5. Structured Peer-to-Peer Networks

5.1 Distributed Hash Tables

5.2 Chord

5.3 CAN

5.1 Distributed Hash Tables

Essential challenge in (most) peer-to-peer systems:

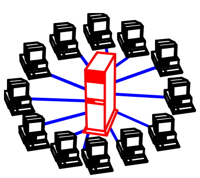
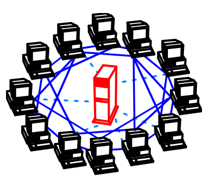
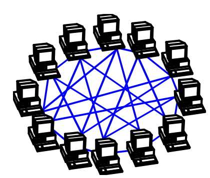
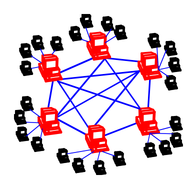
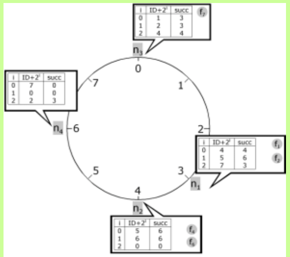
- Location of a data item among the distributed systems:
  - Where shall the item be stored?
  - How does a requester find the location of an item?
- Allow peer nodes to join and leave the system anytime.
- Scalability: keep the complexity for communication and storage scalable.
- Robustness and resilience in case of faults and frequent changes

Distributed Hash Tables serve two purposes:

- to distribute data evenly over a set of peers
- to locate the data in search processes.

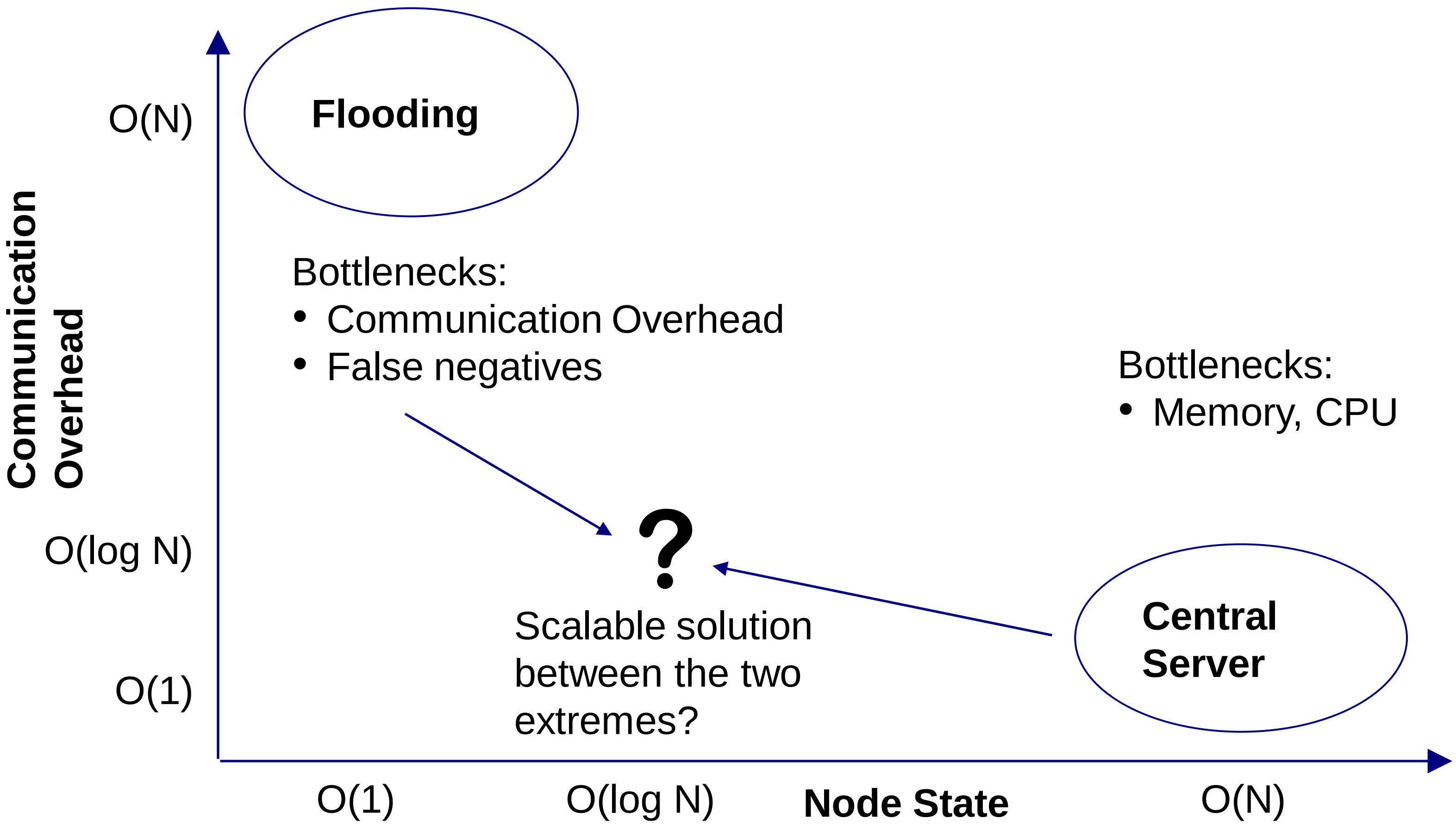
They use a *hash function* for both purposes.

Peer-to-Peer Systems Based on DHTs

<table>
<tr><th rowspan="3">Client-Server</th><th colspan="4">Peer-to-Peer</th></tr>
<tr><td colspan="4">1. Resources are shared between the peers
2. Resources can be accessed directly from other peers
3. Peer is provider (server) and requestor (client): servent concept</td></tr>
<tr><th colspan="3">Unstructured P2P</th><th>Structured P2P</th></tr>
<tr><td rowspan="2">1. Server is the central entity and only provider of service and content.
→ Network managed by the Server
2. Server as the higher performance system.
3. Clients as the lower performance system
Example: WWW</td><th>Centralized P2P</th><th>Pure P2P</th><th>Hybrid P2P</th><th>DHT-Based</th></tr>
<tr><td>1. All features of peer-to-peer included
2. Central entity is necessary to provide the service
3. Central entity is some kind of index/ group database
Example: Napster</td><td>1. All features of peer-to-peer included
2. Any peer entity can be removed without loss of functionality
3. No central entities
Examples: Gnutella 0.4, Freenet</td><td>1. All features of Peer-to-Peer included
2. Any terminal entity can be removed without loss of functionality
3. → dynamic central entities
Example: Gnutella 0.6, JXTA</td><td>1. All features of Peer-to-Peer included
2. Any terminal entity can be removed without loss of functionality
3. → No central entities
4. Connections in the overlay are “fixed”
Examples: Chord, CAN</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td colspan="2">1st Gen.</td><td>2nd Gen.</td><td></td></tr>
</table>

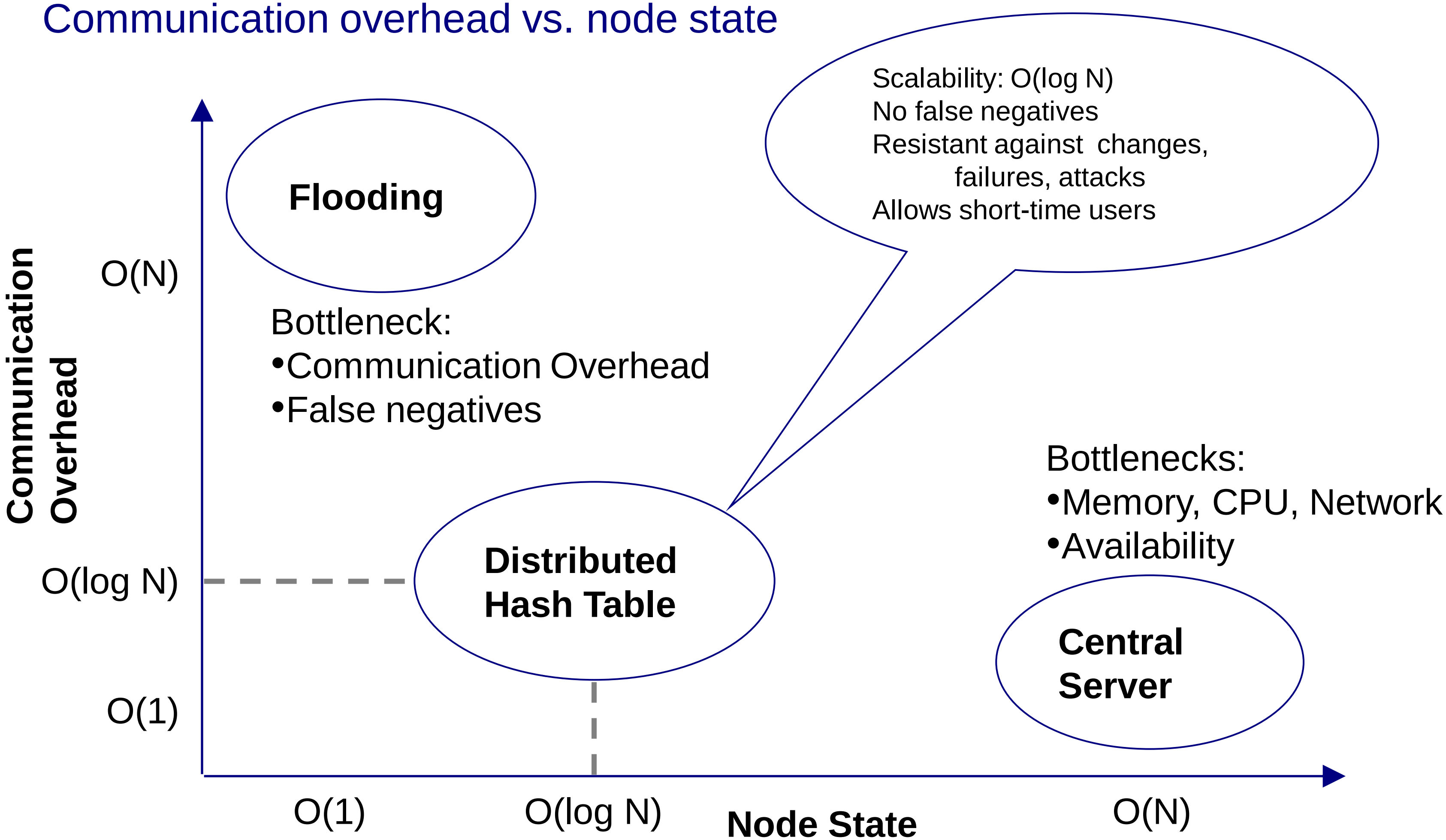
Distributed Indexing (1)

Communication overhead vs. node state



Distributed Indexing (2)

Communication overhead vs. node state



Distributed Indexing (3)

Approach of distributed indexing schemes

- *Data and nodes are mapped into the same address space!*
- Intermediate nodes maintain routing information to target nodes
  - Efficient forwarding to destination node (based on content – not on location)
  - Definitive statement about the existence of content is possible.

Problems

- Maintenance of routing information required
- Fuzzy queries not easily supported (e.g., wildcard searches)

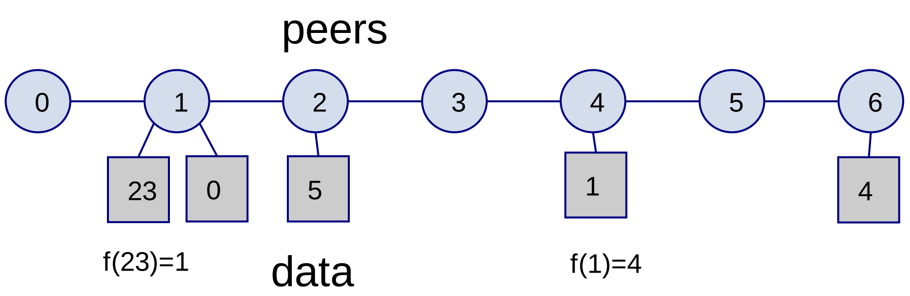
Comparison of Lookup Concepts

System	Per-Node State	Communication Overhead	Fuzzy Queries	No false negatives	Robust-ness
Central Server	O(N)	O(1)	✓	✓	✗
Flooding Search	O(1)	O(N^2)	✓	✗	✓
Distributed Hash Tables	O(log N)	O(log N)	✗	✓	✓

From Classic Hash Tables to Distributed Hash Tables

Classic Hash Table

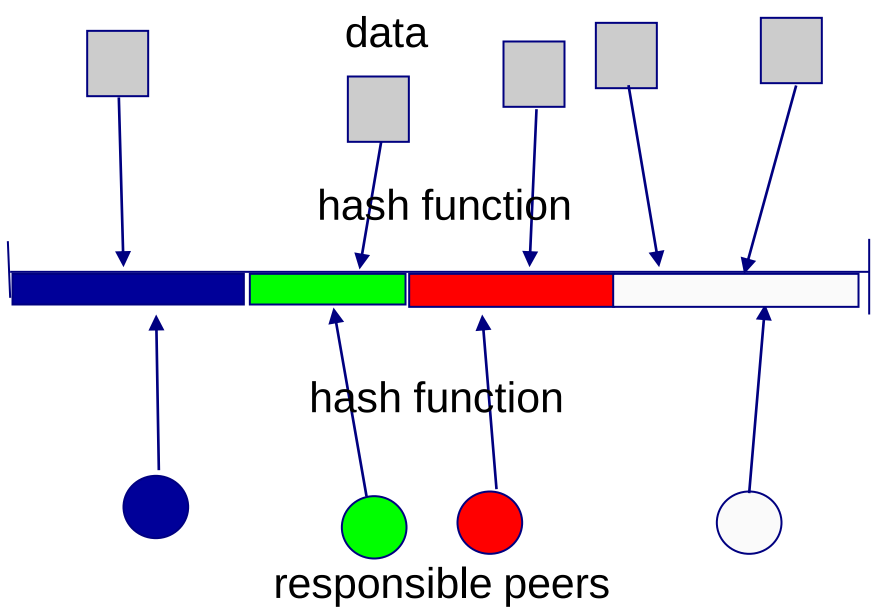
Searching is easy and efficient. However, adding a new peer node changes the hash function!



Distributed Hash Table

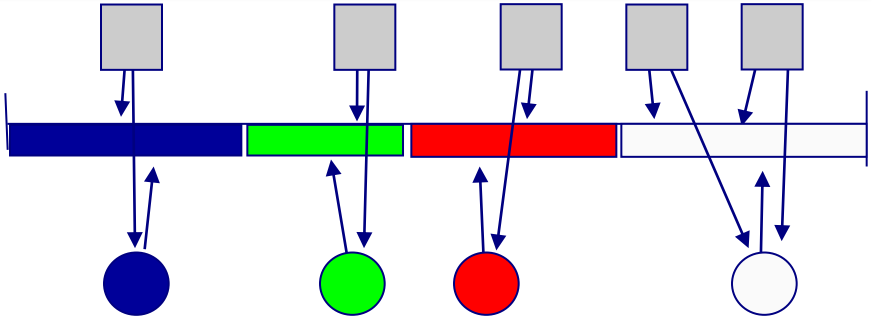
Each peer is responsible for a subset of the data range; that subset is computed by the hash function.

If we search for data, the query is submitted to *the same hash function*.

