Peer-to-Peer and Cloud Computing

Prof. Dr. Jörg Hähner

Part 4: Structured P2P Systems

- Introduction to Distributed Hash Tables
- DHT Routing Geometries and Algorithms:
  - Rings: CHORD
  - Torus: Content-Addressable Networks (CAN)
- Summary

Structured Peer-to-Peer system

- Content(-pointers) placed at precisely specified locations/peers:
  - Peers are responsible for certain ids.
  - Defined mapping from ids to location/peer.
- Topology of overlay network is tightly controlled.
  - Neighbours selected according to their ids.
- Structured overlay network allows for directed routing towards peer storing content(-pointer):
  - Peers implement distributed index for lookup of data items with given ids.
  - Typically implemented as Distributed Hash Table (DHT).

Addressing of data items: key/value-based

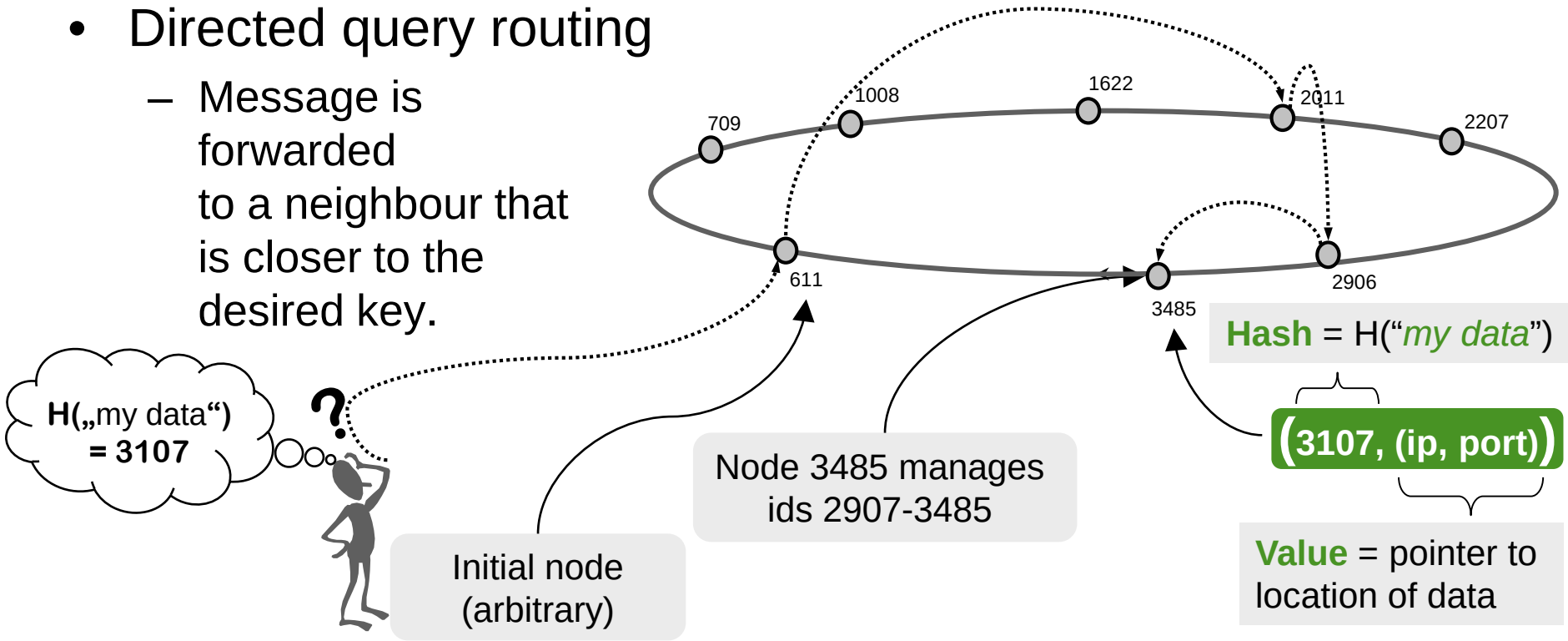
- Each data item is associated with a unique key.
- Value:
  - Data item (direct storage), or
  - Pointer to data item (indirect storage).
- Examples:
  - Key = “Sven Tomforde”, value=“0821-598-4628”.
  - Key = “lecture.ppt”, value=file containing presentation of this lecture → direct storage.
  - Key = “lecture.ppt”, value= URL of file containing presentation of this lecture → indirect storage.

Distributed hash table (DHT)

- Hash value calculated from the data item's key.
  - Example: hash(key = "lecture.ppt") = 80
- Distributed buckets of hash table: peers
- Each peer is responsible for a certain subset of hash values, the corresponding keys and data items.
  - Example: Partitioning of hash values among peers A, B, C
    A: {0,…,100}; B: {101,…,195}, C: {196,…,255}
  - Peer A stores file "lecture.ppt" with hash 80
- Peer only stores data items with correlated hash values.
  - Actual content of data items stored by a peer is uncorrelated since hash values are random.

Distributed Hash Tables (3)

- Routing structure / Distributed index structure
  - Peers have links to certain neighbours that are responsible for certain hashes.
    - Each peer has local view on the distributed hash table.
- Directed query routing
  - Message is forwarded to a neighbour that is closer to the desired key.



- Distributed buckets of hash table: peers
  - N peers → N buckets
- Randomness of hash distributes items evenly among peers.
  - Every peer has roughly the same number of data items.
- **Problem** with standard hashing: changing N has global impact on distribution of data items among buckets.
  - Re-hashing of all data items

  → new distribution among peers

$$h(x) = (a * x + b) \; mod \; N$$

  - Frequent joining and leaving of peers would be critical!

- **Solution:** Consistent hashing
- Properties of consistent hashing:
  - Load balancing
    All peers (buckets) get roughly the same number of data items (similar to standard hashing).
    - Popularity of data item (i.e. actual query load) is not considered.
  - Smoothness
    Peer join/leave the network:
    - $O(K/N)$ keys are moved to/from the peer
    - K is the total number of keys
    - N is the total number of peers.

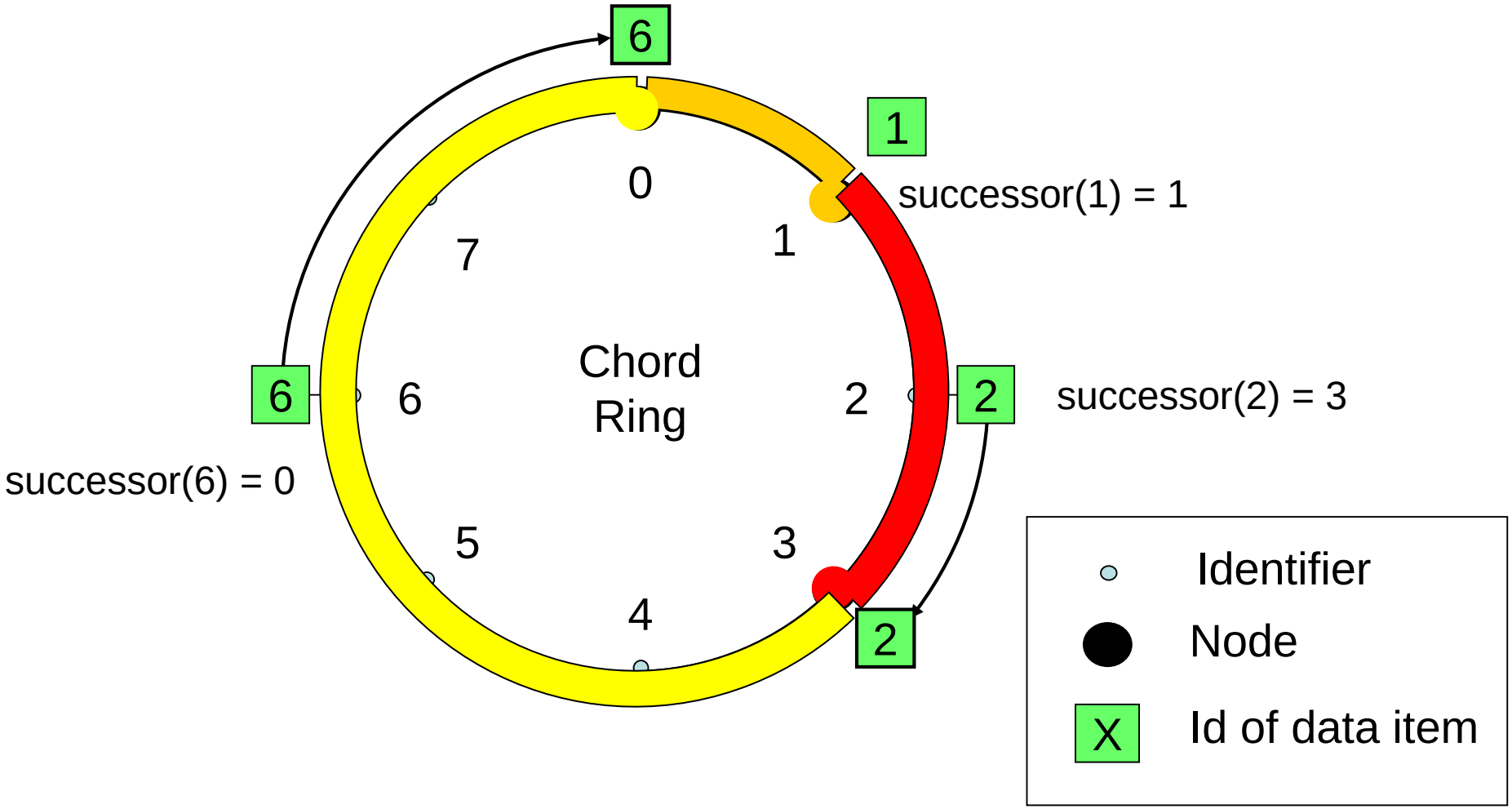
- Introduction to Distributed Hash Tables
- DHT Routing Geometries and Algorithms:
  - Rings: CHORD
  - Torus: Content-Addressable Networks (CAN)
- Summary

Node and Data Item Identifiers

- Chord uses SHA-1 as hash function for assigning identifiers:
  - $0 \leq \text{identifier} < 2^m$ with m = 160
- Data item key is hashed to 160 bit identifier using SHA-1.
- Each node is associated with a 160 bit identifier.
  - SHA-1 hash of IP address & port number.
- Chord uses consistent hashing to assign peers to buckets of DHT.

