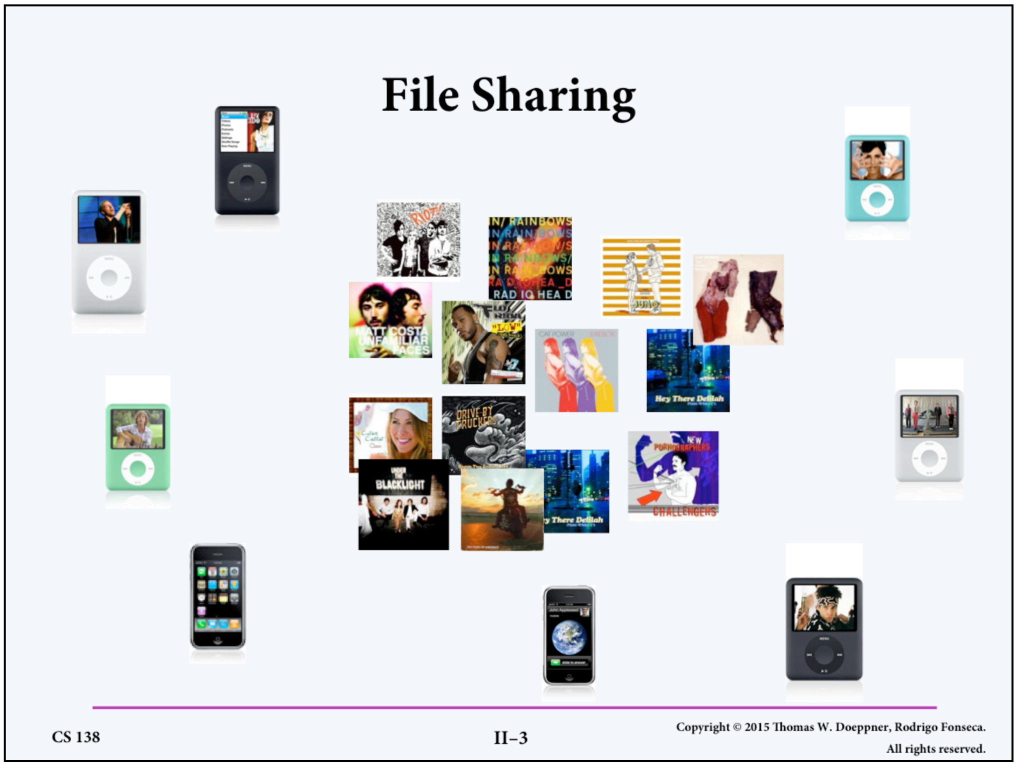
Peer to Peer I

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



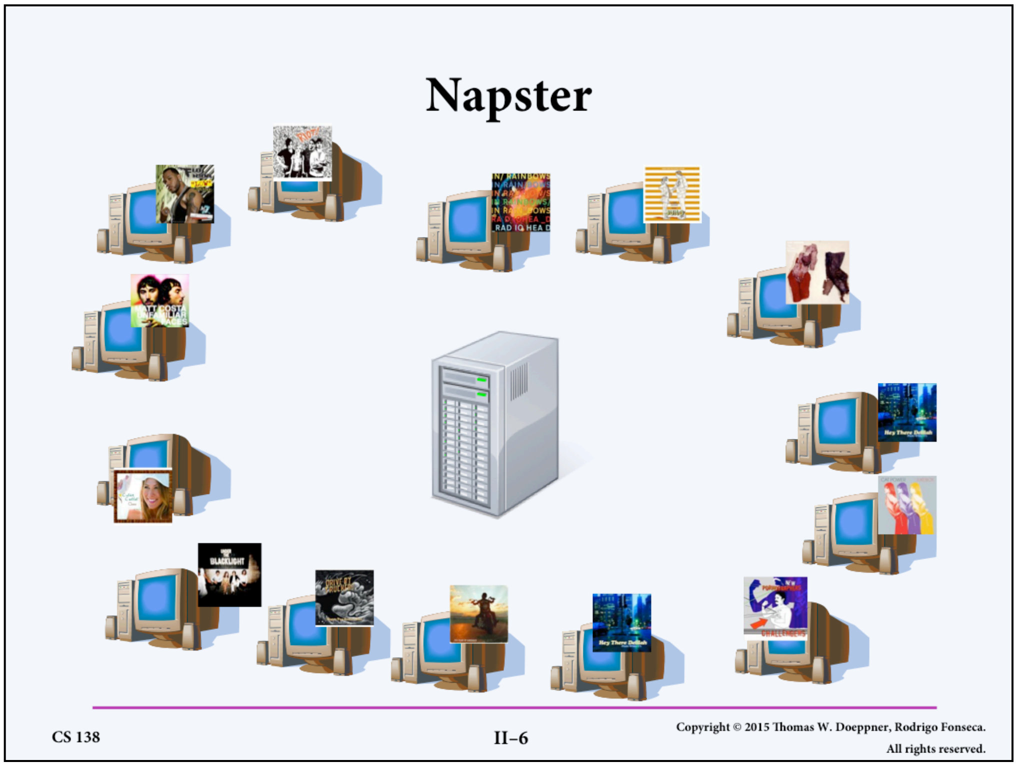
In the late 90's two trends created the conditions for a new killer application for the Internet: music sharing. The two trends were the availability of broadband Internet, and the advent of good-quality audio compression (mp3, 1:12).

Peer-to-Peer Systems

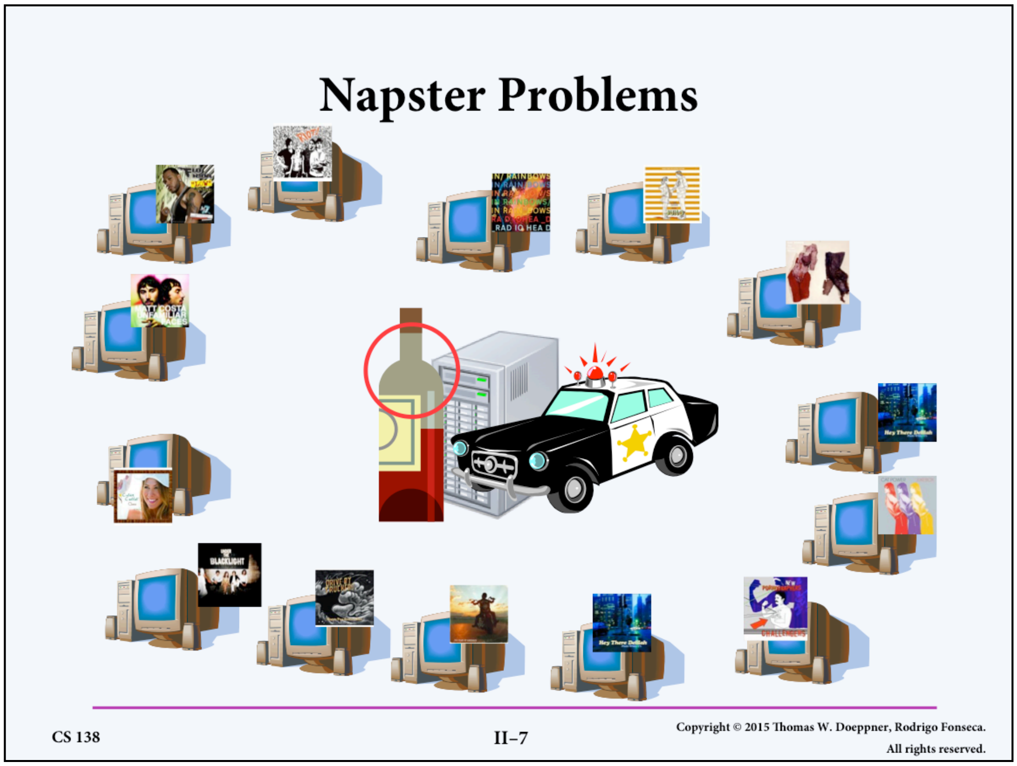
- How did it start?
  - A killer application: file distribution
  - Free music over the Internet! (*not exactly legal…*)
- Key idea: share storage, content, and bandwidth of individual users
  - Lots of them
- Big challenge: coordinate all of these users
  - In a scalable way (not NxN!)
  - With changing 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

