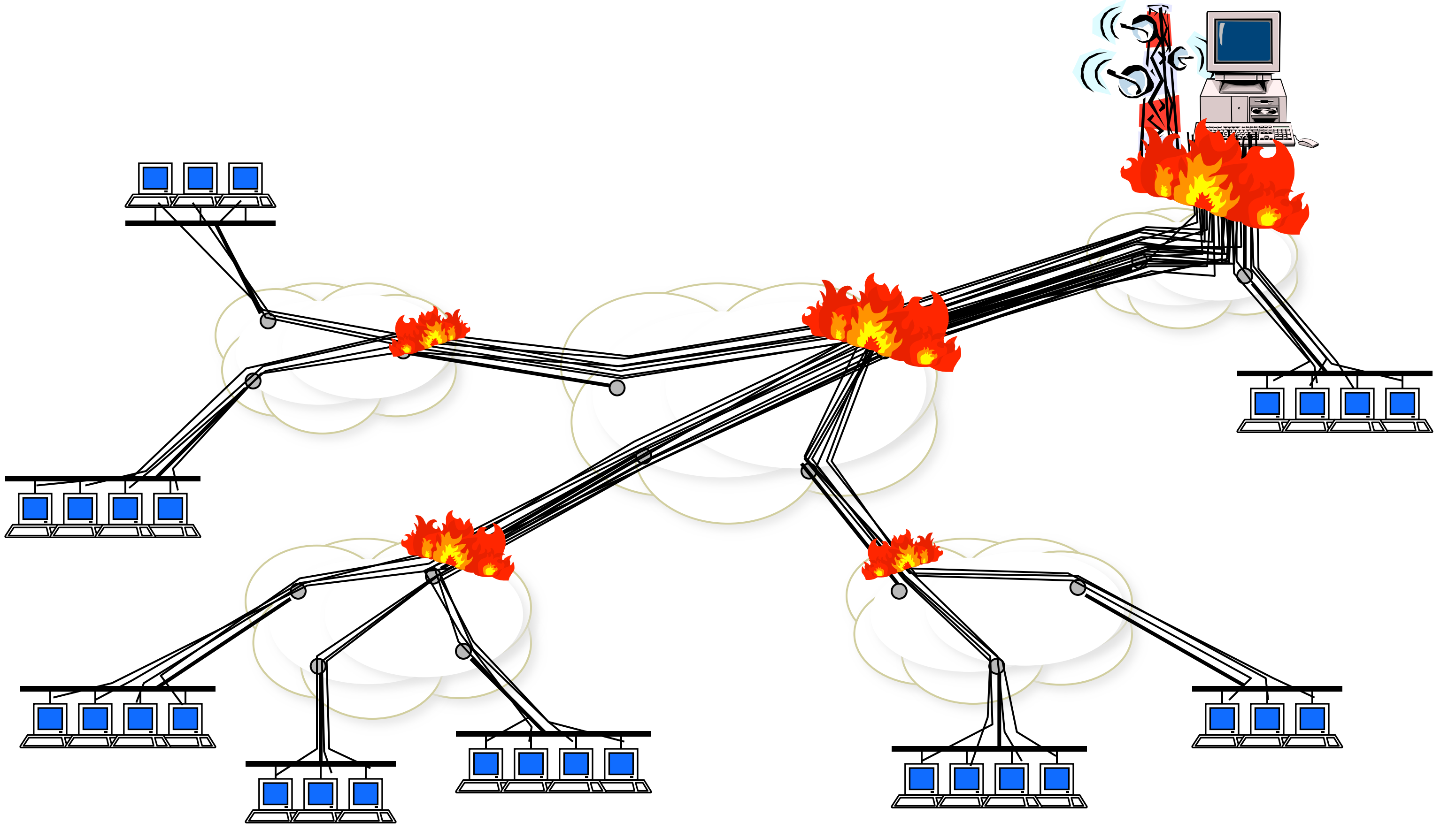
416 Distributed Systems

Mar 3, Peer-to-Peer

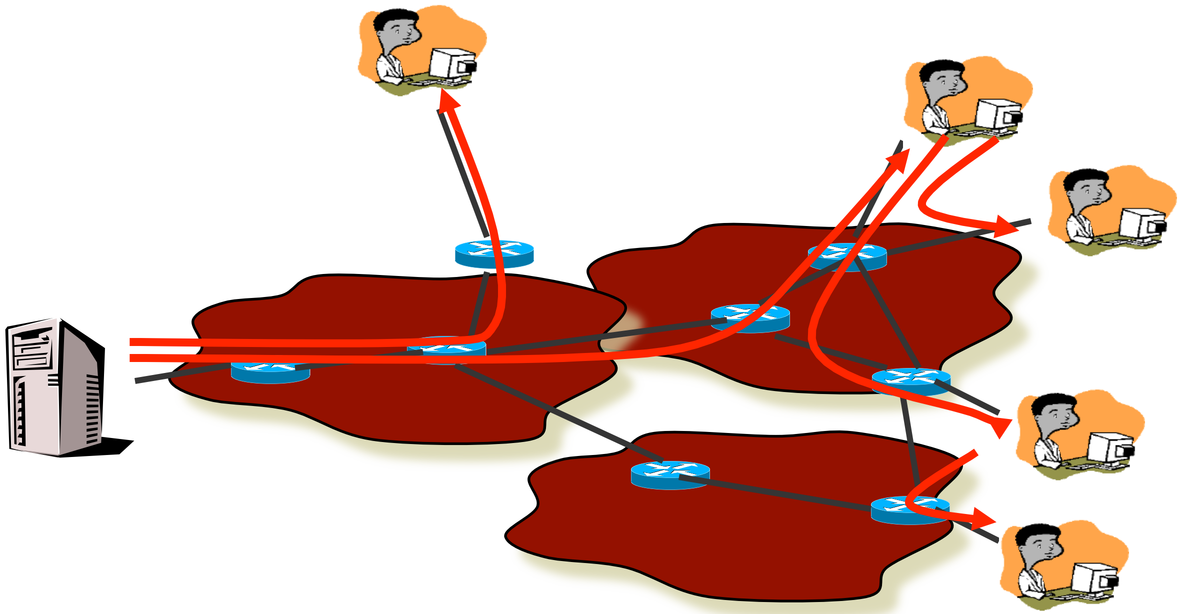
Part 2

Scaling Problem

- Millions of clients ⇒ server and network meltdown



P2P System



- Leverage the resources of client machines (peers)
  - Traditional: Computation, storage, bandwidth
  - Non-traditional: Geographical diversity, mobility, sensors!

