Peer-to-Peer Networks

Chapter 3: Networks, Addressing, and Distributed Hash Tables

Thorsten Strufe

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/KAD

Searching and Addressing

- Two basic ways to find objects:

1. Search for them
2. Address them using an identifier

- Both have pros and cons (see below)

- File sharing initially based on searching, increasingly implementing object addressing

- Difference between searching and addressing is fundamental
  - Determines how network is constructed
  - Determines how objects are placed
  - Impact on efficiency object discovery

Searching, Addressing, and P2P

- We can distinguish two main P2P network types
- Unstructured networks/systems (previous 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, hierarchy,…
  - Network has structure, but peers are free to join anywhere, perform arbitrary neighbor selection, objects reside 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?

Addressing in a nutshell

- Recall: Object -> Name -> ID -> Reference

- ***Content Addressing*** maps the „content“ on a reference

  - $f : O \rightarrow R$ (*O* being the objects, *R* the namespace of references)

  - Consider f globally known:
    - Anybody can directly derive (and access) reference
    - Direct addressing of content (if resource is known…)
    - Location depends on f and resource only

Addressing, slightly more formal ;-)

- More specifically, f is a composite function:
  - $f : O \rightarrow R$
  - $f_1 : O \rightarrow ID_O$ — f_1 maps resource/object to object identifiers (hash)
  - $f_2 : ID_O \rightarrow ID_V$ — f_2 maps object identifiers to node identifiers
  - $f_3 : ID_V \rightarrow A$ — f_3 maps node identifiers to node addresses
  - $f_4 : ID_O \times A \rightarrow R$ — f_4 concatenates node address and object ID

  - $f(o) = f_4\left(f_1(o),\ f_3 \circ f_2 \circ f_1(o)\right)$

Is such functionality always useful?

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

Addressing vs. Searching

- “Addressing” systems find objects by ***addressing*** with unique name
  (cf. URLs in Web, ***location service***)

- “Searching” systems find objects by ***searching*** with keywords that matchdescription
  (cf. Google, ***name- and location / discovery service***)

Addressing

- Pros:
  - Each object uniquely identifiable
  - Object location potentially “efficient” (log no. of steps with log no. neighbors)
- Cons:
  - Need to know ID
  - Need to maintain structure required for addressing

Searching

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

Distributed Hash Tables

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

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

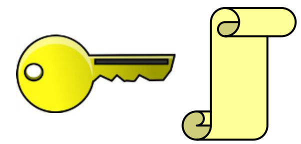
- Searching does not scale in same way

- Original motivation for DHTs:
  More efficient searching and object location in P2P networks

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*

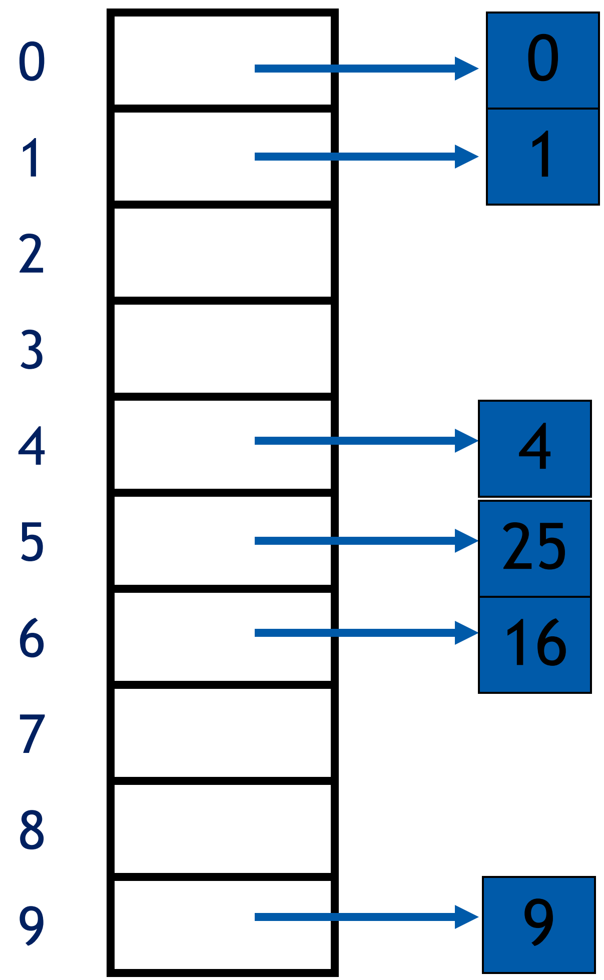


- *Hash function* maps keys to elements in the array

- 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

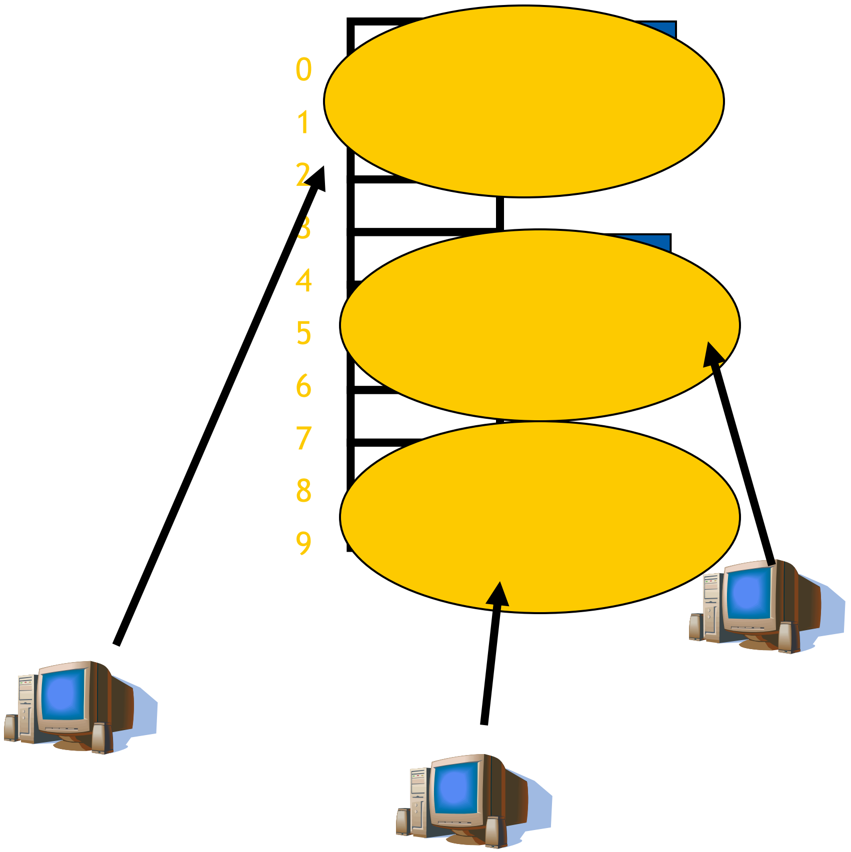


- Hash function:

  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

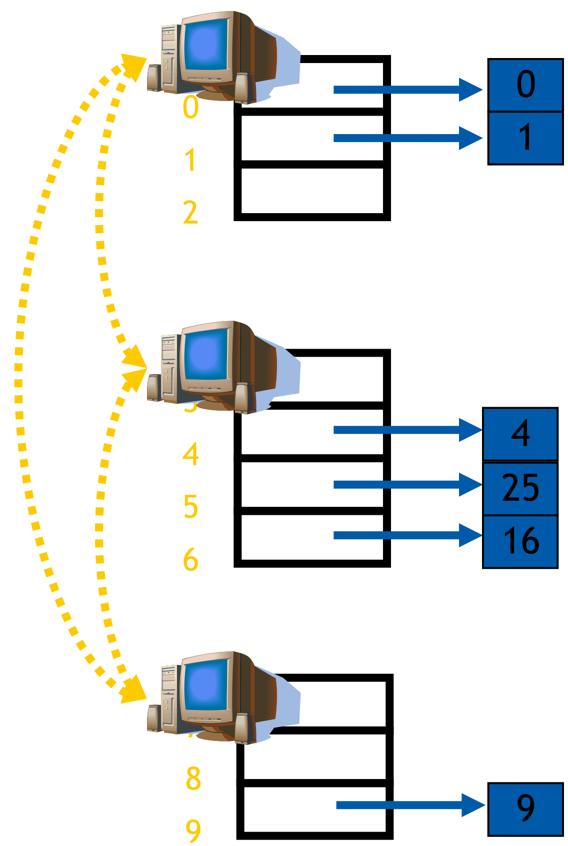
Distributed Hash Table: Idea

- Hash tables are fast for lookups
- Idea: Distribute hash buckets to peers
- Result is Distributed Hash Table (DHT)
- Needed:
  efficient mechanism for finding which peer is responsible for which bucket ($f_2 \circ f_1(o)$) and route towards it (f_3)



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
  - Scalability and efficiency of communication make DHTs performant
- 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 structure and metrics in the overlay network are main difference between different DHTs
- DHT behavior and usage:
  - Node knows ID of resource it wants to find
    - Unique and known object IDs are assumed
  - Node routes a message in overlay to the responsible node
  - Responsible node replies with "object" (or reference to it)
    - Semantics of "object" are application defined
  - *f: O → R* *and Bob's your uncle* ☺

DHT Examples

- In the following look at some example DHTs
  - Chord
  - CAN
  - Tapestry
  - KAD
- 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
- Overlay structure and metric (for routing) are 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 m-bit hash function (SHA-1, gives 160bit ID space)
  - Results in a m-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

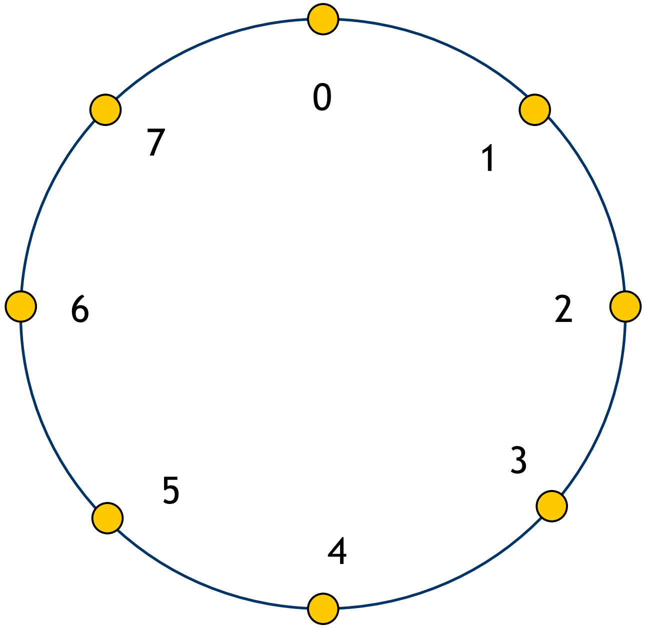
- IDs organized on a ring (interval $[0 .. 2^m-1]$ with wrap-around)
  - Overlay is often called "Chord ring" or "Chord circle"
  - Nodes keep track of predecessor and successor
  - Node ***registers*** objects on the namespace ***between predecessor and itself*** (recall: $f_2 : ID_O \rightarrow ID_V$)
  - ***Distance*** metric $d(id_v, id_u) = (id_u - id_v) \bmod 2^m$
    - Distance is asymmetric, with wrap-around (only clockwise routing)

Chord: Examples

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

Joining: Step-By-Step Example

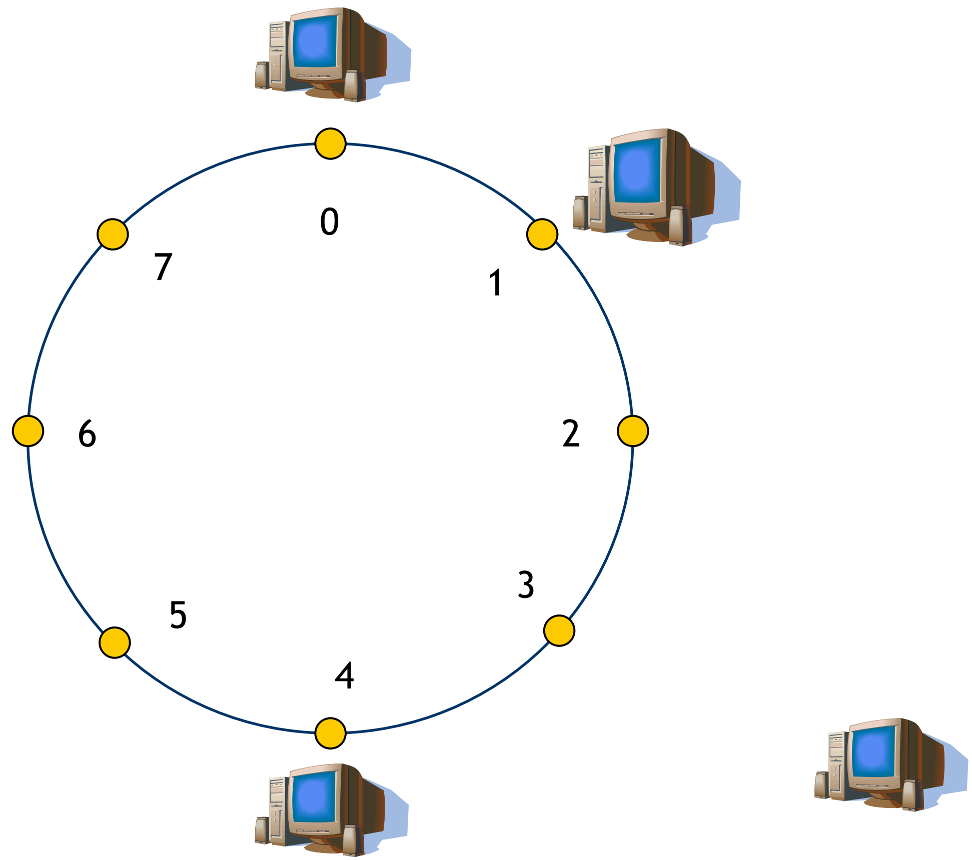
- Setup: Existing network with nodes on 0, 1 and 4
- Note: Protocol messages simply examples
- Many different ways to implement Chord
  - Here only conceptual example
  - Covers all important aspects