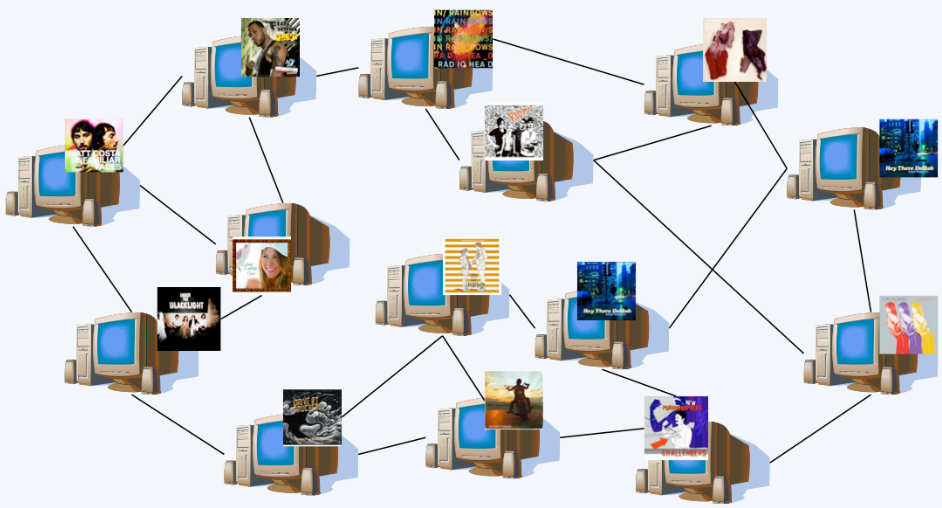


The Napster file sharing service features a central service at which providers of files register their locations and seekers of files find file locations.



A central server is clearly a bottleneck. But we can replicate it locally, as we will see. The main problem, as was discovered by the providers of the original Napster, its presence makes it easy for legal action to be taken against the service.

Gnutella



So a radically different approach appeared: a totally decentralized architecture.

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

See the Wikipedia article (http://en.wikipedia.org/wiki/Gnutella) for a few more details.

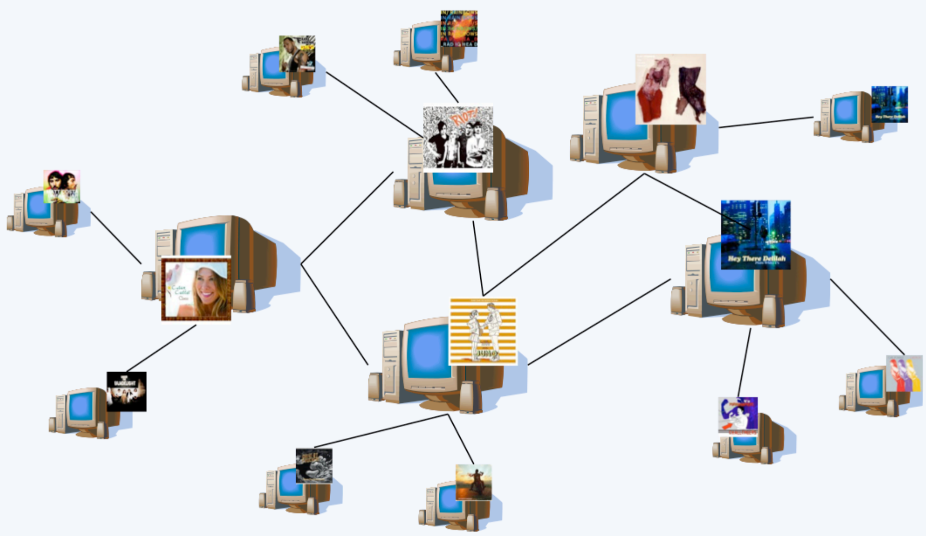
More Details

- 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



This architecture later led to Kazaa, and to Skype!

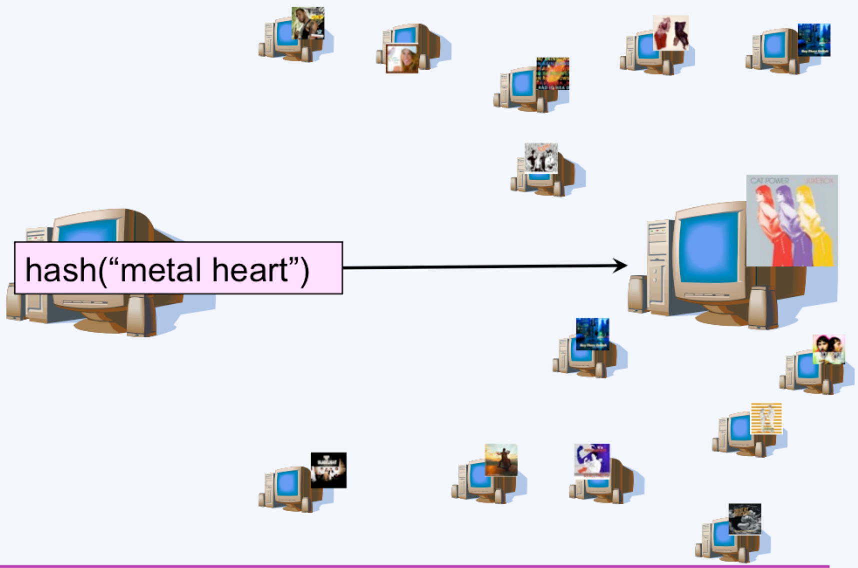
Lessons and Limitations

- 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

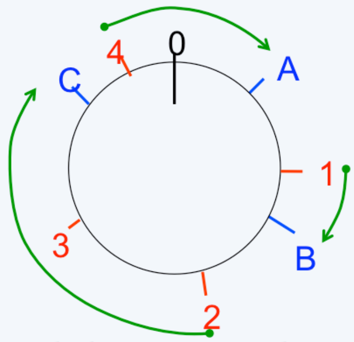
Distributed Hash Tables



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]



