Peer-to-Peer Networks

Chapter 3: Networks, Searching and Distributed Hash Tables

(Part 2)

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?

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…*

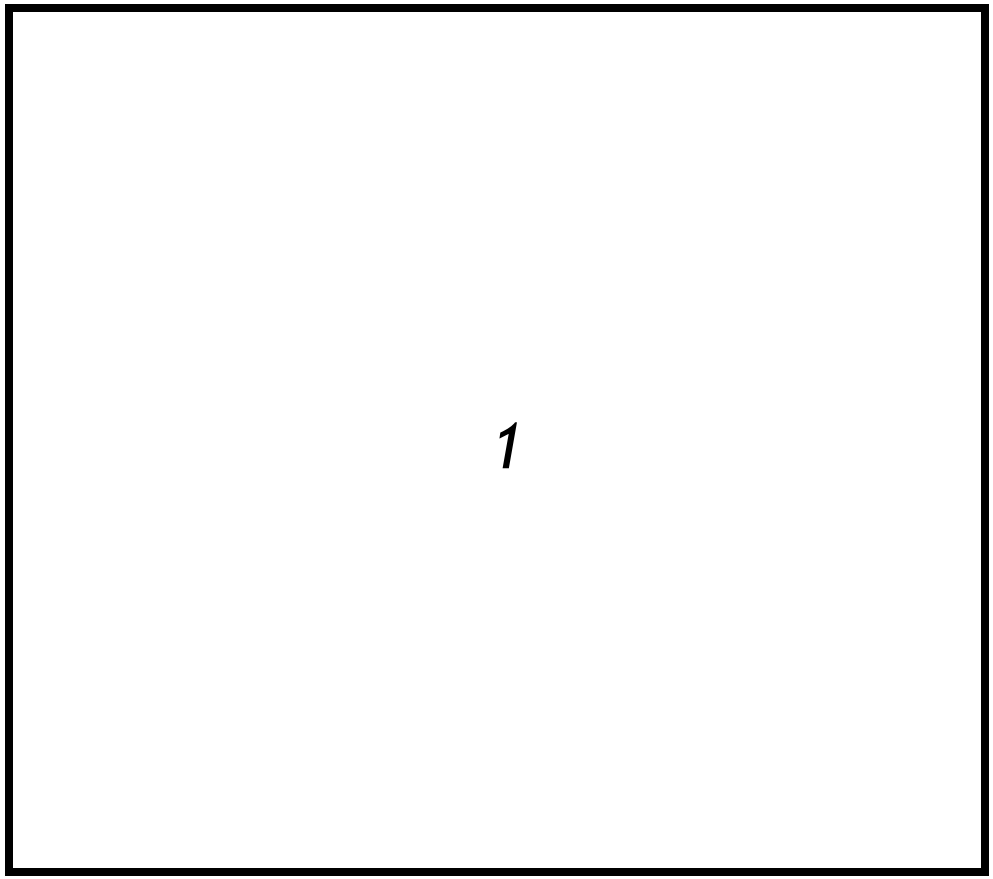
CAN: Basics

- *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*

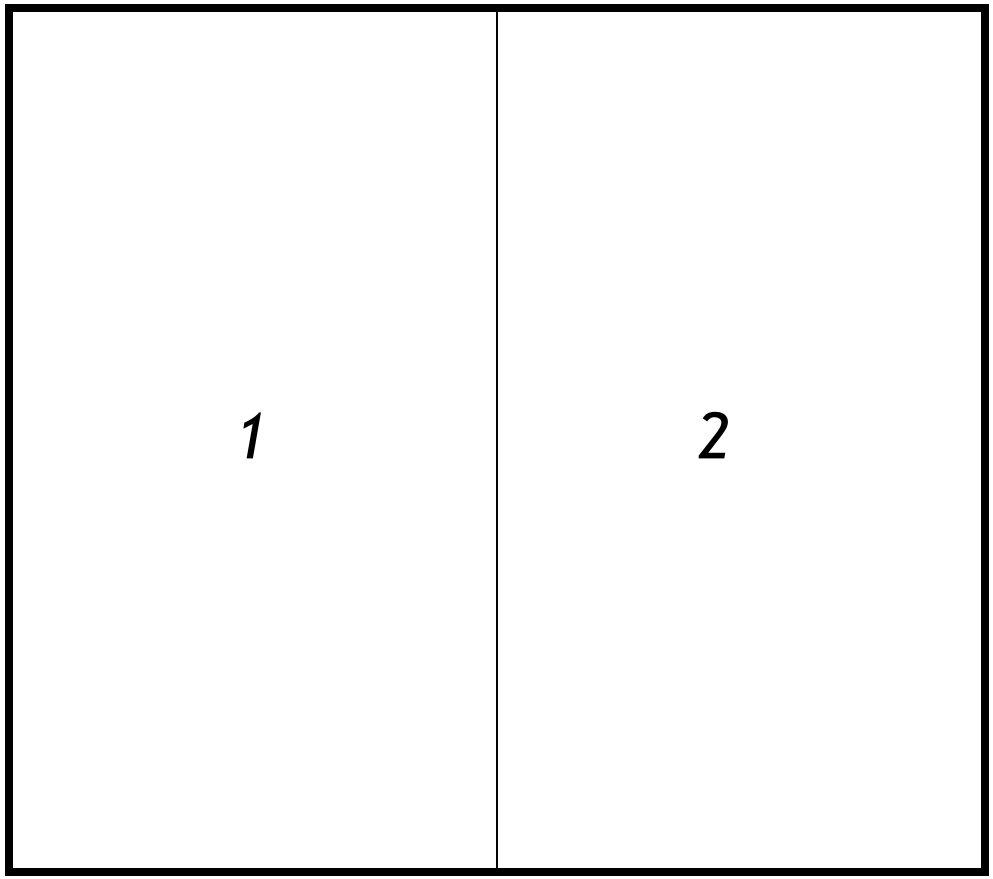
- *Abstractions provided by CAN:*
  - `store` *data at points in the space*
  - `route` *from one point to another*