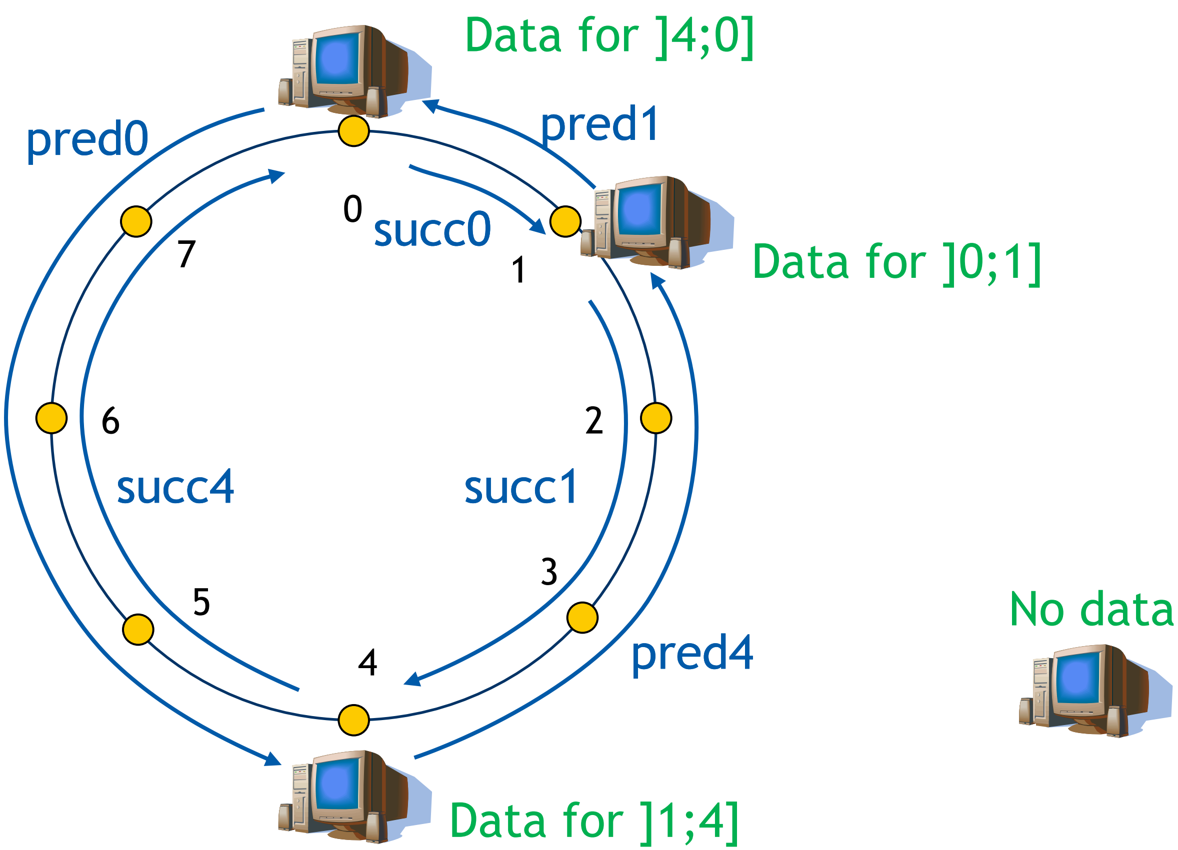
Joining: Step-By-Step: Start

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

- Contact Node1
  - Include own hash

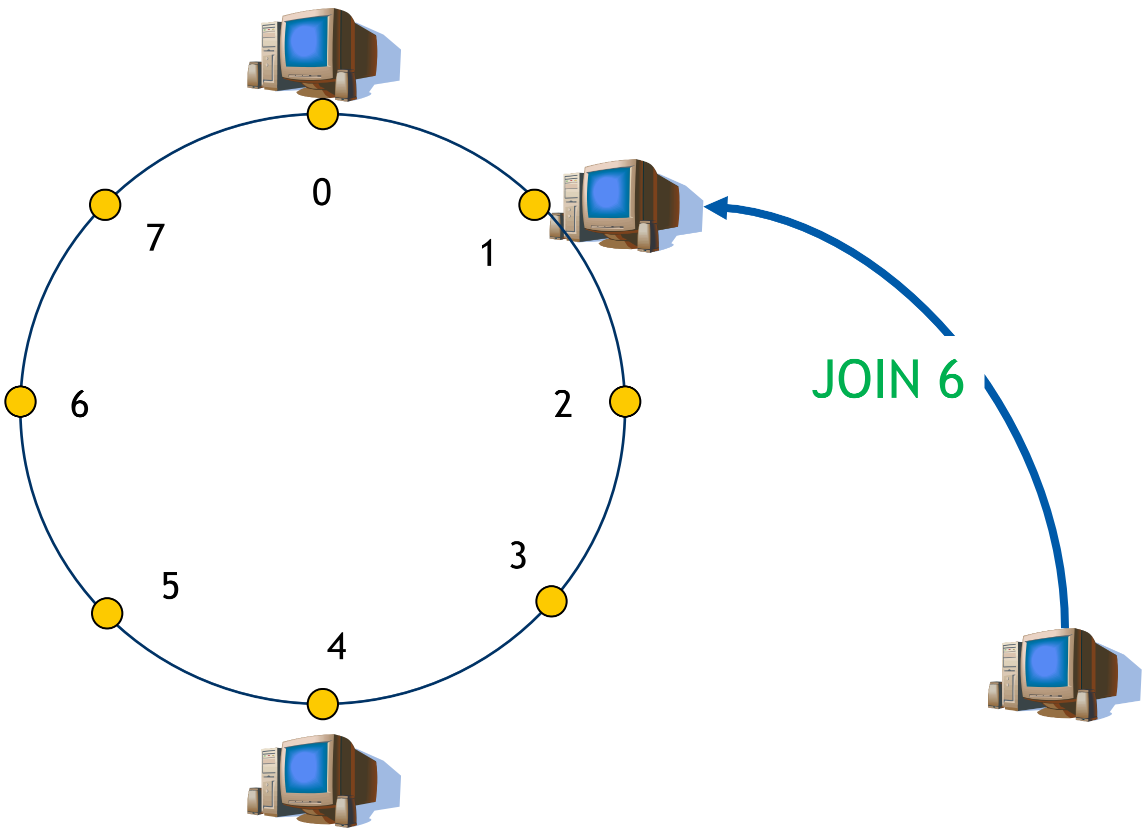


Joining: Step-By-Step: Situation Before Join

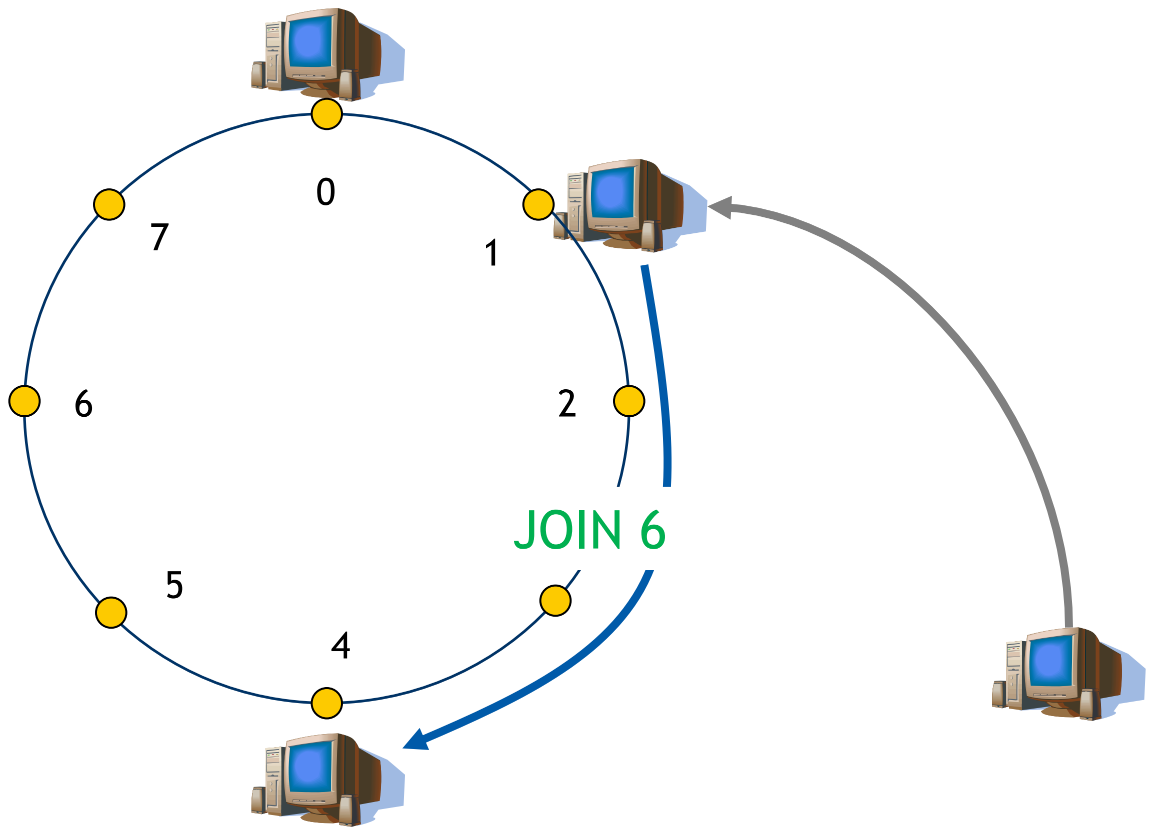


Joining: Step-By-Step: Contact known node

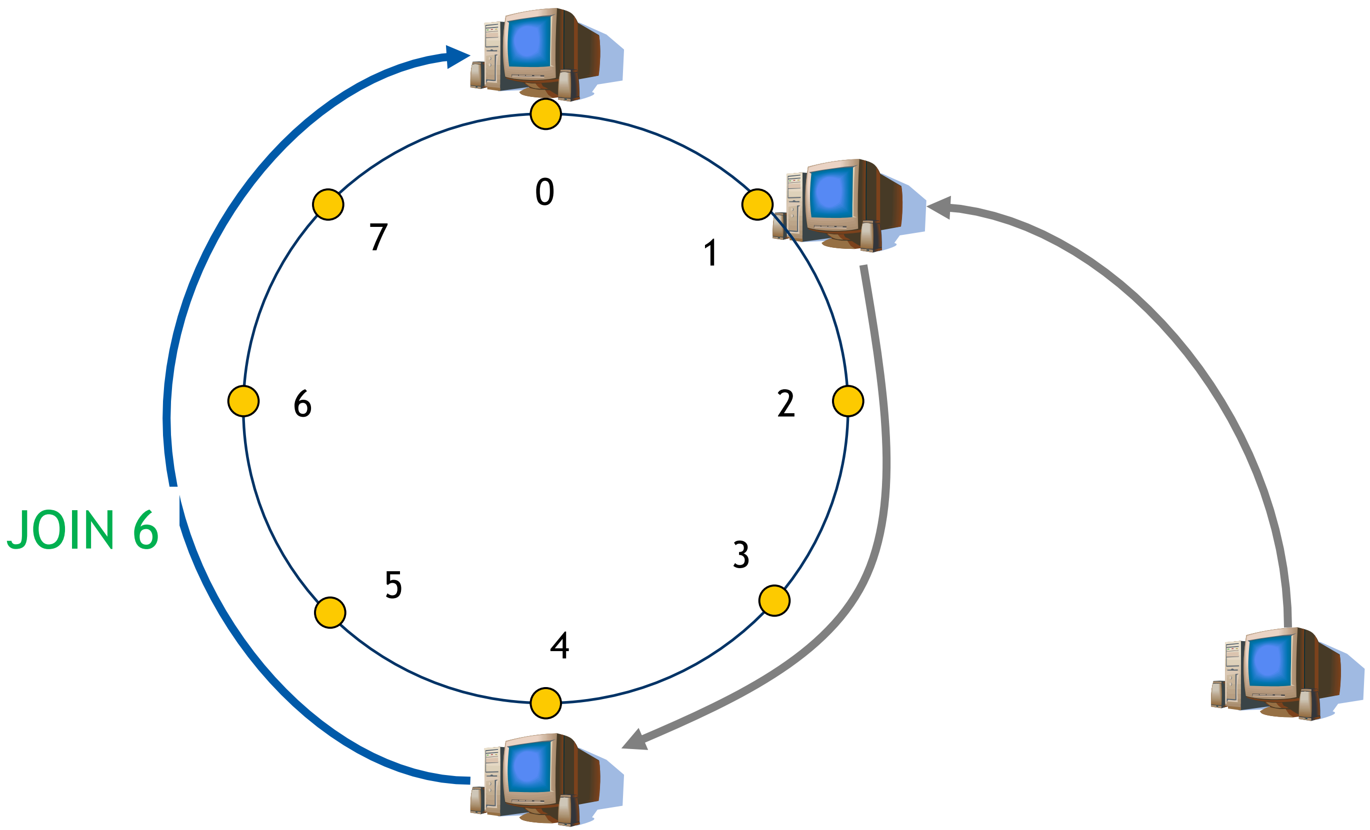
- Arrows indicate open connections
- Example assumes connections are kept open, i.e., messages processed recursively
- Iterative processing is also possible