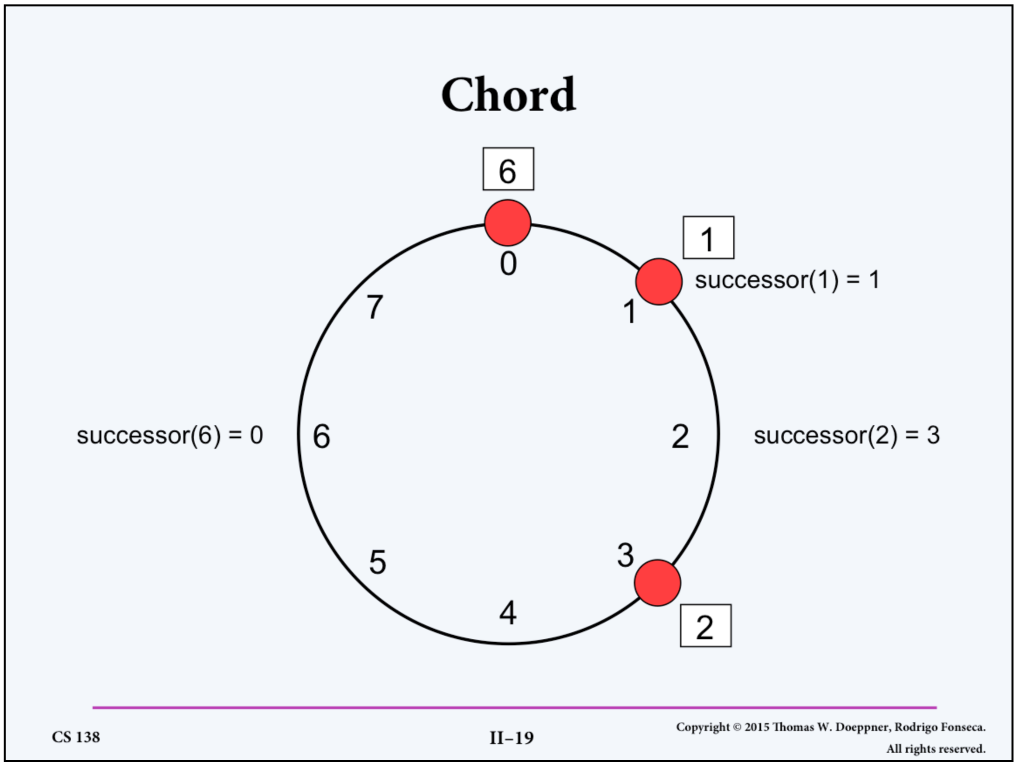
Object	Server
1	B
2	C
3	C
4	A

- 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

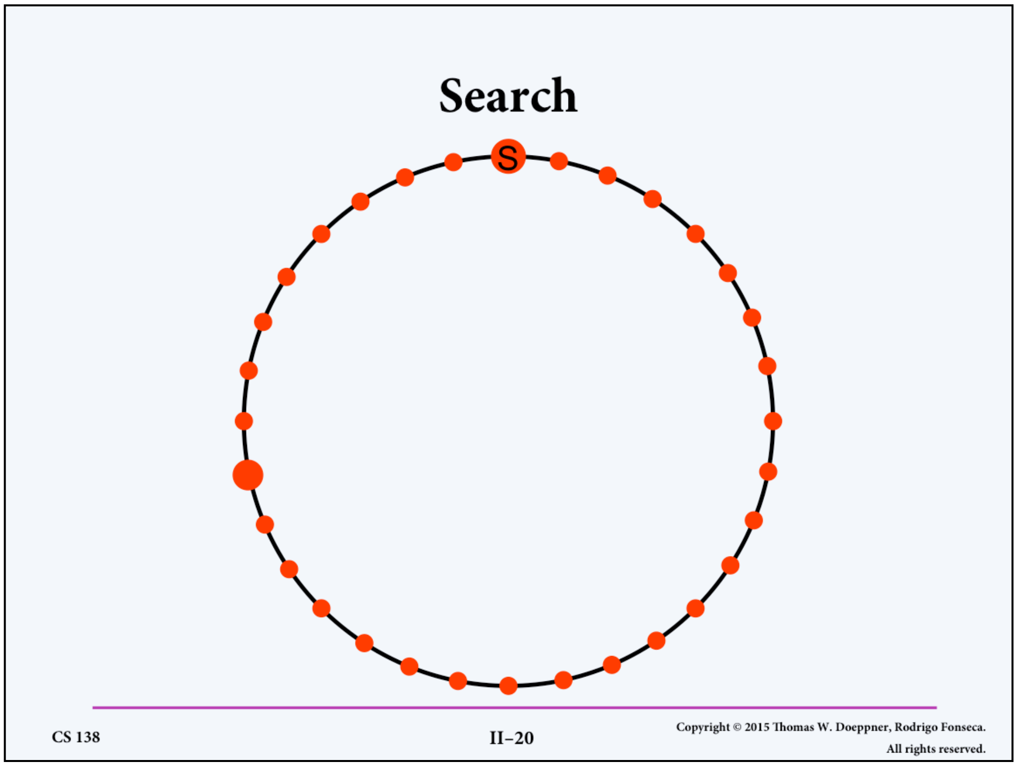
- Distributed hash tables meet overlay networks
  - hash both keys and node IP addresses into identifiers
    - m-bit identifiers, where m is large enough so that probability of collision is negligible
  - lookups resolved in O(log n) messages
  - adding or deleting a node requires $O(\log^2 n)$ messages

A paper explaining Chord can be found at http://pdos.csail.mit.edu/papers/chord:sigcomm01/chord_sigcomm.pdf. The hash function employed is SHA-1. The bounds on the number of messages are “with high probability.”



The range of the hash function is organized as a circle. The red circles represent nodes (computers) whose hashed IP addresses are 0, 1, and 3. To simplify the discussion, we'll ignore the hash function and refer to 0, 1, and 3 as being the nodes themselves rather than their hashed IP addresses. Similarly, we'll refer to keys 0, 1, … 7 rather than saying that we have keys whose values hash to [0, 7]. Given this simplification, if *i* is a key, then *successor(i)* is the node where the key (and associated value) is stored. Things are organized so that key *i* is assigned to the lowest numbered node greater than or equal to *i* (modulo 2^m). Thus *successor(i)* is the number of that node.

If we store with each node *i* the address of *successor(i)*, then, starting from any node, we can find the node containing any particular key (or definitively say the key is not present). Of course, doing this requires O(n) steps.

