Distributed Algorithms

Peer-to-Peer (P2P)

Agenda

- Introduction
- Categorization of P2P Systems
  - Unstructured P2P Systems
  - Structured P2P Systems
- Case Studies
  - Chord
  - CAN

What is P2P?

- P2P systems consist of nodes with equal rights
- Every node acts as both a client and a server → **servant**
- Usually, a node "pays" for its participation by providing access to (some of) its resources (e.g., disk space)
  → **free riding**, **incentives**
- P2P is not really new (IP, email etc.)

Characteristics of P2P Systems

- No central coordination
- No central database
- No peer has a global view of the system
- Global behavior emerges from local interactions
- Self-organizing evolution of the system (growth etc.)
- All existing data and services are in principle accessible by any peer
- Peers are autonomous
- Peers and connections are unreliable

P2P Systems

- Existing Systems
  - Napster (?)
  - Gnutella
  - Freenet

- Research Prototypes
  - Tapestry, OceanStore
  - P-Grid, Gridella
  - Pastry, PAST
  - ...

- JXTA Framework (project initiated by SUN)

Categorization of P2P Systems

- **Architecture**
  - Centralized (Napster, centralized index)
  - Decentralized (Gnutella, Freenet, etc.)
  - Hierarchical (introduction of super-peers)

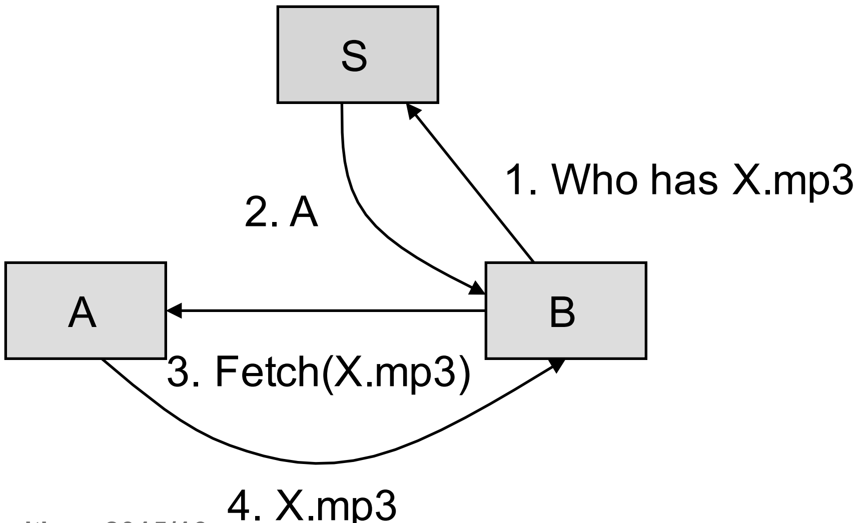
- **Searching for Data** (decentralized architecture)
  - Unstructured (Gnutella, flooding)
  - Structured (Pastry, *n*-ary search trees)

- **Storing Data**
  - Fix (e.g., hashing)
  - Adaptive (Freenet)

Searching in centralized P2P Systems (e.g. Napster)

- (Virtual) central server stores global index which is constructed from clients telling the server which files they share

1. Client contacts server with search query
2. Server answers with lists of clients offering matching data objects (client/server interaction)
3. Client contacts one of the offering clients and downloads data object (P2P interaction)



Disadvantages of centralized Approach

- Central server is
  - single point of failure
  - susceptible for attacks
  - potential bottleneck (CPU, memory, network)

- Operator can be made responsible

- Data management does not depend on P2P interaction, but on client/server interaction

Search in Unstructured P2P Systems

- Search query is flooded into the network
  - Every node propagates query to (all) neighbor nodes
  - Nodes with matching data objects reply to query
  - Replies travel on the same routes back to querying client
  - Lifetime of packets limited by TTL counter
  - Query IDs used to prevent loops

- Similar to a breadth-first-traversal of a graph

Search in Unstructured P2P Systems

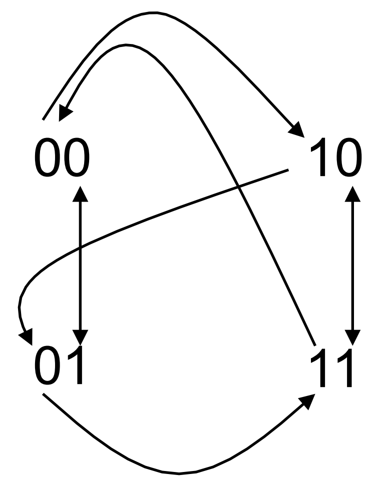
- Advantages
  - Any type of search query can be supported
  - Nodes are autonomous because they can store what they like
  - Addition and removal of nodes is unproblematic
  - Queries are successful with a high probability
  - Fast search („small world property“)
  - High fault-tolerance

- Disadvantage
  - High overhead with respect to network utilization
  - High message complexity of search queries
    → restricted query frequency

- Improved approaches try to decrease message complexity
  (e.g., by “parallel random walks”)

Search in Structured P2P Systems

- Structure is laid on the P2P system
- Query is routed towards matching peers using the structure → directed routing
- Nodes store dedicated data objects
- Example: *x*-ary search trees
  - Pointers to nodes having a distinct digit at some place of the ID
  - *x* digits, *y* length of ID
    $\Rightarrow (x - 1) \cdot y$ pointers stored by a node
    $\Rightarrow n = x^y$ nodes
  - $O(\log(n))$ search steps
- Similar to a depth-first-traversal of a graph



Search in Structured P2P Systems

- Advantages
  - Lower utilization of network
  - Lower message complexity because queries are selectively routed

- Disadvantages
  - Type of search queries is usually limited
    - Often only equality tests are supported because hashing is used to derive the peer id from the respective data element key
  - Autonomy of nodes is restricted as they cannot decide on their own which data items they store
  - Additions and removals of nodes are expensive
  - Less robust against crashed nodes or links