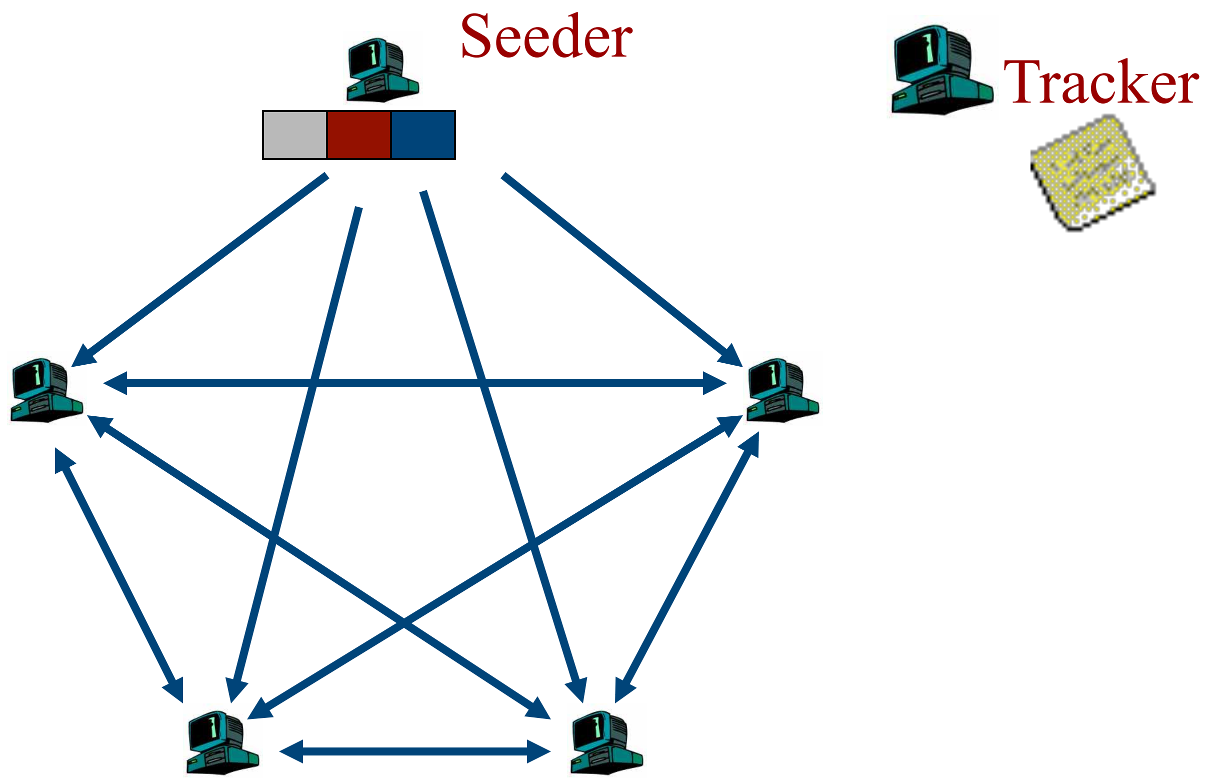
Outline

- BitTorrent
- Routed Lookups – Chord

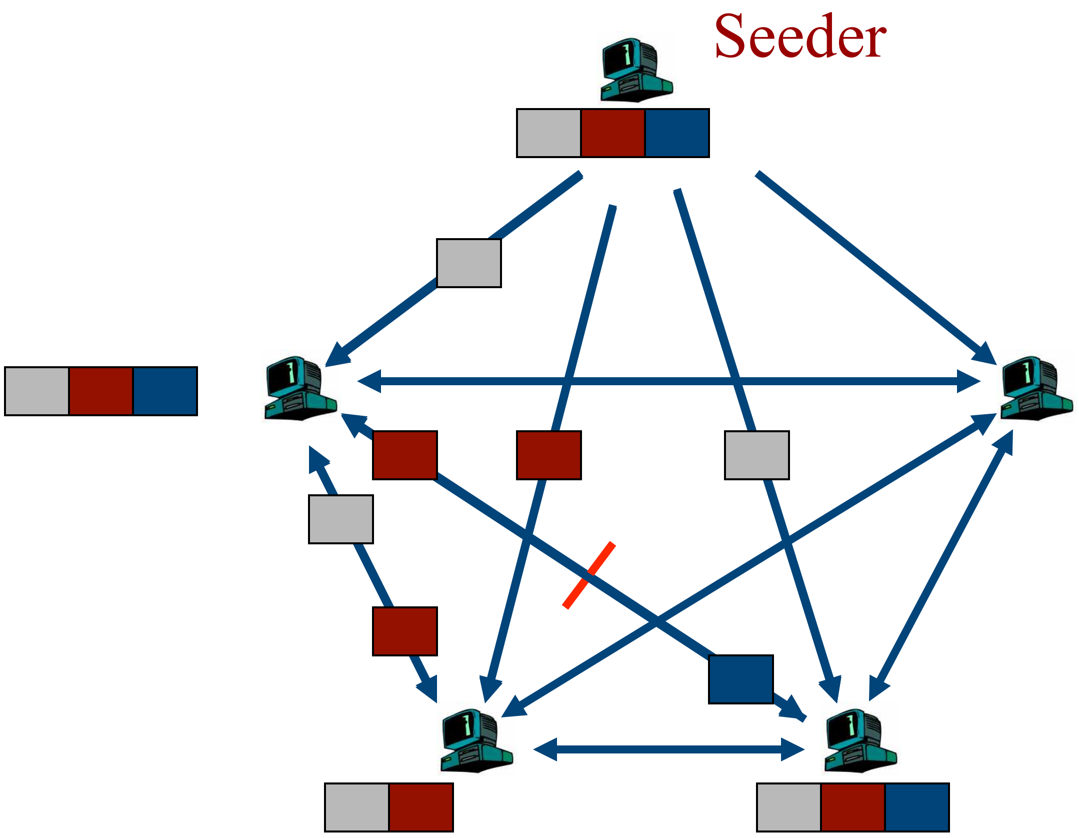
BitTorrent: Overview

- File swarming:
  - **Join**: contact centralized “tracker” server, get a list of peers.
  - **Publish**: Run a tracker server.
  - **Search**: Out-of-band. E.g., use Google to find a tracker for the file you want.
  - **Fetch**: Download chunks of the file from your peers. Upload chunks you have to them.
- Big differences from Napster:
  - Chunk based downloading
  - “few large files” focus
  - Anti-freeloading mechanisms

BitTorrent: Publish/Join



BitTorrent: Fetch



BitTorrent: Sharing Strategy

- Employ “Tit-for-tat” sharing strategy
  - A is downloading from some other people
    - A will let the fastest N of those download from it
  - Be optimistic: occasionally let freeloaders download
    - *Optimistic unchoke*
    - Otherwise no one would ever start!
    - Also allows you to discover better peers to download from when they reciprocate
- Goal: Pareto Efficiency