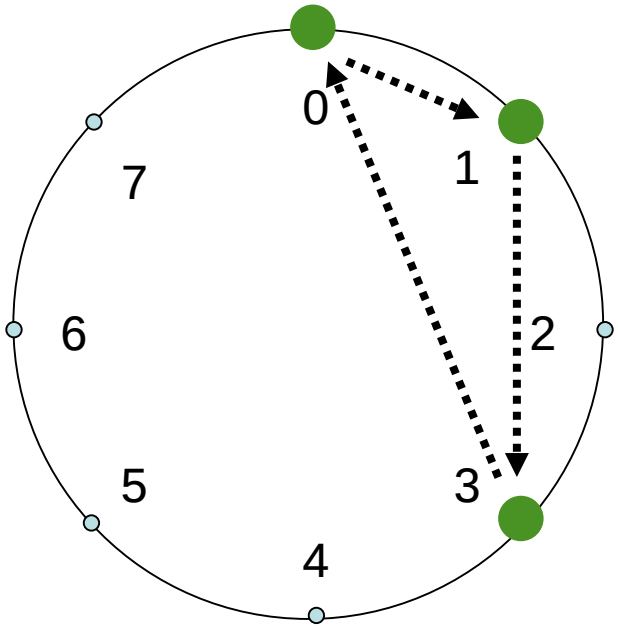
- Identifier ring
  - Identifiers are ordered in an identifier ring.
  - Arithmetic operations in ring: $\text{modulo } 2^{160}$
- Position of node in ring is defined by its identifier
  - Peers are distributed uniformly in ring.
  - This is the result of the random hash function.
- Associating data items to peers
  - Data item with identifier i is managed by the peer that is the clockwise successor of i in the identifier ring.
    → successor node of identifier i: $\text{successor}(i)$

Basic Topology (2)

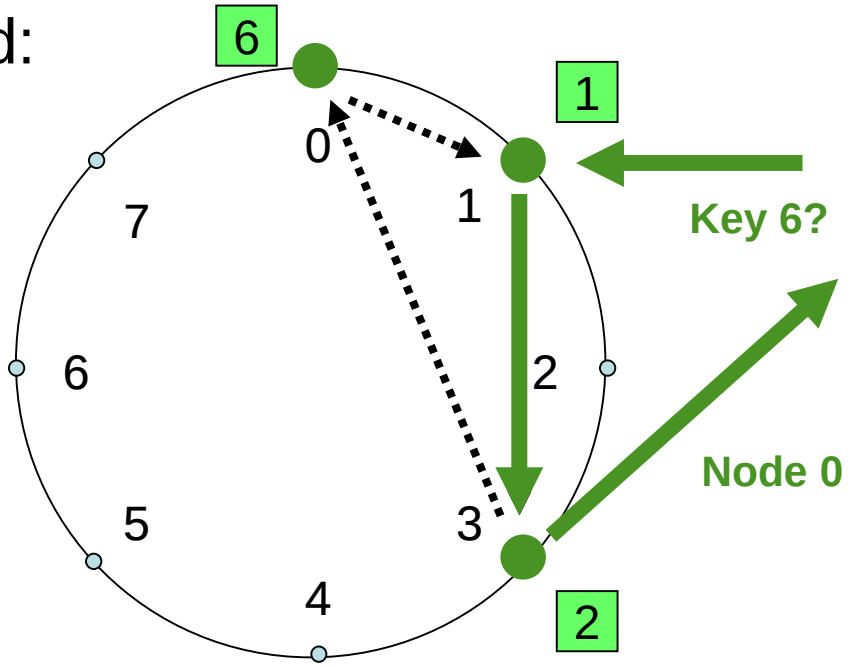


- Overlay network topology determined by links between nodes:
  - Link: Knowledge about another node
  - Stored in routing table on each node.

- Simplest topology: Ring
  - Each node n has a link to clockwise next node.
    → successor of n
  - Note: Don't mix up successor of n and successor(i)!



Forward query clockwise to successor node until successor(k) is found:



Algorithm executed by node n:

```
n.findSuccessor(id):
if id ∈ ]n,successor]
    return successor
else
    return successor.findSuccessor(id)
```

Performance of this Simple Routing Algorithm

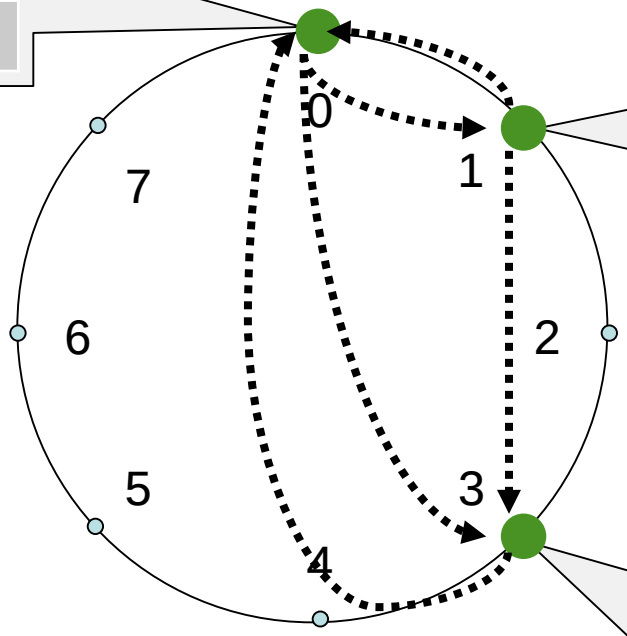
- Small local state: $O(1)$
  - Each node has only one link to its successor.
- High lookup latency: $O(N)$
  - On average, message traverses $N/2$ nodes.
- Improvement:
  - Increase local state
  - Similar idea as Kleinberg model: Add long range contacts → Finger table

Finger Table

- **Idea:** Add limited number of long range contacts.
- Let m be the number of bits of node and key identifiers.
- Each peer adds at most m long range contacts called "fingers":
  - **Finger**: mapping from identifier to successor node address. id=28 → ip=192.168.3.5, port=5000
  - Finger table (routing table) contains at most m fingers.
- Selection of fingers:
  - Let n be the identifier of the peer.
  - The i-th entry in the peer's finger table points to $\text{successor}(n + 2\text{^}i\text{-}1)$ in the circular ring space.

Finger Table (2)

i	Start	Node
1	1	1
2	2	3
3	4	0



i	Start	Node
1	2	3
2	3	3
3	5	0

i	Start	Node
1	4	0
2	5	0
3	7	0

Routing with Finger Tables

Each node n forwards a query for id clockwise

- to farthest finger preceding k
  - No overshooting for correct routing!
- until $n = \text{predecessor}(\text{id})$ and successor of $n = \text{successor}(\text{id})$
  - Then n returns $\text{successor}(\text{id})$ to requestor.

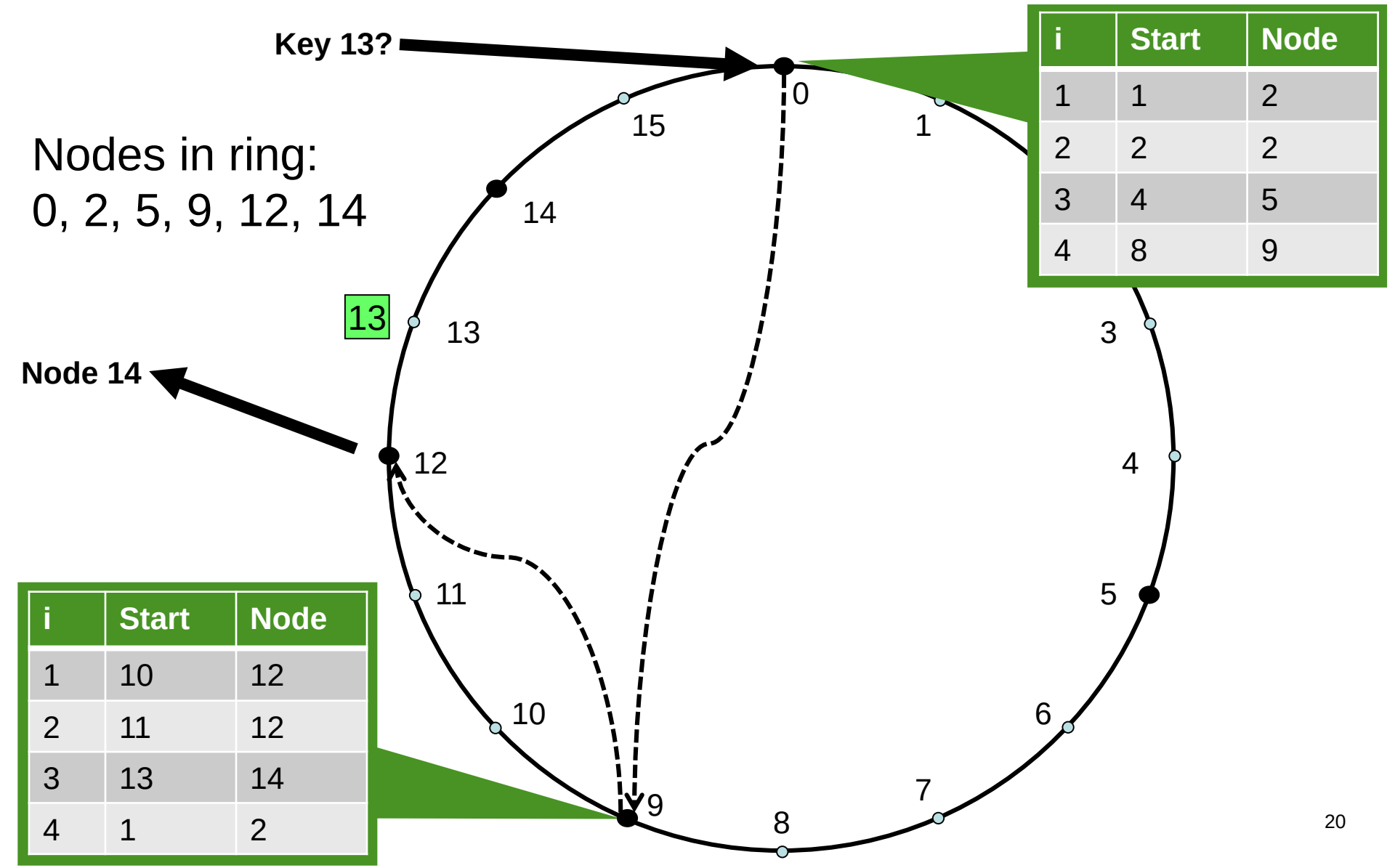
```
n.findSuccessor(id):

if id ∈ ]n,successor]
    return successor
else
    n' = n.closest_preceding_node(id)
    return n'.findSuccessor(id)
```

```
n.closest_preceding_node(id):

  for i = m..1
    if finger[i].node ∈ ]n,id[
       return finger[i].node
```

Example: Routing with Finger Tables



i	Start	Node
1	1	2
2	2	2
3	4	5
4	8	9

i	Start	Node
1	10	12
2	11	12
3	13	14
4	1	2

Performance of Finger Table Routing

- Local state (number of fingers):
  - $O(m)$
  - If nodes are distributed randomly (which can be assumed using a random hash): $O(\log N)$ with N number of nodes
- Lookup latency (routing “hops”)
  - Distance to destination node halfed in each step
    - Distance means: distance in identifier ring
    - At most m steps if ring is completely populated
  - If nodes are distributed randomly
    - $O(\log N)$ with N number of nodes

Node Joins

- Number of Chord nodes is dynamic:
  - Nodes may join or leave at any time.
- For correctness of routing, the following must hold:
  - Each node maintains its correct successor.
  - For every k, node $successor(k)$ stores k.
- For efficiency (short paths), the finger table must be correct:
  - Adding a new node without updating the finger table does not lead to “overshooting“ in routing → routing still correct
    - The old finger still points to a predecessor of k.
  - Remember Kleinberg’s model: Only if the ratio of long and short fingers is correct, the greedy routing algorithm is efficient.