CASE STUDY: CHORD

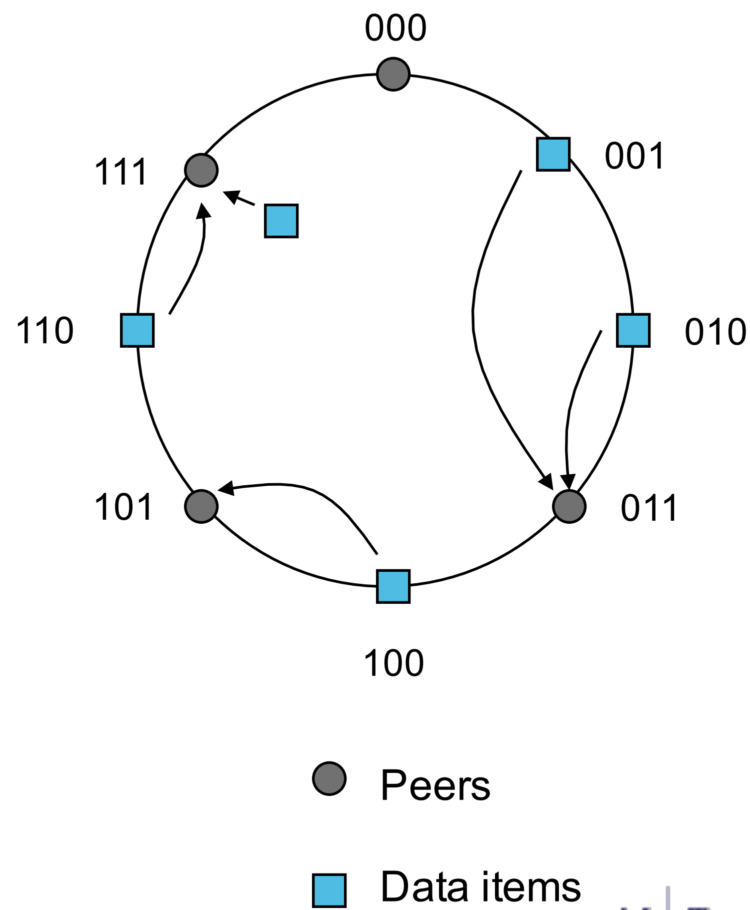
Chord

- Search keys and addresses of peers are hashed on binary keys of length *m*
- Peers and data items are arranged in a circle with at most $n = 2^m$ peers
- A peer with hashed address *p* is responsible for storing all data items with hashed key

  $$\boldsymbol{k \in (\, predecessor(p), p\,]}$$
- Thus, a data item with
  hashed key *k* is stored by the peer with hashed address

  $$\boldsymbol{p = successor(k)}$$

  (*p* is the next peer clockwise)

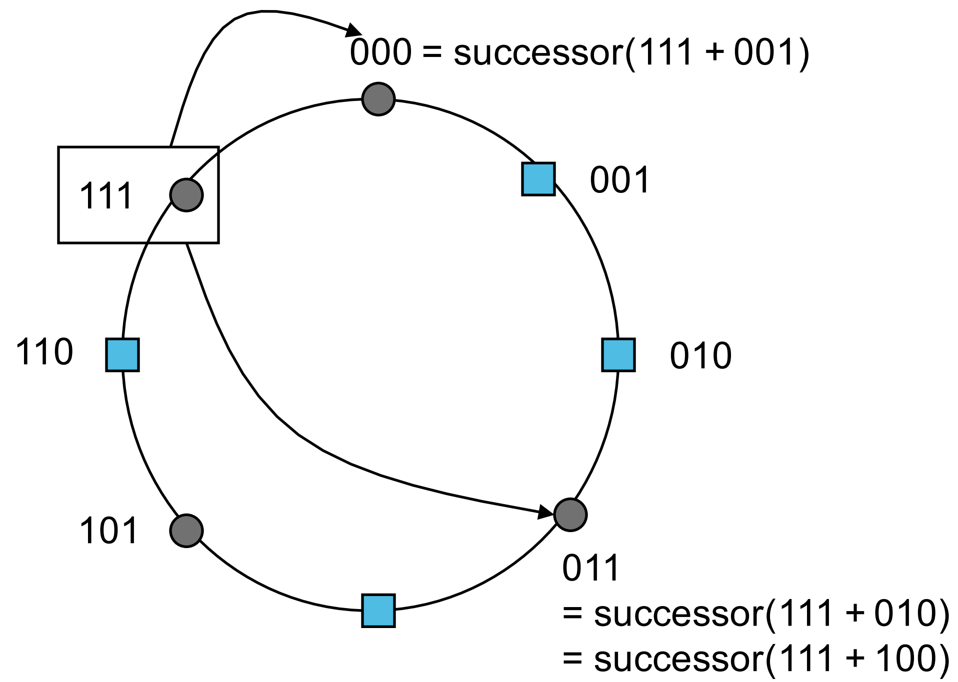


Routing Tables in Chord

- Pointer to successor node would be enough to lookup any node
- But for efficiency, each peer p stores a finger table consisting of the first peer with hashed identifier p_i such that

$$\mathbf{p_i = \textit{finger}(\textit{i}, \textit{p}) = \textit{successor}(\textit{p} + 2^{\textit{i}-1}) \text{ for } \textit{i} = 1,..,\textit{m}}$$

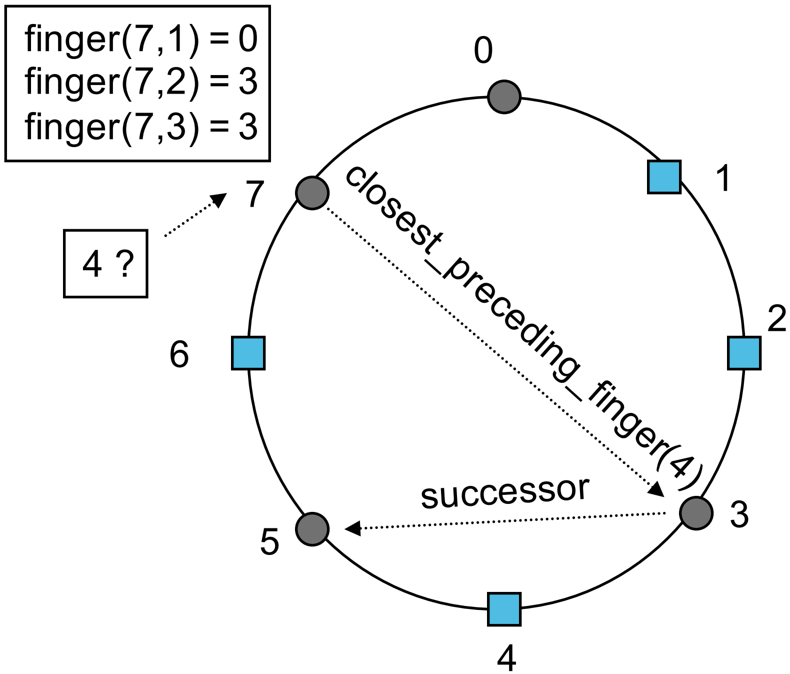
- All arithmetic is done modulo 2^m
- $O(log\ n)$ pointers per node



Searching in Chord

- Search can be initiated at any peer
- Basic idea: Find predecessor of *k*, then go its successor

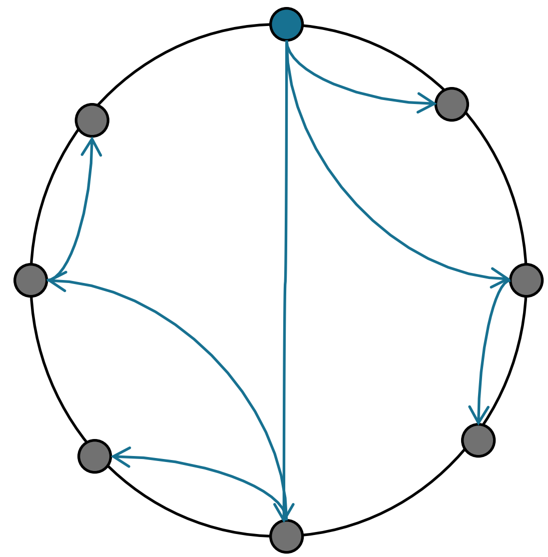
- To find the predecessor of *k*, a peer *p* that receives the query, forwards it to the closest preceding finger of *k* if $k \notin (p, p.successor]$
- This is repeated until the predecessor of *k* is reached
- Then, the query is forwarded to the successor of this node



