Peer-to-Peer Networks

Chapter 3: Networks, Searching and Distributed Hash Tables

Chapter Outline

- Searching and addressing
  - Structured and unstructured networks
- Distributed Hash Tables (DHT)
  - What are DHT?
  - How do they work?
  - What are they good for?
  - Examples: Chord, CAN, Plaxton/Pastry/Tapestry
- Networks and graphs
  - Graph theory meets networking
  - Different types of graphs and their properties

Searching and Addressing

- Two basic ways to find objects:

1. Search for them
2. Address them using their unique name

- Both have pros and cons (see below)
- File sharing built on searching, some start addressing objects
- Difference between searching and addressing is a very fundamental difference
  - Determines how network is constructed
  - Determines how objects are placed
  - "Determines" efficiency of object location
- Let's compare searching and addressing

Searching vs. Addressing

- Recall: Name -> ID -> Reference

- *Content Addressing* maps the „content“ on a reference
  - F(resource) := Reference (resource may be reg. information)
  - Consider F(.) globally be known:
    - Anybody can directly derive (and access) reference
    - Direct addressing of content (if resource is known…)
    - Location depends on F(.) and resource only

Is this always useful?

- Searching may find
  - Names, IDs, References, *Metadata, Content*…
- But: deterministic access is big advantage in large, dist. systems!

Addressing vs. Searching

- "Addressing" networks find objects by addressing them with their unique name (cf. URLs in Web)
- "Searching" networks find objects by searching with keywords that match objects' description (cf. Google)

Addressing

- Pros:
  - Each object uniquely identifiable
  - Object location can be made efficient
- Cons:
  - Need to know unique name
  - Need to maintain structure required for addressing

Searching

- Pros:
  - No need to know unique names
    - More user friendly
- Cons:
  - Hard to make efficient
    - Can solve with money, see Google
  - Need to compare actual objects to know if they are same

Searching, Addressing, and P2P

- We can distinguish two main P2P network types

- Unstructured networks/systems (Last chapter)
  - Cause the need for searching (provide the possibility to search!)
  - Unstructured does NOT mean complete lack of structure
    - Network has graph structure, e.g., scale-free, power-law, hierachy,…
  - Network has structure, but peers are free to join anywhere, choose neighbors freely, objects are stored anywhere

- Structured networks/systems
  - Allow for addressing, deterministic routing
  - Network structure determines where peers belong in the network and where objects are stored
  - How can we build structured networks?

Distributed Hash Tables

- What are DHT?
- How do they work?
- What are they good for?
- Examples:
  - Chord
  - CAN
  - Tapestry (Plaxton-Mesh/Pastry)

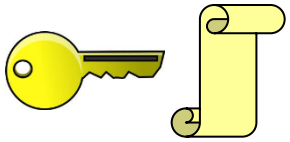
DHT: Motivation

- Why do we need DHTs?
- Searching in unstructured P2P networks is not efficient
  - Either centralized system with all its problems
  - Or decentralized system with all its problems
  - Hybrid systems cannot guarantee discovery either
- Actual file transfer process in P2P network is scalable
  - File transfers directly between peers
- Searching does not scale in same way
- Original motivation for DHTs:

  More efficient searching and object location in P2P networks
- Put another way: Use addressing instead of searching

Recall: Hash Tables

- Hash tables are a well-known data structure
- Hash tables allow insertions, deletions, and lookups in O(1)



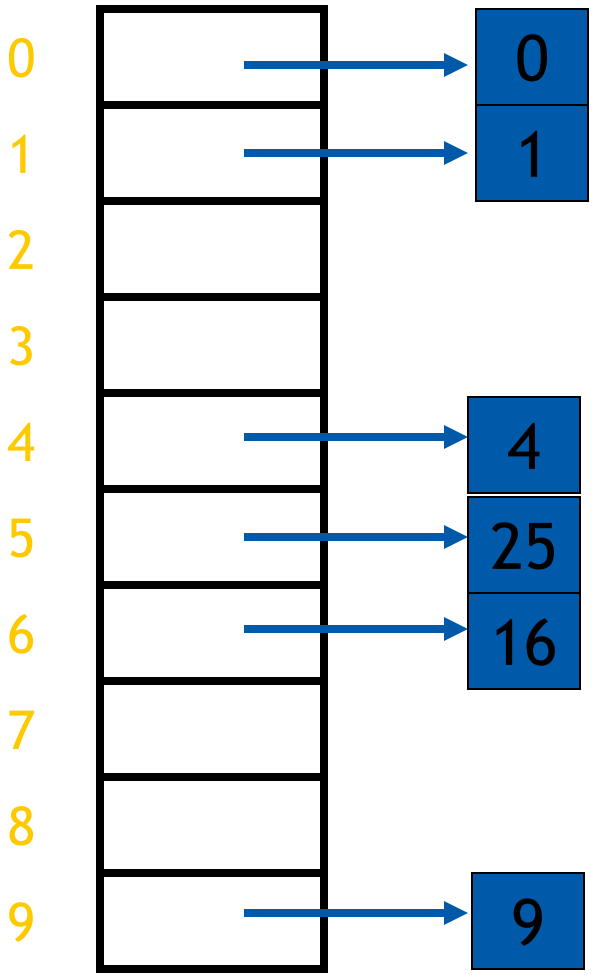
- Hash table is a fixed-size array
  - Elements of array also called *hash buckets*

- *Hash function* maps keys to elements in the array

- Properties of good hash functions:
  - Fast to compute
  - Good distribution of keys into hash table
  - Example: SHA-1 algorithm

www.phdcomics.com

Hash Tables: Example



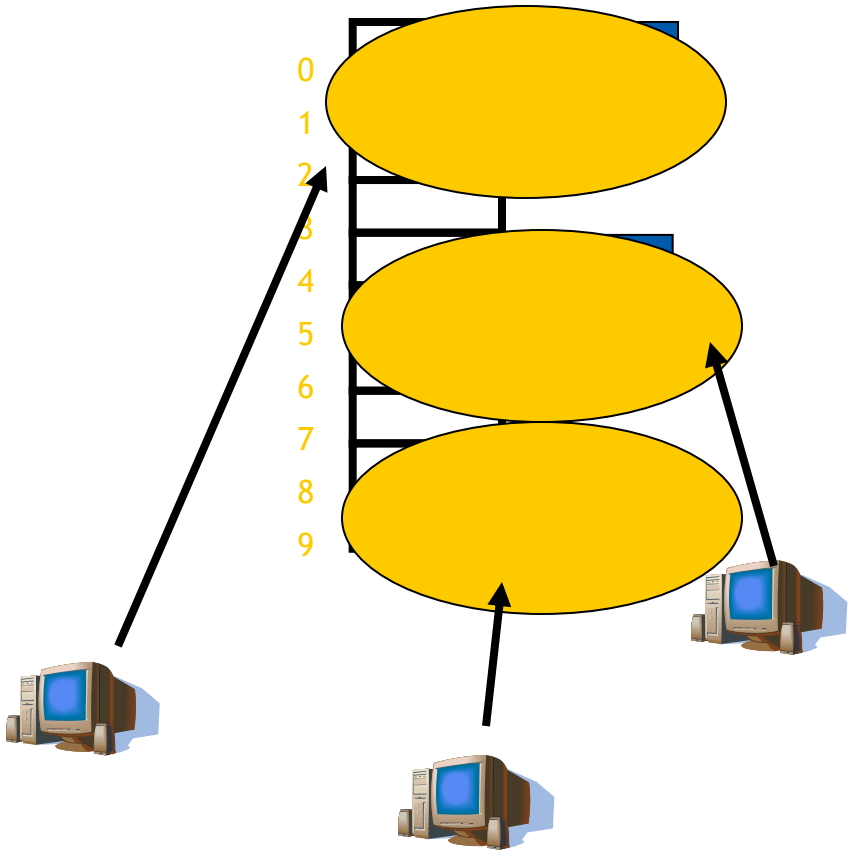
- Hash function:

  hash(x) = x mod 10

- Insert numbers 0, 1, 4, 9, 16, and 25
- Easy to find if a given key is present in the table

