Distributed Hash Tables

DHTs

- Like it sounds – a distributed hash table
- Put(Key, Value)
- Get(Key) -> Value

Interface vs. Implementation

- Put/Get is an abstract interface
  - Very convenient to program to
  - Doesn't require a “DHT” in today's sense of the world.
  - e.g., Amazon's S^3 storage service
    - /bucket-name/object-id -> data
- We'll mostly focus on the back-end log(n) lookup systems like Chord
  - But researchers have proposed alternate architectures that may work better, depending on assumptions!

Last time: Unstructured Lookup

- Pure flooding (Gnutella), TTL-limited
  - Send message to *all* nodes
- Supernodes (Kazaa)
  - Flood to supernodes only
- Adaptive “super”-nodes and other tricks (GIA)
- None of these scales well for searching for needles

Alternate Lookups

- Keep in mind contrasts to...
- Flooding (Unstructured) from last time
- Hierarchical lookups
  - DNS
  - Properties? Root is critical. Today's DNS root is widely replicated, run in serious secure datacenters, etc. Load is asymmetric.
    - Not always bad – DNS works pretty well
    - But not fully decentralized, if that's your goal

P2P Goal (general)

- Harness storage & computation across (hundreds, thousands, millions) of nodes across Internet
- In particular:
  - Can we use them to create a gigantic, hugely scalable DHT?

P2P Requirements

- Scale to those sizes...
- Be robust to faults and malice
- Specific challenges:
  - Node arrival and departure – system stability
  - Freeloading participants
  - Malicious participants
  - Understanding bounds of what systems can and cannot be built on top of p2p frameworks

DHTs

- Two options:
  - lookup(key) -> node ID
  - lookup(key) -> data
- When you know the nodeID, you can ask it directly for the data, but specifying interface as -> data provides more opportunities for caching and computation at intermediaries
- Different systems do either. We'll focus on the problem of *locating the node responsible for the data*. The solutions are basically the same.