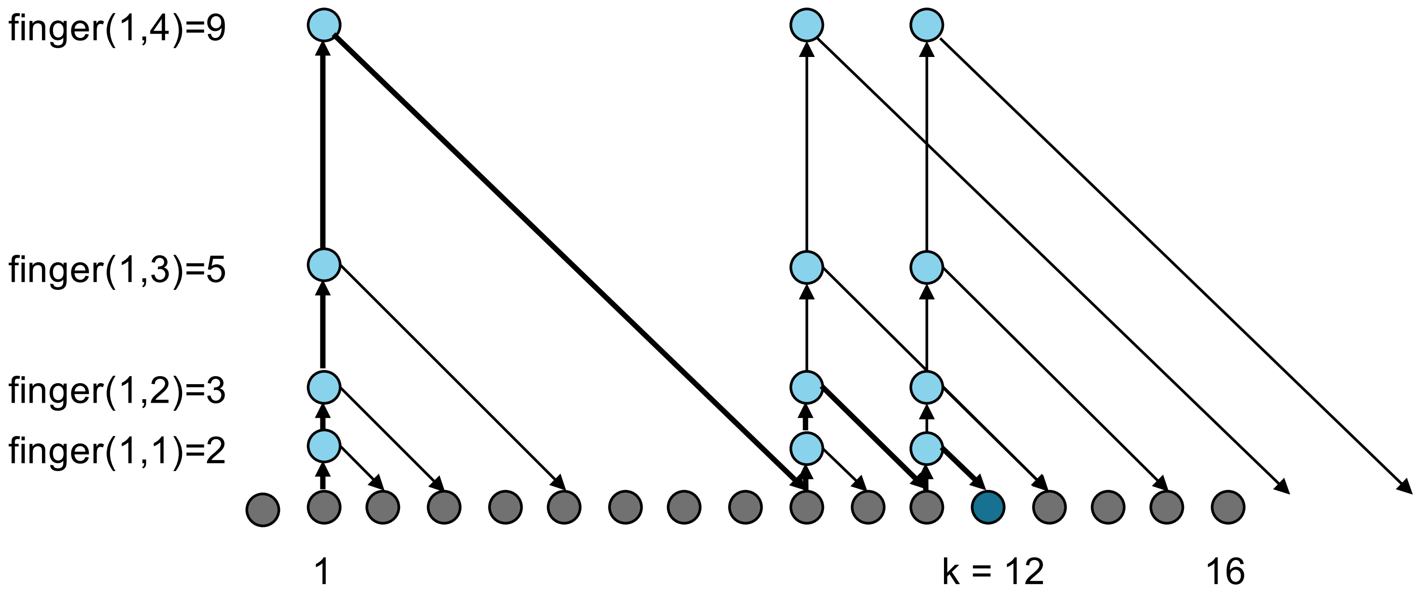
Searching in Chord

- Search complexity is *O*(*log n*) because remaining distance is at least halved with every forwarding step

- Example:
  In a ring with m =3, each node can be reached from any other node in at most three hops



A Tree-Perspective on CHORD



Joining and Leaving the CHORD Network

- Invariants that must be maintained
  - Each node's successor is correct. For efficiency reasons, the nodes' finger tables should also be correct.
  - For every key k, *successor*(k) is responsible for data item k

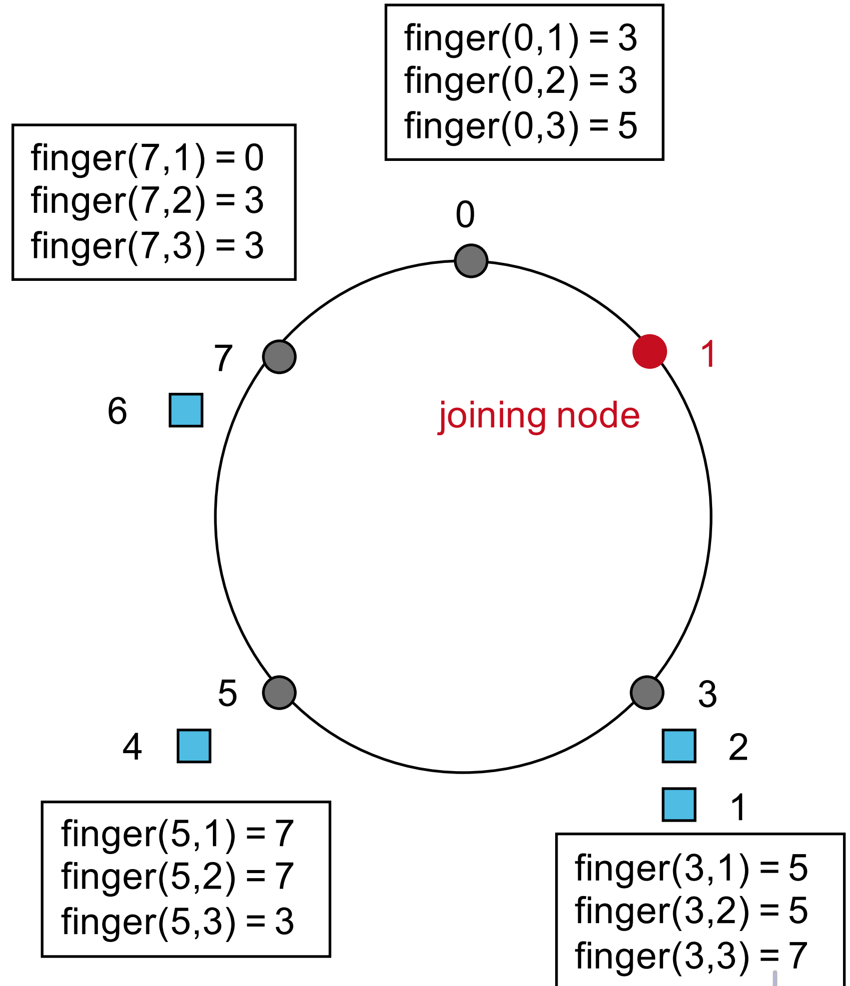
- To maintain these invariants, data items must be moved and finger tables must be updated when nodes join or leave
  - Data items are only transferred between neighboring nodes
  - Only an $O(1 / n)$ fraction of the data items is moved, which is also the minimum required to maintain a balanced load
  - Join or leave requires $O(log^2 n)$ messages with high probability to update finger table

- Simultaneous joins and leaves are possible

Node Join

- Finger table of joining node must be initialized
- Some of the nodes whose finger table points to the successor of the new node must update their finger tables to point to the new node instead

