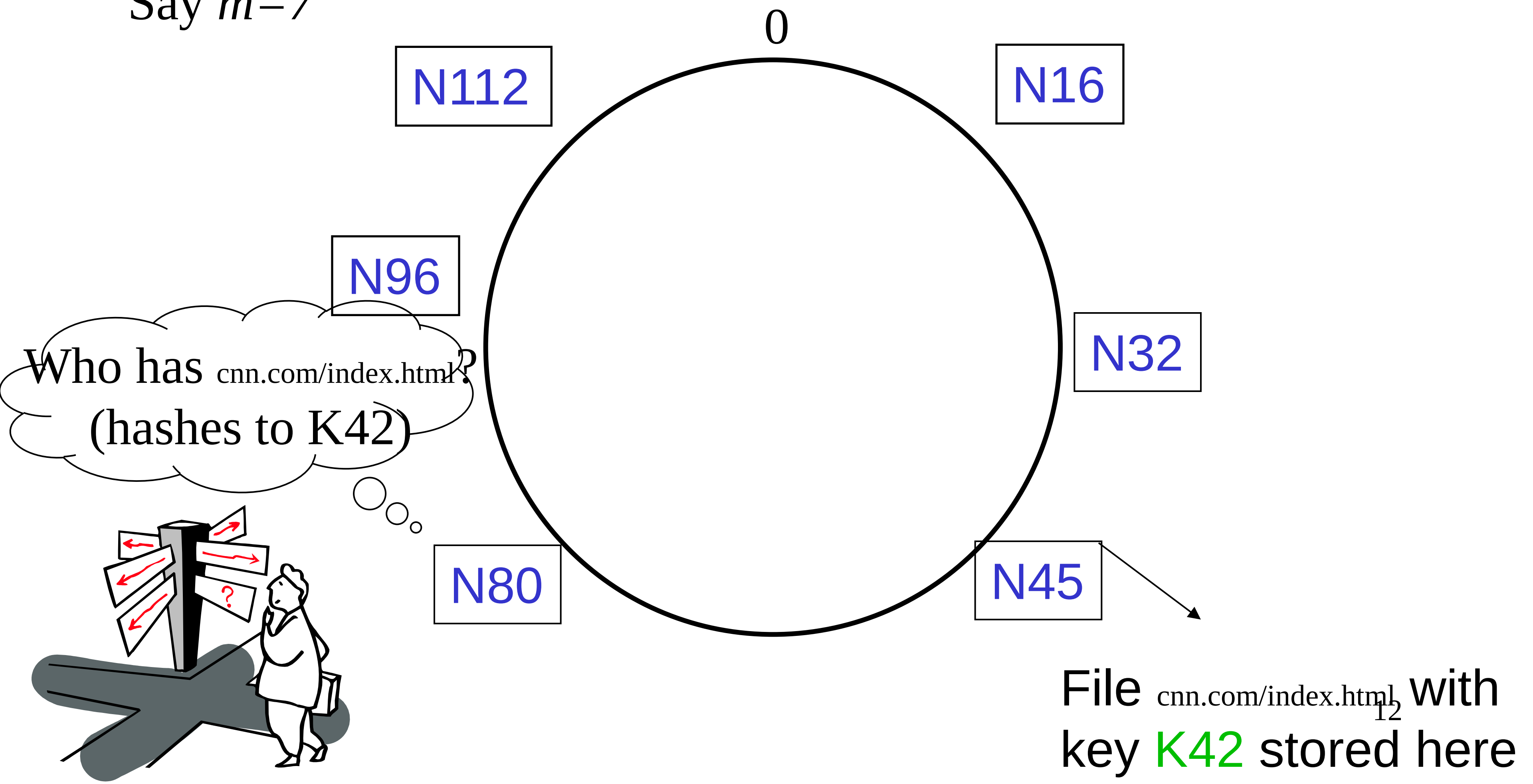
Mapping Files

Say $m=7$



Search

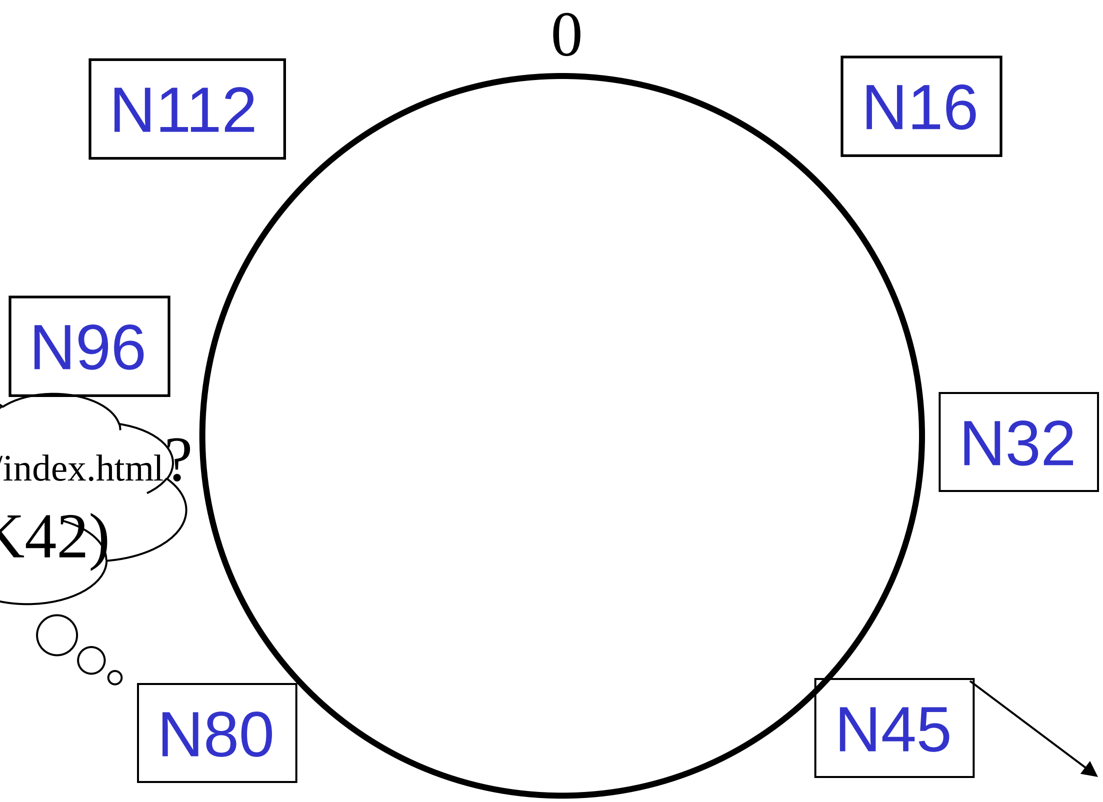
Say $m=7$

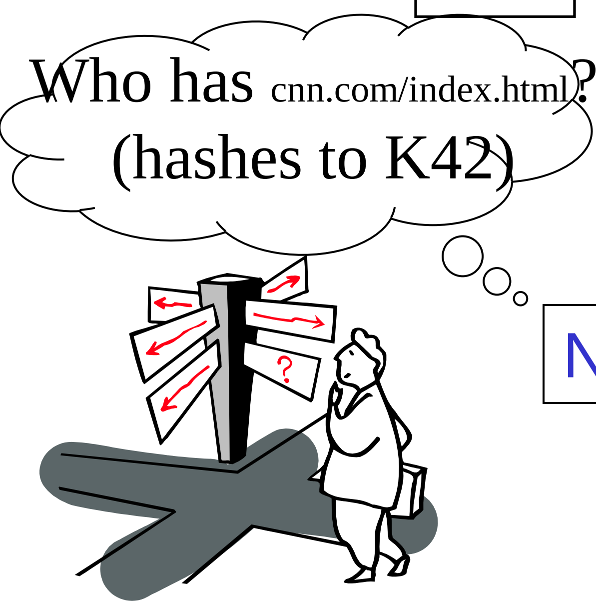


Search

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*



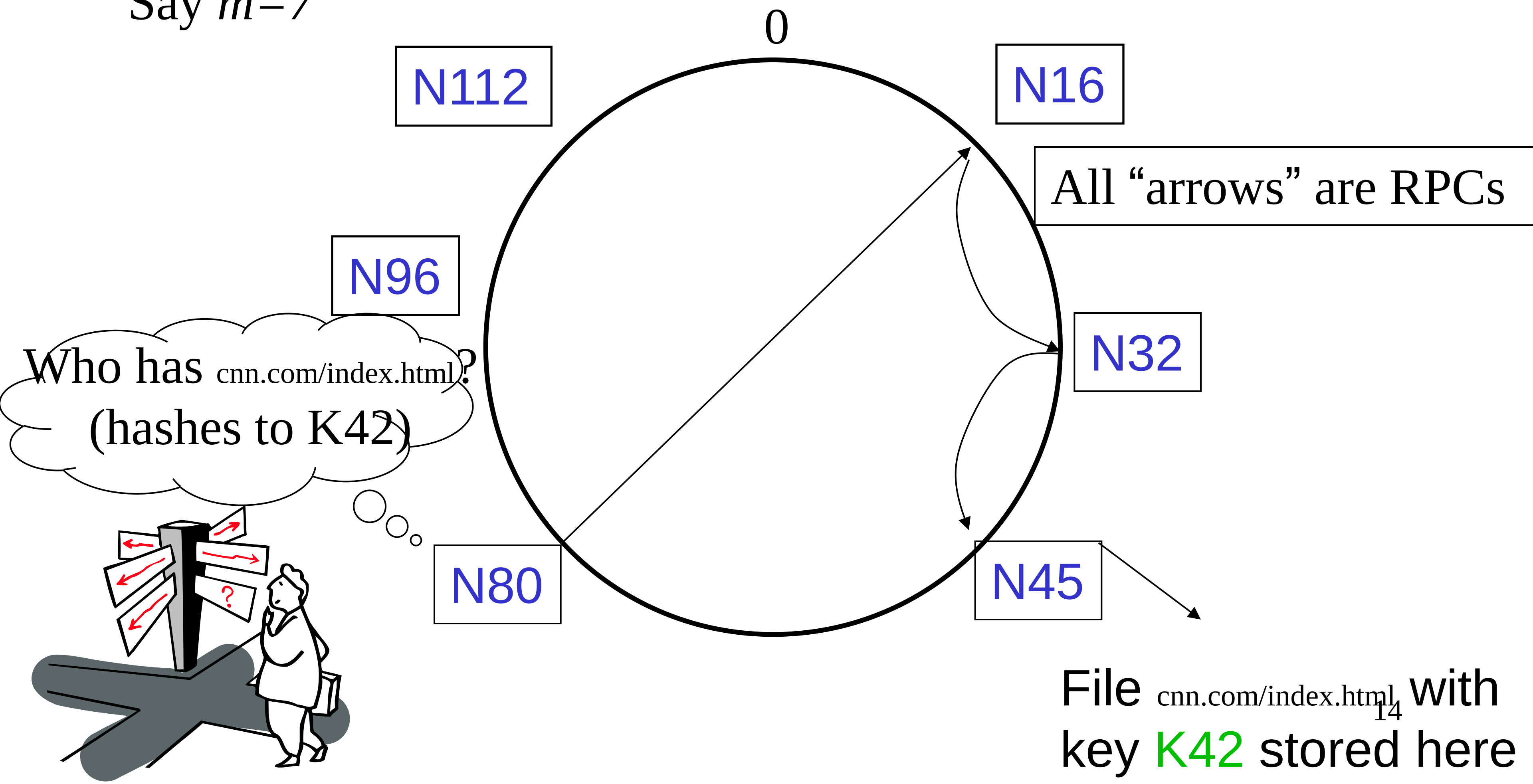


