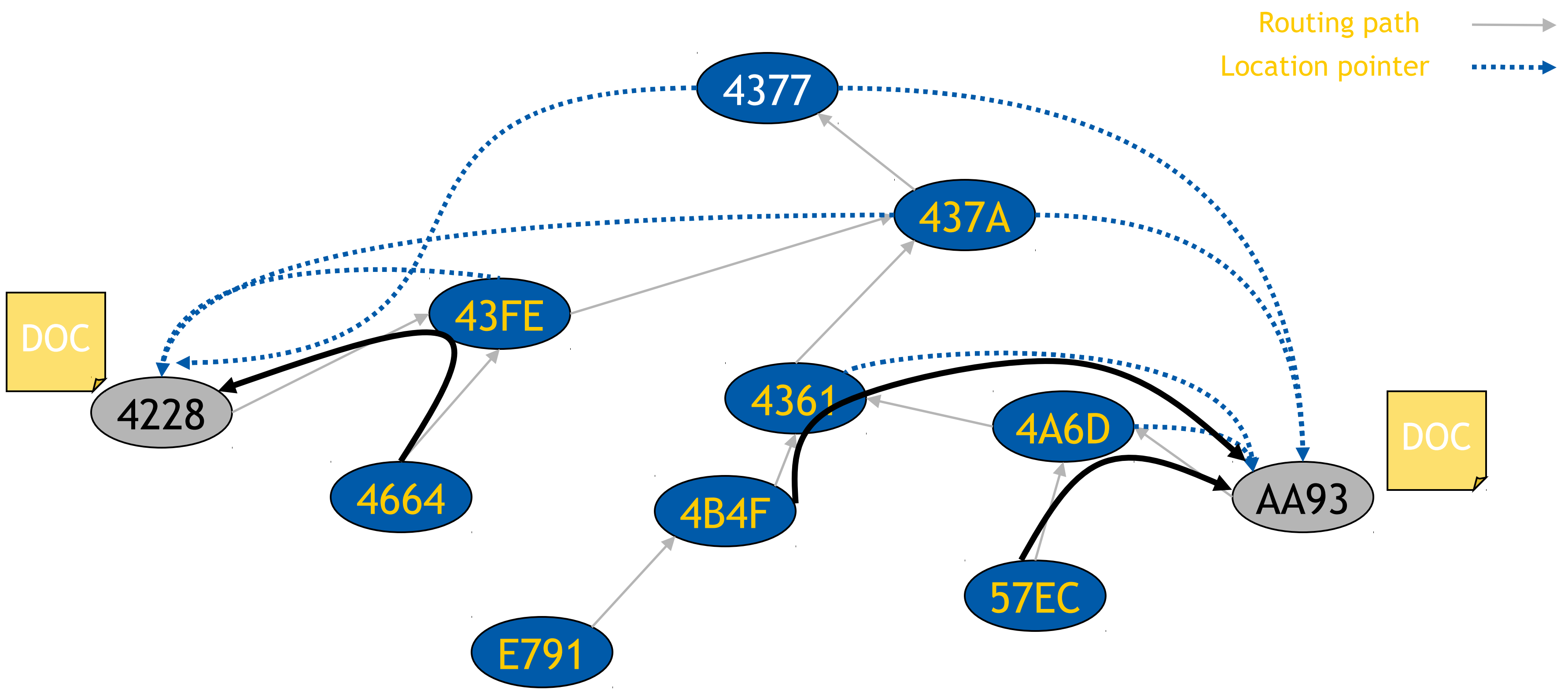
Tapestry: Querying Example



- 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: Assign multiple roots 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 “guarantee” 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 new nodes join, surrogate may change

Tapestry: Performance

- Messages routed in O($\log_b$ N) hops
  - 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

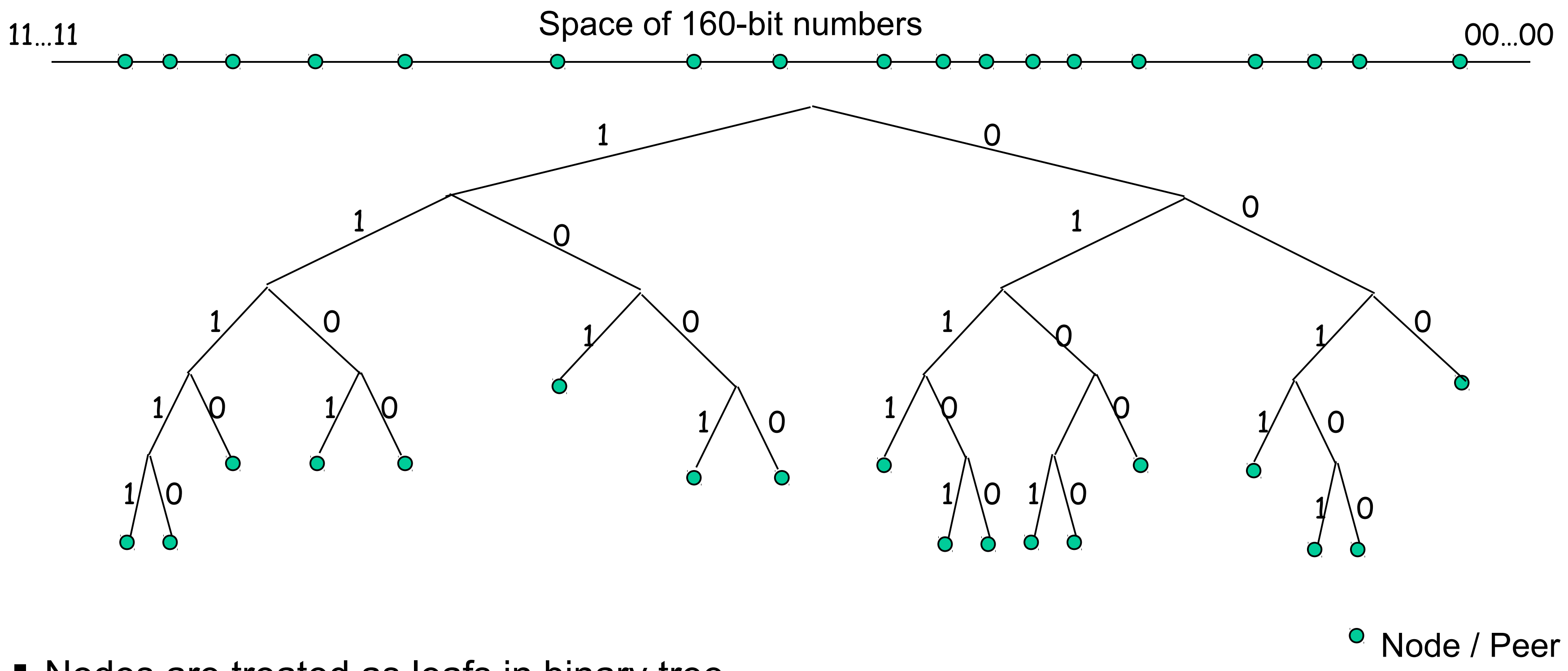
Kademlia: A Peer-to-peer Information System Based on the XOR Metric