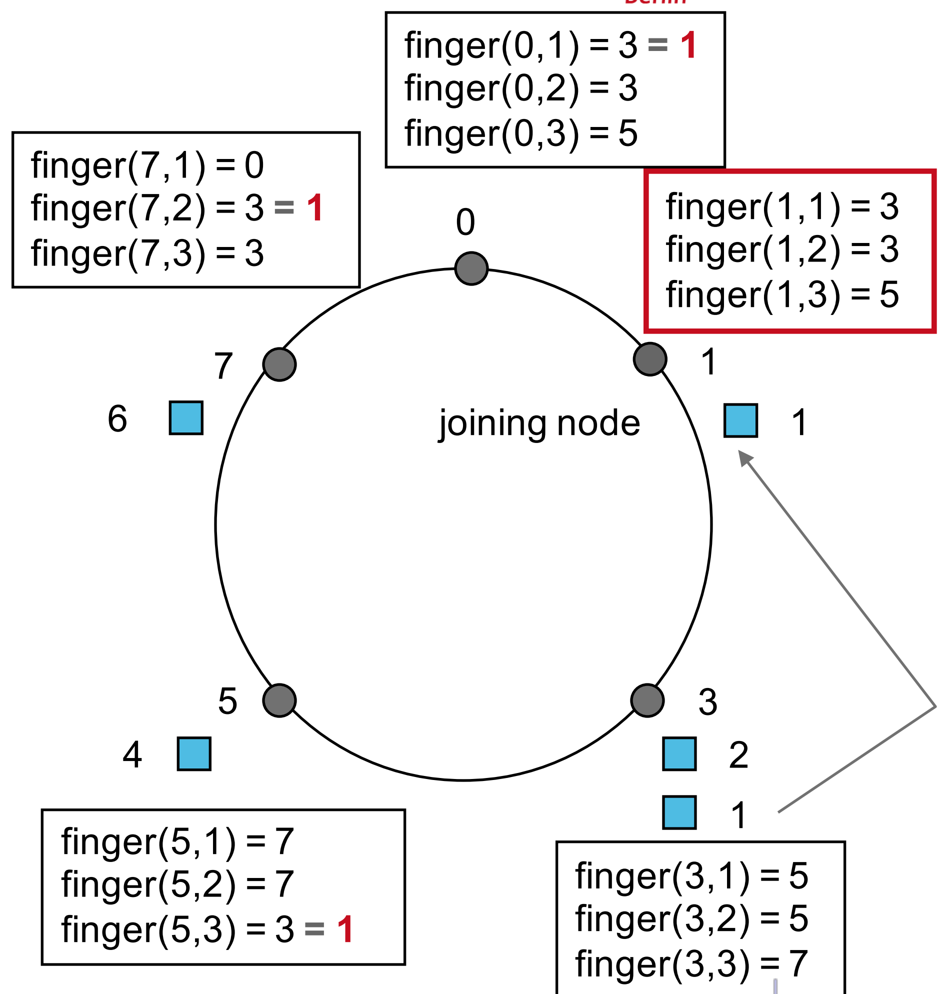
- Data items assigned to joining node must be moved from successor node to joining node



Node Join

- Finger table of joining node must be initialized
- Some of the nodes whose finger table points to the successor of the new node must update their finger tables to point to the new node instead

- Data items assigned to joining node must be moved from successor node to joining node



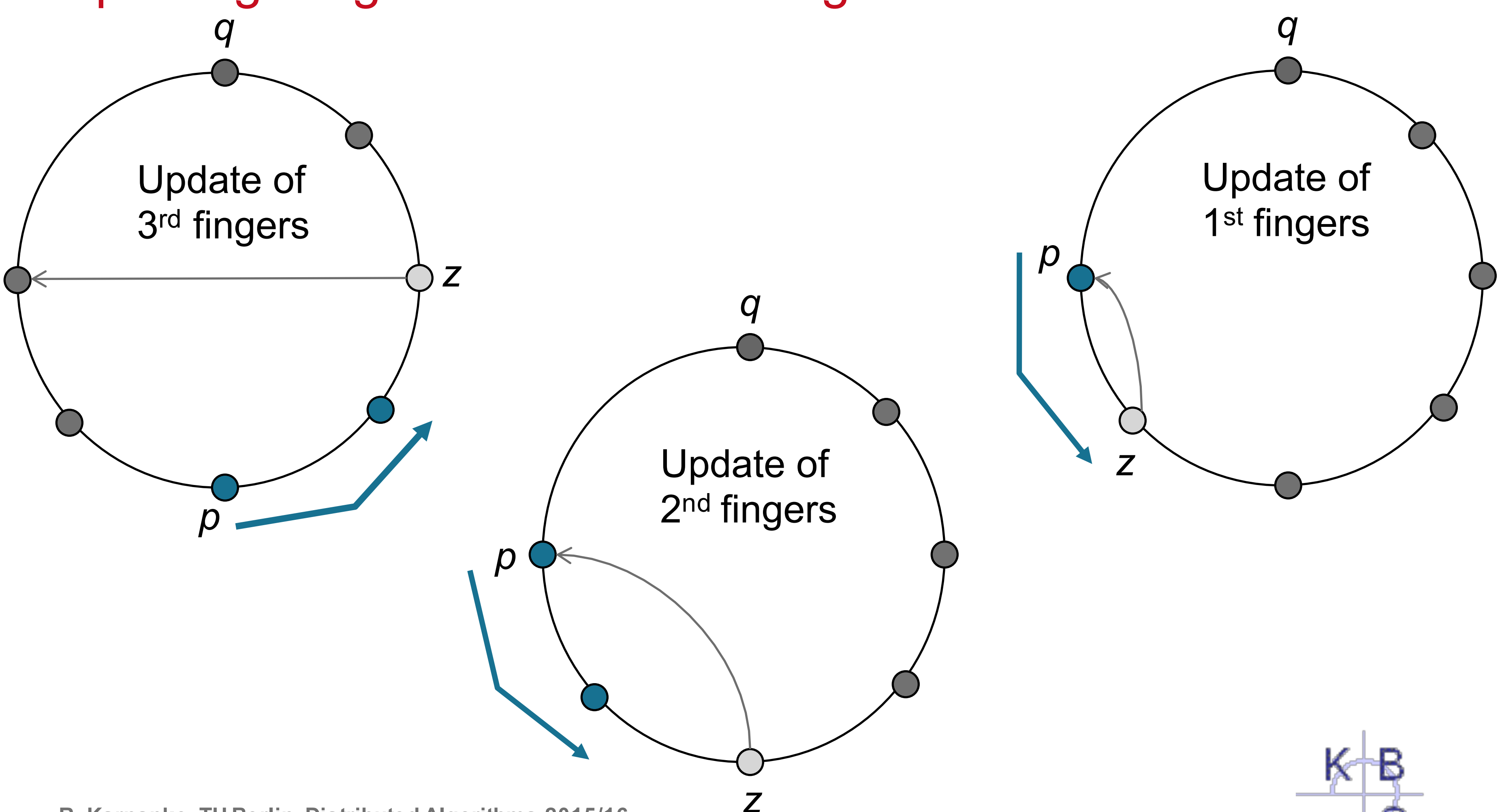
Updating Finger Tables of Existing Nodes

- A new node *q* will become the *i*-th finger of node *p* iff
  - Node *p* *precedes* *q* by at least 2^{i-1}
  - The *i*-th finger of node *p* *succeeds* *q*
- The first node *p* that can meet these two conditions is the immediate predecessor of $q - 2^{i-1}$

- Thus, for a given *q*, the algorithm starts at $q - 2^{i-1}$, and then walks counter-clock-wise on the circle, until it encounters a node *z* whose *i*-th finger *precedes* *q*