- Each node also maintains a predecessor pointer:
  - IP address & port number & identifier of predecessor in ring.
  - Required to ensure ring validity (see stabilisation algorithm).
- Tasks to be executed when node n joins:
  - Initialise predecessor, successor, and fingers of n.
  - Update predecessors, successors, and fingers of other nodes.
  - Transfer data items from n's successor to n.
    - Following the principle of consistent hashing, only a small number of data items have to be moved.
- Assumption:
  - n knows some node n' already part of the Chord ring
    → bootstrapping!

Entering Ring & Updating Local Finger Table

```
n.initFingerTable(n‘):
  n.succ = finger[1].node =
    n‘.findSuccessor(finger[1].start)
  n.pred = n.succ.pred
  n.succ.pred = n
  n.pred.succ = n

for i = 2..m
    if finger[i].start ∈ ]n,finger[i-1].node]
      finger[i].node = finger[i-1].node
    else
      finger[i].node =
      n‘.findSuccessor(finger[i].start)
```

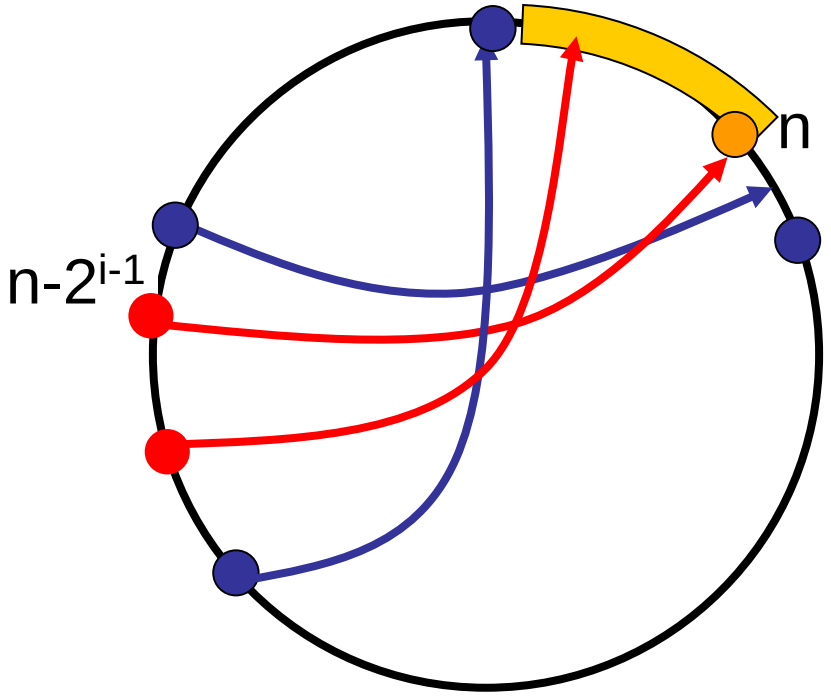
Optimisation:

- Do not search via n‘ if finger node is already known
- Reduces communication costs from O(*m* log *N*) to O($\log^2 N$)
- O(log *N*) cost if *n* copies its successor’s finger table

Updating Finger Tables of Existing Nodes

```
// executed by joining node
n.updateOthers:
  for i = 1..m
    p = findPredecessor(n-2^(i-1))
    p.updateFingerTable(n,i)

n.updateFingerTable(s,i):
  if s ∈ ]n,finger[i].node]
    finger[i].node = s
    predecessor.
       updateFingerTable(s,i)
```



Possible effects of failing nodes or concurrent joins:

1) Finger tables and successors are correct:
   - Everything works as expected.
2) Successors are correct, finger tables are incorrect:
   - Performance degradation (longer paths), but correct routing.
3) Successors are incorrect:
   - Query may not be forwarded to target node.
   - Application has to repeat query after some time.

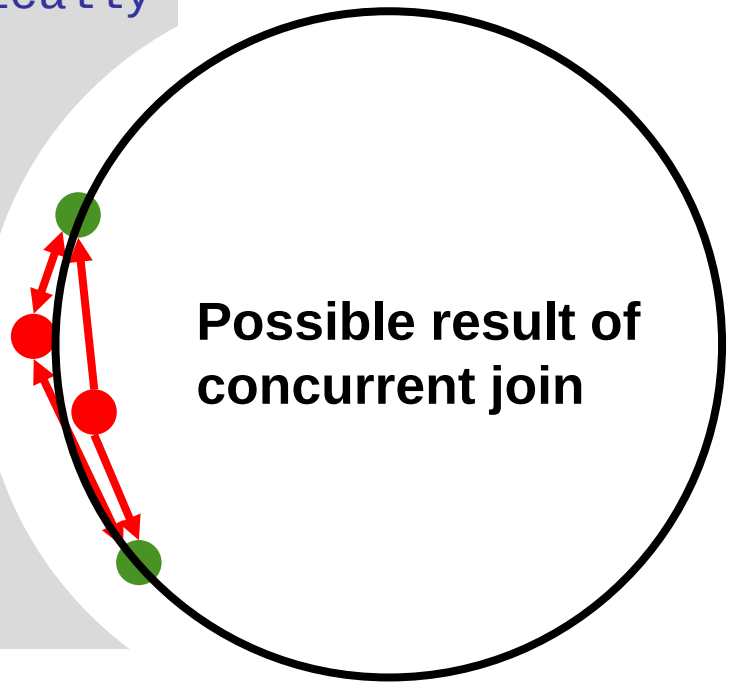
- Presented join protocol does not assure correctness for concurrent joins!
- Stabilisation protocol to assure robustness for dynamic node populations and concurrent joins and leaves.

Stabilisation Protocol

- Concurrent joins may lead to incorrect successor and predecessor entries.
- Stabilisation protocol ensures integrity of ring for correct routing:

```
n.stabilise(): // executed periodically
  p = successor.predecessor
  if p ∈ ]n,successor[
    successor = p
  successor.notify(n)

n.notify(n‘):
  if n‘ ∈ ]predecessor,n[
          || predecessor == nil)
    predecessor = n‘
```



Stabilisation protocol updates finger tables for efficient routing:

```
// executed periodically
n.fixFingers():
  i = random number in [2,...,m]
  finger[i].node = find_successor(finger[i].start)
```

Note: routing is correct even if fingers are incorrect!
→ Correct successor will be found.

Leaving Nodes

- Node leaves Chord gracefully.
- Leaving node notifies predecessor and successors.
  - Notification tells predecessor about new successor, and successor about new predecessor.
- Predecessor and successor update their successor and predecessor, respectively.
- Stabilisation protocol updates finger tables.

Failing Nodes

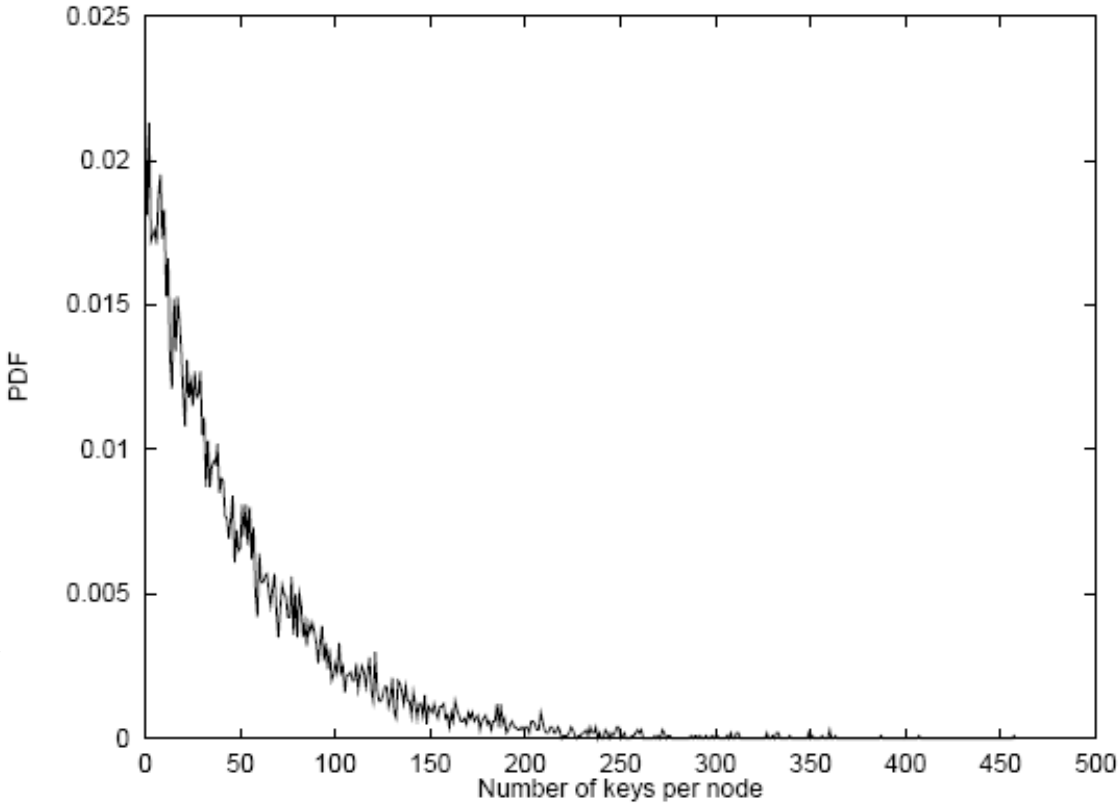
- Each node remembers a fixed small number (*i*) of additional nodes following its successor in the ring.
  - Nodes receive successor list of their successor and remove last entry.
- Each node checks liveness of its successor and predecessor.
  - (a) Periodically, or (b) reactively on forwarding messages.
- predecessor fails, it is simply removed (predecessor = nil).
- successor fails, the successor of the successor becomes the new successor.
  - If the successor of the successor has also failed, then the successor of the successor of the successor, etc.
  - Bootstrapping if more than *i* nodes in sequence fail concurrently (very unlikely).
- Stabilisation protocol updates successors, predecessors, and finger tables.

- When nodes fail
  - stabilisation assures correct routing, however,
  - stabilisation does not assure that content is still available.
- Content must be transferred to new successor of failed node.
- Basic approaches:
  - Content owner sends periodic content updates:
    - - Communication overhead even if object is not requested.
    - - Content not available until update.
    - + No storage overhead.
  - Replication of content in overlay network:
    - + Content immediately available in network after node failure.
    - - Storage overhead for replicas.

- Multiple logical Chord networks on physical nodes:
  - Different hash functions for each network.
    - Each node has different ids in each network.
      → Object replicated at different physical nodes.
  - *n* replicas of each object with *n* logical networks.
  - Parallel or sequential search in logical networks.
- One Chord network with replicas at different nodes:
  - Replication at multiple successors of object.
    - For ring robustness, each node already knows addresses of multiple successors.
    - If successor fails, successor of successor can take over immediately.
  - *n* replicas of each object if object is replicated at *n*-1 successors.