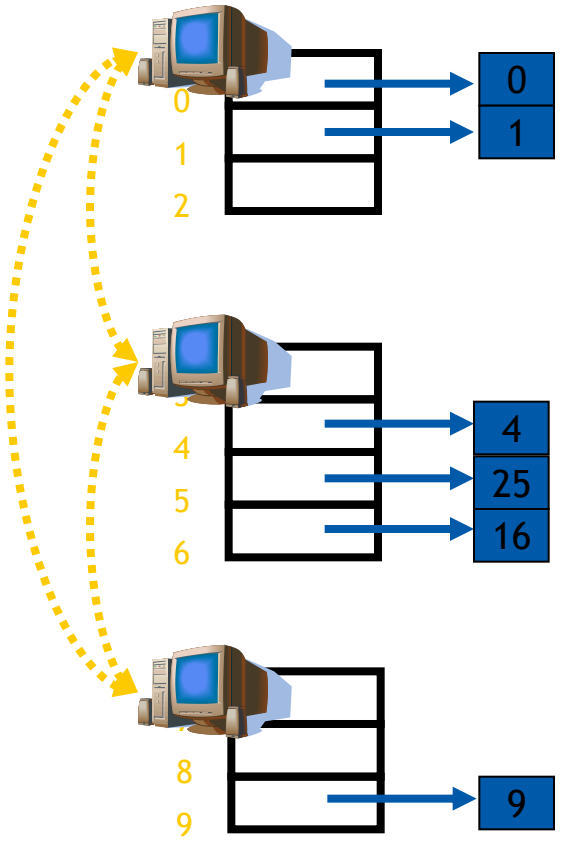
Distributed Hash Table: Idea

- Hash tables are fast for lookups
- Idea: Distribute hash buckets to peers
- Result is Distributed Hash Table (DHT)
- Need efficient mechanism for finding which peer is responsible for which bucket and routing between them



DHT: Principle

- In a DHT, each node is responsible for one or more hash buckets
  - As nodes join and leave, the responsibilities change
- Nodes communicate among themselves to find the responsible node
  - Scalable communications make DHTs efficient
- DHTs support all the normal hash table operations



Summary of DHT Principles

- Hash buckets distributed over nodes
- Nodes form an overlay network
  - Route messages in overlay to find responsible node
- Routing scheme in the overlay network is the difference between different DHTs
- DHT behavior and usage:
  - Node knows “object” name and wants to find it
    - Unique and known object names assumed
  - Node routes a message in overlay to the responsible node
  - Responsible node replies with “object”
    - Semantics of “object” are application defined

DHT Examples

- In the following look at some example DHTs
  - Chord
  - CAN
  - Tapestry
- Several others exist too
  - Pastry, Plaxton, Kademlia, Koorde, Symphony, P-Grid, CARP, …
- All DHTs provide the same abstraction:
  - DHT stores key-value pairs
  - When given a key, DHT can retrieve/store the value
  - No semantics associated with key or value
- Routing in overlay is the main difference

Chord

- Chord was developed at MIT
- Originally published in 2001 at Sigcomm conference

- Chord's overlay routing principle quite easy to understand
  - Paper has mathematical proofs of correctness and performance

- Many projects at MIT around Chord
  - CFS storage system
  - Ivy storage system
  - Plus many others…

Chord: Basics

- Chord uses SHA-1 hash function
  - Results in a 160-bit object/node identifier
  - Same hash function for objects and nodes
- Node ID hashed from IP address
- Object ID hashed from object name
  - Object names somehow assumed to be known by everyone

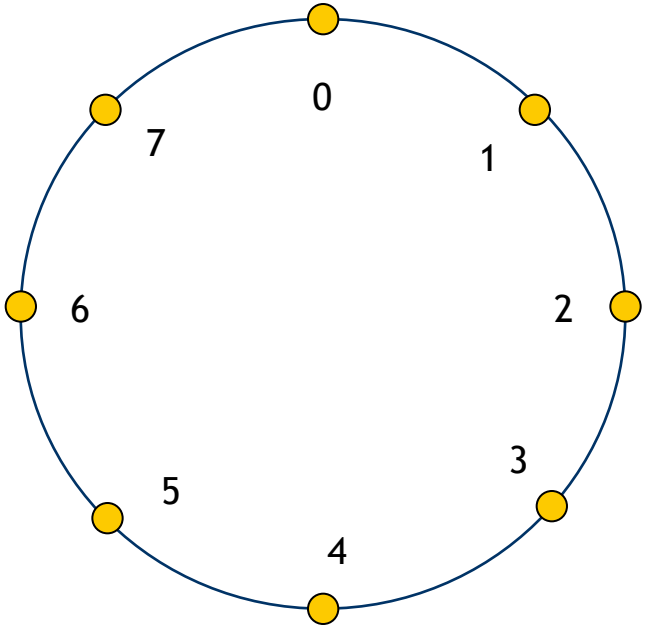
- SHA-1 gives a 160-bit identifier space
- Organized in a ring which wraps around
  - Overlay is often called “Chord ring” or “Chord circle”
  - Nodes keep track of predecessor and successor
  - ***Node registers objects on the namespace between predecessor and itself***

Chord: Examples

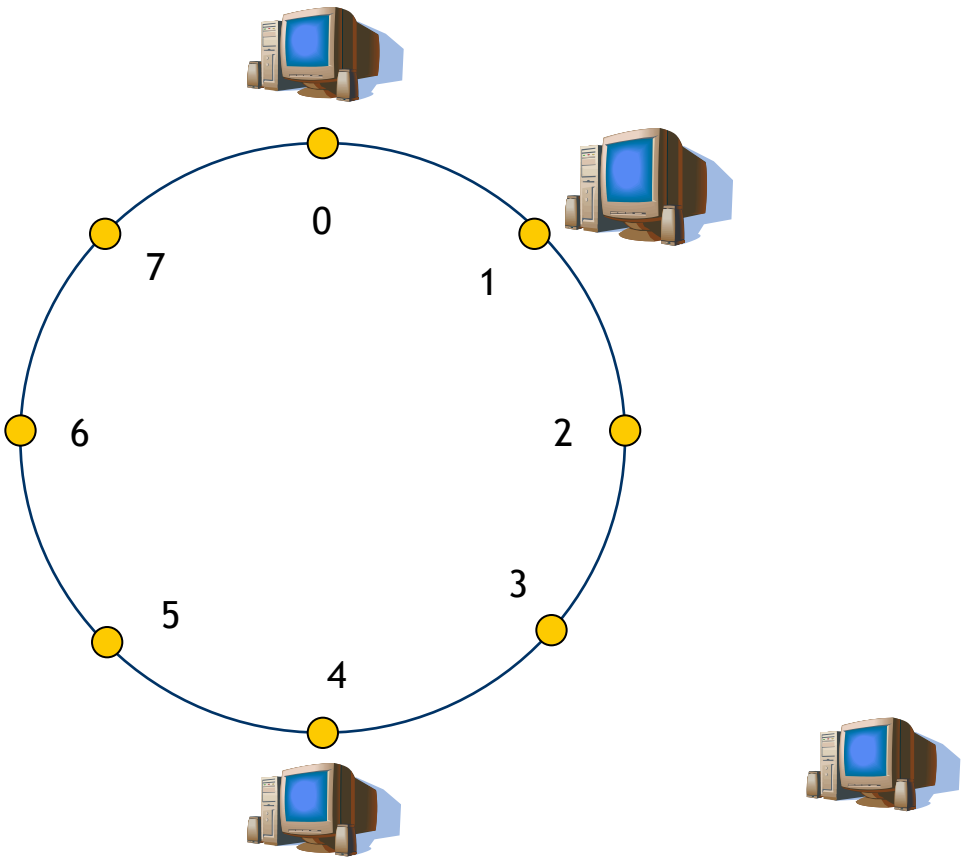
- Below examples for:
  - How to join the Chord ring
  - How to store and retrieve values