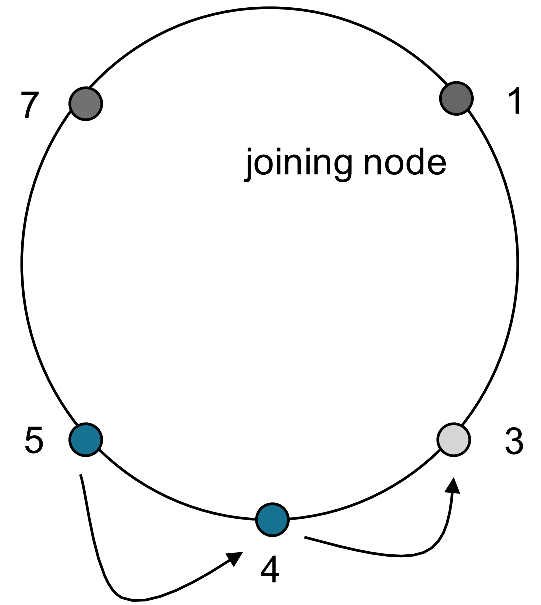
- To simplify the join and leave mechanisms, each node in Chord maintains also a predecessor pointer
- The number of nodes that need to be updated when a node joins the network is O(log *n*) with high probability

Updating Finger Tables of Existing Nodes



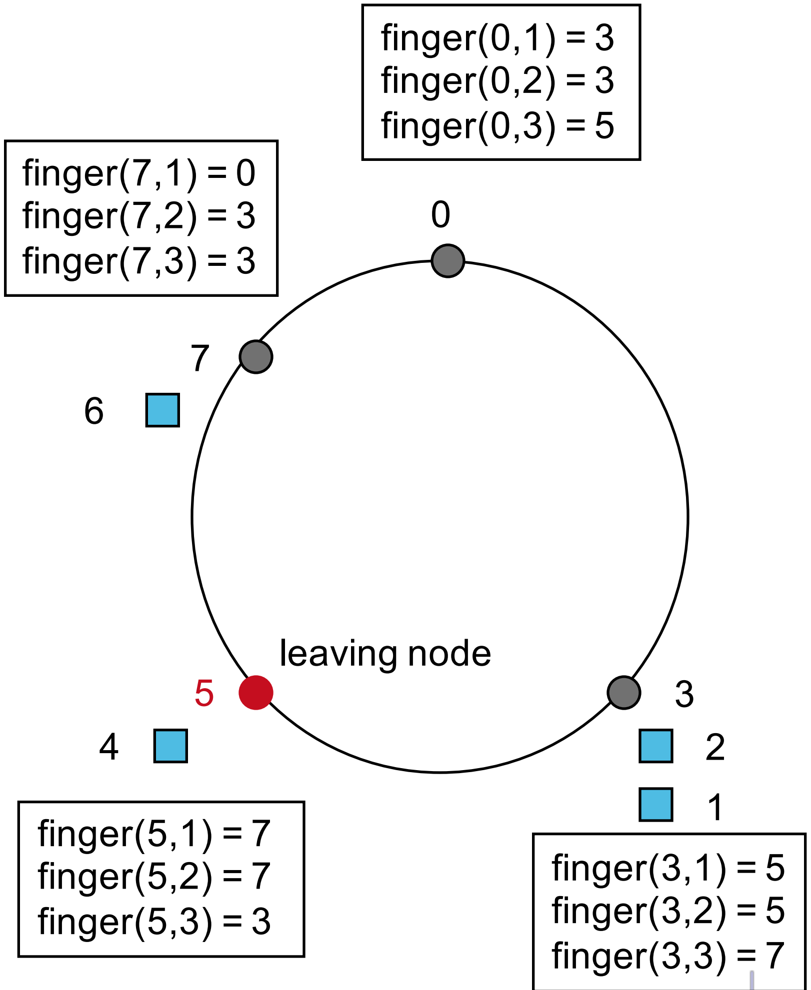
Updating Finger Tables of Existing Nodes

- Example: Update of those 3rd fingers pointing to node 3, but which must now point to the new node 1
- Nodes affected must have an id lower or equal to (1 – 4) % 8 = 5
- Update algorithm starts at node 5 and walks counterclockwise until it reaches node 3 whose 3rd finger points to node 7
- Thus, the 3rd fingers of nodes 5 and 4 are updated to point to the new node 1



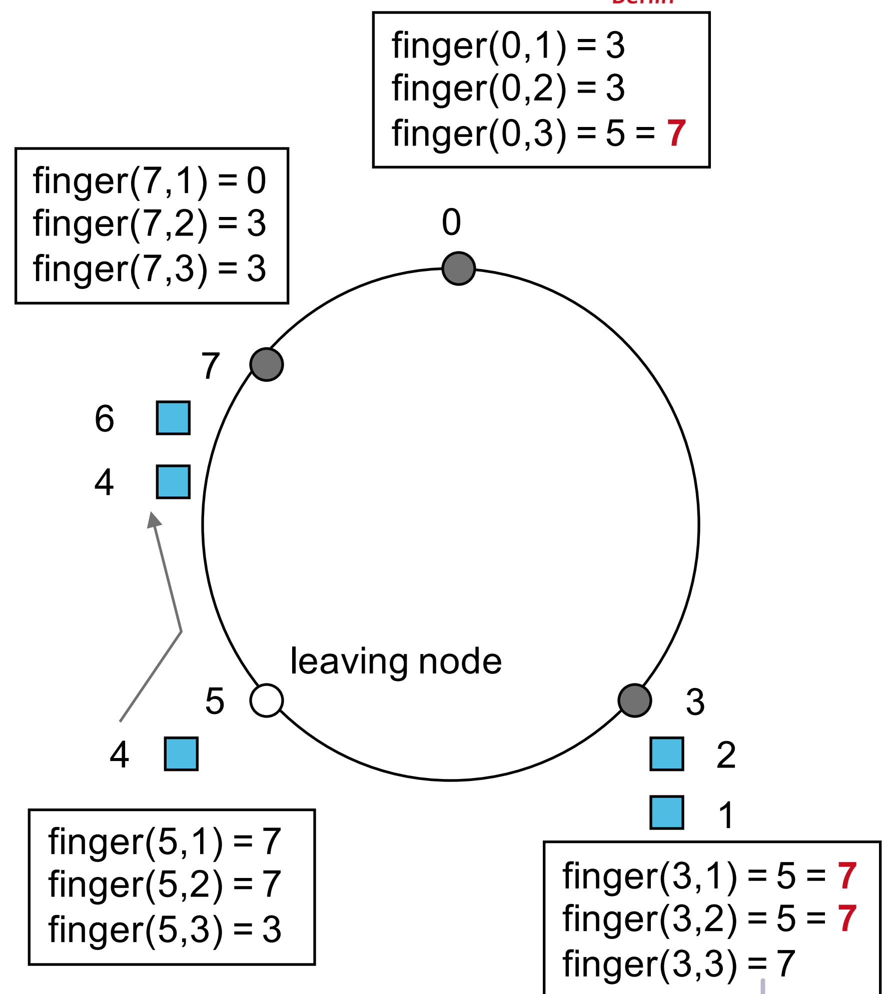
Node Leave

- Data items assigned to leaving node are moved to successor node
- Those nodes whose
  finger table points to the leaving note must update their finger table to point to the successor of the leaving node instead



Node Leave

- Data items assigned to leaving node are moved to successor node
- Those nodes whose finger table points to the leaving note must update their finger table to point to the successor of the leaving node instead



