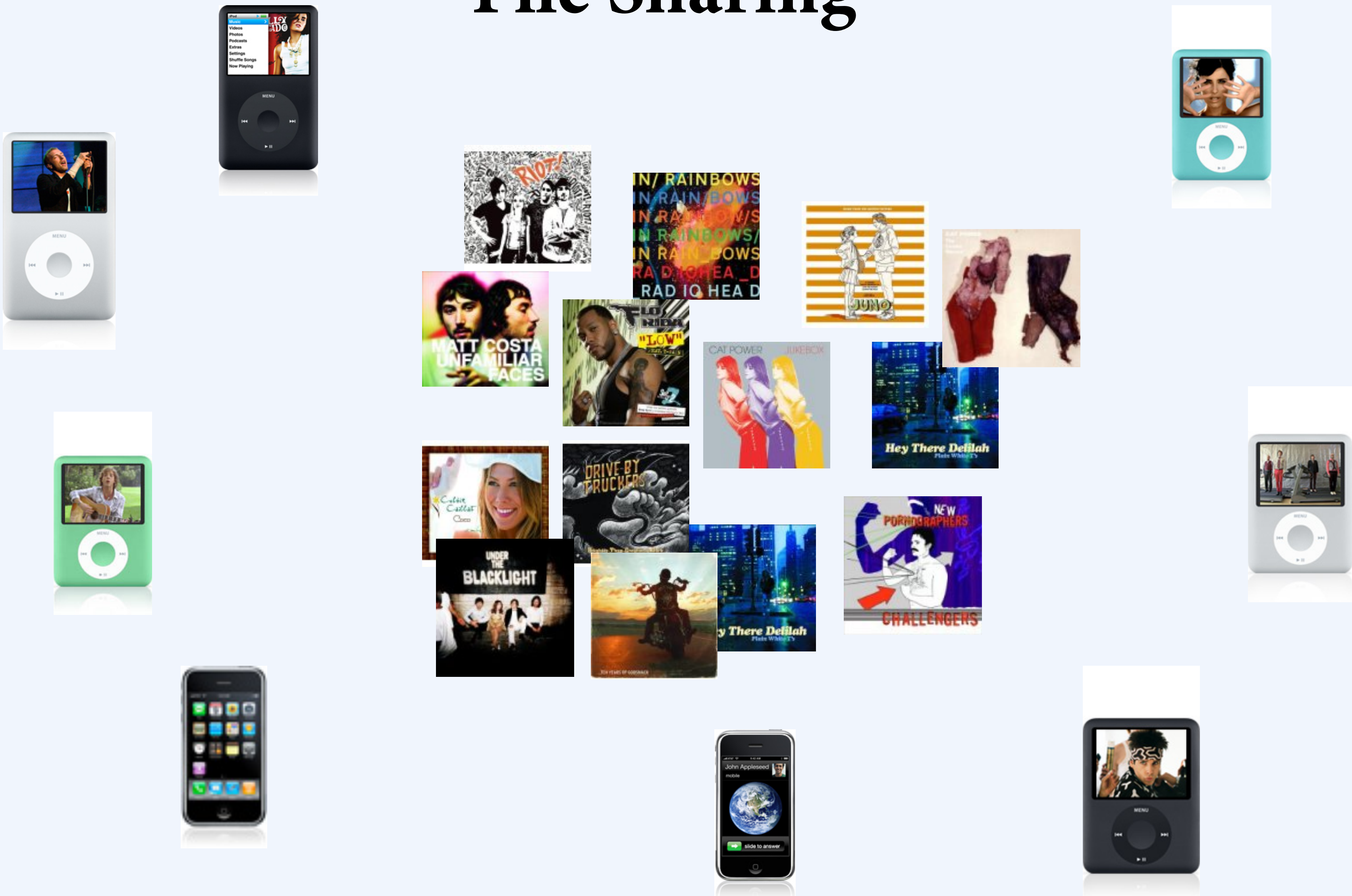
Peer to Peer I

Roadmap

- This course will feature key concepts in Distributed Systems, often illustrated by their use in example systems
- Start with Peer-to-Peer systems, which will be useful for your projects
  - Napster, Gnutella
  - Chord (this class)
  - Tapestry (next class)
  - Use in filesystems

File Sharing



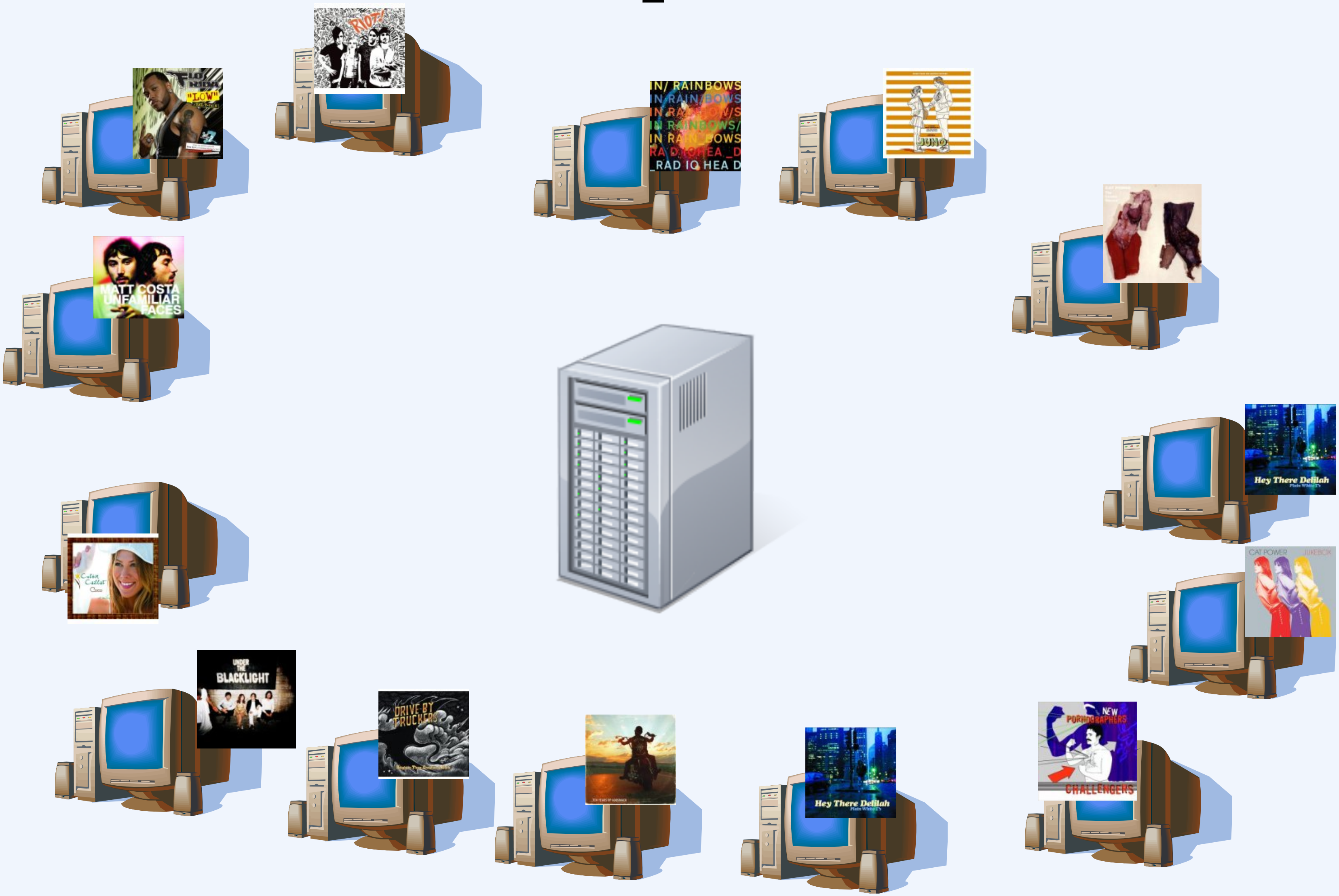
Peer-to-Peer Systems

- How did it start?
  - A killer application: file distribution
  - Free music over the Internet! (*not exactly legal…*)
- Key idea: share storage, content, and bandwidth of individual users
  - Lots of them
- Big challenge: coordinate all of these users
  - In a scalable way (not NxN!)
  - With changing population (aka *churn*)
  - With no central administration
  - With no trust
  - With large heterogeneity (content, storage, bandwidth, …)

3 Key Requirements

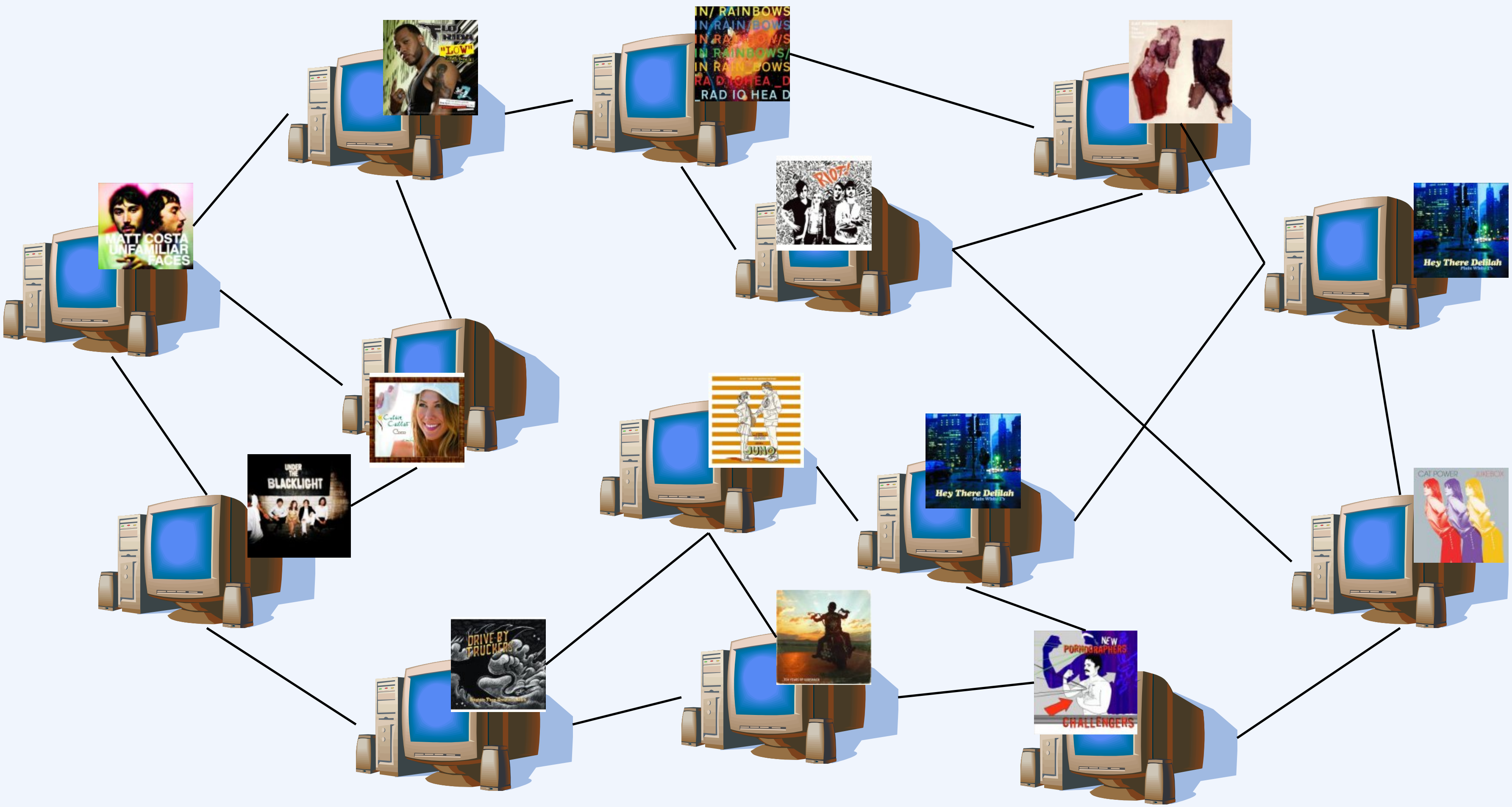
- P2P Systems do three things:
- Help users determine what they want
  - Some form of search
  - P2P version of Google
- Locate that content
  - Which node(s) hold the content?
  - P2P version of DNS (map name to location)
- Download the content
  - Should be efficient
  - P2P form of Akamai

