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

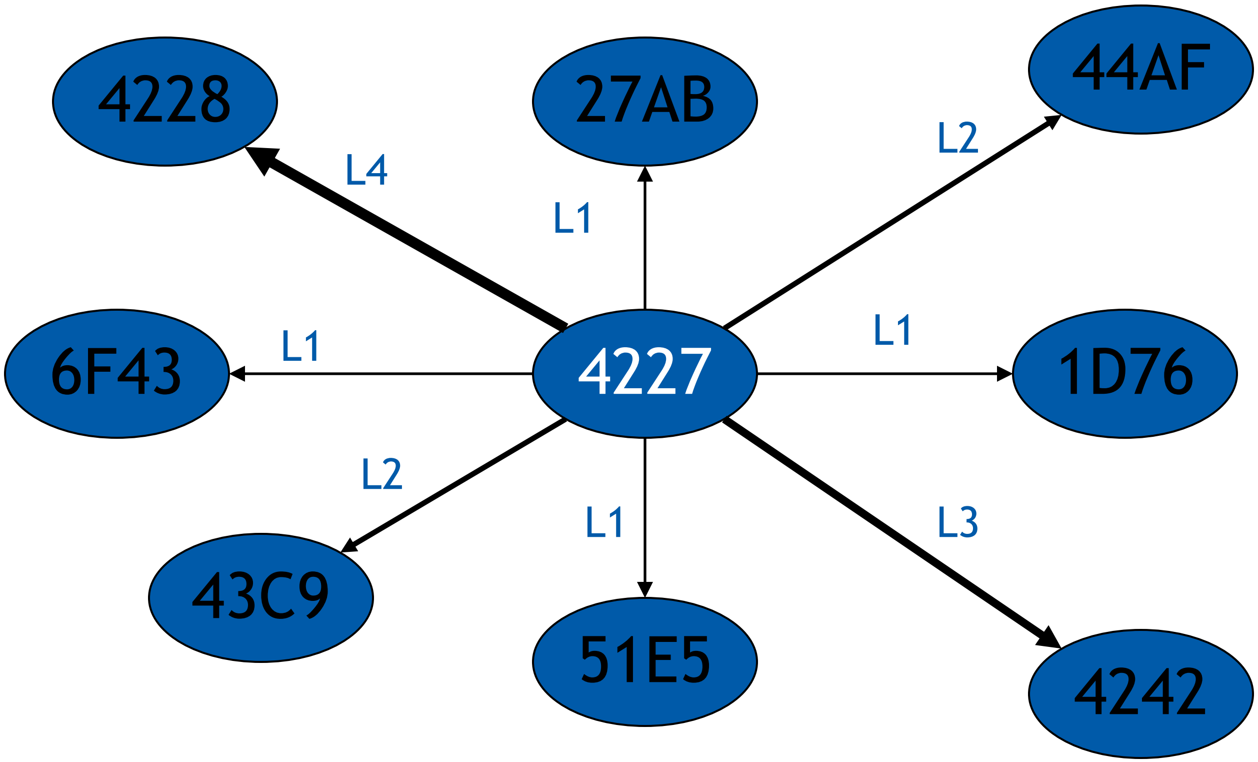
Tapestry: Plaxton Network

- Plaxton network (or Plaxton mesh) based on prefix routing (similar to IP)
  - 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