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

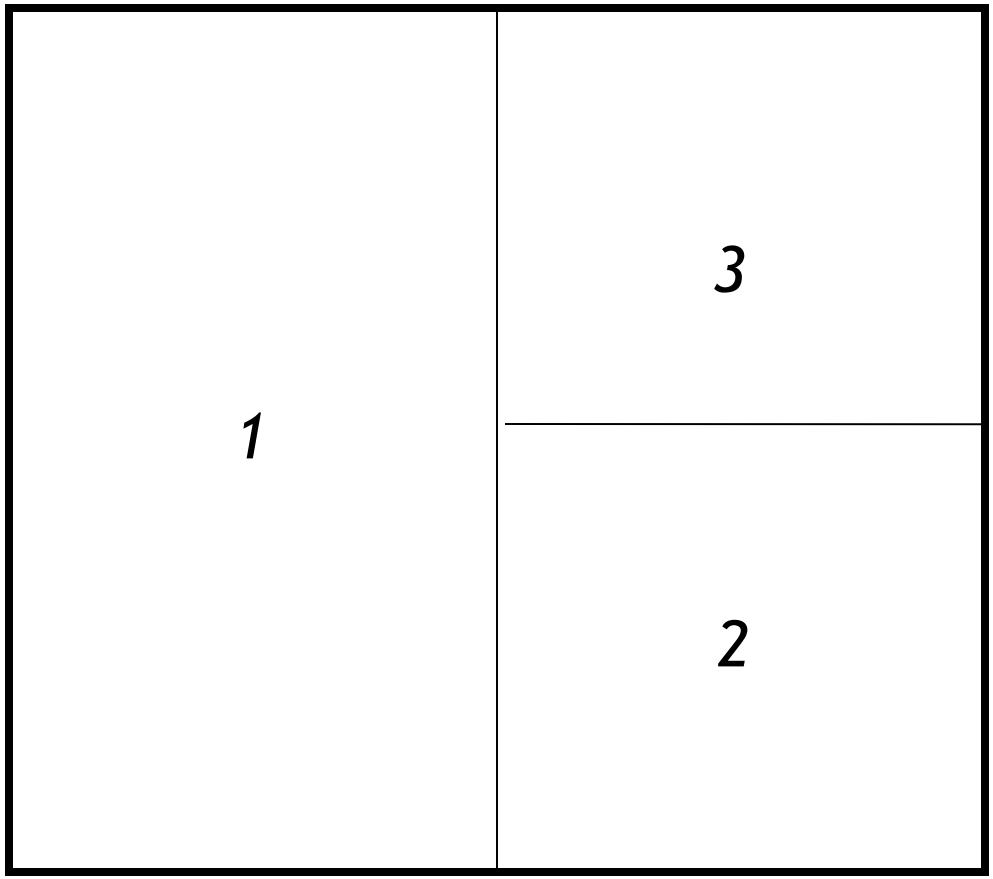
CAN: Partitioning



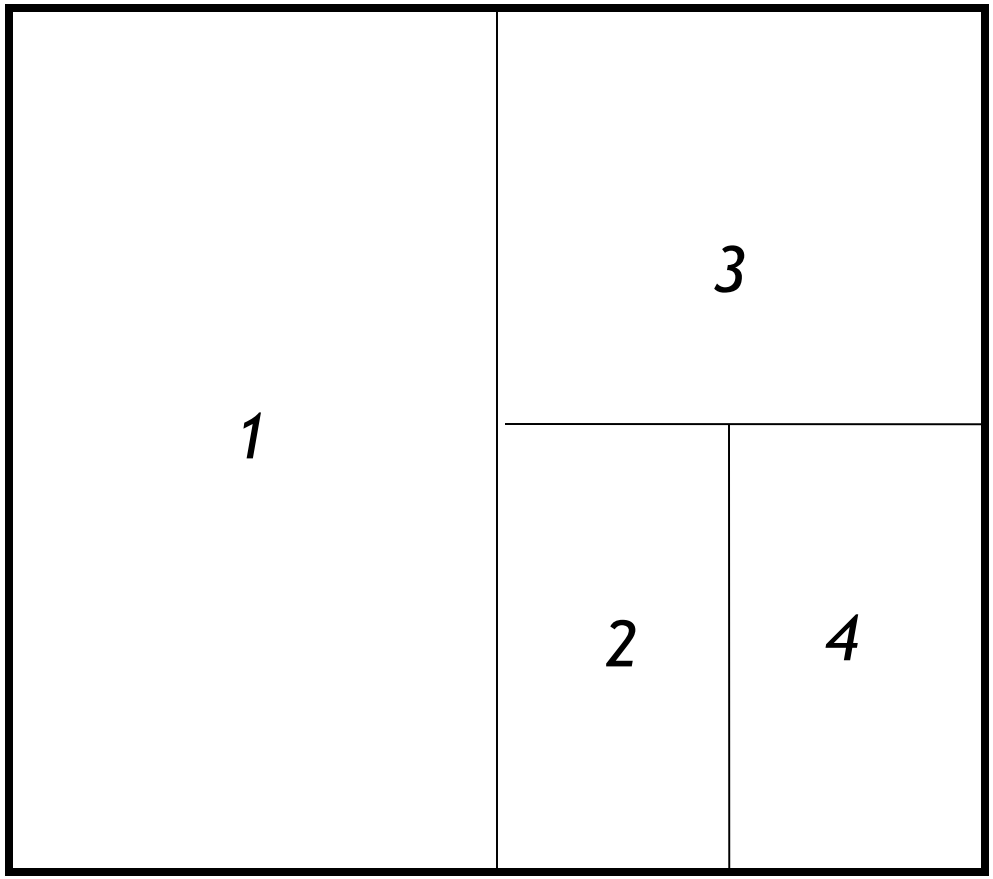
CAN: Partitioning



CAN: Partitioning

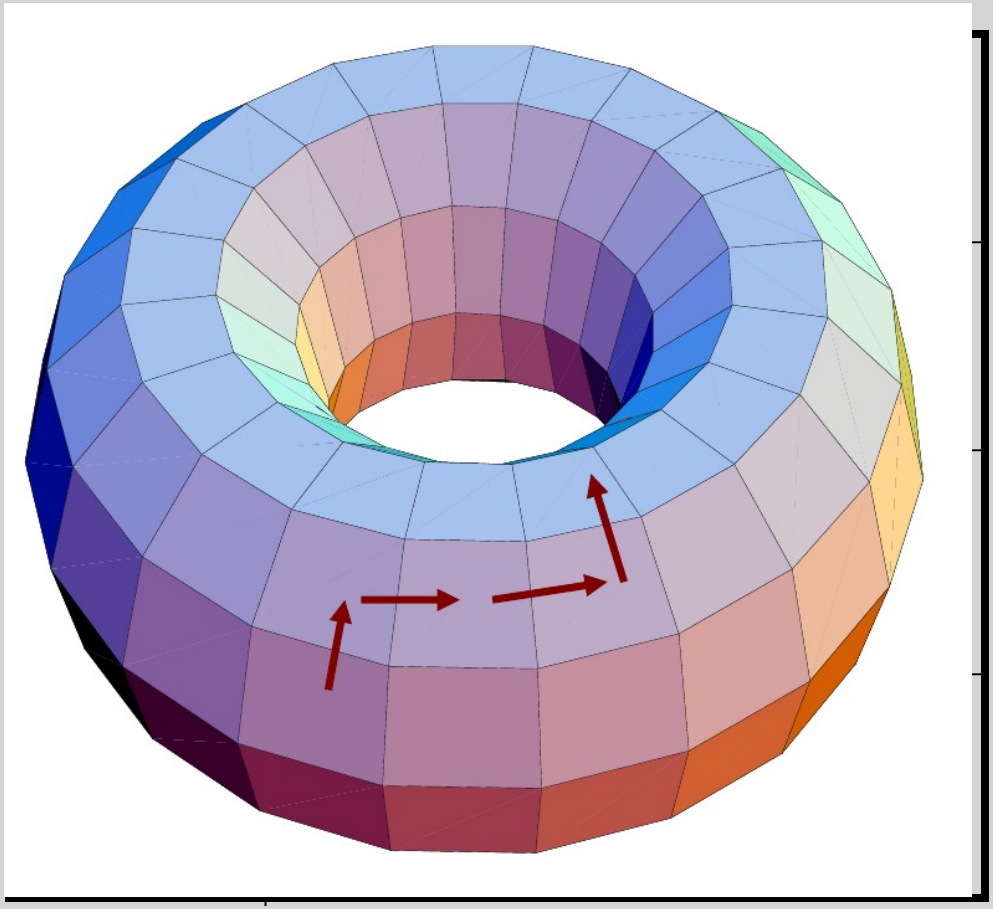