Evaluation: Load Balancing

- Ideal: consistent hashing
  - Number of keys managed by each peer should be about K/N
- Simulation:
  - 100,000 nodes
  - 10^5-10^6 keys
- Result:
  - Load is *not* evenly distributed
  - Max. 9.1 x mean
  - Reason: Random node id distribution (hash) does *not* lead to equally sized buckets for each peer!



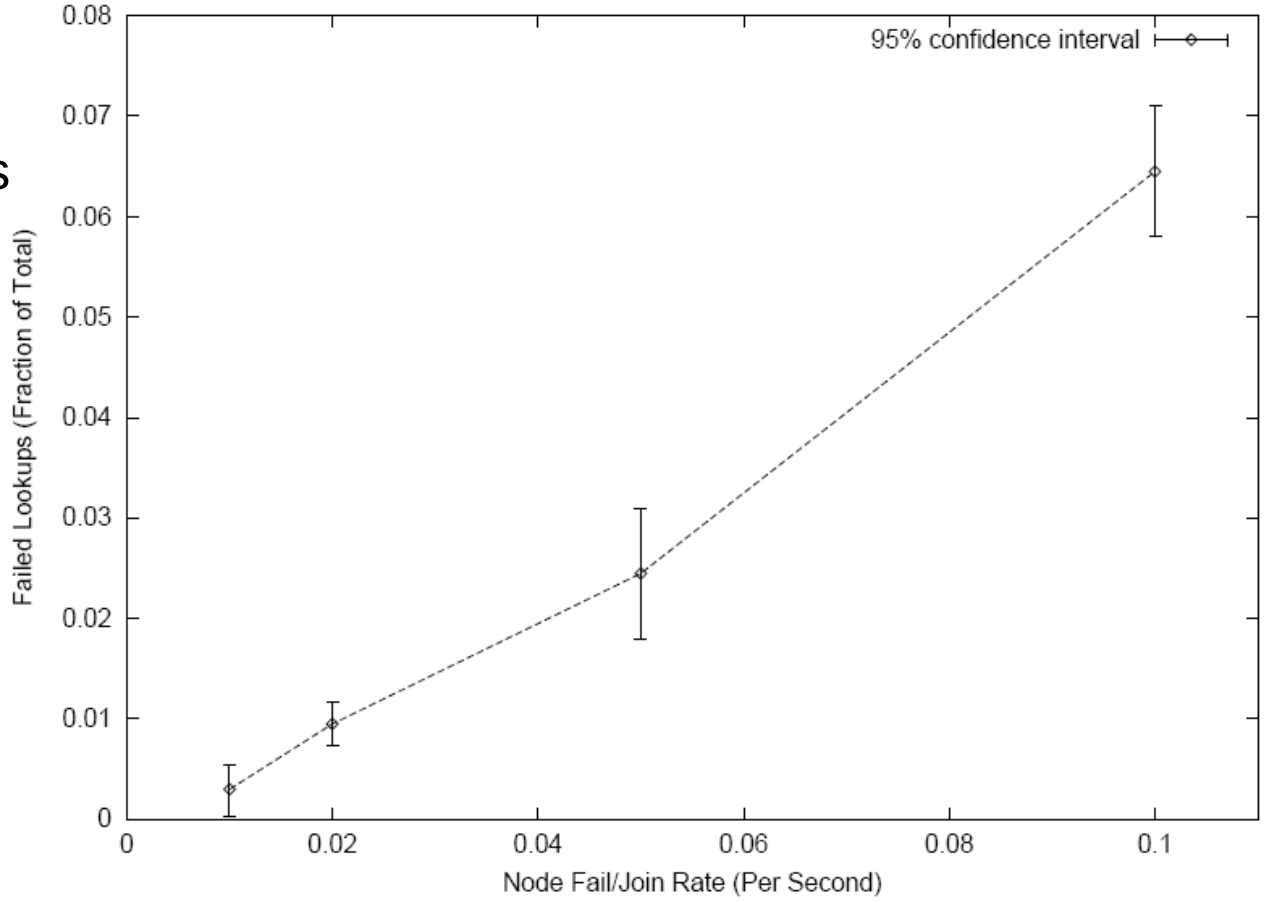
[Stoica, Morris, Karger, Kaashoek, Balakrishnan: „Chord: a scalable peer-to-peer lookup service of internet applications.“ ACM SIGCOMM, 2001]

Evaluation: Load Balancing (2)

- Load of a node may exceed average by a factor of $O(\log N)$
- One solution: virtual nodes
  - Each physical node implements small number (e.g. $\log N$) of virtual nodes with independent identifiers.
  - Increases average path length to $O(\log(N \log N))$.
    - Still reasonably short path lengths.
  - Increases local state (routing table) of physical node to $\log^2 N$.
    - Still reasonably small.
  - Decreases load imbalance significantly.

Robustness under Churn

- Simulation:
  - Each node runs stabilisation every 30s.
  - Network size: 500 nodes.
  - Churn rate is varied.
- Results:
  - 6.5% failed lookups if one node fails/joins every 10s on average.

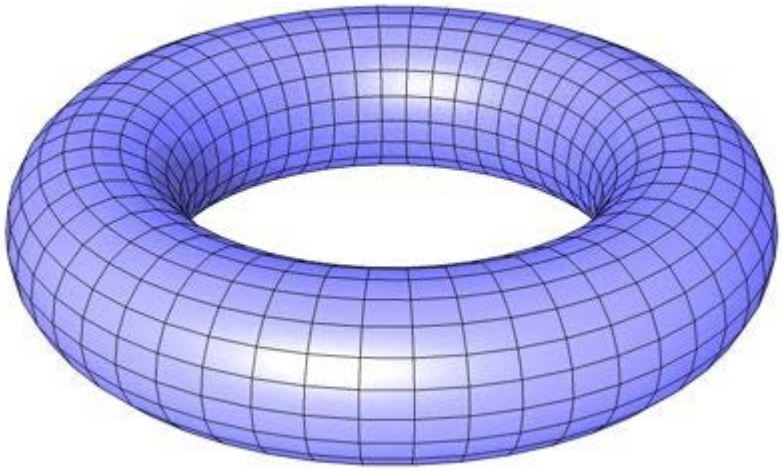


- Introduction to Distributed Hash Tables
- DHT Routing Geometries and Algorithms:
  - Rings: CHORD
  - Torus: Content-Addressable Networks (CAN)
  - Plaxton trees
- Summary

- Goal: Internet-scale distributed hash-table functionality
  - Keys are associated with values (content)
  - Each peers stores certain values
  - Basic CAN functionality: Lookup of values associated with keys

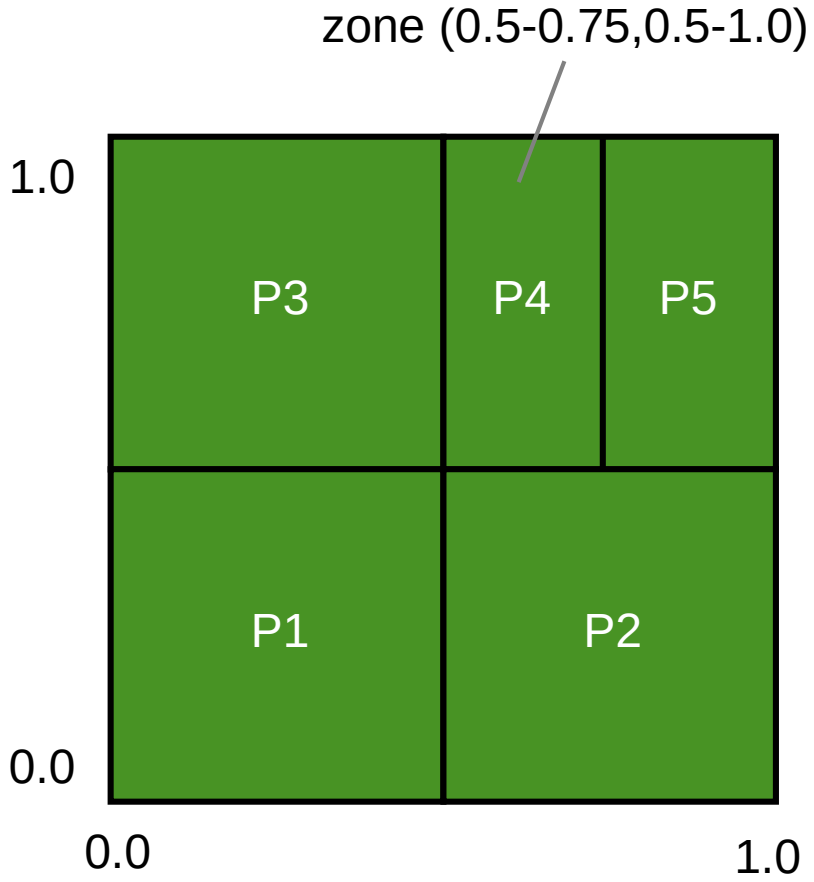
- Routing geometry: Torus
  - d-dimensional Cartesian coordinate space

    $d = 2$: $[0,1] \times [1,0]$
  - Coordinates wrap around at the edges → torus



Routing Geometry: Torus

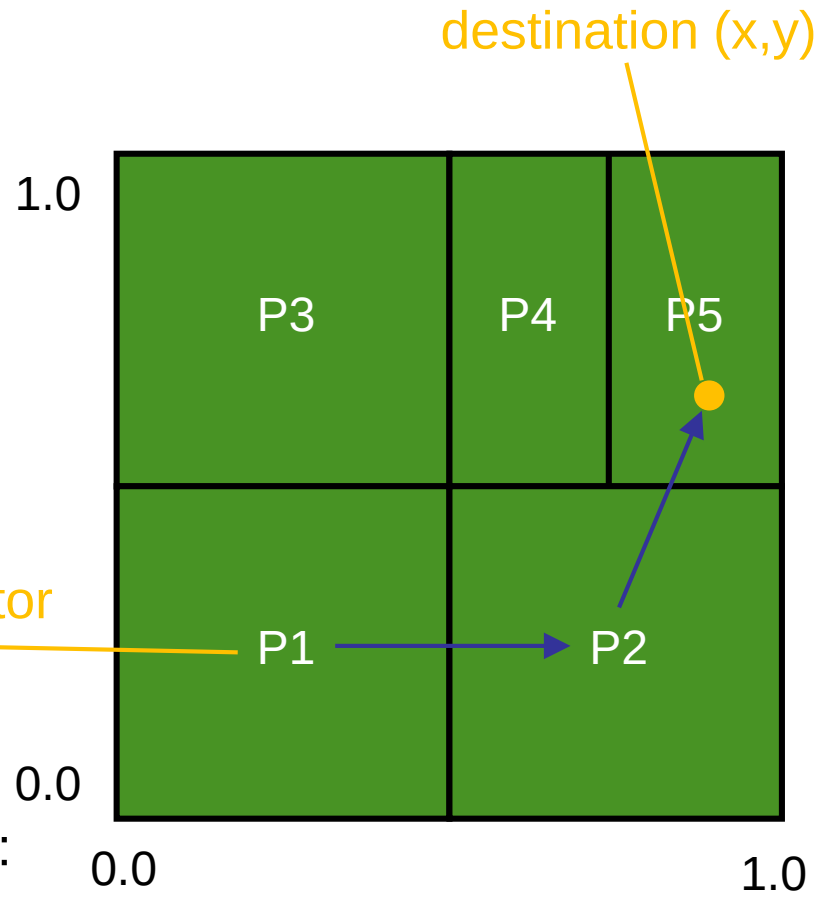
- Coordinate space is purely virtual.
  - No relation to geography of physical world (geographic location of peer).
- Keys are mapped to point in space.
  - Deterministic hash function
- Space is partioned into zones.
  - Each zone owned by one peer.
- Peer stores (key,value) pair if key is mapped to its zone.