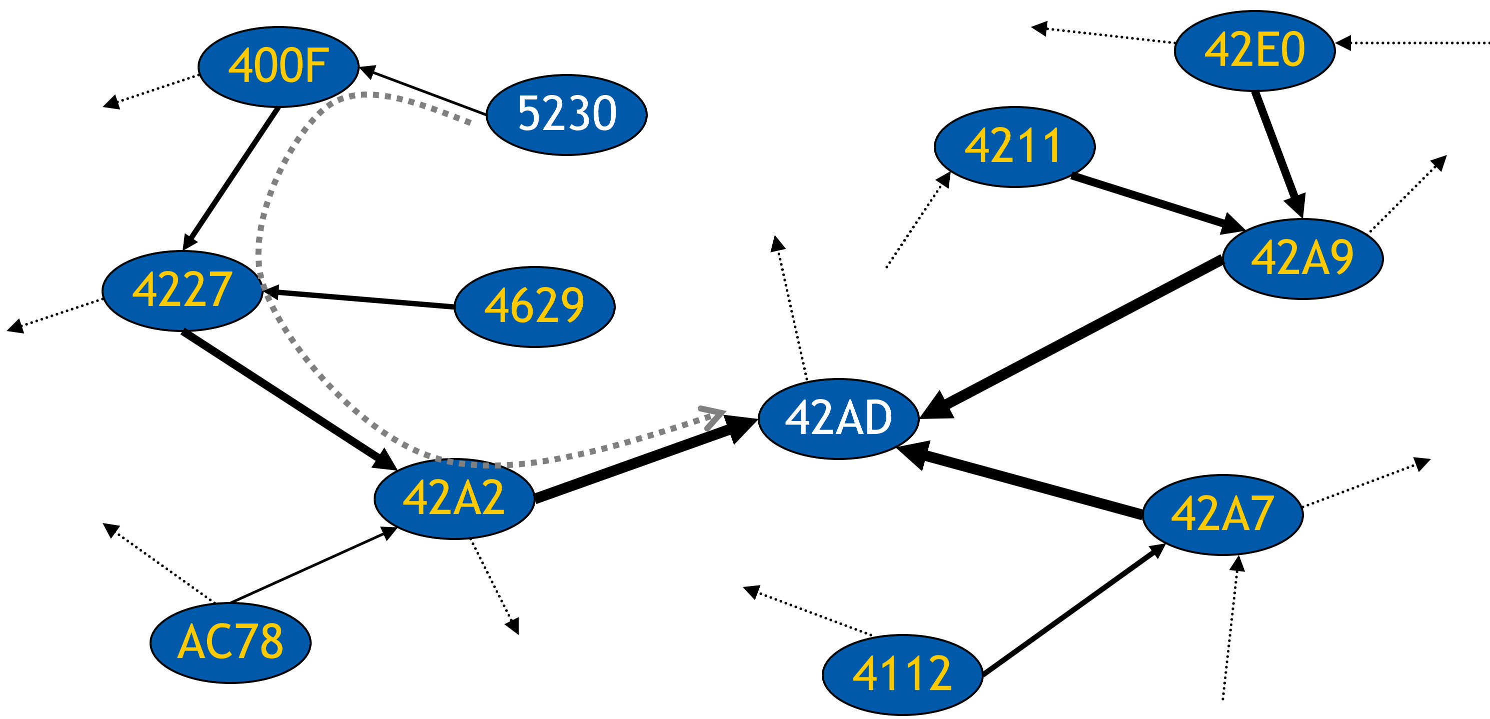


Tapestry: Neighbor Map for 4227

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

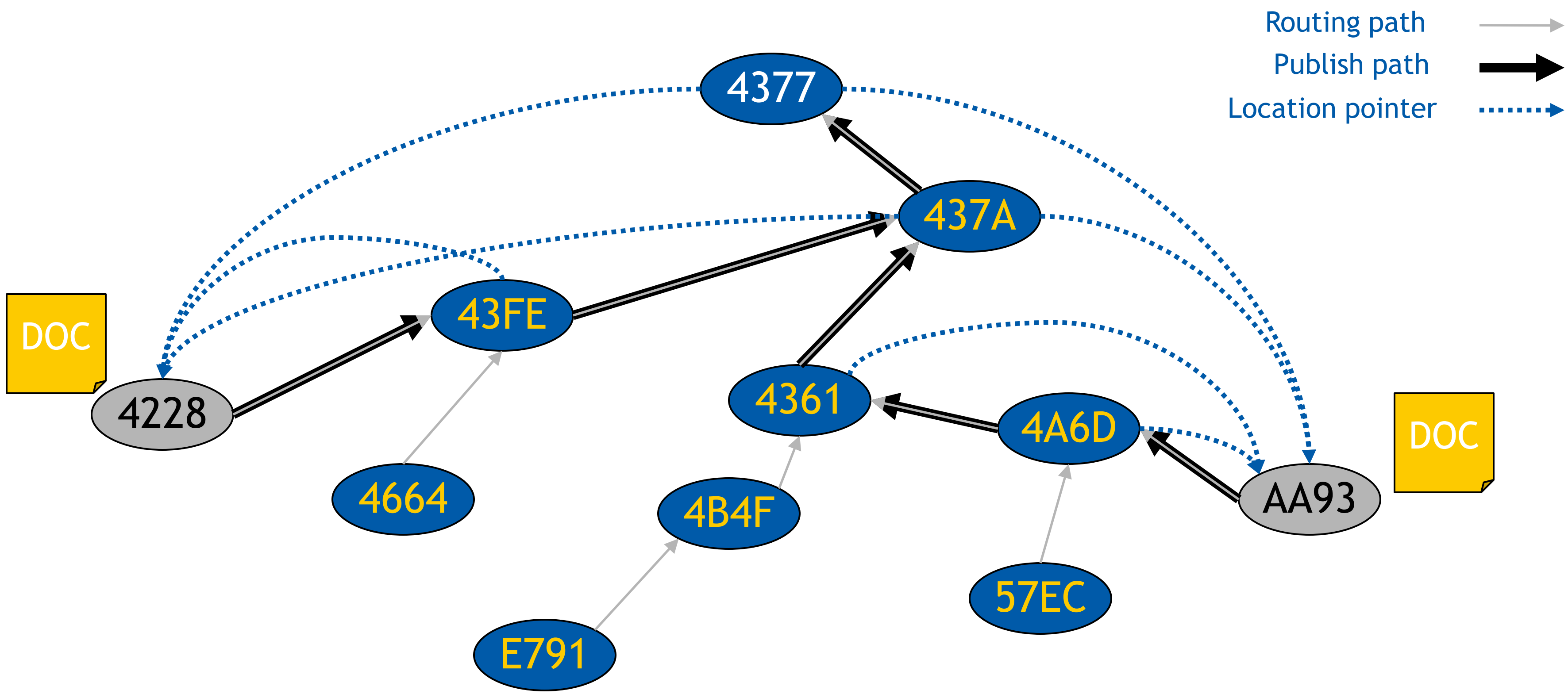


- Route message from 5230 to 42AD
- Always route to node closer to target
  - At n^{th} hop, look at $n+1^{st}$ level in neighbor map --> “always” one digit more
- Not all nodes and links are shown

Tapestry: Properties

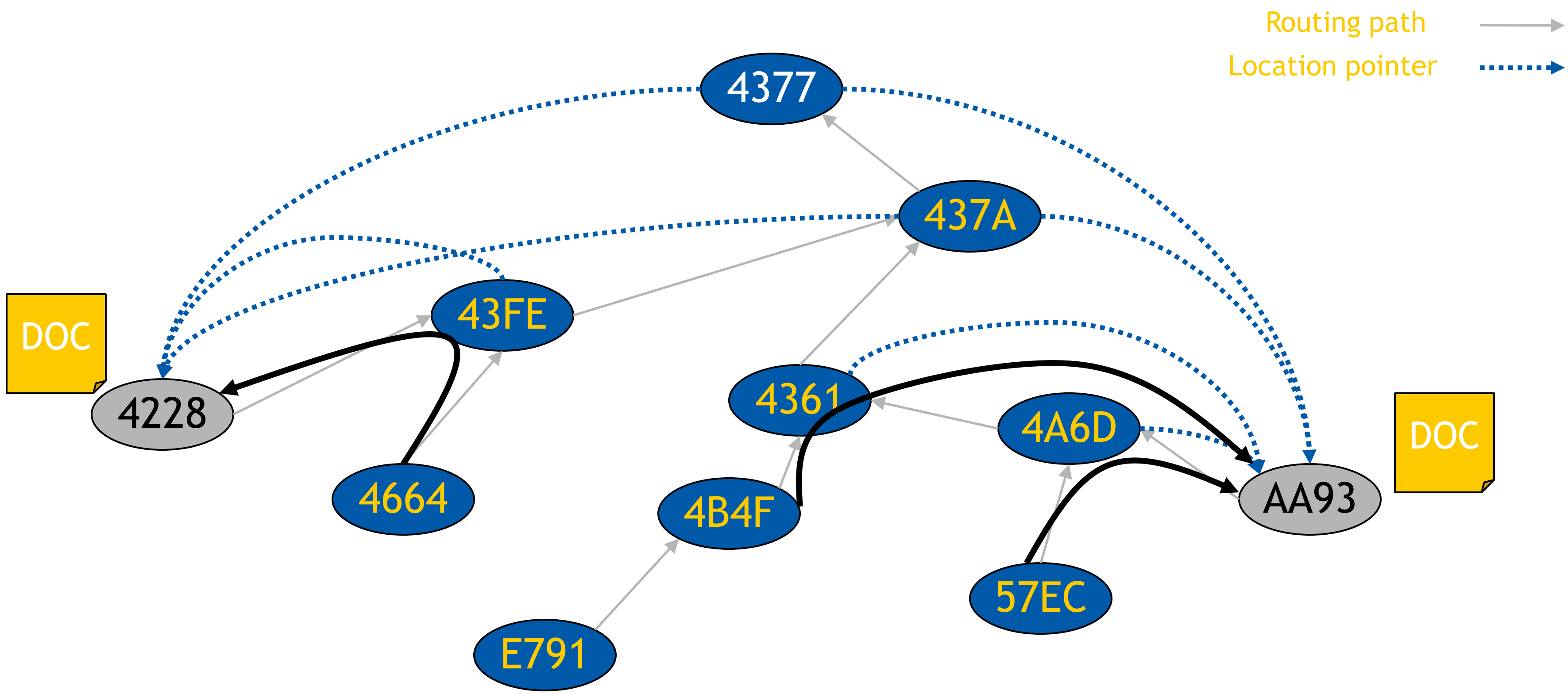
- Node responsible for objects which have same ID
  - Unlikely to find such node for every object
  - Node responsible also for “nearby” objects (surrogate routing, see below)
- Object publishing:
  - Responsible nodes store only pointers
    - Multiple copies of object possible (replica!)
    - Each 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