- Petar Maymounkov and David *Mazières, New York University*
- International workshop on peer-to-peer systems (IPTPS'02)
- Aims
  - Quick storage and retrieval of Index information
  - Tolerance to node failures
  - Balancing storage and bandwidth load
  - Minimize number of Control messages
- Features
  - A DHT based technique
  - Parallel asynchronous queries and redundancy in routing table
  - Can route queries through low-latency paths.
  - Configuration messages spreads with key lookup
- Applications
  - Overnet network is based on Kademlia concepts
  - eDonkey implements Kademlia

Kademlia: Protocol Overview

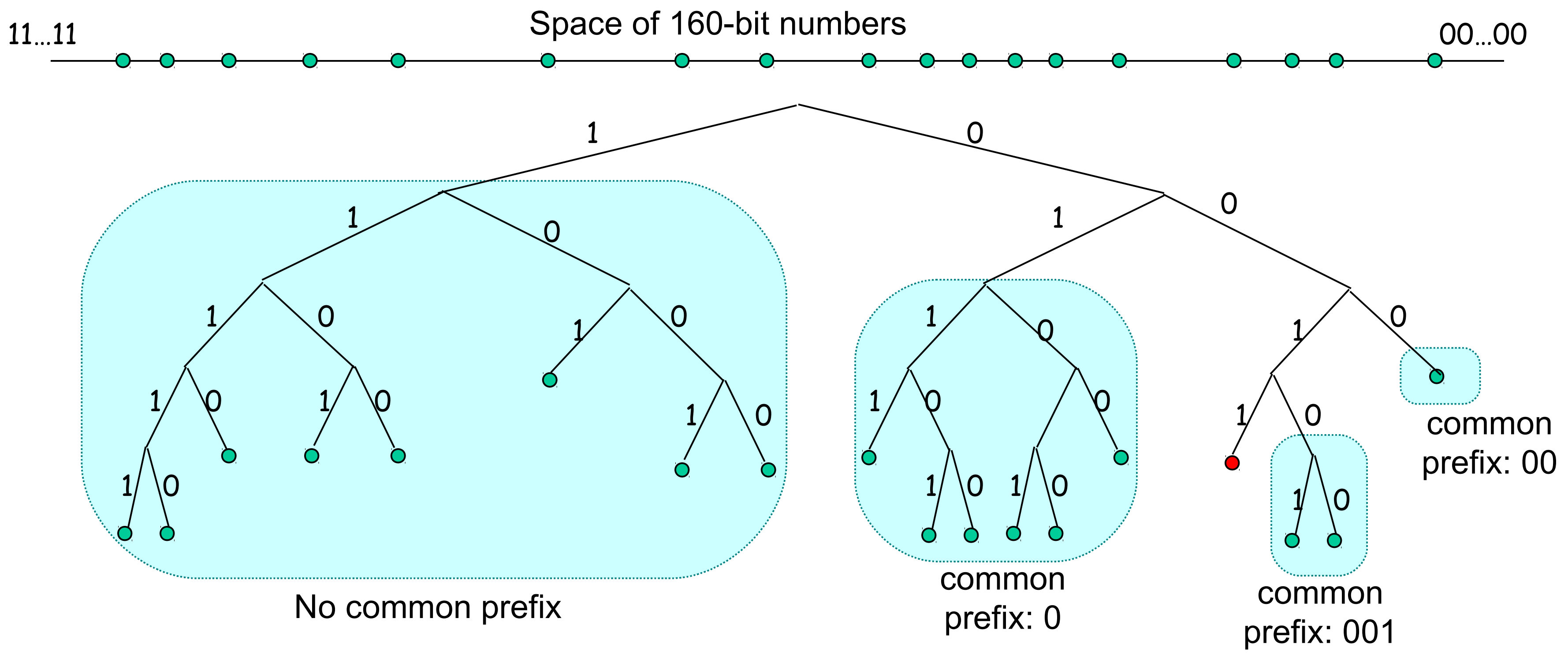
- Kademlia protocol consists of 4 RPCs
  - **PING**$_{n \to m}$
    - Probe node **m** to see if its online
  - **STORE**$_{n \to m}$**(Key, Value)**
    - Instructs node **m** to store a <key, value> pair
  - **FIND_NODE**$_{n \to m}$
    - In: **T**, 160-bit ID
    - Out: **k contacts** (<IP:Port,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