Joining: Step-By-Step: Routing along the network



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

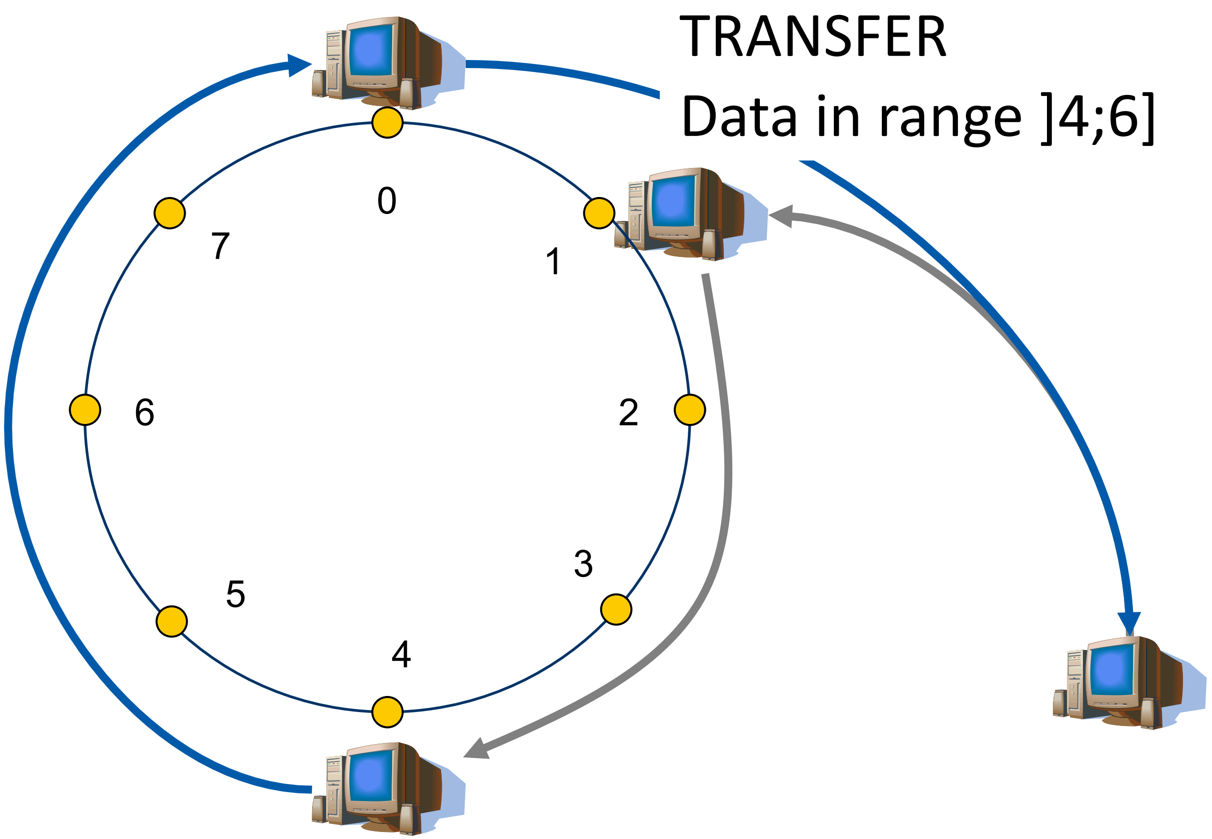


Joining: Step-By-Step: Joining Successful + Transfer

Joining is successful

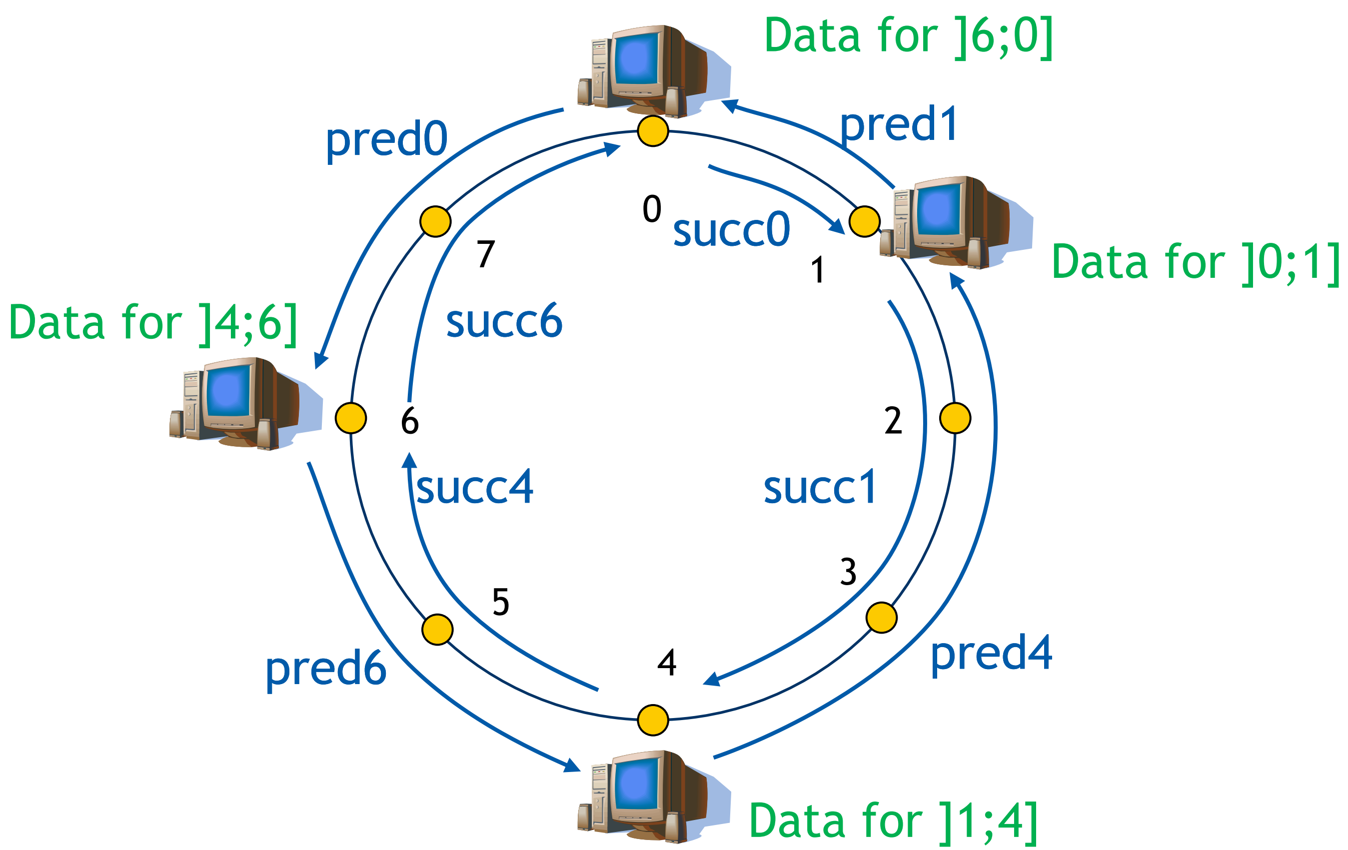
Old responsible node transfers data that should be in new node

New node informs Node4 about new successor (not shown)



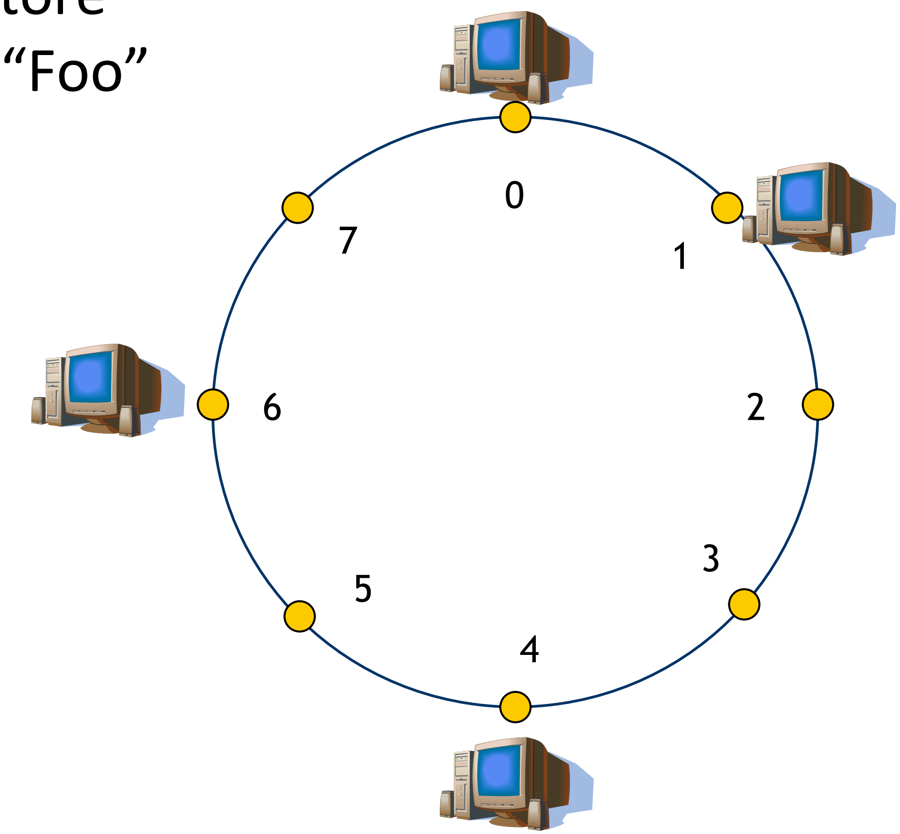
Note: Transferring can happen also later