Tapestry: Publishing Example



- Two copies of object “DOC” with ID 4377 created at AA93 and 4228
- AA93 and 4228 publish object DOC, messages routed to 4377
  - Publish messages create location pointers on the way
- Any subsequent query can use location pointers

Tapestry: Querying Example



- Requests initially route towards 4377
- When they encounter the publish path, use location pointers to find object
- Often, no need to go to responsible node
- Downside: Must keep location pointers up-to-date

Tapestry: Making It Work

- Previous examples show a Plaxton network
  - Requires global knowledge at creation time
  - No fault tolerance, no dynamics
- Tapestry adds fault tolerance and dynamics
  - Nodes join and leave the network
  - Nodes may crash
  - Global knowledge is impossible to achieve

Tapestry: Fault-Tolerant Routing

- Tapestry keeps mesh connected with keep-alives
  - Both TCP timeouts and UDP "hello" messages
  - Requires extra state information at each node

- Neighbor table has backup neighbors
  - For each entry, Tapestry keeps 2 backup neighbors
  - If primary fails, use secondary
    - Works well for uncorrelated failures

- When node notices a failed node, it marks it as invalid
  - Most link/connection failures short-lived
  - Second chance period (e.g., day) during which failed node can come back and old route is valid again
  - If node does not come back, one backup neighbor is promoted and a new backup is chosen

Tapestry: Fault-Tolerant Location

- Responsible node is a single point of failure

- *What can we do?*
- Solution: Map IDs, assign multiple "IDs" per object
  - Add *"salt"* to object name and hash as usual
  - Salt = globally constant sequence of values (e.g., 1, 2, 3, …)

- Same idea as CAN's multiple realities
- This process makes data more available, even if the network is partitioned
  - With s roots, availability is $P \approx 1 - (1/2)^s$
  - Depends on partition
- These two mechanisms improve fault-tolerance
  - In most cases :-)
  - Problem: If the only out-going link fails…

Tapestry: Surrogate Routing

- Responsible node is node with same ID as object
  - Such a node is unlikely to exist
- Solution: surrogate routing
- What happens when there is no matching entry in neighbor map for forwarding a message?

- Node picks (deterministically) one entry in neighbor map
  - Details are not explained in the paper :(

- Idea: If “missing links” are deterministically picked, any message for that ID will end up at same node
  - This node is the surrogate
- If nodes join or leave, surrogate may change

Tapestry: Performance

- Messages routed in max $O(\log_b N)$ hops $(O(\log_b m)\ldots)$
  - At each step, we resolve one more digit in ID
  - N is the size of the namespace (e.g, SHA-1 = 40 (hex) digits)
  - Surrogate routing adds a bit to this, but not significantly

- State required at a node is $O(b \log_b N)$
  - Tapestry has c backup links per neighbor, $O(cb \log_b N)$
  - Additionally, same number of backpointers

Kademlia: A Peer-to-peer Information System Based on the XOR Metric

- Petar Maymounkov and David Mazières (NY Uni) at IPTPS ’02

- Aims:
  - Quick storage and retrieval of index information
  - Tolerance to node failures
  - Balancing storage and communication load
  - Minimize the number of control messages

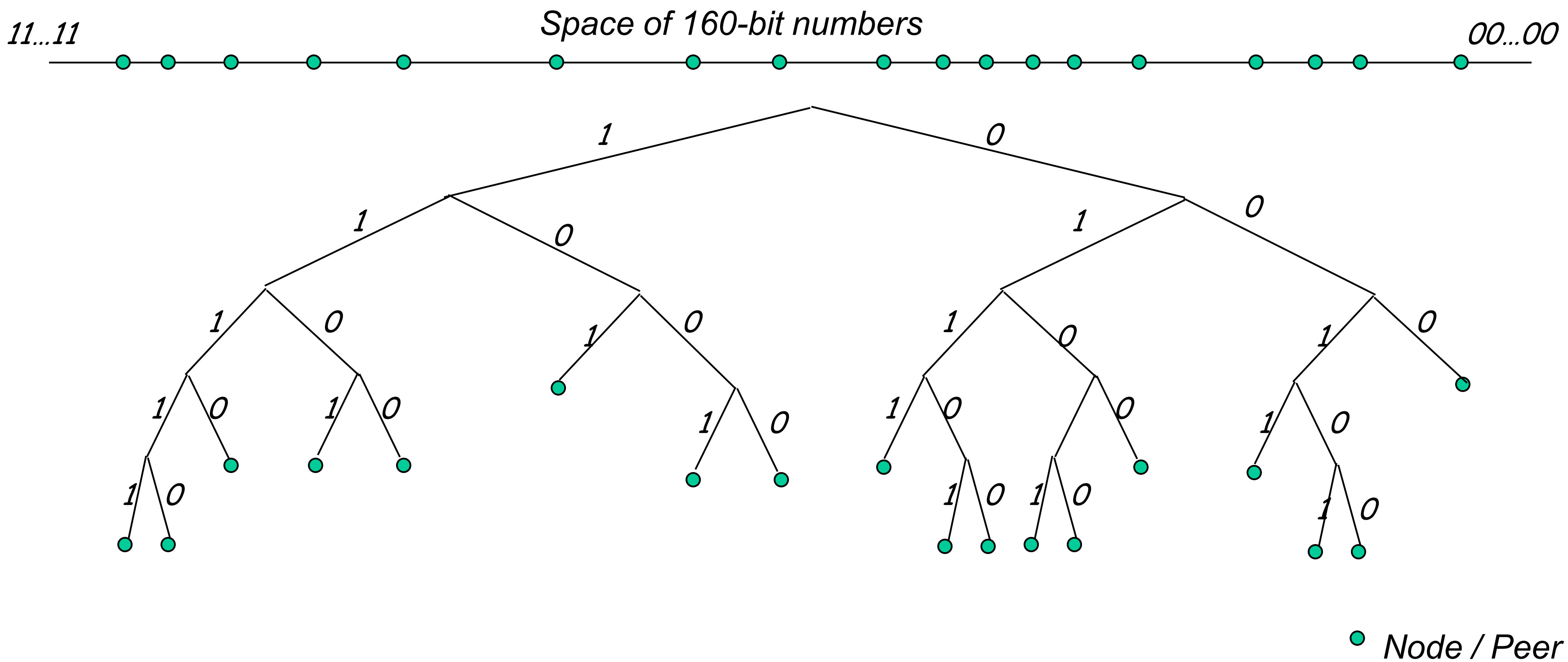
- Ideas:
  - DHT-based approach
  - Parallel asynchronous queries to find low-latency paths
  - „In-band“ messaging: signalling msgs are piggy-backed with key lookups

- Instances:
  - Overnet/Kad (eMule/aMule)
  - Kashmir (Bittorrent)
  - Storm worm (Peacomm)