  - Game Theory: “No change can make anyone better off without making others worse off”
  - Does it work? How would you cheat?
  - (not perfectly, but perhaps good enough?)

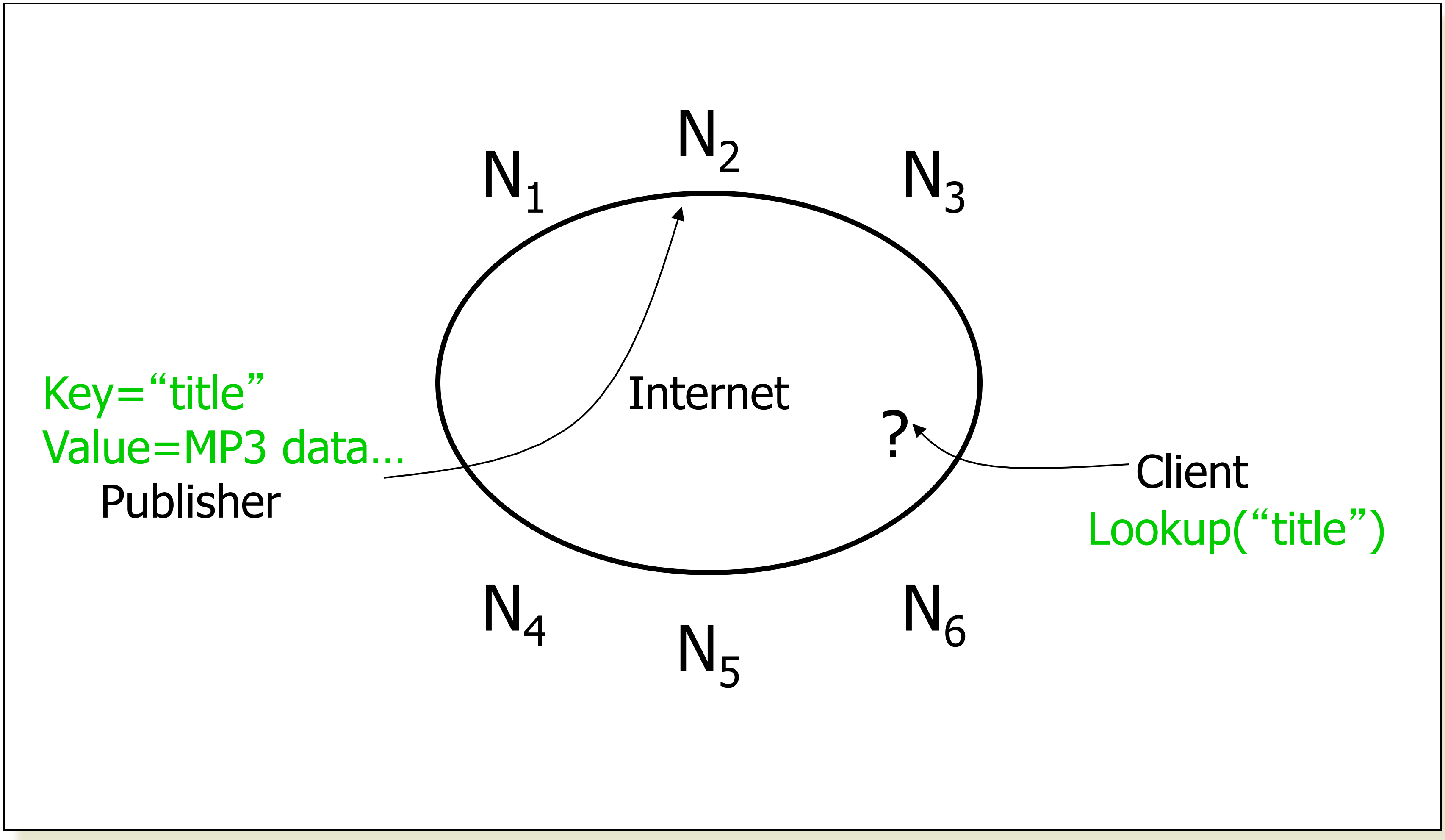
BitTorrent: Summary

- Pros:
  - Works reasonably well in practice
  - Gives peers incentive to share resources; avoids freeloaders
- Cons:
  - Pareto Efficiency relative weak condition
  - Central tracker server needed to bootstrap swarm
    - Alternate tracker designs exist (e.g., DHT-based trackers)

Outline

- BitTorrent
- Routed Lookups – Chord

The Lookup Problem



DHT: Overview (1)

- Goal: make sure that an item (file) identified is always found in a reasonable # of steps
- Abstraction: a distributed hash-table (DHT) data structure
  - insert(id, item);
  - item = query(id);
  - Note: item can be anything: a data object, document, file, pointer to a file…
- Implementation: nodes in system form a distributed data structure
  - Can be Ring, Tree, Hypercube, Skip List, Butterfly Network, ...