File cnn.com/index.html with key K42 stored here

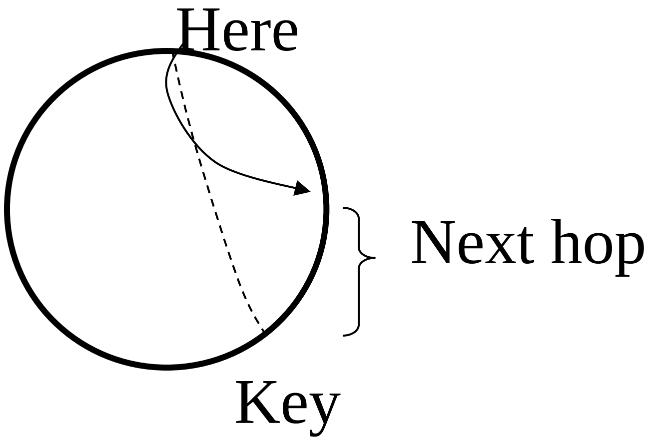
Search

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

Say $m=7$



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* (why?)

Takes at most *m* steps: 2^m is at most a constant multiplicative factor above *N*, lookup is *O(log(N))*

– (intuition): after *log(N)* forwardings, distance to key is at most $2^m / N$ (why?)

Number of node identifiers in a range of $2^m / N$

is *O(log(N))* with high probability (why? SHA-1!)

So using *successors* in that range will be ok

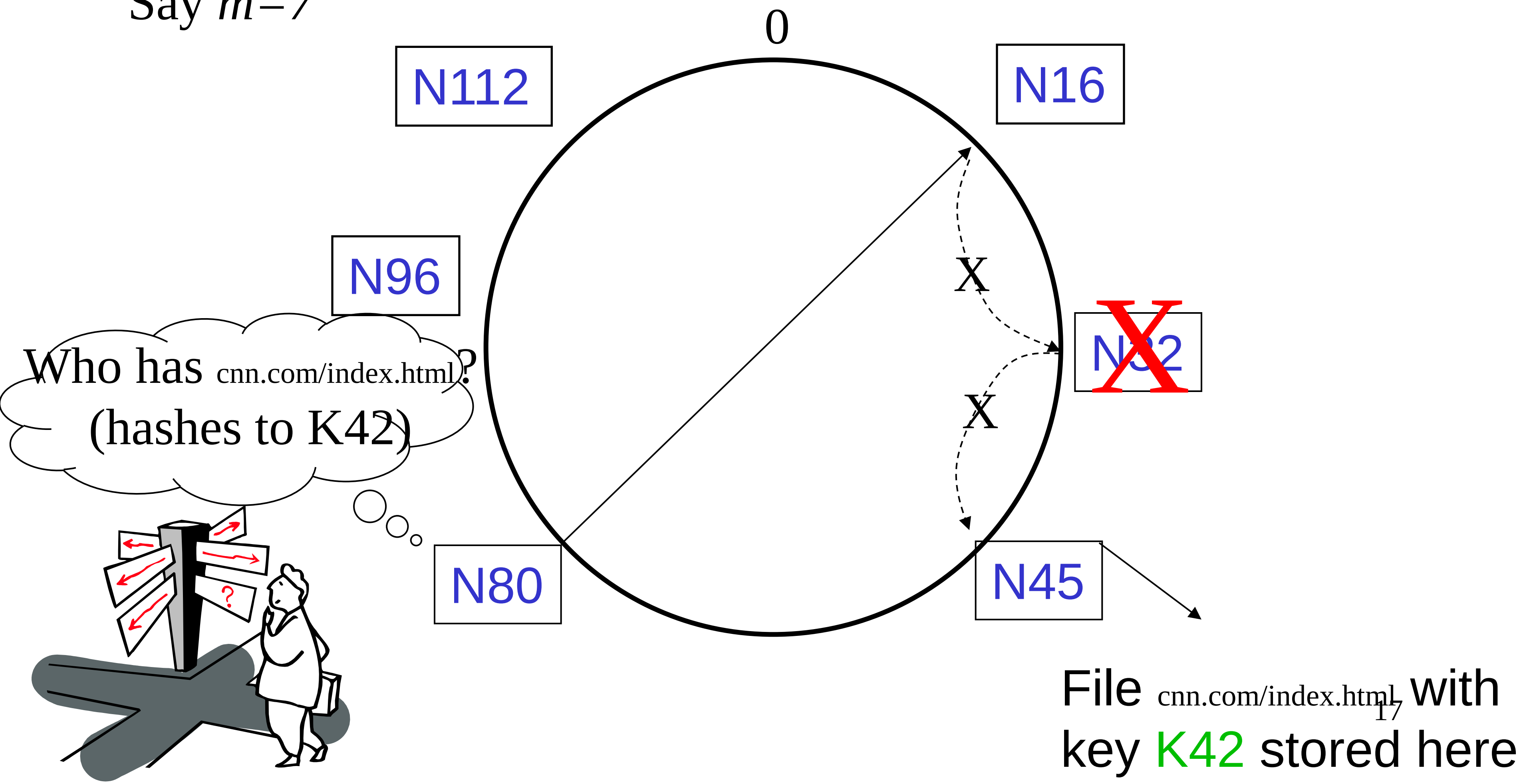
Analysis (contd.)