Kademlia: Protocol Overview

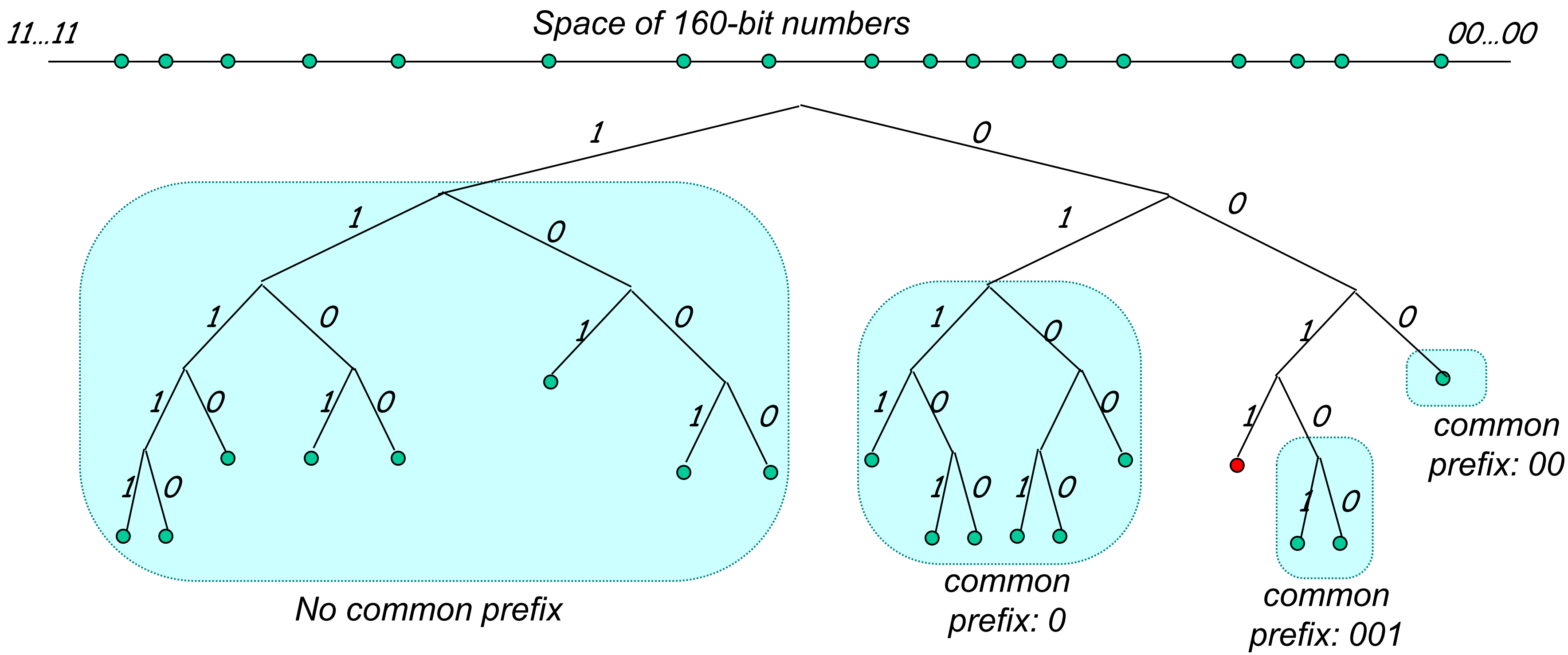
- Kademlia protocol consists of 4 Remote Procedure Calls (RPCs):
  - **PING**$_{v \rightarrow w}$
    - Probe node **w** to see if its online
  - **STORE**$_{v \rightarrow w}$**(Key, Value)**
    - Instructs node **w** to store a <key, value> pair
  - **FIND_NODE**$_{v \rightarrow w}$**(T)**
    - In: **T**, 160-bit ID
    - Out: **k contacts** (<IP:Port, NodeID>) "closest" to **T**
  - **FIND_VALUE**$_{v \rightarrow w}$**(T)**
    - In: **T**, 160-bit ID
    - Out: Value, if a STORE(T, Value) previously received, else **k contacts** (<IP:Port, NodeID>) "closest" to T

Kademlia: Basic Idea



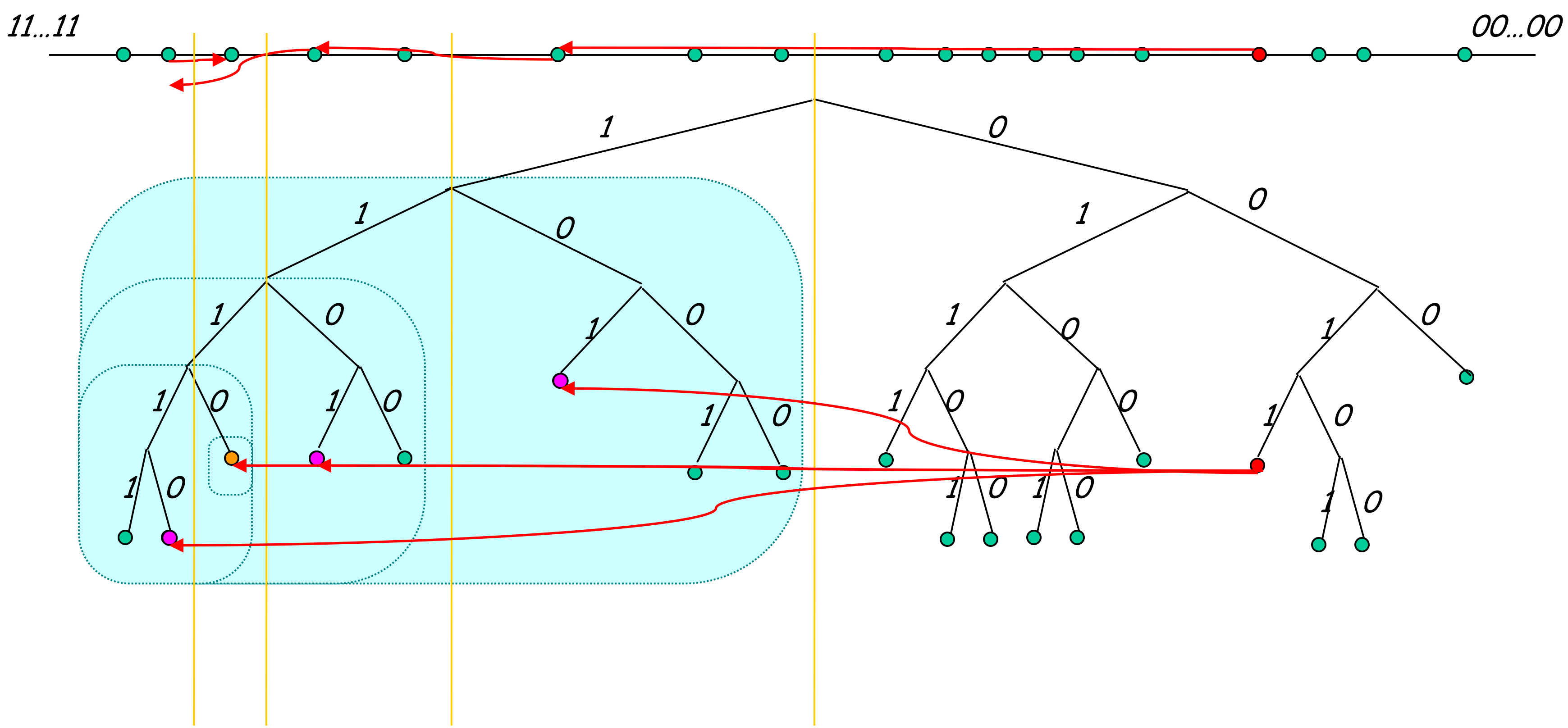
- Nodes are treated as leafs in binary tree
- Position in the tree is determined by the shortest unique prefix of its ID
- A node is responsible for all “closest” IDs, i.e. IDs having same prefix as itself
- Distance between ID x and y is measured as $d(x,y) = x \oplus y$
  - e.g. $d(010101_b, 110001_b) = 100100_b$ **XOR** $d(21_{10}, 49_{10}) = 36_{10}$
  - Nodes/IDs in same subtree (i.e. with longest common prefix) are closer