Napster


IN RAINBOWS
RADIOHEAD
JUNO
MATT COSTA
UNFAMILIAR FACES
Hey There Delilah
CAT POWER
JUKEBOX
UNDER THE BLACKLIGHT
DRIVE BY TRUCKERS
NEW PORNOGRAPHERS
CHALLENGERS
Hey There Delilah

Napster Problems

Gnutella


IN RAINBOWS
RADIOHEAD
MATT COSTA
UNFAMILIAR FACES
Hey There Delilah
UNDER THE BLACKLIGHT
JUNO
Hey There Delilah
DRIVE-BY TRUCKERS
NEW PORNOGRAPHERS
CHALLENGERS

Some Details

- Participants interconnect via *overlay network*
- To send a query:
  - send request to each directly connected node
  - proceed for some maximum number of hops
    - node having desired file sends back its identity
      - over reverse query route in original Gnutella
      - direct via UDP in later Gnutella
    - querier chooses a source (if necessary)
    - sends it a push request
      - transfer via HTTP

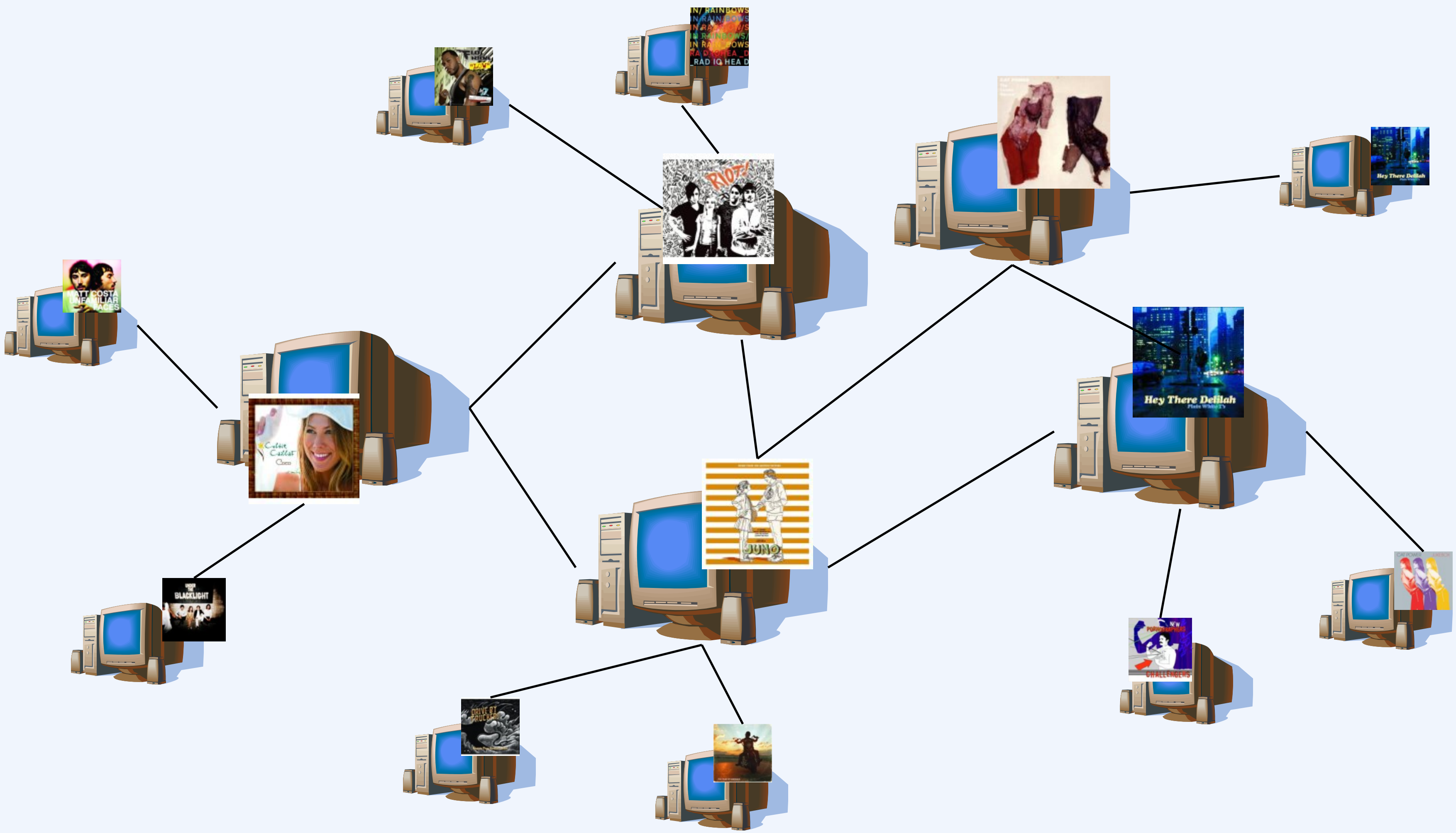
More Details

- **Joining the overlay network:**
  - obtain addresses of some number of network nodes
    - wired into code
    - check web site
    - etc.
  - contact them; they produce address of other nodes
  - connect to *n* of them
  - keep others cached for later use

Problems

- Flaky network connections
- Flaky computers
- Flaky users

Solution: Ultrapeers



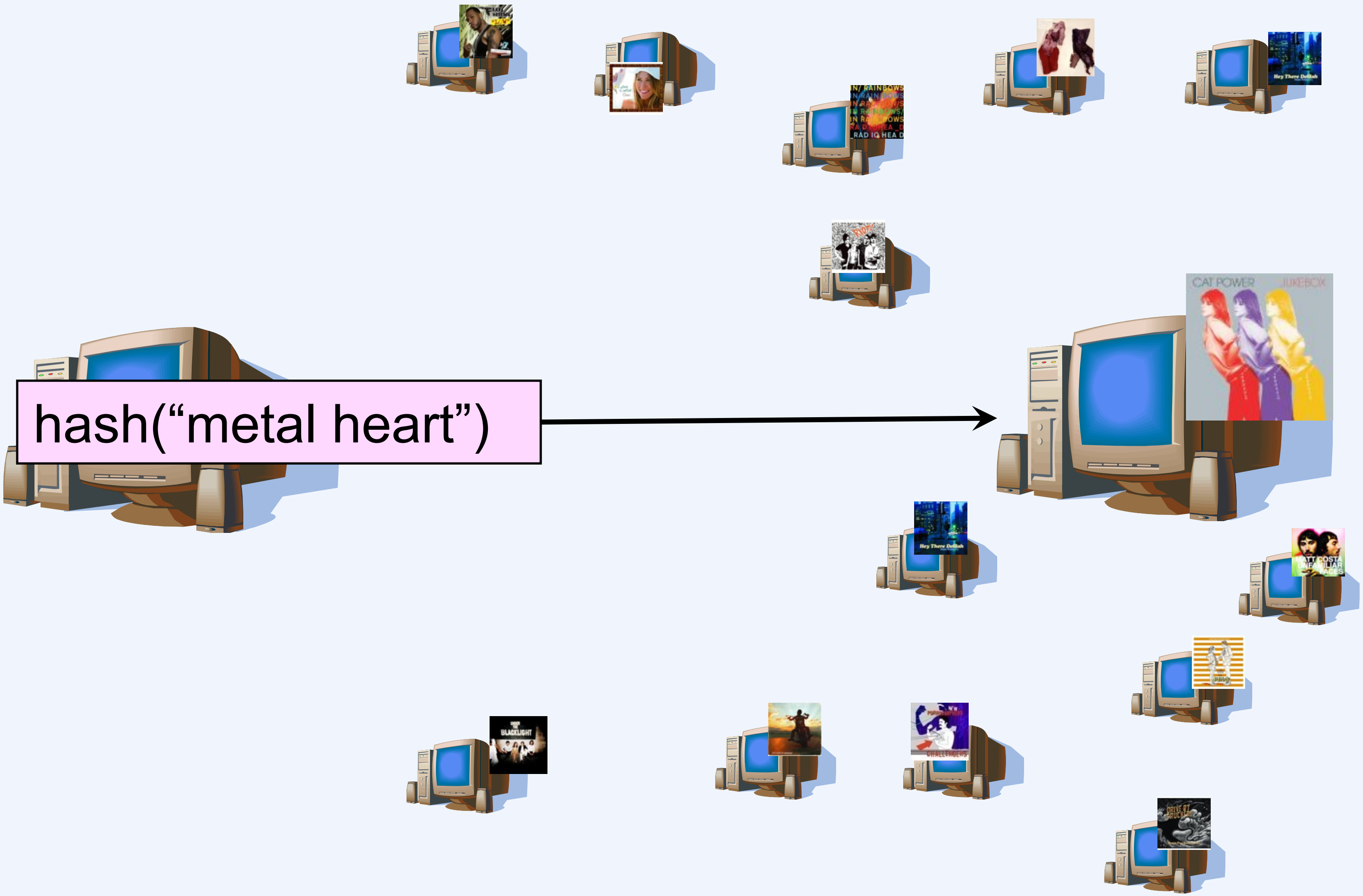
Lessons and Limitations

- Client-server simple and effective
  - But not always feasible
- Things that flood-based systems do well
  - Decentralization of visibility and liability
  - Finding popular stuff
  - Fancy *local* queries
- Things that flood-based systems do poorly
  - Scale (exponential increase in traffic vs hops)
  - Finding unpopular stuff
  - Fancy *distributed* queries
  - Vulnerabilities: data poisoning, tracking, etc.
  - Guarantees about anything (answer quality, privacy, etc.)