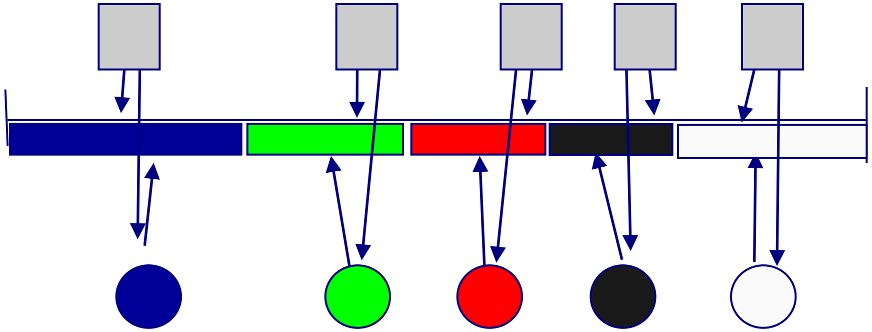


Insertion into a Distributed Hash Table

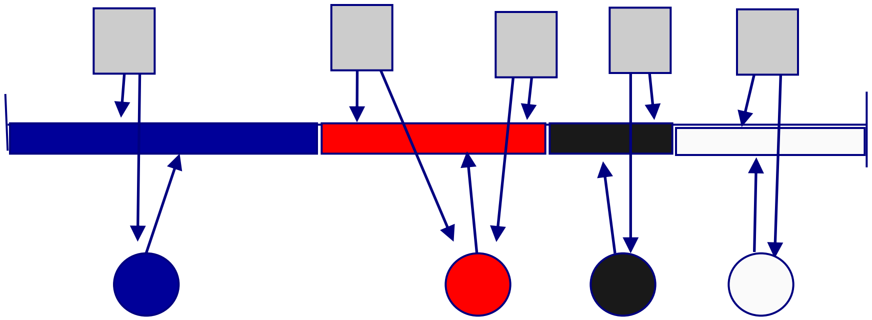
- Peers are hashed to a specific area
- Documents are also hashed to a specific area
- Each peer is responsible for his area



- When a new node is added to the network the neigbors share their range with it.



- When a node leaves the network
  his neighbors take over his share.



Distributed Management of Data

Sequence of operations

1. Mapping of nodes and data into the same address space

- Peers and content are addressed using flat identifiers (IDs)
- Common address space for data and nodes
- Nodes are responsible for data in certain parts of the address space
- Association of data to nodes can change since nodes may disappear

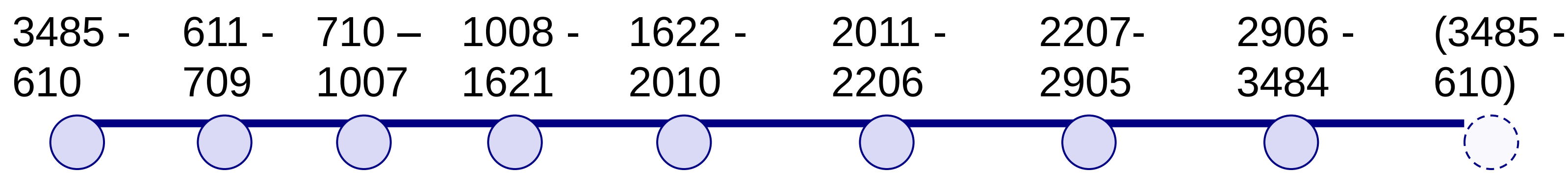
2. Storing / looking up data in the DHT

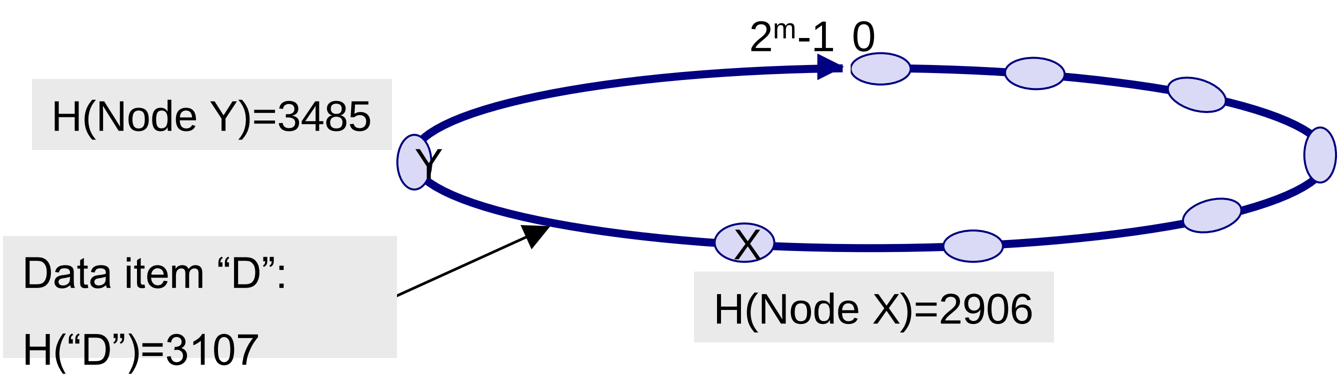
- Search for data = routing to the responsible node
  - Responsible node not necessarily known in advance
  - Deterministic statement about the availability of data possible

Addressing in Distributed Hash Tables

Step 1: Mapping of content/nodes into the same address space

- Usually: 0, …, 2^m-1 >> number of objects to be stored
- Mapping of data and nodes into an address space (with a hash function)
  - e.g., Hash(*String*) mod 2^m: H(„*my data*“) → 2313
- Association of parts of the address space with peer nodes



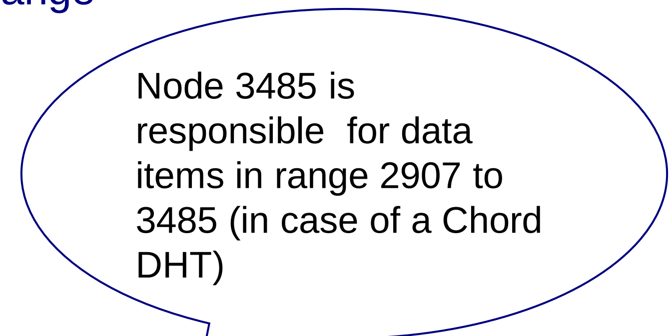


Often, the address space is visualized as a circle.

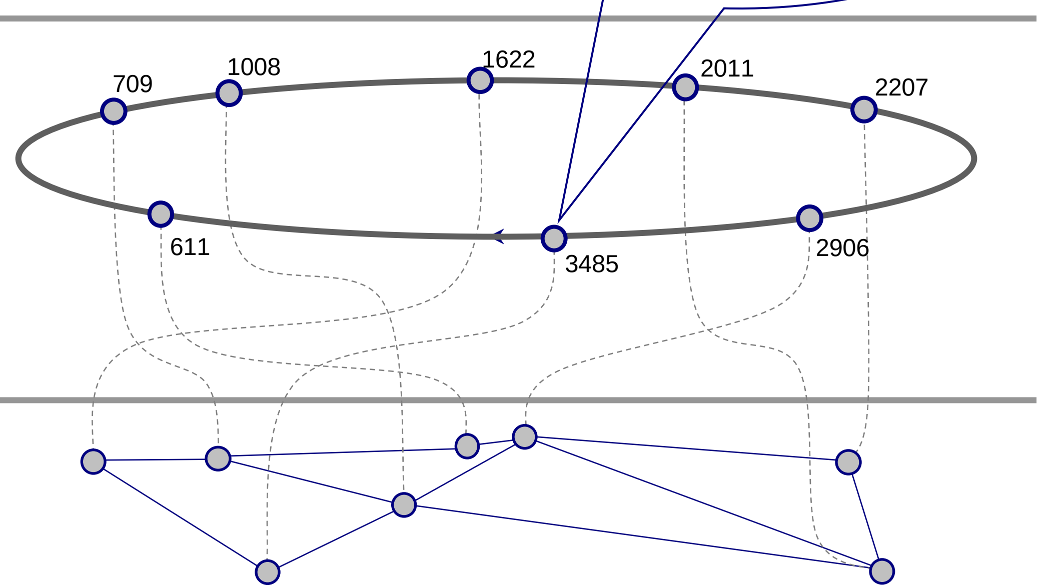
Association of Address Space with Nodes

Each node is responsible for a part of the value range

- Sometimes with redundancy
- Continuous adaptation
- Real (underlay) and logical (overlay) topology are uncorrelated



Logical view of the Distributed Hash Table



Mapping to the real Internet topology

Step 2: Routing to a Data Item (1)

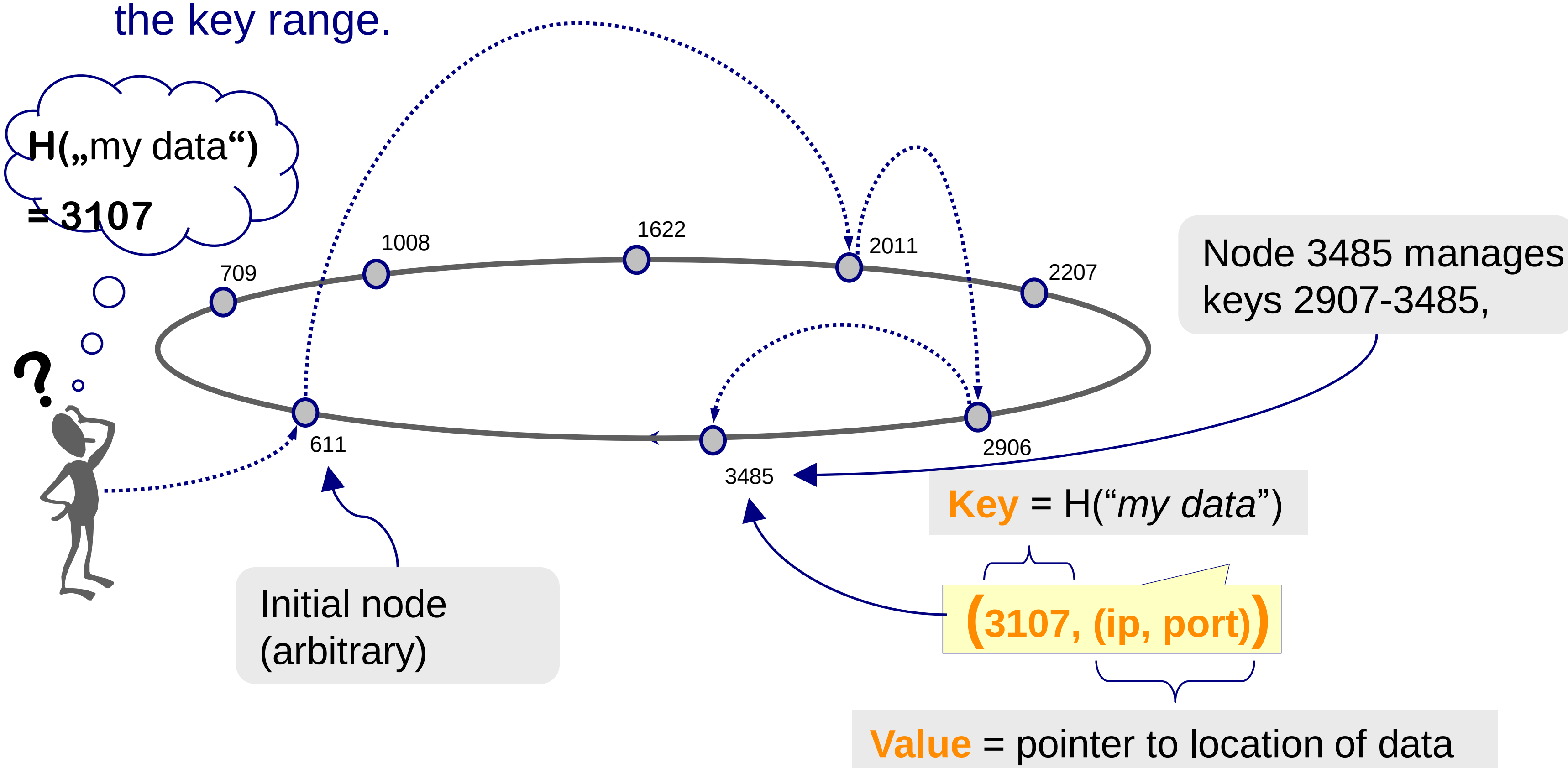
Step 2: Storing/looking up data (content-based routing)

Goal: a small and scalable effort

- O(1) with a centralized hash table
  - But: management of a centralized hash table is very costly.
- Minimum overhead with *distributed hash tables*
  - O(log N) DHT hops to locate object
  - O(log N): number of keys and routing information per node (N = # nodes)

Step 2: Routing to a Data Item (2)

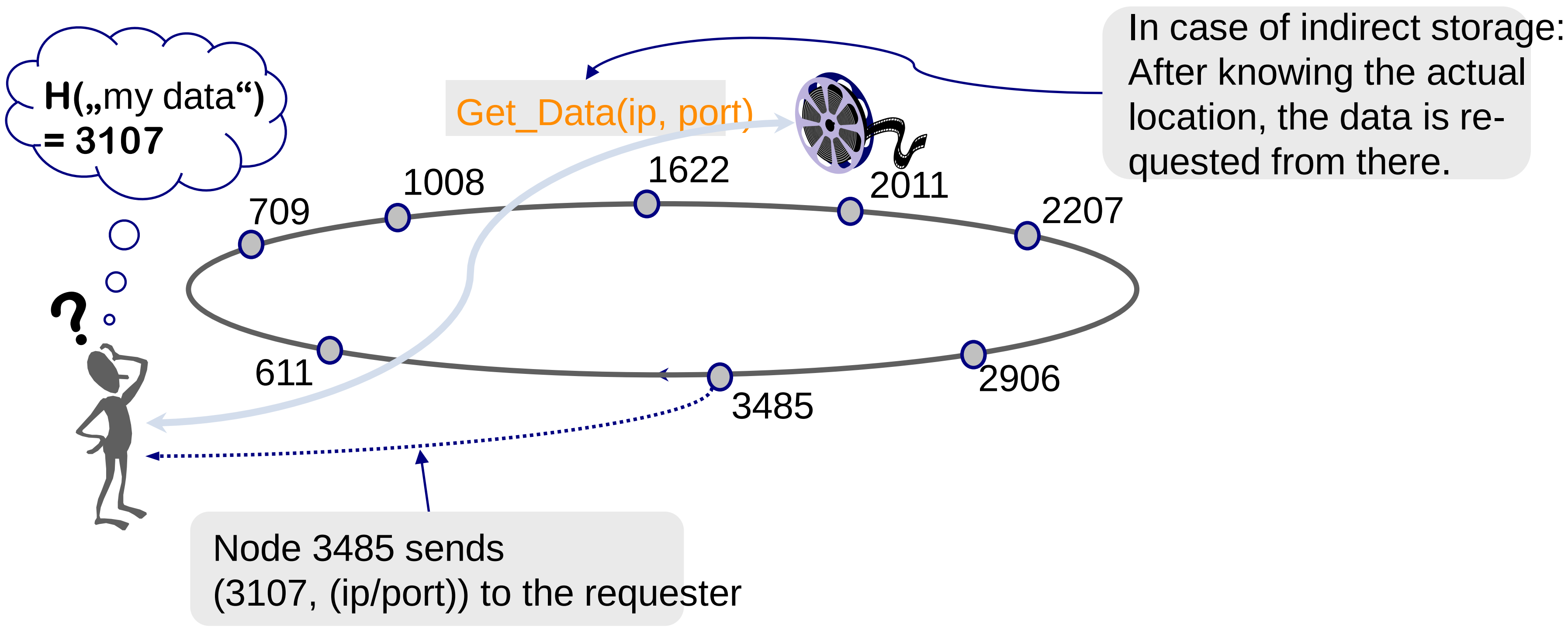
Routing to a key/value pair
- Start lookup at an arbitrary node of the DHT
- Route the request (possibly indirectly) to the peer node responsible for the key range.



Step 2: Routing to a Data Item (3)

Getting the content

- The key/value pair is delivered to the requester.
- The requester analyzes the key/value tuple (and downloads the data from the actual location in case of indirect storage).



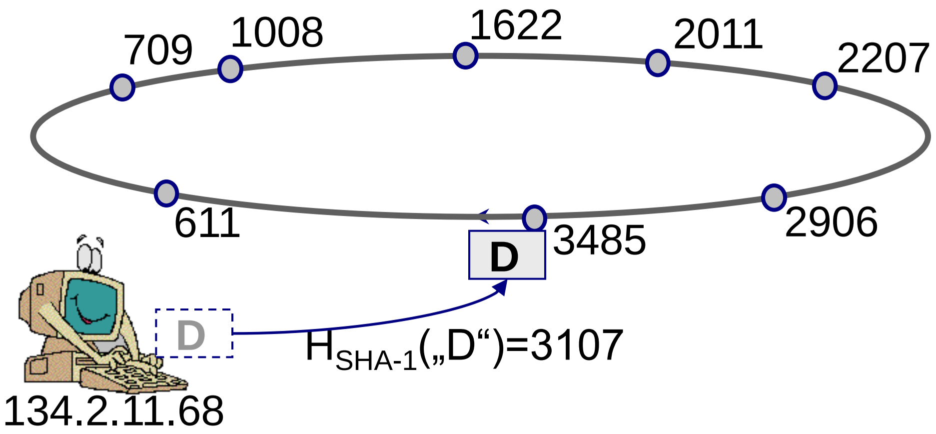
Association of Data with IDs – Direct Storage

How is content stored in the nodes?