Kademlia: Basic Idea (2)



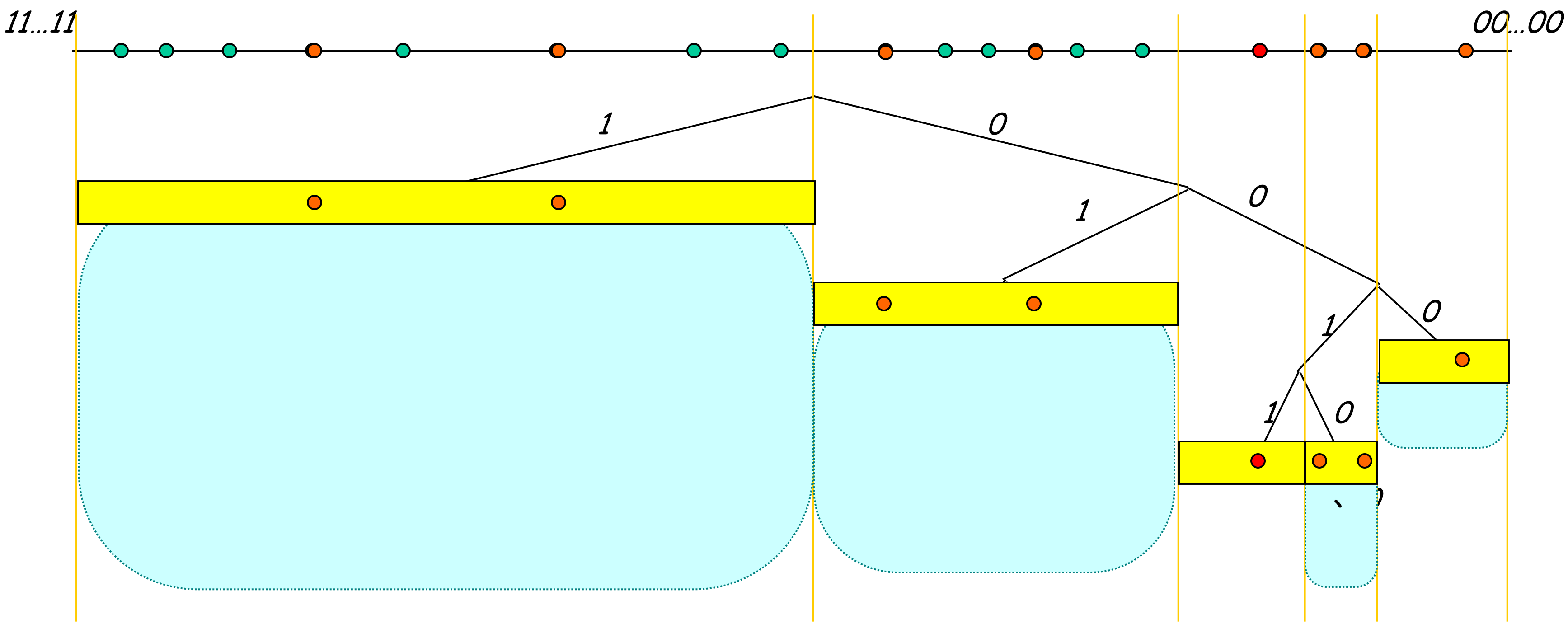
- For any node (say the red node with prefix 0011) the binary tree is divided into a series of maximal subtrees that do not contain the node.
- A node must know at least one node in each of these subtrees.

Kademlia: Basic Idea (3)



- *Consider a query for ID 111010... initiated by node 0011100...*

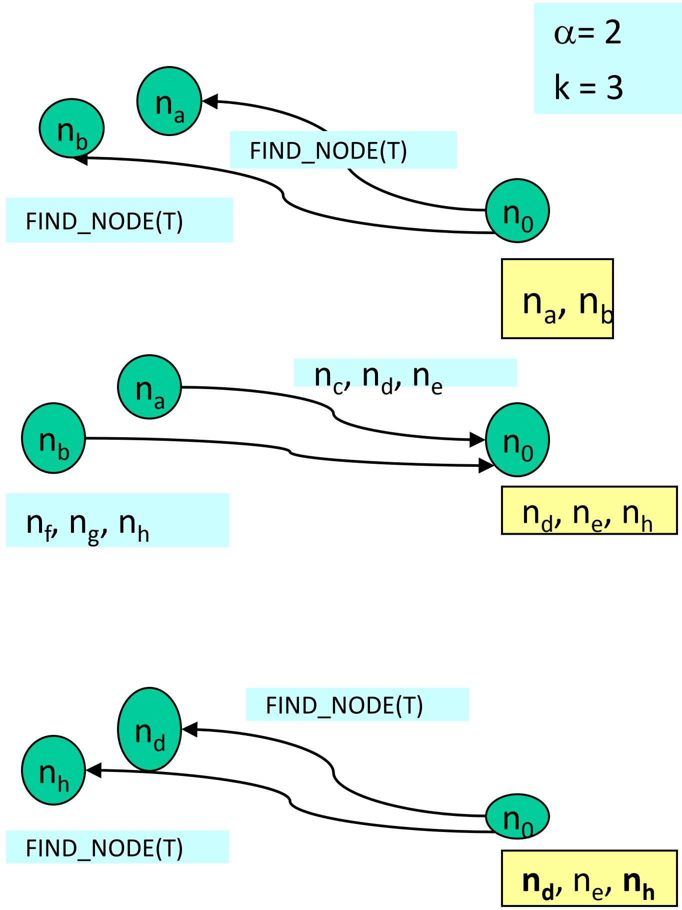
Kademlia: Routing Table



- Consider routing table for node with ***prefix 0011***
- Binary tree is divided into set of subtrees according to their prefix
- The routing table is composed of a series of k-buckets, corresponding to each of the subtrees
- In a 2-bucket example, each bucket will have at least 2 contacts for its key range
- Contacts are described as **<IP:Port, NodeID>**