- *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)

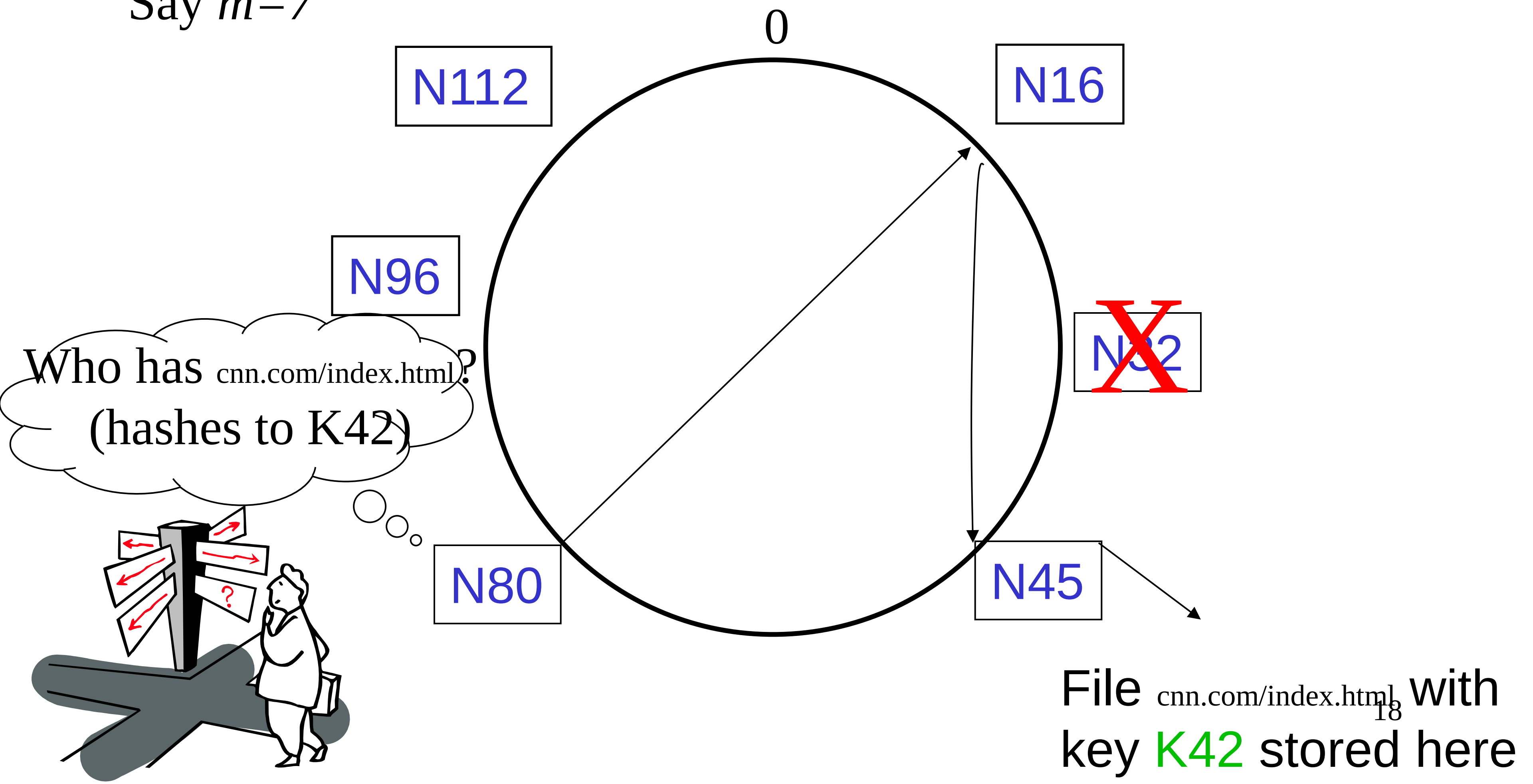
Say *m=7*



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

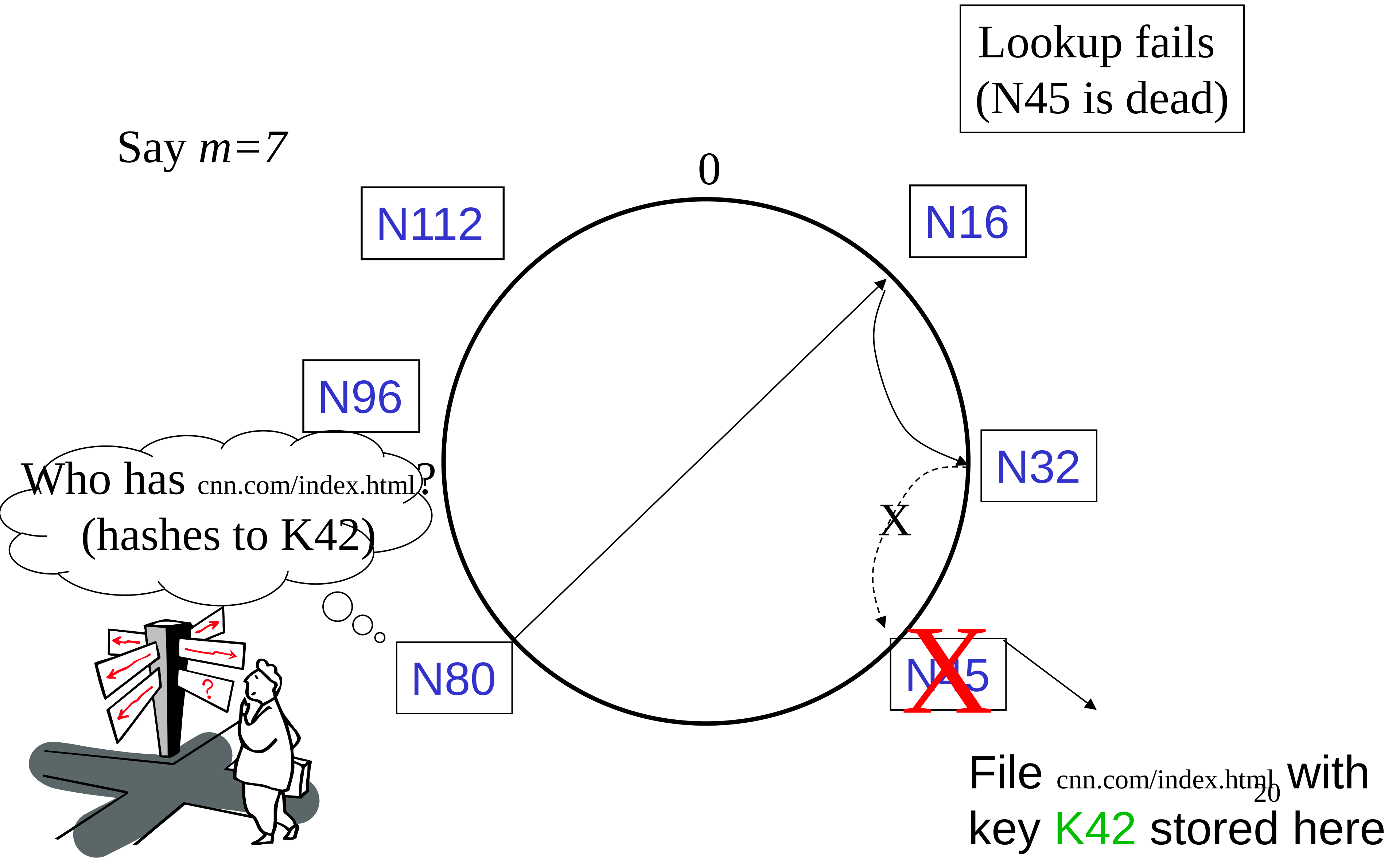
- Choosing *r=2log(N)* suffices to maintain *lookup correctness* w.h.p.
  - Say 50% of nodes fail