DHT: Overview (2)

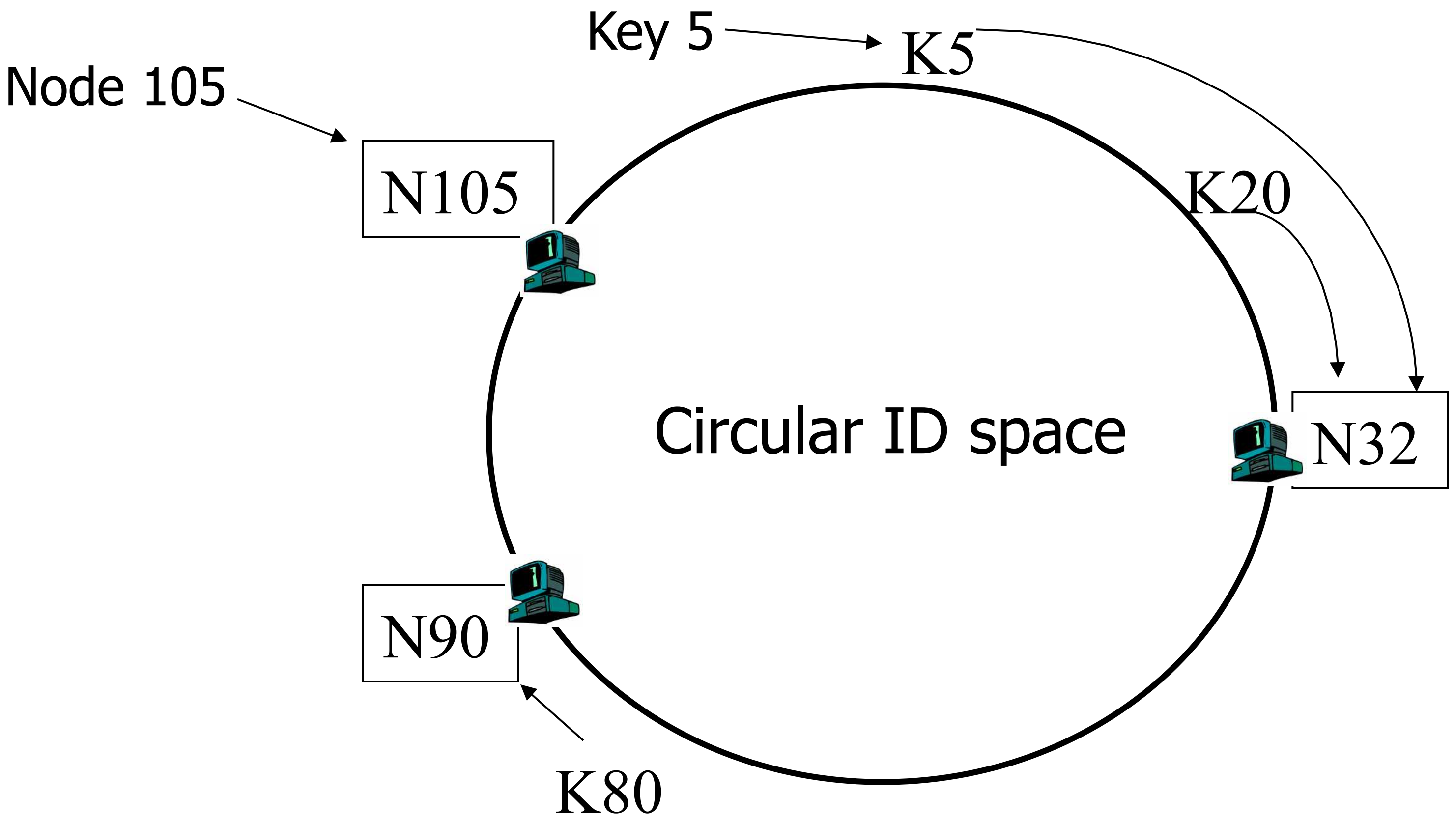
- *Structured* Overlay Routing:
  - **Join**: On startup, contact a "bootstrap" node and integrate yourself into the distributed data structure; get a *node id*
  - **Publish**: Route publication for *file id* toward a close *node id* along the data structure
  - **Search**: Route a query for file id toward a close node id. Data structure guarantees that query will meet the publication.
  - **Fetch**: Two options:
    - Publication contains actual file => fetch from where query stops
    - Publication says "I have file X" => query tells you 128.2.1.3 has X, use IP routing to get X from 128.2.1.3

DHT: Example - Chord

- Associate to each node and file a unique *id* in an *uni*-dimensional space (a Ring)
  - E.g., pick from the range $[0...2^m]$
  - Usually the hash of the file or IP address
- Routing properties:
  - Routing table size is O(log *N*) , where *N* is the total number of nodes
  - Guarantees that a file is found in O(log *N*) hops