Joining: Step-By-Step: Done

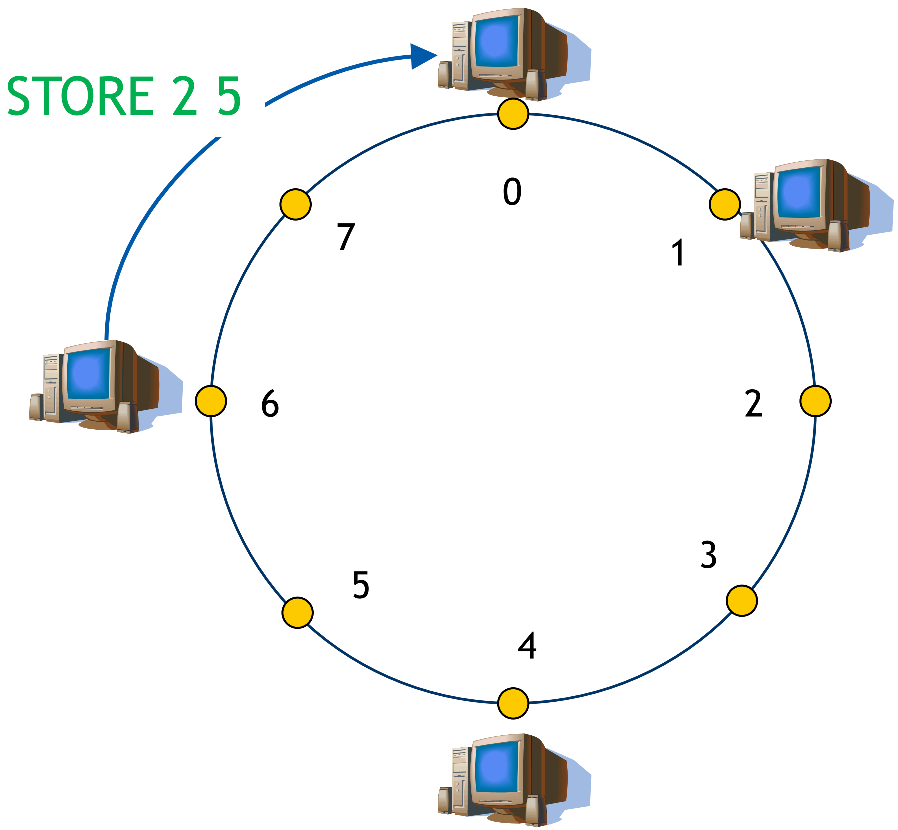


Storing a Value

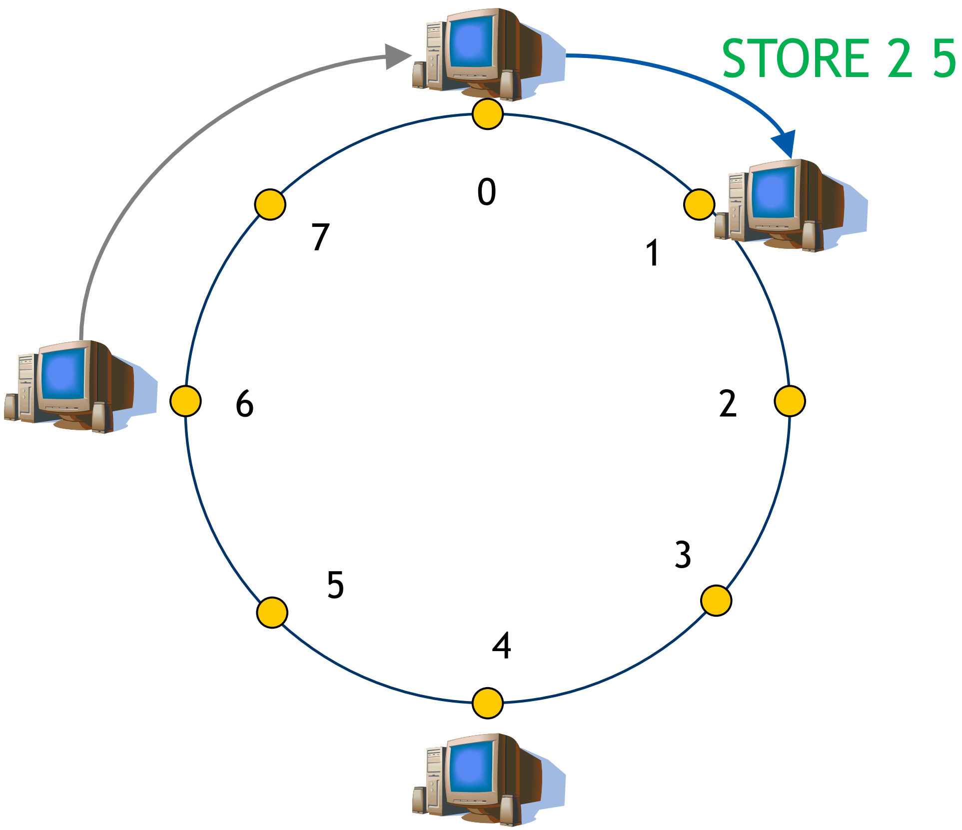
- Node 6 wants to store object with name “Foo” and value 5
- hash(Foo) = 2



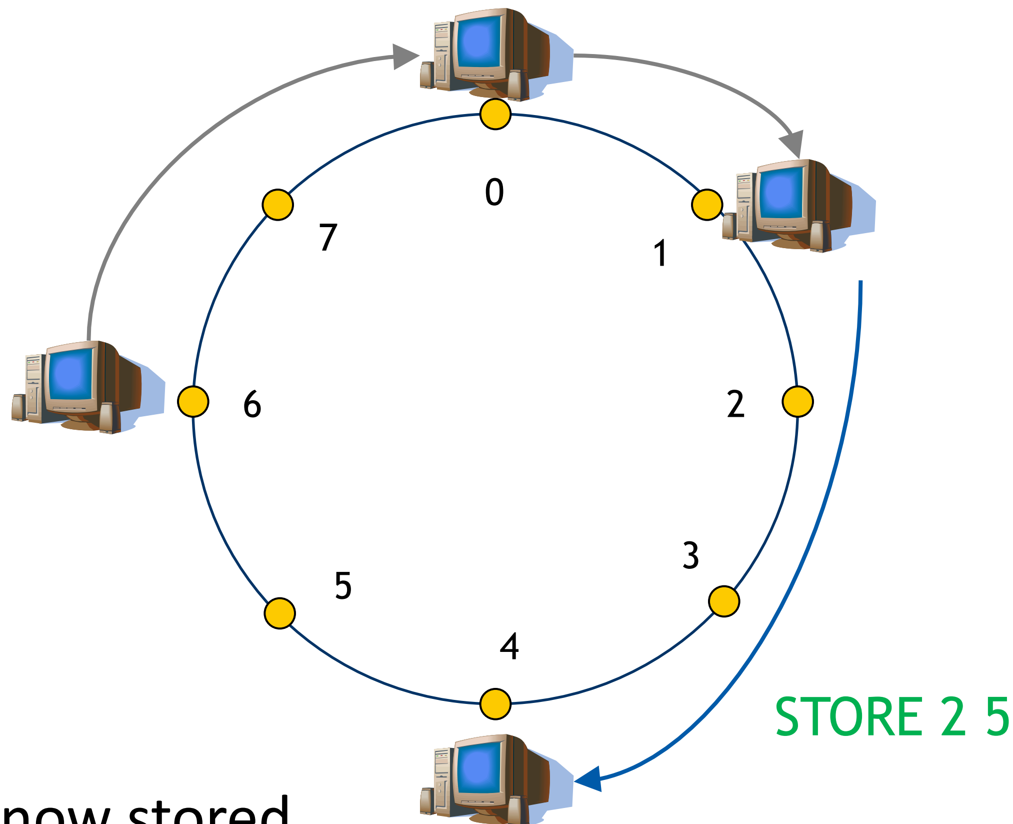
Storing a Value



Storing a Value



Storing a Value

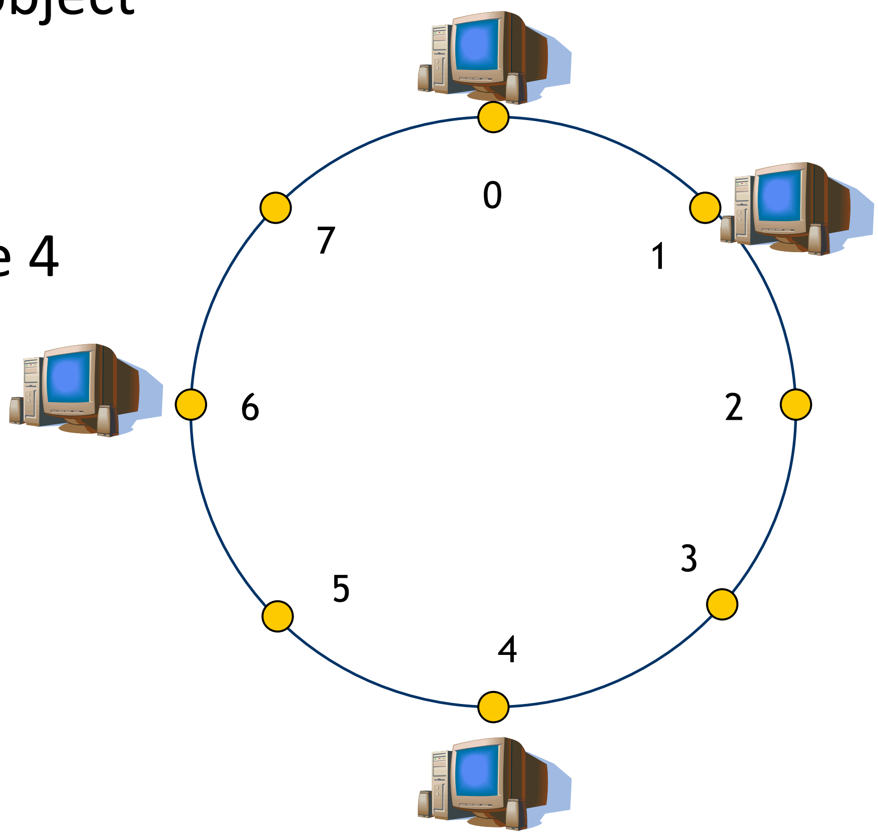


Value is now stored in node 4.

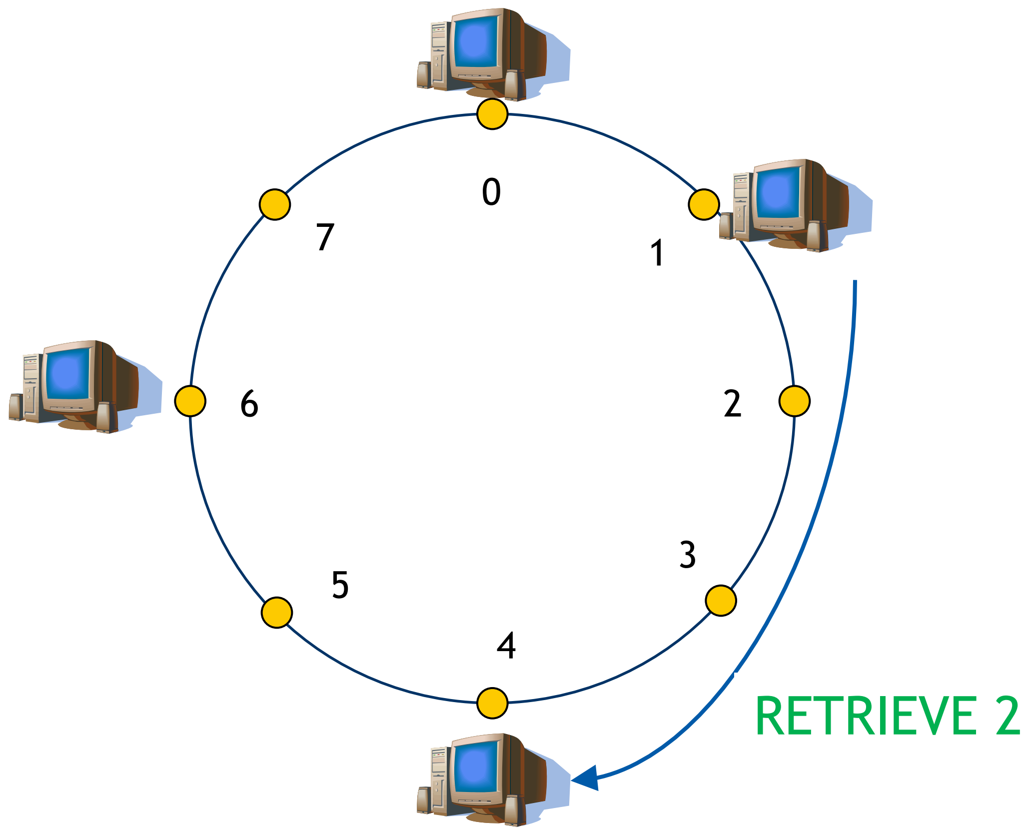
Retrieving a Value

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

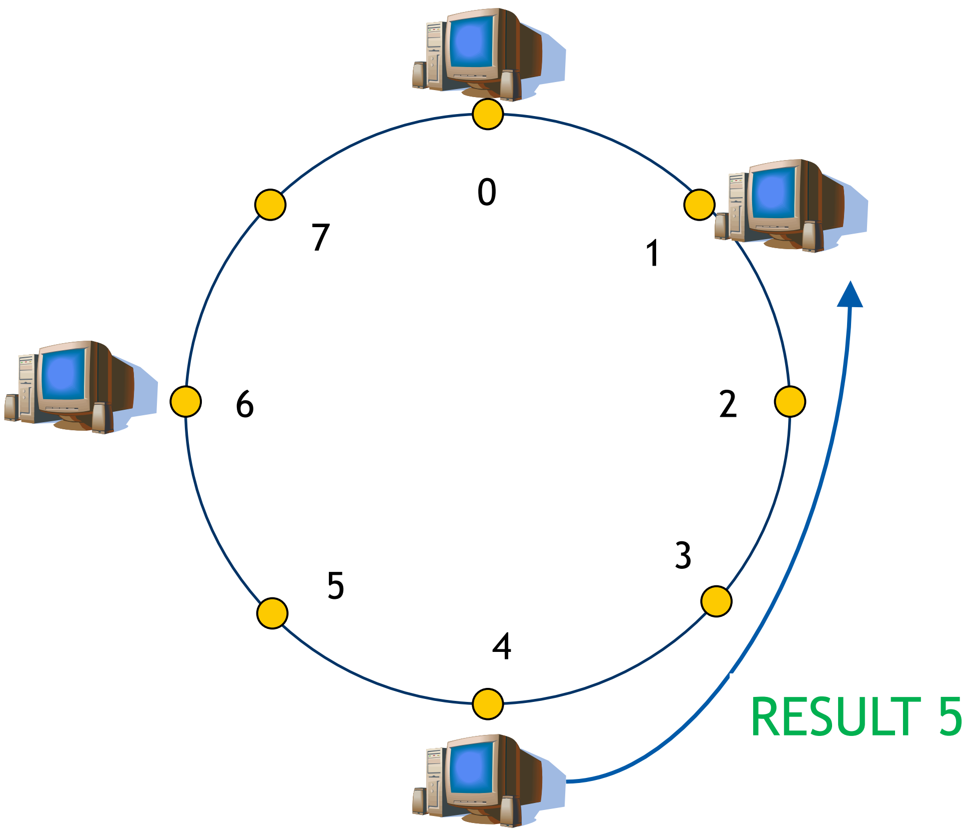
→ Foo is stored on node 4



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 on average! 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