  - Pr(at given node, at least one successor alive)=

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

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

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

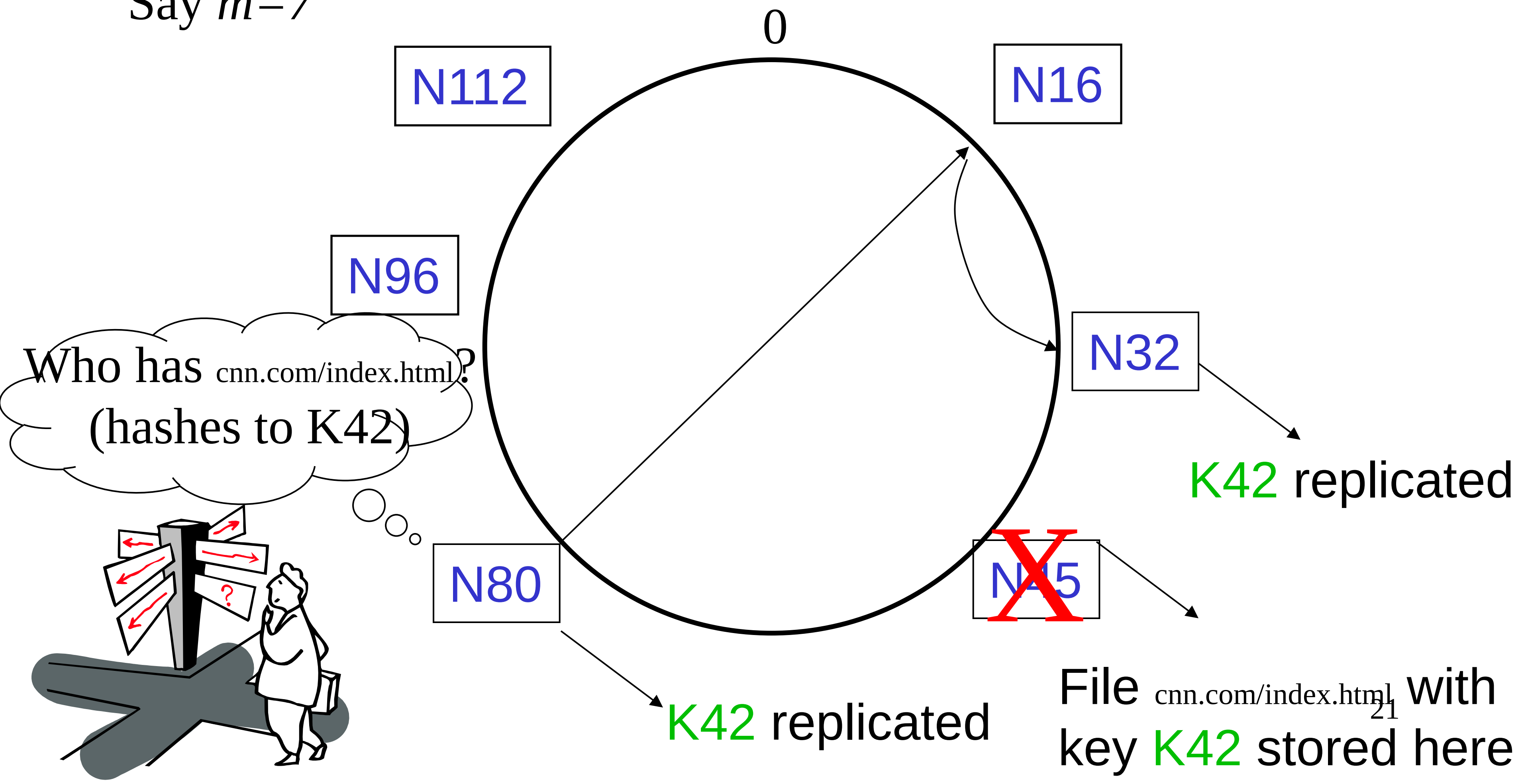
Search under peer failures (2)