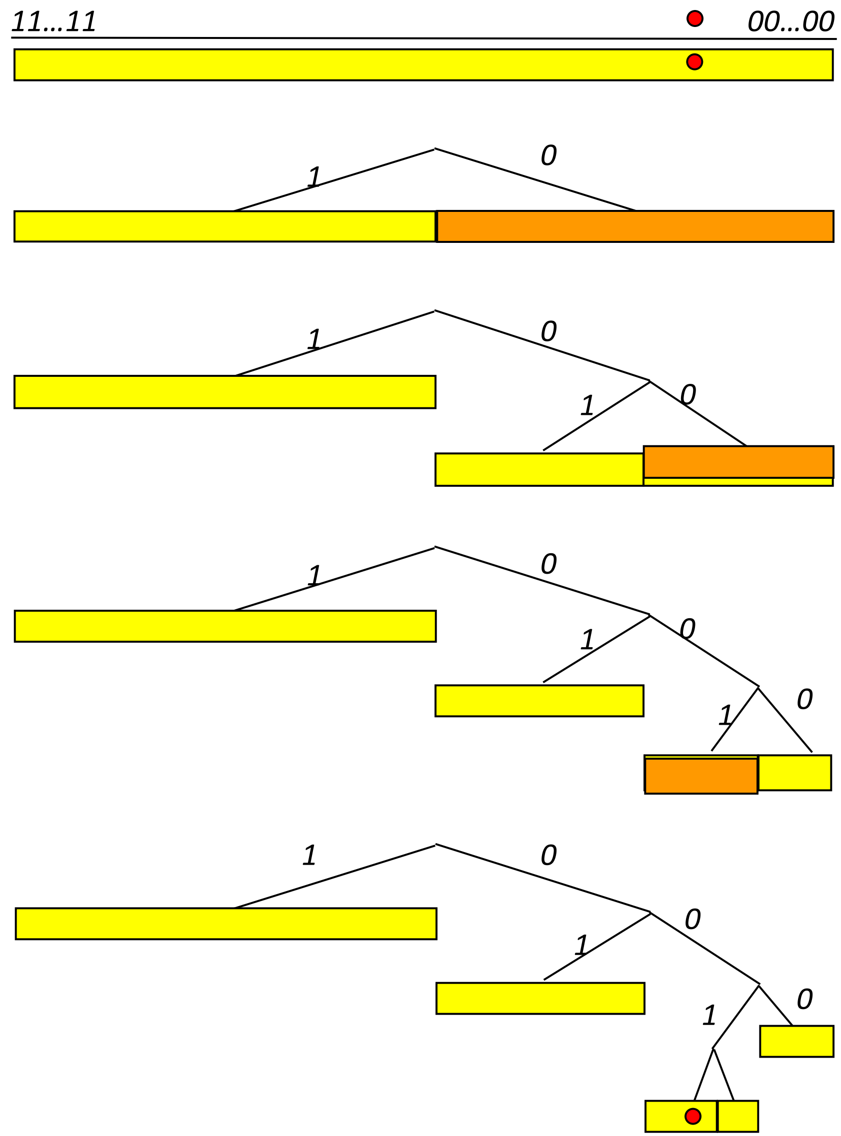
Query Routing Algorithm

- **Goal:** Find k nodes closest to ID T
- **Initial Phase:**
  - Select ∝ nodes closest to T from no's routing table
  - Send FIND_NODE(T) to each of the ∝ nodes in parallel
- **Iteration:**
  - Select ∝ nodes closest to T from the results of previous RPC
  - Send FIND_NODE(T) to each of the ∝ nodes in parallel
  - Terminate when a round of FIND_NODE(T) fails to return any closer nodes
- **Final Phase:**
  - Send FIND_NODE(T) to all of k closest nodes not already queried
  - Return when results from all the k-closest nodes retrieved.