- ***Behold:*** 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:

$$finger[i] = u : succ(id_v + 2^{i-1} \bmod 2^m) \text{ with } 1 \leq i \leq m$$

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

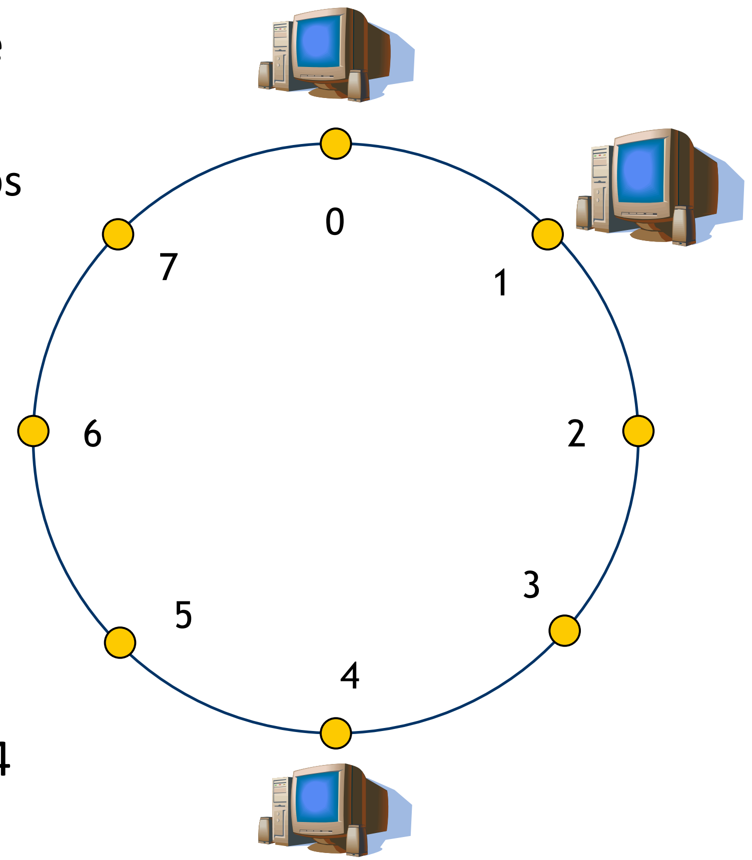
Chord: Scalable Routing

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

Two key properties:

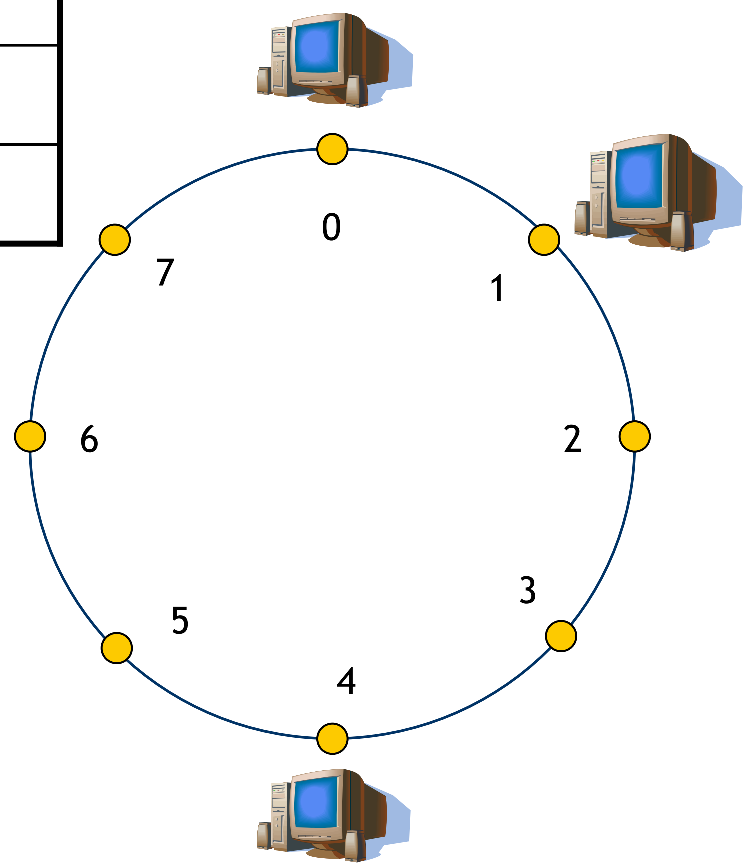
- Each node only stores information about a small number of nodes
- Cannot determine the successor of an arbitrary ID in general

- 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

To Hash or not to hash?

[redacted]

Addressing possible but no searching, because Hashes H(foo) are used…

Why not store the names un-hashed („foo“)?

Node-ID allocation

Node-ID is allocated by hashing the IP-Address…

- Does this have dis-advantages?

- Advantages, too, may be?

CAN: Content Addressable Network

- CAN developed at UC Berkeley
- *(Ratnasamy, Francis, Handley, Karp, Shenker)*
- Originally published in 2001 at Sigcomm conference(!)

- CANs overlay routing easy to understand
  - Paper concentrates more on performance evaluation
  - Also discussion on how to improve performance by tweaking

- CAN project did not have much of a follow-up
  - Only overlay was developed, no bigger extensions
  - Interestingly enough, the idea is coming back with a twist…

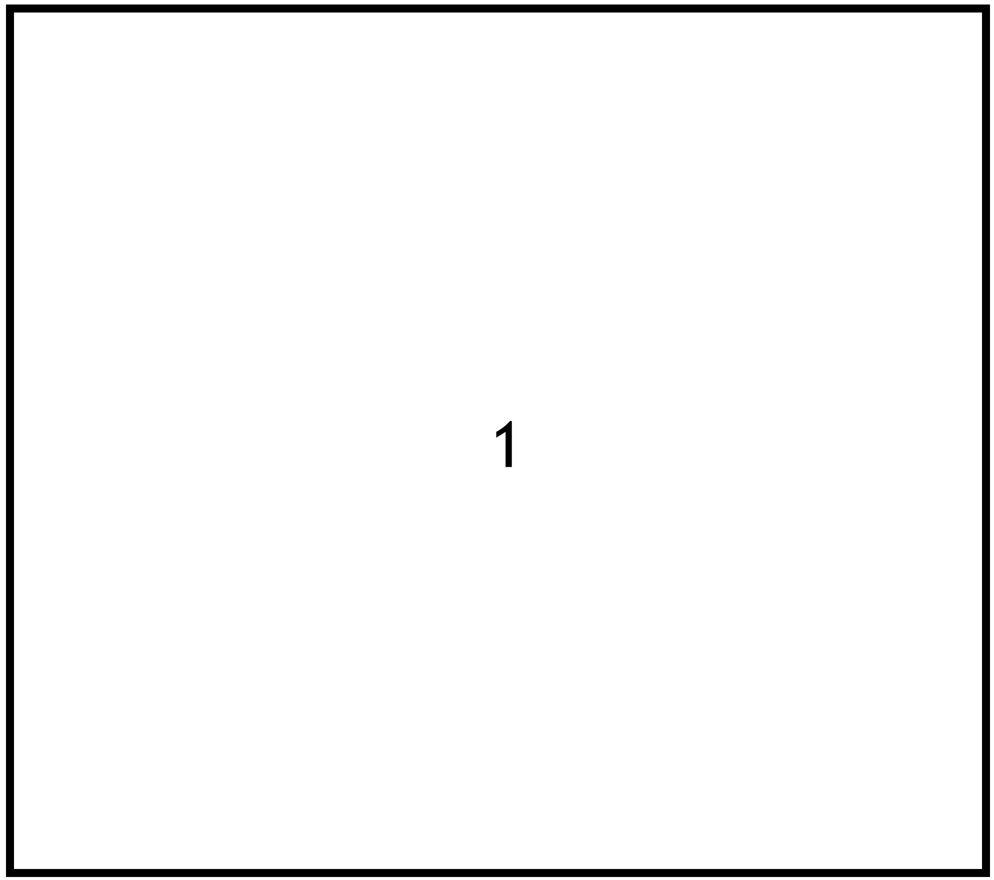
CAN: Basics

- CAN based on N-dimensional Cartesian coordinate space
  - Our examples: N = 2
  - One hash function for each dimension
- Entire space is partitioned amongst all the nodes
  - Each node owns a zone in the overall space

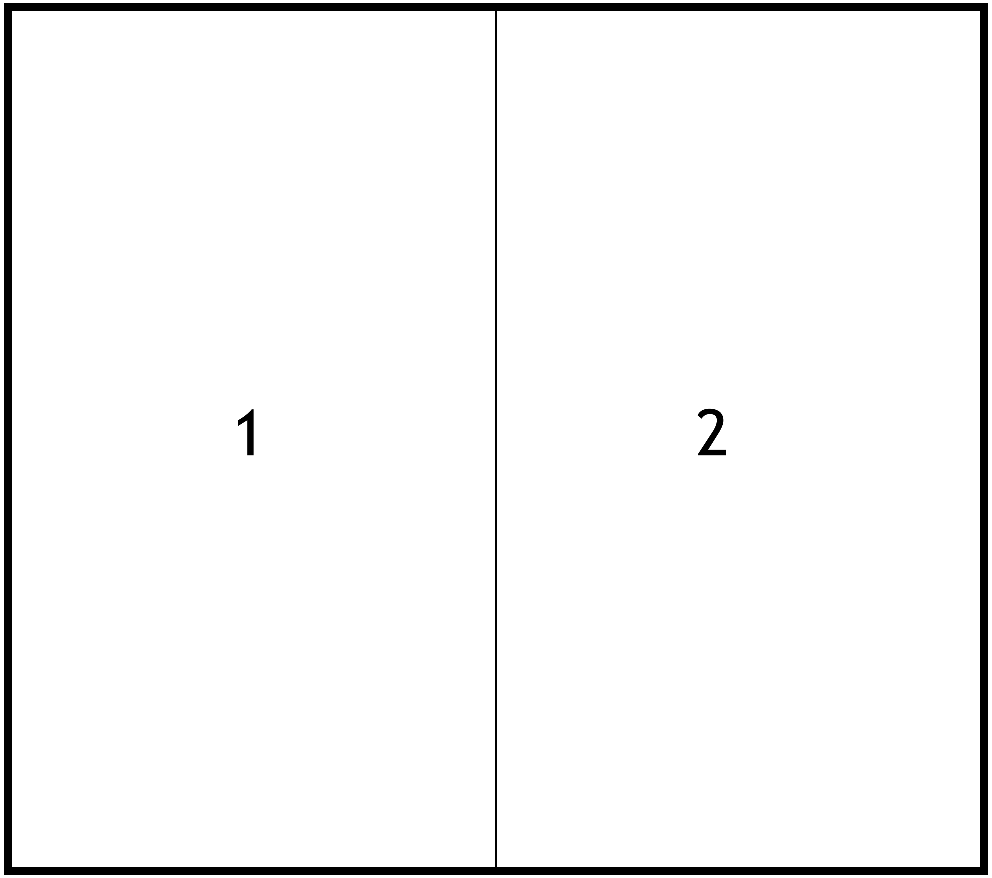
- Abstractions provided by CAN:
  - `store` data at points in the space
  - `route` from one point to another

- Point = Node that owns the zone in which the point (coordinates) is located
- Order in which nodes join is important