- Example:
  H("*my data*") = 3107 is mapped into the DHT address space.

Direct storage

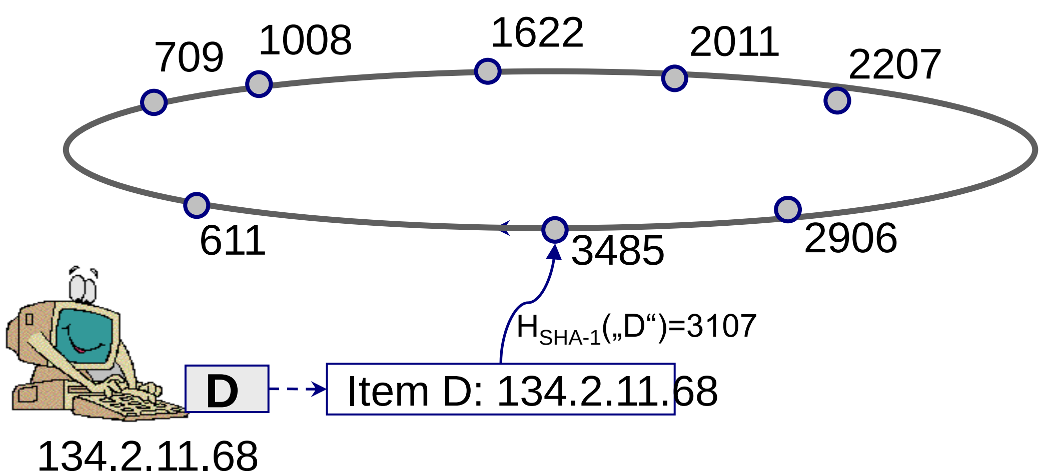
- Content is stored in the node responsible for H("*my data*")
  → Okay if the amount of data is small (<1 kB). Inflexible for large contents.



Association of Data with IDs – Indirect Storage

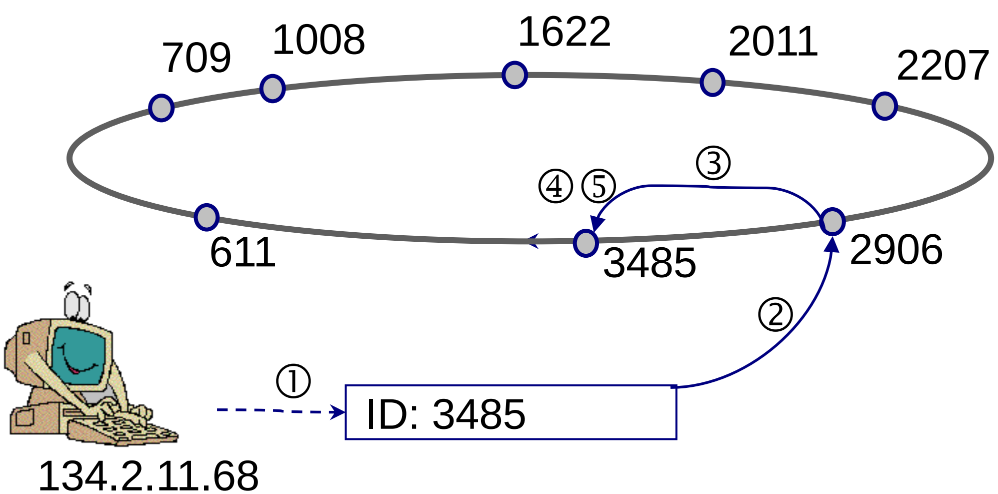
Indirect storage

- Nodes in a DHT store tuples (key,value)
  - Key = Hash(„*my data*") → 2313
  - Value is then the *storage address* of the content:
    (IP, Port) = (134.2.11.140, 4711)
- More flexible, but requires one step more to reach the content.



Node Arrival

1. Calculation of node ID
2. New node contacts DHT via arbitrary node.
3. A particular hash range is assigned to the node.
4. The key/value pairs of this hash range are stored on the new node (usually with redundancy).
5. The node is integrated into the routing environment.



Node Failure / Node Departure

Failure of a node

- Use of redundant storage of the key/value pairs (if a node fails)
- Use of redundant/alternative routing paths
- Key/value usually still retrievable as long as at least one copy remains.

Departure of a node

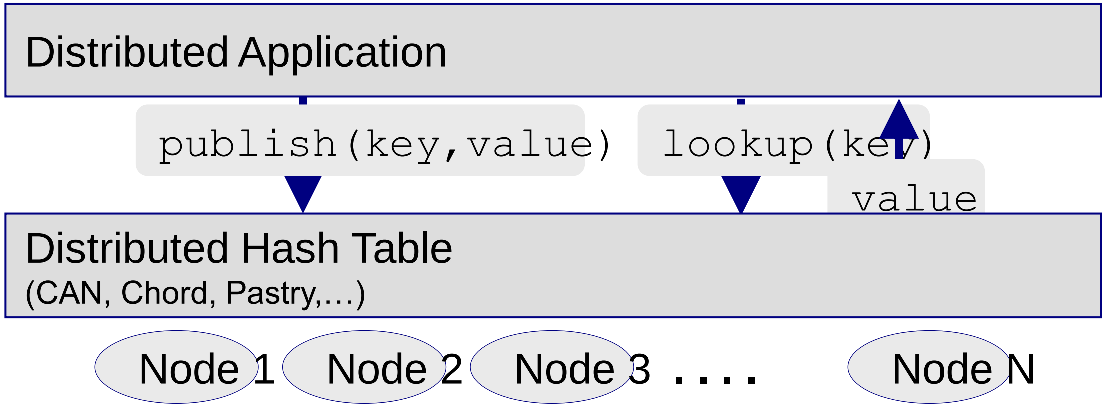
- New partitioning of the hash range to neighbor nodes
- Copy the key/value pairs to the neighbor nodes
- Remove the departing node from the routing environment.

DHT Interfaces

Generic interface of Distributed Hash Tables

- Provisioning of information
  - publish(key,value)
- Requesting of information (search for content)
  - lookup(key)
- Reply
  - value

DHT approaches are then **interchangeable** (implementing the same inter-face).



Conclusions

- Data and nodes are mapped into the same address space.
- Use of routing information for efficient search for content.
- Keys are evenly distributed across nodes of a DHT
  - No bottlenecks
  - A continuous increase in the number of stored keys is possible
  - Failures of nodes can be tolerated
  - Survival of attacks is possible
- A self-organizing system
- Simple and efficient realization
- Supports a wide spectrum of applications:
  - Flat (hash) key without a semantic meaning
  - Value depends on the application

5.2 Chord

Overview

- Developed at UC Berkeley and MIT, published at ACM SIGCOMM in 2001
- An early and successful algorithm
  - Simple and elegant
  - Easy to understand and implement
  - Many improvements and optimizations exist
- **Main functions**
  - Routing
    - A flat logical address space: 160-bit identifiers instead of IP addresses
    - Efficient routing in large systems: log(N) hops for a network of N nodes
  - Self-organization
    - Can handle frequent node arrival, departure and failure (“churn”)

Chord Interface and Identifiers

User interface

- put (key,value) inserts data into Chord
- value = get (key) retrieves data from Chord

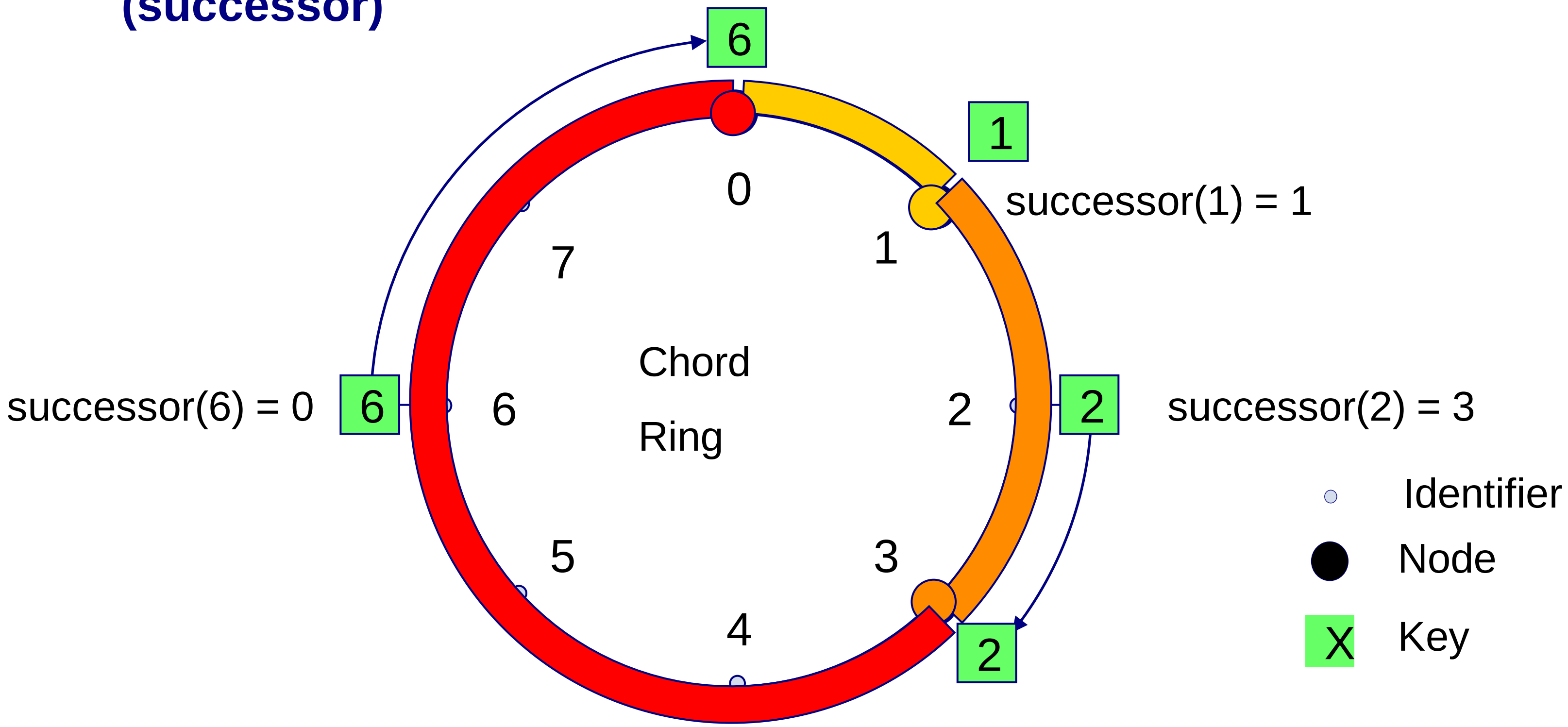
Identifiers

- Derived from the hash function
  - e.g., SHA-1, 160-bit output → 0 <= identifier < 2^{160}
- *Key* associated with each data item
  - e.g., key = SHA-1(value)
- *ID* associated with each host
  - e.g., id = SHA-1(IP address ⊕ port)

Chord Topology (1)

The Chord ring

- Keys and IDs are placed on a ring, i.e., all arithmetic happens modulo 2^{160}
- **(key, value) pairs are managed by the clockwise next node (successor)**

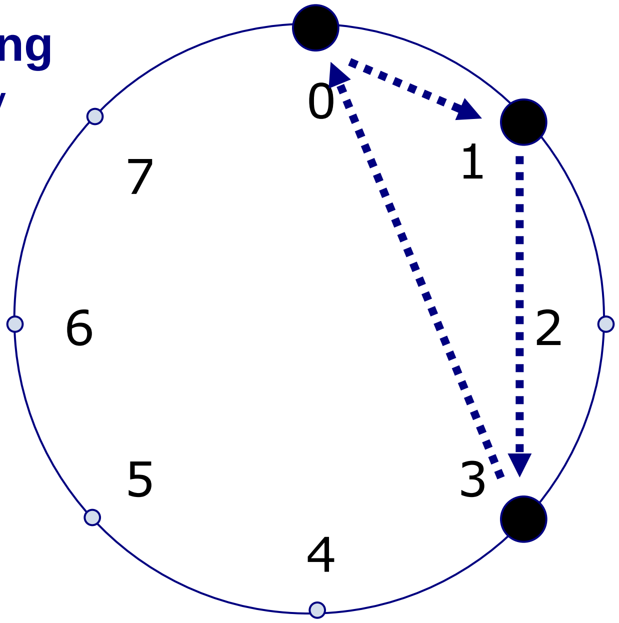


Chord Topology (2)

Topology determined by links between nodes
- Link: knowledge about another node
- Stored in a routing table on each node

Simplest topology
- circular linked list

Principle of consistent (distributed) hashing
- Initial idea: balance load among nodes by using a hash function to map nodes/data into the linear address space.
- Each node has a link to the next node (clockwise)



Chord Routing (1)

Primitive routing in distributed hashing

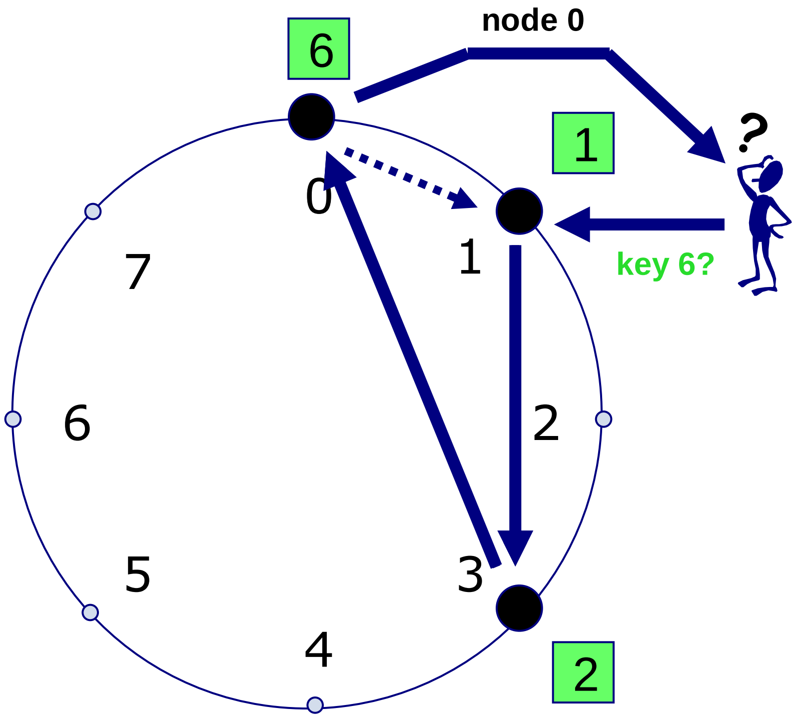
- Forward query for key k to the next node until successor(k) is found
- Return result to the source of the query

Advantages

- Simple
- Little node state needed

Disadvantages

- Poor lookup efficiency:
  N/2 hops on the average
  for N nodes (= O(N))
- Per-node state just O(1)
- Poor scalability
- A node failure breaks the circle.