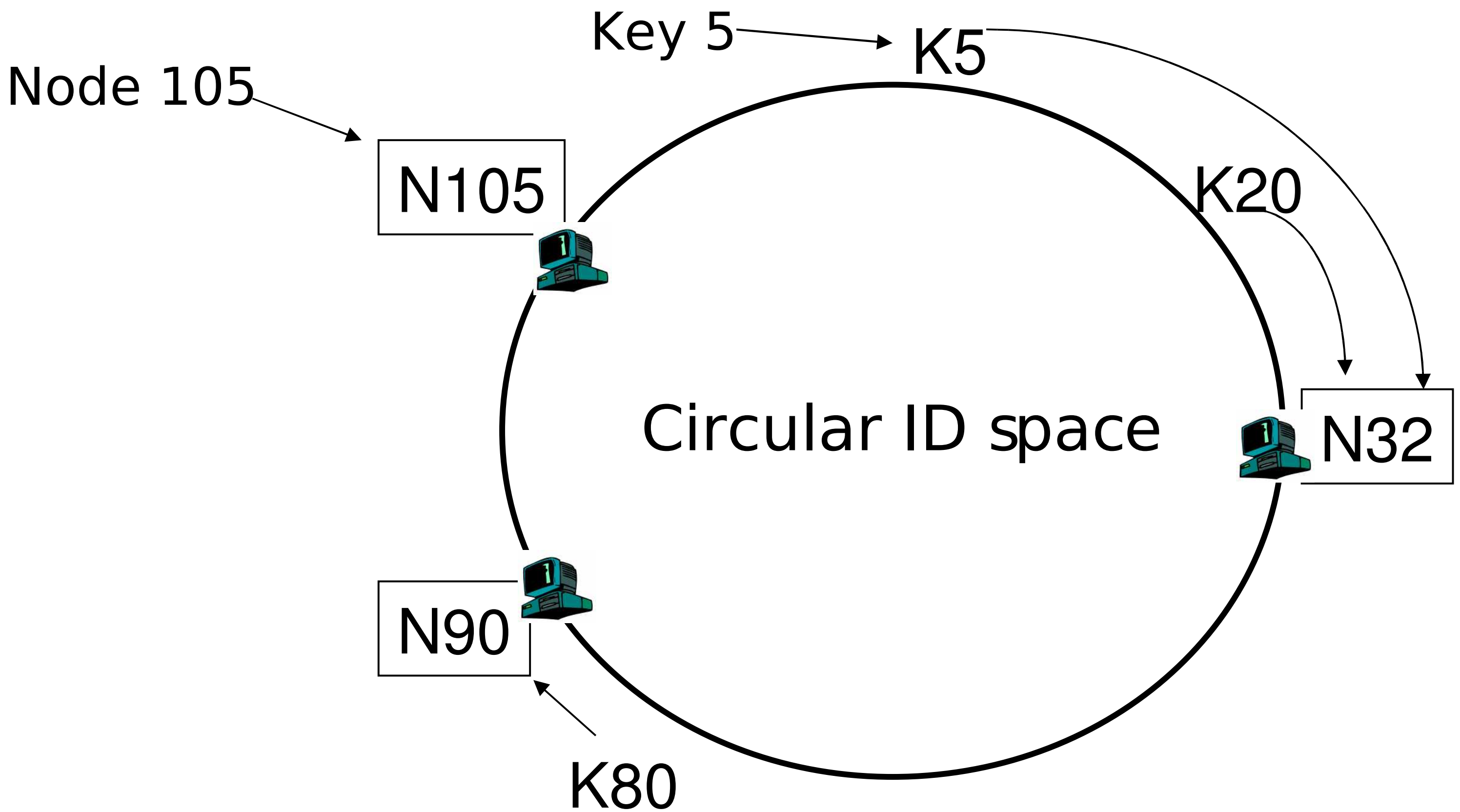
Algorithmic Requirements

- Every node can find the answer
- Keys are load-balanced among nodes
  - Note: We're not talking about *popularity* of keys, which may be wildly different. Addressing this is a further challenge...
- Routing tables must adapt to node failures and arrivals
- How many hops must lookups take?
  - Trade-off possible between state/maint. traffic and num lookups...

Consistent Hashing

- How can we map a key to a node?
- Consider ordinary hashing
  - func(key) % N -> node ID
  - What happens if you add/remove a node?
- Consistent hashing:
  - Map node IDs to a (large) circular space
  - Map keys to same circular space
  - Key “belongs” to nearest node