Fault Tolerance in CHORD

- A data item is stored at the *r* nodes succeeding its key
- Each node keeps a list of its *r* immediate successor nodes for direct lookup
- Queries referring to a failed node are redirected to the next living successor

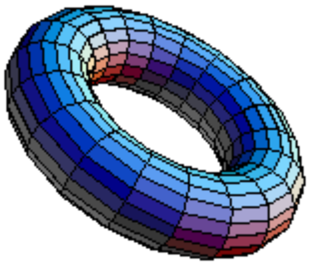
- If the closest preceding finger has failed (detected by a timeout), a query is routed to the preceding entry in the finger list or a node in the successor list

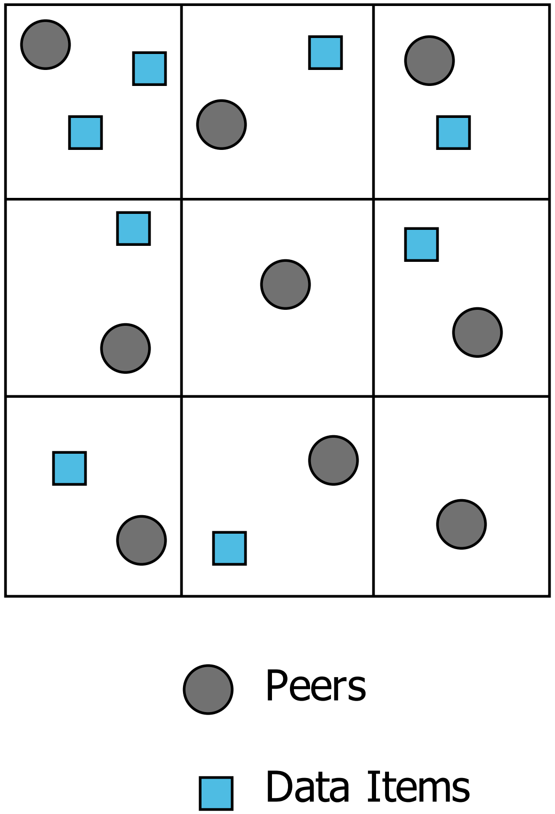
- A randomized stabilization procedure periodically corrects finger tables and successor lists to compensate for failures

CASE STUDY: CAN

Content-Addressable Networks (CAN)

- Keys are hashed into coordinates on a *d*-dimensional torus
- Each peer is responsible for an exclusive zone (subvolume of the space) and stores those data items whose keys are hashed into this zone

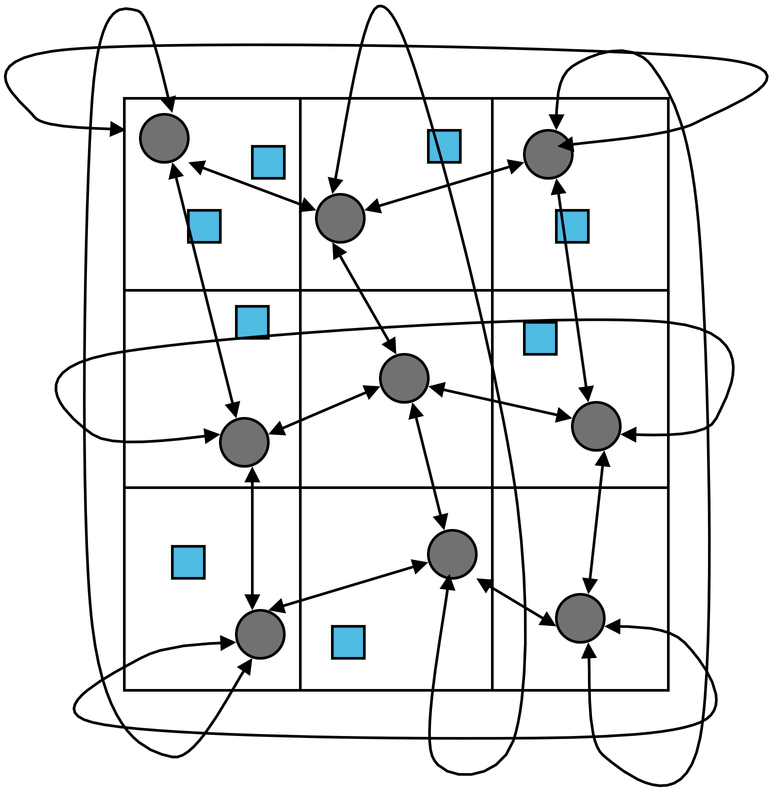




http://mathworld.wolfram.com/Torus.html

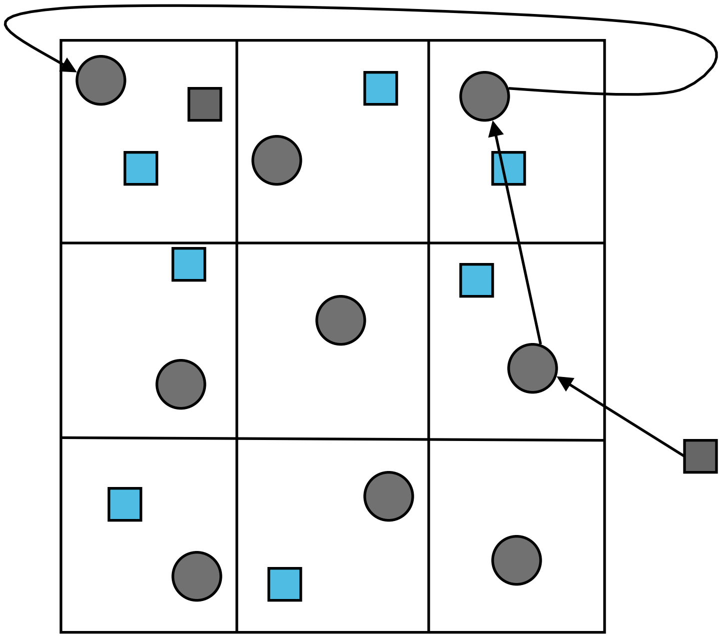
Routing Tables in CAN

- For routing, each peer stores the peers responsible for the neighboring zones
- Two zones are neighbored if they have a common border in $d - 1$ dimensions ($d = 2 \rightarrow$ line)
- Since each node has $2 \cdot d$ neighbors, its routing table contains $2 \cdot d$ entries
- Hence, routing table size is independent of the overall number of peers → scalability



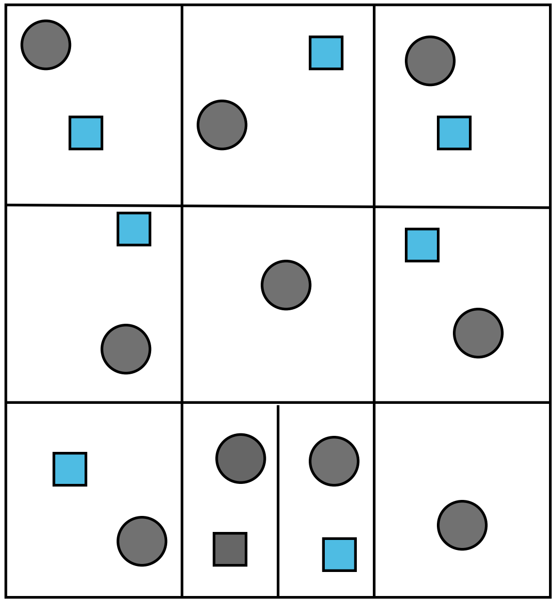
Searching in CAN

- Can be initiated at any peer
- If a peer receiving a query does not store the requested item, it forwards the query to one of its neighbors having the closest distance to the requested item (there can be at most *d* of them)
- If a node to that a query should be forwarded has failed, the query is forwarded to the next best peer
- Search complexity $O(d \cdot n^{1/d})$