Chord Routing (2)

Advanced routing in distributed hashing

- Store links to z next neighbors
- Forward queries for k to the farthest known predecessor of k
- For z = N: a fully meshed routing system
  - Lookup efficiency: O(1)
  - Per-node state: O(N)
- Still poor scalability

Scalable routing

- A mix of short- and long-distance links is required:
  - Accurate routing in the node's vicinity
  - Fast routing progress over large distances
  - Bounded number of links per node

Chord's routing table: *finger table*

- Stores log(N) links per node
- Covers exponentially increasing distances:
  - Node n: entry i (*i-th "finger"*) points to successor($n+2^i$)

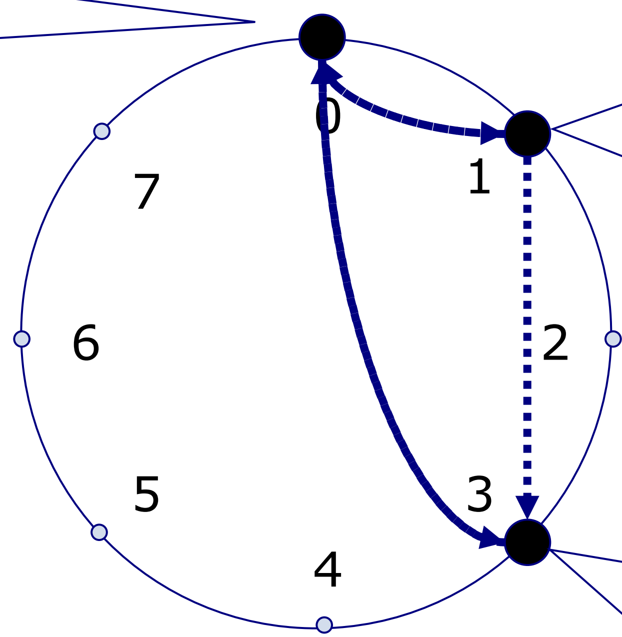
Chord Routing (3)

Chord routing: Example 1

finger table at node 0

i	n+2^i	succ.
0	1	1
1	2	3
2	4	0

keys
6



finger table at node 1

i	n+2^i	succ.
0	2	3
1	3	3
2	5	0

keys
1

finger table at node 3

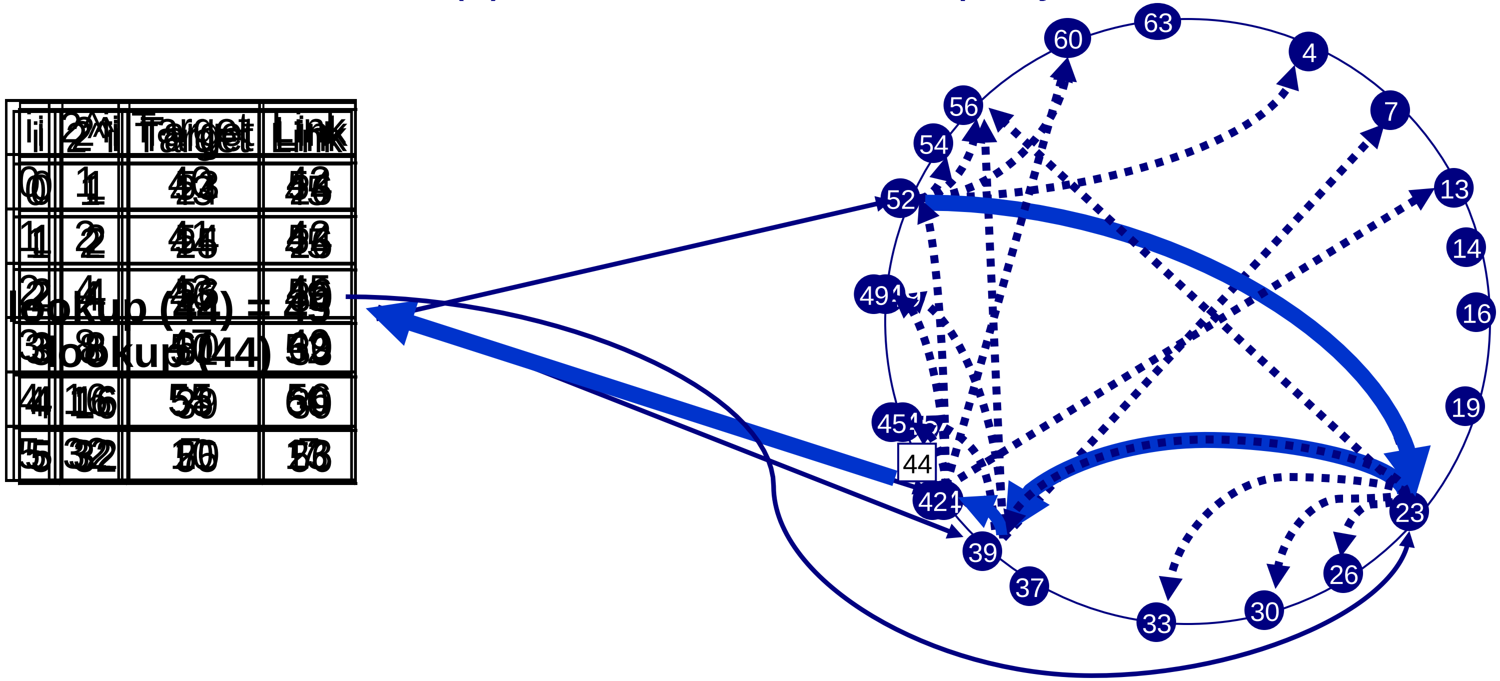
i	n+2^i	succ.
0	4	0
1	5	0
2	7	0

keys
2

Chord Routing (4)

Chord routing: Example 2

- Each node n forwards the query for key k clockwise
  - *to the farthest finger preceding k*
  - until n = predecessor(k) and successor(n) = successor(k)
  - returns successor(n) to the source of the query



Chord Self Organization (1)

Handle a changing network environment

- Arrival of new nodes
- Departure of participating nodes
- Failure of nodes

Maintain consistent system state for routing

- Keep routing information up to date
  - The correctness of the routing algorithm depends on the correct successor information.
  - Routing efficiency depends on correct finger tables.
- Fault tolerance required for all operations.

Chord Self Organization (2)

Chord soft-state approach

- Nodes delete (key,value) pairs after a timeout of 30 s to some minutes.
- Applications need to refresh (key,value) pairs they wish to store periodically.
- Worst case: data unavailable for the refresh interval after a node failure.

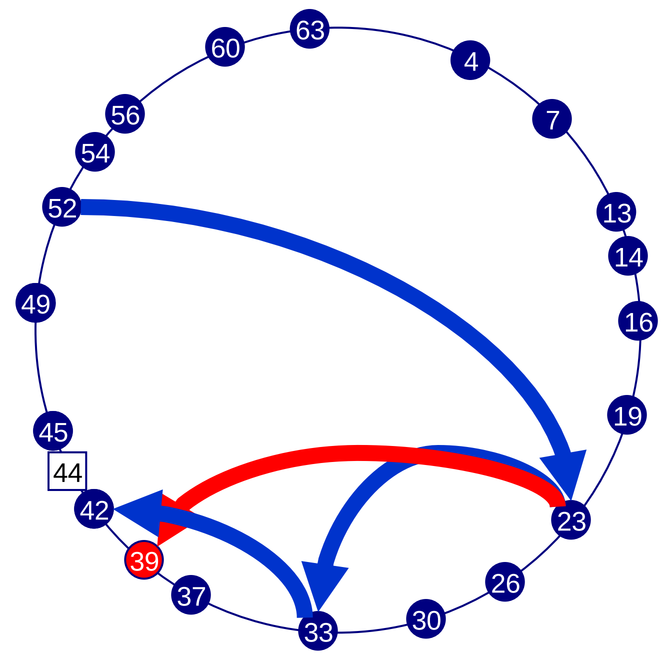
Chord Self Organization (3)

Finger failures during routing

- query cannot be forwarded to the finger entry
- forward to the previous finger (*do not overshoot destination node*)
- trigger repair mechanism: replace finger by its successor

Active finger maintenance

- periodically check liveness of fingers
- replace with correct nodes in case of failures
- trade-off: maintenance traffic vs. correctness and timeliness



Chord Self Organization (4)

Successor failure during routing

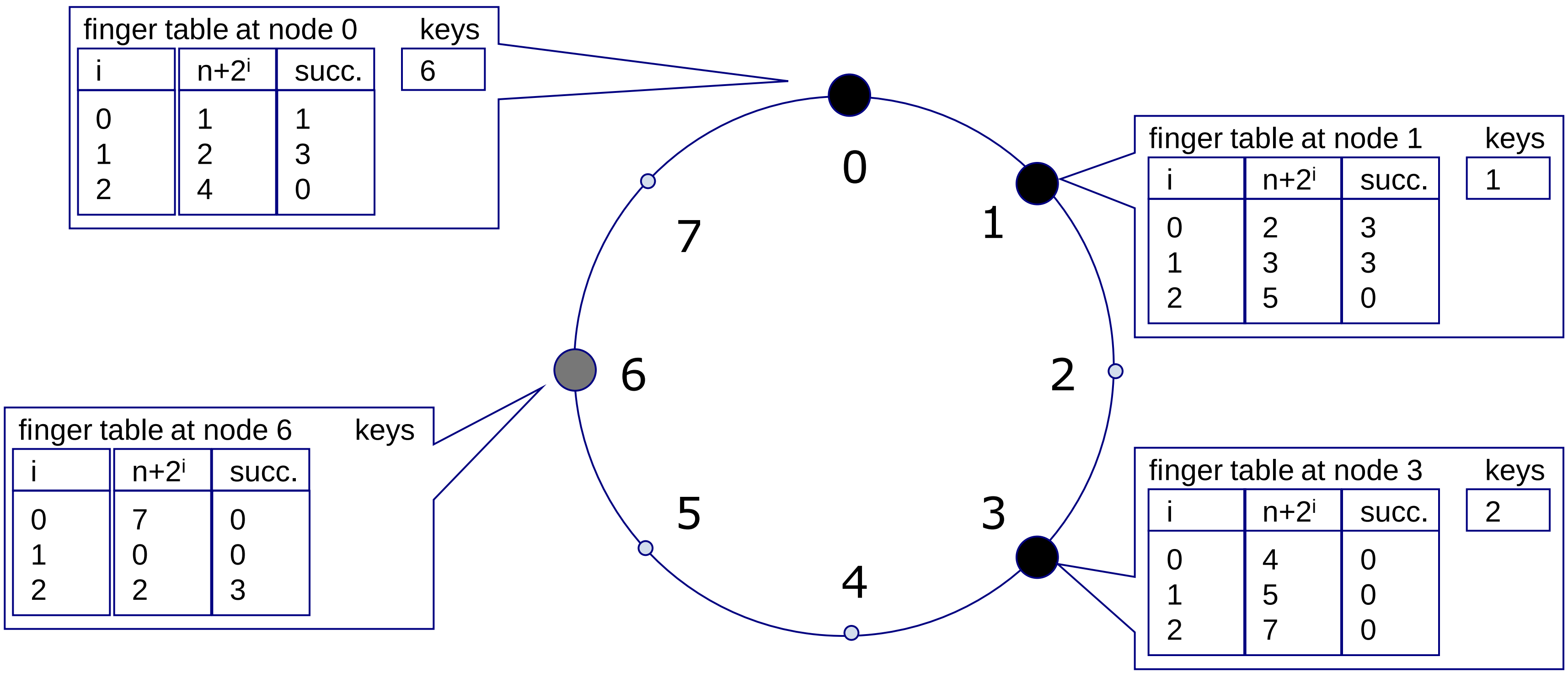
- Last step of routing can return a failed node to the source of the query
  -> all queries for the successor fail
- Store n successors in a *successor list*
  - successor[0] fails -> use successor[1], etc.
  - routing fails only if n consecutive nodes fail simultaneously.

Active maintenance of the successor list

- periodic checks, similar to finger table maintenance
- crucial for correct routing

Chord: Node Join (1)

- New node picks its ID
- Contacts existing node responsible for his range
- Constructs finger table via standard routing/lookup
- Retrieves (key, value) pairs from his successor.



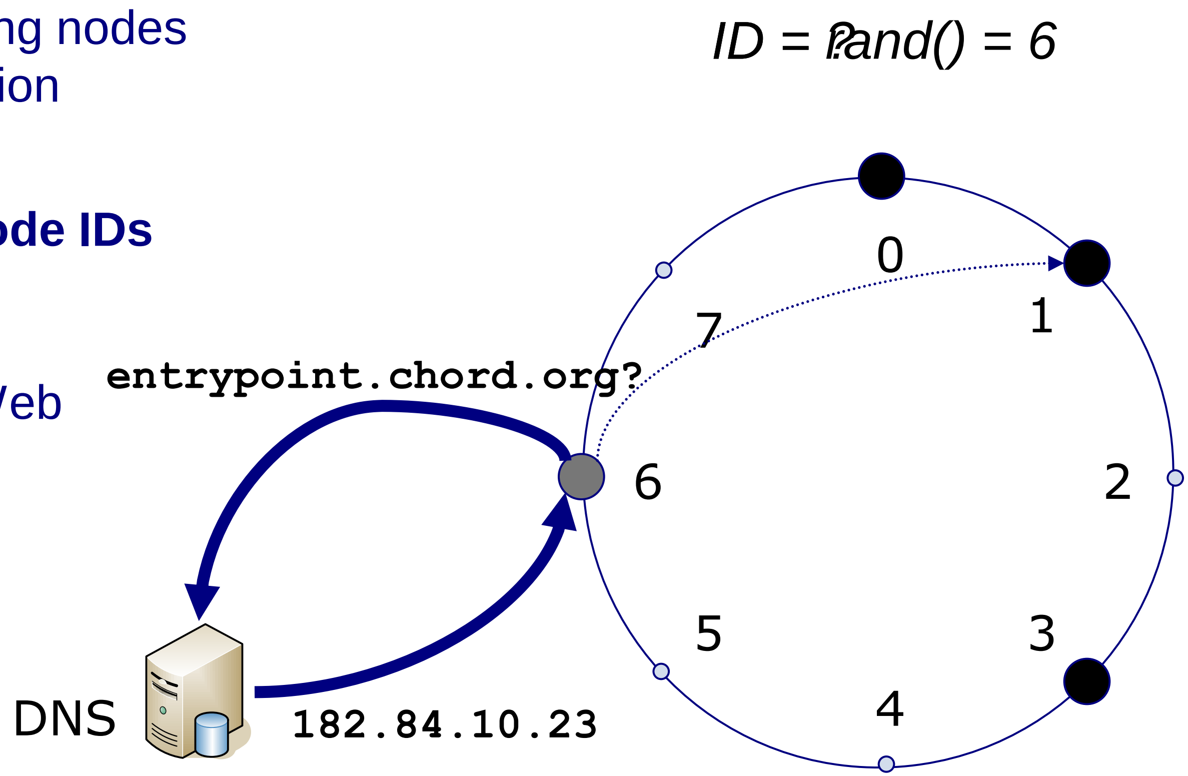
Chord: Node Join (2)

Examples for choosing new node IDs

- random ID: equal distribution assumed
- hash IP address and port
- place new nodes based on
  - load of the existing nodes
  - geographic location
  - etc.

Retrieval of existing node IDs

- Controlled flooding
- DNS aliases
- Published through Web
- etc.