Joining: Step-By-Step Example

- Setup: Existing network with nodes on 0, 1 and 4
- Note: Protocol messages simply examples
- Many different ways to implement Chord
  - Here only conceptual example
  - Covers all important aspects



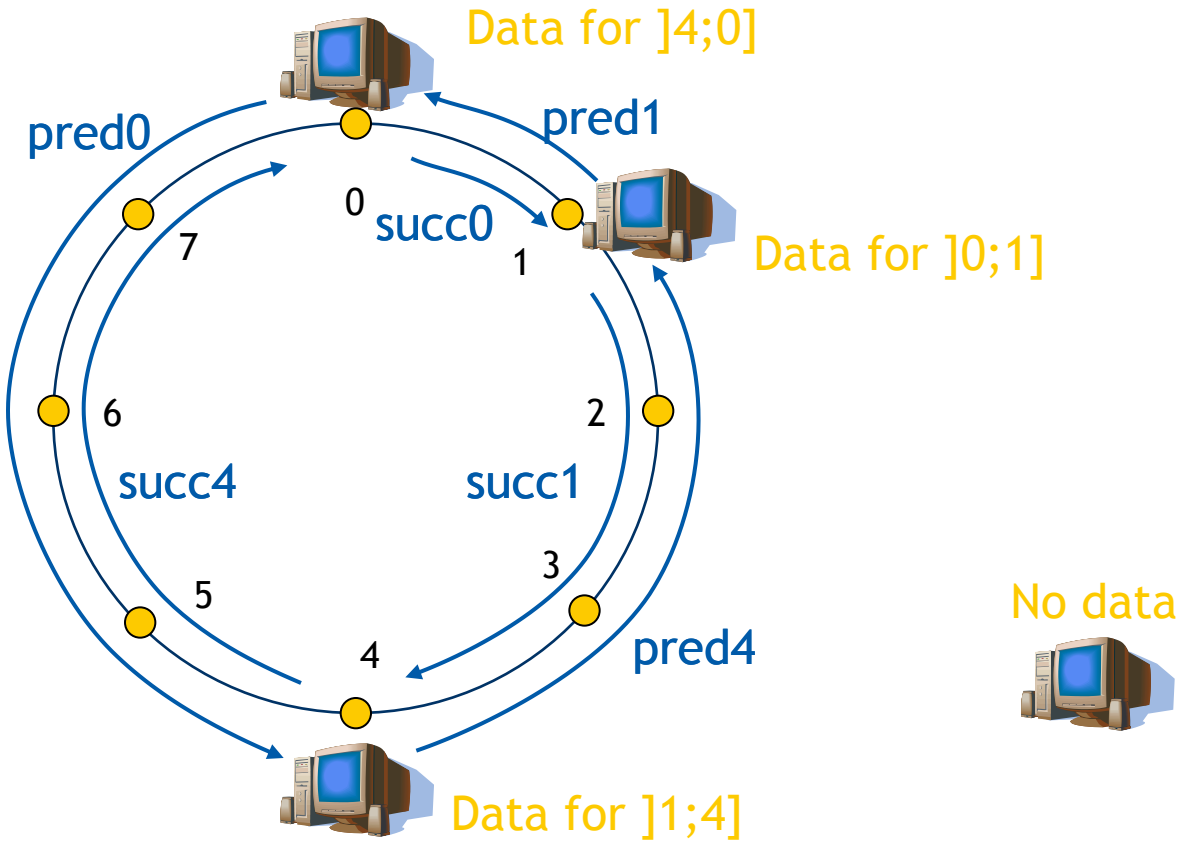
Joining: Step-By-Step Example: Start



- New node wants to join
- Hash of the new node: 6
- Known node in network: Node1