Node Joins: Finding a Zone

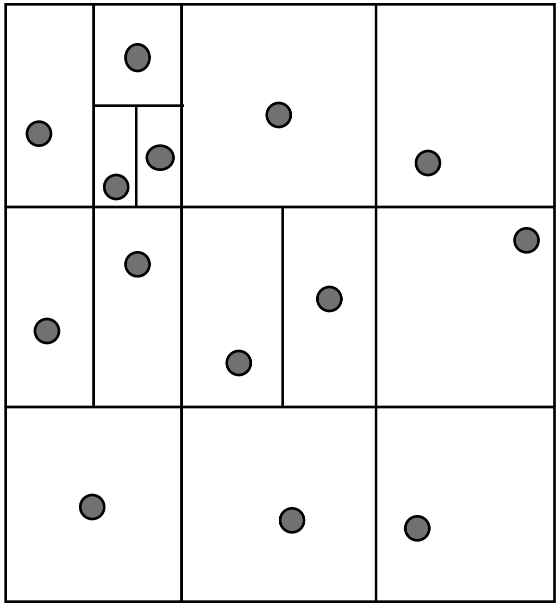
- The joining node randomly chooses a point *P* in the space and sends a *JOIN* request for point *P* via an arbitrary node already in the CAN
- When the node in whose zone *P* lies receives the JOIN request, it splits its zone in halves and assigns one half to the new node
- The (key, value) pairs from the half zone to be handed over are transferred to the new node



Node Joins: Zone Splitting

- Splitting is done alternatingly by assuming a certain ordering of the dimensions in deciding along which dimension a zone is to be split next.

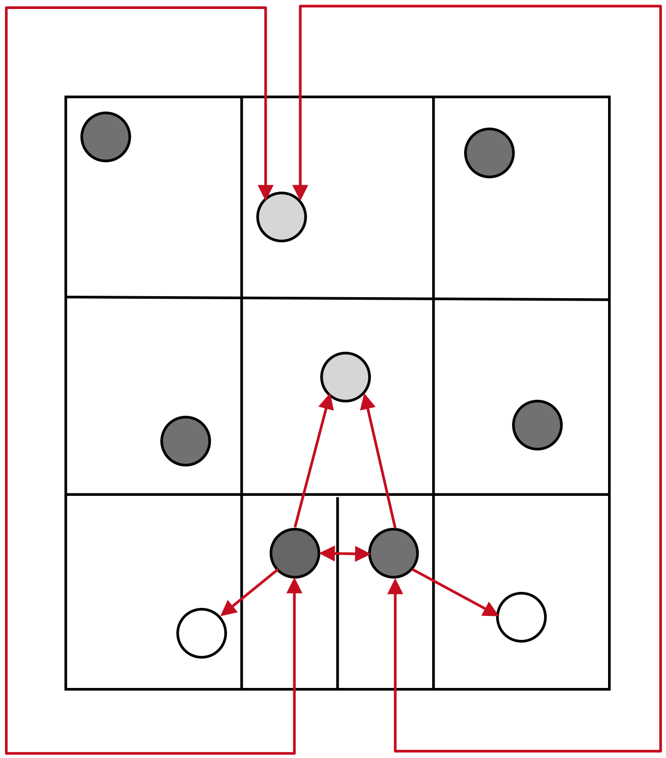
- For a two dimensional space, a zone is first split along the
  X dimension, then along the
  Y dimension, then along the
  X dimension, and so on.



Node Joins: Joining the Routing

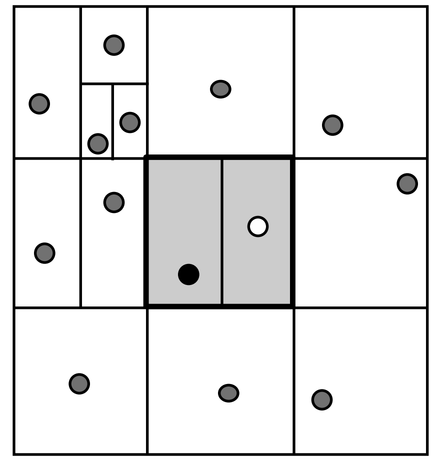
- The new node learns its neighbor set from the previous occupant of the zone
- This set is a subset of the previous occupant's neighbors, plus that occupant itself
- Similarly, the previous occupant updates its neighbor set to eliminate those nodes that are no longer neighbors
- Finally, both the new and the old nodes' neighbors must be informed of this reallocation of the space

  → $O(d)$ nodes affected

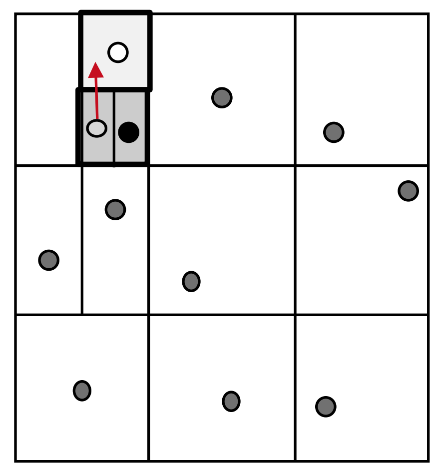


Node Leaves

- When a node leaves, its zone is merged with the zone of its sibling neighbor



- If this is not possible because the sibling zone is split, two sibling leafs of this zone are merged and one of the two respective nodes takes over the vacant zone



- In both cases, the data items are transferred to the node now responsible for them

Possible Optimization Goals

- Reduce latency
  - Reduce average number of hops per query to reach destination
  - Reduce average latency per hop in the overlay network

- Increase fault tolerance to ensure availability of data items and routing paths
  - Store data items at multiple nodes

Latency Optimization by Increasing Dimensions

- Advantages
  - Reduces the average path length (was $O(d \cdot n^{1/d})$)
  - More neighbors per node leads to higher routing fault tolerance

- Disadvantages
  - More state (bigger routing tables)