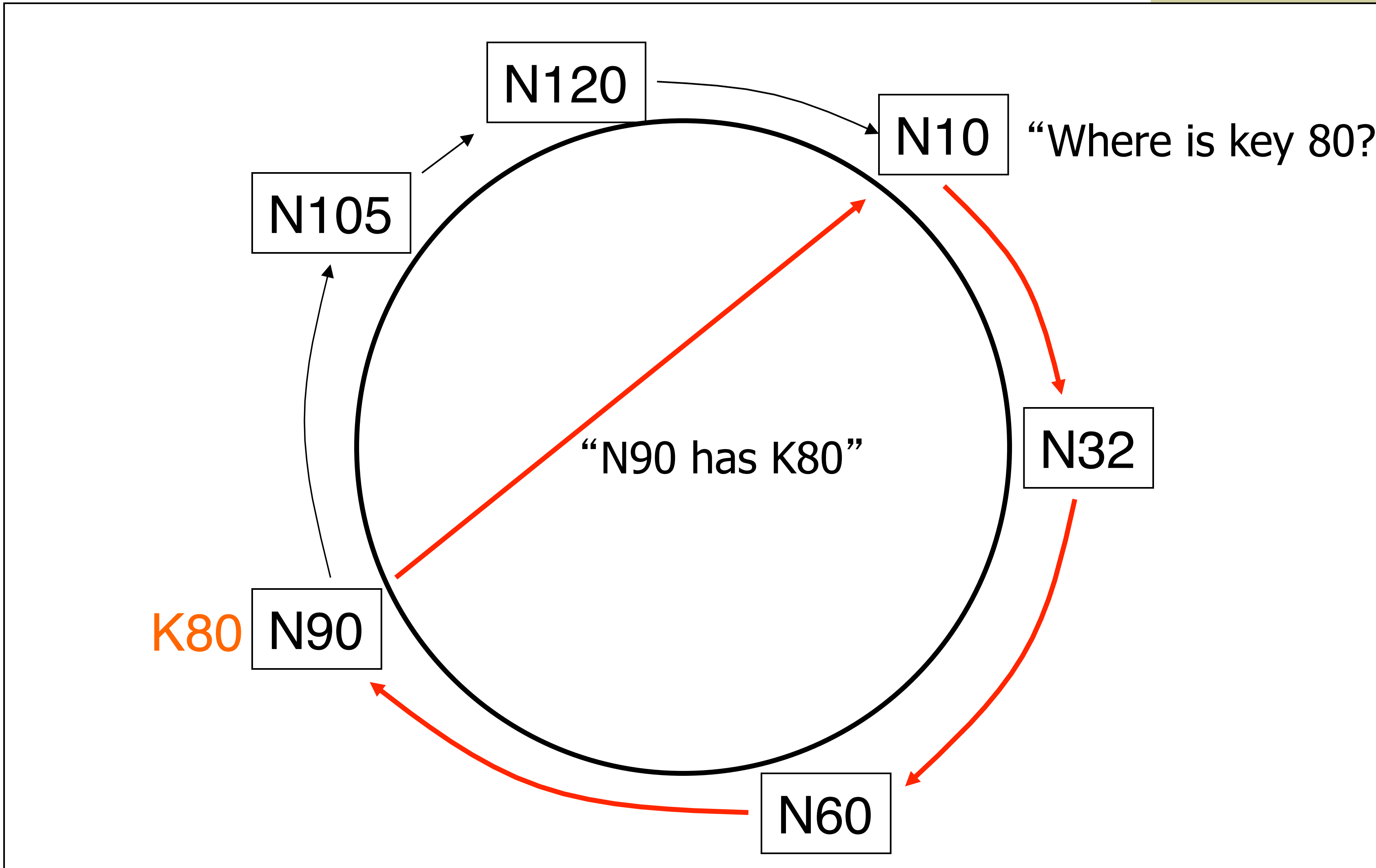
from MIT in 2001

DHT: Consistent Hashing

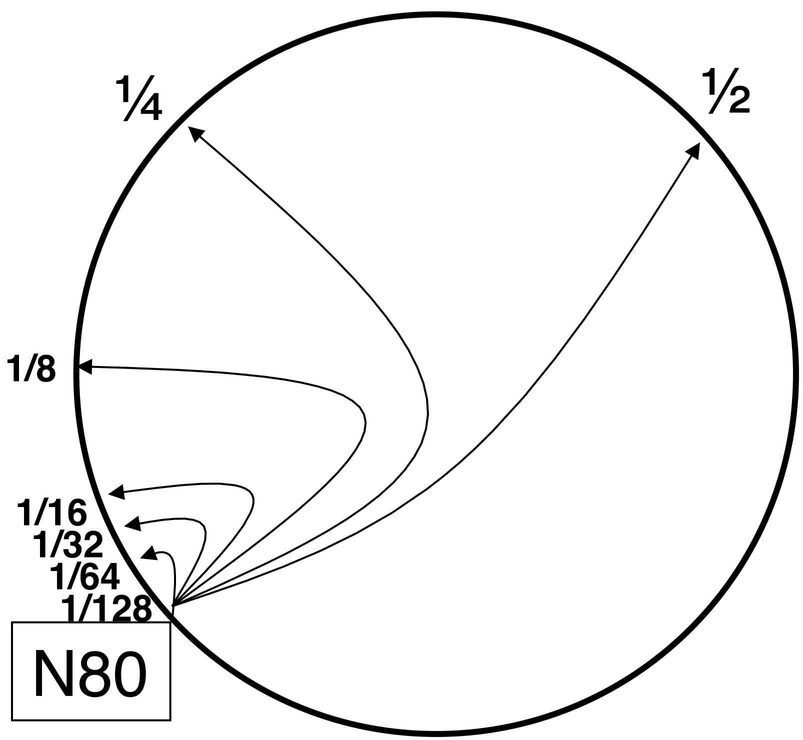


A key is stored at its successor: node with next higher ID

Routing: Chord Basic Lookup



Routing: Finger table - Faster Lookups

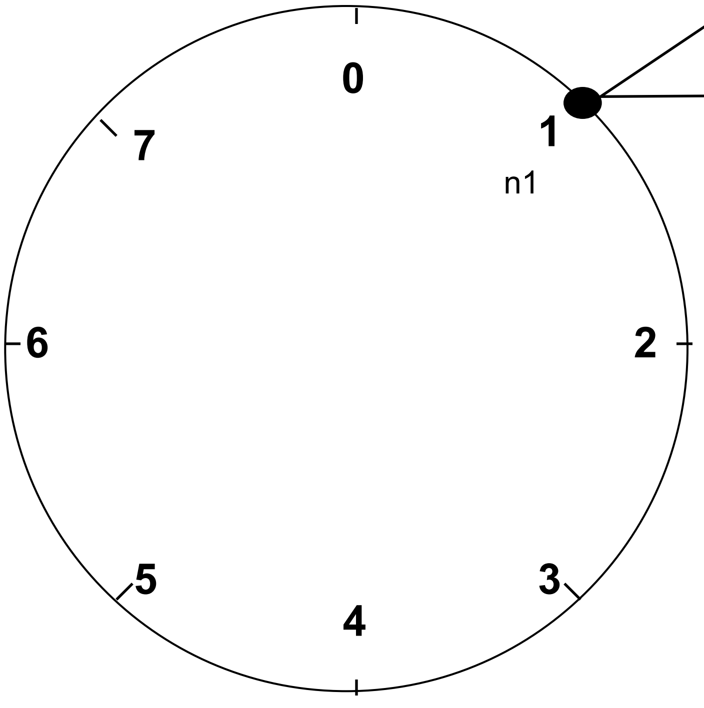


Routing: Chord Summary

- Assume identifier space is $0 \ldots 2^m$
- Each node maintains
  - Finger table