CAN: Partitioning



CAN: Partitioning

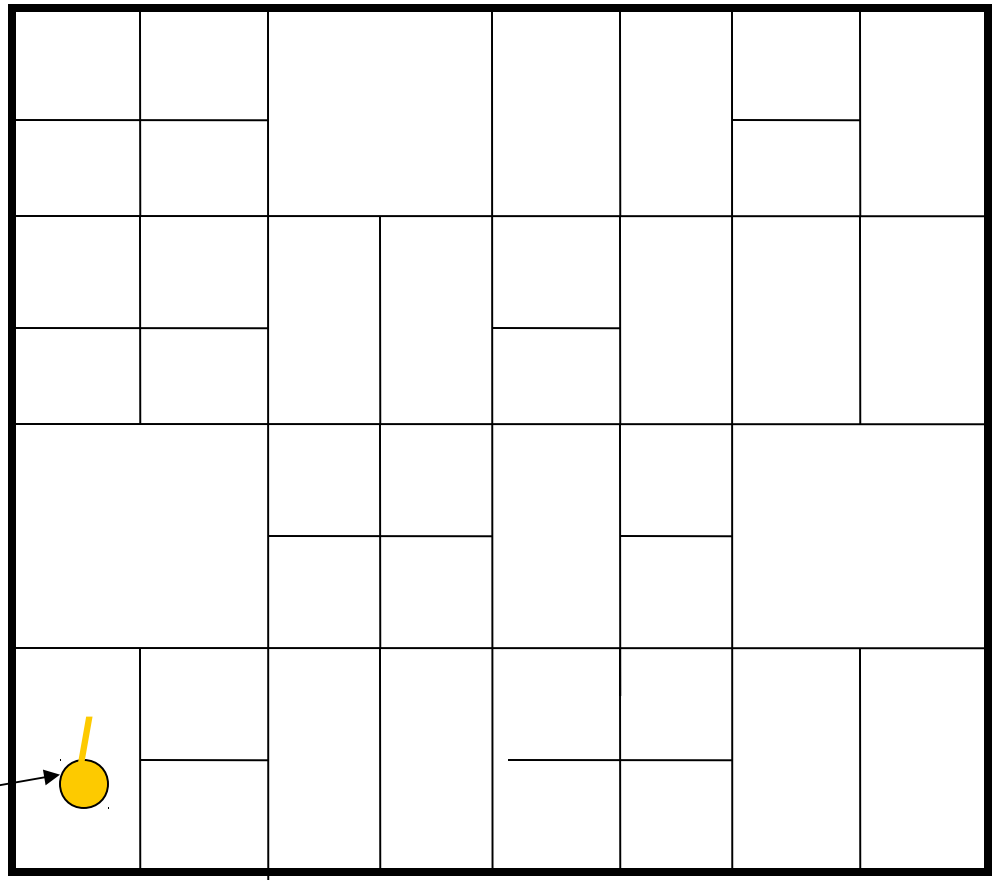
- *CAN forms a d-dimensional torus*



CAN: Examples

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

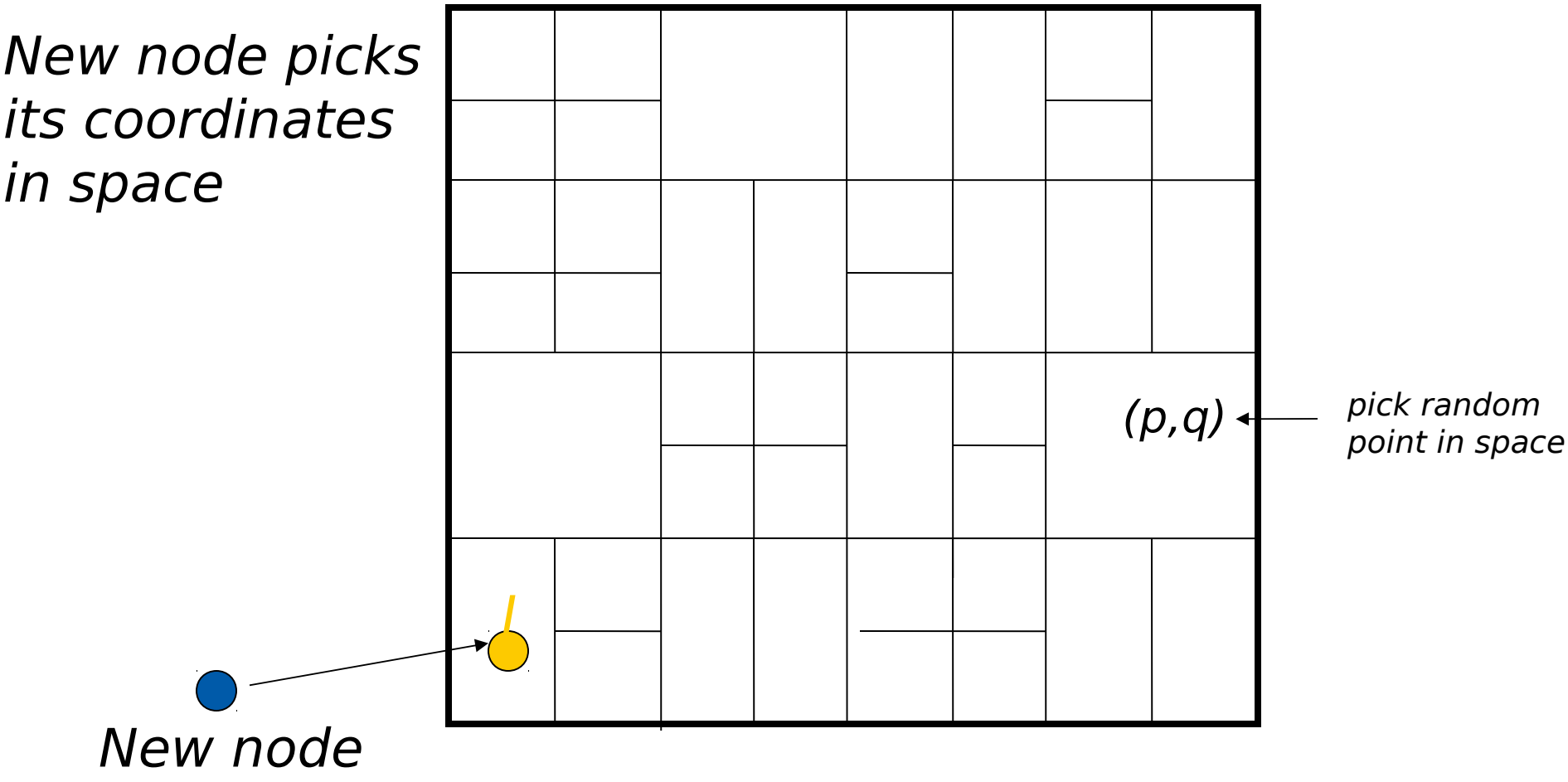