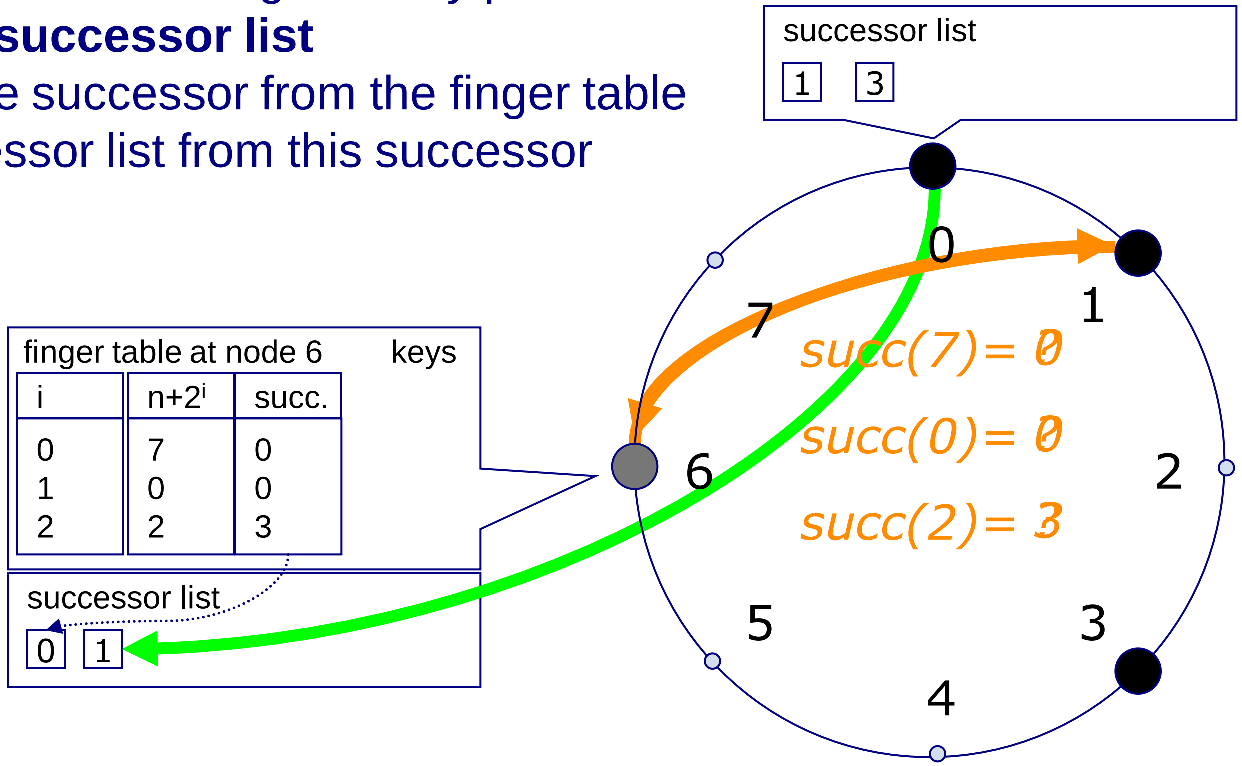
Chord: Node Join (3)

Construction of finger table

- iterate over finger table rows
- for each row: query entry point for successor
- use standard Chord routing on entry point

Construction of successor list

- add immediate successor from the finger table
- request successor list from this successor

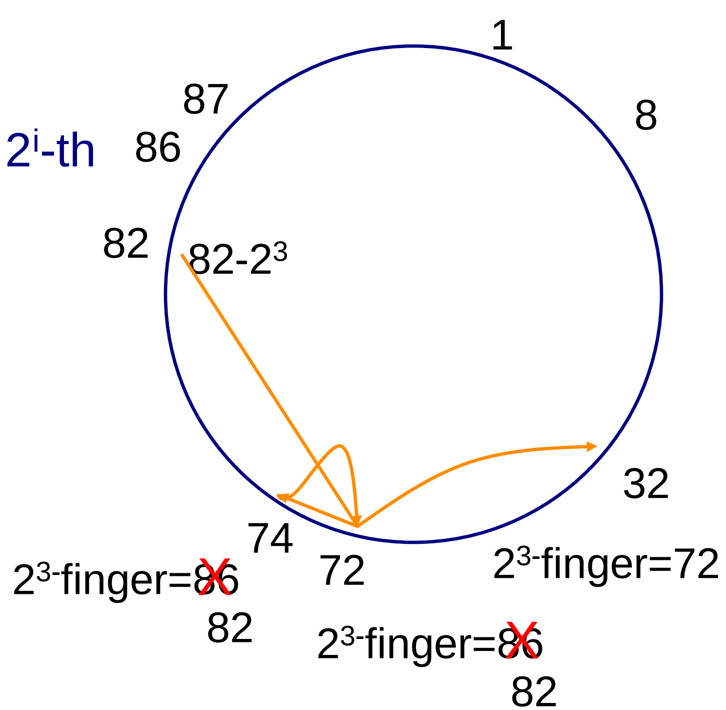


Chord: Node Join (4)

Update of finger pointers: Example

- Node 82 joins
- Finger entries to node 86 may now point to the new node 82
- Candidates for updates:
  - Nodes (counter-clockwise) whose 2^i-th finger entry have to point to 82
- Check predecessor's t_i of keys $(s – 2^i)$
  - route to $s - 2^i$
- If t's 2^i-finger points to a node beyond 82:
  - change t's 2^i-finger to 82
  - set t to predecessor of t and repeat
- ELSE continue with 2^{i+1}

example for i = 3



O(log² N) for looking up and updating the finger entries.

Chord: Node Departure (1)

Planned node departure

- a clean shutdown instead of failure

For simplicity: *treat as a failure*

- system already tolerant to failures
- soft state: automatic state restoration (state is lost for a short period)
- invalid finger table entries: reduced routing efficiency

For efficiency: handle explicitly

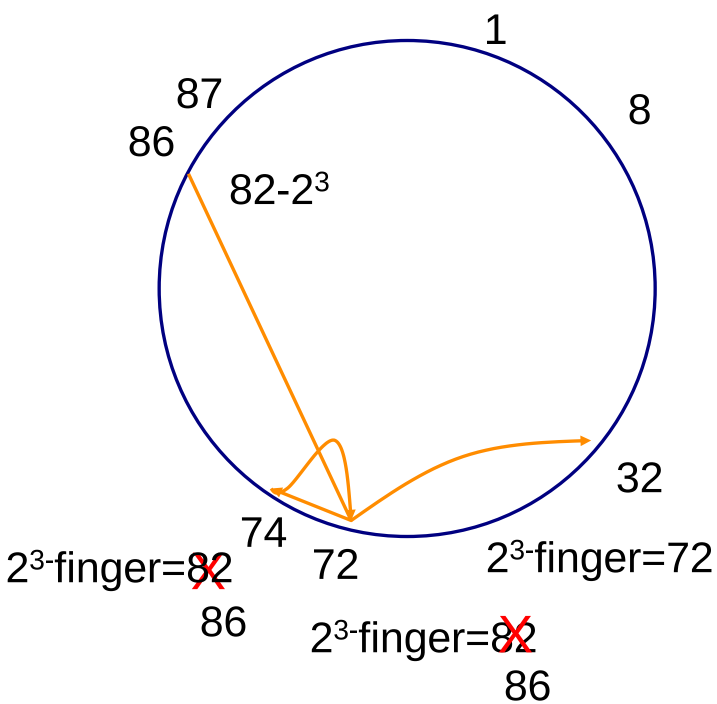
- notification by departing node of
  - successor, predecessor, nodes at finger distances
- copy (key, value) pairs before shutdown

Chord: Node Departure (2)

Similar procedure as with node join

- Update of fingers pointing to departing node similar to the node join procedure

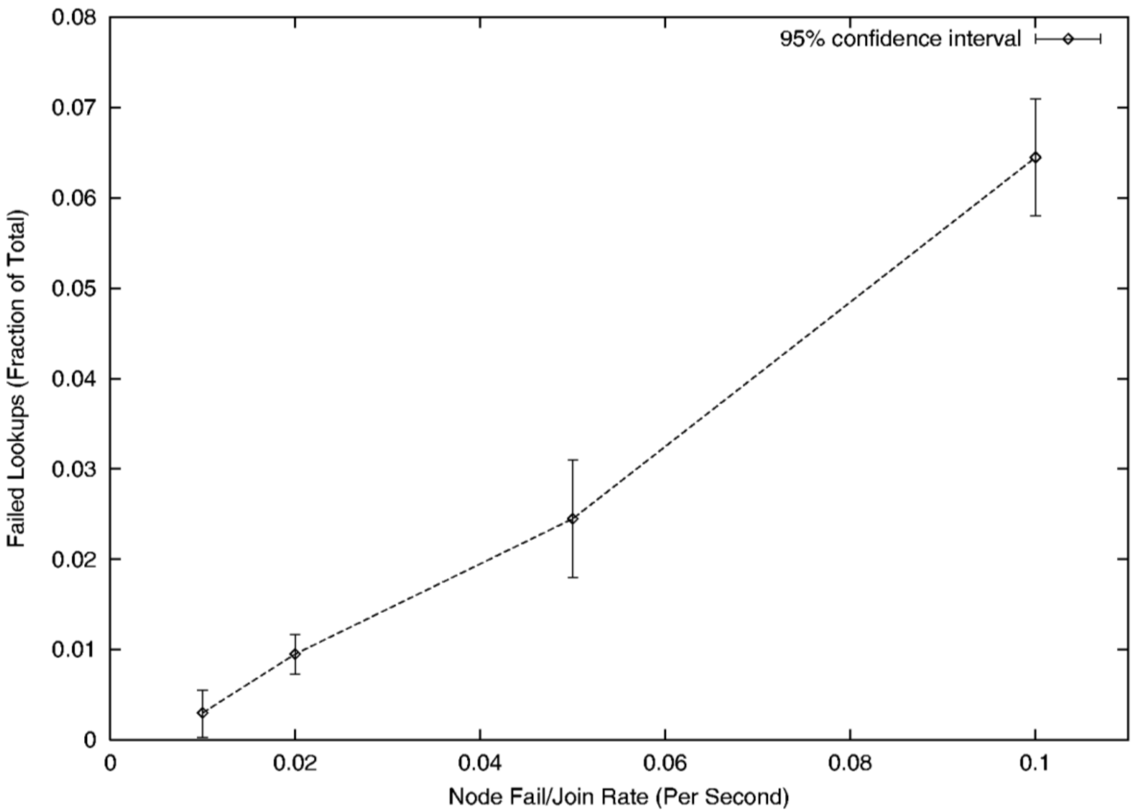
Example for i=3



Chord: Performance (1)

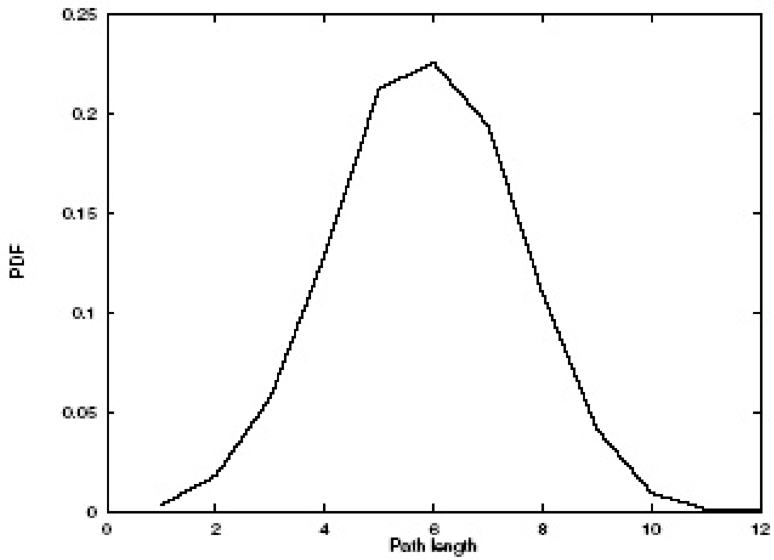
Impact of node failures on lookup failure rate

- lookup failure rate roughly equivalent to node failure rate

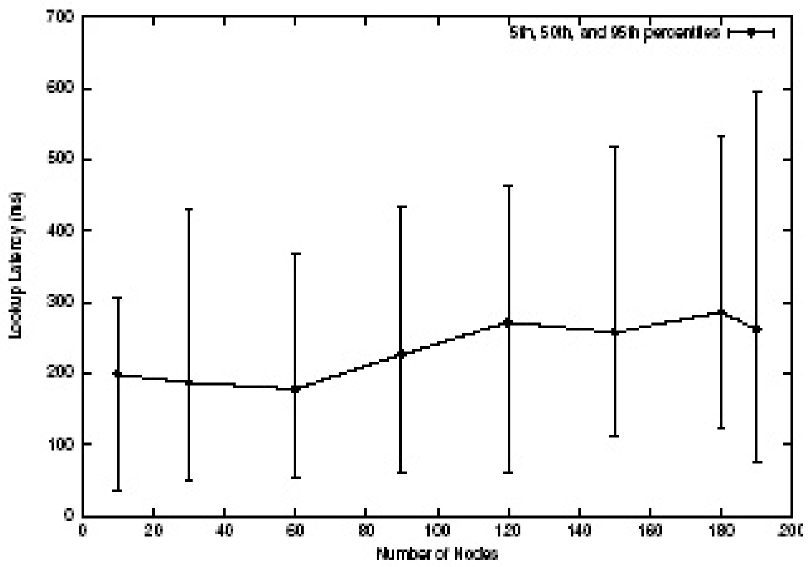


Chord: Performance (2)

Average path length

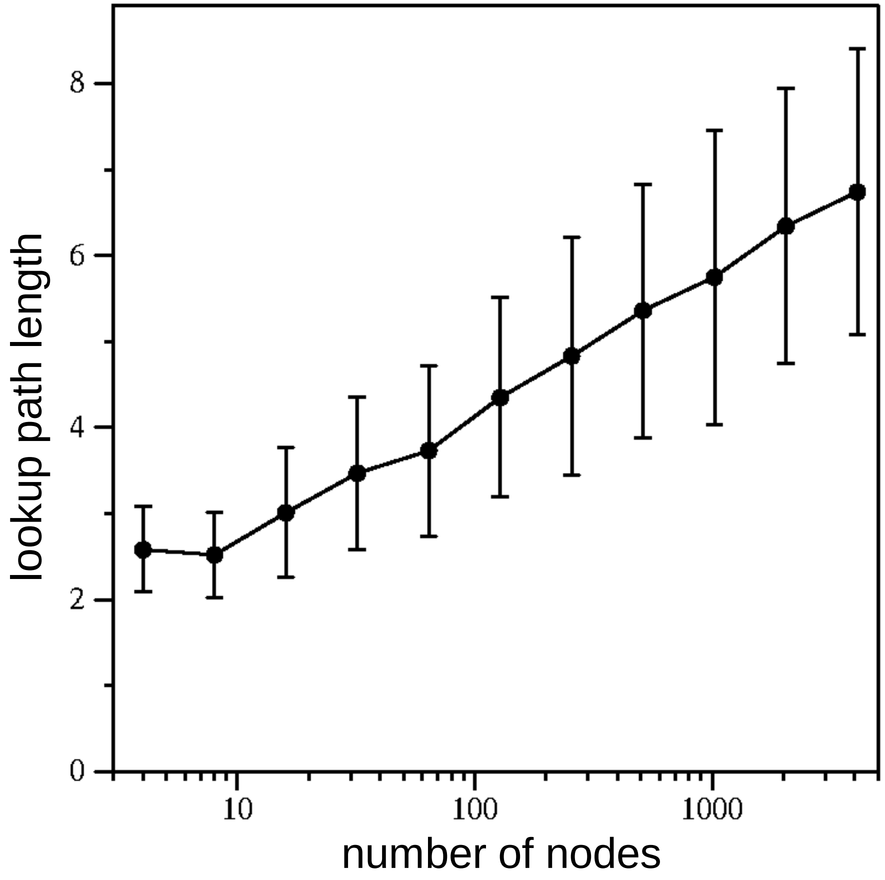


Moderate impact of number of nodes on lookup latency



Chord: Performance (3)

Lookup latency (number of hops/messages): ~ 1/2 $\log_2(N)$
Confirms the theoretical estimation.



Conclusions for Chord

Complexity

- Messages per lookup: **O(log N)**
- Memory per node: **O(log N)**
- Messages per management action (join/leave/fail): **$O(\log^2 N)$**

Advantages

- Theoretical models and proofs exist about the complexity
- Simple and flexible

Disadvantages

- No notion of node proximity and proximity-based routing optimizations
- Chord rings may become disjoint (partitioned) in realistic settings

By today, many improvements were published

- e.g., provisions for proximity, bi-directional links, load balancing, etc.