Note: the coordinates space wraps around at the “edges“ (not shown in the figure).

Torus-based Routing

- Requestor calculates key coordinates
  - Coordinates are included in query message
- Query message forwarding: Greedy "geographic" forwarding
  - Message is sent to neighbour closest to destination.
- Routing table of node:
  - Entry: IP address and zone coordinates of neighbour.
  - Neighbours (d-dimensional space):
    - Zones overlap along *d*-1 dimensions
    - Zones about along 1 dimension



- Average neighbour number for *d*-dimensional space and *n* equal zones:
  - 2*d* neighbours → $O(1)$ for fixed d
  - Total number of nodes can be increased without increasing per node state (in contrast to Chord & Tapestry).
- Average path length:
  - $(d/4)(n^{1/d})$
    - $n^{1/d}$ nodes along each dimension
    - Average distance to destination along each dimension $(1/4)(n^{1/d})$
  - Path length scales as $O(n^{1/d})$
    - Longer paths than Chord & Tapestry
    - Same scaling properties as Chord &Tapestry if $d = (\ln n)/2$
      → However, then node degree is not independent of *n*

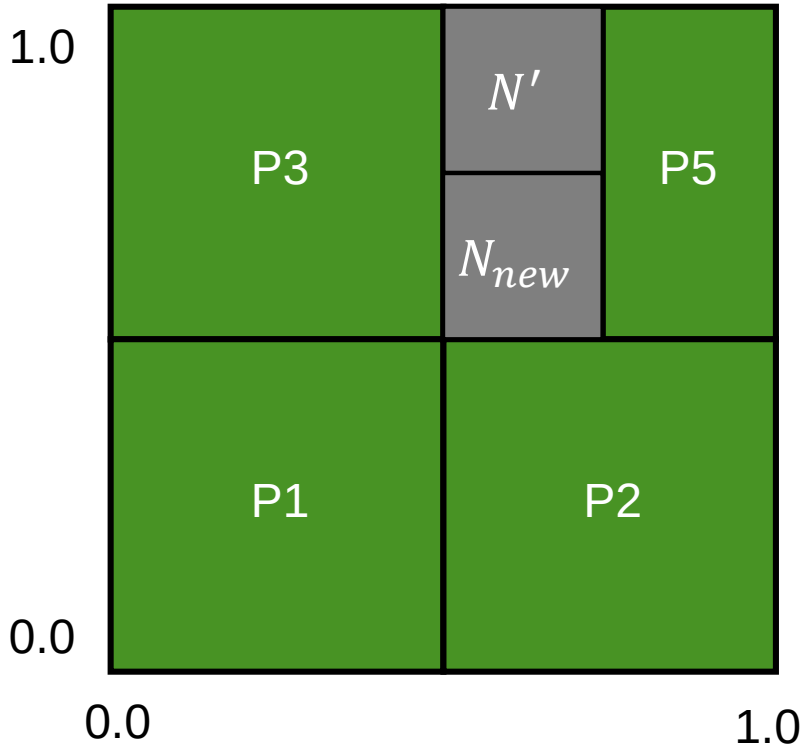
1. Bootstrapping: New node must find some node being part of CAN!
   - “Out-of-band signalling”: Any bootstrap mechanism usable.
2. Assign zone to new peer:
   - Some peer must split its zone.
   - New node gets half of split zone.
3. Update neighbour routing tables:
   - Neighbours of split zone must update their routing tables.

Zone Assignment

- New node N_{new}
  - randomly chooses a point *P* in the space.
  - sends message to *P* using greedy CAN routing.
- Message is received by node *N'* of zone containing *P.*
- *N'* splits its zone in half:
  - Zones are split according to defined ordering of dimensions.
  - 2D example: first split along X axis, then along Y, then along X, …
- Old node keeps half of the zone.
- New node gets half of the zone:
  - Migration of (key,value) pairs within N_{new} 's zone from *N'* to N_{new}.

Neighbour Table Updates

- N_{new} learns its neighbours from N‘:
  - N‘ + subset of neighbours of N‘
  - Example: N’, P3, P5, P2
- N’ adds N_{new} as neighbour and removes non-neighbours:
  - Example: N_{new}, P3, P5, P2
- Immediate update messages are sent by N_{new} and N‘ to neighbours:
  - Neighbours update routing table
- Update affects only O(d) nodes:
  - Overhead independent of (possibly large) n



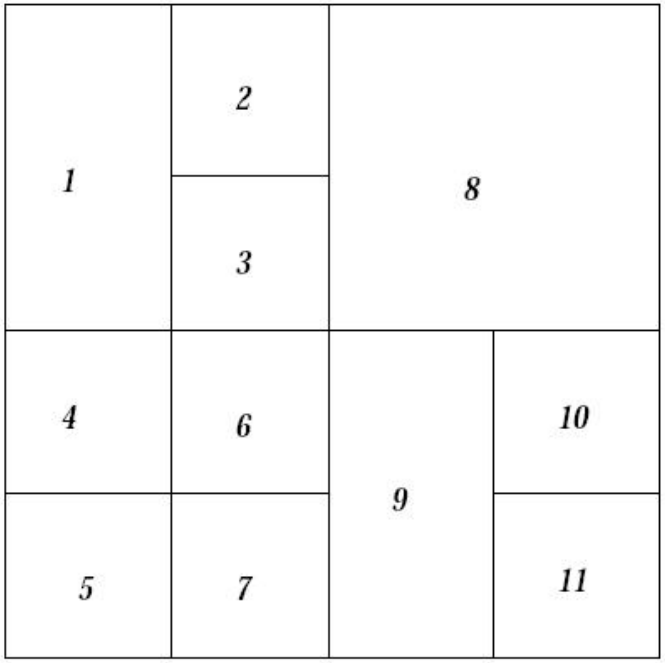
Let node $N_{leaving}$ leave the network gracefully:

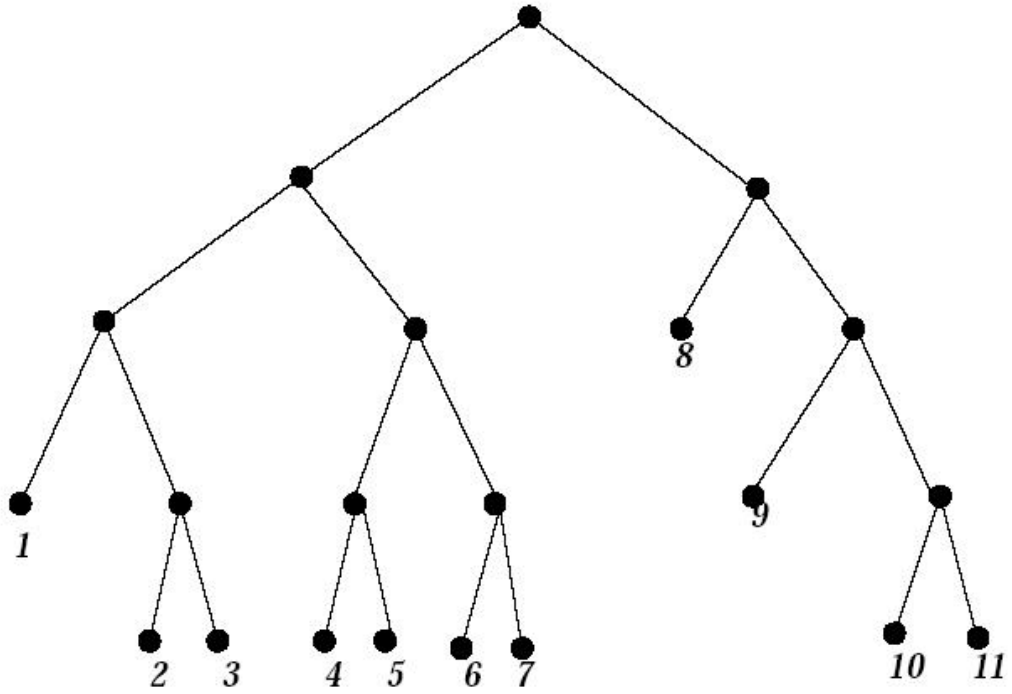
- $N_{leaving}$ hands over its zone to a neighbouring node called takeover node (formally defined later):
  - (key,value) pairs are migrated from $N_{leaving}$ to takeover node.
  - Neighbours of $N_{leaving}$ are notified and update their routing tables.
- $N_{leaving}$'s zone is merged with takeover node's zone if this leads to a valid zone.
- If no valid merging is possible:
  - Takeover node temporarily handles two zones.
  - Zones can be re-arranged such that every node handles only one zone.

- Nodes send periodic update messages to neighbours:
  - Failure of node N_{dead} is discovered by prolonged absence of update message.
- If failure is discovered by node N':
  - N' removes N_{dead} from neighbour list.
  - N' forwards *recovery message* to takeover node.
- Questions:
  - How to identify takeover node *consistently*?
  - How to route recovery message to takeover node *reliably*?