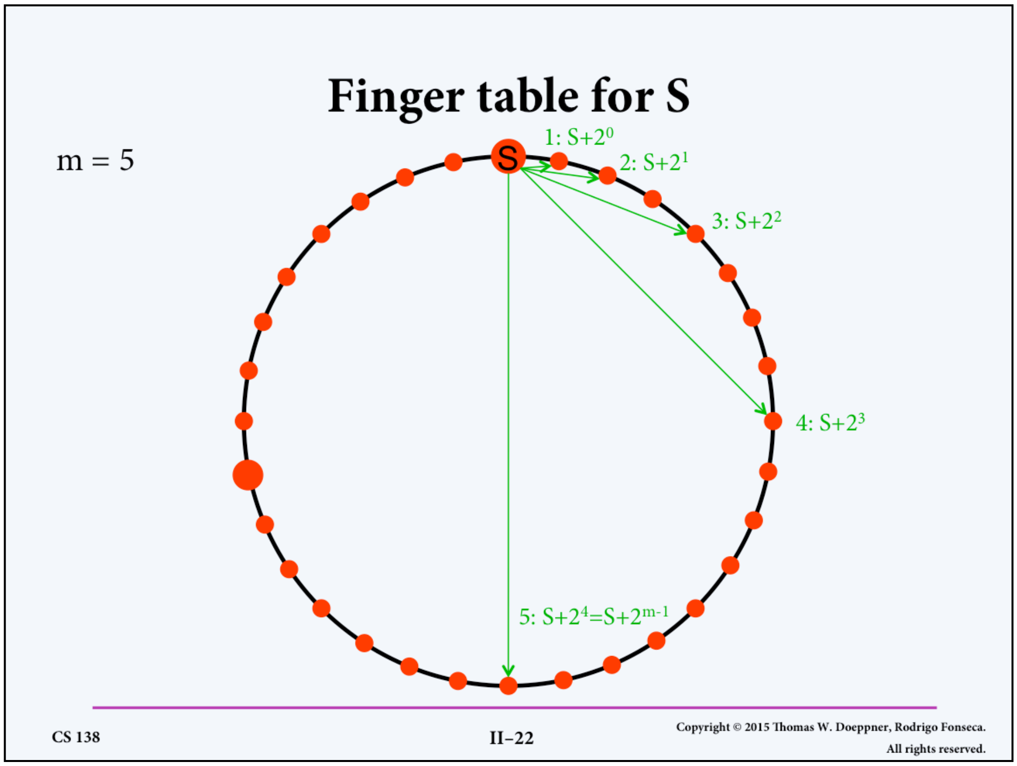


Search requires a number of messages that is linear in the number of nodes — not good.

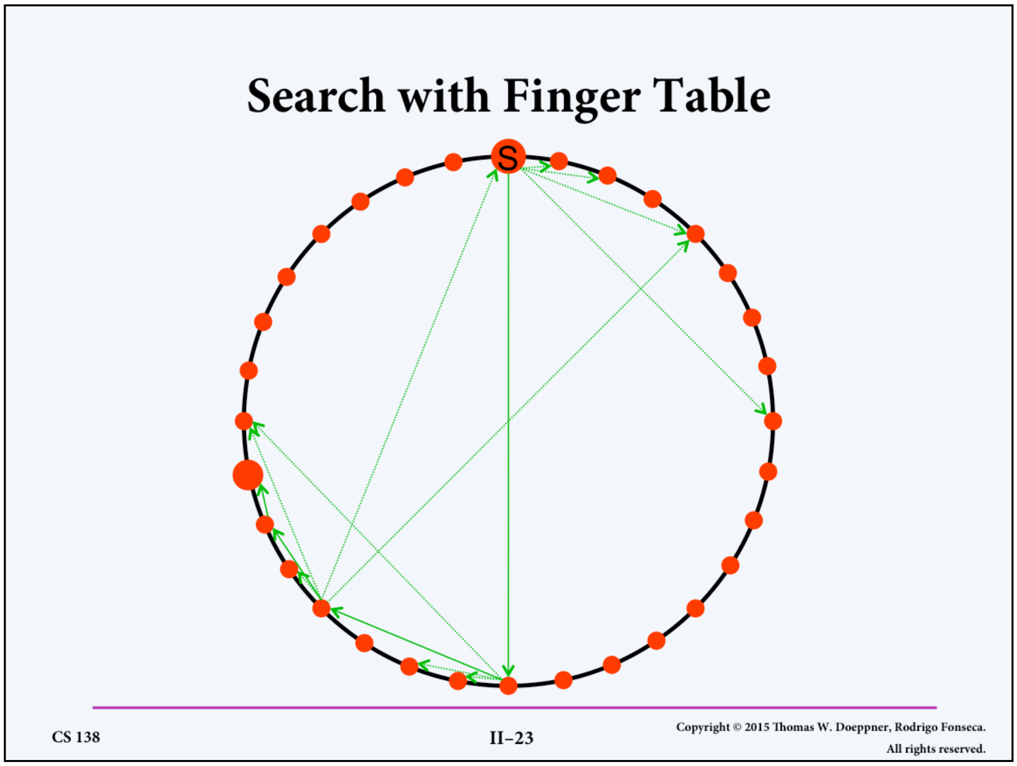
Speeding It Up

- 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)

Note that the numbering of finger-table entries starts with 1 (not 0)!



With the addition of the finger table, search requires log(N) messages, where N is the number of nodes.



With the addition of the finger table, search requires log(N) messages, where N is the number of nodes.