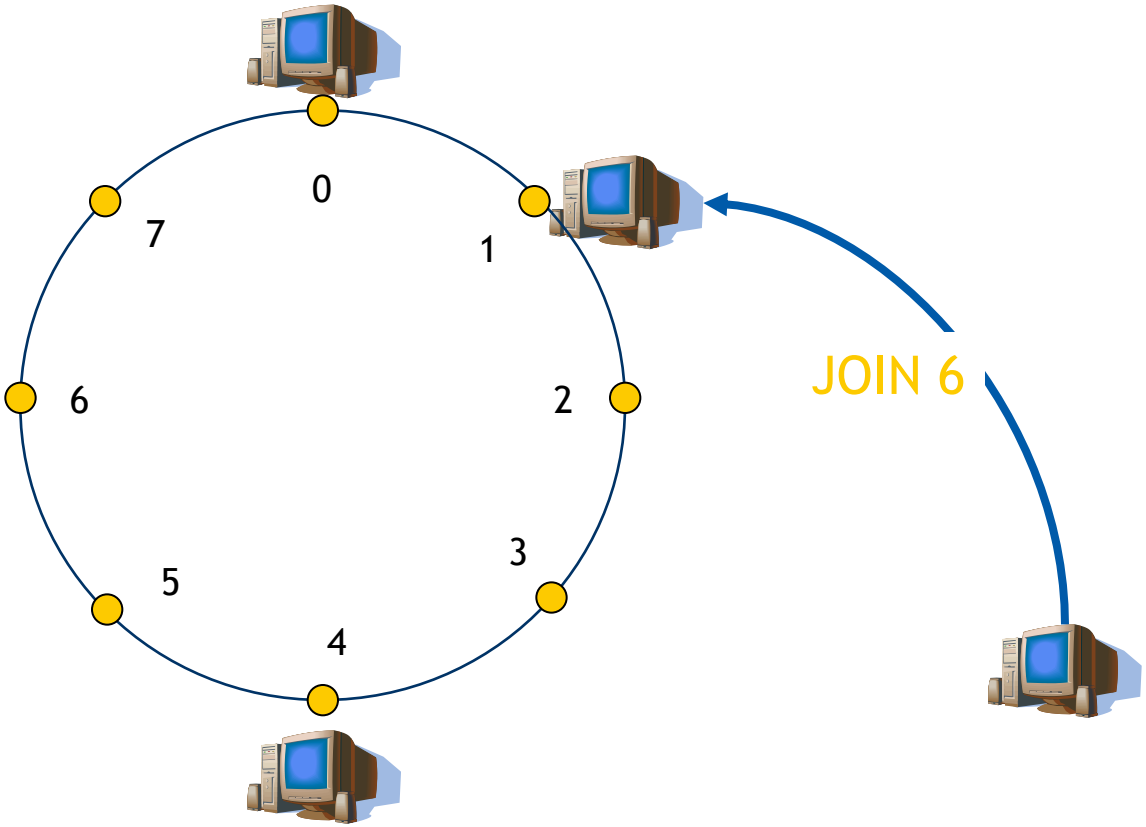
- Contact Node1
  - Include own hash

Joining: Step-By-Step Example: Situation Before Join

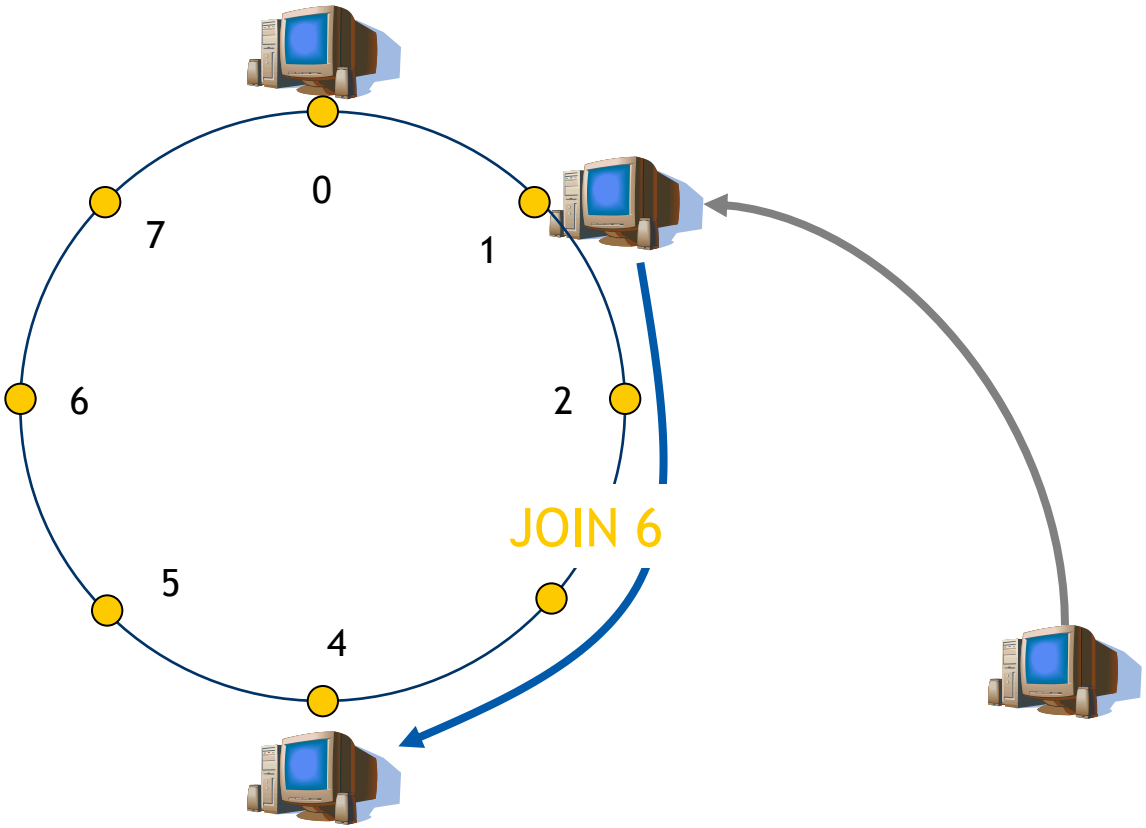


Joining: Step-By-Step Example: Contact known node

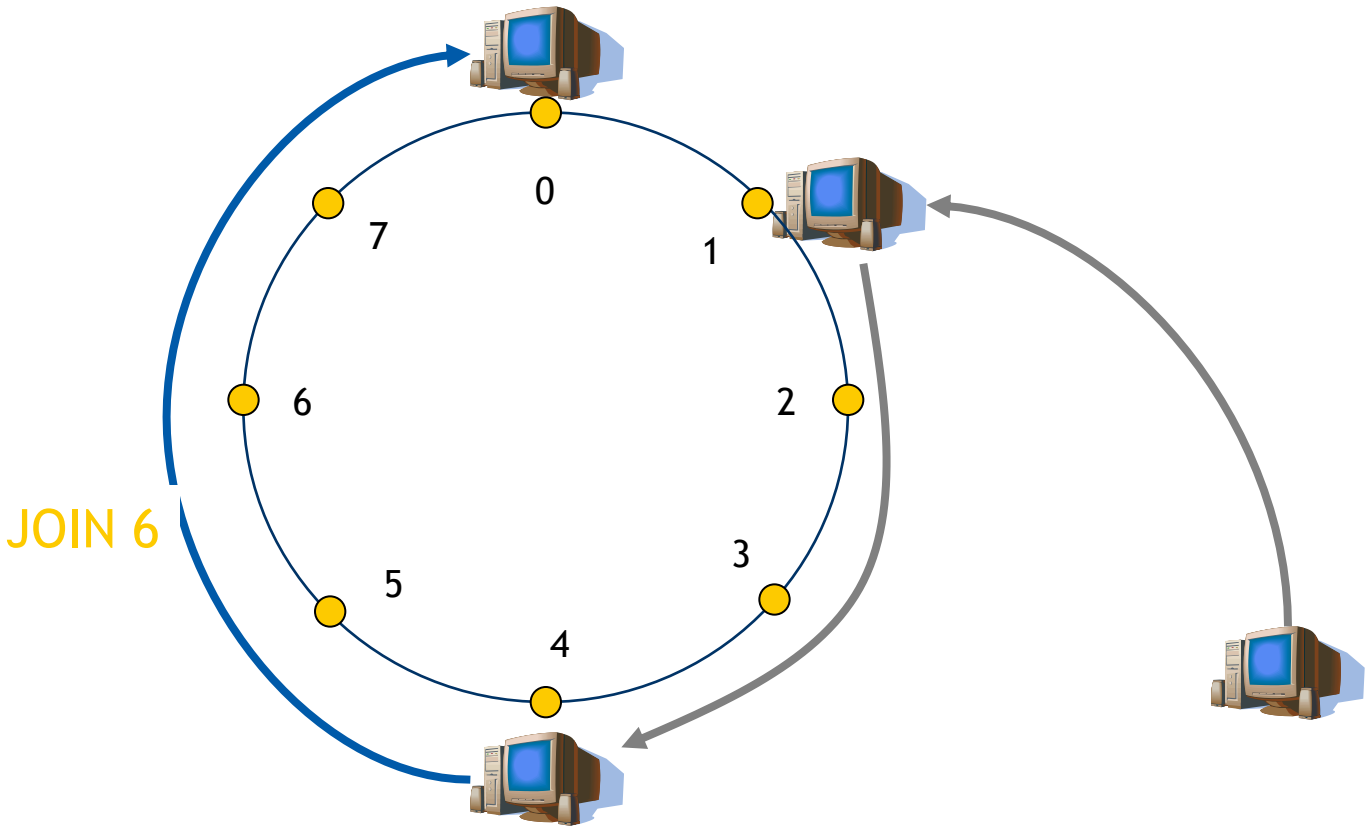
- Arrows indicate open connections
- Example assumes connections are kept open, i.e., messages processed recursively
- Iterative processing is also possible