Second generation P2P

- Structured P2P systems, mostly academic efforts
- Goal: solve the scalable decentralized location problem

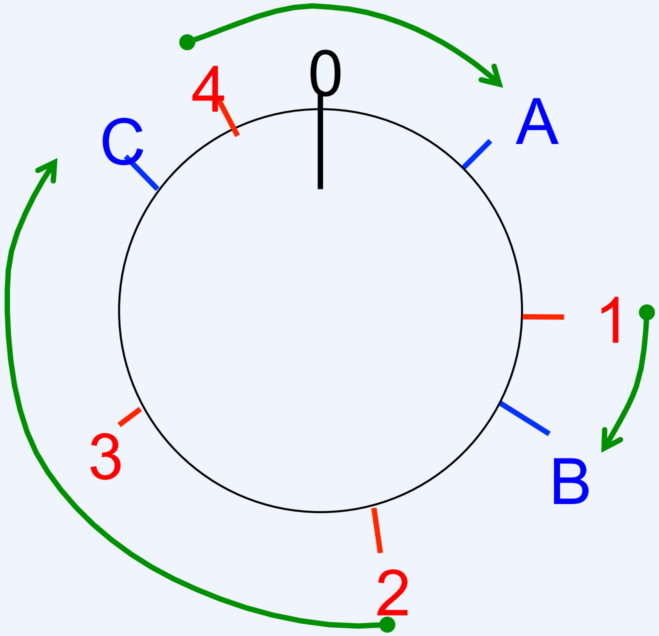
Distributed Hash Tables



Straw man: modulo hashing

- Say you have N servers
- Map requests to servers as follows:
  - Number servers 0 to N-1
  - Compute hash of content: h = hash (name)
  - Redirect client to server #p = h mod N
- Keep track of load in each proxy
  - If load on proxy #p is too high, try again with a different hash function (or "salt")
- Problem: most caches will be useless if you add or remove proxies, change value of N

Consistent Hashing [Karger et al., 99]



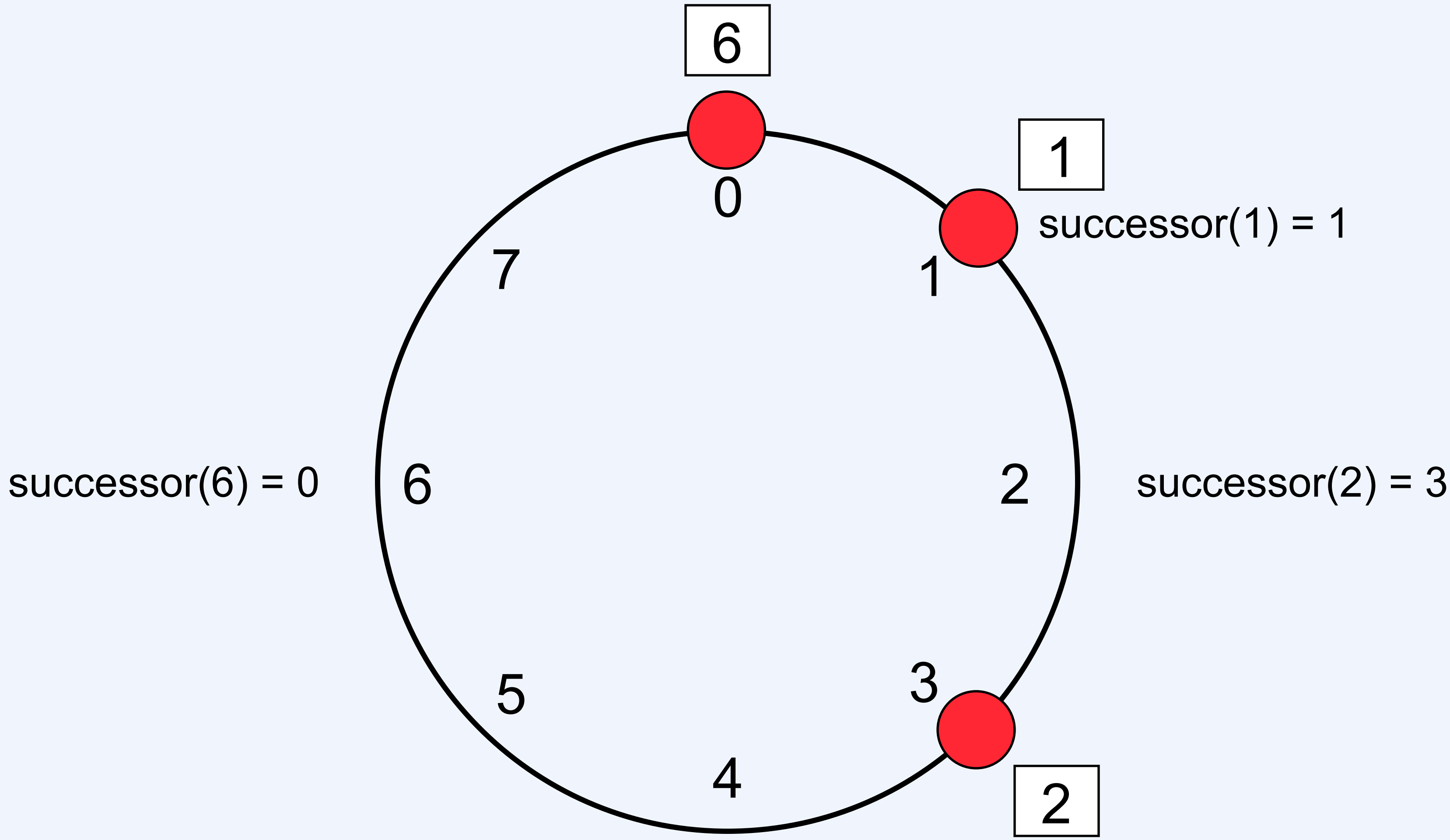
Object	Server
1	B
2	C
3	C
4	A

- Servers and objects mapped to points on a circle using hash
- An object is assigned to its successor server
- Minimizes data movement on change!
  - Only O(1/N) objects moved on server leave/join
  - Which ones?

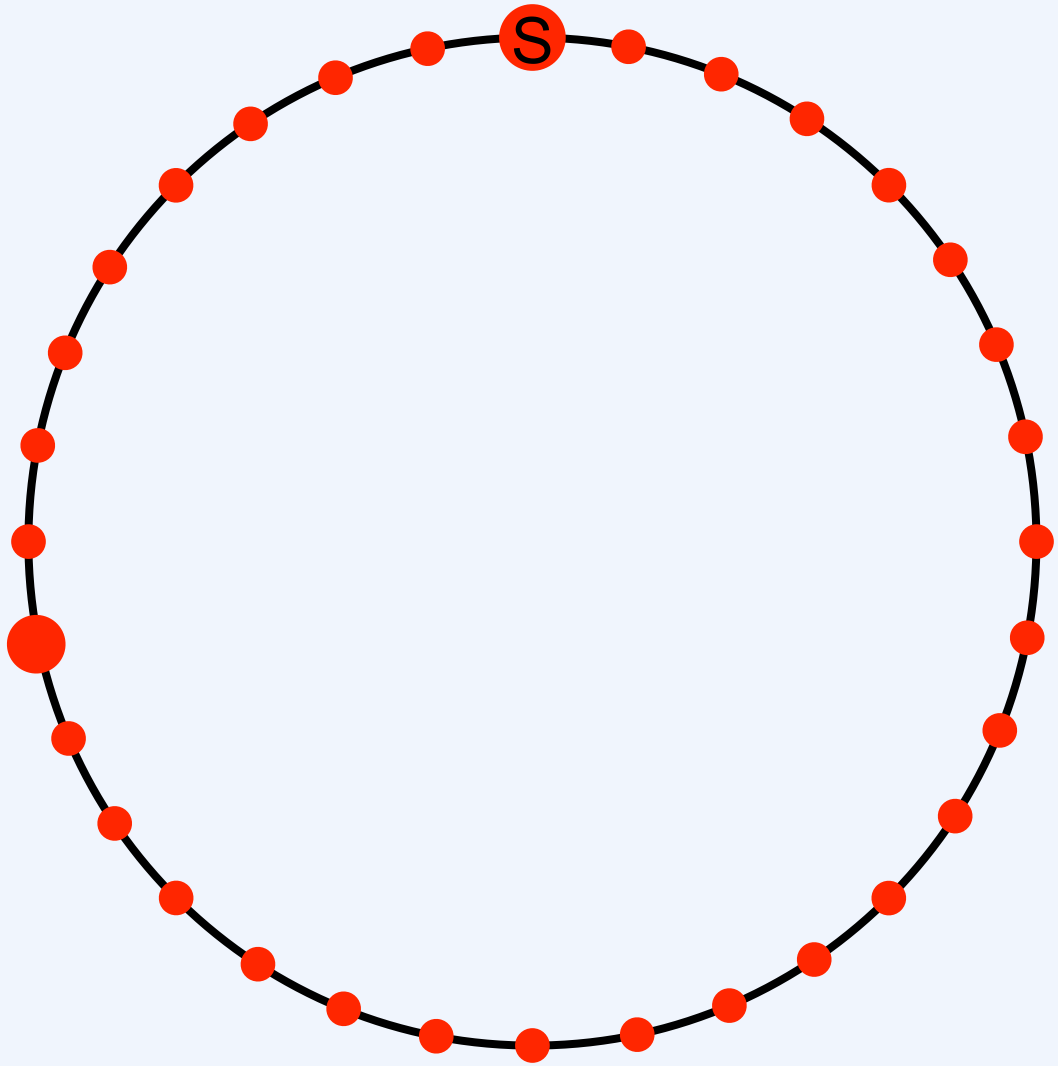
Chord

- Distributed hash tables meet overlay networks
  - hash both keys and node IP addresses into identifiers
    - m-bit identifiers, where m is large enough so that probability of collision is negligible
  - lookups resolved in O(log n) messages
  - adding or deleting a node requires $O(\log^2 n)$ messages