CAN: Node Insertion



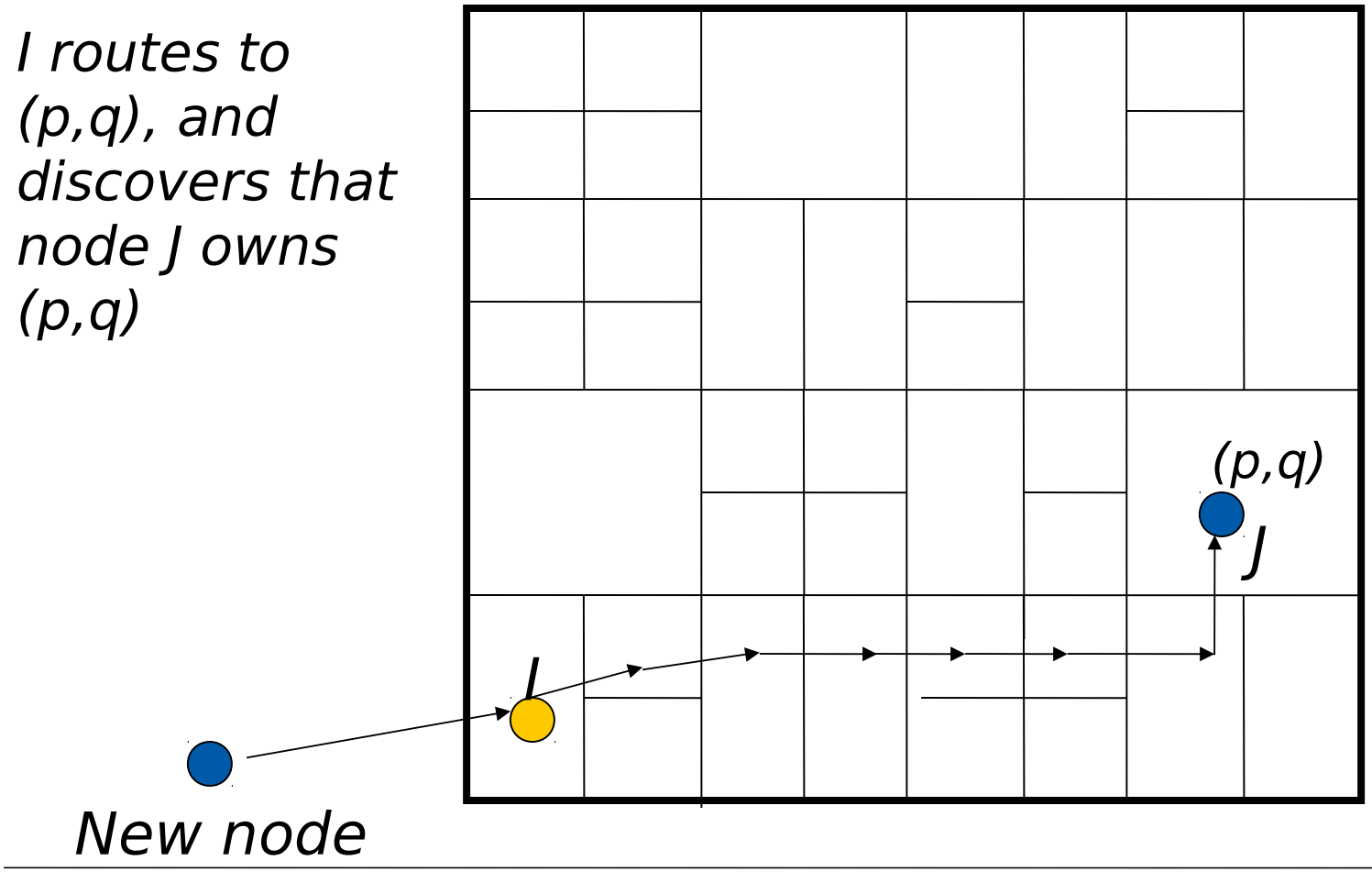
Discover some node "I" already in CAN

New node

CAN: Node Insertion

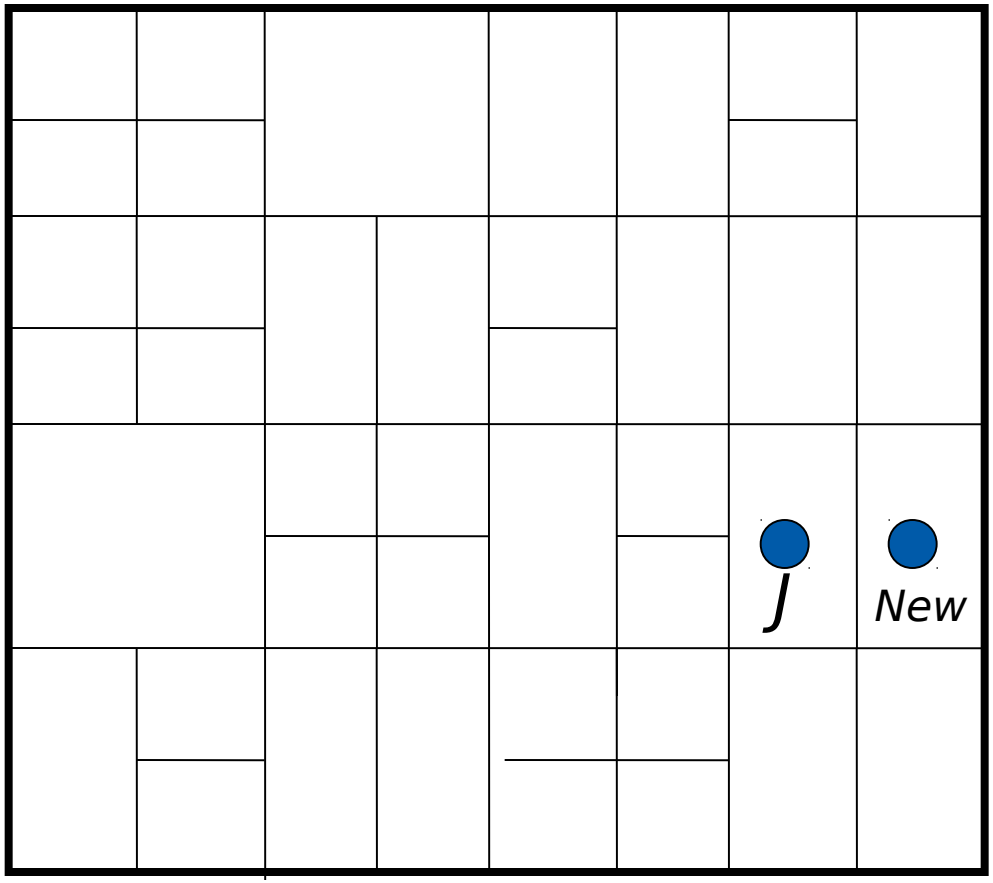


CAN: Node Insertion

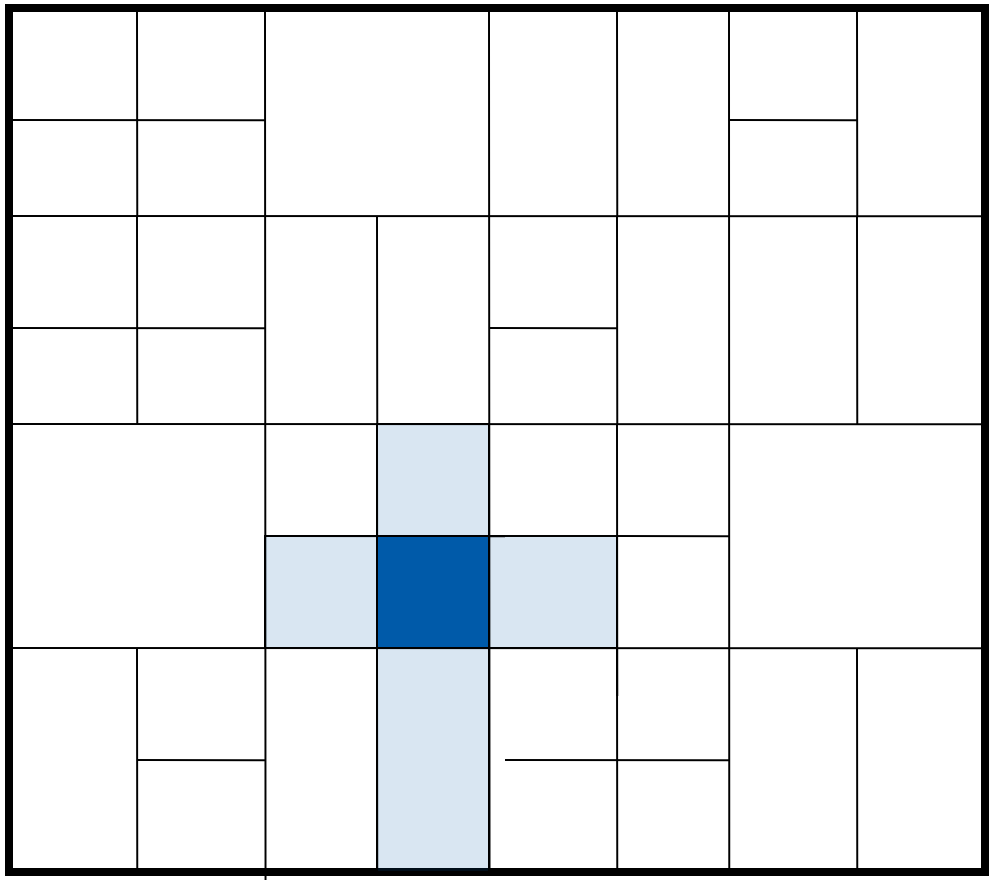