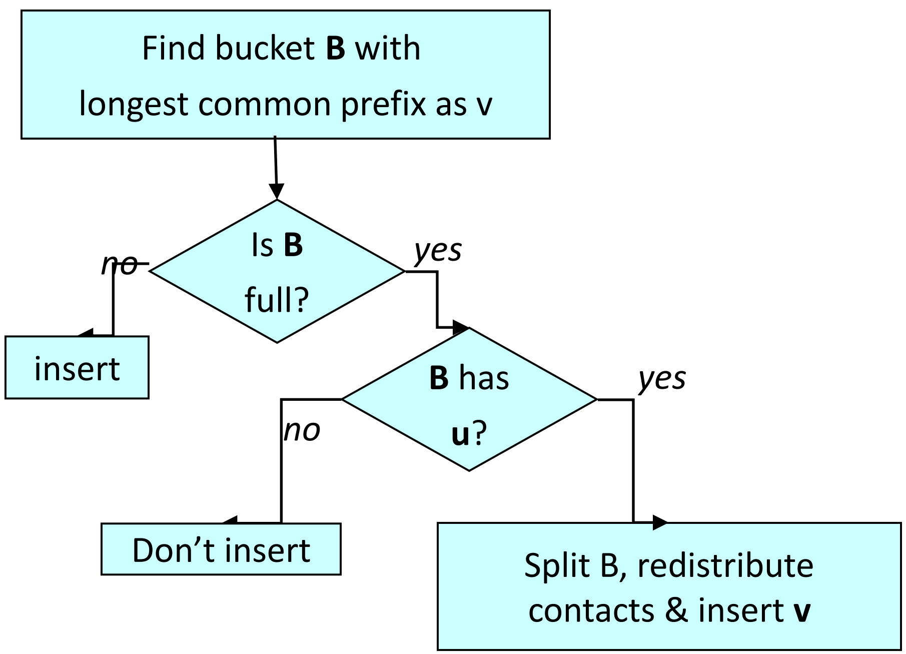
Node Joining & Routing Table Evolution

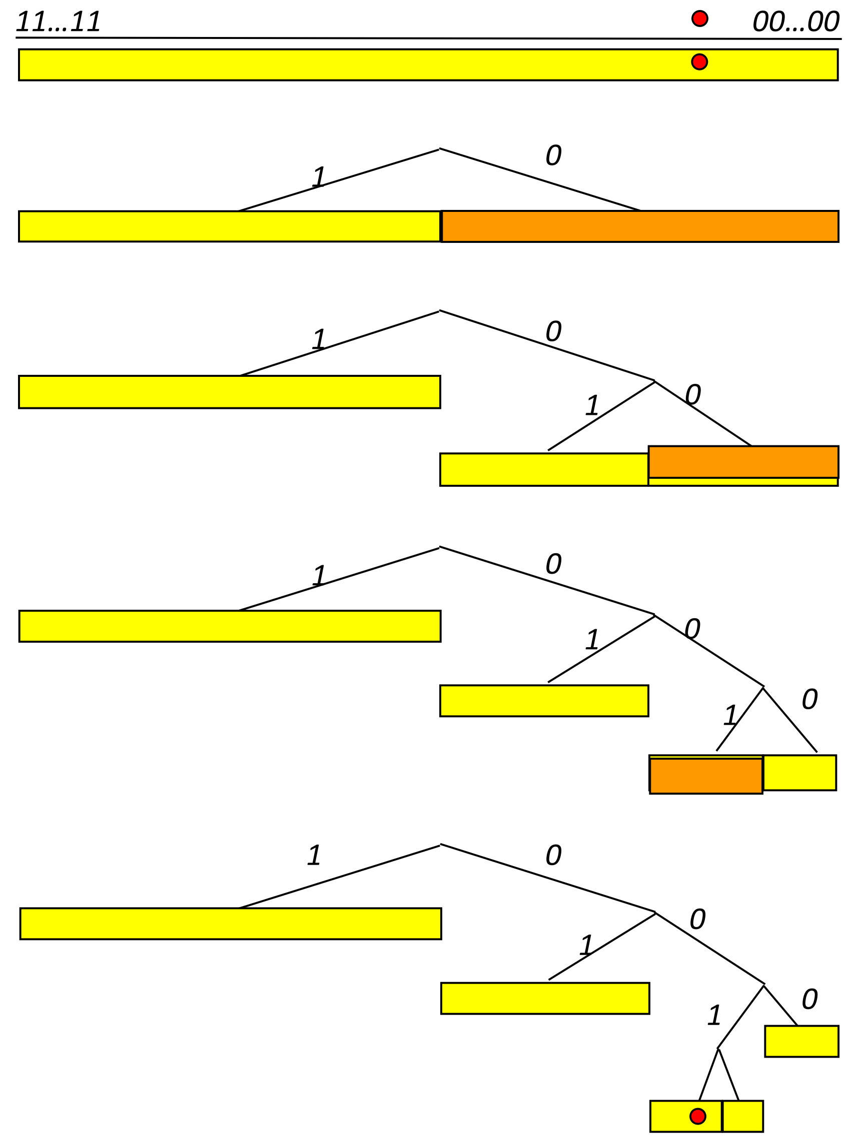
- Joining Node (**u**):
  - Borrow an alive node's ID (**w**) off-line
  - Initial routing table has a single k-bucket containing u and w.
  - **u** performs `FIND_NODE(u)` to learn about other nodes



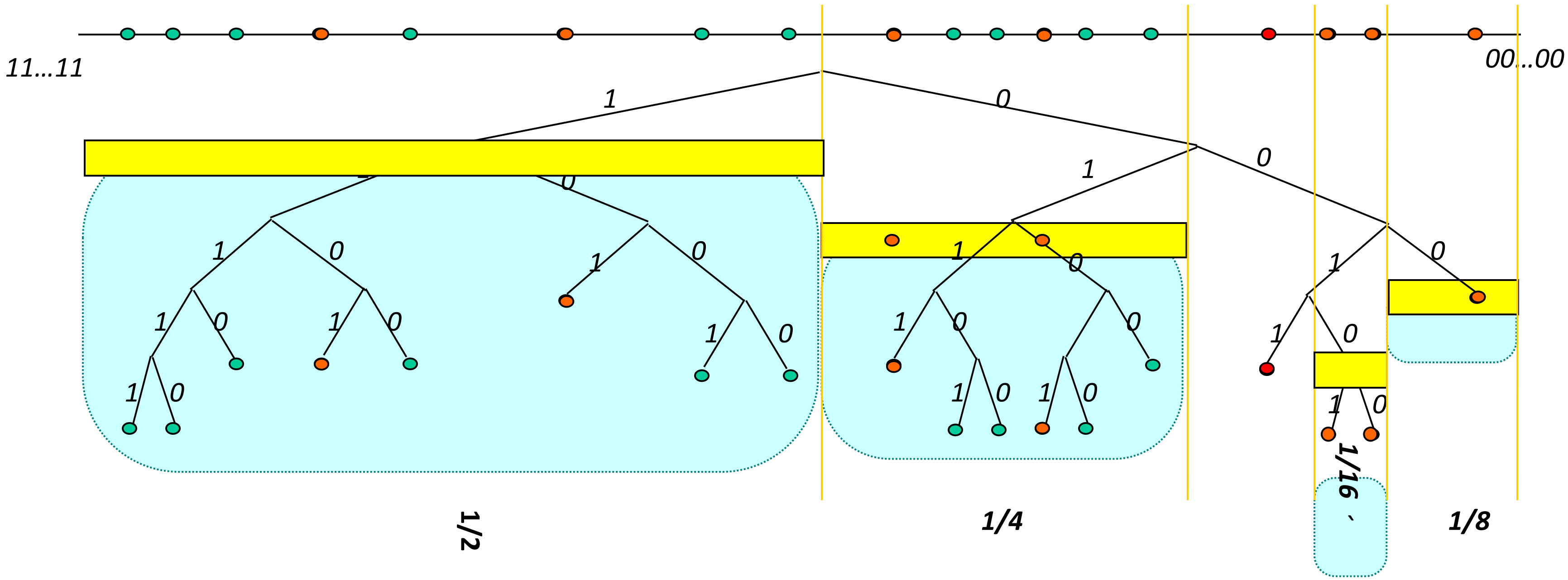
Node Joining & Routing Table Evolution

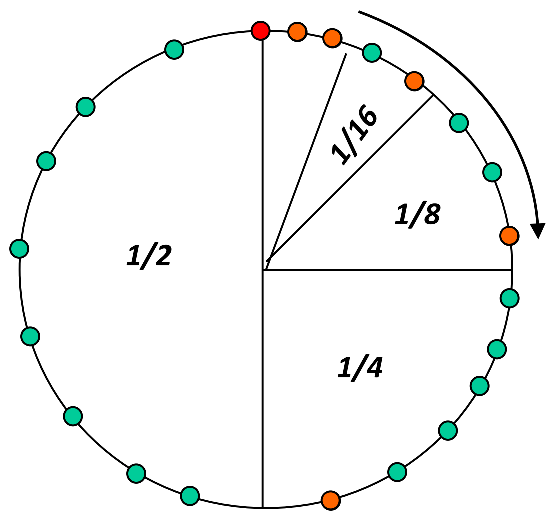
- Joining Node (**u**):
  - Borrow an alive node's ID (**w**) off-line
  - Initial routing table has a single k-bucket containing u and w.
  - **u** performs `FIND_NODE(u)` to learn about other nodes
- *Inserting new entry (**v**):*