    - Entry *i* in the finger table of *n* is the first node that succeeds or equals $n + 2^i$
  - Predecessor node
- An item identified by *id* is stored on the successor node of *id*

Routing: Chord Example

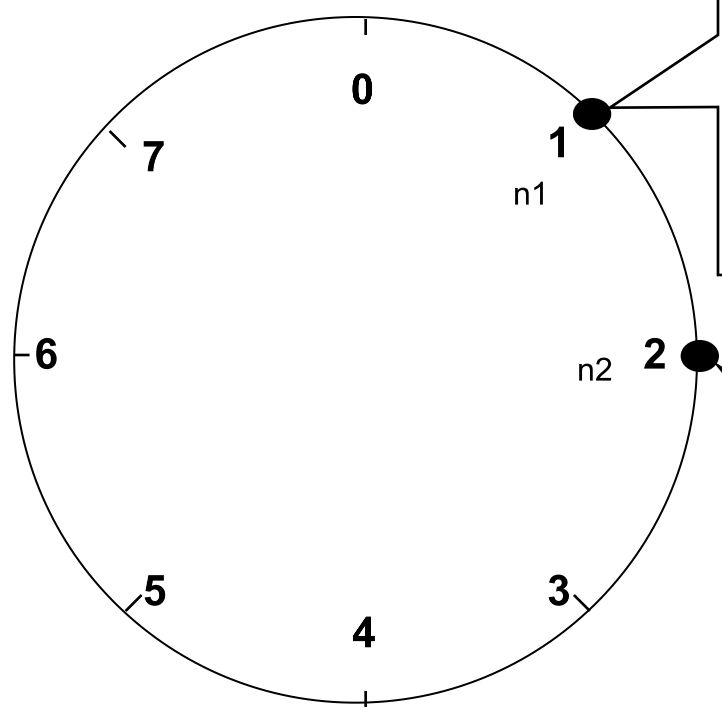
- Assume an identifier space 0..7
- Node n1:(1) joins→all entries in its finger table are initialized to itself



i	$id+2^i$	succ
0	2	1
1	3	1
2	5	1

Routing: Chord Example

- Node n2(2) joins



Succ. Table (n1)