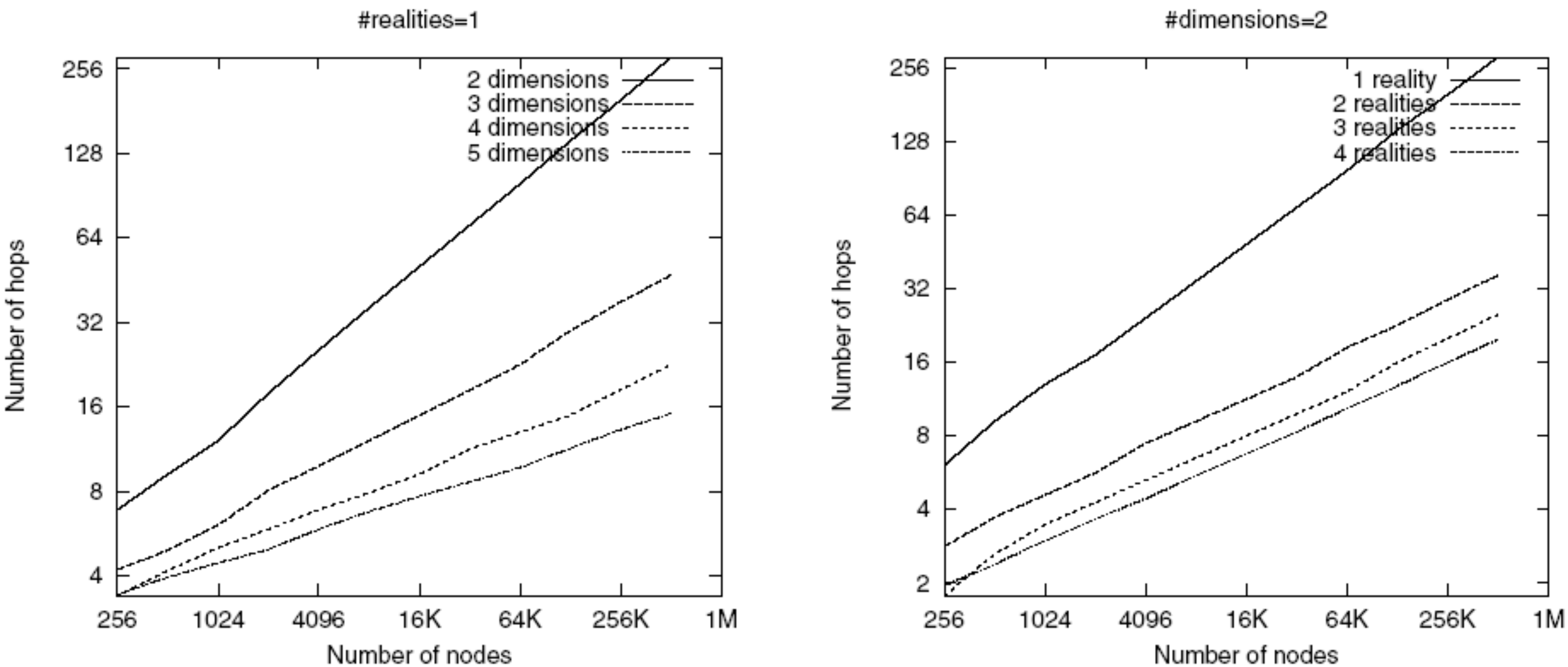
Latency Optimization by using Multiple Realities

- We can have *r* different coordinate spaces
- Nodes hold a zone in each of them
- Creates *r* replicas of the (key, value) pairs

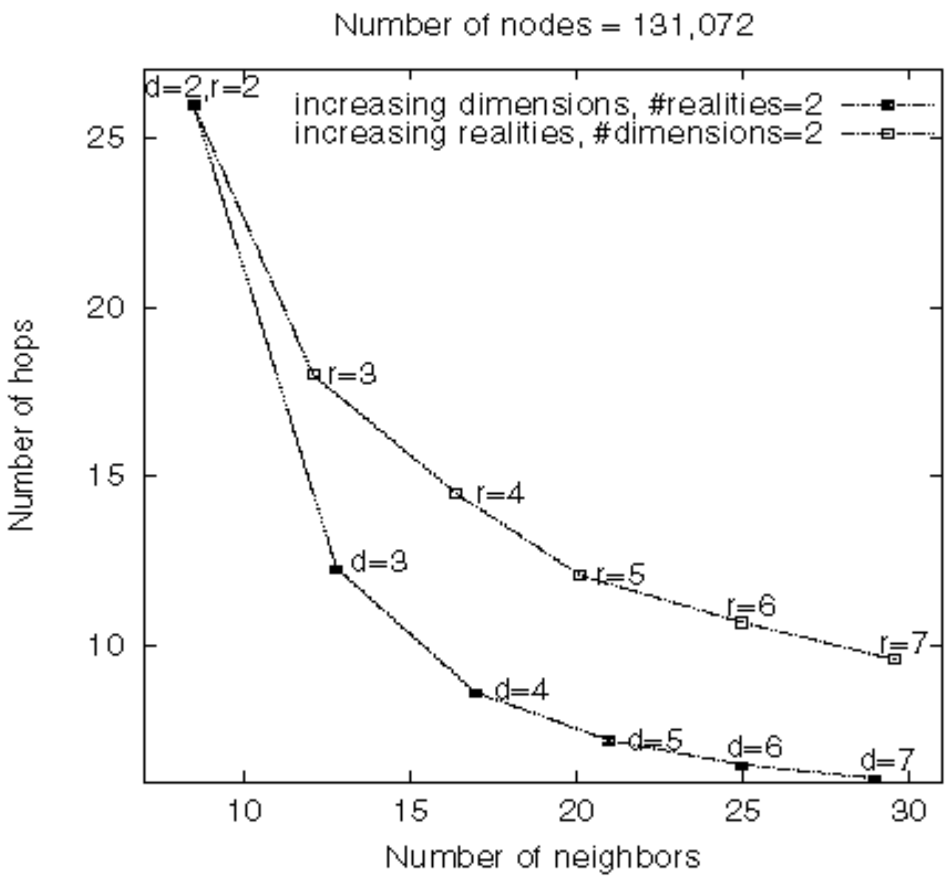
- Advantages
  - Increases robustness because search can be continued in another reality if the search fails in one reality
  - Higher fault tolerance for (key, value) pairs as they are “mirrored” in multiple realities
  - Reduces path length because search can be continued in that reality where the target is closest

- Disadvantages
  - Higher state (maintaining *r* times the state of one reality)

Multiple Realities vs. Increased Dimensions



Multiple Realities vs. Increased Dimensions



Latency Optimization by Adaptive Routing

- Choose next hop according to the maximum ratio of progress and round trip time
- Does not reduce the path length but the latency of individual hops
- Lowers the per-hop latency in experiments between
  24% and 40% (depending on the number of dimensions)

Overloading Zones

- Here, several peers are responsible for the same zone
- Splits are performed only if a maximum occupancy is reached
- Data items are replicated across or assigned exclusively to nodes
- Nodes know all other nodes in the same zone but only one of the nodes in each neighbored zone (RTT is measured infrequently to optimize latency for this hop)

- Advantages
  - Increases fault tolerance if replication is used

Current Research Topics in the Area of P2P

- Freeriding
- Efficient Search
- Complex search queries in structured P2P Systems
- Security, Anonymity, Trust
- Adaptive, self-organizing routing algorithms
- Transactions in P2P systems
- Quality of Service (e.g., congestion control)
- ...

Bibliography

1. **K. Aberer and M. Hauswirth. *Peer-to-Peer Information Systems: Concepts and Models, State-of-the-Art, and Future Systems*. Tutorial given at 18th International Conference on Data Engineering, 2002.**

2. I. Stoica, R. Morris, D. Karger, F. Kaashoek, H. Balakrishnan. *Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications*. In Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM), San Diego, California, United States, 2001. ACM Press.

3. S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. *A Scalable Content-Addressable Network*. In Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM), pages 161--172, San Diego, California, United States, 2001. ACM Press.