Chord



Search

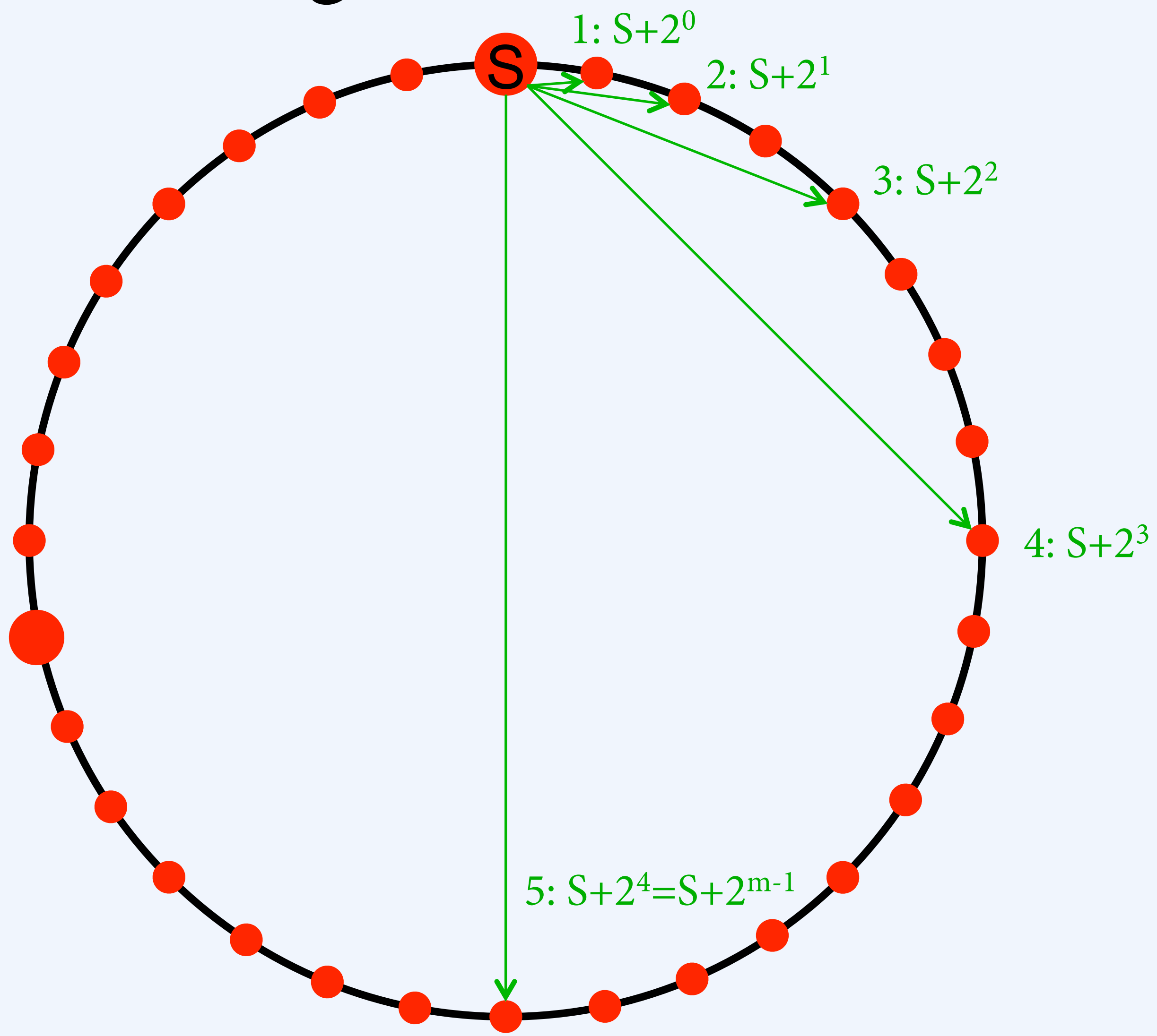


Speeding It Up

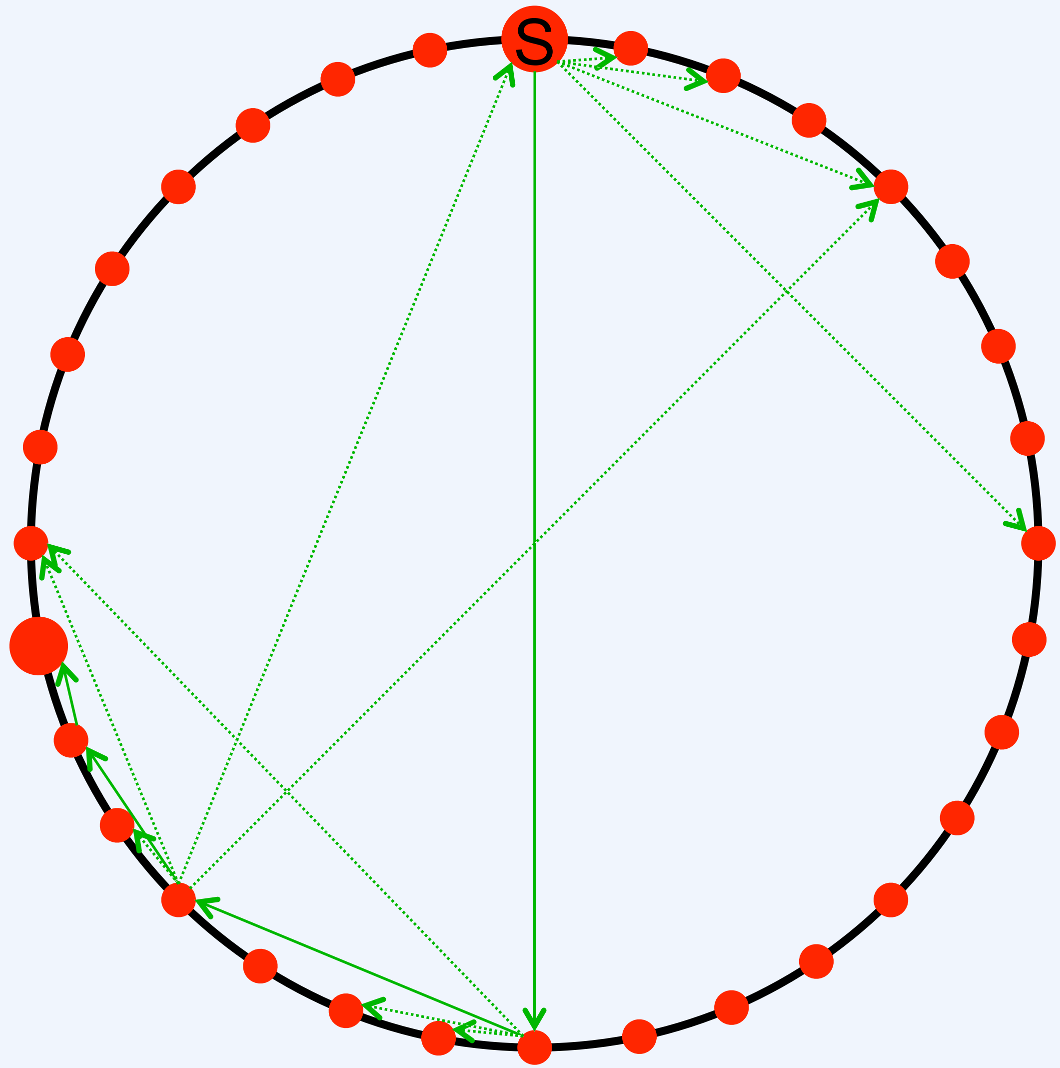
- Find highest-numbered node that is smaller than the key (modulo 2^m)
  - the next node is the successor of the key
- Finger table
  - contains pointers to nodes:
    - half-way around circle
    - ¼-way around circle
    - etc.
  - include with each node *i* an m-entry table
    - j^{th} entry refers to the smallest-numbered node that exceeds i by at least 2^{j-1} (modulo 2^m)

Finger table for S

m = 5



Search with Finger Table

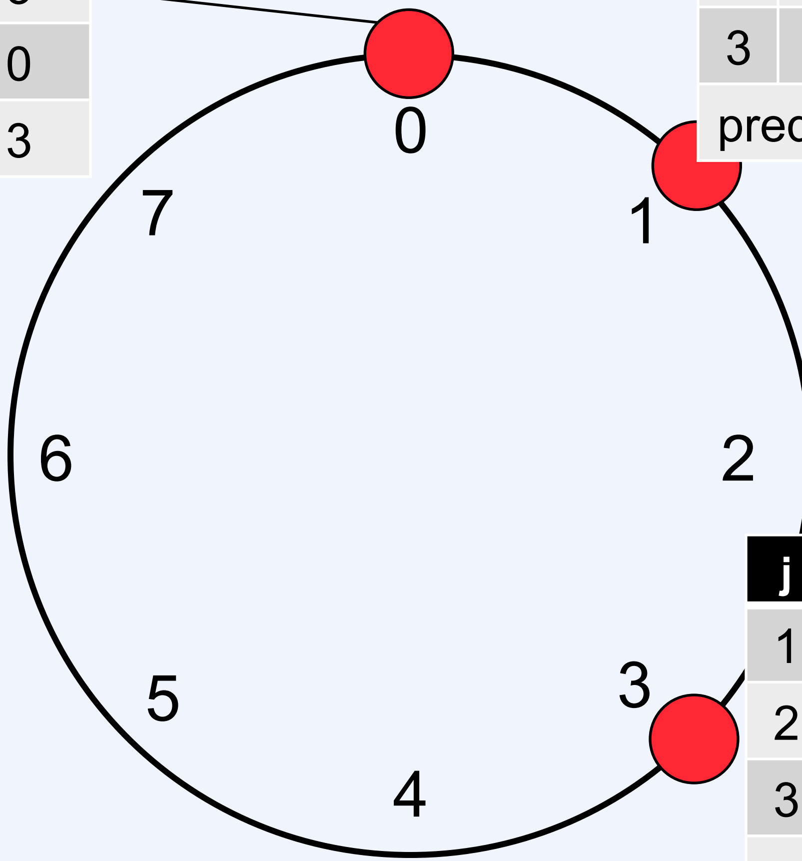


Finding an Object

```
n.find_successor(id) {
   n1 = find_predecessor(id)
   return n1.successor
}

n.find_predecessor(id) {
   n1 = n
   while (id ∉ (n1, n1.successor])
      n1 = n1.closest_preceding_finger(id)
   return n1
}

n.closest_preceding_finger(id) {
   for i = m downto 1
      if (finger[i].node ∈ (n, id))
         return finger[i].node
   return n
}
```