DHT: Consistent Hashing

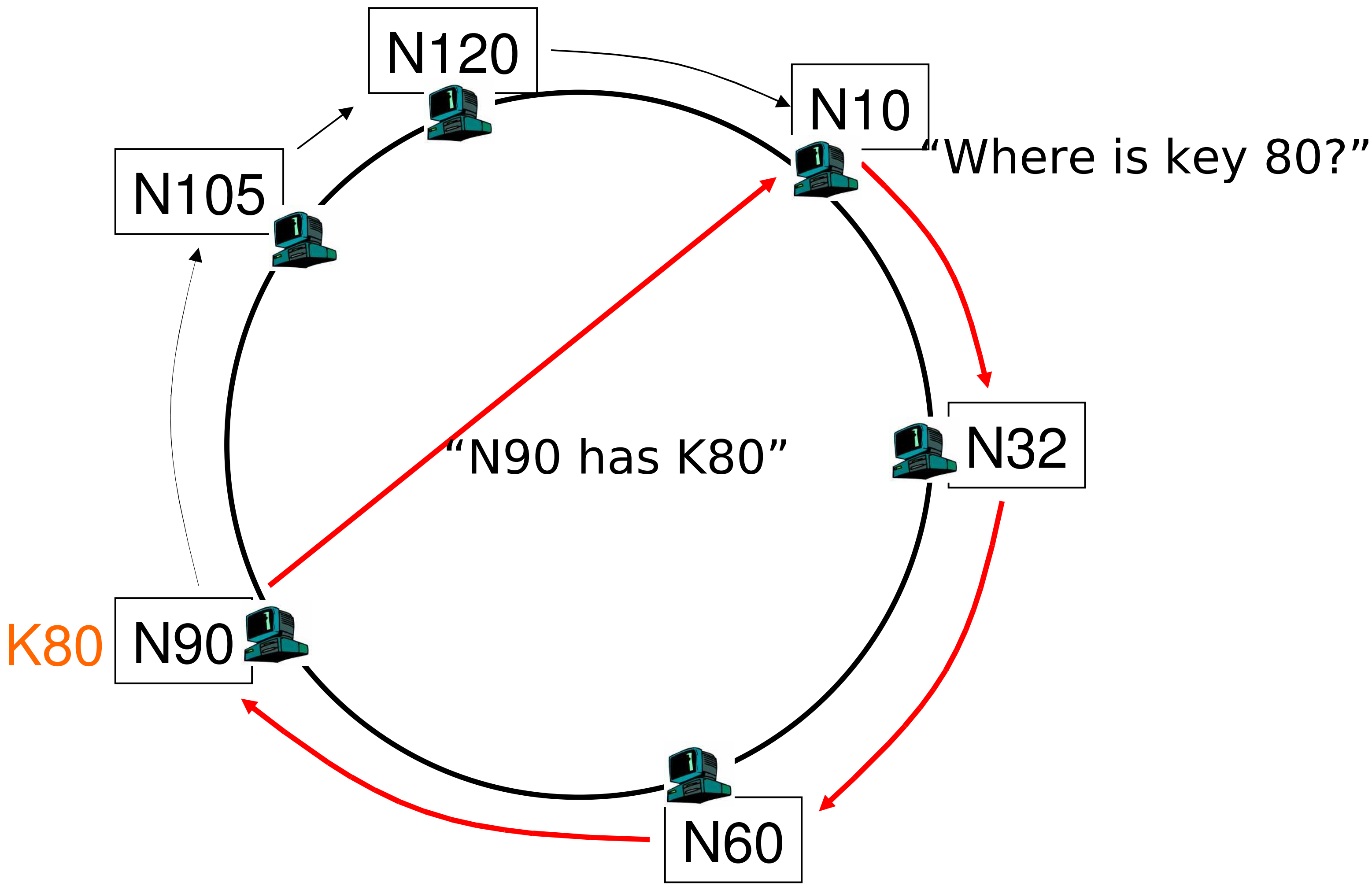


A key is stored at its successor: node with next higher ID

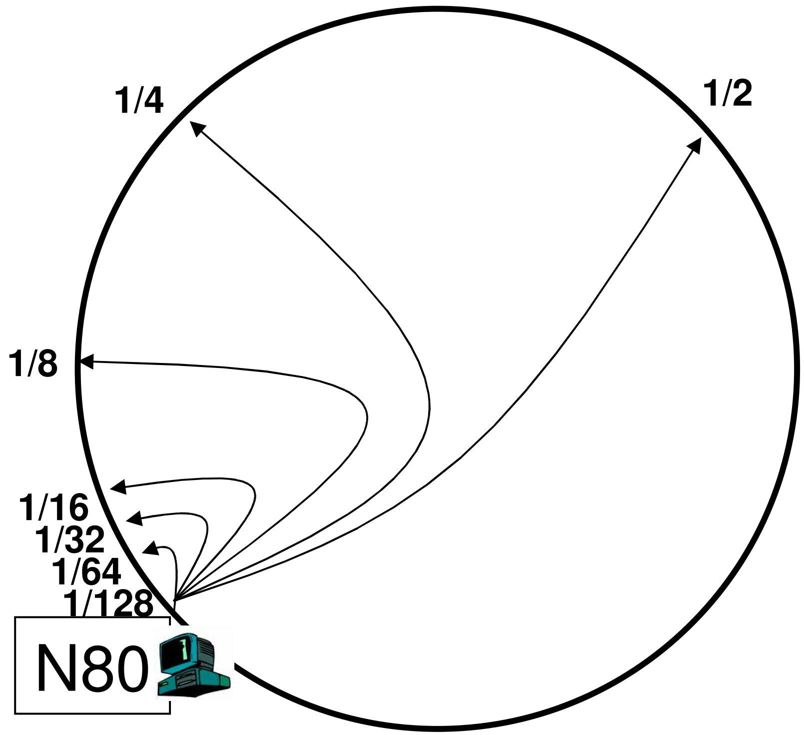
Consistent Hashing

- Very useful algorithmic trick outside of DHTs, etc.
  - Any time you want to not greatly change object distribution upon bucket arrival/departure
- Detail:
  - To have good load balance
  - Must represent each bucket by log(N) “virtual” buckets