Identification of Takeover Node

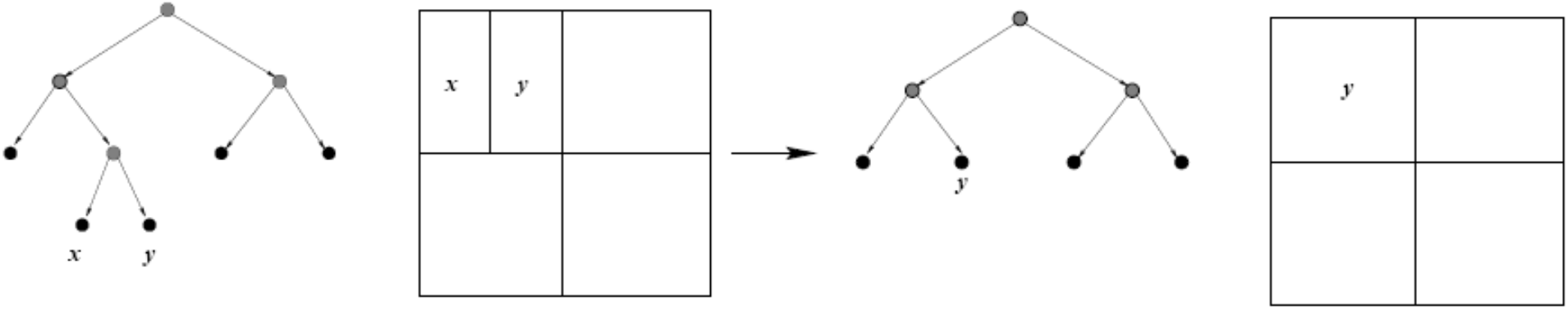
- Takeover node: well-defined node that takes over the zone of a leaving/failed node $N_{leaving}$.
- (Conceptually) defined by partition tree:





Identification of Takeover Node (2)

- If sibling *y* of $N_{leaving}$ is a leaf, then *y* is the takeover node.
- $N_{leaving}$'s and the takeover node's (y's) zones are merged.
- Example:

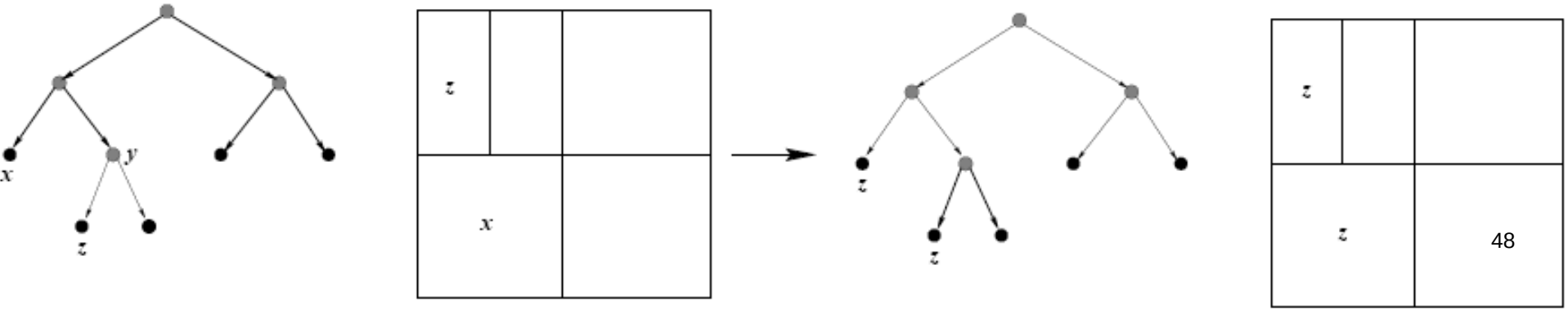


Note: x = $N_{leaving}$

Identification of Takeover Node (3)

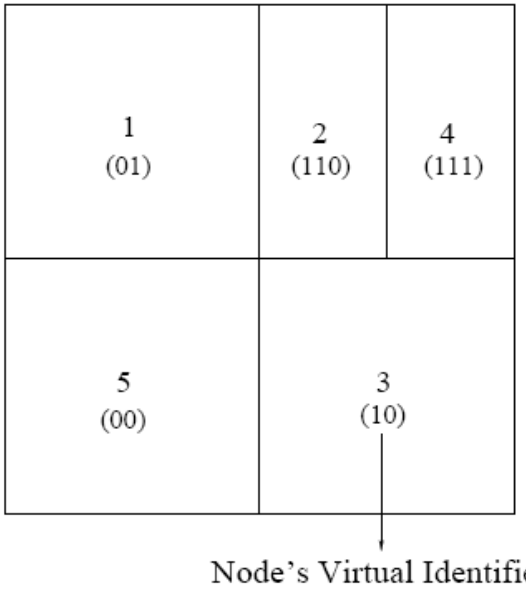
- If sibling *y* of $N_{leaving}$ is no leaf, then start depth first search in sub-tree rooted at *y* until a leaf node z is found:
  - z becomes the takeover node.
  - $N_{leaving}$'s zone and z's zone cannot be merged!
    - Alternative 1: z hands over one zone when a new node joins at z.
    - Alternative 2:
      - z takes over $N_{leaving}$'s zone.
      - Sibling of z merges its zone with z's zone.
- Example:

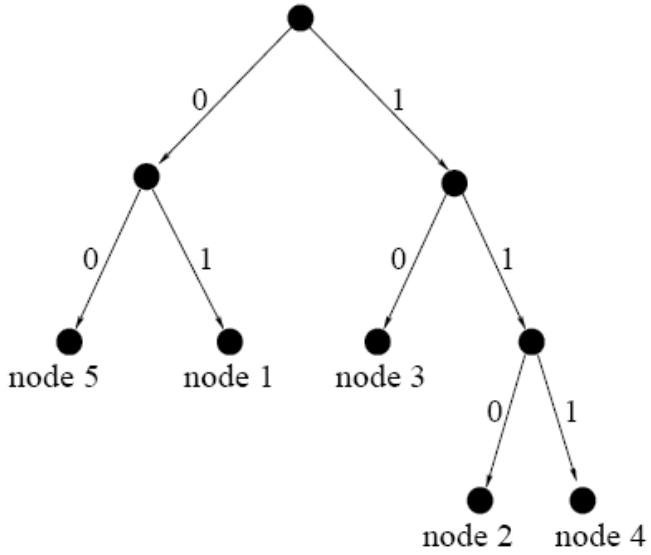
Note: x = $N_{leaving}$



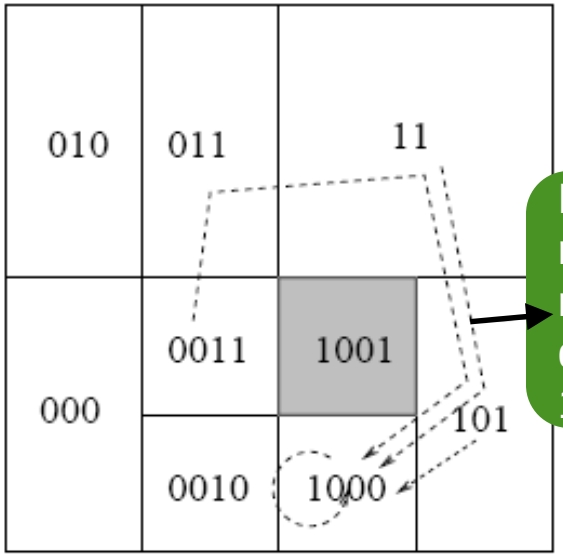
Identification of Takeover Node (4)

- Nodes do not explicitly store the partition tree.
- Each node has a Virtual ID (VID) identifying its position in the tree:
  - Binary string defining the path from the root of the partition tree to the node (assigned during join).
- VID identifies takeover node.
  - Node with VID numerically closest to VID of $N_{leaving}$.





- If N‘ discovers a neighbor failure, it forwards a *recovery message* to the takeover node
  - Task: Forward message to node with VID closest to VID of N_{failed}.
  - N’ does not know VID or zone of takeover node a priori.
- Forwarding Algorithm 1: Greedy forwarding on VIDs
  - Each node forwards recovery message to the neighbour whose VID is closest to N_{failed}‘s VID.
  - Does not always work correctly!



Neighbours‘ route recovery messages to destination 1001

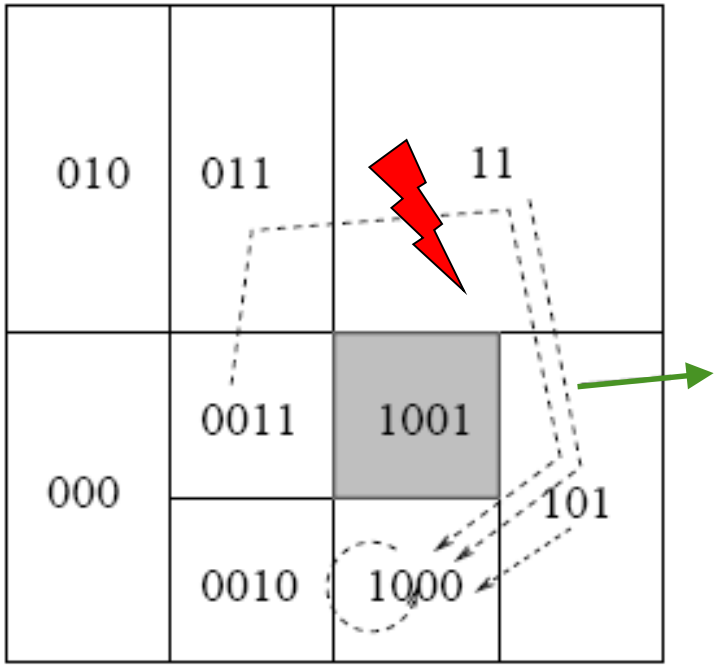
Node 1001‘s neighbours route recovery messages towards VID=1001;

Recovery messages terminate at takeover node 1000.

- Each neighbour of N_{failed} independently sends a recovery message:
  - Takeover node learns about all neighbours of failed node
  - Takeover node updates its own neighbour table
  - Takeover node notifies neighbours to update their neighbour tables

- Without precaution, greedy forwarding of recovery messages might not reach the (same) takeover node → inconsistent decision!
- Example: 1001 and 11 fail simultaneously
  - Message from 0011 terminates at 011
  - Message from 101 terminates at 1000

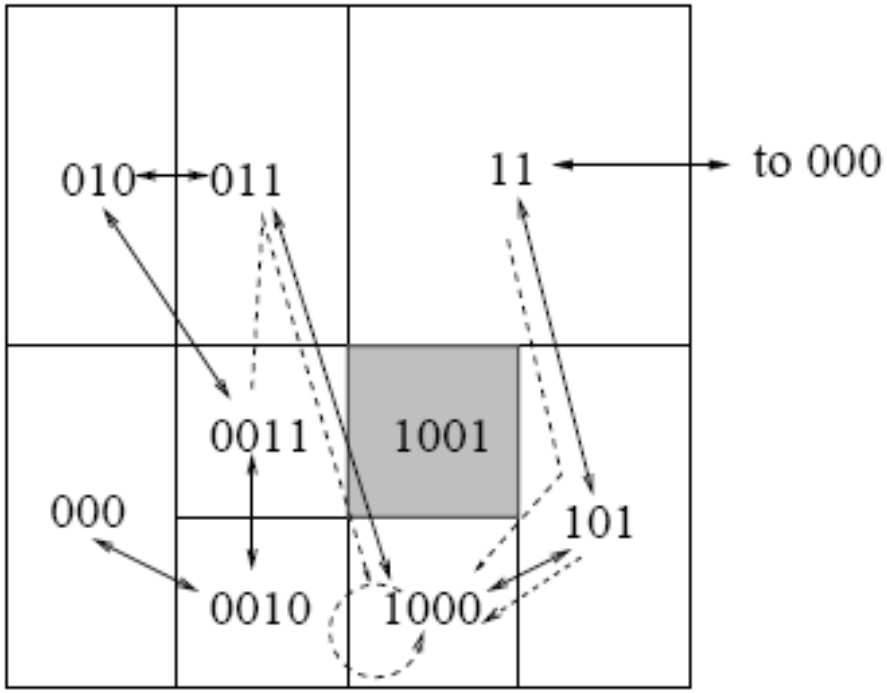
≠



Neighbours' route recovery messages to destination 1001.