Chord

j	start	interval	node
1	1	[1, 2)	1
2	2	[2, 4)	3
3	4	[4, 0)	0
predecessor			3

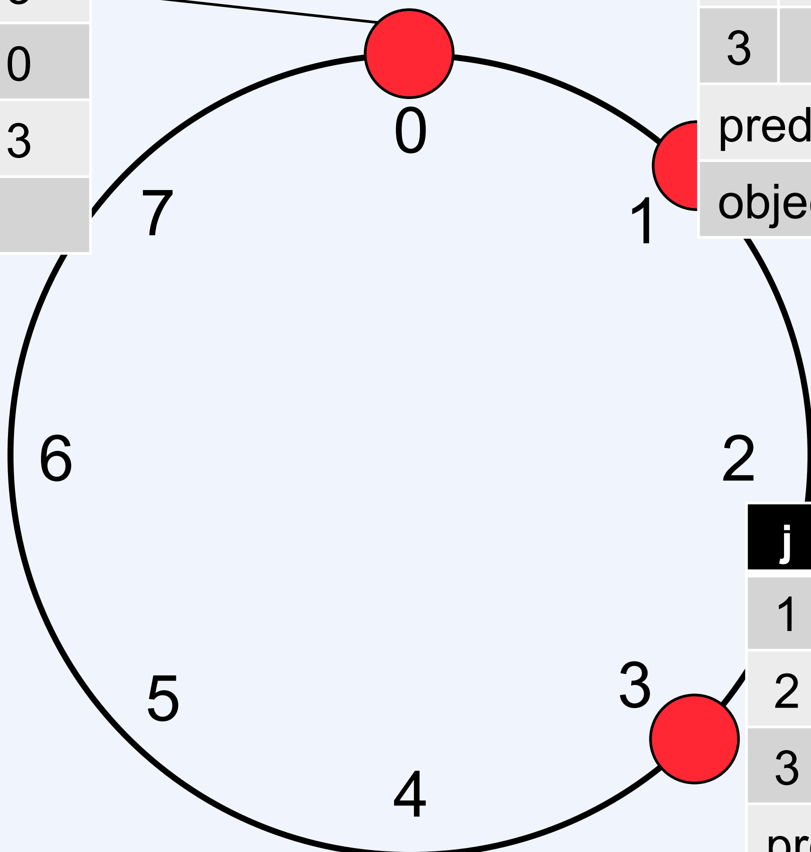


j	start	interval	node
1	2	[2,3)	3
2	3	[3,5)	3
3	5	[5, 1)	0
predecessor			0

j	start	interval	node
1	4	[4,5)	0
2	5	[5,7)	0
3	7	[7,3)	0
predecessor			1

Chord with Objects

j	start	interval	node
1	1	[1, 2)	1
2	2	[2, 4)	3
3	4	[4, 0)	0
predecessor			3
objects	5, 7		



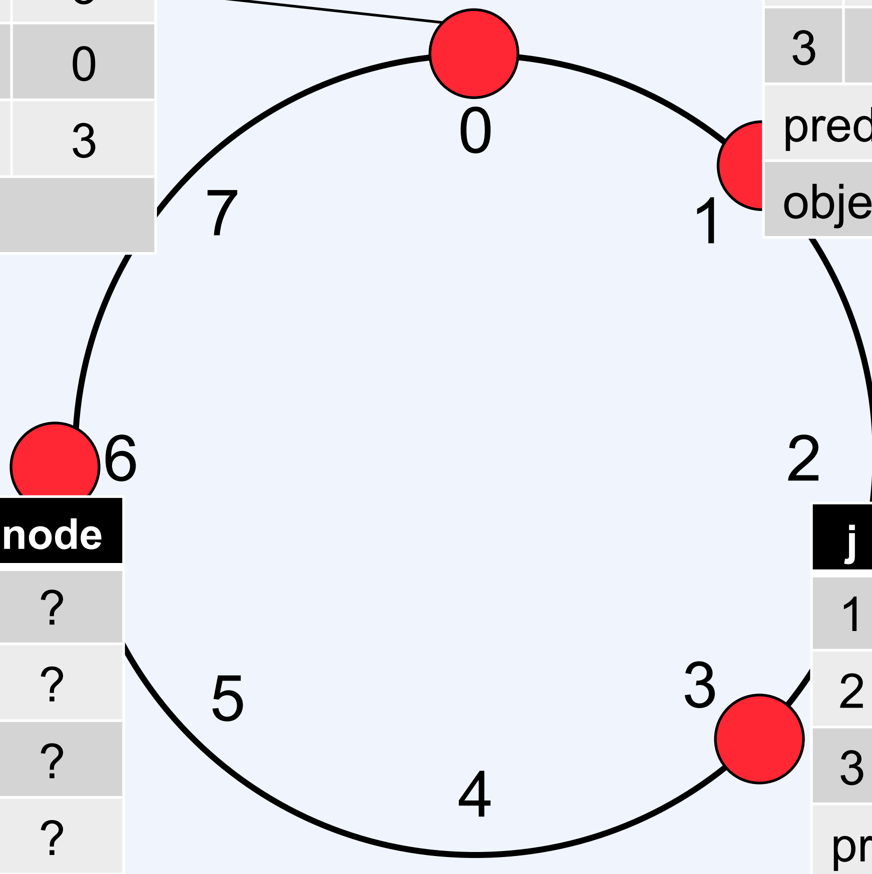
j	start	interval	node
1	2	[2,3)	3
2	3	[3,5)	3
3	5	[5, 1)	0
predecessor			0
objects	1		

j	start	interval	node
1	4	[4,5)	0
2	5	[5,7)	0
3	7	[7,3)	0
predecessor			1
objects	2		

Adding a Node

j	start	interval	node
1	1	[1, 2)	1
2	2	[2, 4)	3
3	4	[4, 0)	0
predecessor			3
objects	5, 7		

j	start	interval	node
1	2	[2,3)	3
2	3	[3,5)	3
3	5	[5, 1)	0
predecessor			0
objects	1		

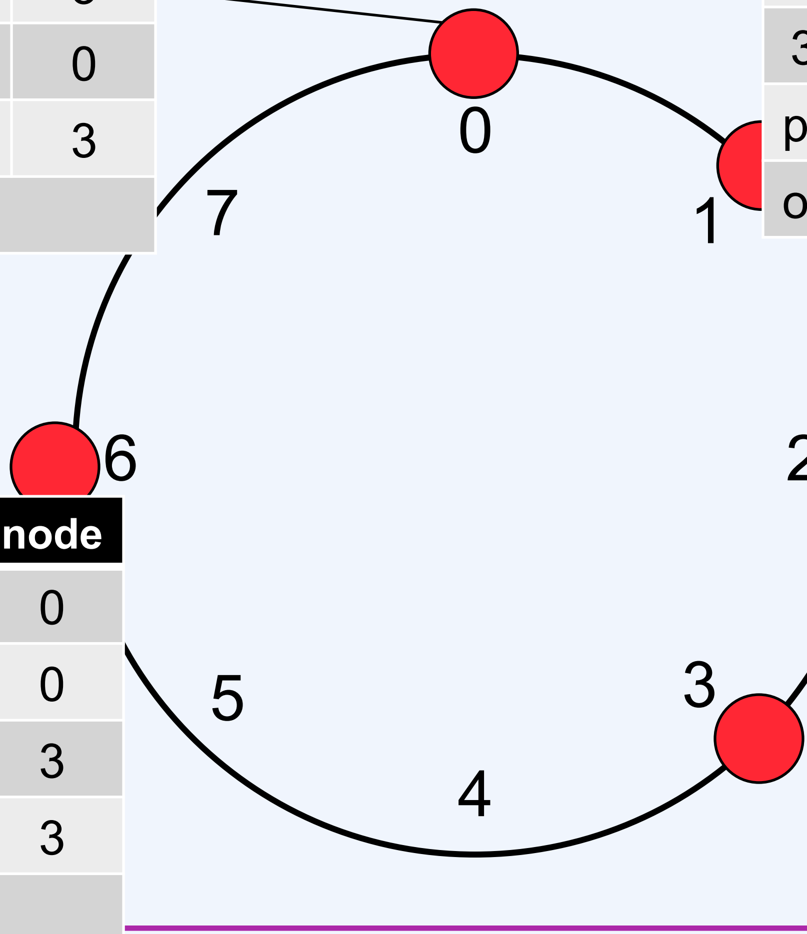


j	start	interval	node
1	7	[7, 0)	?
2	0	[0, 2)	?
3	2	[2, 6)	?
predecessor			?
objects			

j	start	interval	node
1	4	[4,5)	0
2	5	[5,7)	0
3	7	[7,3)	0
predecessor			1
objects	2		

Setting up the Finger Table

j	start	interval	node
1	1	[1, 2)	1
2	2	[2, 4)	3
3	4	[4, 0)	0
predecessor			3
objects	5, 7		



j	start	interval	node
1	2	[2,3)	3
2	3	[3,5)	3
3	5	[5, 1)	0
predecessor			0
objects	1		

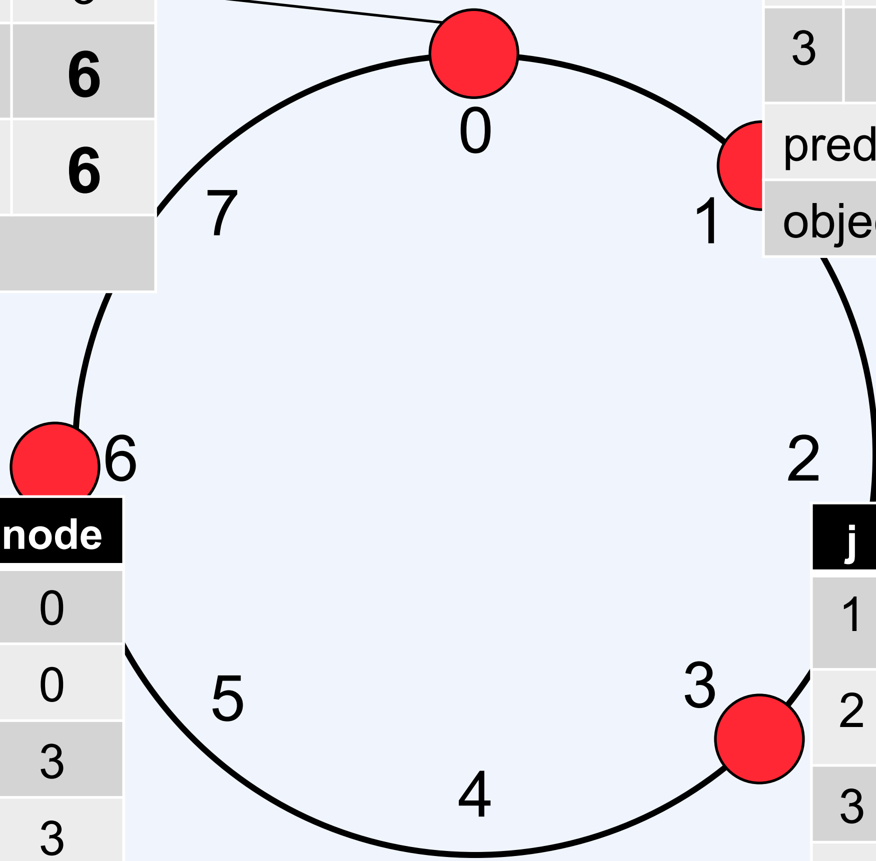
j	start	interval	node
1	7	[7, 0)	0
2	0	[0, 2)	0
3	2	[2, 6)	3
predecessor			3
objects			

j	start	interval	node
1	4	[4,5)	0
2	5	[5,7)	0
3	7	[7,3)	0
predecessor			1
objects	2		

Updating Others' Tables

j	start	interval	node
1	1	[1, 2)	1
2	2	[2, 4)	3
3	4	[4, 0)	**6**
predecessor			**6**
objects	5, 7		

j	start	interval	node
1	2	[2,3)	3
2	3	[3,5)	3
3	5	[5, 1)	**6**
predecessor			0
objects	1		