1. While not in id's predecessor
2. Find last finger f < id
3. Recurse

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
}
```


In this pseudo code (taken from the aforementioned paper), *m.foo(x)* means to place a remote procedure call to node *m*, executing procedure *foo* with argument *x*. *m.x* means to place a remote procedure call to node *m*, retrieving the value of variable *x*.

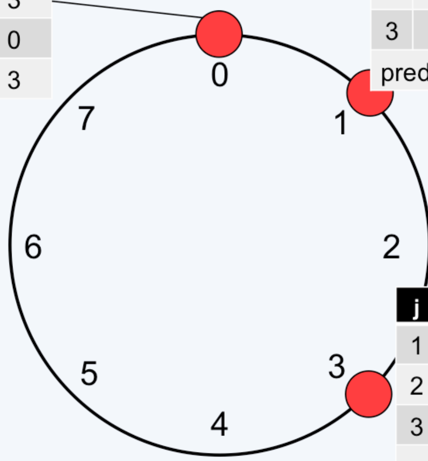
The while loop in *find_predecessor* continues until *n1* is the highest-numbered node less than *id* (modulo 2^m), which will be the case when *id* is between *n1* and the node that comes after it (*n1.successor*).

The for loop in *closest_preceding_finger* finds the highest-numbered row in the finger table that refers to a node that comes before *id*.

Chord

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

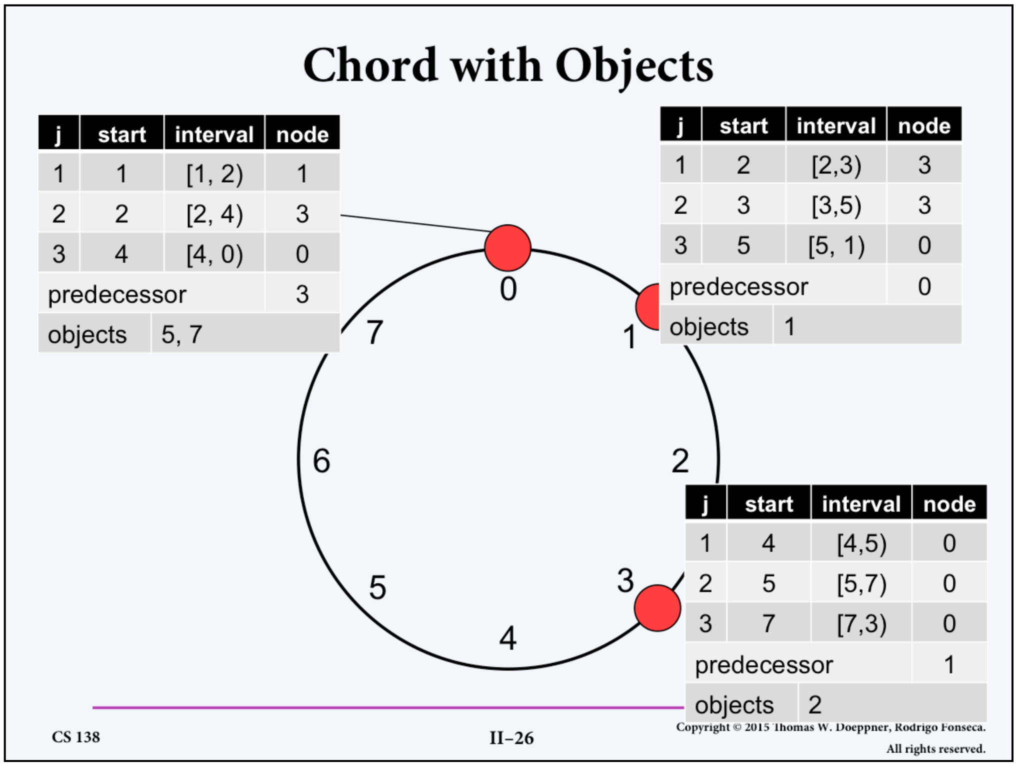
j	start	interval	node
1	2	[2,3)	3
2	3	[3,5)	3
3	5	[5, 1)	0
predecessor			0



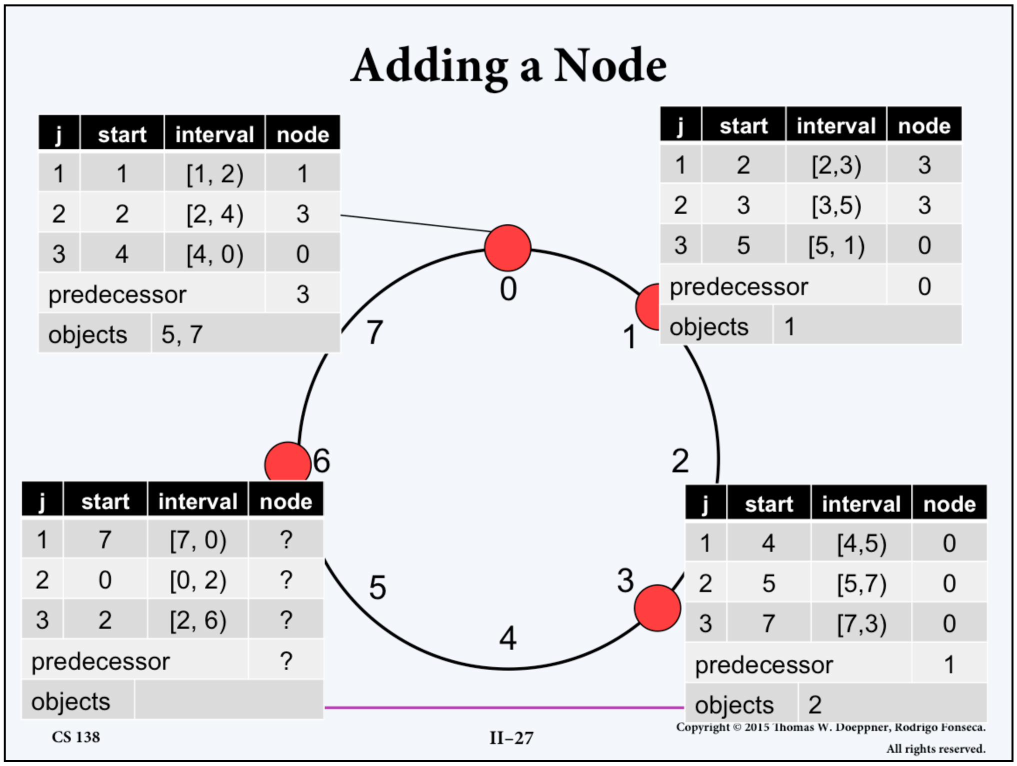
j	start	interval	node
1	4	[4,5)	0
2	5	[5,7)	0
3	7	[7,3)	0
predecessor			1

Here are the finger tables for our example. Each row represents the portion of the key space covered by the row ("finger").
Start: first key in the sequence of keys covered;
Interval: entire sequence of keys covered;
Node: node number of the first node whose number is greater than or equal to the value in the start column.

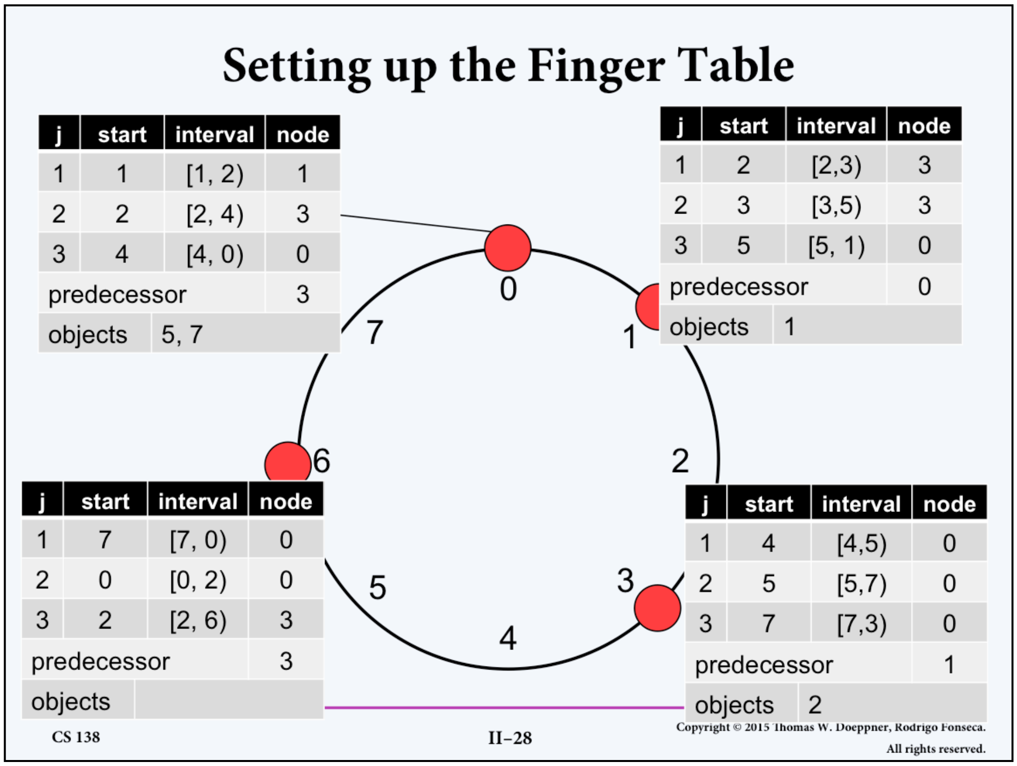
Note that the node listed in the first row of each table is the next node in the ring. This is somewhat confusingly called the "successor node" (if x is a hashed key, then successor(x) might be x if node x exists; but successor(x) if x is a node is always the next node in the ring). It's convenient to also list the predecessor node for each node; it's given at the end of each table.



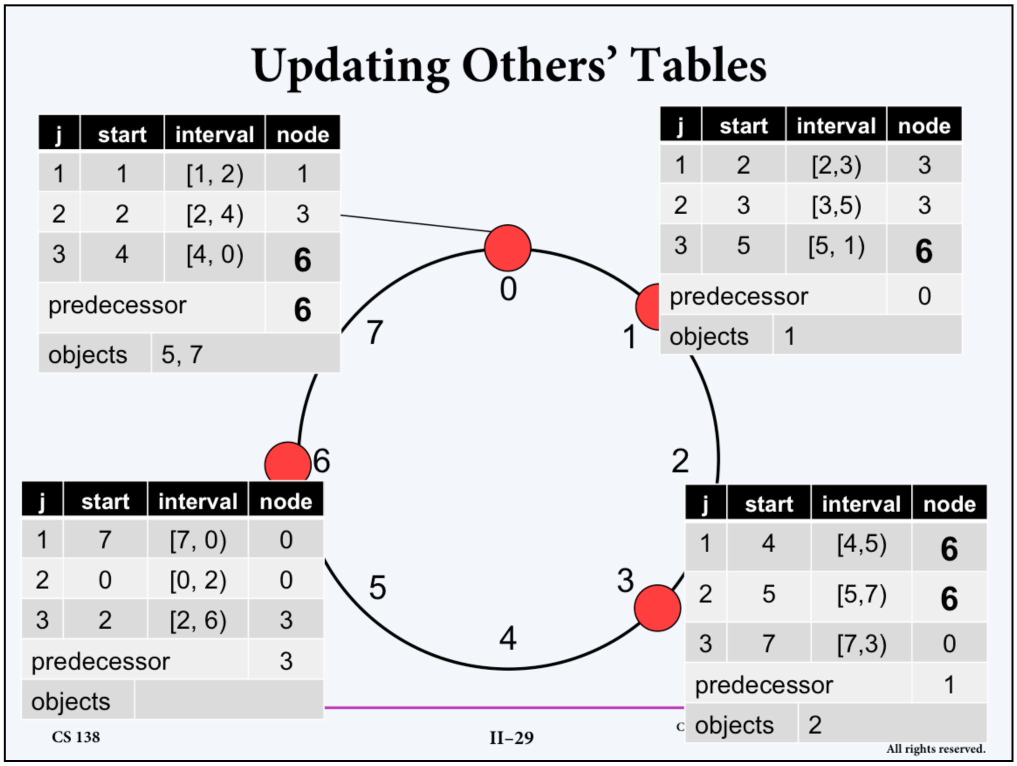
Here we include in the finger tables the objects stored at each node.



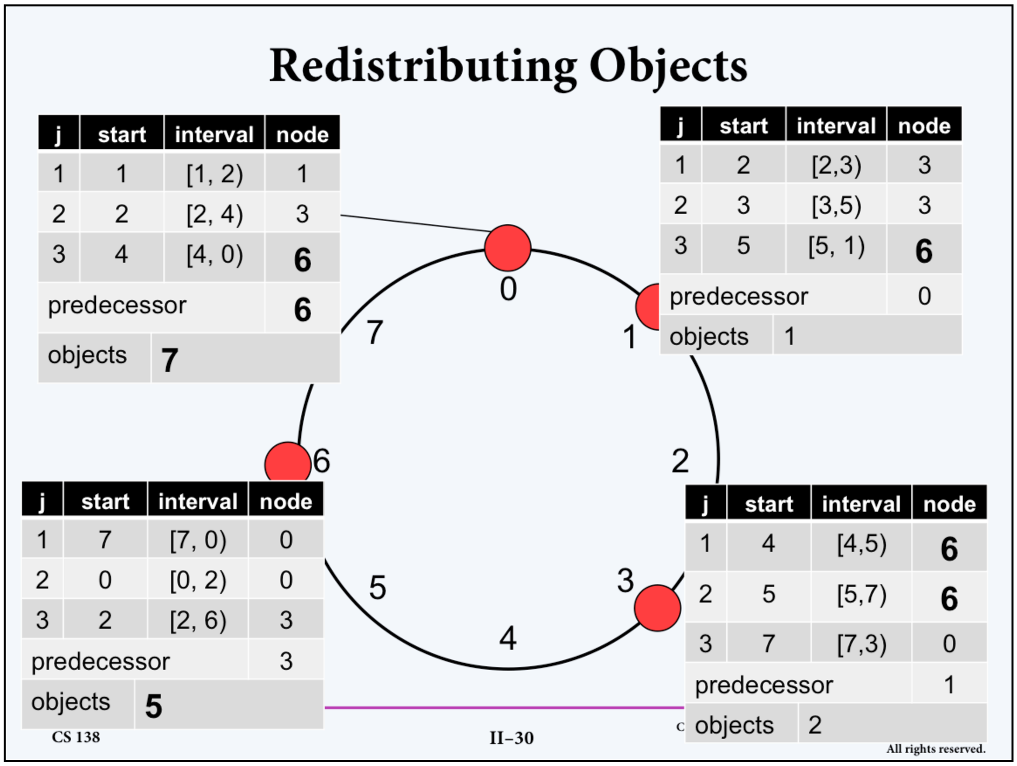
Adding a node requires both adjusting all the finger tables to accommodate the new node and moving the objects that should be stored at that node.



First we set up the finger table of the new node.



Next we modify the finger tables of previously existing nodes to accommodate the new node.



Finally we redistribute the objects.

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

Now for an algorithm. The first step is to set up the new node's finger table. This step requires m·log(N) messages.

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).

Note that if there are no nodes within the interval covered by a row of the finger table we are constructing for a new node, then the node entry for the next row, i.e., the first node greater than or equal to the start of the interval, is the same as that of the current row.

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
```