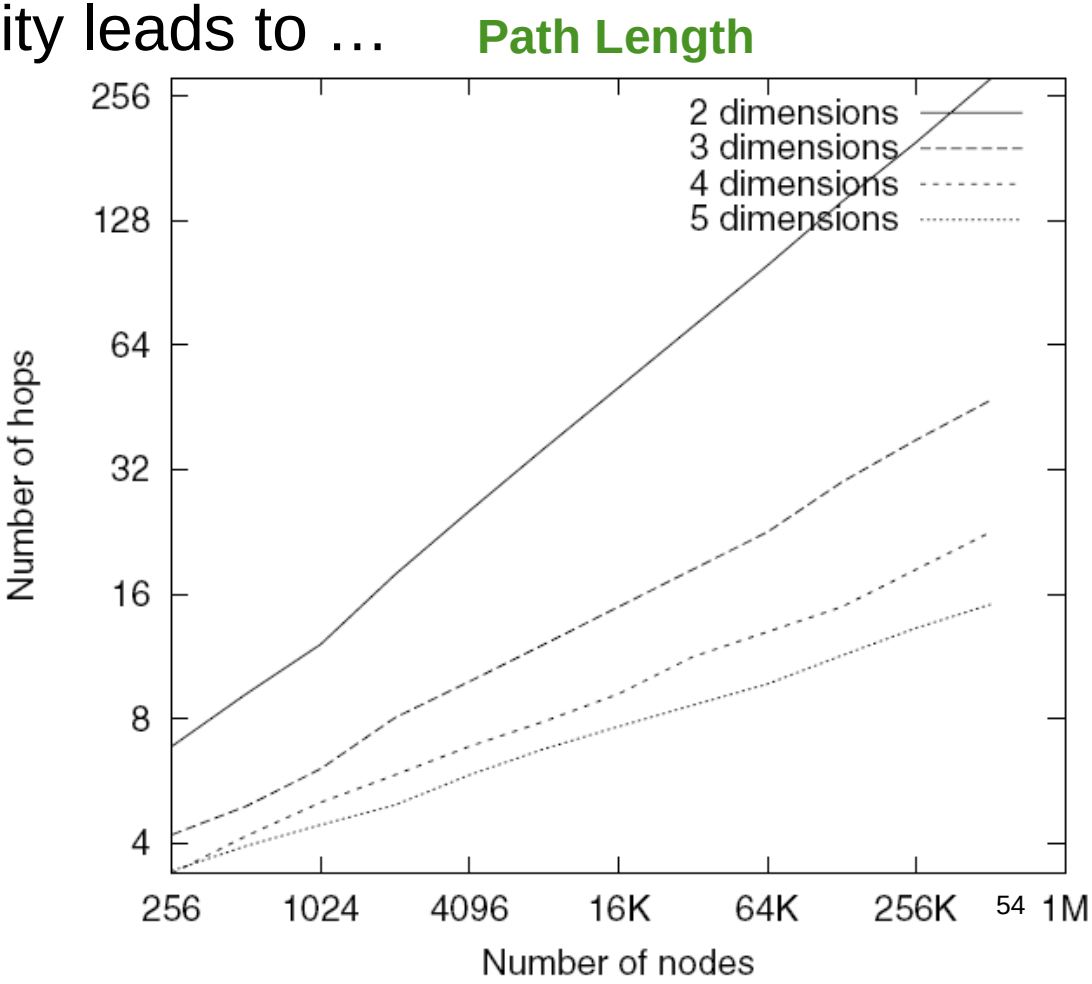
Idea borrowed from Chord:

- Nodes link to successor and predecessor according to VIDs
- Forwarding Algorithm 2:
  - Greedy forwarding along ring
- Linked list is maintained even if nodes fail:
  - Stabilisation protocol similar to Chord.
  - Path between any two nodes exist.
    → Recovery message reaches takeover node.



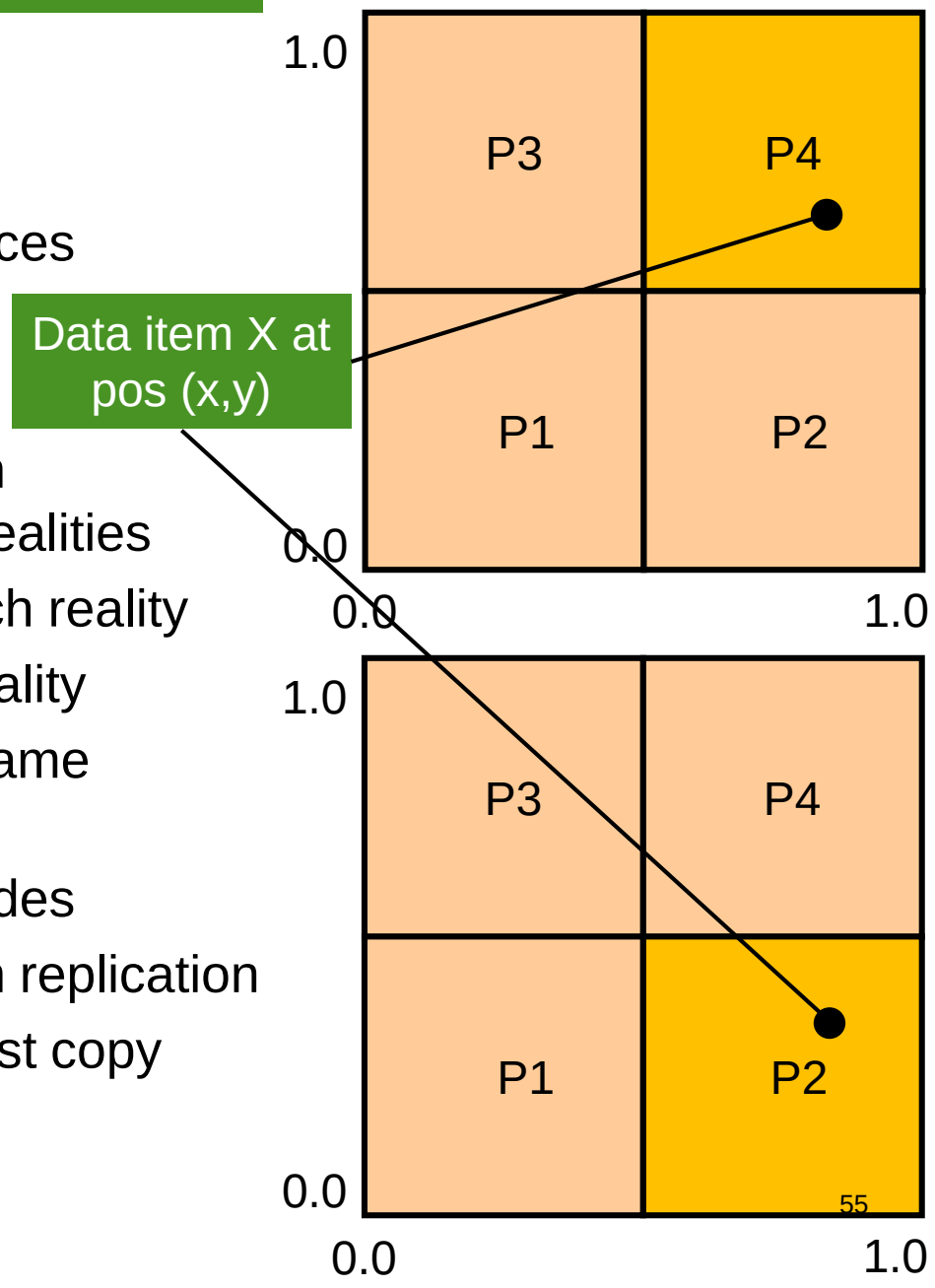
Ordered link list of VIDs maintained in the face of node failures using Chord‘s stabilisation algorithm.

- CAN does not restrict dimensionality
- Increased dimensionality leads to …
  - … shorter paths
    → path length scales as $O(d(n^{1/d}))$
  - … increased per node state → small additional number of neighbours
  - … improved fault tolerance
    → Alternative paths

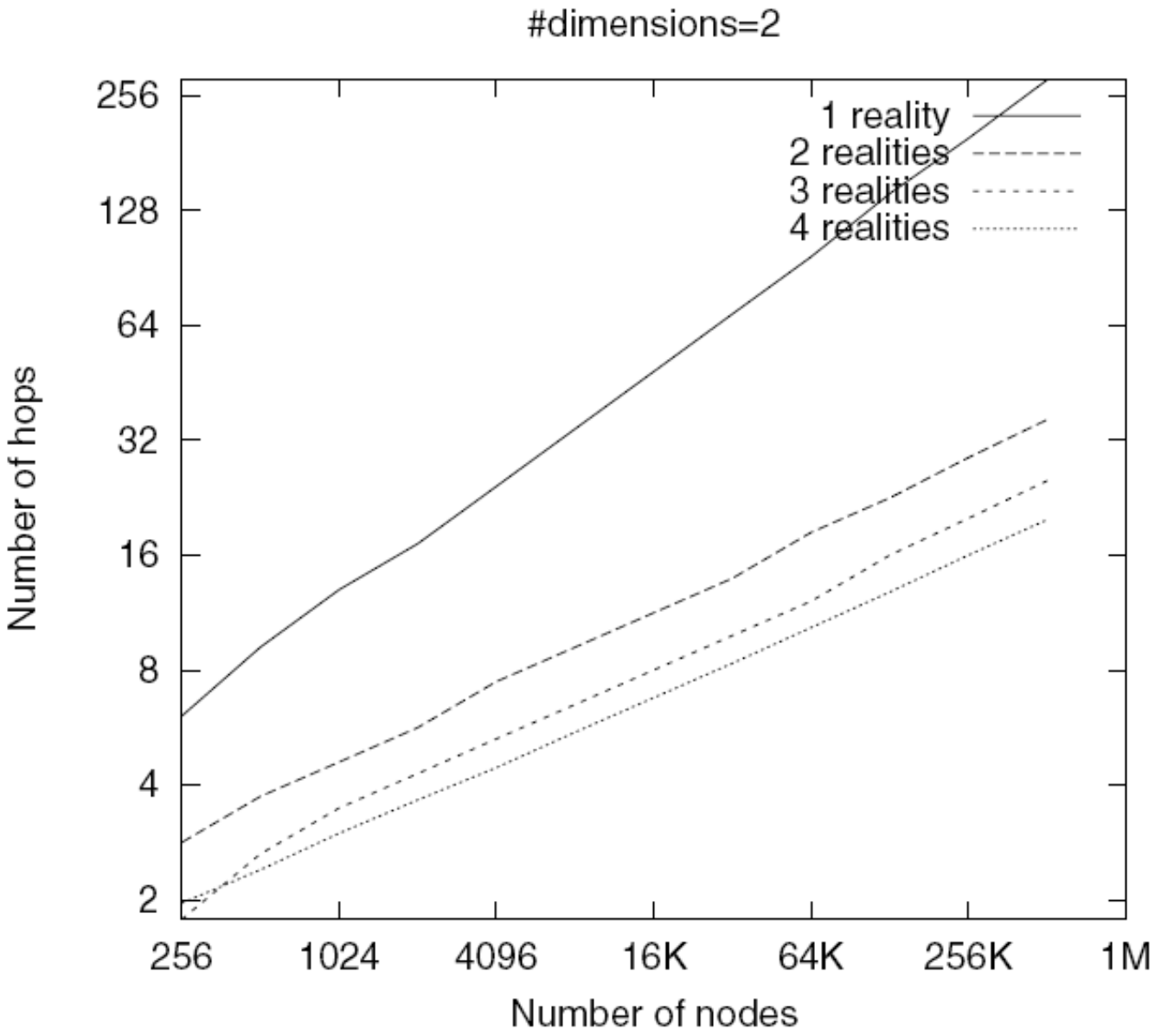


Optimisation 2: Multiple Realities

- Usage of multiple coordinate spaces
  - Each space one “reality“
- Each node is part of every reality
  - Increased robustness through alternative paths in different realities
- Nodes own different zones in each reality
- Data item is replicated in each reality
  - Data item key is mapped to same coordinate in each reality
  - Item replicated at different nodes
  - Increased robustness through replication
- Node routes query towards closest copy
  - Shorter paths