CAN: Node Insertion

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

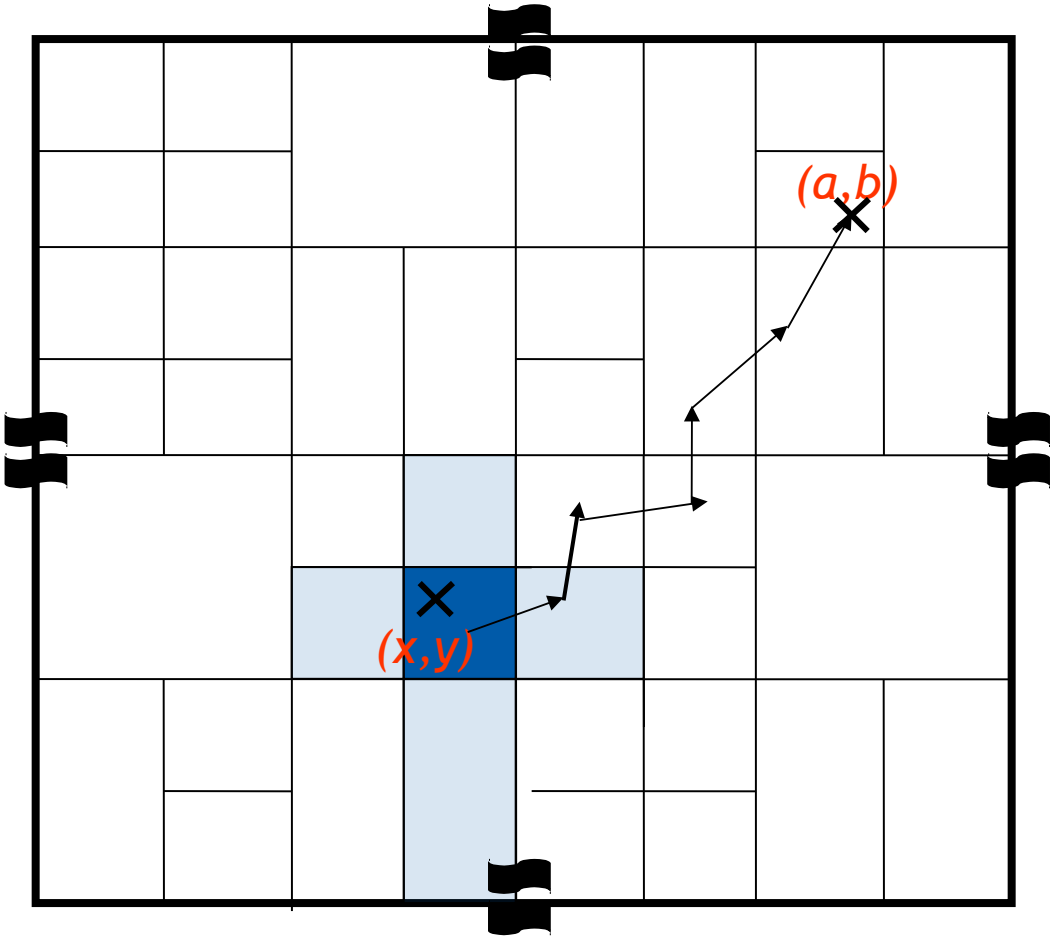


CAN: Routing Table



That‘s it. ☺

CAN: Routing



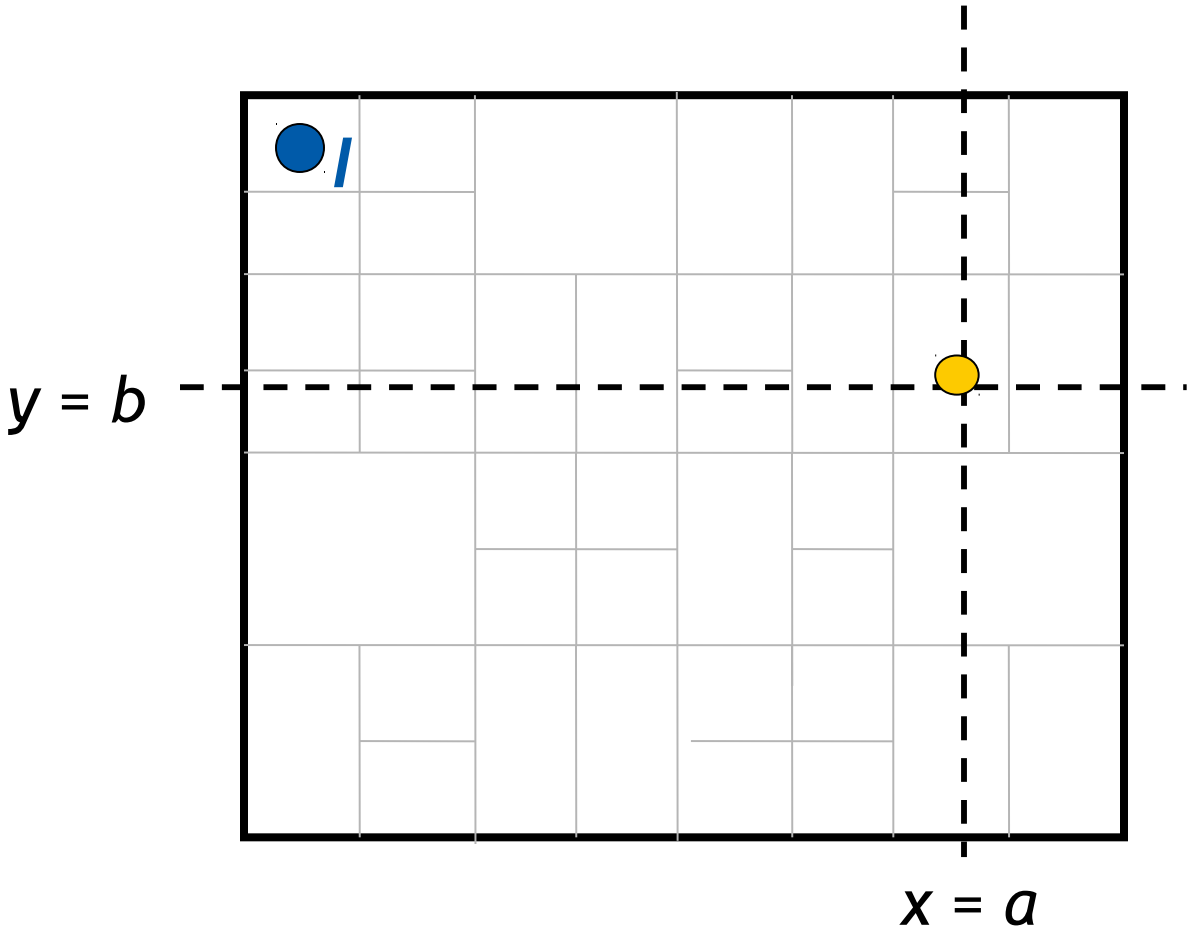