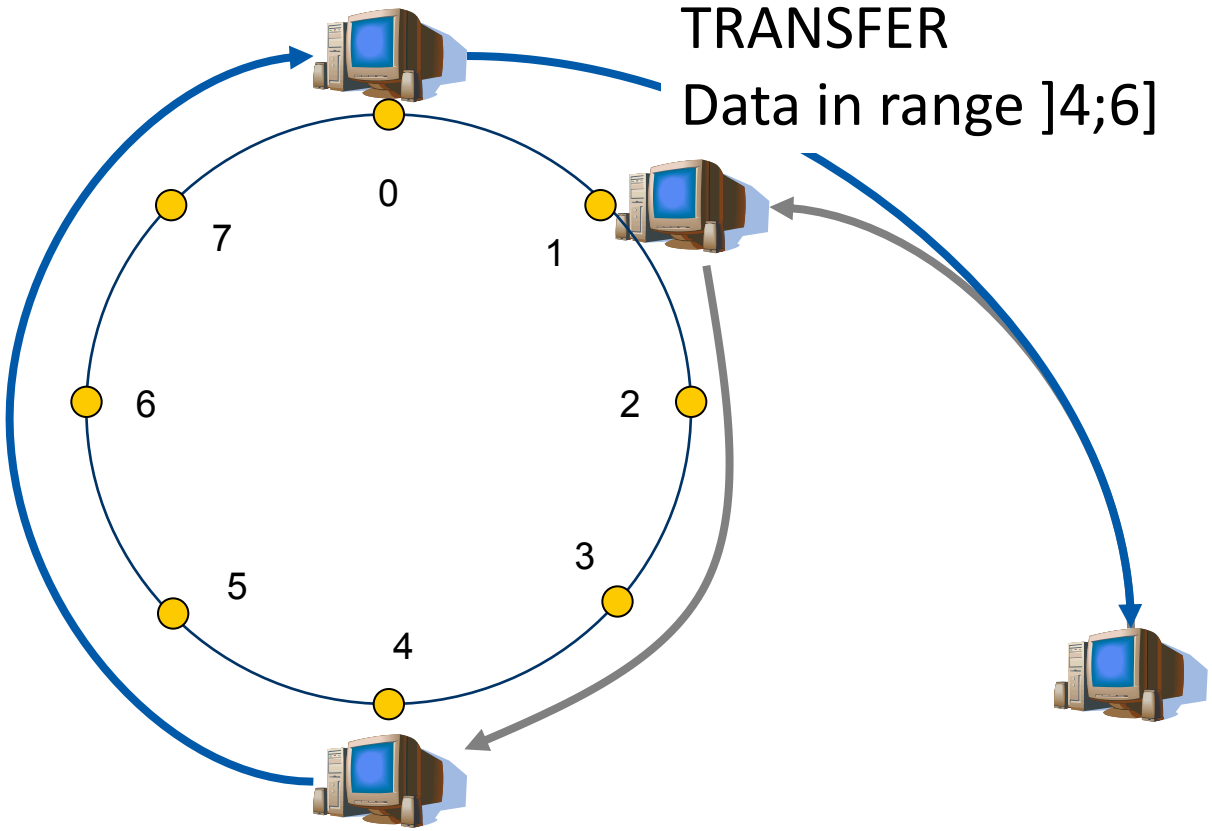
Joining: Step-By-Step Example: Join gets routed along the network



Joining: Step-By-Step Example: Successor of New Node Found



Joining: Step-By-Step Example: Joining Successful + Transfer



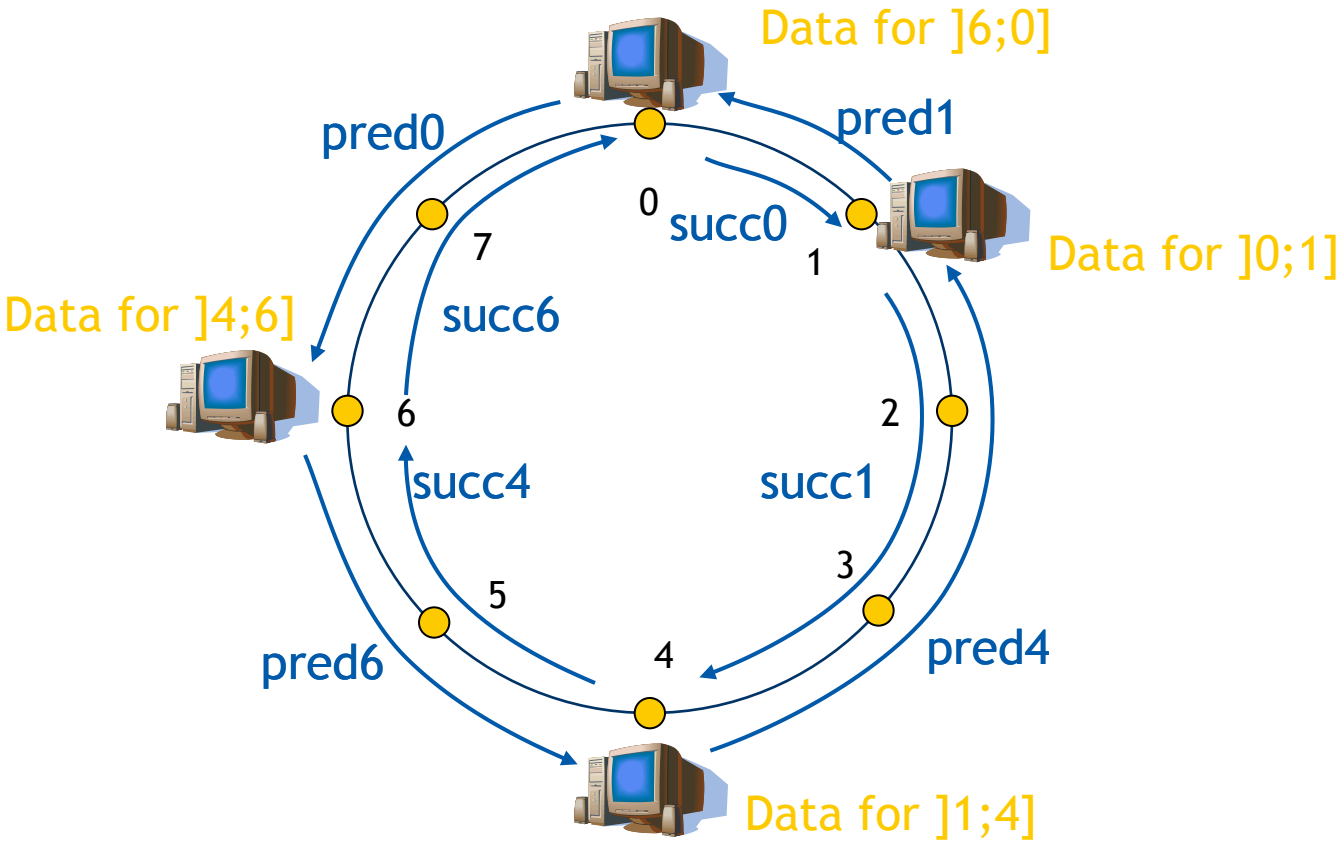
Joining is successful

Old responsible node transfers data that should be in new node

New node informs Node4 about new successor (not shown)

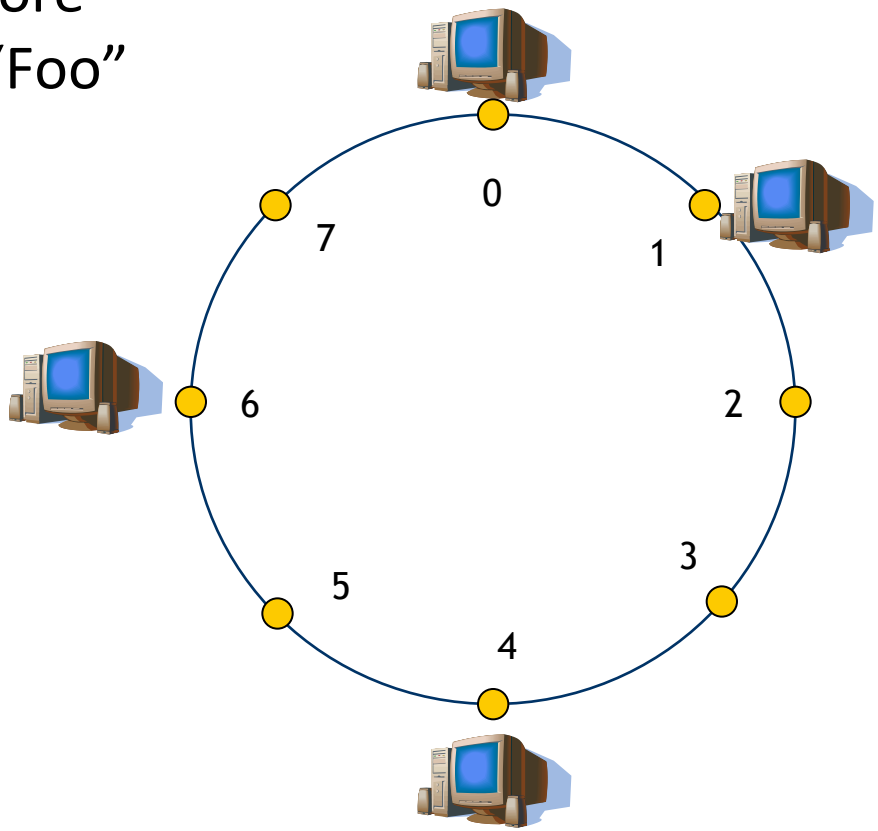
Note: Transferring can happen also later