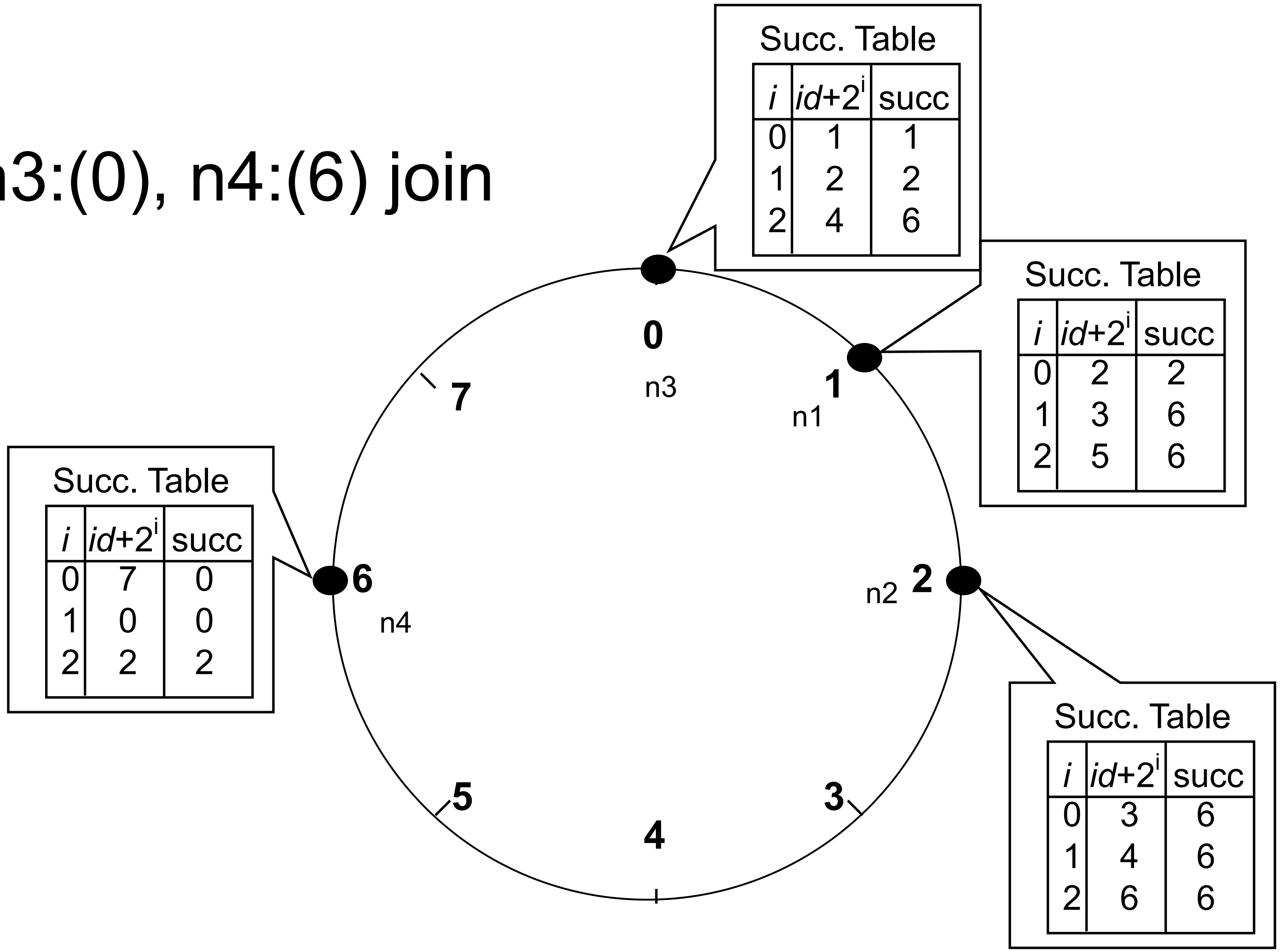
i	$id+2^i$	succ
0	2	2
1	3	1
2	5	1

Succ. Table (n2)

i	$id+2^i$	succ
0	3	1
1	4	1
2	6	1

Routing: Chord Example

- Nodes n3:(0), n4:(6) join



Succ. Table (n3, node 0)

i	*id*+2^i	succ
0	1	1
1	2	2
2	4	6

Succ. Table (n1, node 1)

i	*id*+2^i	succ
0	2	2
1	3	6
2	5	6

Succ. Table (n2, node 2)

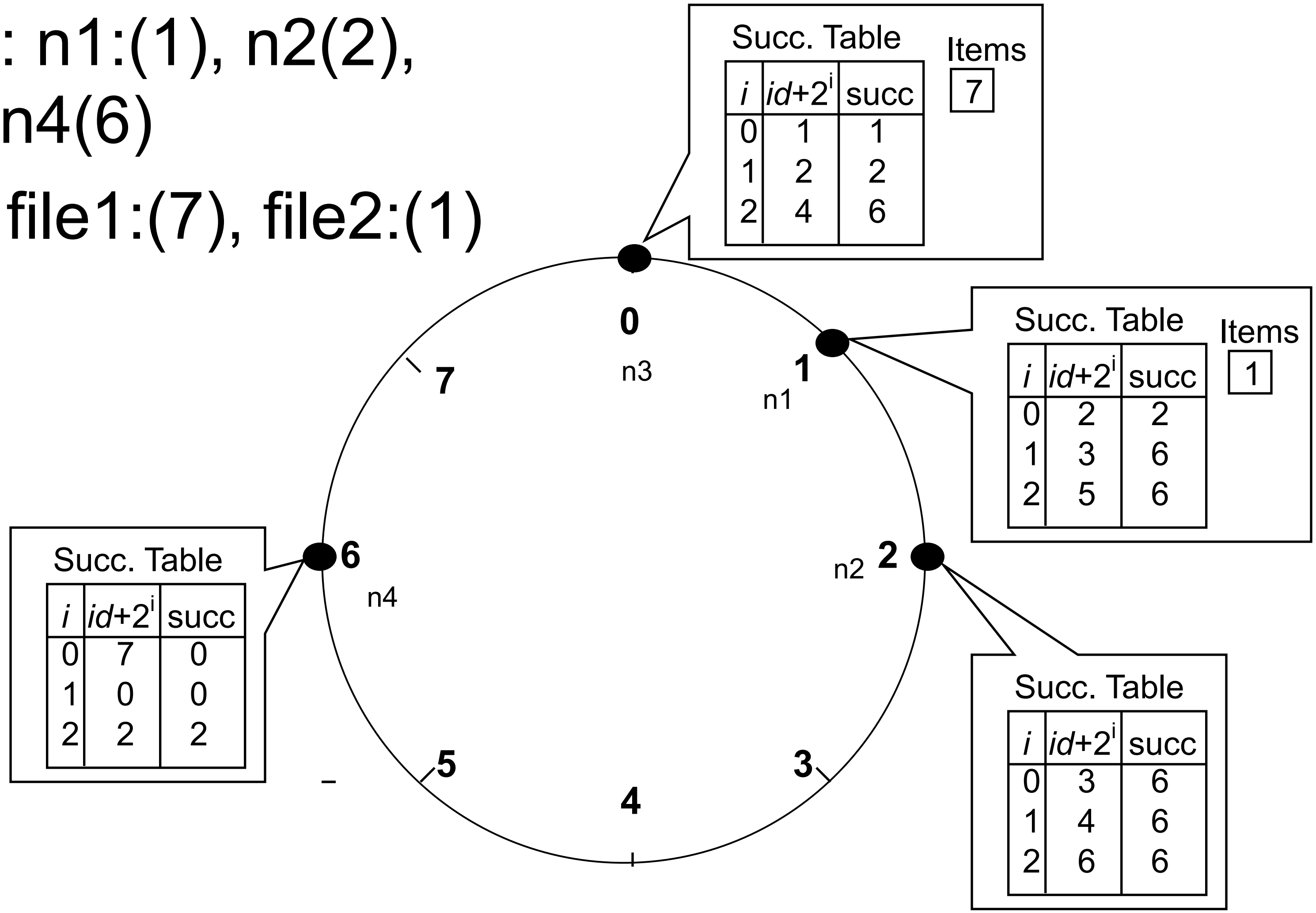
i	*id*+2^i	succ
0	3	6
1	4	6
2	6	6

Succ. Table (n4, node 6)

i	*id*+2^i	succ
0	7	0
1	0	0
2	2	2

Routing: Chord Examples

- Nodes: n1:(1), n2(2), n3(0), n4(6)
- Items: file1:(7), file2:(1)



Succ. Table (n3, node 0) — Items: 7

i	*id*+2^i	succ
0	1	1
1	2	2
2	4	6

Succ. Table (n1, node 1) — Items: 1

i	*id*+2^i	succ
0	2	2
1	3	6
2	5	6

Succ. Table (n2, node 2)