Kademlia: Basic Idea



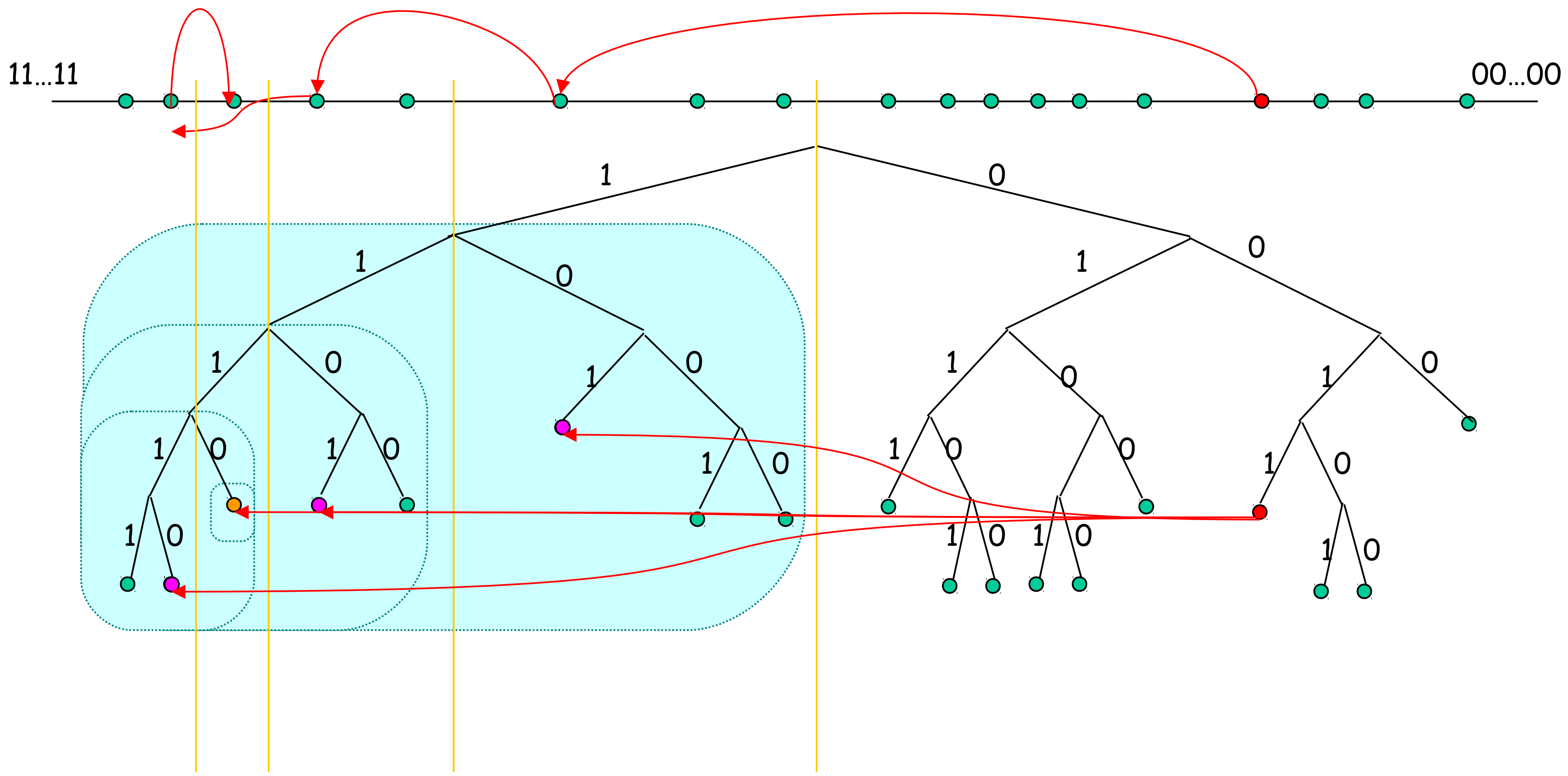
- Nodes are treated as leafs in binary tree
- Node’s position in the tree is determined by the shortest unique prefix of its ID
- A Node is responsible for all “closest” IDs, i.e. IDs having same prefix as itself
- Distance between ID x and y is measured as d(x,y) = x y
  - e.g. d(010101_b, 110001_b) = 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