Joining: Step-By-Step Example: All Is Done

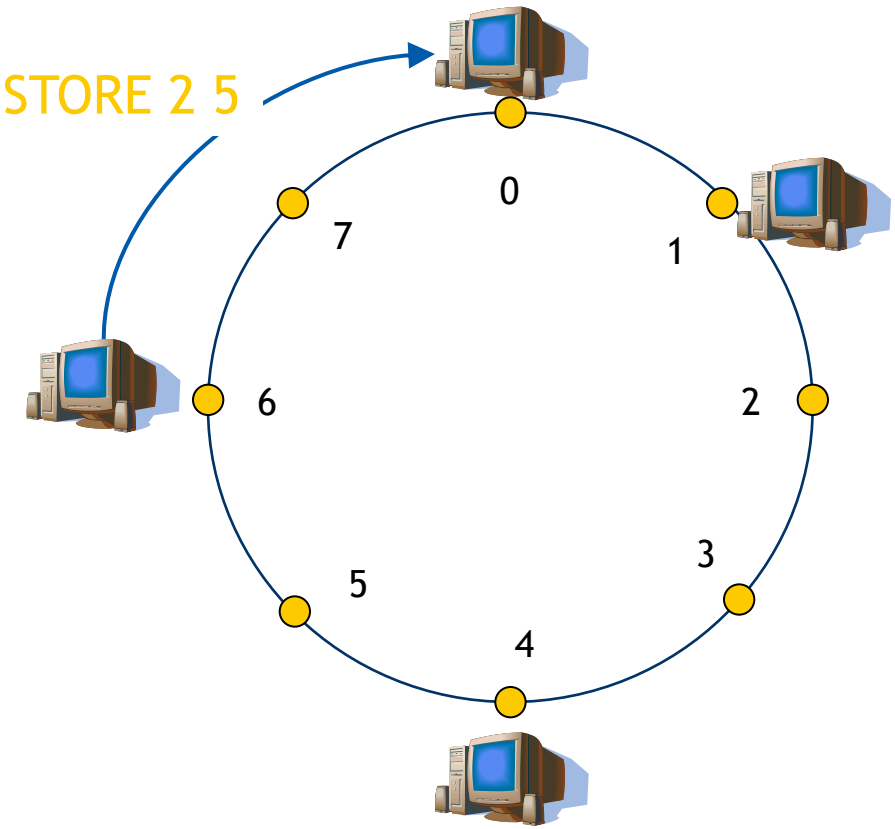


Storing a Value

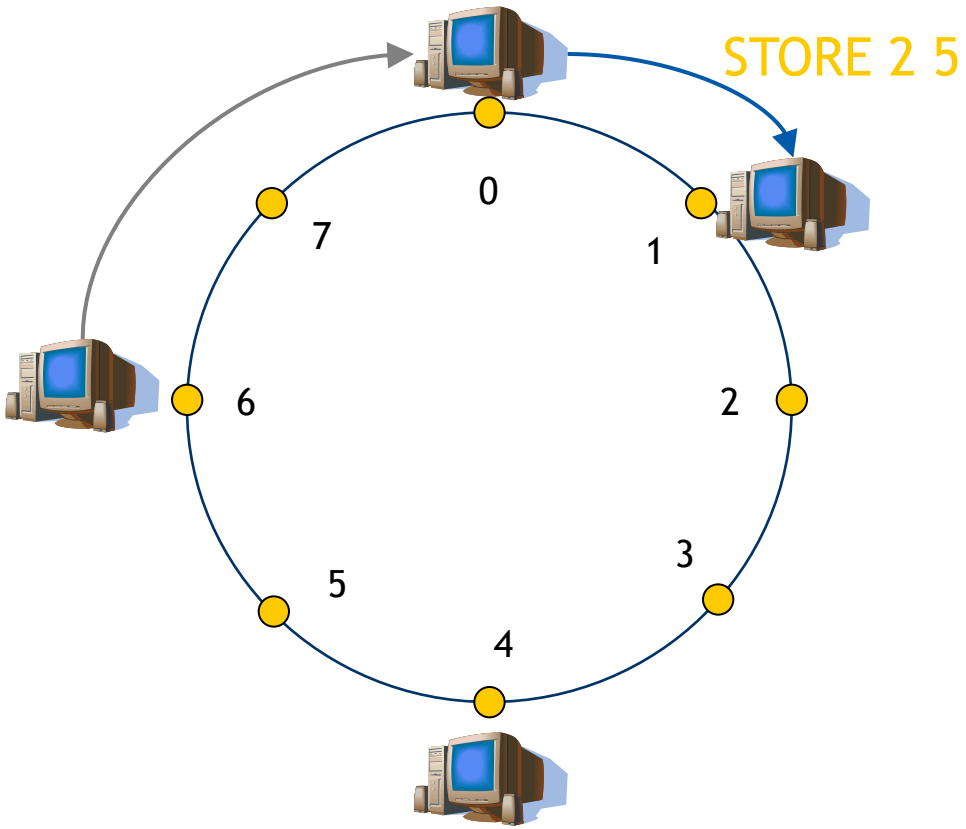
- Node 6 wants to store object with name “Foo” and value 5
- hash(Foo) = 2