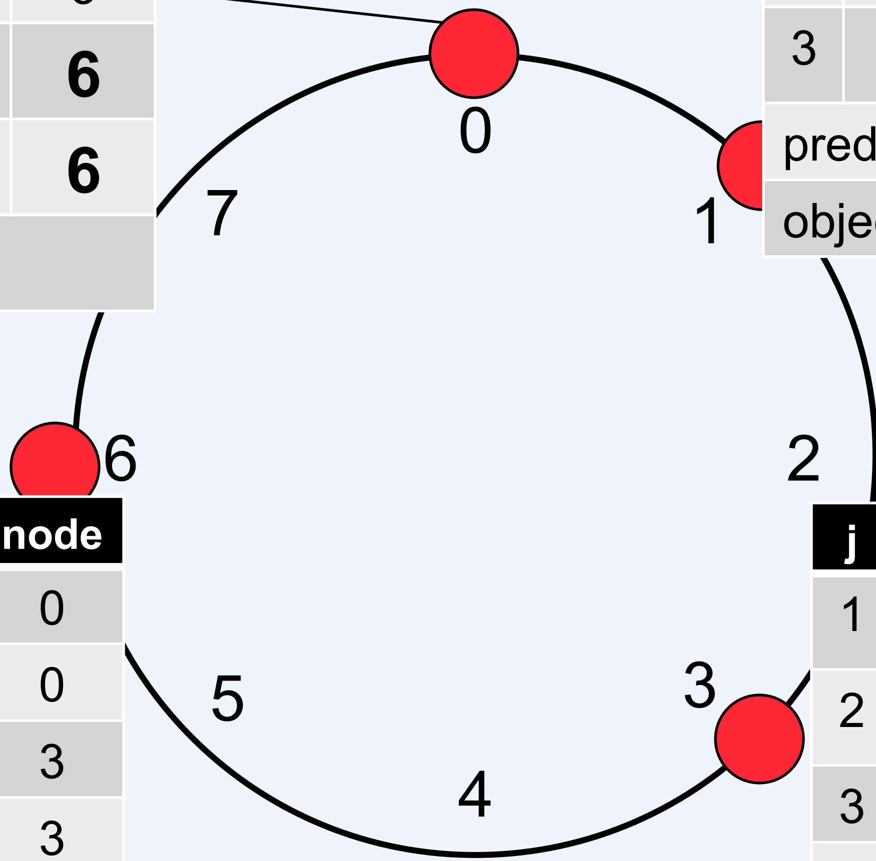
j	start	interval	node
1	7	[7, 0)	0
2	0	[0, 2)	0
3	2	[2, 6)	3
predecessor			3
objects			

j	start	interval	node
1	4	[4,5)	**6**
2	5	[5,7)	**6**
3	7	[7,3)	0
predecessor			1
objects	2		

Redistributing Objects

j	start	interval	node
1	1	[1, 2)	1
2	2	[2, 4)	3
3	4	[4, 0)	**6**
predecessor			**6**
objects	**7**		

j	start	interval	node
1	2	[2,3)	3
2	3	[3,5)	3
3	5	[5, 1)	**6**
predecessor			0
objects	1		



j	start	interval	node
1	7	[7, 0)	0
2	0	[0, 2)	0
3	2	[2, 6)	3
predecessor			3
objects	**5**		

j	start	interval	node
1	4	[4,5)	**6**
2	5	[5,7)	**6**
3	7	[7,3)	0
predecessor			1
objects	2		

Adding a Node

adding node *n*

1) **Initialize n's finger table**

   – **find some existing node p**

     for i = 1 to m

       finger[i].node =

       p.find_successor(finger[i].start)

     predecessor = finger[1].node.predecessor

An Improvement

There are no nodes in this range …

j	start	interval	node
1	7	[7, 0)	0
2	0	[0, 2)	?
3	2	[2, 6)	?
predecessor			3
objects			

Node 6's partially filled-in finger table

… thus the node value is the same as the one above it (0).

Adding a Node (Improved)

adding node *n*

1) Initialize n's finger table

 – find some existing node p

```
finger[1].node =
 p.find_successor(finger[1].start)
for i = 1 to m-1
    if (finger[i+1].start ∈ (n, finger[i].node])
        finger[i+1].node = finger[i].node
    else
        finger[i+1].node =
                p.find_successor(finger[i+1].start)
predecessor = finger[1].node.predecessor
```

Adding a Node

2) Update others' finger tables

```
for i = 1 to m
    // find last node p whose i^th finger might be new_node
    p = find_predecessor(new_node-2^(i-1))
    p.update_finger_table(new_node, i)

n.update_finger_table(s, i)
    if (s ∈ [n, finger[i].node))
     finger[i].node = s
     p = predecessor
     p.update_finger_table(s, i)
```

Adding a Node

3) Move objects in (predecessor, n] from the node immediately following the new node

Issues

- What if a search takes place while a node is being added?
- What if multiple nodes are added concurrently?

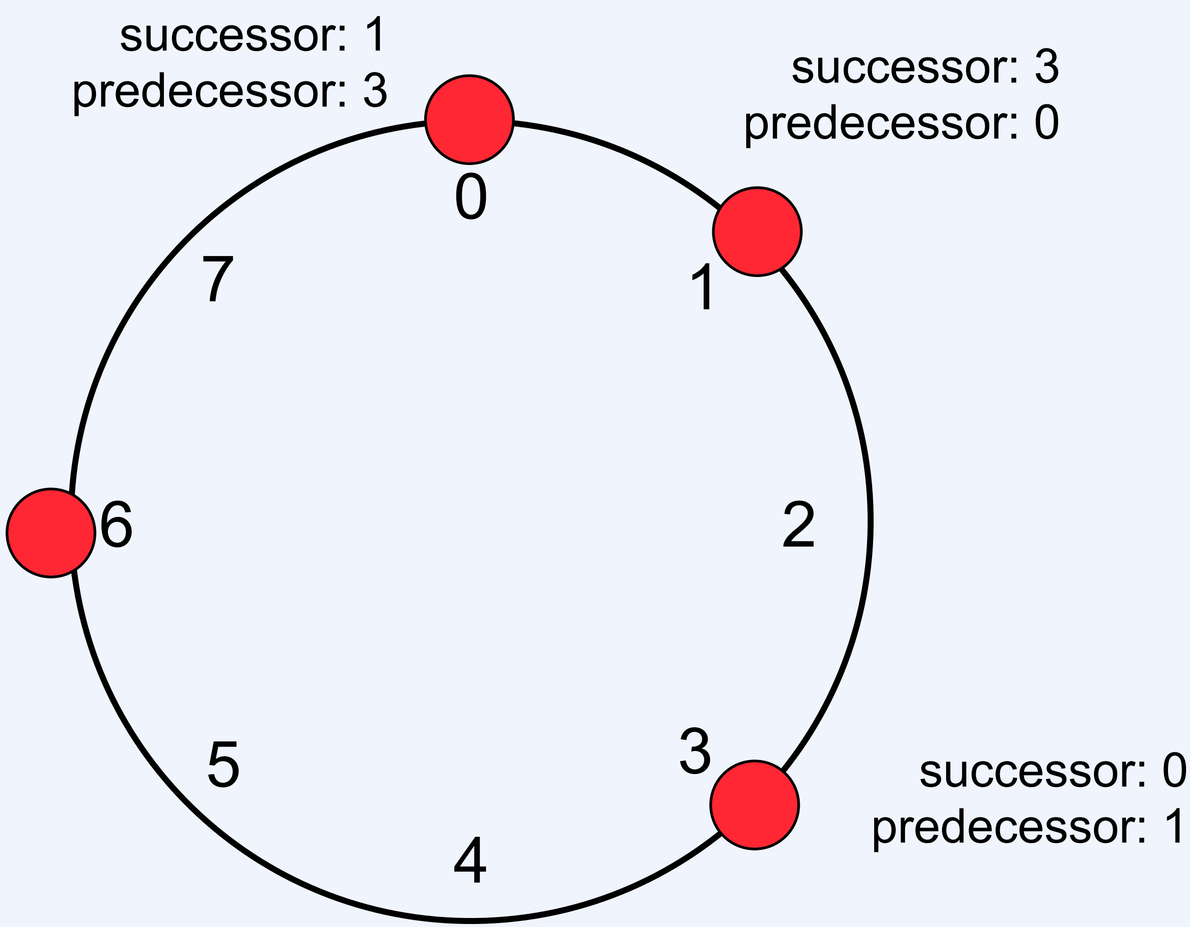
Invariants

- Each node's successor link is correct
- For every key k, successor(k) is responsible for k

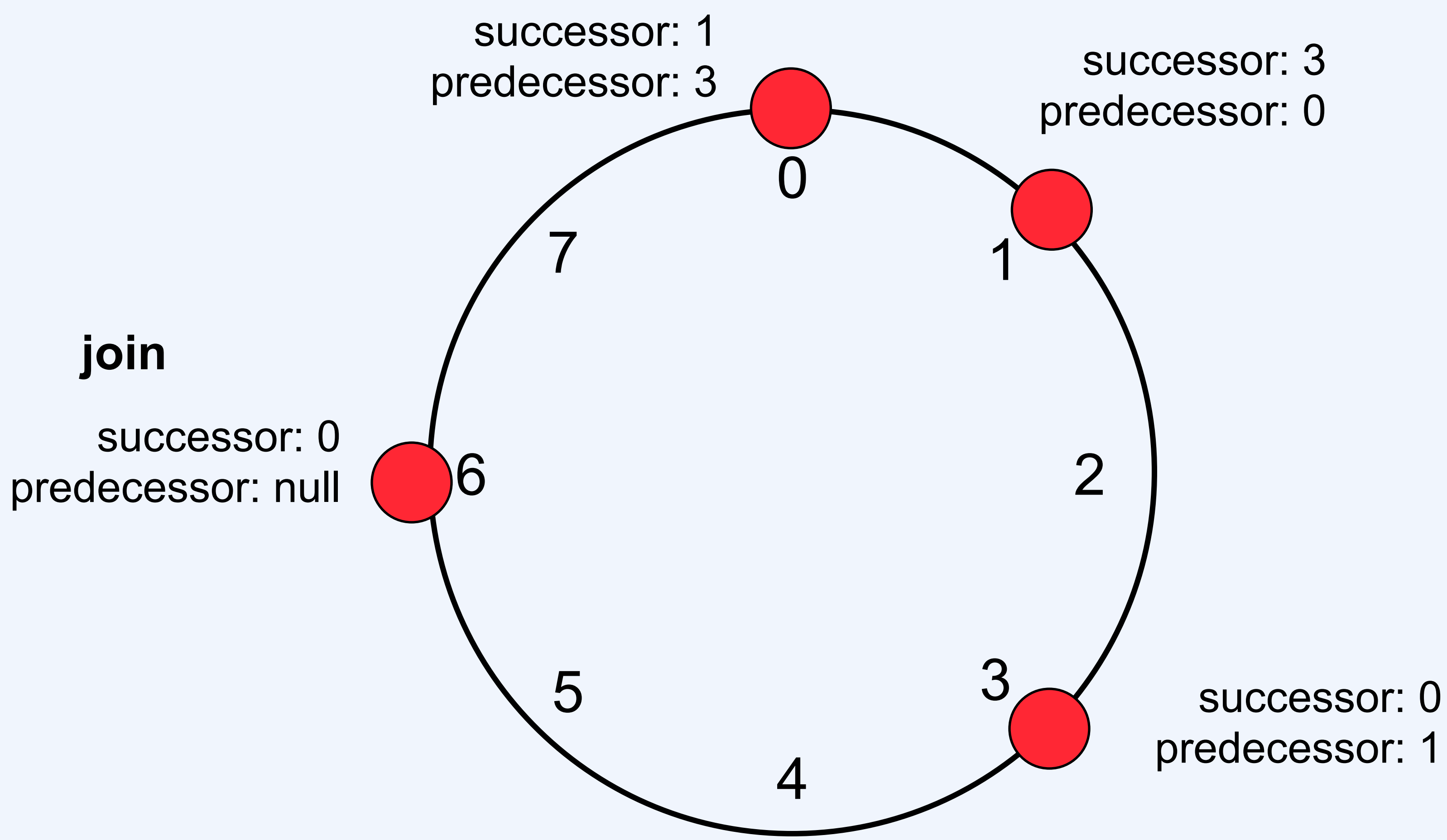
Stabilization

```
n.join(p) //p is some node you know
    predecessor = null
    successor = p.find_successor(n)

// this is run periodically
// verify n's successor, and tell n's successor about n
n.stabilize()
    x = successor.predecessor()  //what is your
      predecessor?
    if (x is between n and successor)
        successor = x
    successor.notify(n)

n.notify(p) // "p to n: I think I am your predecessor"
    if (predecessor == null or p is between predecessor
      and n)
        predecessor = p
        transfer appropriate keys to predecessor
```