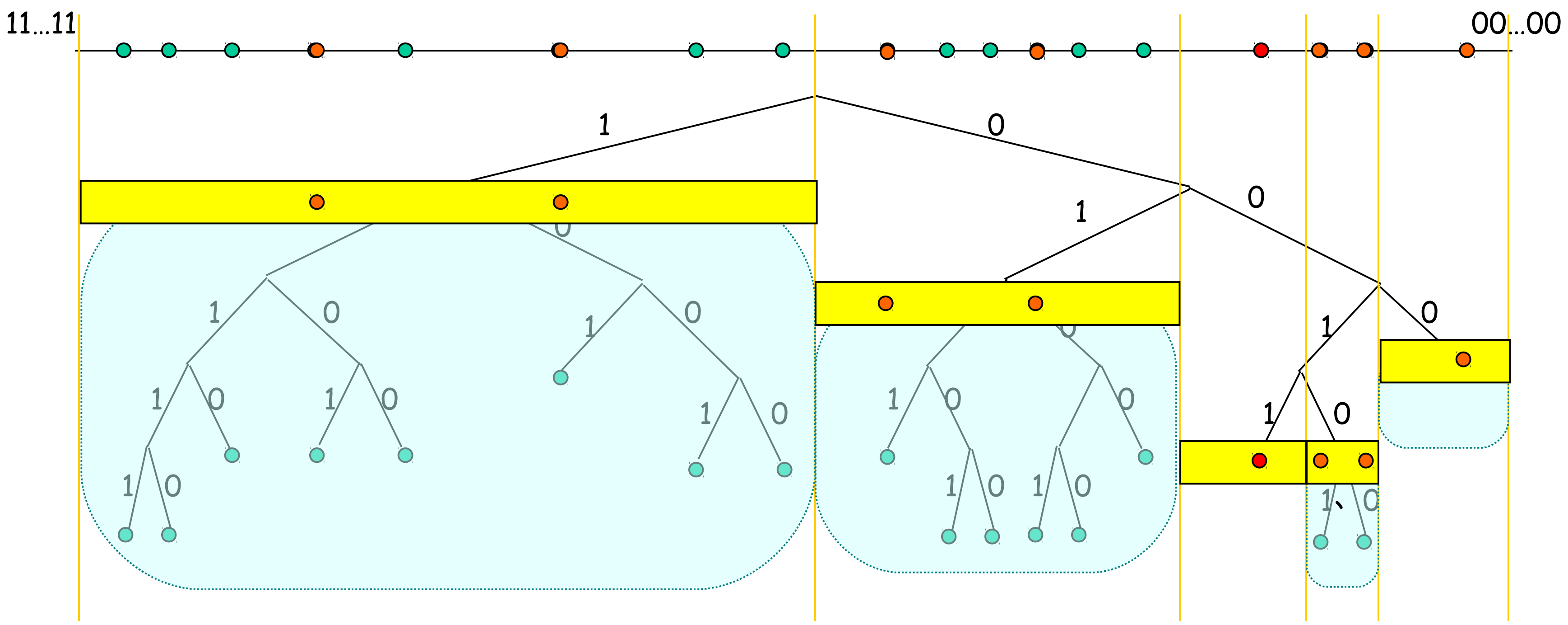
- For any node (say the red node with prefix 0011) the binary tree is divided into a series of maximal subtrees that do not contain the node.
- A node must know at least one node in each of these subtrees.

Kademlia: Basic Idea



- Consider a query for ID 111010… initiated by node 0011100…

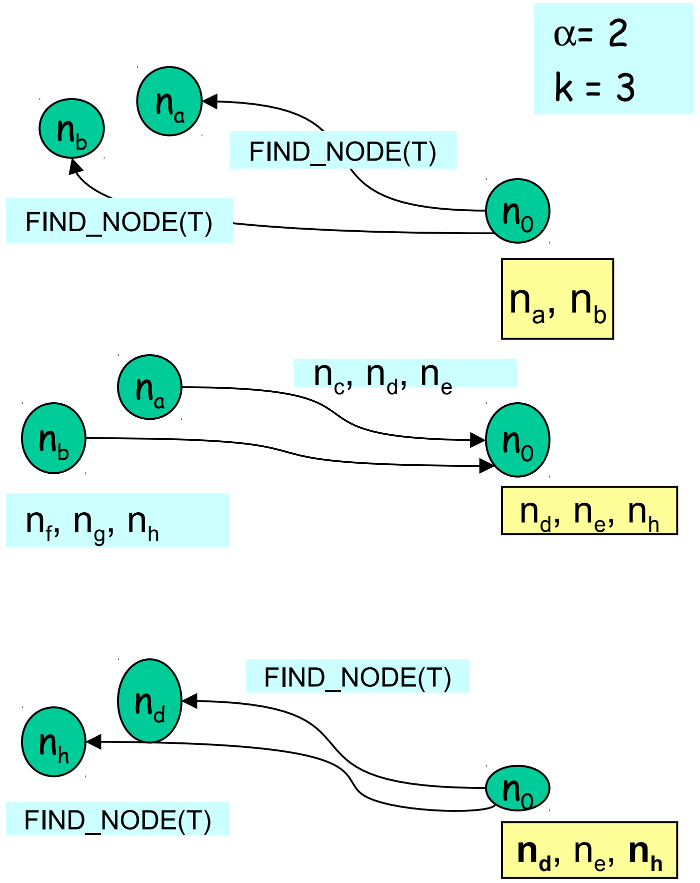
Kademlia: Routing Table



- Consider routing table for a node with prefix 0011
- Its binary tree is divided into a series of subtrees
- The routing table is composed of a series of **k- buckets** corresponding to each of these subtrees
- Consider a 2-bucket example, each bucket will have atleast 2 contacts for its key range
- A contact consist of **<IP:Port, NodeID>**