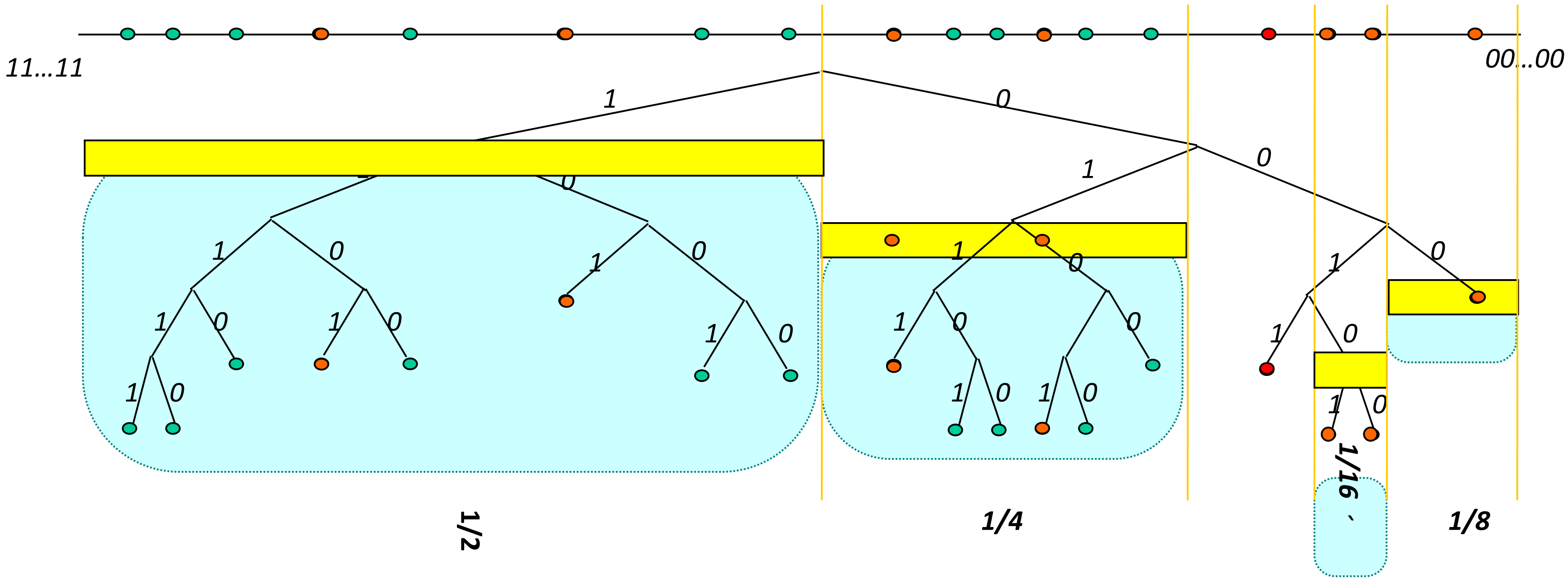
Kademlia vs Chord

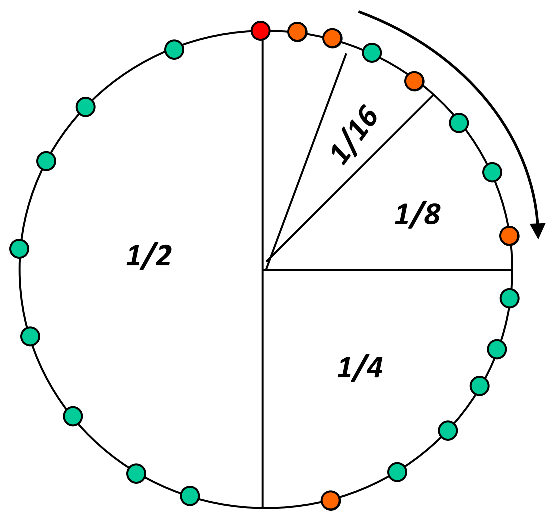




- Chord routing table is rigid, has only one way information flow
  - complicates recovery process
  - Incoming traffic cannot be used for reinforcing routing table.
  - Less fault-tolerance

Kademlia vs Chord





- Chord routing table is rigid, has only one way information flow
  - complicates recovery process
  - Incoming traffic cannot be used for reinforcing routing table.
  - Less fault-tolerance

Kademlia vs Pastry

*Sample routing table in **Pastry***

NodeId 10233102

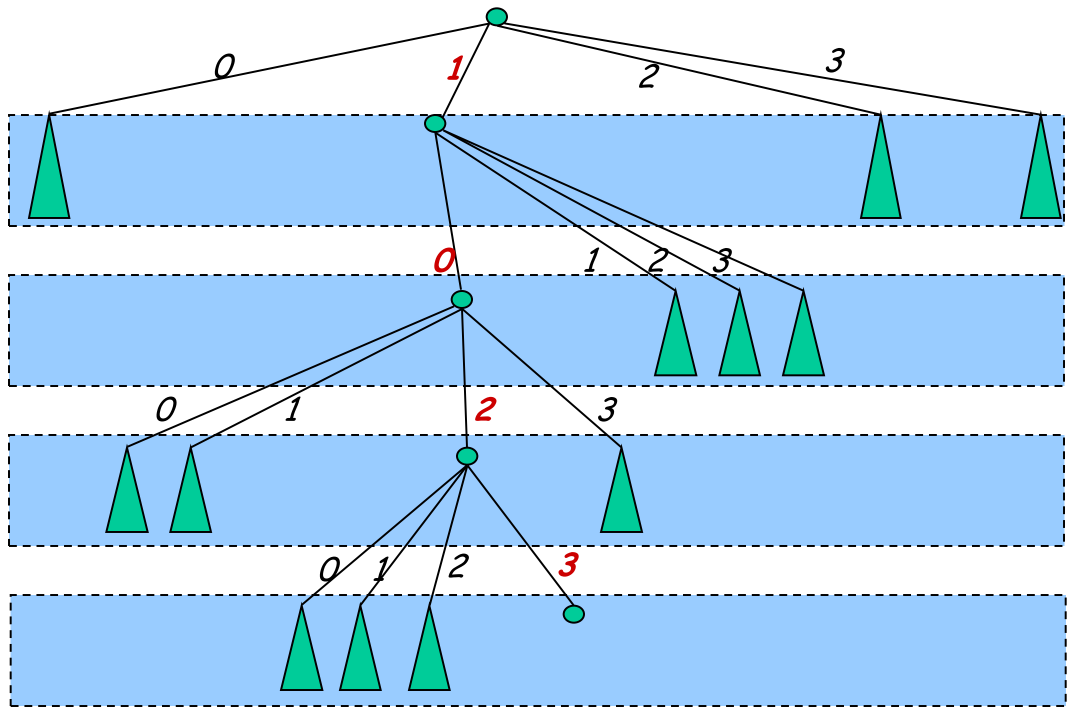
Leaf set	SMALLER	LARGER	
10233033	10233021	10233120	10233122
10233001	10233000	10233230	10233232

Routing table

-0-2212102	1	-2-2301203	-3-1203203
0	1-1-301233	1-2-230203	1-3-021022
10-0-31203	10-1-32102	2	10-3-23302
102-0-0230	102-1-1302	102-2-2302	3
1023-0-322	1023-1-000	1023-2-121	3
10233-0-01	1	10233-2-32	
0		102331-2-0	
		2	

Neighborhood set

13021022	10200230	11301233	31301233
02212102	22301203	31203203	33213321



- *Pastry can not store redundant information in routing table, hence less tolerant to node failure*
- *Pastry has higher control message overhead*
- *Pastry has complex routing table. Two phase routing*
  - *Routing table: for initial hops*
  - *Leaf set: for last few hops*

Kademlia: Summary

- Strengths
  - Low control message overhead
  - Tolerance to node failure and leave
  - Capable of selecting low-latency path for query routing
  - Provable performance bounds

- Weaknesses
  - Non-uniform distribution of nodes in ID-space results into imbalanced routing table and inefficient routing
  - Balancing of storage load is not truly solved
  - Originally underspecified, plethora of different implementations
  - Hard to provide analytical results
  - Non-deterministic results of routing (time, neighborhood)