The improved version requires $\log^2(N)$ messages. (Can you figure out why?)

Adding a Node

2) **Update others' finger tables**

```
for i = 1 to m
      // find last node p whose ith 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)
```


The code shown here requires O($\log^3$(N)) messages. An algorithm requiring O($\log^2$(N)) messages is possible, but we cover an entirely different approach instead.

Adding a Node

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

Note that the only objects that need to be moved are stored in 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?

Because of the difficult answers to these questions, we drop the approach we just looked at and try something different.

Invariants

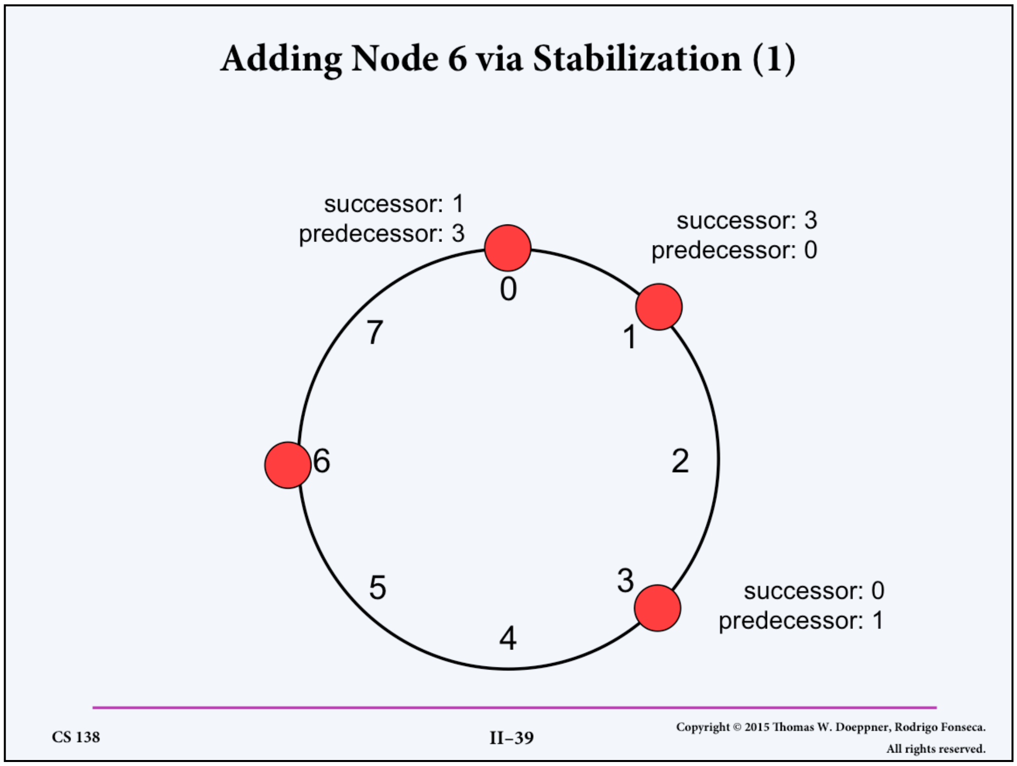
- Each node's successor link is correct
- For every key k, successor(k) is responsible for k