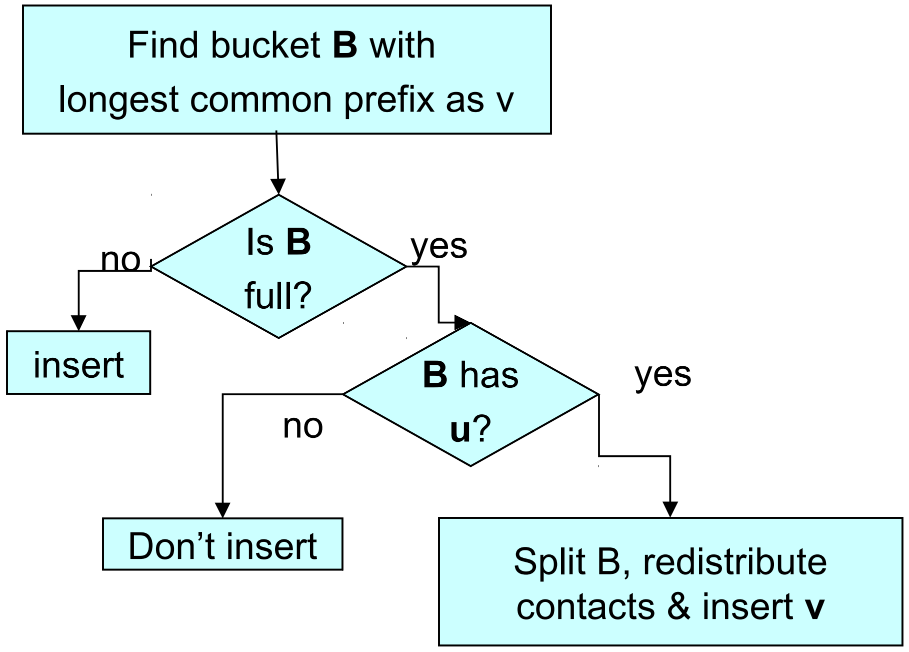
Query Routing Algorithm

- **Goal**: Find k nodes closest to ID T
- **Initial Phase**:
  - Select α nodes closest to T from n_o's routing table
  - Send FIND_NODE(T) to each of the α nodes in parallel
- **Iteration**:
  - Select α nodes closest to T from the results of previous RPC
  - Send FIND_NODE(T) to each of the α 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 have results from all the k-closest nodes.