5.3 CAN (Content-Addressable Network)

An early and successful algorithm

Simple and elegant

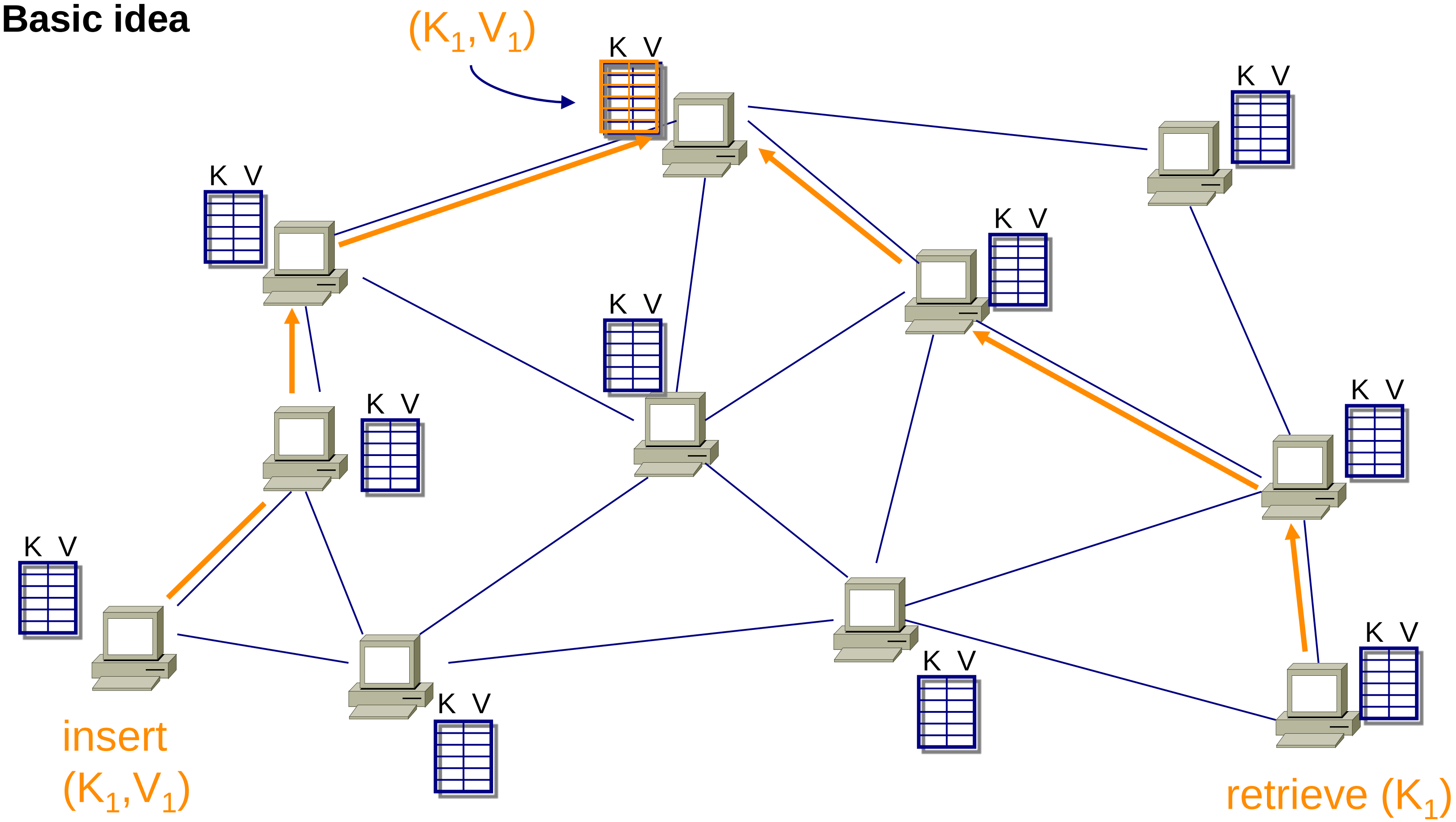
- Intuitive to understand and implement
- Many improvements and optimizations exist
- Published by Sylvia Ratnasamy et al. in 2001

Main responsibilities

- CAN is a distributed system that maps keys to values.
- CAN uses **distributed hashing**.
- Keys are hashed into a **D-dimensional space**
- Interface:
  - insert(key, value)
  - retrieve(key)

CAN Overview (1)

Basic idea



CAN Overview (2)

Solution

Virtual Cartesian coordinate space

Entire space is partitioned amongst all the nodes. Every node “owns” a zone in the overall space.

Abstraction

- can store data at “points” in the space
- can route from one “point” to another

A point is a node that owns the enclosing zone.

CAN Overview (3)

D-dimensional value space

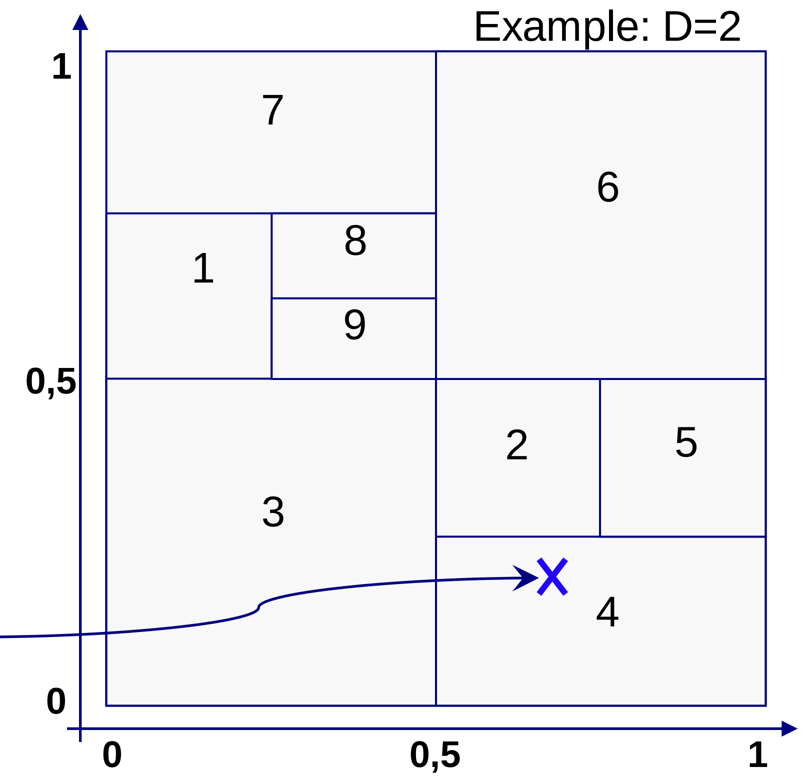
Hash value corresponds to a point in the D-dimensional space.

- H(„movie.avi“) $\rightarrow$ 4711 $\rightarrow$ (0.7, 0.2)
- DHT stores (key, value)-pairs

Complexity

- Search effort: $O(\frac{D}{4} N^{\frac{1}{D}})$
- Memory requirement: *O(D) = O(1)*

H(„movie.avi“) $\rightarrow$ (0.7, 0.2)

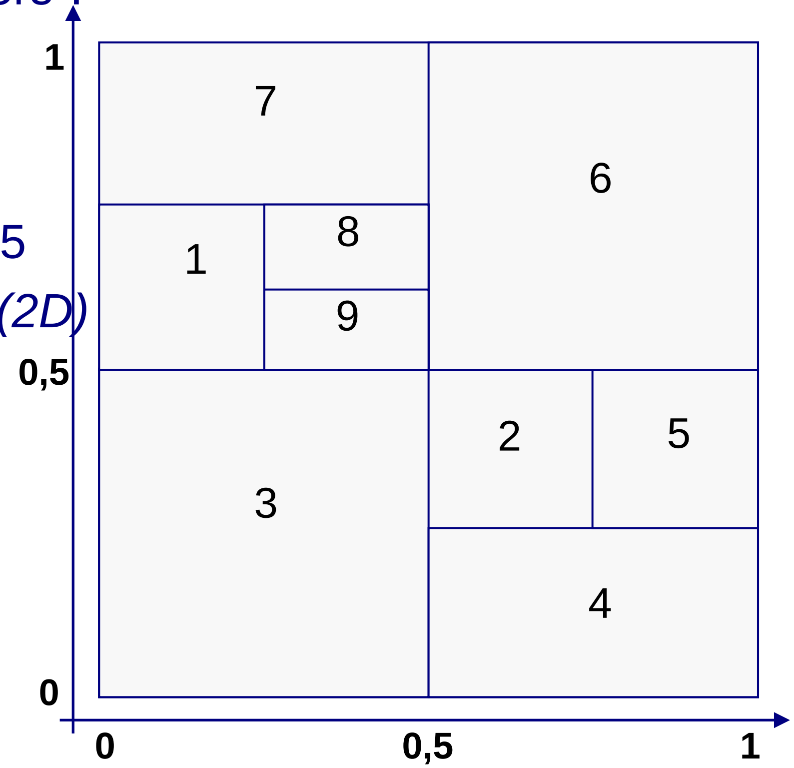


CAN Overview (3)

An overlay node manages one partition (rectangle) of the value space.

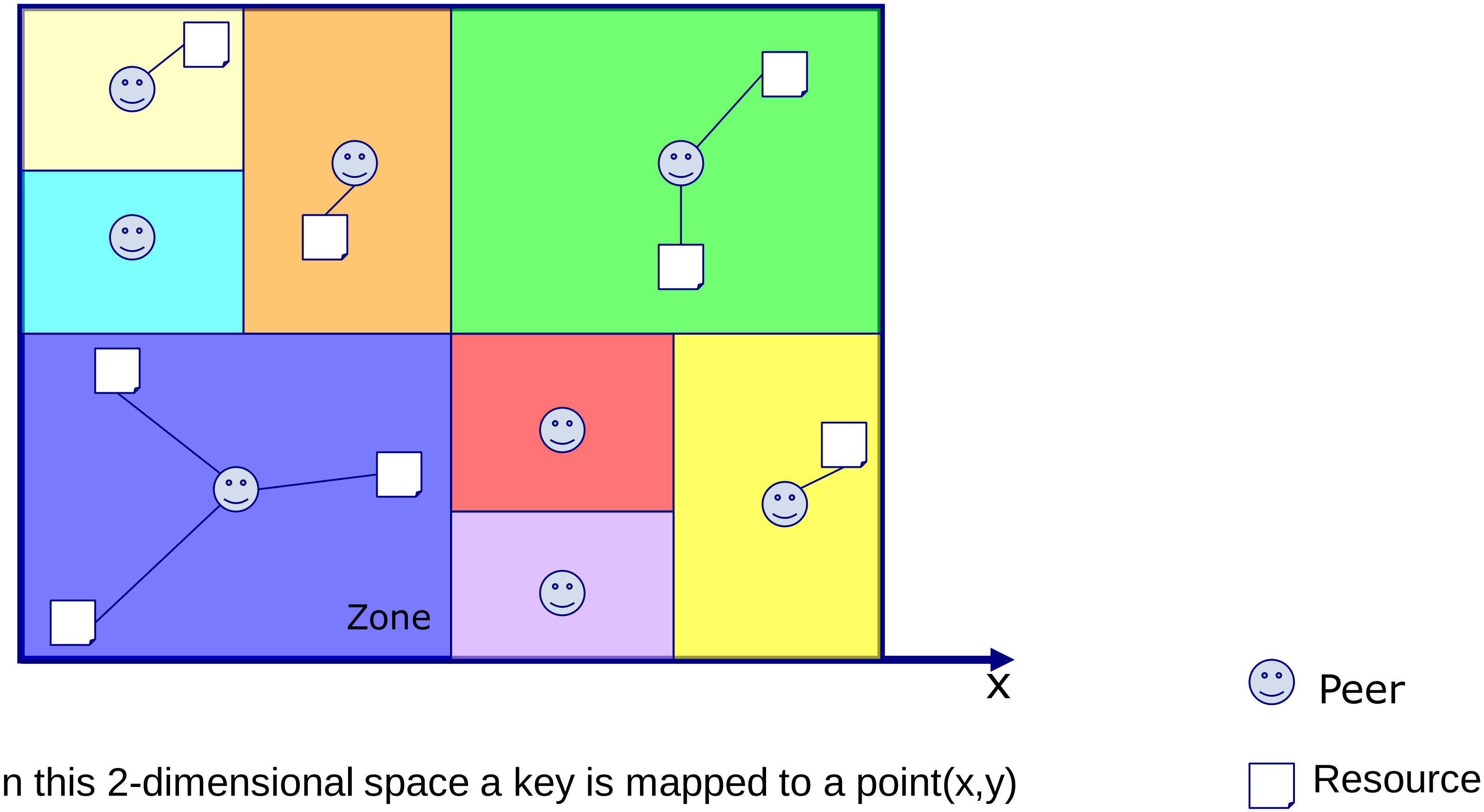
- Example: node 4 manages all values in x $\in$ [0.5, 1], y $\in$ [0, 0.25]
- Adjacent partitions are called “neighbors”:
  - Nodes 6, 2 and 4 are neighbors of node 5
  - „wrap around“ on DHT-borders: node 3 is also a neighbor of node 5
  - Expected number of neighbors: *O(2D)*

  → *independent of the size of the CAN network!*



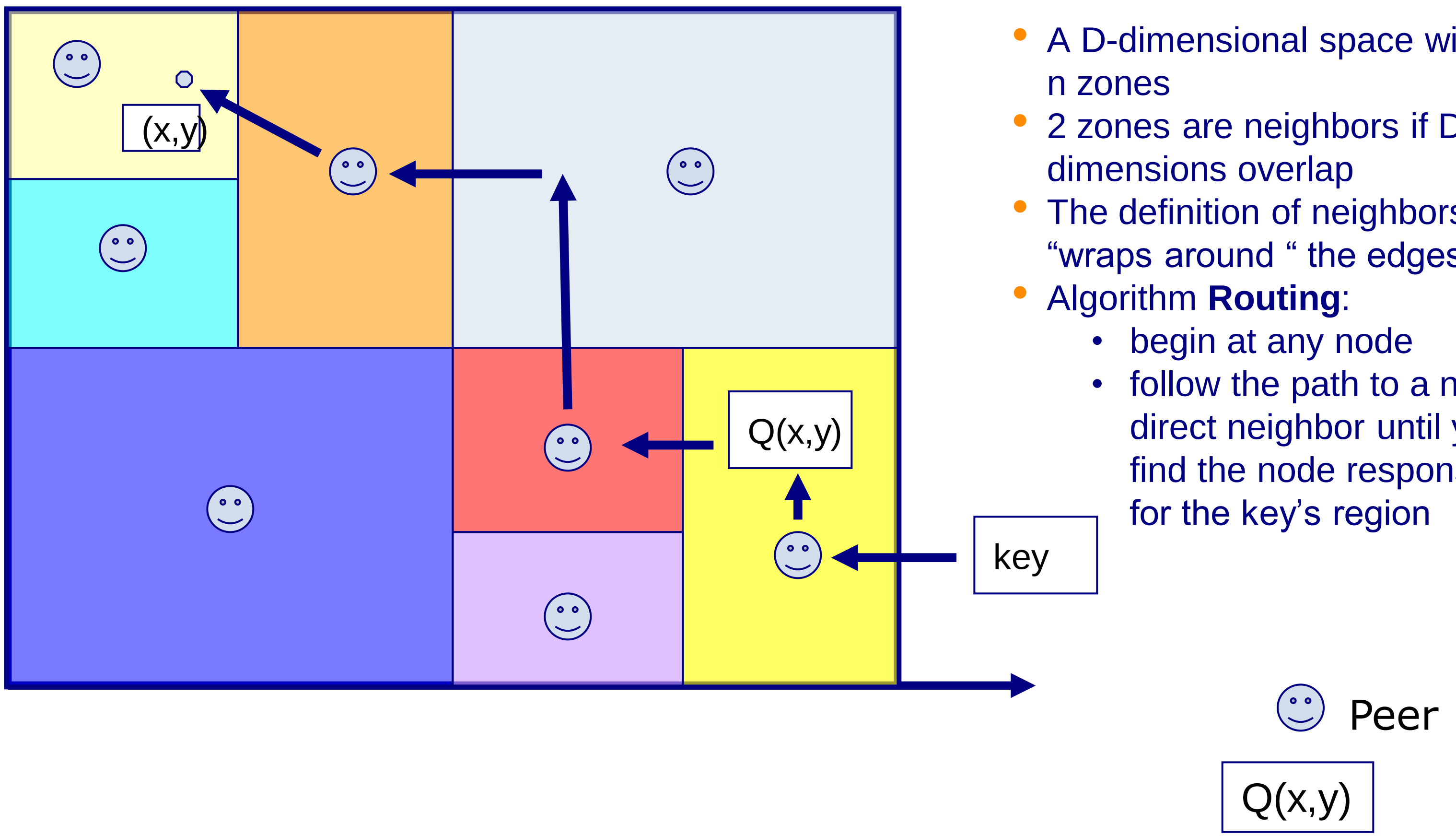
CAN Setup

State of the system at time t



In this 2-dimensional space a key is mapped to a point(x,y)