Stabilization

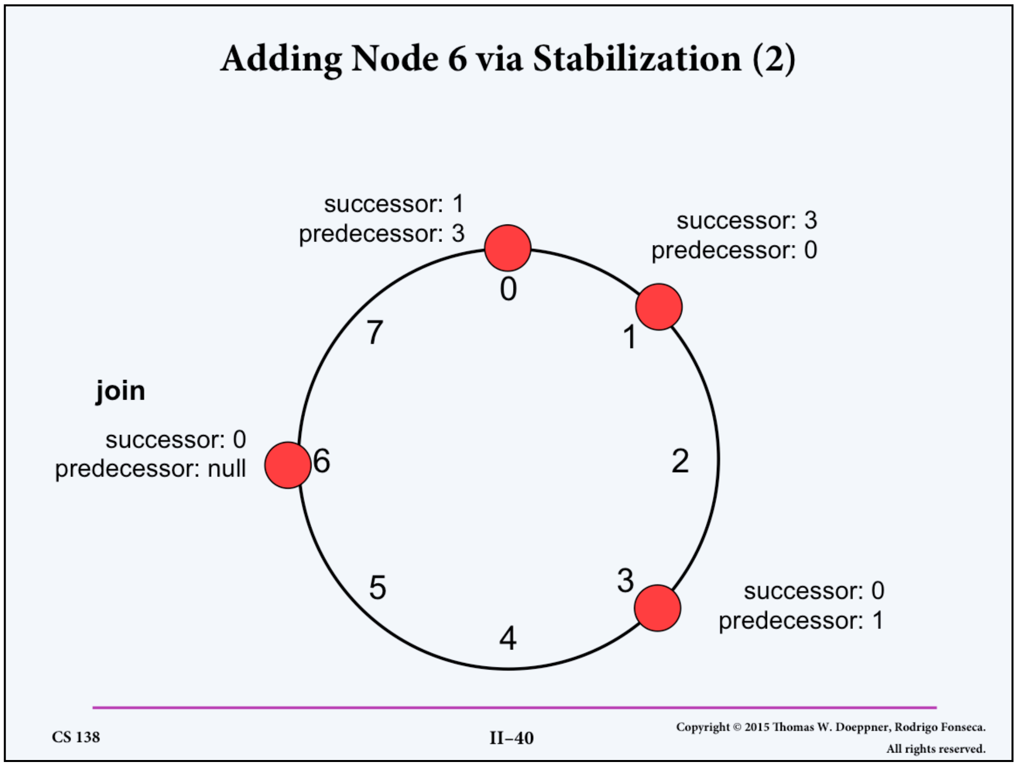
```
n.join(p) //p is some node you know
    predecessor = null
    successor = p.find_successor(n)

// 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
```

CS 138 II–38

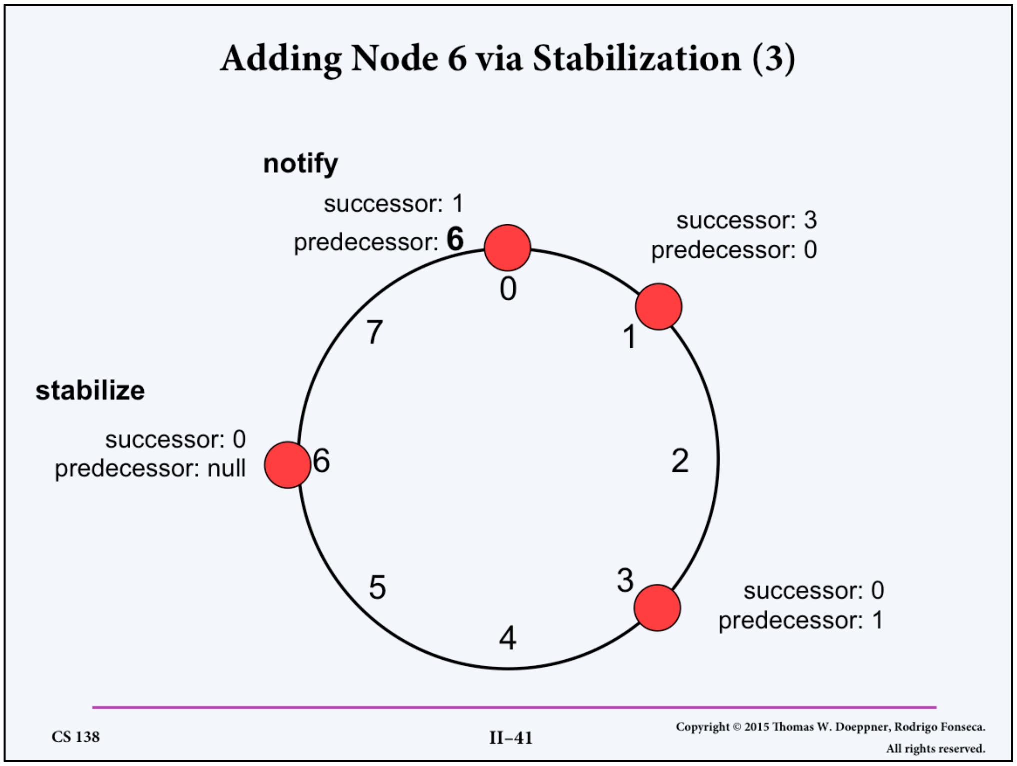
For the moment, we ignore finger tables. This code provides the minimum necessary functionality so that the first invariant is preserved. A real implementation could be more aggressive. To make sense of this code, see the next few slides.