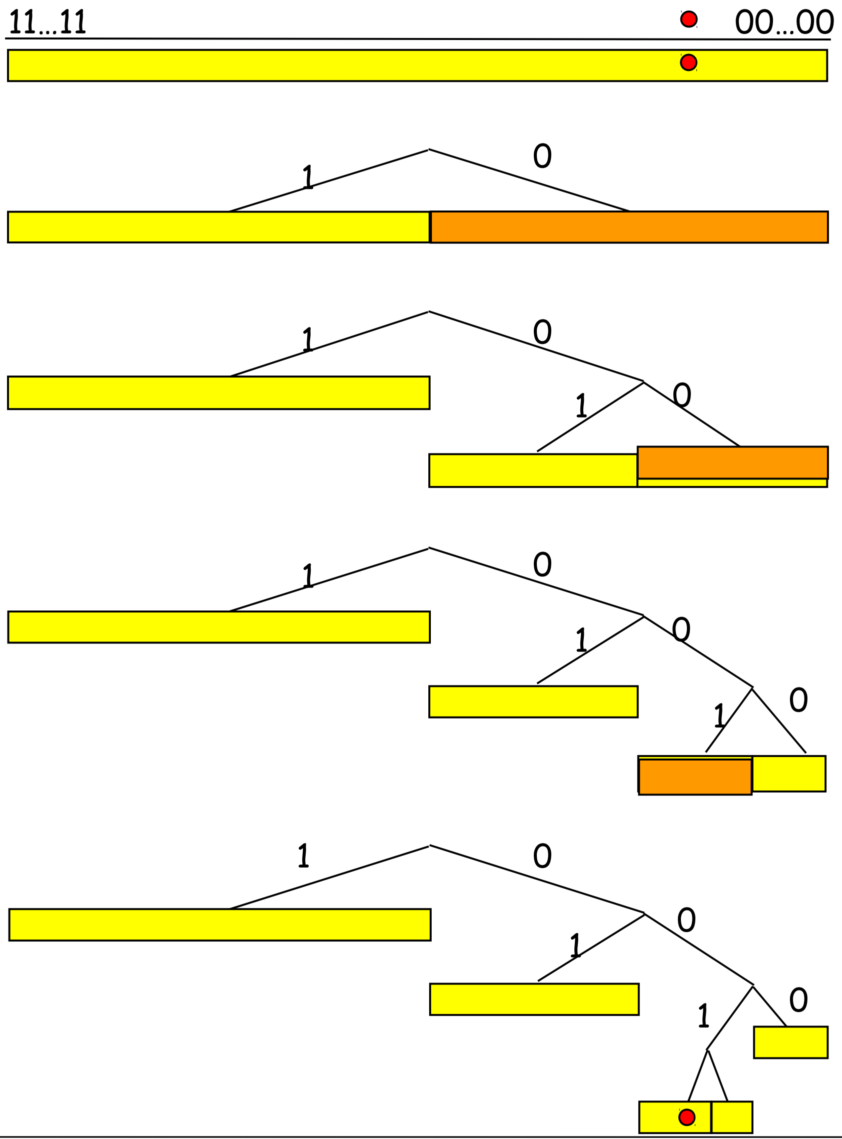


Node Joining & Routing Table Evolution

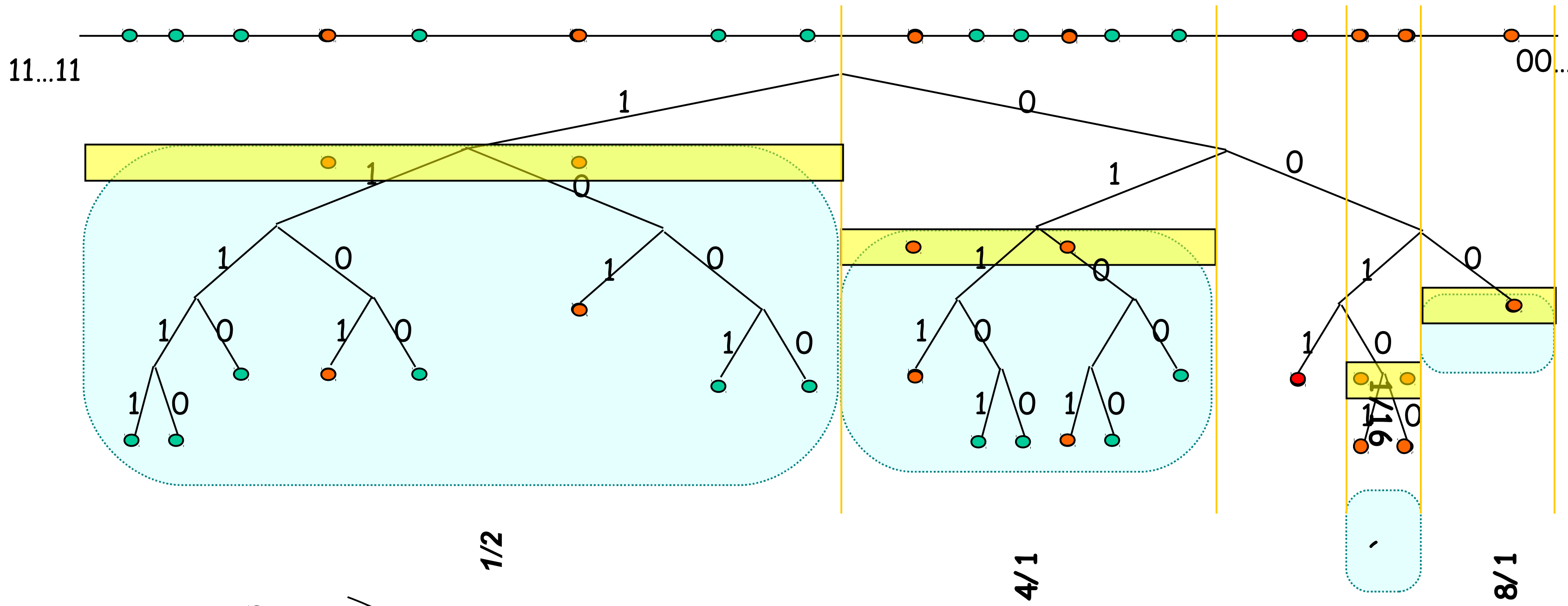
- Joining Node (**u**):
  - Borrow an alive node's ID (**w**) off-line
  - Initial routing table has a single k-bucket containing u and w.
  - **u** performs FIND_NODE(**u**) to learn about other nodes
- Inserting new entry (**v**)

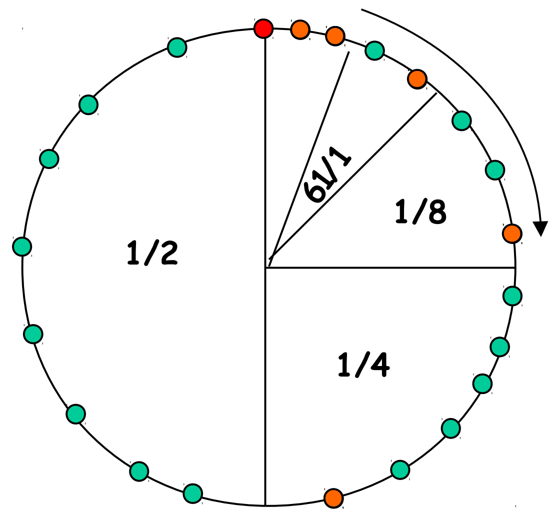


Find bucket **B** with longest common prefix as v
Is **B** full?
no
yes
insert
B has **u**?
no
yes
Don't insert
Split B, redistribute contacts & insert **v**



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: 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
  - No experimental results provided

DHT: Comparison

	Chord	CAN	Tapestry
Type of network	Ring	N-dimensional	Prefix routing
Routing	O(log n)	$O(d \cdot n^{1/d})$	$O(\log_b N)$
State	O(log n)	O(d)	$O(b \cdot \log_b N)$
Caching efficient	+	++	++
Robustness	-/+	+++	++
IP Topology-Aware	N	N/Y	Y
Used for other projects	+++	--	++

Note: n is number of nodes, N is size of Tapestry's namespace

Quick Interlude: Deterministic Routing in P2P

- Consider the DHT (Chord, CAN, Tapestry)
- Is this routing deterministic? [1]
- What happens when it‘s not?
- Is this a problem?
- How can we deal with this?

[1] (DON‘T try wikipedia on this!)

Yet Another Quick Comparison

- Unstructured P2P
  - Select neighbors randomly -> flood
  - Select neighbors, create hierarchy -> ask SN, flood
- Structured P2P
  - Select random ID, locate neighbors to create routing structure
  - Route requests
- Are there further possibilities?
  - Distance Vector Routing -> gossipping
  - Random Walks
- Can we increase the success („hit“) rate?
  - Replicate registration
  - Additionally may decrease response times!

Other DHTs

- Many other DHTs exist too
  - Pastry, similar to Tapestry
  - Kademlia, uses XOR metric
  - Kelips, group nodes into k groups, similar to KaZaA
  - Plus some others…
- Overnet P2P network (also eDonkey) uses Kademlia
  - Wide-spread deployed DHT
- All DHTs provide same API
  - “KBR”: Key based routing
  - In principle, DHT-layer is interchangeable

Issues Beyond Searching and Addressing

- Great! Now we can find resources by name or id
  - Search it using Flooding, Hierarchy, Gossiping, Random Walks
  - Address it using CAN, DHT, etc…

Is this very useful?
What can go „wrong“?

- All the P2P networks create an overlay and delegate requests
  - Neighbor selection random
    - Number of neighbors randomly distributed (node degree distribution)
    - Neighbors randomly distributed around the world
  - Next-hop selection / delegation
    - Random
    - Hierarchy
    - Greedy

Avoiding Useless Traffic (and Delays!)

- Can we avoid delegating from DE -> AUS -> US -> GB -> TV...?
  - „**Stress**“: amount of identical packets traversing the same physical link
  - „**Delay Stretch**“: ratio of the overall sum of hops on the **overlay path**, divided by the number of hops on the **unicast path**

- Can we create a „location aware“ overlay?
  - BTW: what is „location“ on the Internet?