Search under peer failures (2)

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

Say $m=7$



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, e.g., CSIL
    - Common feature in all distributed systems, including clouds

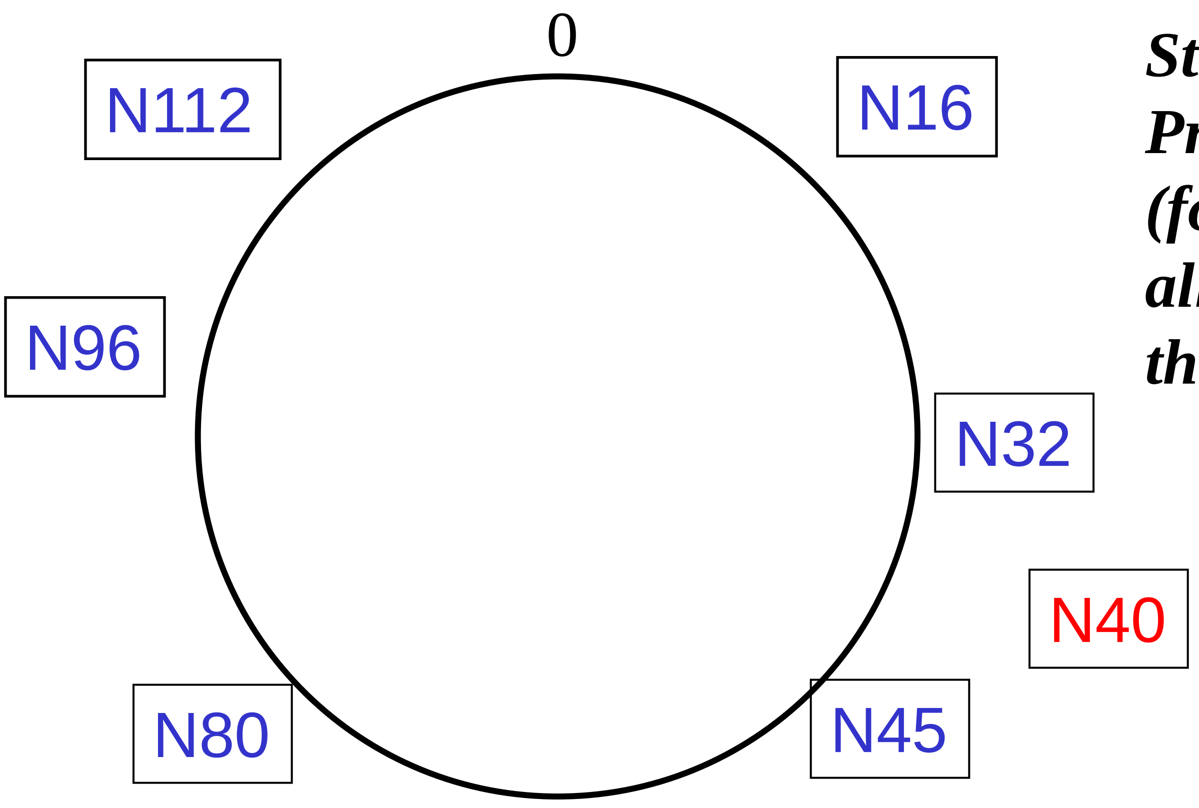
So, all the time, need to:

→ Update *successor*s and *finger*s, and copy keys

New peers joining

Introducer directs N40 to N45 by routing to K40
N32 updates successor to N40
N40 initializes successor to N45, and inits fingers from it
N40 periodically talks to its neighbors to update finger table