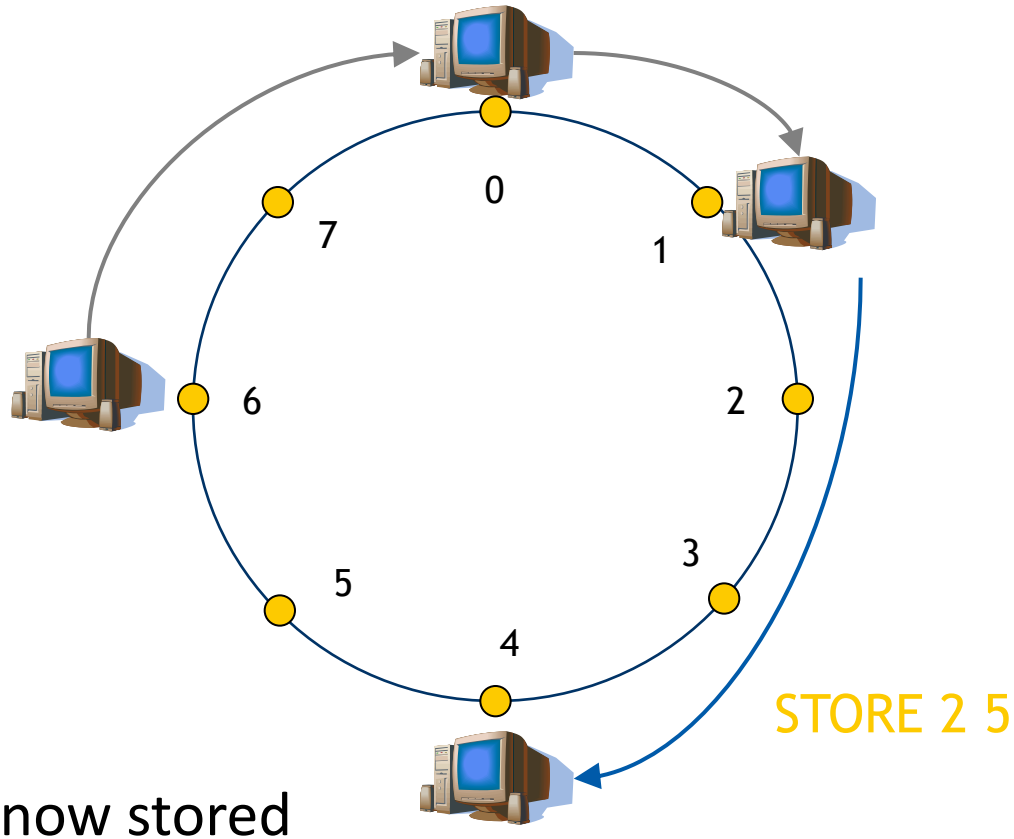
Storing a Value



Storing a Value



Storing a Value

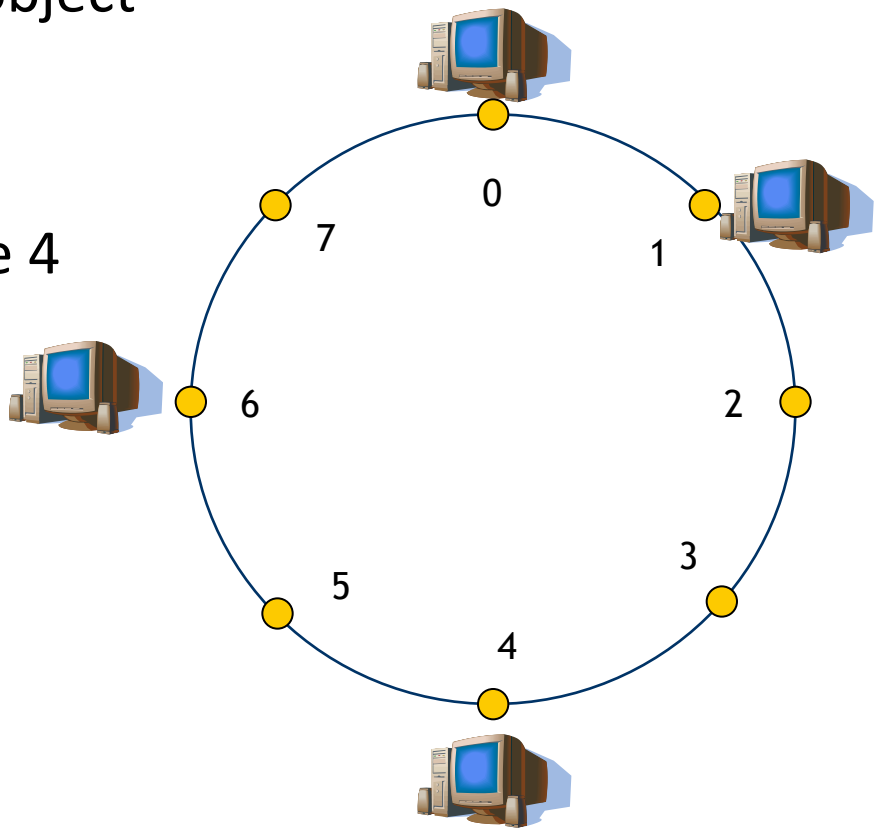


Value is now stored in node 4.