CAN Routing (1)



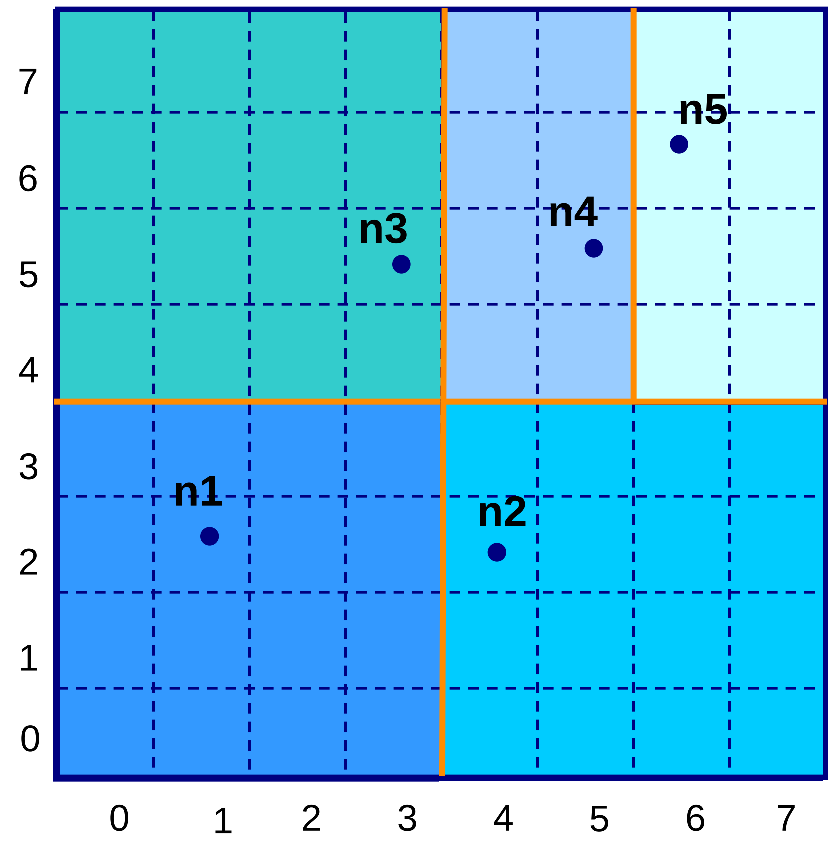
- A D-dimensional space with n zones
- 2 zones are neighbors if D-1 dimensions overlap
- The definition of neighbors “wraps around “ the edges
- Algorithm **Routing**:
  - begin at any node
  - follow the path to a nearer direct neighbor until you find the node responsible for the key’s region

CAN Routing (2)

Each node manages a rectangle with ratio 1:1, 1:2 or 2:1 (if D=2)

Example

- Dimension = 2, x=0…8, y=0…8
- Node n1 is the first node and thus manages the entire space
- Node n2 joins the CAN-Network: the space is split between n1 und n2
- Join of node n3
- Join of node n4
- Join of node n5



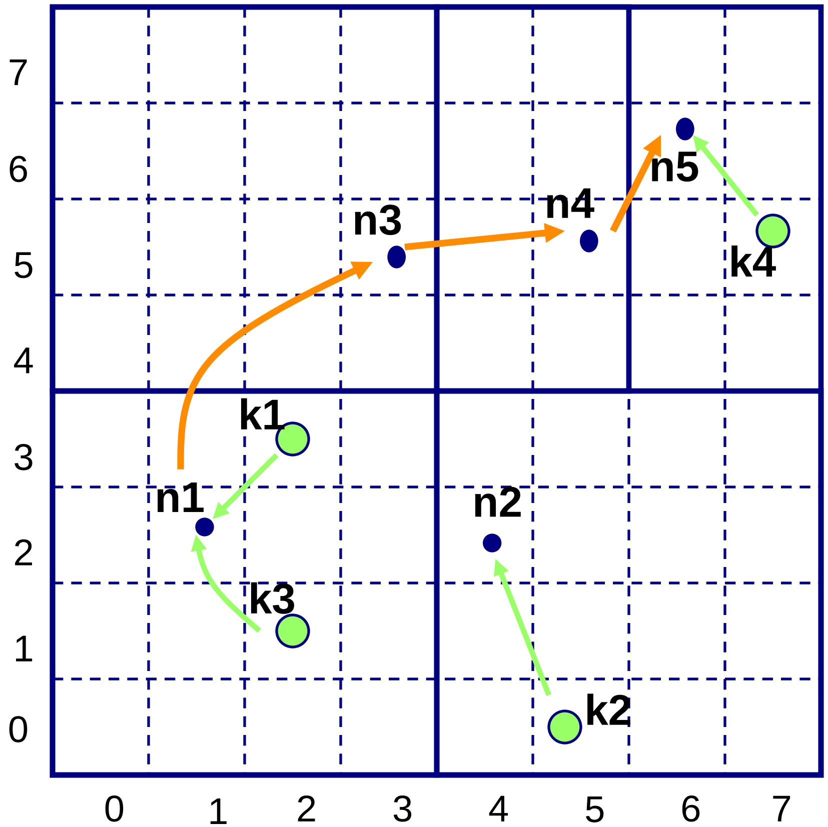
CAN Routing (3)

Data location is associated with coordi-nates derived from the key.

A (key, value)-pair is stored at the node responsible for the respective section.

A query for a key is always forwarded via neighbors:

- Entry point at some known node, e.g., n1
- Lookup for key k4



CAN Routing (4)

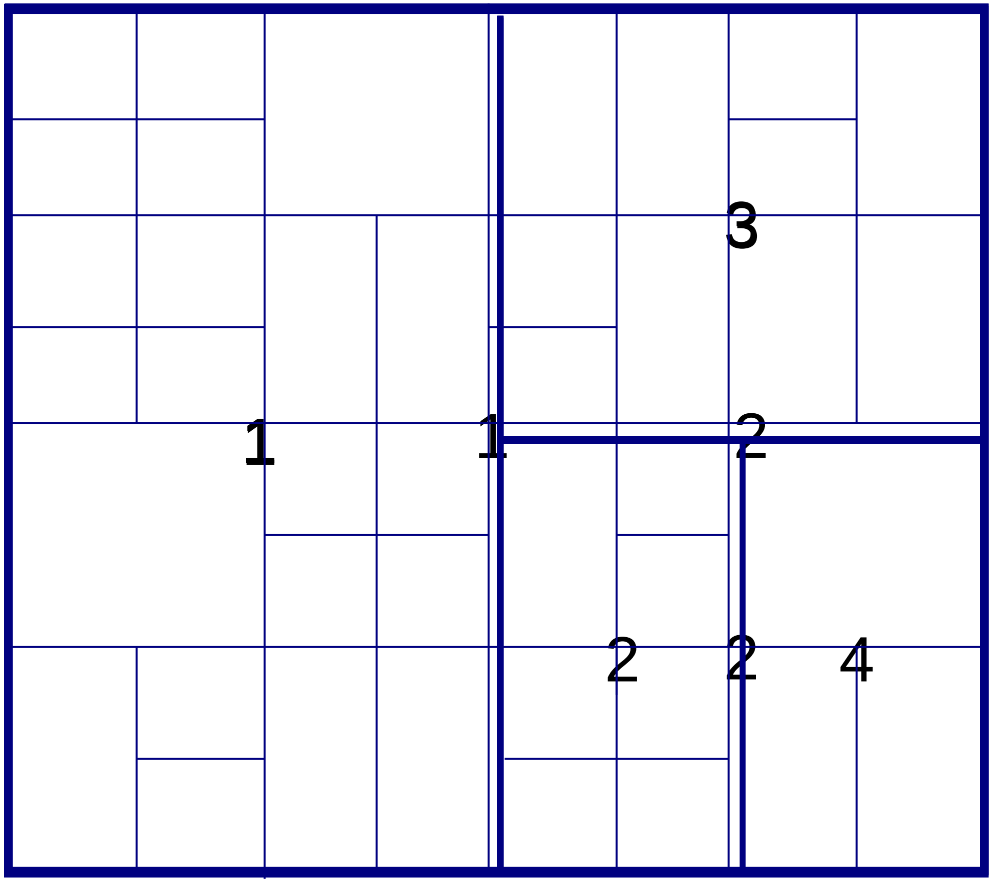
Path selection in CAN

- Routing along the *shortest path* in the D-dimensional space
- Details:

  The distance decreases continuously

  - effort: $O(\frac{D}{4} N^{\frac{1}{D}})$ hops

CAN: A Simple Example (1)



CAN: A Simple Example (2)

node I::insert(K,V)

node J::retrieve(K)

(1) a = $h_x(K)$

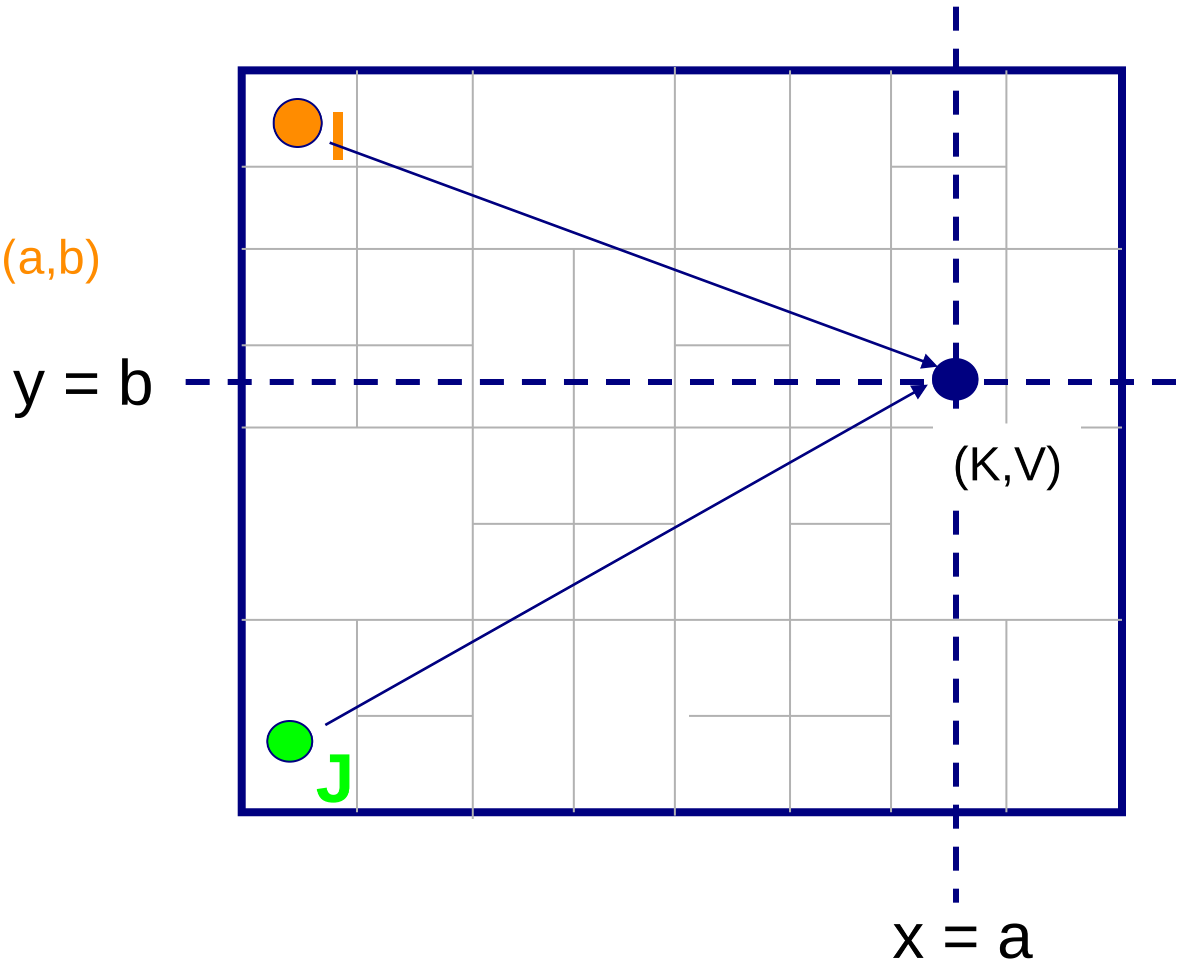
(1) a = $h_x(K)$

b = $h_y(K)$

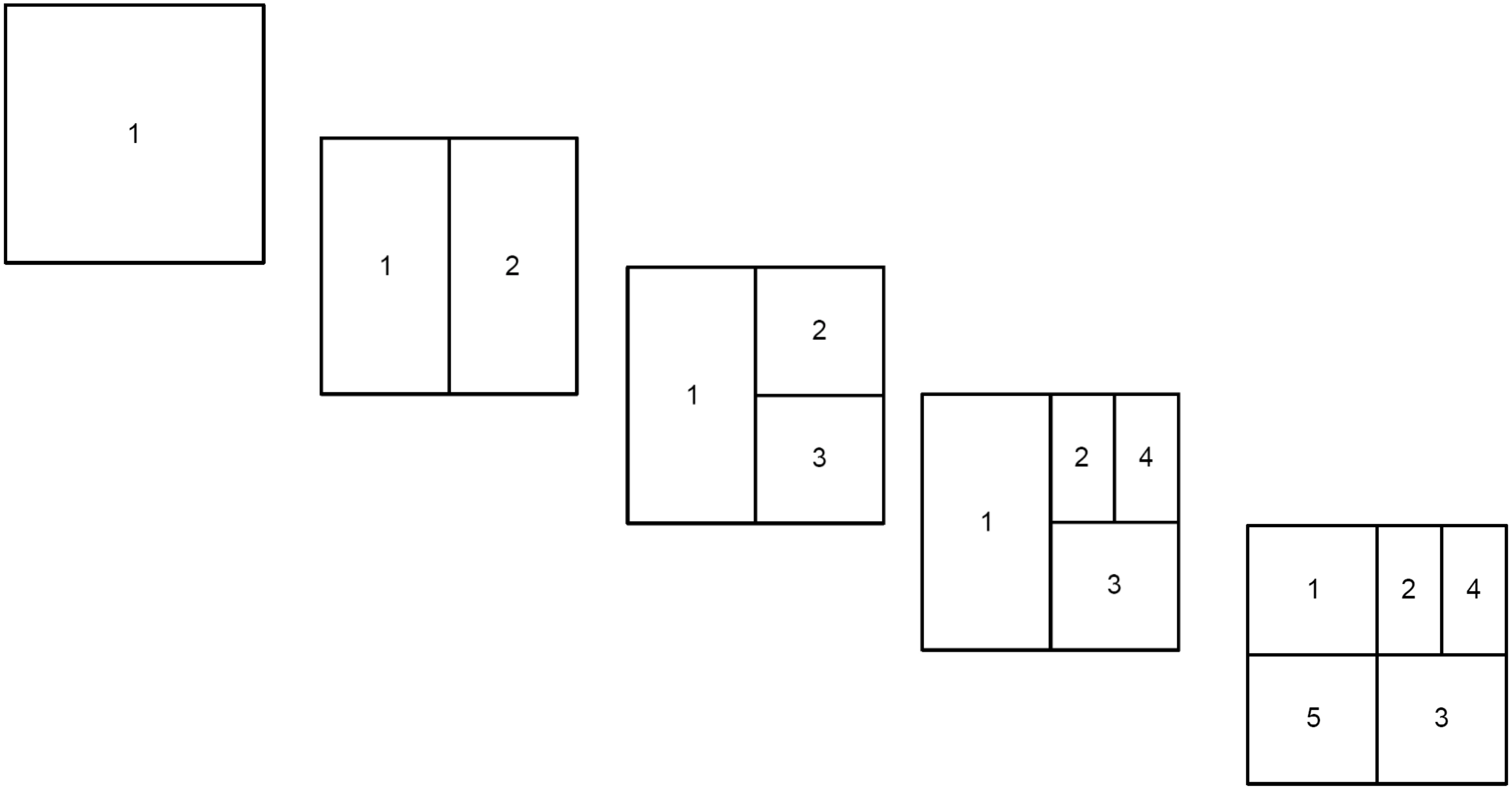
(2) route "retrieve(K)" to (a,b)