Greedy Routing: minimize distance to target

CAN: Storing Values

node I::insert(K,V)

$a = h_x(K)$

$b = h_y(K)$



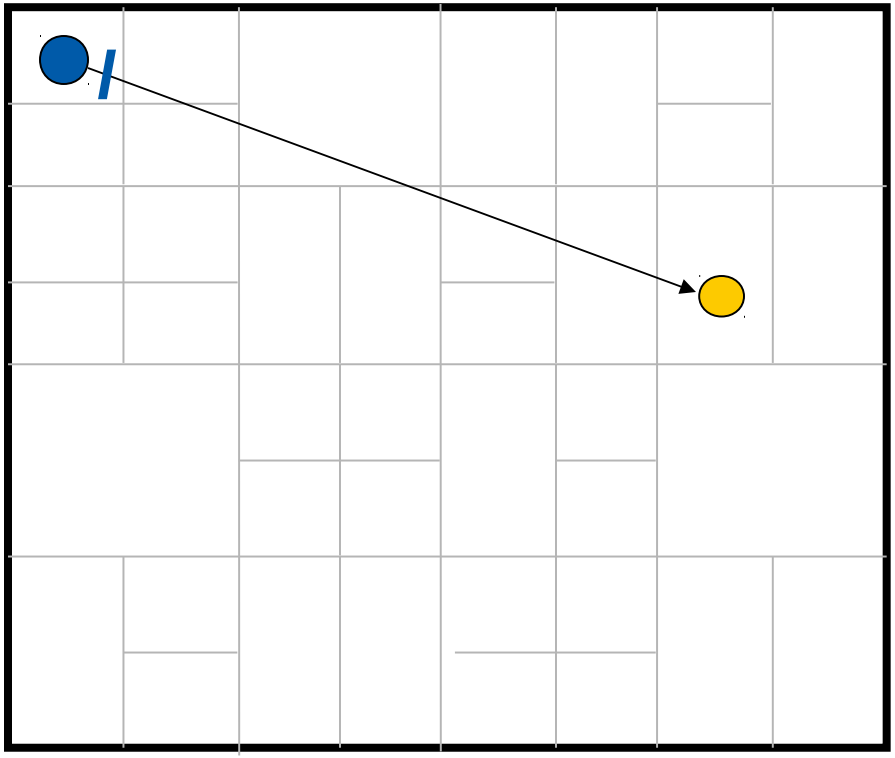
CAN: Storing Values

node I::insert(K,V)

(1) a = h_1(K)

b = h_d(K)

(2) route(K,V) -> (a,b)