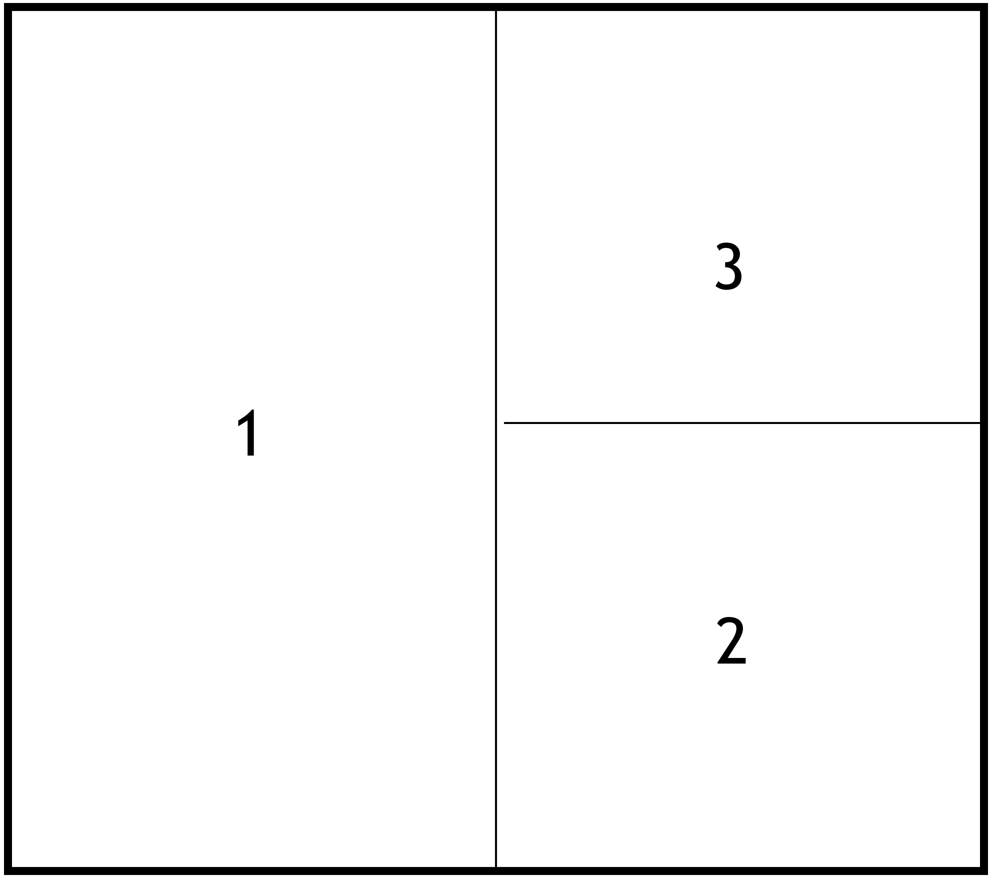
CAN: Partitioning



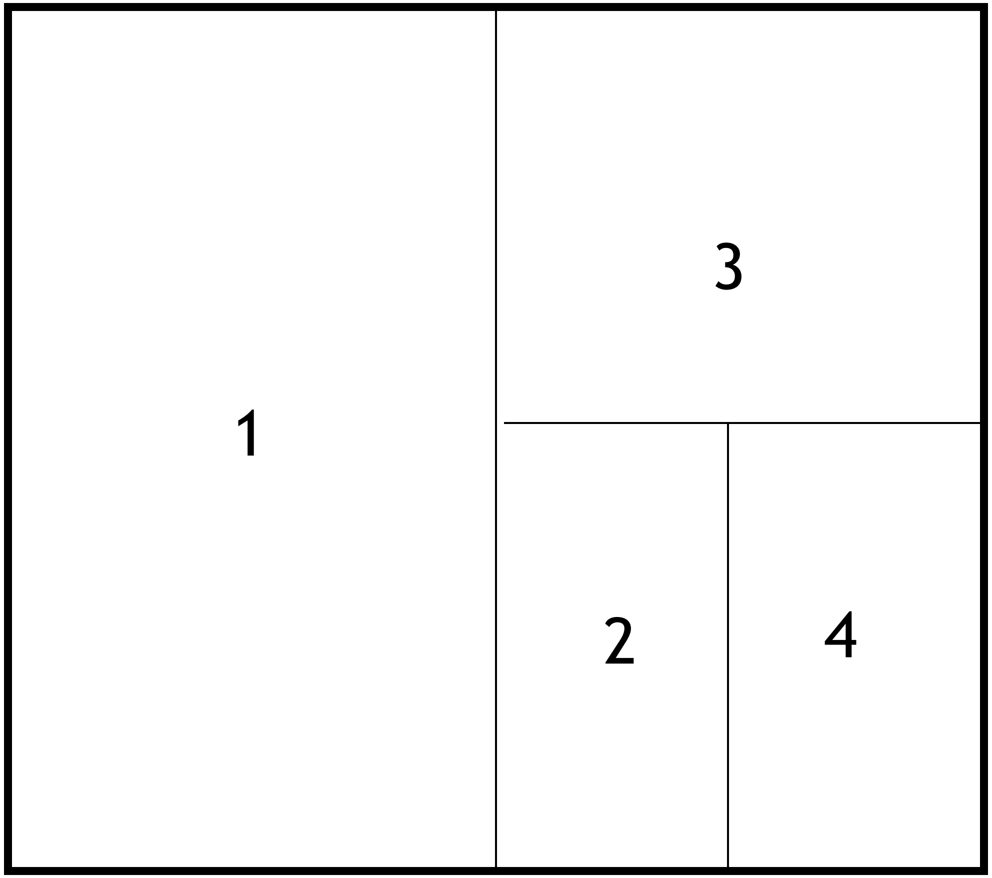
CAN: Partitioning



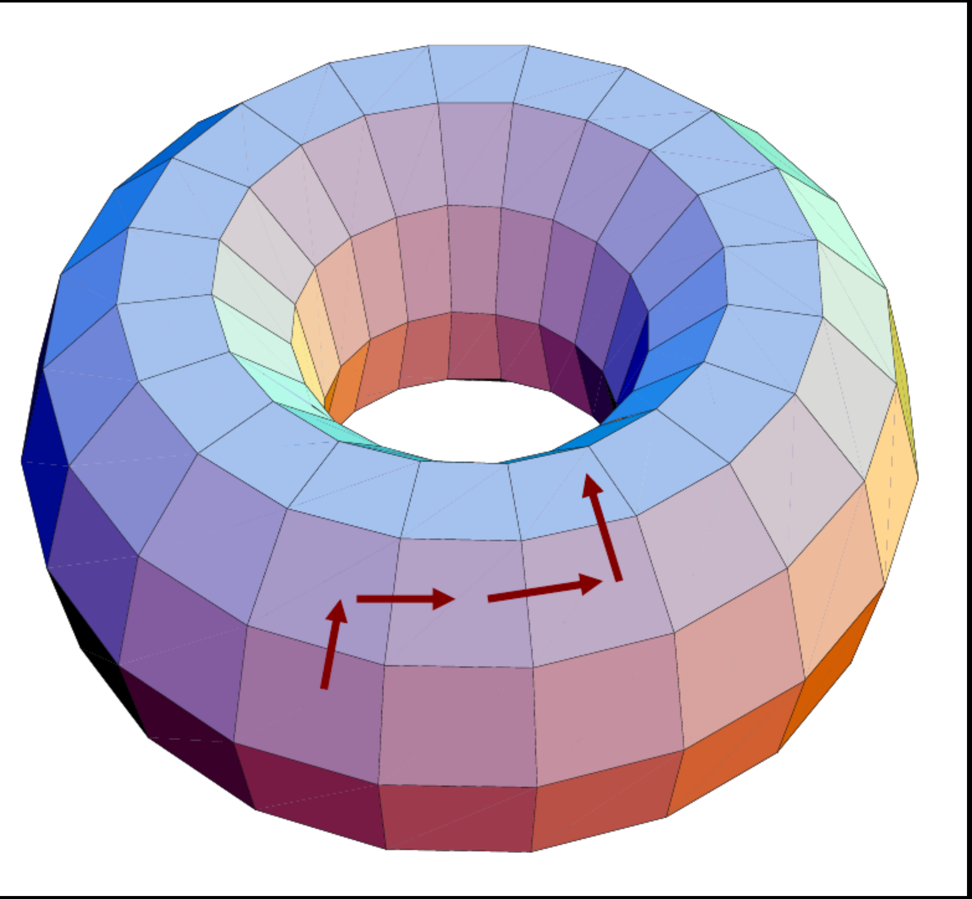
CAN: Partitioning



CAN: Partitioning



CAN: Partitioning

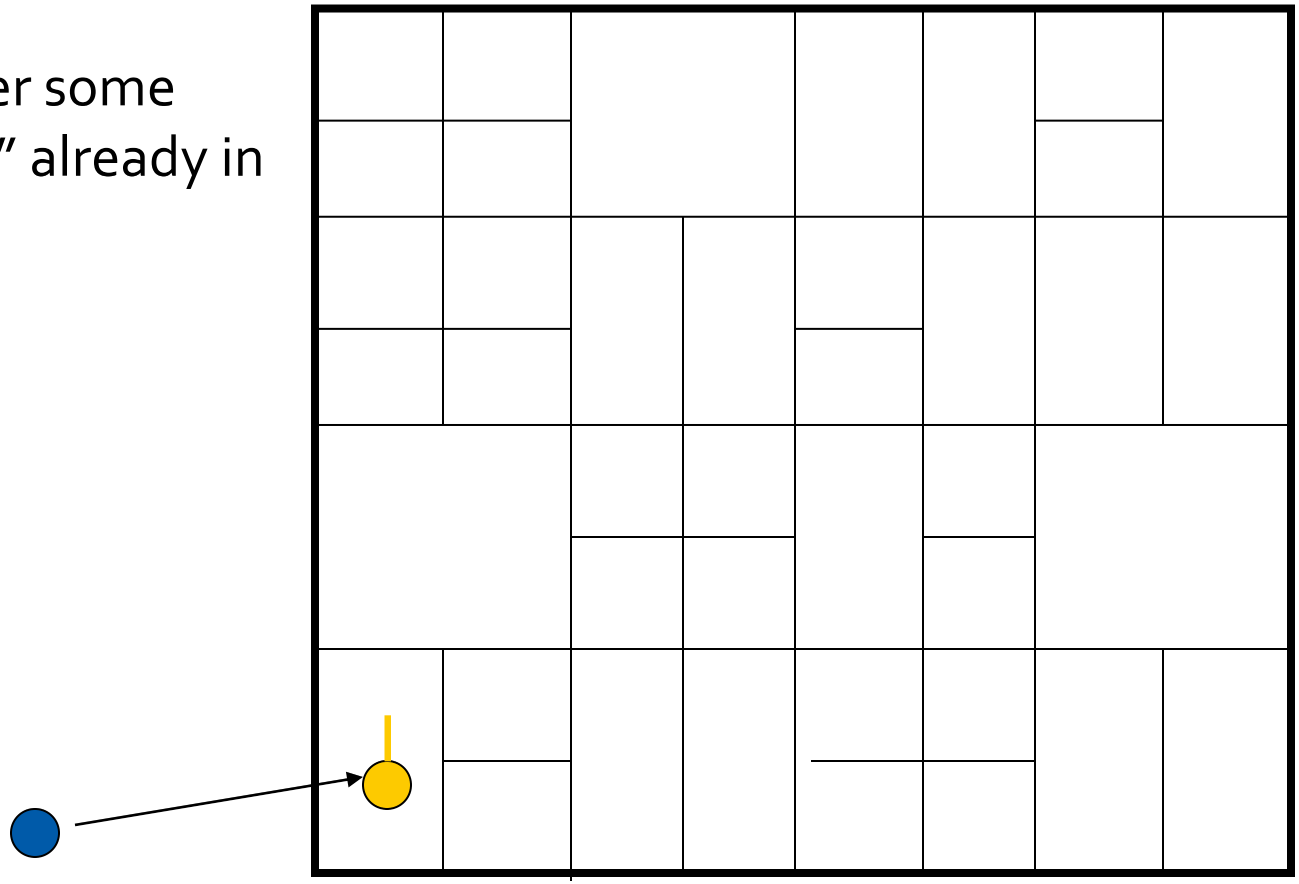


CAN: Examples

- Below examples for:
  - How to join the network
  - How routing tables are managed
  - How to store and retrieve values