DHT: Chord Basic Lookup



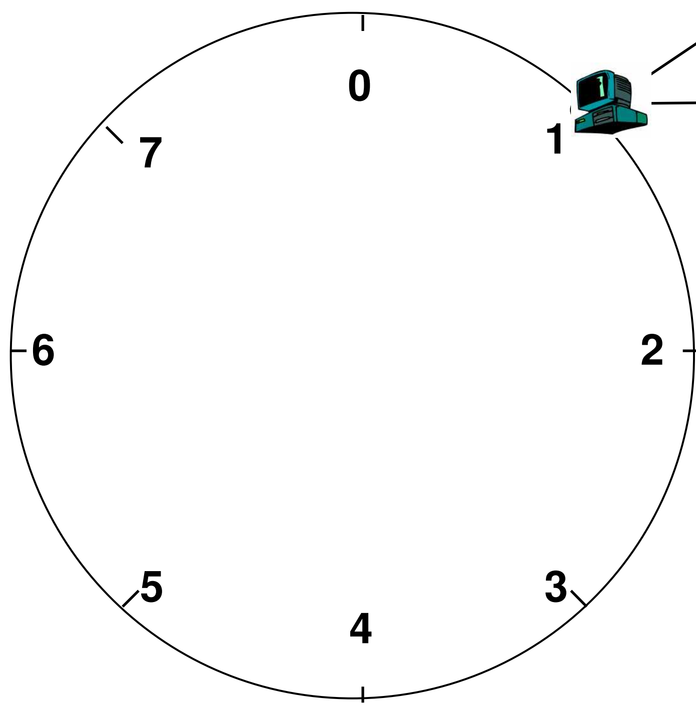
DHT: Chord "Finger Table"



- Entry *i* in the finger table of node *n* is the first node that succeeds or equals $n + 2^i$
- In other words, the ith finger points $1/2^{n-i}$ way around the ring