Say *m=7*

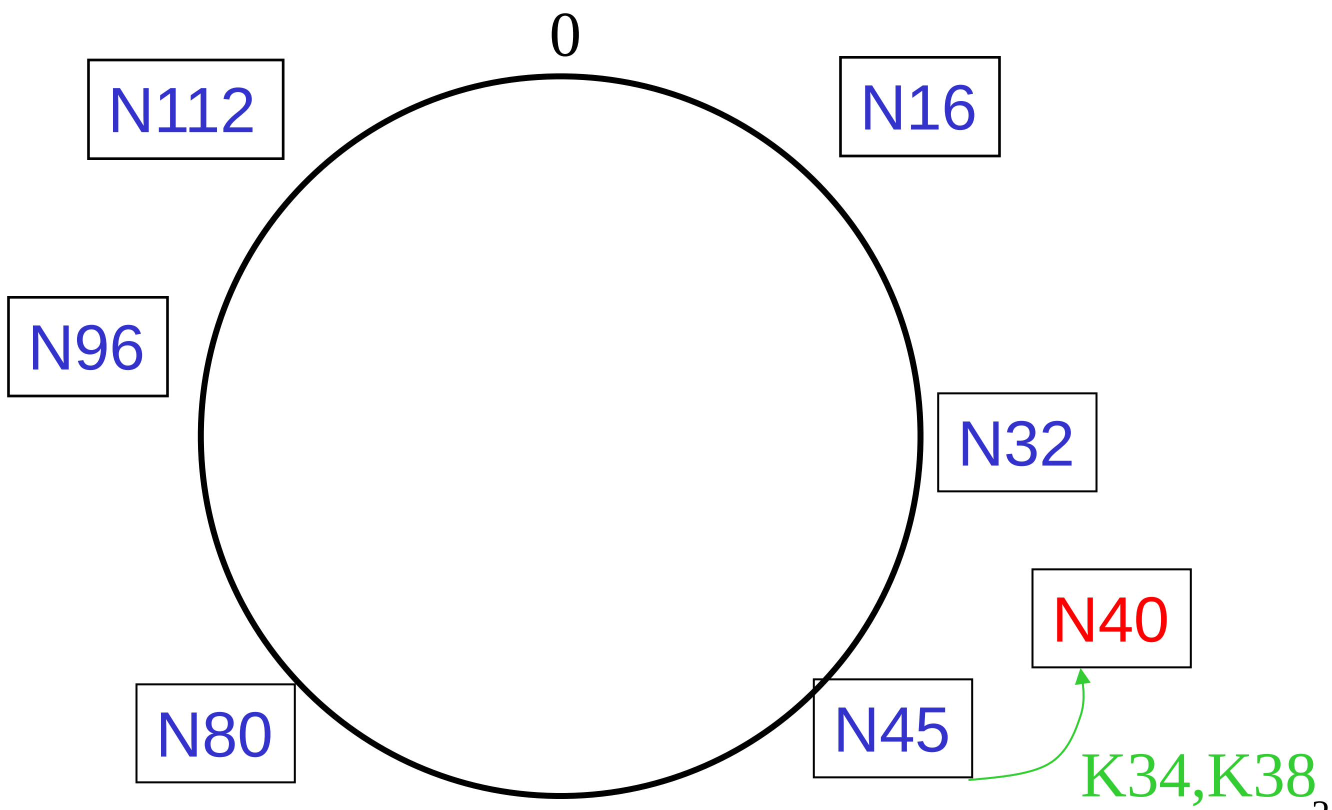


Stabilization Protocol (followed by all nodes all the time)

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)

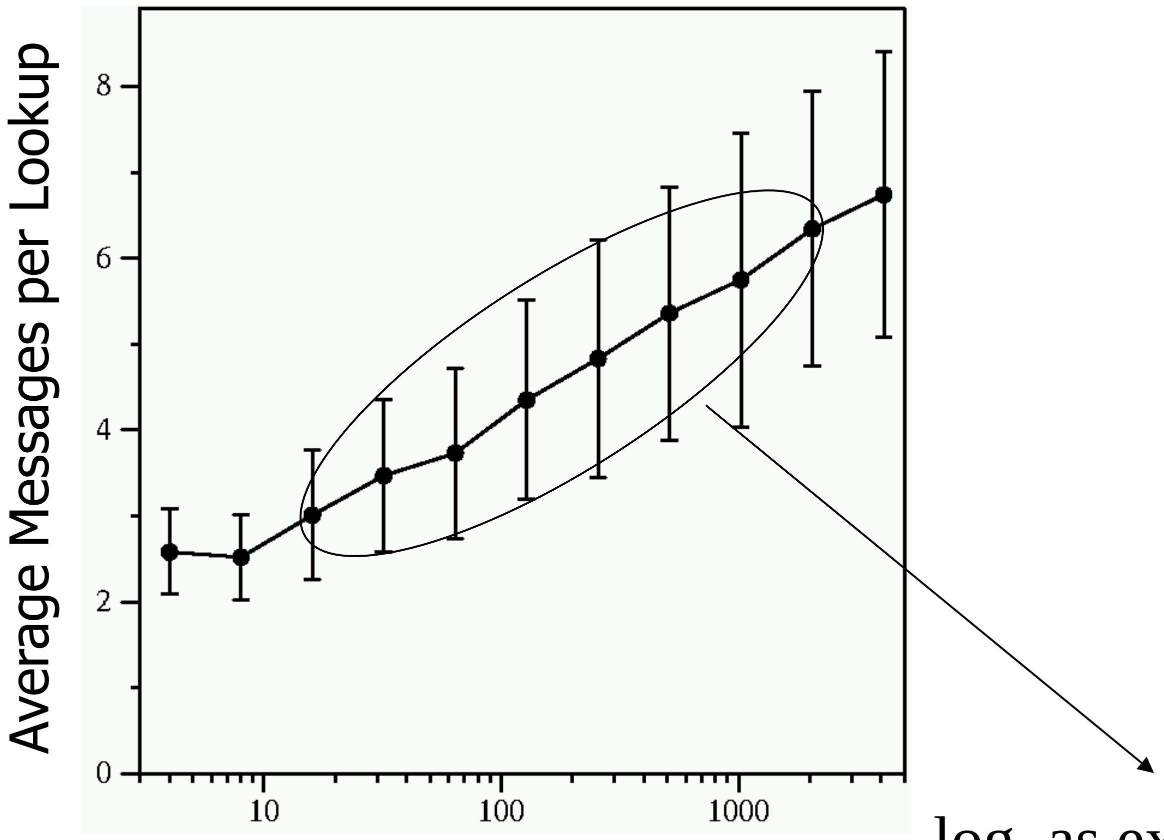
- A new peer affects *O(log(N))* other finger entries in the system, on average [Why?]
- Number of messages per peer join= *O(log(N)*log(N))*