CAN: Node Insertion

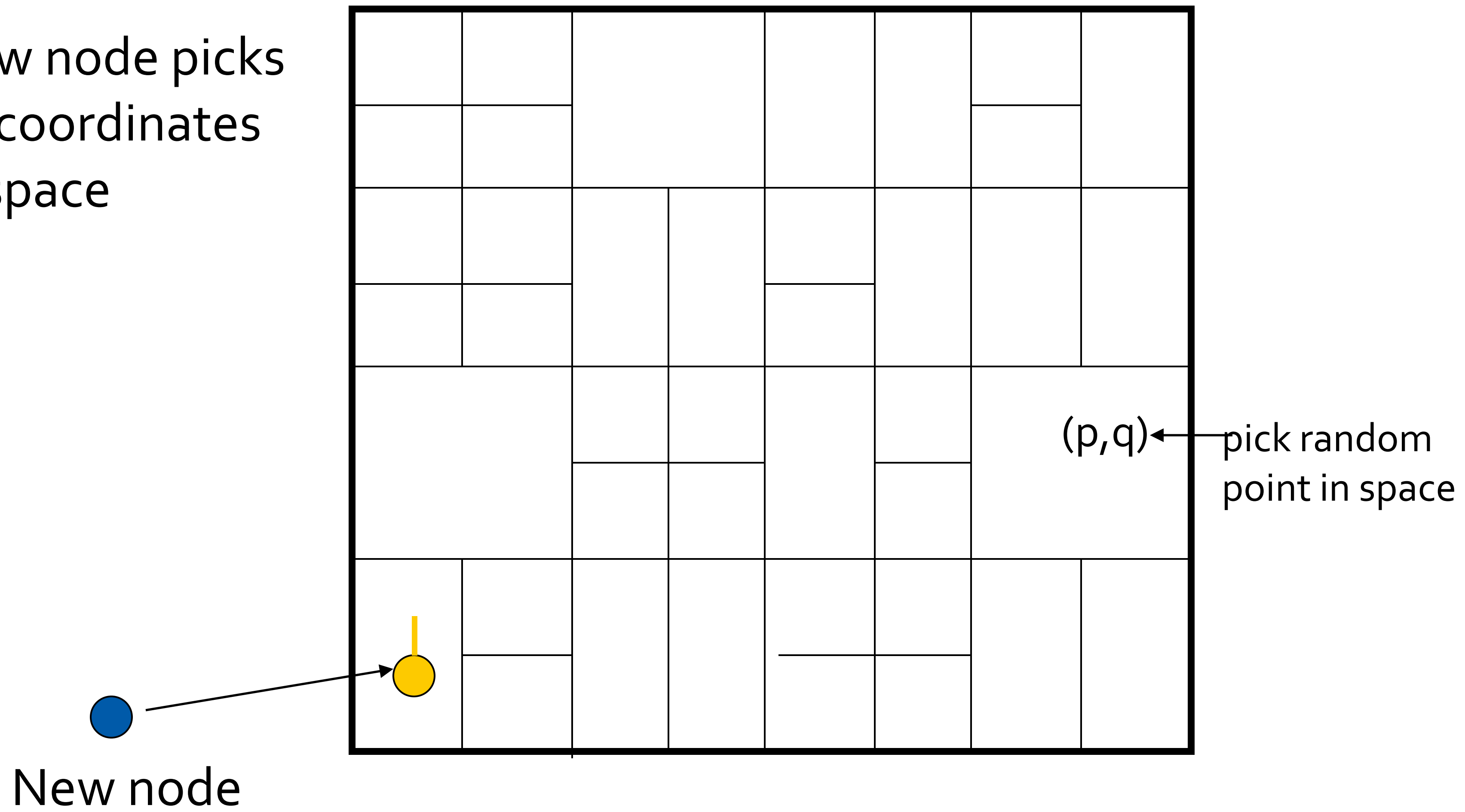
Discover some node "I" already in CAN



New node

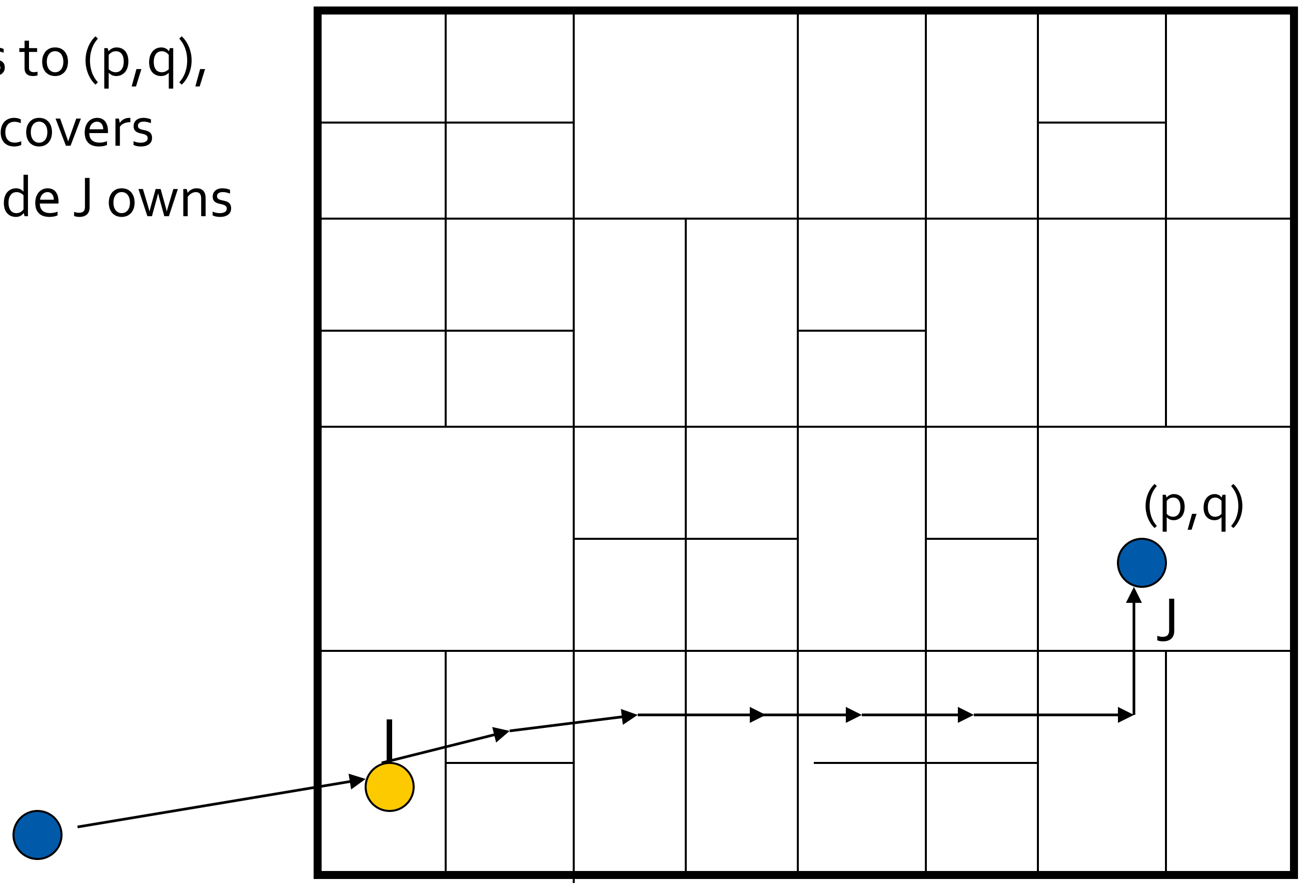
CAN: Node Insertion

New node picks its coordinates in space



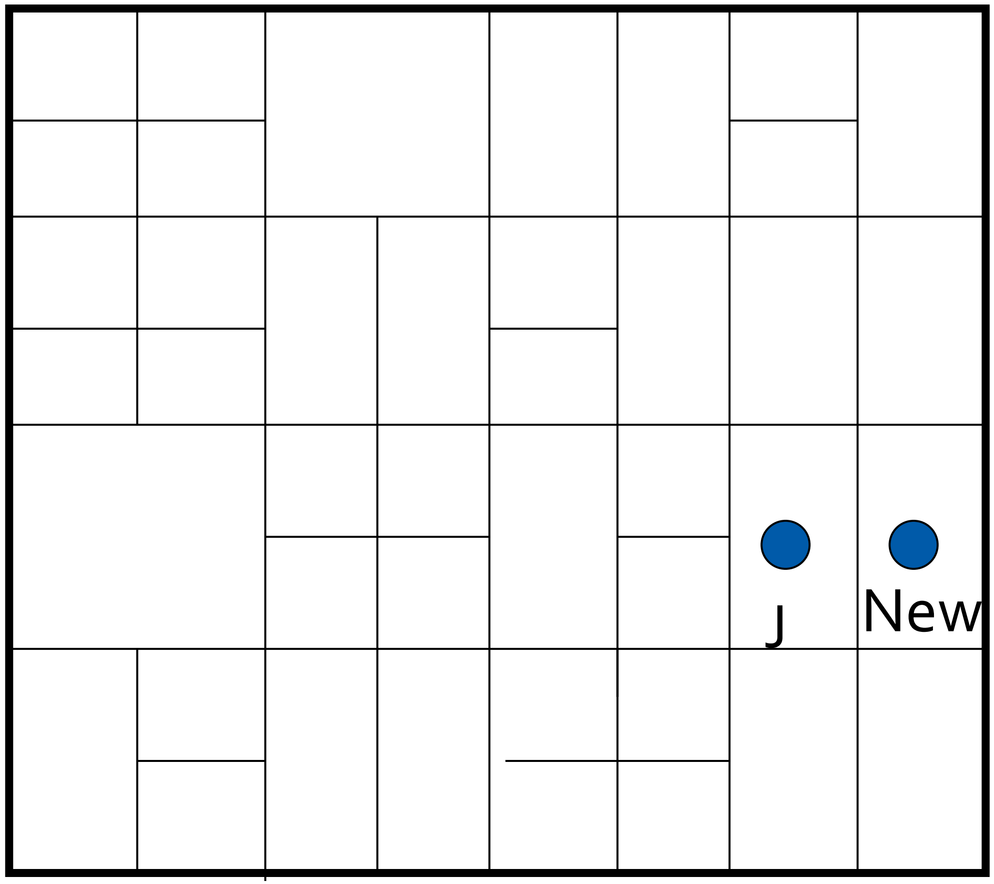
CAN: Node Insertion

I routes to (p,q), and discovers that node J owns (p,q)