    Darmstadt (DFN) -> Berlin (DFN) can be a lot „closer“ than Darmstadt (DFN) -> Weiterstadt (DSL)!
  - Common (mis-) used metrics:
    - RTT (ECHO, ping)
      - But: DSL without fastpath has ping times like TU-Darmstadt -> Vanuatu...
    - Bandwidth between end-hosts
      - Which bandwidth? Overall? Available? How do we measure that?
    - IP-Hops
      - Stuttgart is in same distance cmp. to New York (www.dfn.de vs www.ny.com)

Avoiding Traffic ctd.

- Which degrees of freedom do we have?
  - Select neighbor
  - Select next hop
  - Select ID !?

    (The respective others kept conventional…)

- Location-based neighbor selection
  - Pastry, Tapestry, etc.: only store the closest in routing tables
- Location-based next-hop selection
  - Any: from all neighbors that are closer to resource select the nearest
- Topology-based ID selection
  - „Learn“ ID depending on
- All of them have pros and cons

CAN revisited: Location aware DHT

- Synthetic coordinates used as ID in DHT

  $\mathtt{mapV(v)} := \vec{v} = [v_1||\ldots||v_d]$

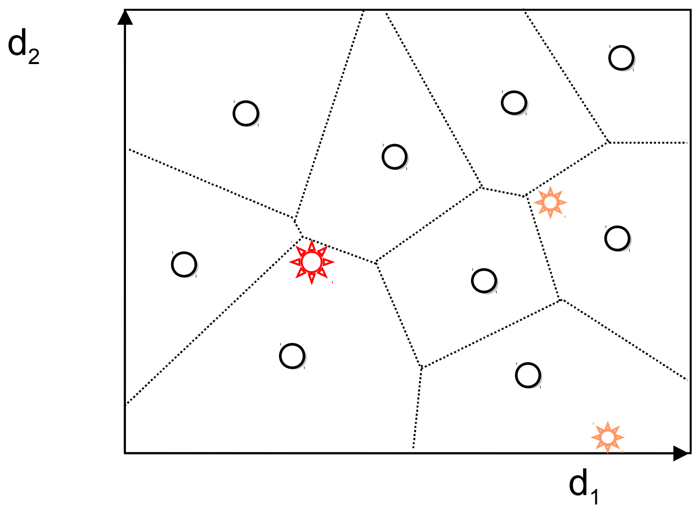
- Registration
  - Map resource ("**o**") in the coordinate space

    $\mathtt{mapO(o)} := \vec{o} = [o_1||\ldots||o_d]$

  - Register at different coordinates using well known functions:

    $\mathtt{M1}(\vec{o}) = -\vec{o} = (-o_1, \ldots, -o_d)$

    $\mathtt{M2}(\vec{o}) = ((o_1 + o_{max}) \bmod (2 * o_{max}), \ldots, (o_d + o_{max}) \bmod (2 * o_{max}))$



- Routing
  - Greedy-Routing:

    $\mathtt{nextHop} = v : |\vec{v} - \vec{o}| = \min \{ |\vec{v} - \vec{o}|, v \in \mathtt{Neighbors} \}$

- Overlay-Construction
  - Select all "direct" neighbors in the coordinate space (in all directions)
  - Additional neighbors in different distances in diverse directions

Strufe: Ein Peer-to-Peer-basierter Ansatz für die Live-Übertragung multimedialer Daten. PhD thesis

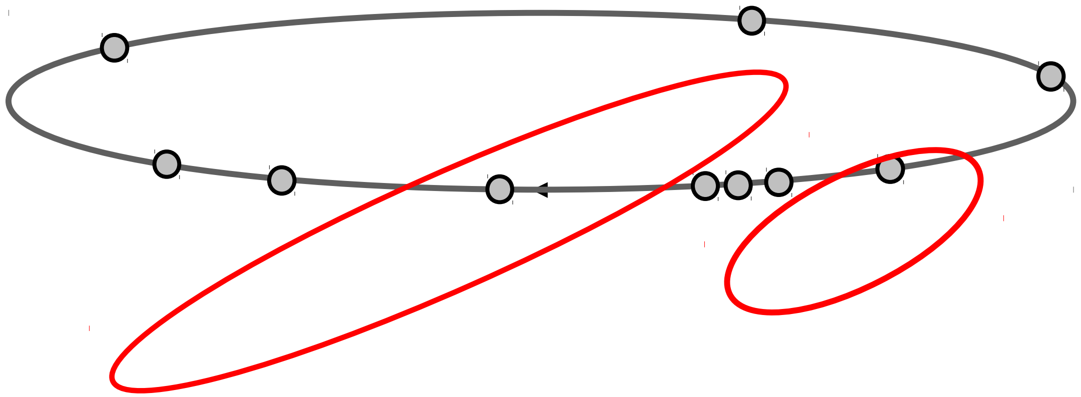
What About the Load at Peers?