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

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



Partitioning of CAN Ranges (1)



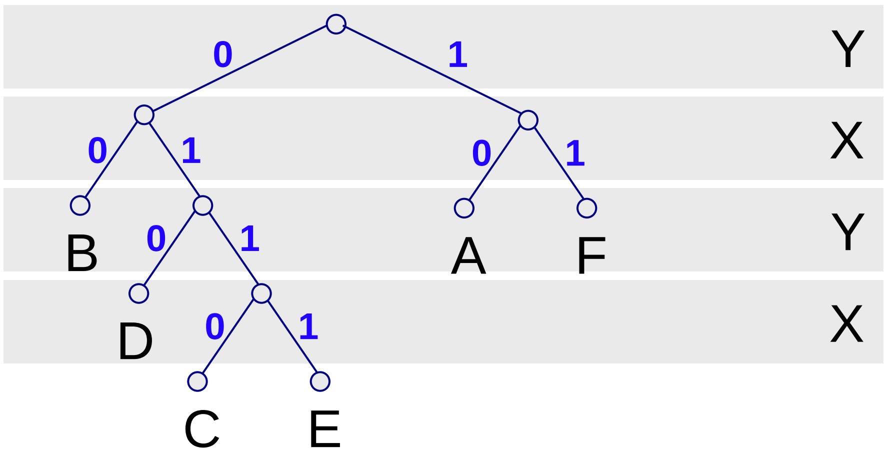
Partitioning of CAN Ranges (2)

Partitioning is performed according to some rules

- Strict sequencing of value range partitioning
- According to the order D
  - For example: x, y, z, x, y, z, ... if D=3

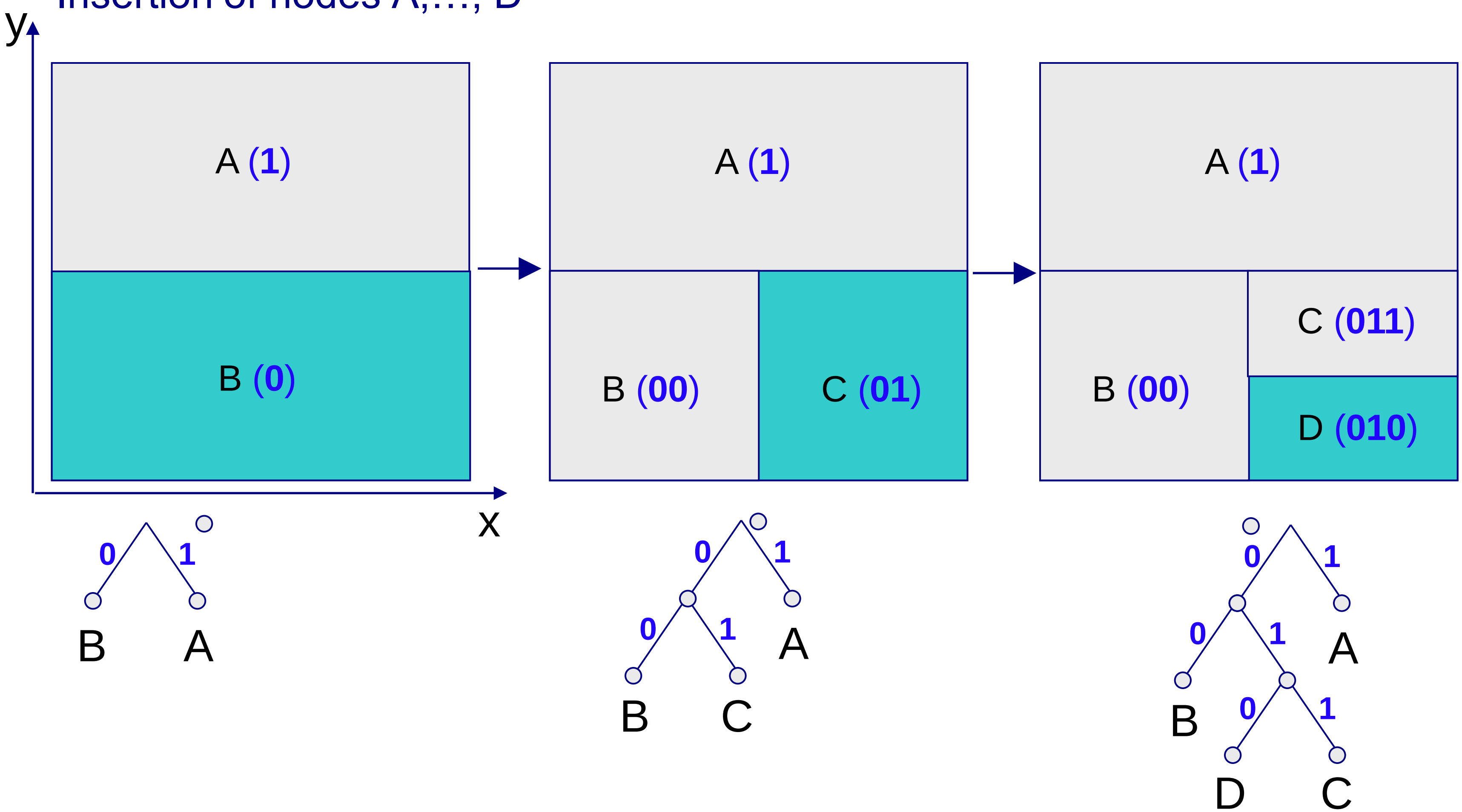
Partitioning tree

- Reflects „history“ of the partitioning process
- Important for fusion of ranges in the case of exit or failure of nodes



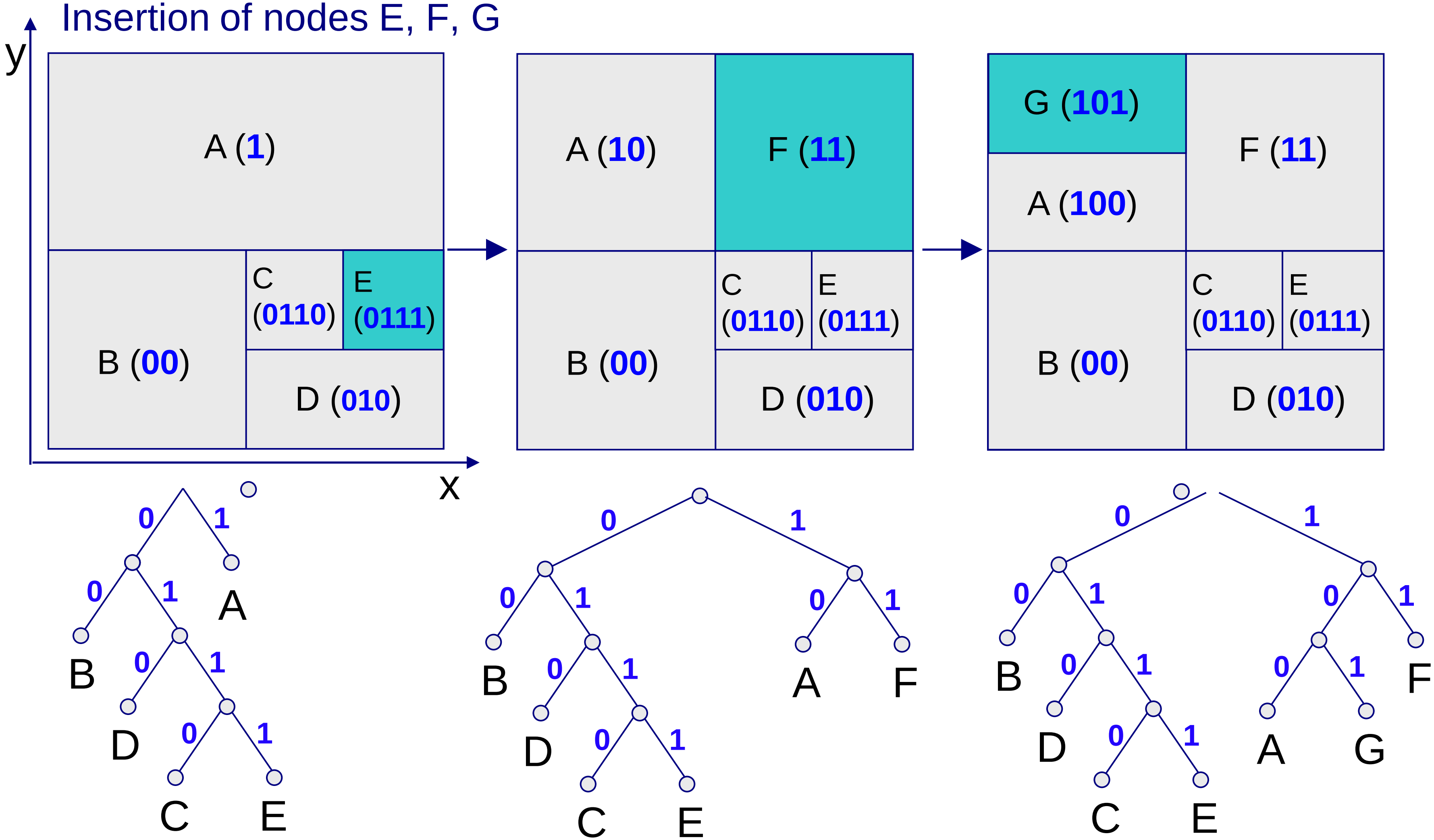
Structure of a CAN – Example (1)

Insertion of nodes A,…, D



Structure of a CAN – Example (2)

Insertion of nodes E, F, G



Removal of a Node from a CAN

Removal of a node from a CAN

- Region and thus managed key/value pairs are transferred to a neighbor:
  - Ideal case: regions can be merged according to prior partitioning → tree
  - Otherwise: neighbor with smallest number of key gets both regions to manage.
- Exit of a node: regular transfer procedure
- Failure of a node: TAKEOVER procedure
  - Non-appearance of periodic update information at neighbors
  - Neighbors initiate timer in proportion to the size of the region
  - Smallest neighbor signals TAKEOVER to other neighbors and takes over the region
- Restructuring (topological optimization) is performed **in the back-ground.**

Performance Improvement of CAN (1)

Complexity of CAN

- State information per node: O(D) (independent of N!)

Routing: $O(\frac{D}{4} N^{\frac{1}{D}})$ hops (within overlay!) (linear tendency!)

  - Effort = O(log N), with D = log N
  - Problem: N has to be known before

- Approaches for improvement of the search effort
  - Also applicable for other DHT approaches

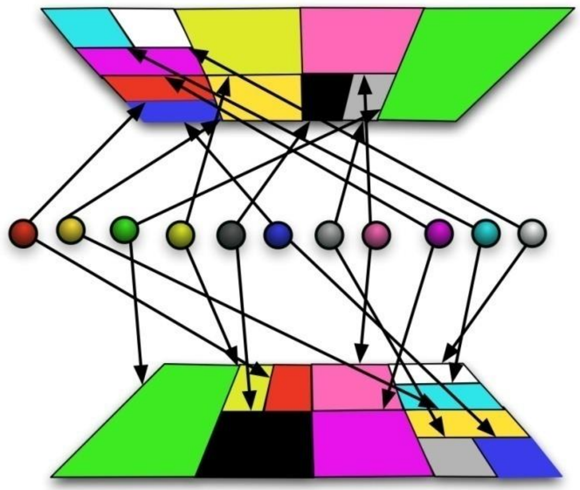
Performance Improvement of CAN (2)

More dimensions

- Increases the number of neighbors – decreases the index structure
- More path selection possibilities

More concurrent coordinate systems (*realities*)

- More concurrent distributed hash tables – nodes are members of *r* hash tables
- (K,V)-tuple is saved in *r* hash tables
    - Mapping of keys onto *r* different coordinate systems via different hash functions
    - All “realities” are checked in each routing step



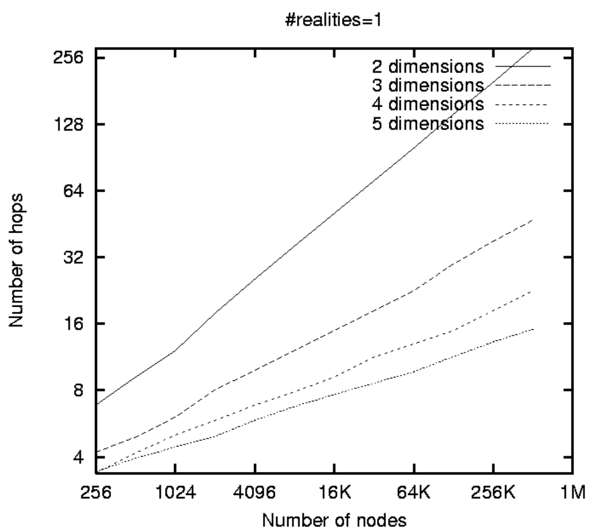
More Dimensions vs. Parallel Coordinate Systems

More dimensions

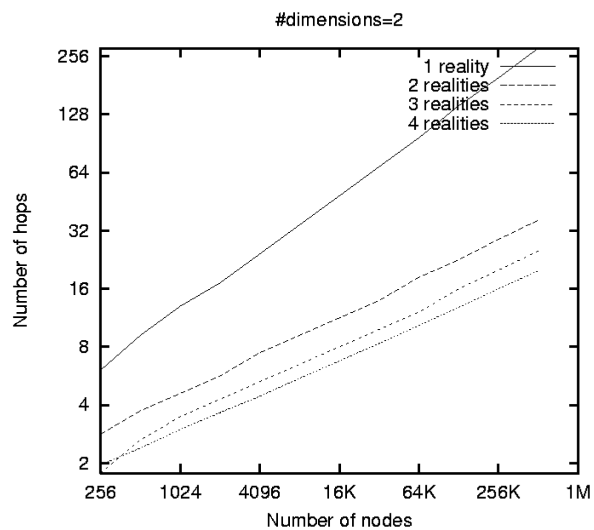
- more neighbors
- more routing possibilities
- more state information O(2D)

More coordinate systems (*r*)

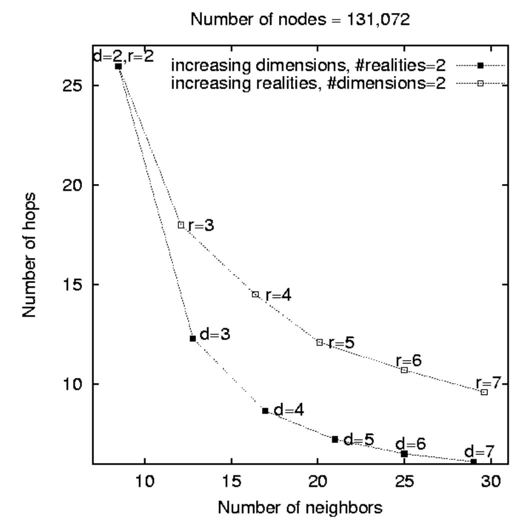
- *r* possibilities for routing
- state information *O(rD)*
- *r*-fold redundancy



Increase of dimensions



Increase of realities



Comparison

- Conclusion: more dimensions lead to shorter paths in the overlay (...but more coordinate systems increase the redundancy).

Other Improvements for CAN

Routing metrics

- measure the delay between neighbors
- choose the neighbors with the shortest delay

Overlapping regions

- k nodes jointly manage one area
- more redundancy
- faster routing paths because of less number of zones

Equal (uniform) partitioning of regions

- Target zone tests during the join procedure: are there “large” neighbors in the proximity, being more qualified for partitioning?

Conclusions for CAN

- CAN is a peer-to-peer system based on a DHT.
- It operates with D dimensions. The number of dimensions determines the efficiency.
- Access to a key in $O(\frac{D}{4} N^{\frac{1}{D}})$
- Efficient algorithms for joining and leaving nodes exist.
- Problem: N has to be known beforehand!