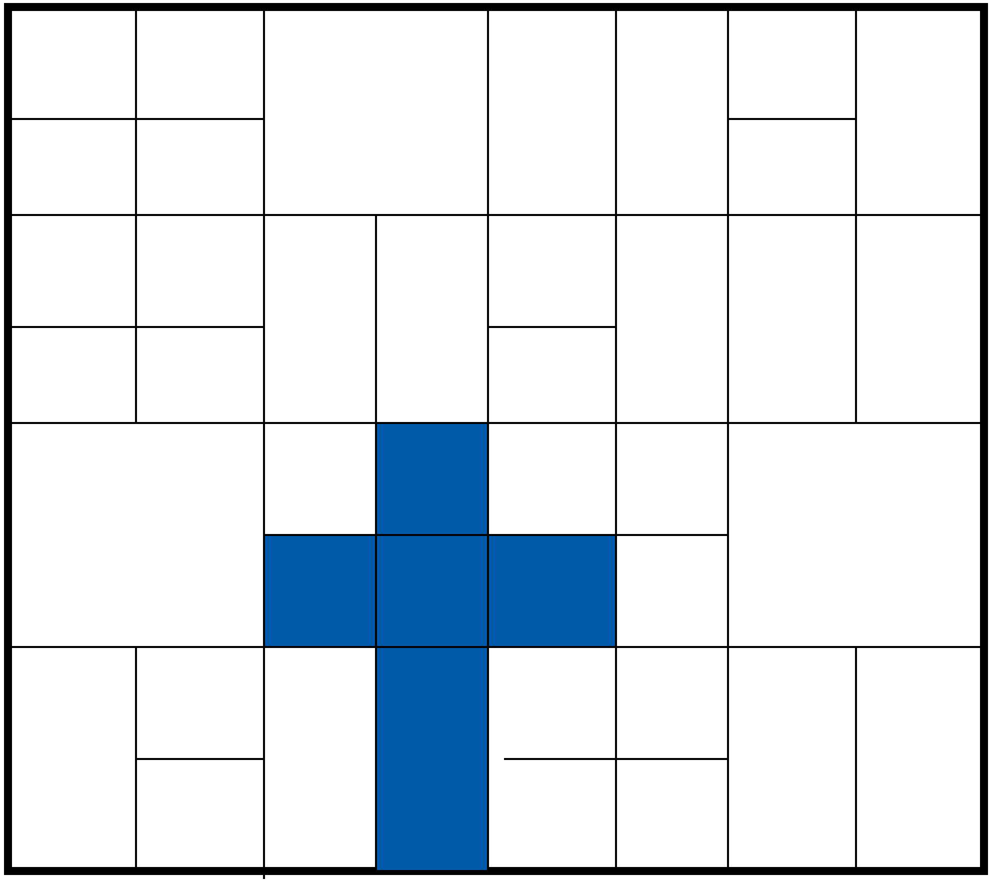


CAN: Node Insertion

Split J's zone in half. New owns one half

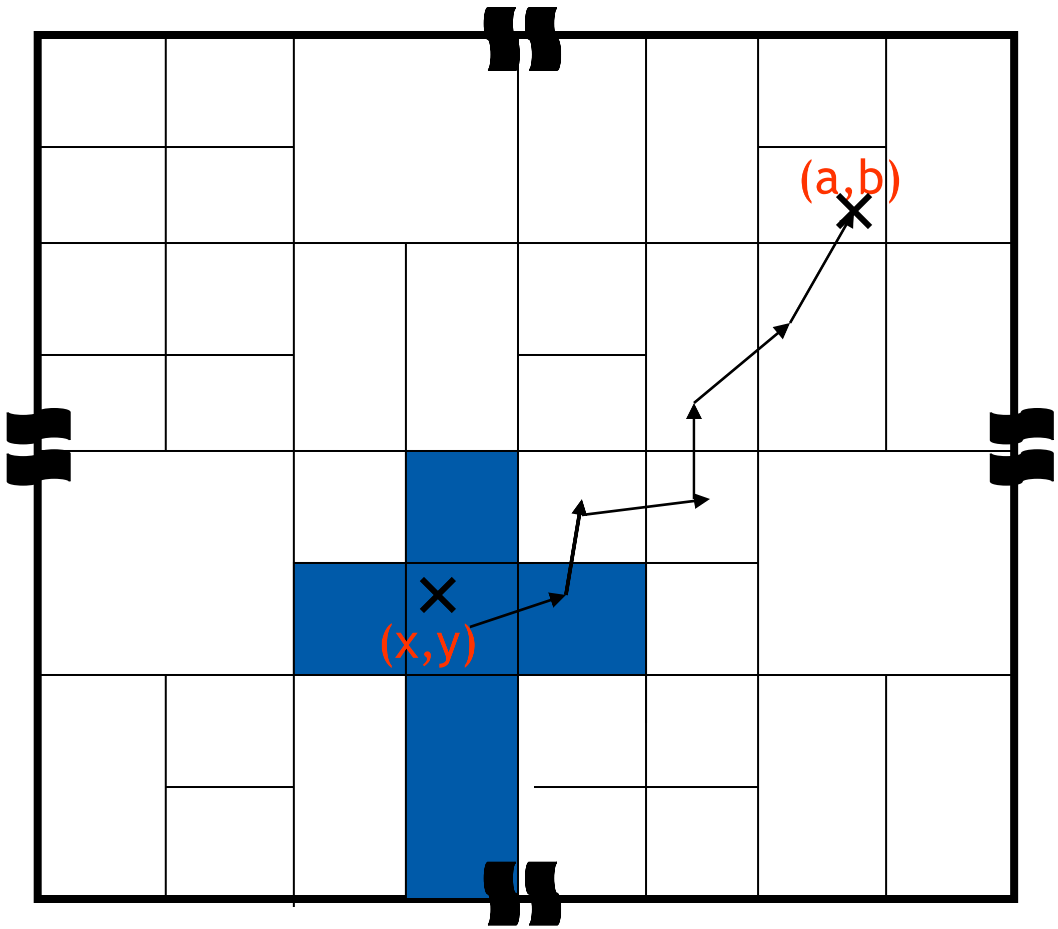


CAN: Routing Table



That's it. ☺

CAN: Routing



Greedy Routing: minimize distance to target

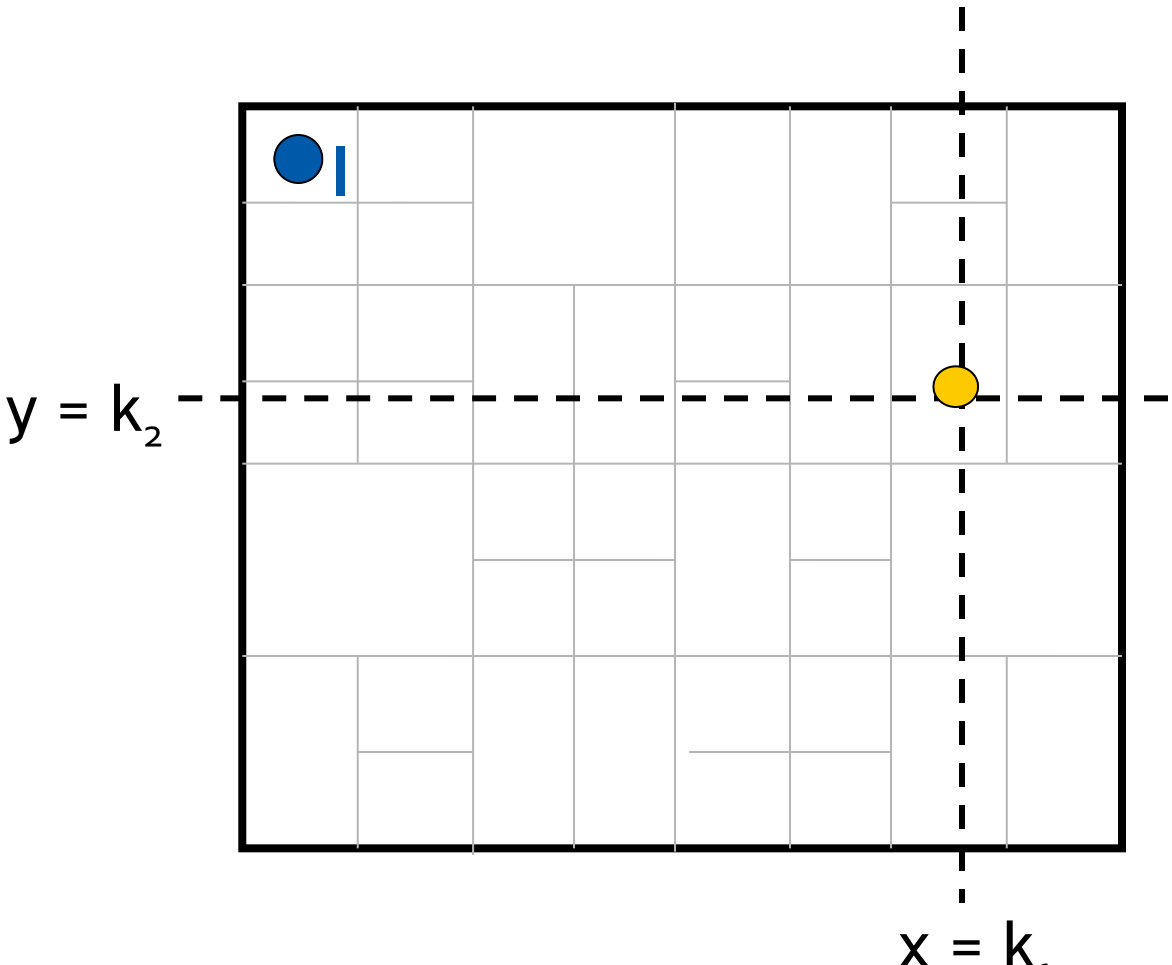
CAN: Storing Values

node I::insert(K,V)

$K = (k_1, k_2)$

$k_1 = h_x(V)$

$k_2 = h_y(V)$



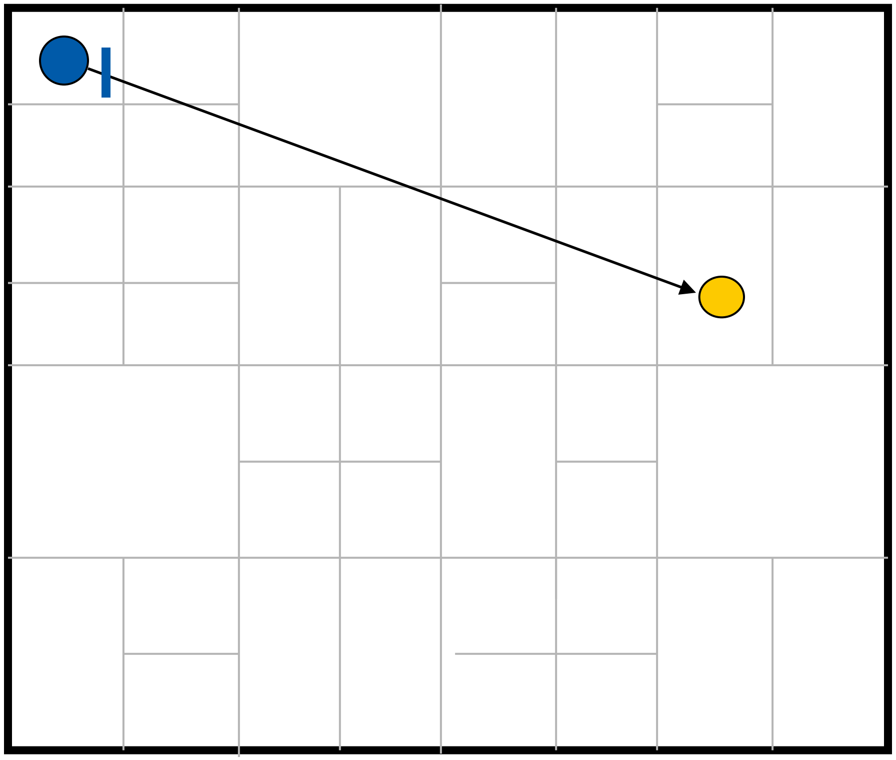
CAN: Storing Values

node I::insert(K,V)

(1) $k_1 = h_x(V)$

$k_2 = h_y(V)$

(2) route(K,V) -> (k_1, k_2)



CAN: Storing Values

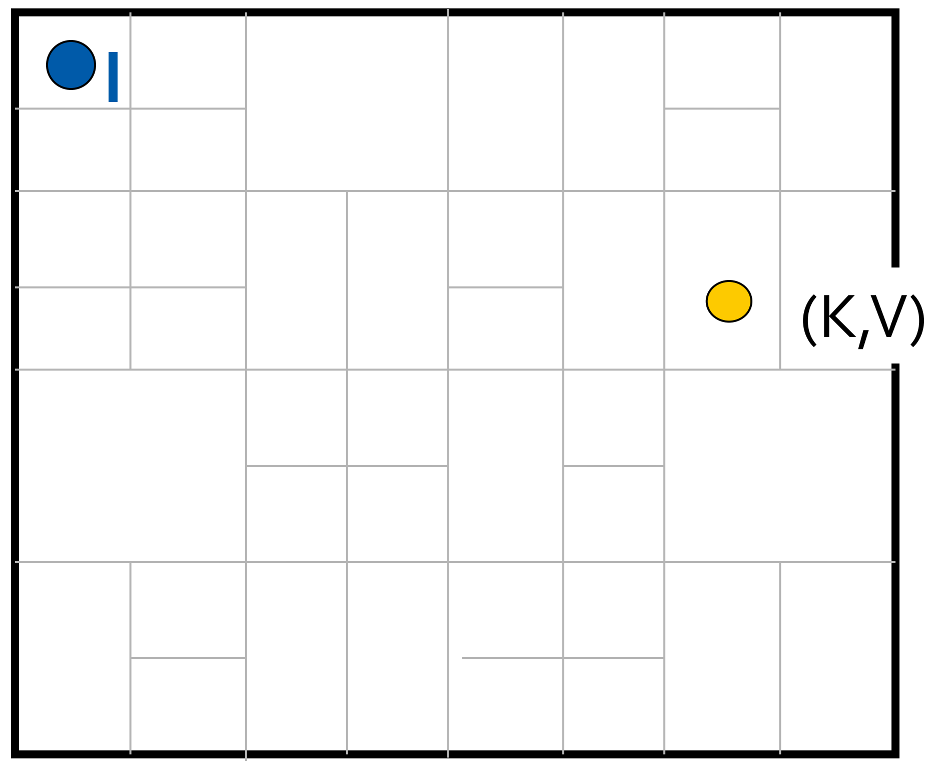
node I::insert(K,V)

(1) $k_1 = h_1(V)$

$k_d = h_d(V)$

(2) route(K,V) -> (k_1, k_2)

(3) (k_1, k_2) stores (K,V)



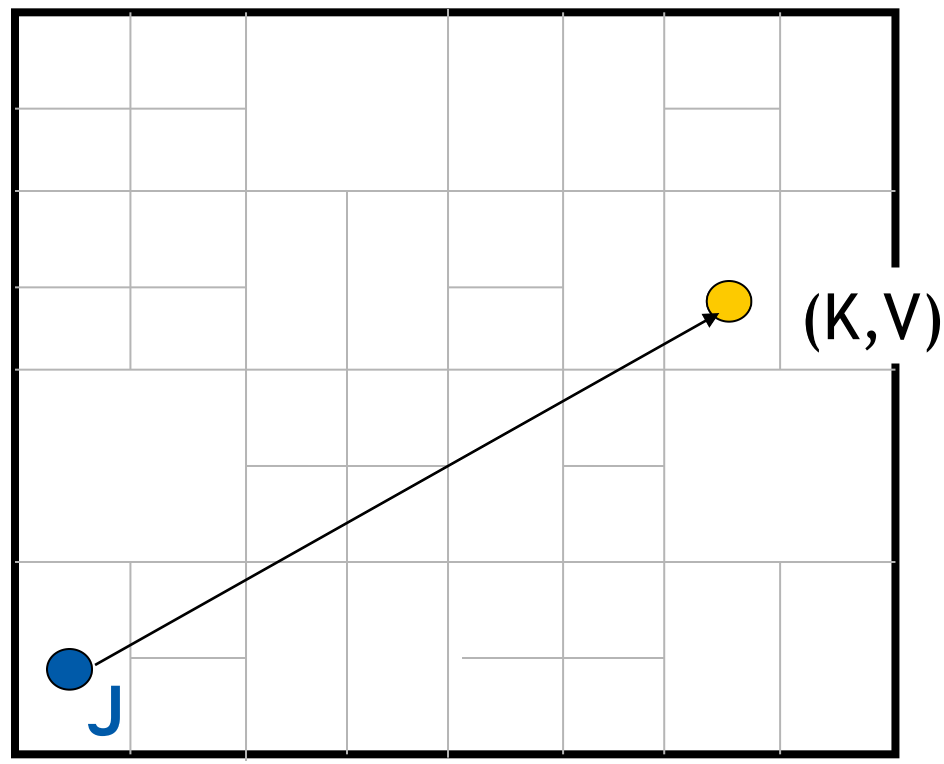
CAN: Retrieving Values

node J::retrieve(K)

(1) $k_1 = h_1(V)$

$k_d = h_d(V)$

(2) route "retrieve(K)" to (k_1, k_2)



CAN: Improvements

- Possible to increase number of dimensions *d*
  - Small increase in routing table size
  - Shorter routing path, more neighbors for fault tolerance
- Multiple realities (= coordinate spaces)
  - Use more hash functions
  - Similar properties as increased dimensions (yet, not the same!)
- Routing weighted by round-trip times
  - Take into account network topology
  - Forward to the "best" neighbor

CAN: More Improvements

- Use well-known landmark servers (e.g., DNS roots)
  - Nodes join CAN in different areas, depending on distance to landmarks
    - Pick points “near” landmark
  - Idea: Geographically close nodes see same landmarks
- Uniform partitioning
  - New node splits the largest zone in the neighborhood instead of the zone of the responsible node

CAN: Performance

- State information at node *O(d)*
  - Number of dimensions is *d*
  - Need two neighbors in all coordinate axis
  - Independent of the number of nodes!
- Routing takes $O(dn^{1/d})$ hops
  - Network has *n* nodes
  - Multiple dimensions (and realities) improve this
  - Routing improved by multiple dimensions
- Multiple realities mainly improve availability and fault tolerance