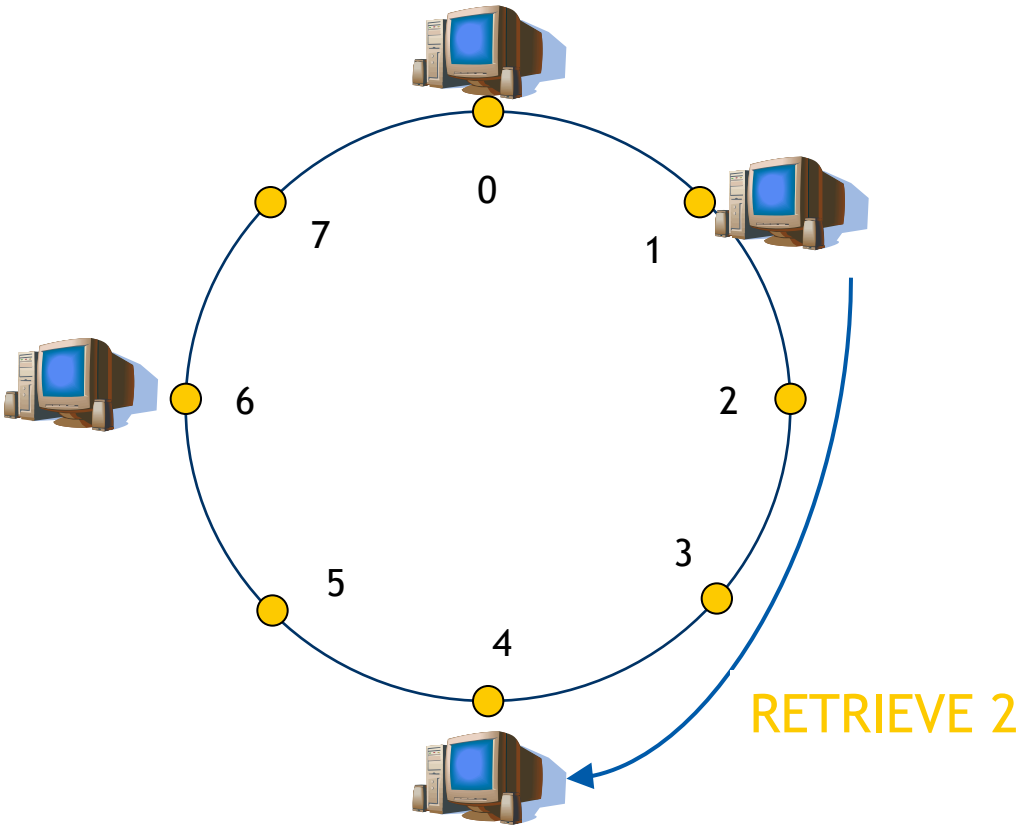
Retrieving a Value

- Node 1 wants to get object with name “Foo”
- hash(Foo) = 2

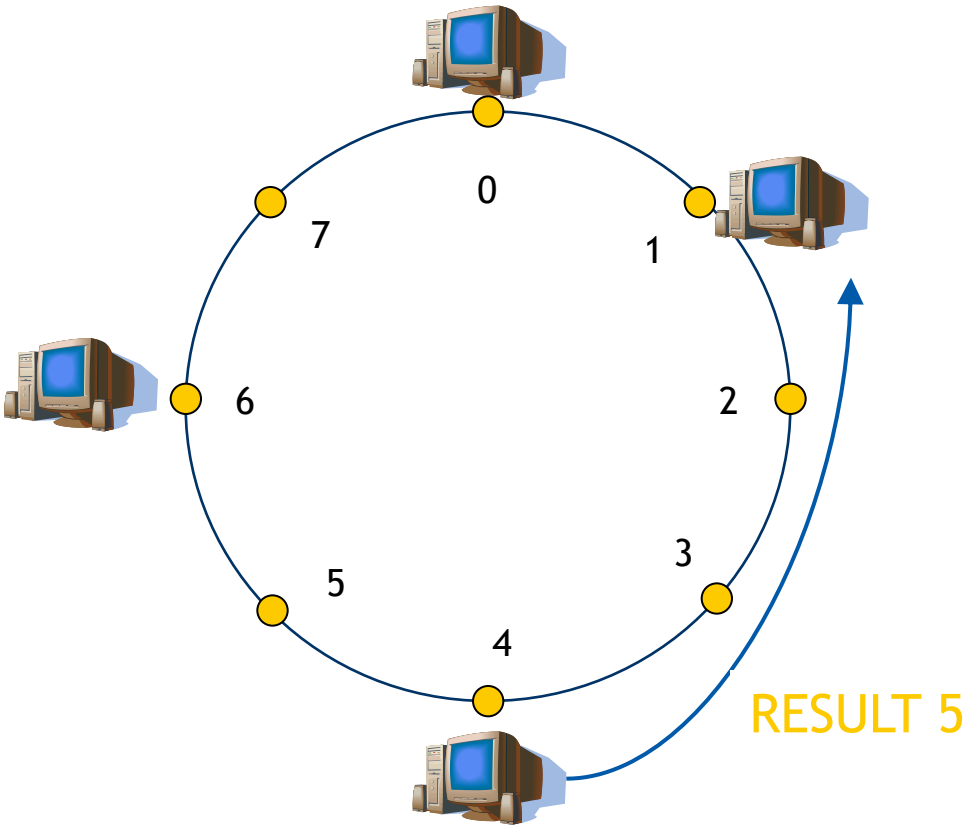
→ Foo is stored on node 4



Retrieving a Value



Retrieving a Value



Chord: Scalable Routing

- Routing happens by passing message to successor
- What happens when there are 1 million nodes?
  - On average, need to route 1/2-way across the ring
  - In other words, 0.5 million hops! Complexity *O(n)*
- How to make routing scalable?
- Answer: Finger tables
- Basic Chord keeps track of predecessor and successor
- Finger tables keep track of more nodes
  - Allow for faster routing by jumping long way across the ring
  - Routing scales well, but need more state information
- Finger tables not needed for correctness, only performance improvement

Chord: Finger Tables

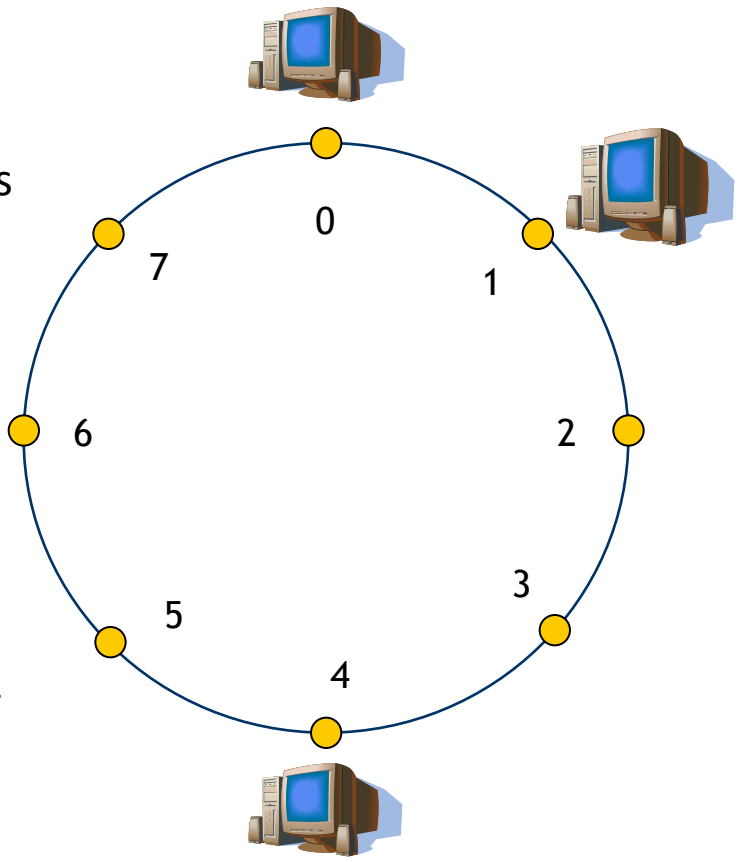
- In *m*-bit identifier space, node has up to *m* fingers
- Fingers are stored in the finger table

- Row *i* in finger table at node *v* contains first node *s* that succeeds *v* by at least 2^{i-1} on the ring (namespace, not nodes!)
- In other words:

$$finger[i] = u : |u| >= |v| + 2^{i-1} \bmod 2^m$$

- First finger is the successor
- Distance to *finger[i]* is at least 2^{i-1}

Chord: Scalable Routing



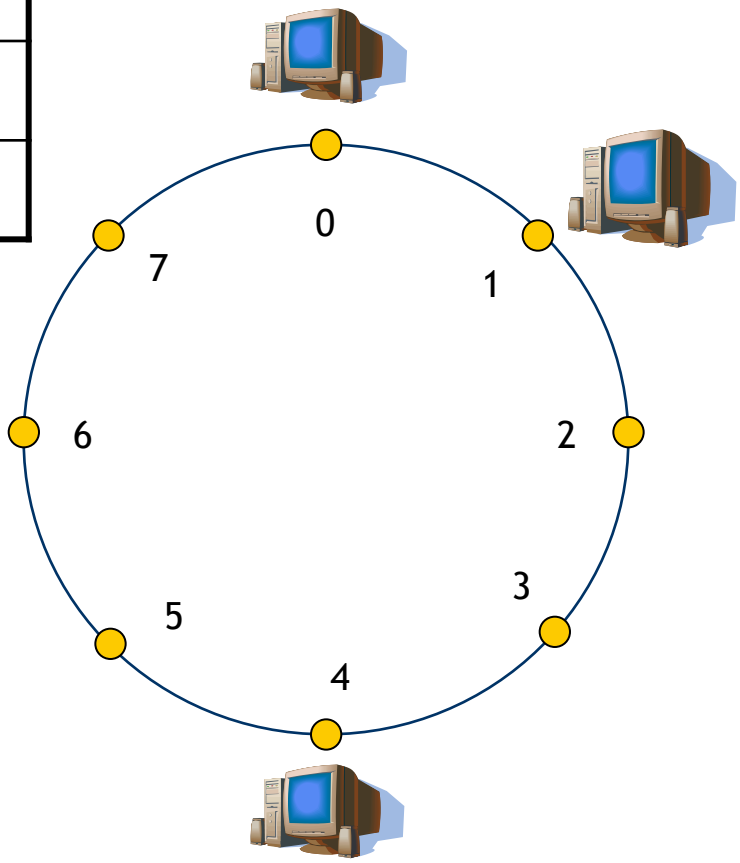
- Finger intervals increase with distance from node n
  - If close, short hops and if far, long hops

Two key properties:

- Each node only stores information about a small number of nodes
- Cannot in general determine the successor of an arbitrary ID

- Example has three nodes at 0, 1, and 4
- 3-bit ID space --> 3 rows of fingers

Chord Finger Tables (Ex)



Start	Int.	Succ.
1	[1,2)	1
2	[2,4)	4
4	[4,0)	4

Start	Int.	Succ.
2	[2,3)	4
3	[3,5)	4
5	[5,1)	0

So for node 4…

Start	Int.	Succ.
5	[5,6)	0
6	[6,0)	0
0	[0,4)	0

Chord: Performance

- Search performance of "pure" Chord *O(n)*
  - Number of nodes is *n*
- With finger tables, need *O(log n)* hops to find the correct node
  - Fingers separated by at least 2^{i-1}
  - With high probability, distance to target halves at each step
  - In beginning, distance is at most 2^m
  - Hence, we need at most *m* hops
- For state information, "pure" Chord has only successor and predecessor, *O(1)* state
- For finger tables, need *m* entries
  - Actually, only *O(log n)* are distinct
  - Proof is in the paper