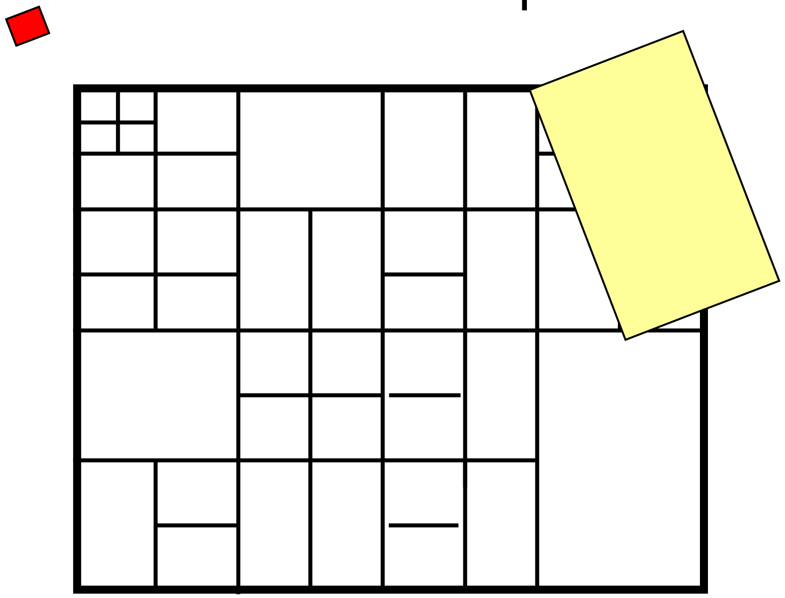
- „A major property of P2P systems (/DHT) is their inherent load balancing.“
  - Requests are served from all peers equally
  - Task of uploading files is shared between all downloading peers
  - Rather random neighbor selection leads to fair allocation of requests
  - Random ID selection leads to good distribution of the namespace…
- What kind of load?
  - Messaging load
  - Request processing load

Load Imbalance

So what can go wrong?

- Uneven distribution of names in ID space (Zipf!)
- Neighbor selection random (preferential attachment?) -> uneven in-degree, uneven incoming requests
- ID selection random -> normally distributed name space allocation

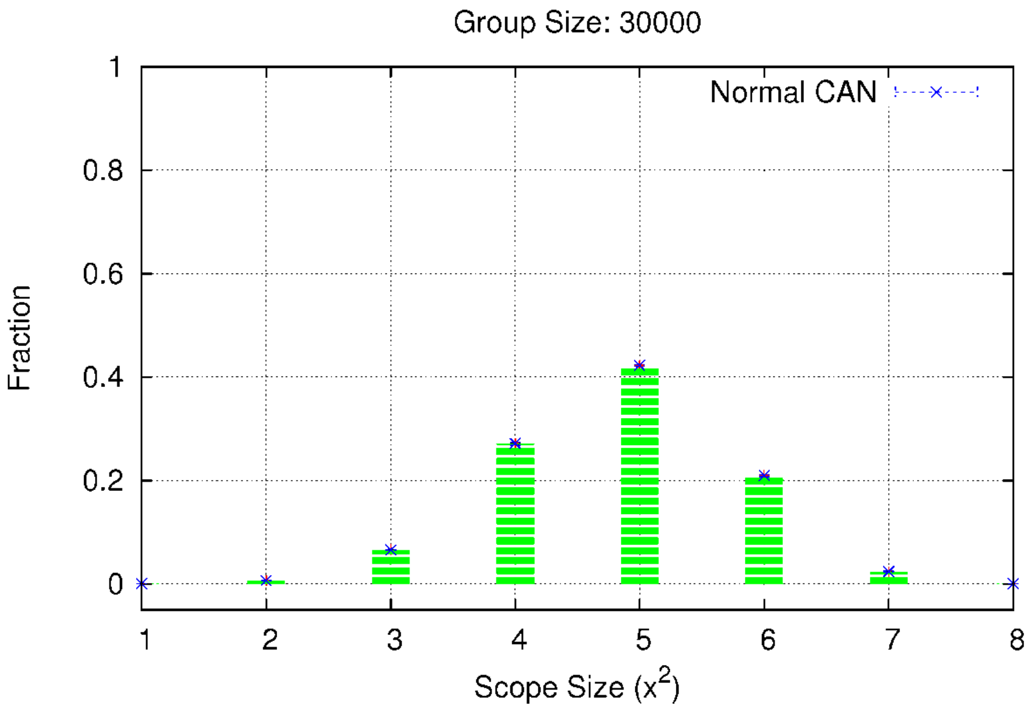


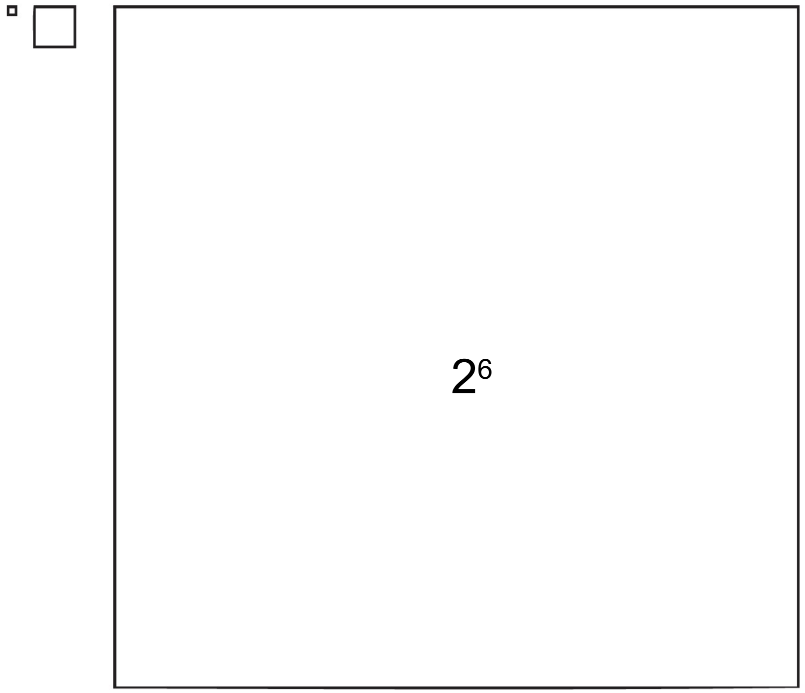


- Heterogeneity of peers! (High-end PC at TU Darmstadt vs. My mobile phone...)

Difference in Area Sizes…

- Nodes in CAN are allocated areas differing up to factor 2^8 in size easily (s.b., only 30k nodes)…





Tiny example for comparison…