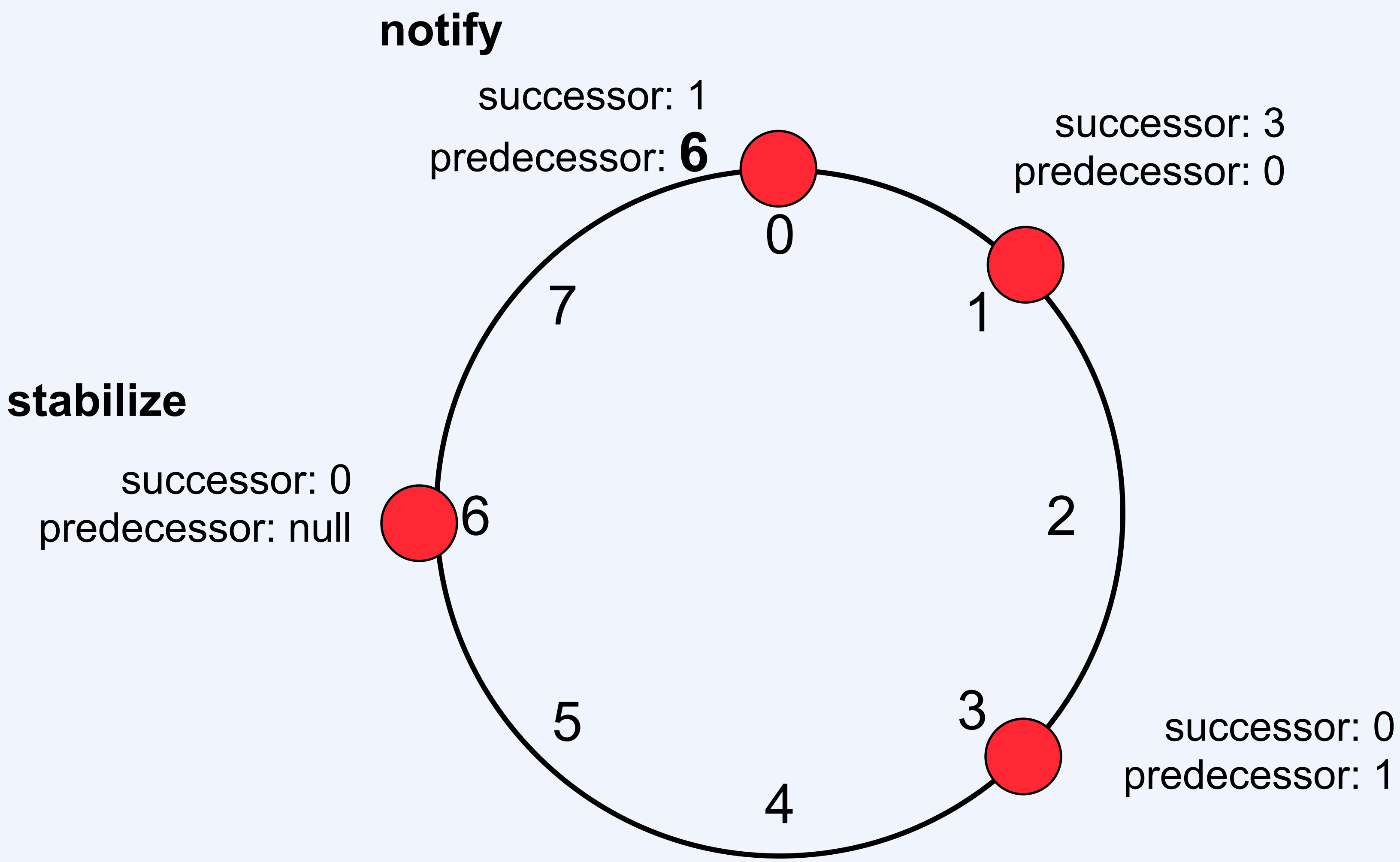
Adding Node 6 via Stabilization (1)



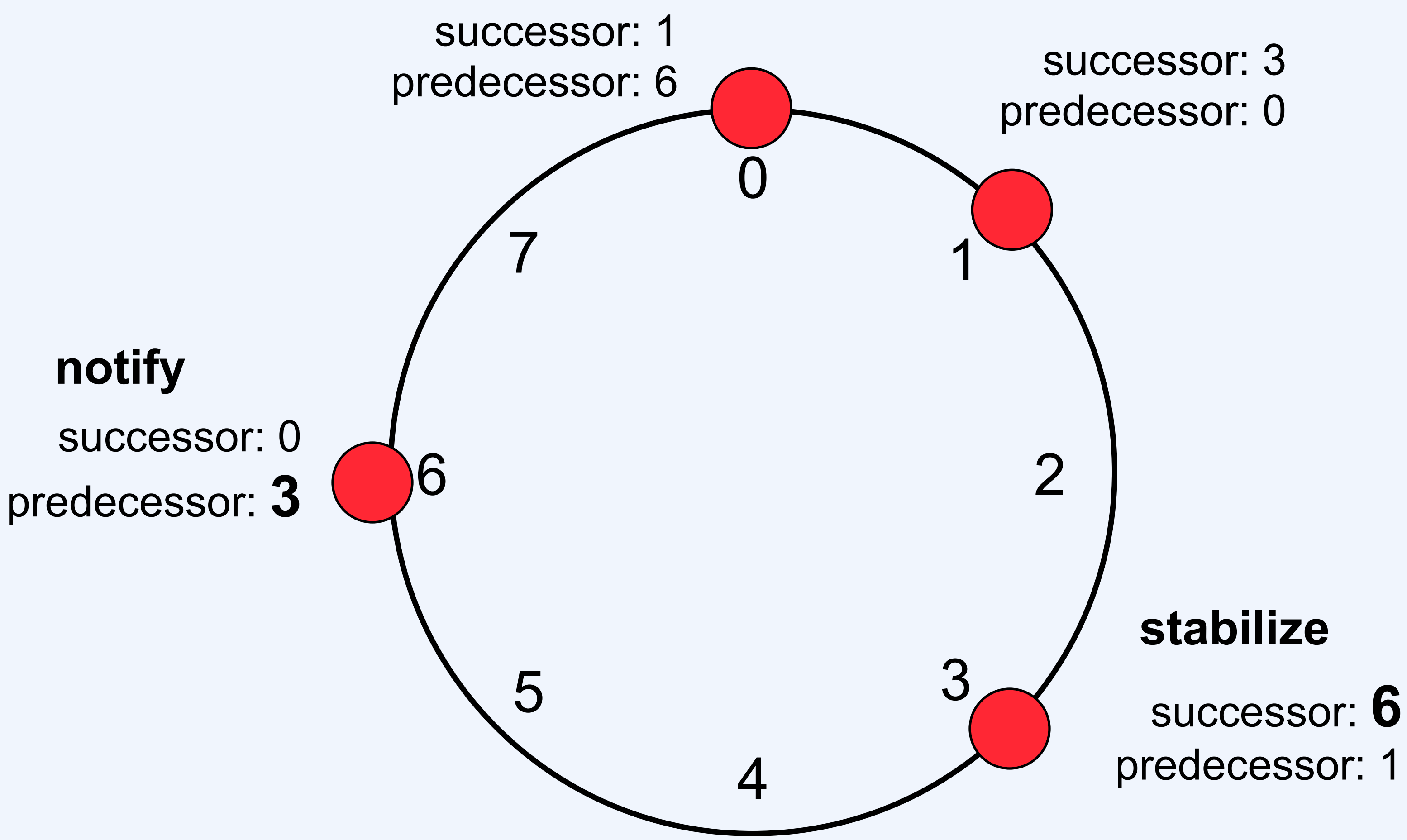
Adding Node 6 via Stabilization (2)



Adding Node 6 via Stabilization (3)



Adding Node 6 via Stabilization (4)



Transferring Objects

- When?
  - not until new node is fully linked in
  - could be a race between a search and the transfer
- What to do?
  - if search fails, search again after a delay

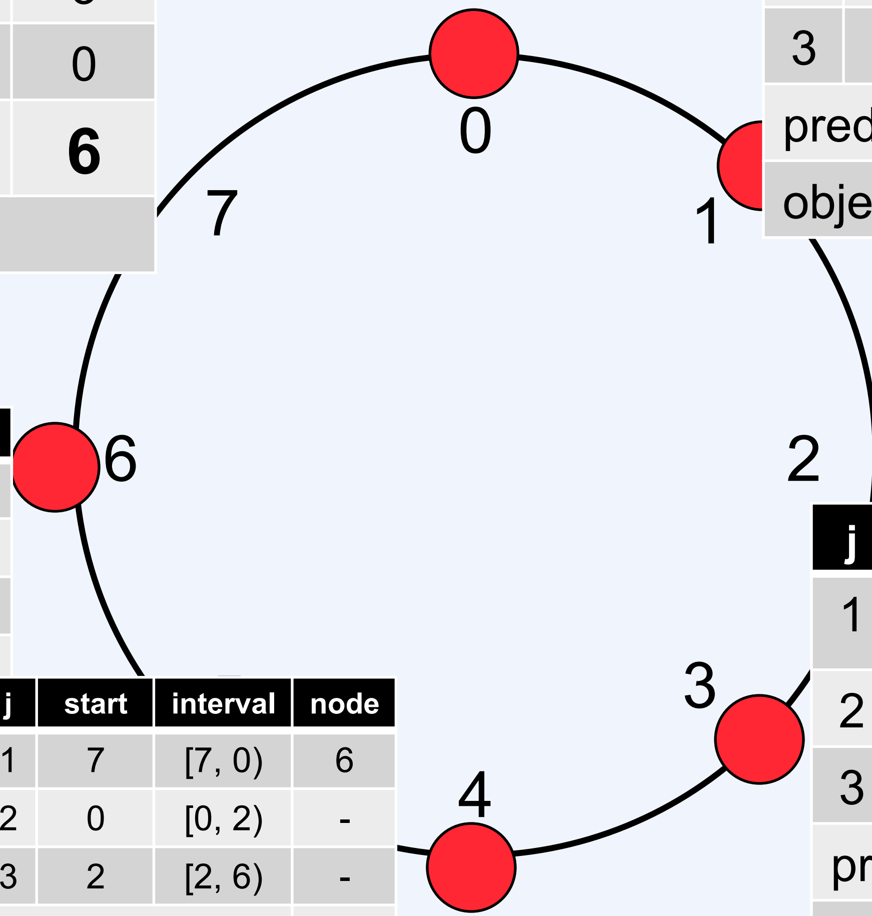
Finger Tables?