DHT: Chord Join

- Assume an identifier space [0..8]
- Node n1 joins

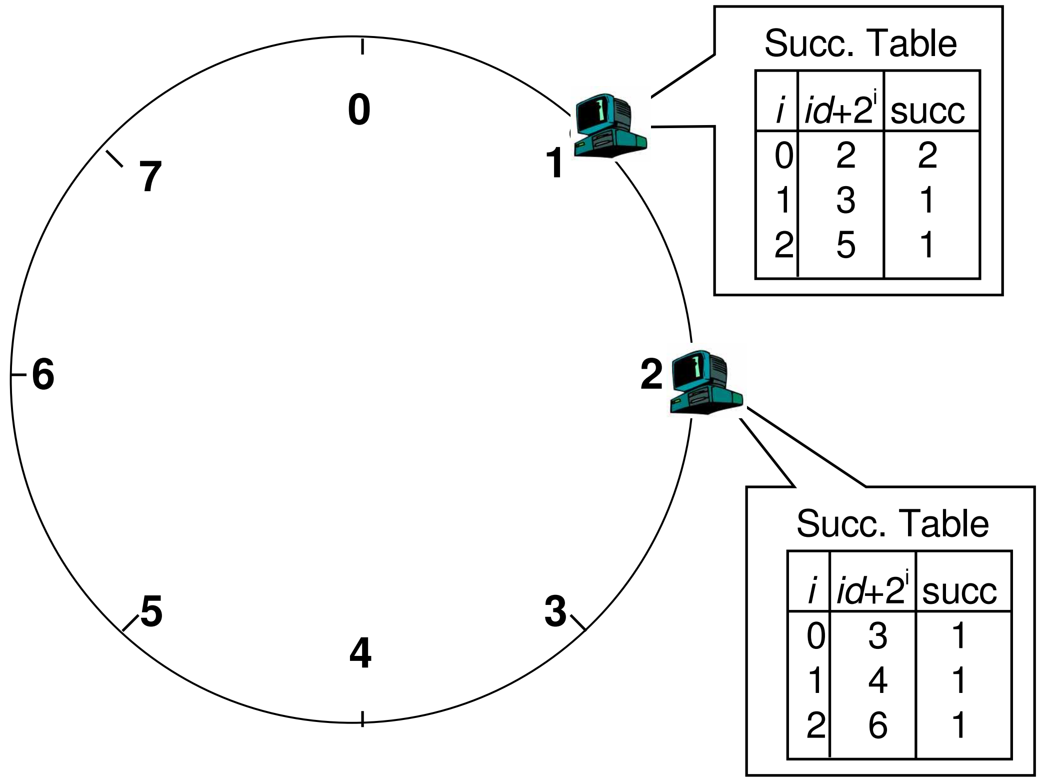


Succ. Table

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

DHT: Chord Join

- Node n2 joins



Node 1 Succ. Table:

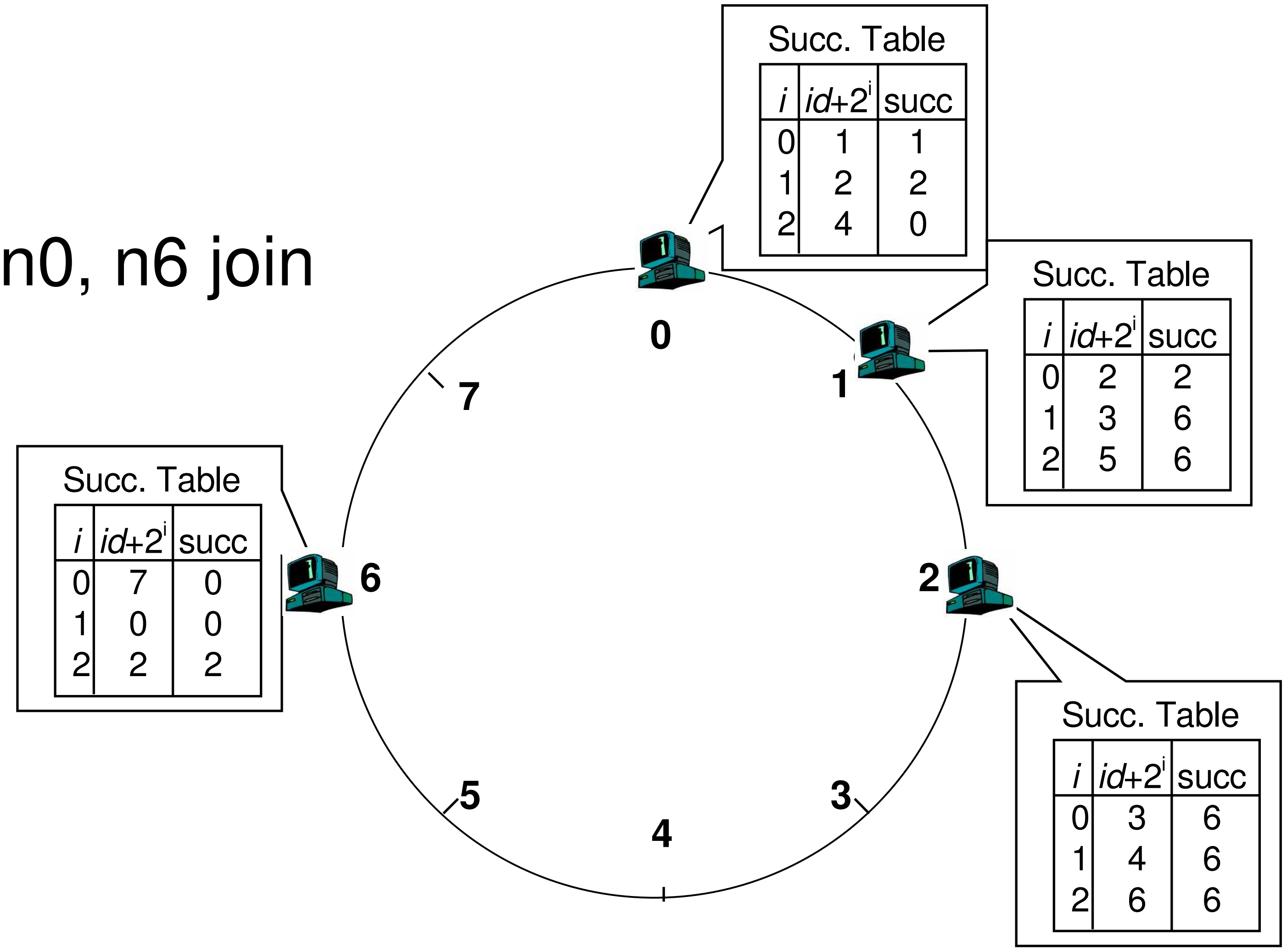
i	$id+2^i$	succ
0	2	2
1	3	1
2	5	1

Node 2 Succ. Table:

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

DHT: Chord Join

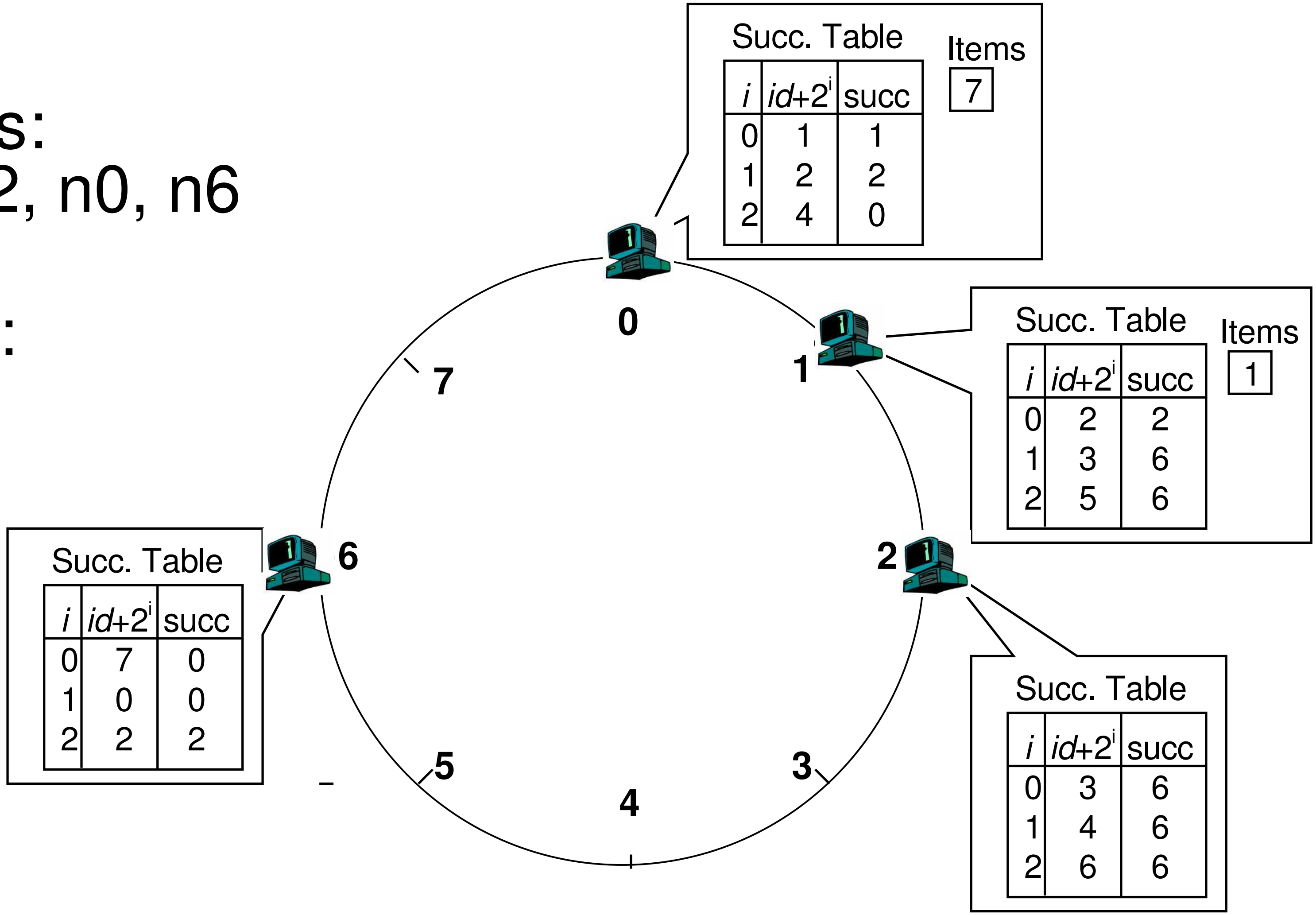
- Nodes n0, n6 join



DHT: Chord Join

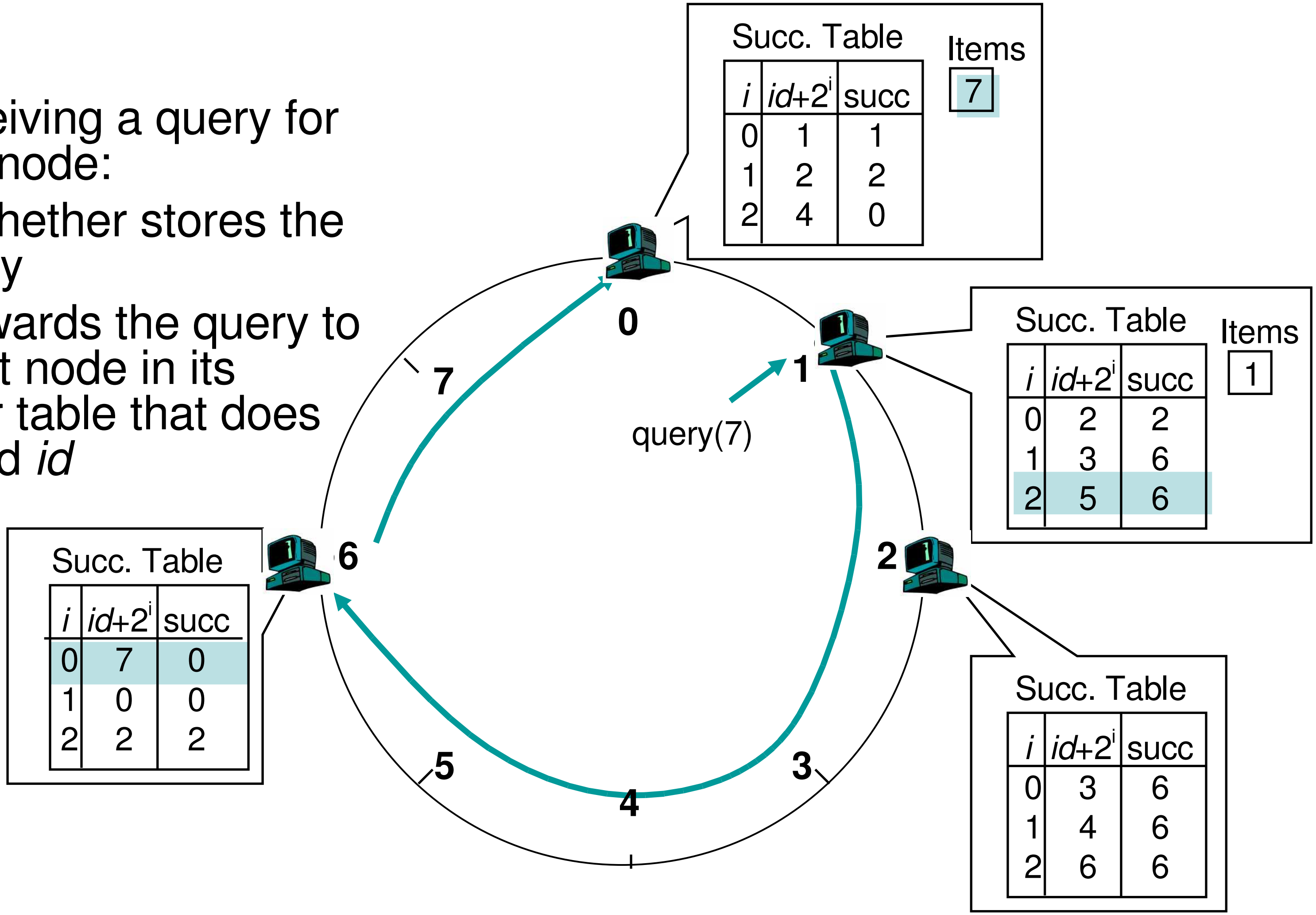
- Nodes:
  n1, n2, n0, n6

- Items:
  f7, f2



DHT: Chord Routing

- Upon receiving a query for item *id*, a node:
- Checks 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.

Alternate structures

- Chord is like a skiplist: each time you go ½ way towards the destination. Other topologies do this too...

Tree-like structures

- Pastry, Tapestry, Kademlia
- Pastry:
  - Nodes maintain a “Leaf Set” size |L|
    - |L|/2 nodes above & below node's ID
    - (Like Chord's successors, but bi-directional)
  - Pointers to log_2(N) nodes at each level i of bit prefix sharing with node, with i+1 *different*
    - e.g., node id 01100101
    - stores to neighbor at 1, 00, 010, 0111, ...

Hypercubes

- the CAN DHT
- Each has ID
- Maintains pointers to a neighbor who differs in one bit position
- Only one possible neighbor in each direction
- But can route to receiver by changing any bit

So many DHTs...

- Compare along two axes:
  - How many neighbors can you choose from *when forwarding*? (Forwarding Selection)
  - How many nodes can you choose from *when selecting neighbors?* (Neighbor Selection)
- Failure resilience: Forwarding choices
- Picking low-latency neighbors: Both help

Proximity

- Ring:
  - Forwarding: log(N) choices for next-hop when going around ring
  - Neighbor selection: Pick from 2^i nodes at “level” i (great flexibility)
- Tree:
  - Forwarding: 1 choice
  - Neighbor: 2^i-1 choices for ith neighbor

Hypercube

- Neighbors: 1 choice
  - (neighbors who differ in one bit)
- Forwarding:
  - Can fix any bit you want.