CAN: Storing Values

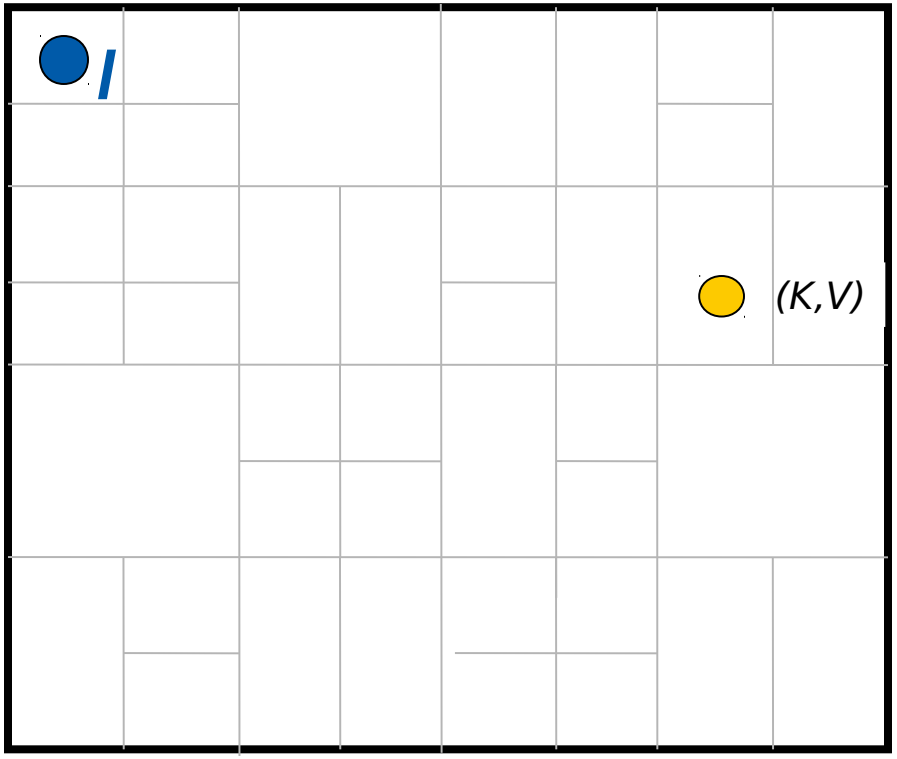
node I::insert(K,V)

(1) a = h_1(K)

b = h_d(K)

(2) route(K,V) -> (a,b)

(3) (a,b) stores (K,V)



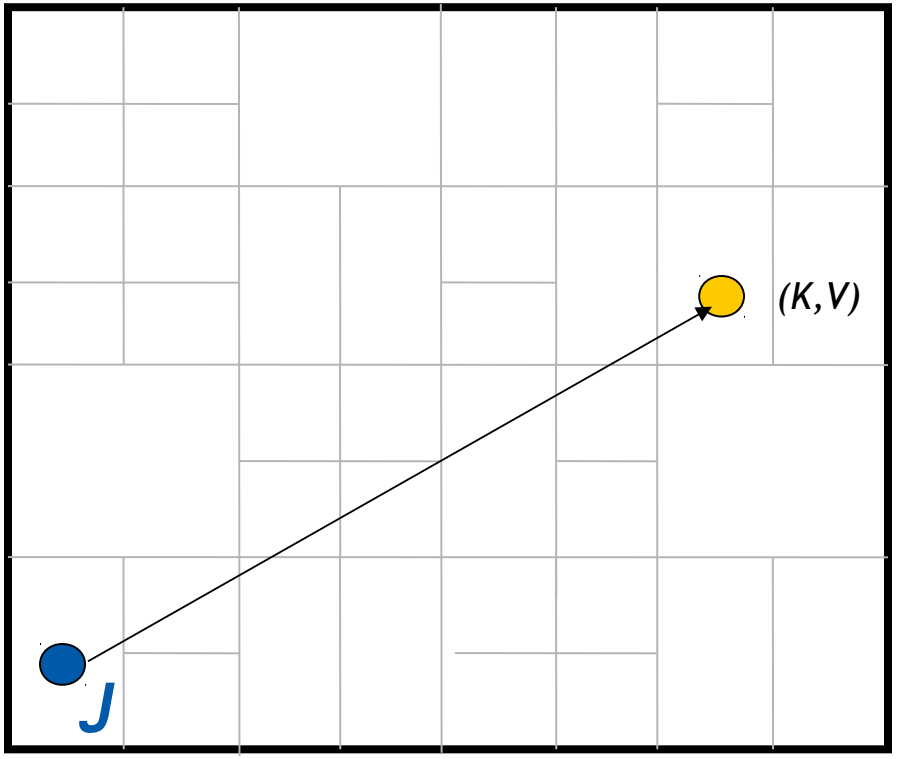
CAN: Retrieving Values

node J::retrieve(K)

(1) $a = h_1(K)$

$b = h_d(K)$

(2) route "retrieve(K)" to (a,b)



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*

Tapestry

- *Tapestry developed at UC Berkeley(!)*
  - *Different group from CAN developers*
- *Tapestry developed in 2000, but published in 2004*
  - *Originally only as technical report, 2004 as journal article*
- *Many follow-up projects on Tapestry*
  - *Example: OceanStore*

- *Tapestry based on work by Plaxton et al.*
- *Plaxton network has also been used by **Pastry***
- *Pastry was developed at Microsoft Research and Rice University*
  - *Difference between Pastry and Tapestry minimal*
  - *Tapestry and Pastry add dynamics and fault tolerance to Plaxton network*