- Similar set of operations for dealing with peers leaving
  - For dealing with failures, need to couple above mechanisms with *failure detectors*

Experimental Results

- Sigcomm 01 paper had results from simulation of a C++ prototype
- SOSP 01 paper had more results from a 12-node Internet testbed deployment
- We'll touch briefly on the first set
- 10000 peer system

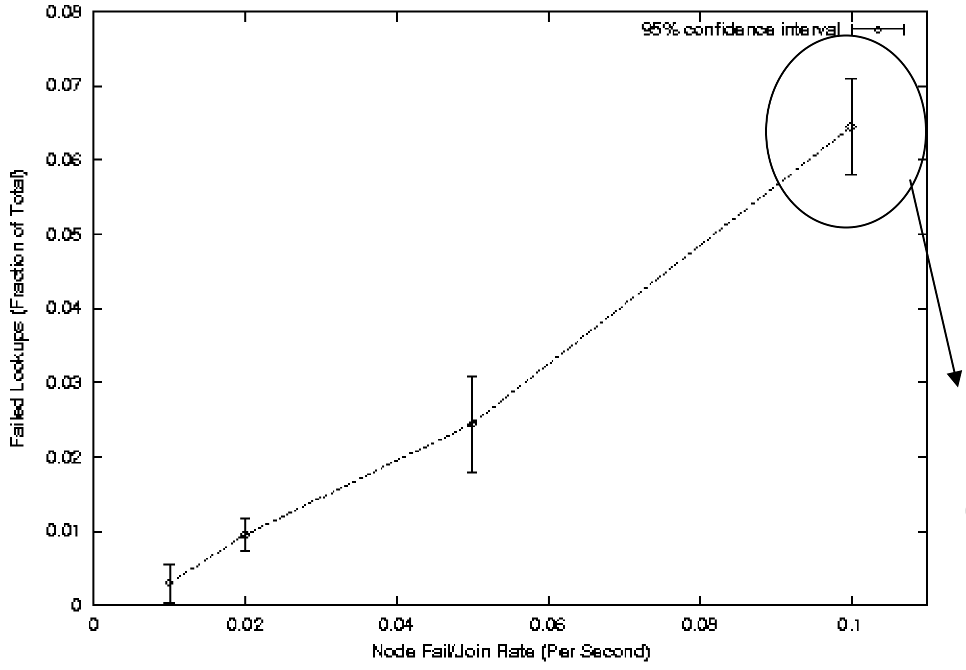
Lookups



log, as expected

Number of Nodes

Fault-tolerance



500 nodes (avg. path len=5)
Stabilization runs every 30 s

1 joins&fails every 10 s
(3 fails/stabilization round)
=> 6% lookups fail

Wrap-up Notes

- Memory: *O(log(N))* successor pointer, *m* finger entries
- Indirection: store a pointer instead of the actual file
- Does not handle partitions (can you suggest a possible solution?)

Summary of Chord

- Chord protocol
  - More structured than Gnutella
  - *O(log(N))* memory and lookup cost
  - Simple lookup algorithm, rest of protocol complicated
  - Stabilization works, but how far can it go?

Wrap-up Notes

Applies to all p2p systems

- How does a peer join the system
  - Send an http request to well-known url for that P2P service - `http://www.myp2pservice.com`
  - Message routed (after DNS lookup) to a well known server which then initializes new peers' neighbor table
  - Server only maintains a partial list of online clients

Announcements

- Next lecture – Mutual Exclusion
  - Reading: Sections 15.2
- MP2
  - By now you should have a working heartbeat mechanism, and by Thursday you should have finished everything
  - Due 10/6 mifnight
  - Demos on Monday 10/7 – watch Piazza for signup sheet
- Midterm Exam is Oct 15th during class hours
  - All material until Lecture 12
  - Location may be same or different (watch Piazza)

Optional Slides

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.
  - [TechReport on Chord webpage] defines *weak* and *strong* notions of stability
  - Each stabilization round at a peer involves a constant number of messages
  - Strong stability takes $O(N^2)$ stabilization rounds (!)