  - N/2 (expected) ways to forward
- So:
  - Neighbors: Hypercube 1, Others: 2^i
  - Forwarding: tree 1, hypercube logN/2, ring logN

How much does it matter?

- Failure resilience *without* re-running routing protocol
  - Tree is much worse; ring appears best
  - But all protocols can use multiple neighbors at various levels to improve these #s
- Proximity
  - Neighbor selection more important than route selection for proximity, and draws from large space with everything but hypercube

Other approaches

- Instead of log(N), can do:
  - Direct routing (everyone knows full routing table)
    - Can scale to tens of thousands of nodes
    - May fail lookups and re-try to recover from failures/additions
  - One-hop routing with sqrt(N) state instead of log(N) state
- What's best for real applications? Still up in the air.

DHT: Discussion

- Pros:
  - Guaranteed Lookup
  - O(log *N*) per node state and search scope
    - (Or otherwise)
- Cons:
  - Hammer in search of nail? Now becoming popular in p2p – Bittorrent "Distributed Tracker". But still waiting for massive uptake. Or not.
  - Many services (like Google) are scaling to huge #s without DHT-like log(N) techniques

Further Information

- We didn't talk about Kademlia's XOR structure (like a generalized hypercube)
- See “The Impact of DHT Routing Geometry on Resilience and Proximity” for more detail about DHT comparison
- No silver bullet: DHTs very nice for exact match, but not for everything (next few slides)

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 cable modem upload bandwidth is tiny - perhaps 10 Gbytes/day
  - Takes many days to upload contents of 200GB hard drive. Very expensive leave/replication situation!

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)