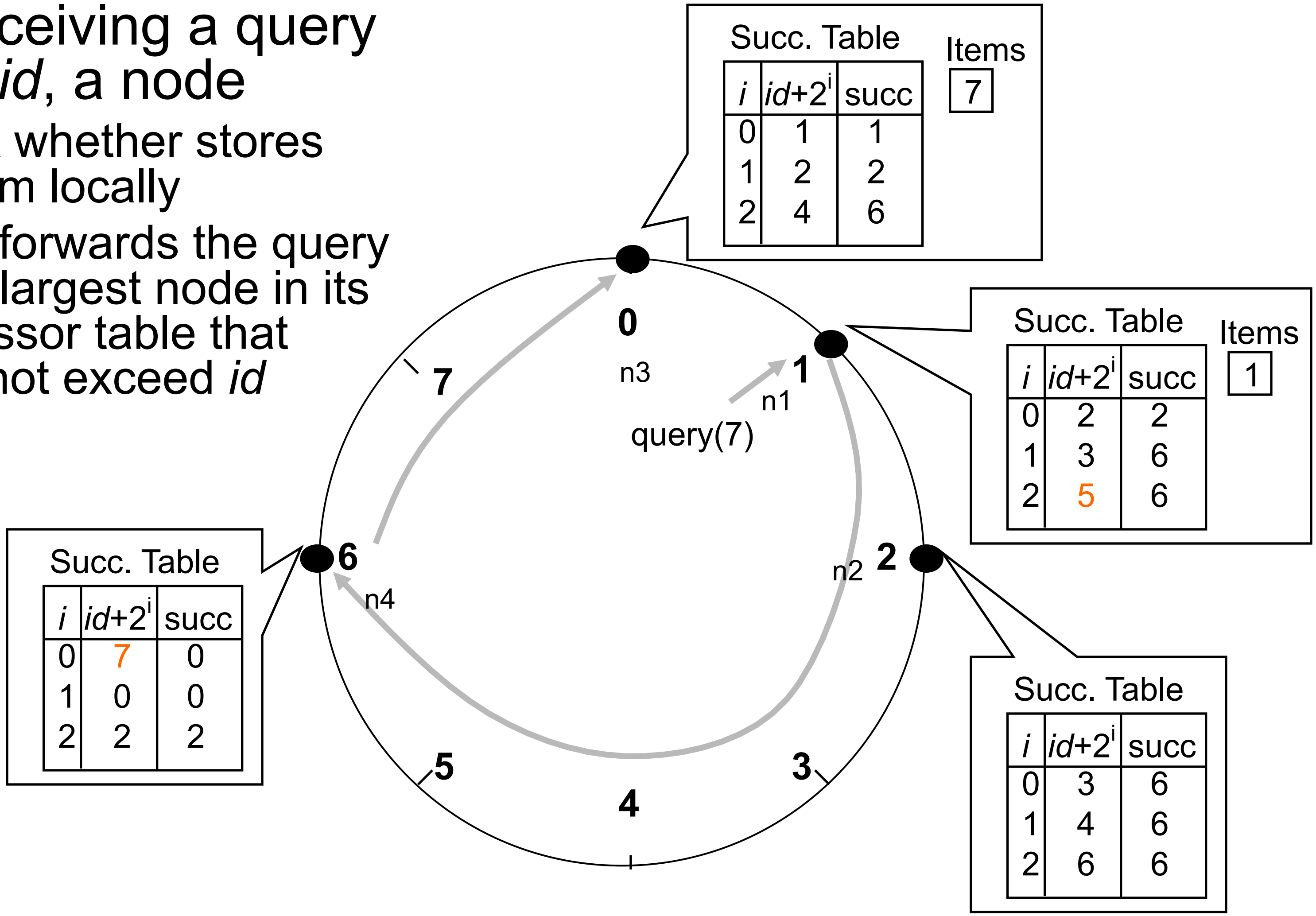
i	*id*+2^i	succ
0	3	6
1	4	6
2	6	6

Succ. Table (n4, node 6)

i	*id*+2^i	succ
0	7	0
1	0	0
2	2	2

Routing: Query

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



DHT: Chord Summary

- Routing table size?
  - Log *N* fingers
- Routing time?
  - Each hop expects to 1/2 the distance to the desired id => expect O(log *N*) hops.

DHT: Discussion

- Pros:
  - Guaranteed Lookup
  - O(log *N*) per node state and search scope
- Cons:
  - No one uses them? (only one file sharing app)
  - Supporting non-exact match search is hard

What can DHTs do for us?

- Distributed object lookup
  - Based on object ID
- De-centralized file systems
  - CFS, PAST, Ivy
- Application Layer Multicast
  - Scribe, Bayeux, Splitstream
- Databases
  - PIER

When are p2p / DHTs useful?

- Caching and “soft-state” data
  - Works well! BitTorrent, KaZaA, etc., all use peers as caches for hot data
- Finding read-only data
  - Limited flooding finds hay
  - DHTs find needles
- BUT

A Peer-to-peer Google?

- Complex intersection queries (“the” + “who”)
  - Billions of hits for each term alone
- Sophisticated ranking
  - Must compare many results before returning a subset to user
- Very, very hard for a DHT / p2p system
  - Need high inter-node bandwidth
  - (This is exactly what Google does - massive clusters)

Writable, persistent p2p

- Do you trust your data to 100,000 monkeys?
- Node availability hurts
  - Ex: Store 5 copies of data on different nodes
  - When someone goes away, you must replicate the data they held
  - Hard drives are *huge*, but edge network upload bandwidth is tiny
  - May take days to upload contents of a hard drive. P2P replication/fault-tolerance expensive.

P2P: Summary

- Many different styles; remember pros and cons of each
  - centralized, flooding, swarming, and structured routing
- Lessons learned:
  - Single points of failure are very bad
  - Flooding messages to everyone is bad
  - Underlying network topology is important
  - Not all nodes are equal
  - Need incentives to discourage freeloading
  - Privacy and security are important
  - Structure can provide theoretical bounds and guarantees