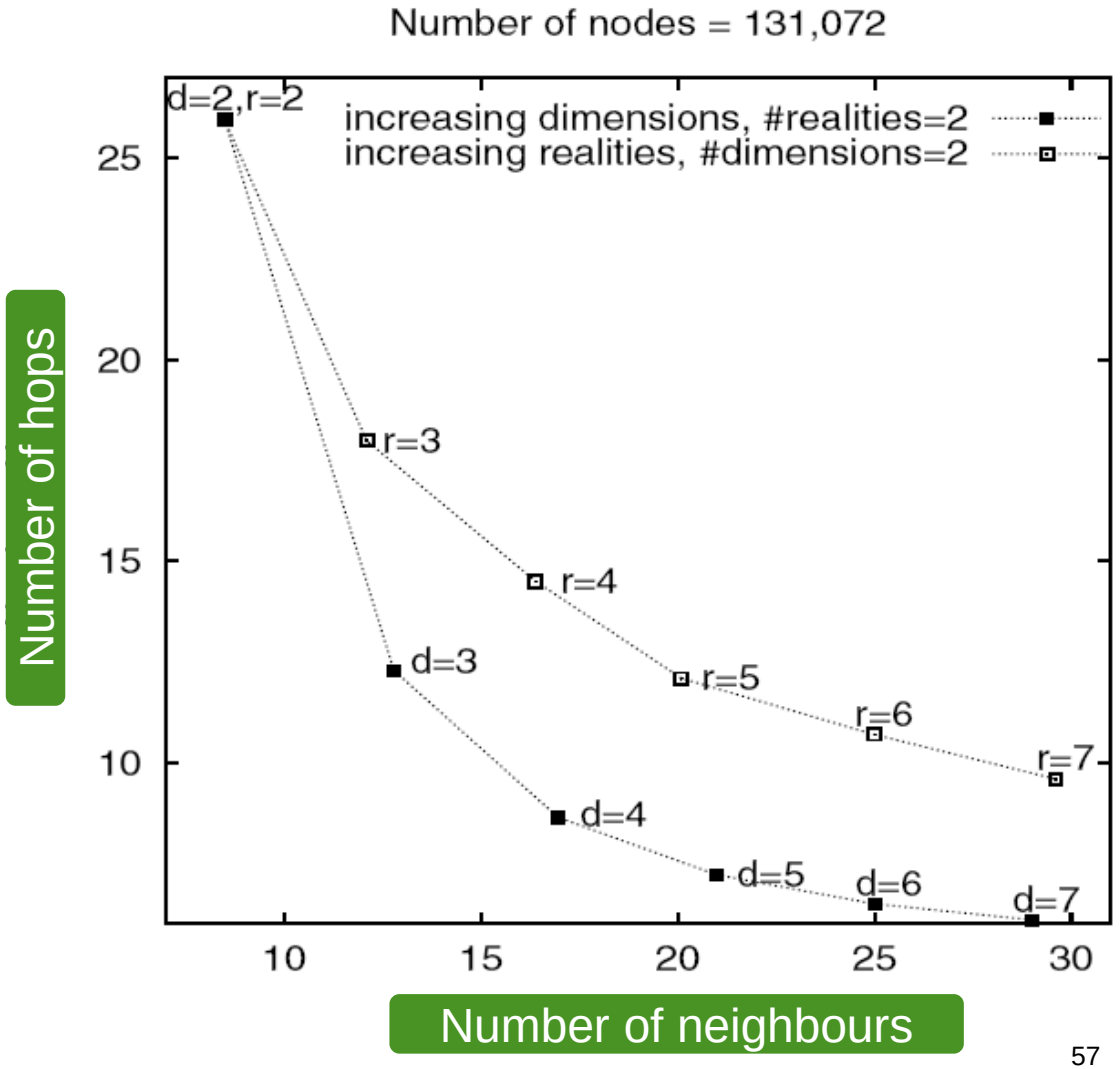


Path Length with Multiple Realities



Observations

- Increased dimensions lead to better performance compared to multiple realities with same local state.
- However: multiple realities increase robustness through replication.



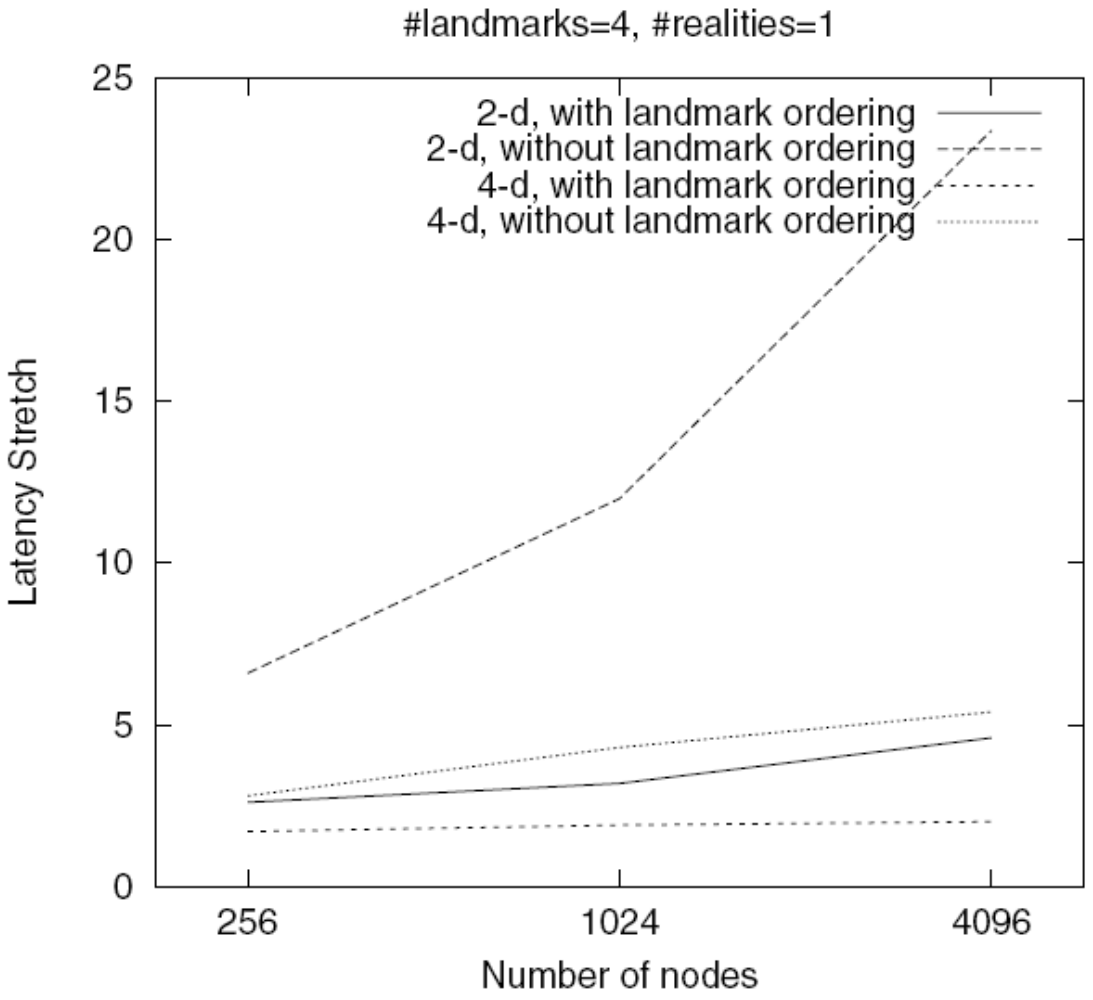
Optimisation 3: Improved Routing Metrics

- Plain CAN uses only geometric distance as routing metric:
  - Possibly not optimal if overlay construction does not consider underlay topology.
  - Path via geometrical closer hop may be slower than path via farther node.
- Improved metric: consider network latency between nodes:
  - Nodes measure round trip time (RTT) to neighbours.
  - Message forwarded to neighbour with minimum ratio RTT/distance.
  - Improved individual hops: complementary to reduced path length using multiple dimensions or realities.
- Improves query latency by up to 40%!
  - More dimensions lead to larger improvements.
    → More alternative choices of next hop!

- Construct overlay by considering underlay topology:
  - Close nodes in underlay should also be close in overlay.
  - Routing metrics only selected best link but do not adapt topology.
- Distributed binning (“Klasseneinteilung”) of CAN nodes:
  - *m* well-known landmark nodes
    → I.e. DNS root servers
  - Nodes individually measure RTT to landmarks.
  - Nodes order landmarks according to RTT.
    → *m*! possible orderings
  - Nodes with same ordering are in the same part of the overlay.
  - Reduces path length!

Optimisation 4: Geographic Network Layout (2)

- Ordering of dimensions: x,y,x,y,x,y,.
- Space is divided into *m* parts along *x* dimension, *m*-1 parts along *y* dimension, *m*-2 parts along *x* dimension, etc.
- Node joins in partition of its landmark ordering.
- Binning may lead to uneven distribution of nodes.
  → Load balancing becomes an issue!



- Introduction to Distributed Hash Tables
- DHT Routing Geometries and Algorithms:
  - Rings: CHORD
  - Rings: Symphony
  - Torus: Content-Addressable Networks (CAN)
  - Plaxton trees
- Summary

When to use a structured Peer-to-Peer system?

- Mostly queries for single index keys or certain key combinations:
  - Arbitrary combinations of keys are not well-supported by plain DHTs.
  - Complex queries require additional mechanisms.
- 100% recall is required.
- Support search for unpopular content that is not replicated at many hosts.
  - DHT will find any content (popular or unpopular) efficiently!

DHT routing geometries

- Chord (Ring), Tapestry (Plaxton tree)
  - O(log(*n*)) links per node
  - O(log(*n*)) path length
- Symphony (Ring)
  - O(*k*) links per node
  - O((1/k) $\log^2$ n) path length
- CAN (Torus)
  - O(2*d*) links per node
  - O($dn^{1/d}$) path length
- Viceroy (Butterfly):
  - O(1) links per node
  - O(log(*n*)) path length