- If finger tables aren't updated, is correctness affected?

Adding Nodes 6 and 4 via Stabilization

j	start	interval	node
1	1	[1, 2)	1
2	2	[2, 4)	3
3	4	[4, 0)	0
predecessor			**6**
objects	7		

j	start	interval	node
1	2	[2,3)	3
2	3	[3,5)	3
3	5	[5, 1)	0
predecessor			0
objects	1		



j	start	interval	node
1	7	[7, 0)	0
2	0	[0, 2)	-
3	2	[2, 6)	-
predecessor			4
objects	5		

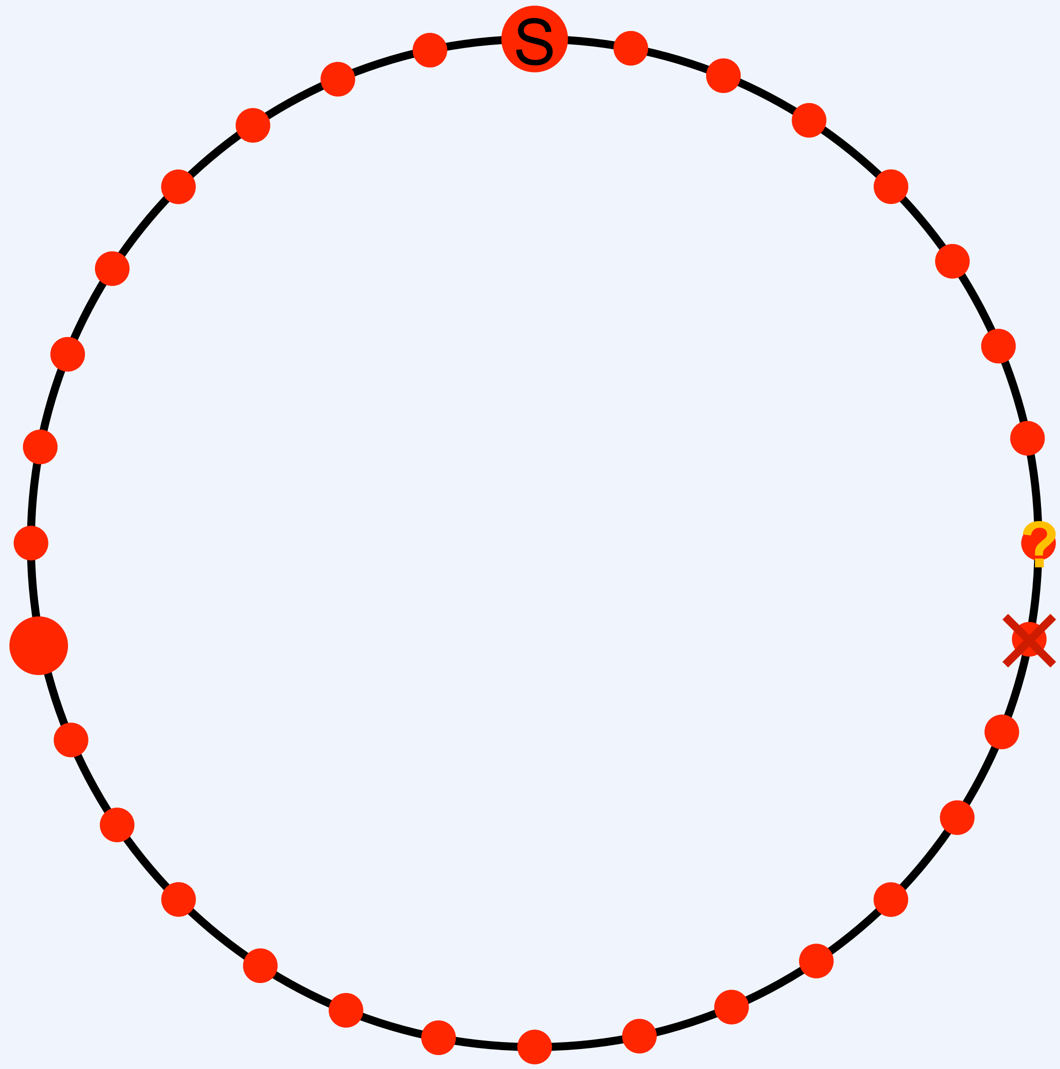
j	start	interval	node
1	7	[7, 0)	6
2	0	[0, 2)	-
3	2	[2, 6)	-
predecessor			3
objects			

j	start	interval	node
1	4	[4,5)	**4**
2	5	[5,7)	0
3	7	[7,3)	0
predecessor			1
objects	2		

Updating Finger Tables

```
// this is run periodically
n.fix_next_finger()
    //i is initialized to 1 outside of the function
    finger[i].node = find_successor(finger[i].start)
    i++
    if i > m - 1
      i = 1
```

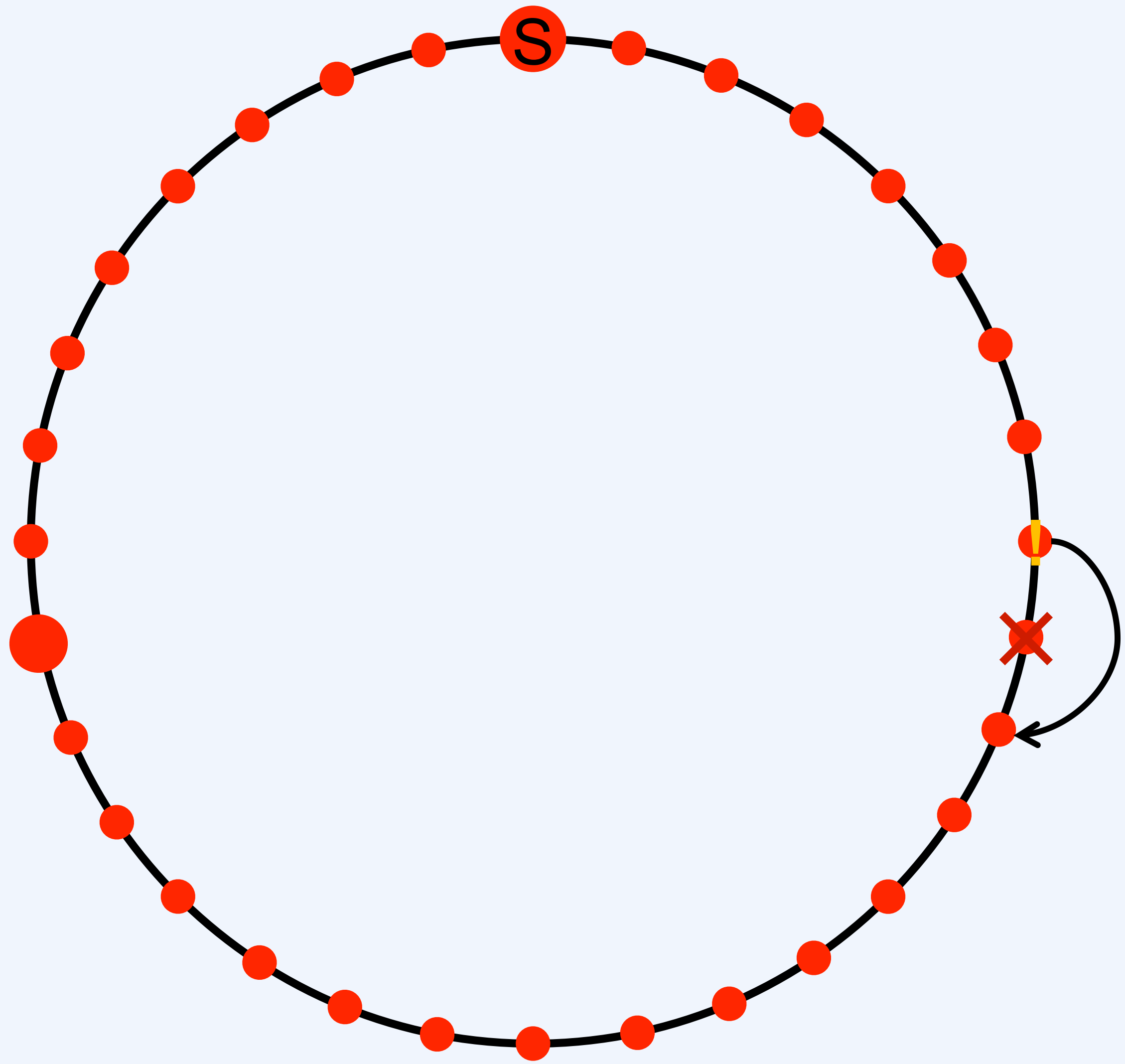
Failures



What to Do?

- Each node keeps list of r nearest successors
  - if one does not respond, switch to next
- Also replicate data at the r successors

Failures



We didn't cover

- Detailed failed recovery
- In 2012, using a formal model of Chord in Alloy, Pamela Zave showed that Chord is not correct!
  - E.g., multiple simultaneous joins can result in wrong order
  - Node leaving and then rejoining with the same id can lead to node pointing at itself
  - Has a version of the spec she claims correct
- Very subtle bugs, took over 10 years to find, over 2000 citations, "Test of Time" award
- If you are interested, take Logic for Systems, 1950-Y!

Next Class

- Tapestry, another DHT
- Chord assignment due 16^{th}, Tuesday!

- Make sure you have done:
  - Collaboration policy
  - Piazza
  - Github