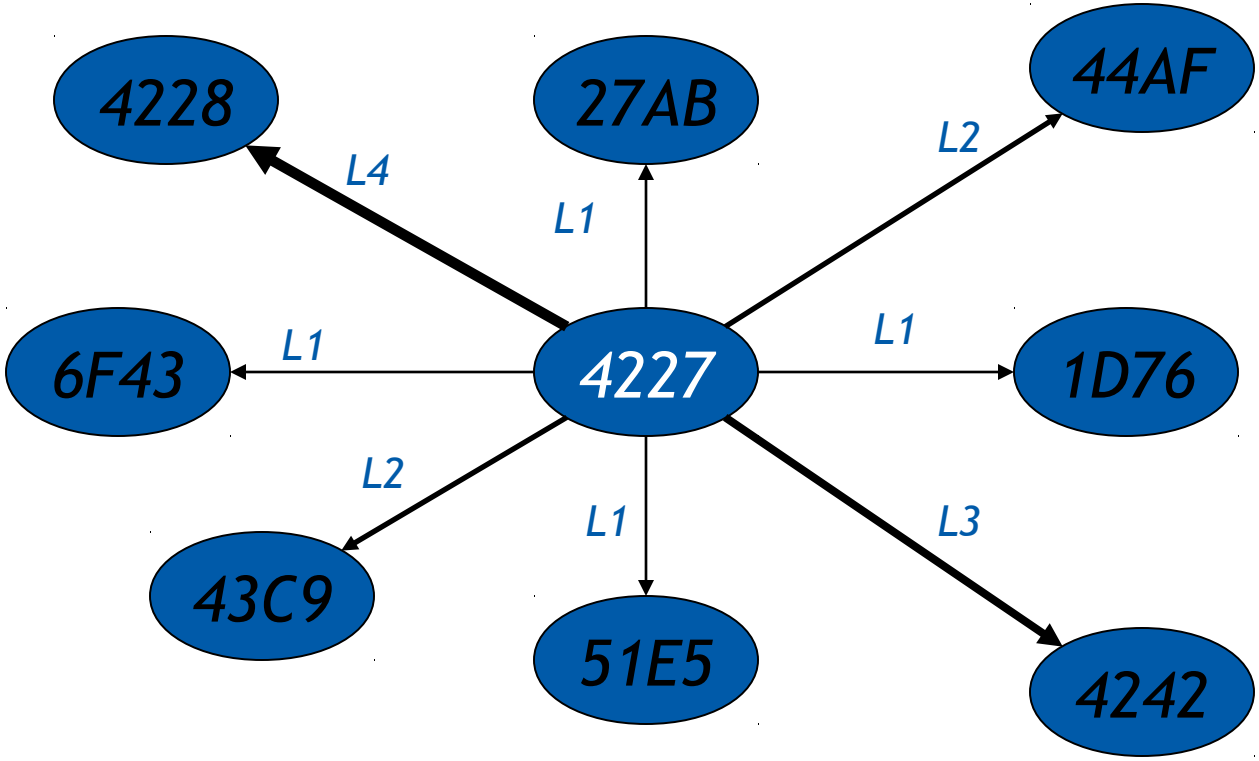
Tapestry: Plaxton Network

- *Plaxton network (or Plaxton mesh) based on prefix routing (similar to IP address allocation)*
  - *Prefix and postfix are functionally identical*
  - *Tapestry originally postfix, now prefix…*
- *Node ID and object ID hashed with SHA-1*
  - *Expressed as hexadecimal (base 16) numbers (40 digits)*
  - *Base is very important, here we use base 16*

- *Each node has a neighbor map with multiple levels*
  - *Each level represents a matching prefix up to digit position in ID*
  - *A given level has number of entries equal to the base of ID*
  - *i^{th} entry in j^{th} level is closest node which starts prefix(N,j-1)+"i"*
  - *Example: 9th entry of 4th level for node 325AE is the closest node with ID beginning with 3259*

Tapestry: Routing Mesh

- *(Partial) routing mesh for a single node 4227*
- *Neighbors on higher levels match more digits*

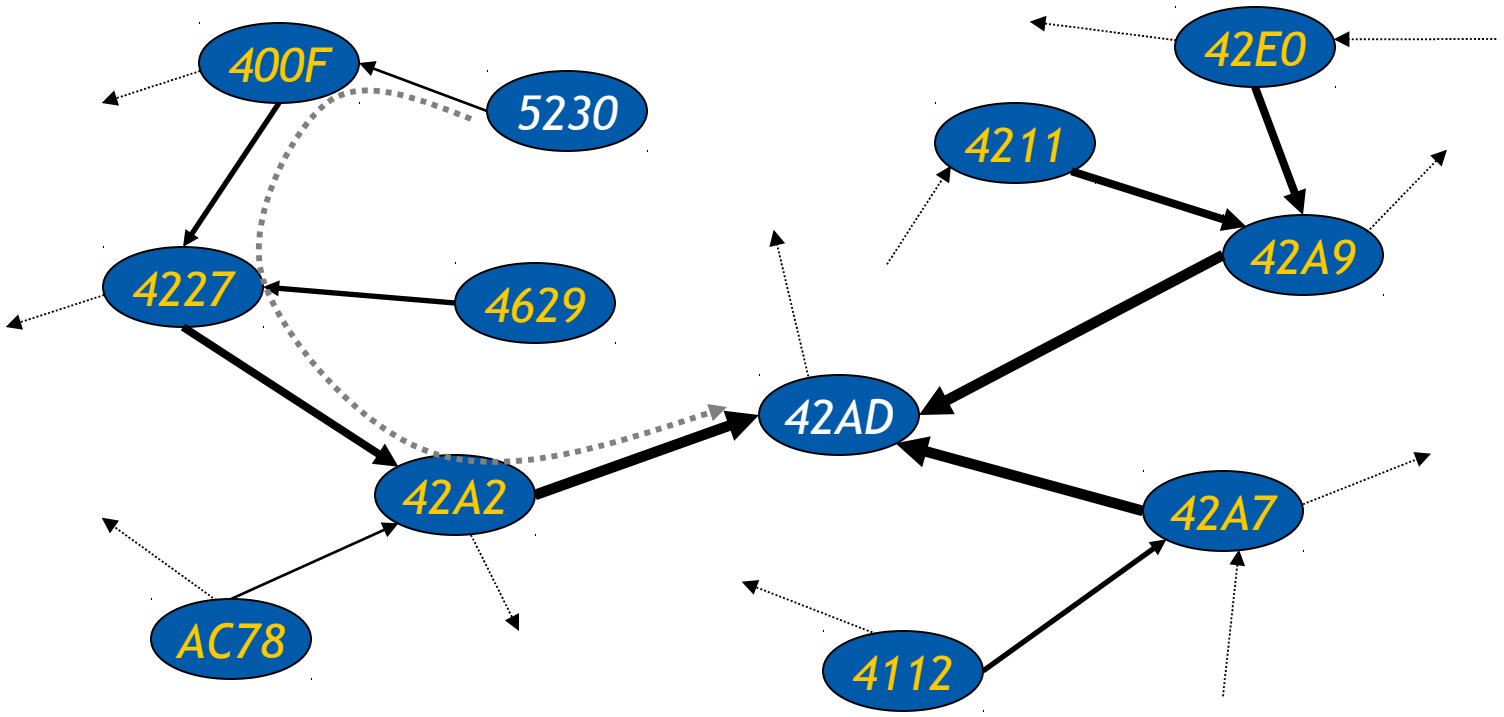


Tapestry: Neighbor Map for 4227

Level	1	2	3	4	5	6	8	A
1	1D76	27AB			51E5	6F43		
2			43C9	44AF				
3								42A2
4							4228	

- *There are actually 16 columns in the map (base 16)*
- *Normally more (most?) entries would be filled*

Tapestry: Routing Example



- *Route message from 5230 to 42AD*
- *Always route to node closer to target*
  - *At n^{th} hop, look at $n+1^{st}$ level in neighbor map --> "always" one digit more*
- *Not all nodes and links are shown*

Tapestry: Properties

- *Node responsible for objects which have same ID*
  - *Unlikely to find such node for every object*
  - *Node responsible also for "nearby" objects (surrogate routing, see below)*
- *Object publishing:*
  - *Responsible nodes store only pointers*
    - *Multiple copies of object possible (replica!)*
    - *Each copy must publish itself*
  - *Pointers cached along the publish path*
  - *Queries routed towards responsible node*
  - *Queries "often" hit cached pointers*
    - *Queries for same object go (soon) to same nodes*
- *Note: Tapestry focuses on storing objects*
  - *Chord and CAN focus on values, but in practice no difference*