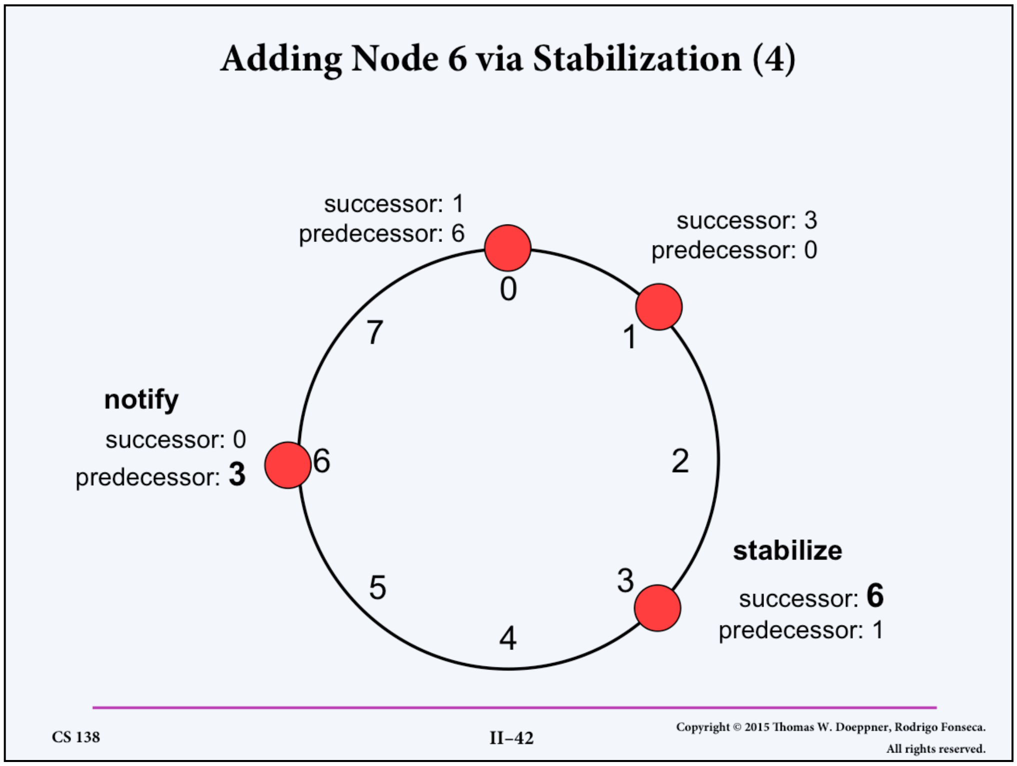
Focusing just on the successor and predecessor links, we follow what happens when node 6 is added to the ring.



Node 6 calls join to add itself. At this point, no other node knows of its existence.



Node 6 now calls stabilize. Stabilize itself does nothing, but it calls notify on node 0 (6's successor). Node 0 sets its predecessor to be 6.



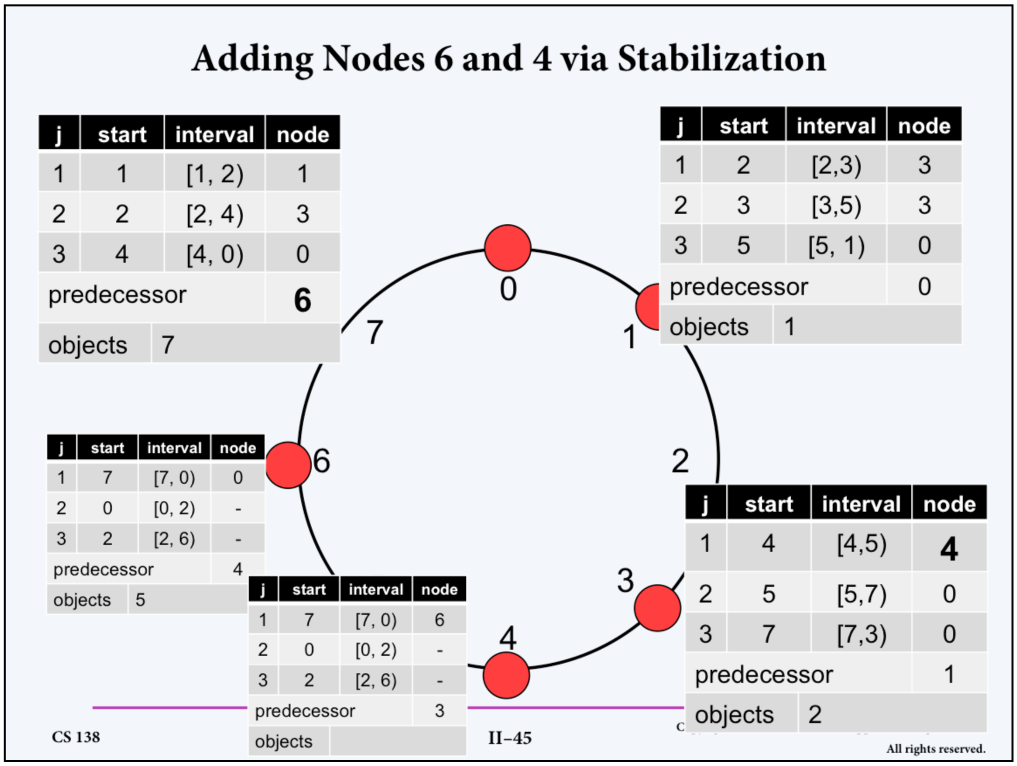
Node 3 now calls stabilize and discovers that its successor's predecessor is not itself. So it sets its successor to 6 (its successor's predecessor) and calls notify on node 6. Node 6 now sets its predecessor to be 3 — it's now fully linked in.

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?



This slide shows the effect of adding nodes 6 and 4 in our earlier example by merely executing the stabilization code.

Note that the finger table is used to find the predecessor of the object’s successor.

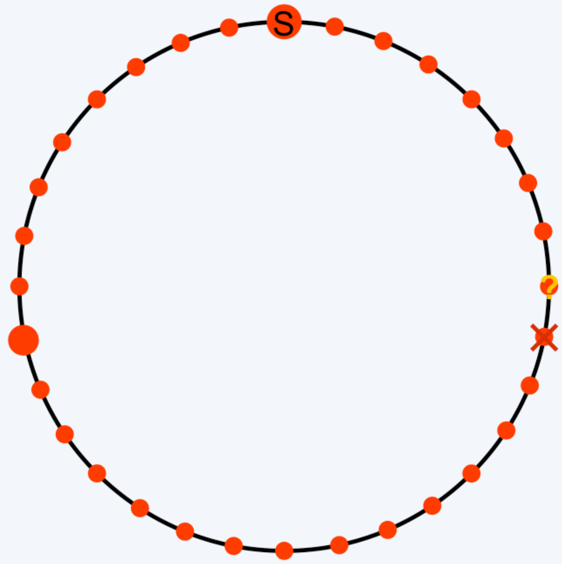
Thus searches initiated at nodes 0 and 1 for object 5 will identify node 3 as the predecessor.

It’s then necessary to follow successor links until 5 is located at its successor, node 6.

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
```

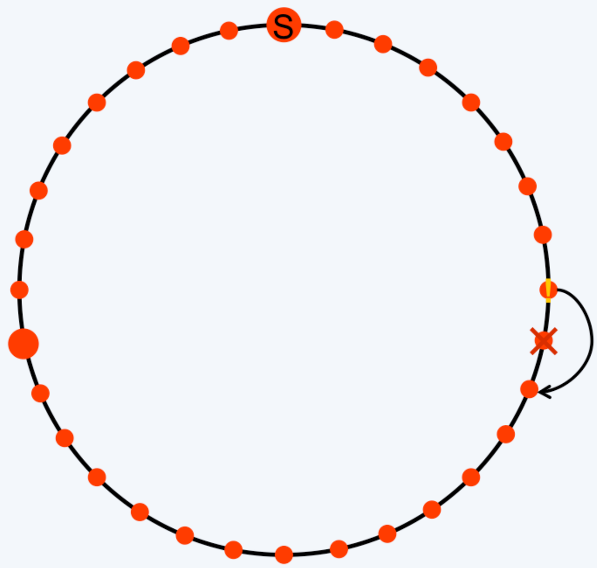
Failures



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!

This can be seen here: http://www2.research.att.com/~pamela/chord.html

Next Class

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

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