Peer-to-Peer Systems and Applications

Lecture 3: Distributed Hash Tables (2)

Chapter 8 and 9:

Part III: Structured
Peer-to-Peer Systems

*Original slides for this lecture provided by K. Wehrle, S. Götz, S. Rieche (University of Tübingen)

0. Lecture Overview

1. Selected DHT Algorithm: Chord
   1. Identifiers and Topology
   2. Routing
   3. Self-Organization: Failure Tolerance, Node Arrival
   4. Summary

2. Storage Load Balancing in Distributed Hash Tables
   1. Chord without Load Balancing
   2. Algorithms for Load Balancing in DHTs
      1. Power of Two Choices
      2. Virtual Servers
      3. Thermal-Dissipation-based Approach
      4. A Simple Address-Space and Item Balancing
   3. Comparison of Load-Balancing Approaches

3. Reliability in Distributed Hash Tables
   1. Redundancy vs. Replication
   2. Replication

TECHNISCHE UNIVERSITÄT DARMSTADT

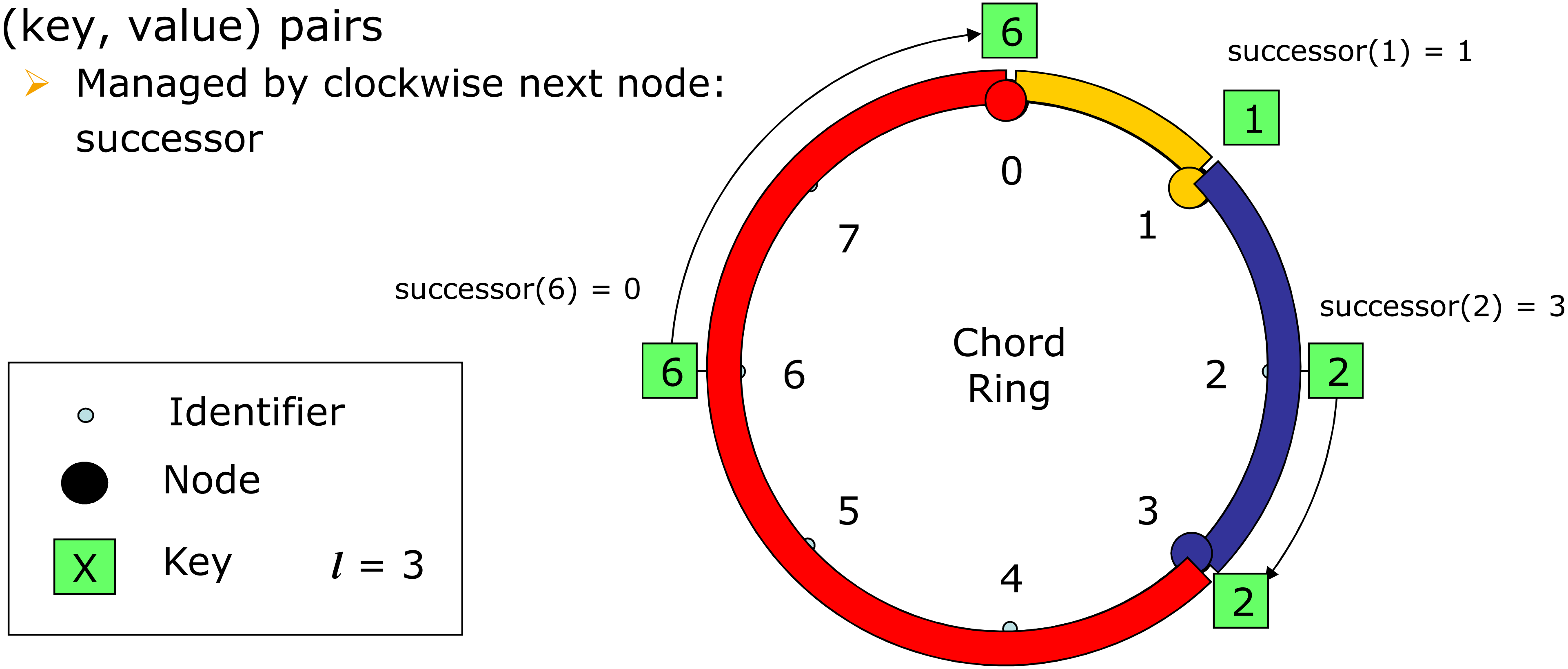
1. Selected DHT Algorithm: Chord

Topology, Routing, Self-Organization: Failure Tolerance, Node Arrival

1.1. Identifiers and Topology

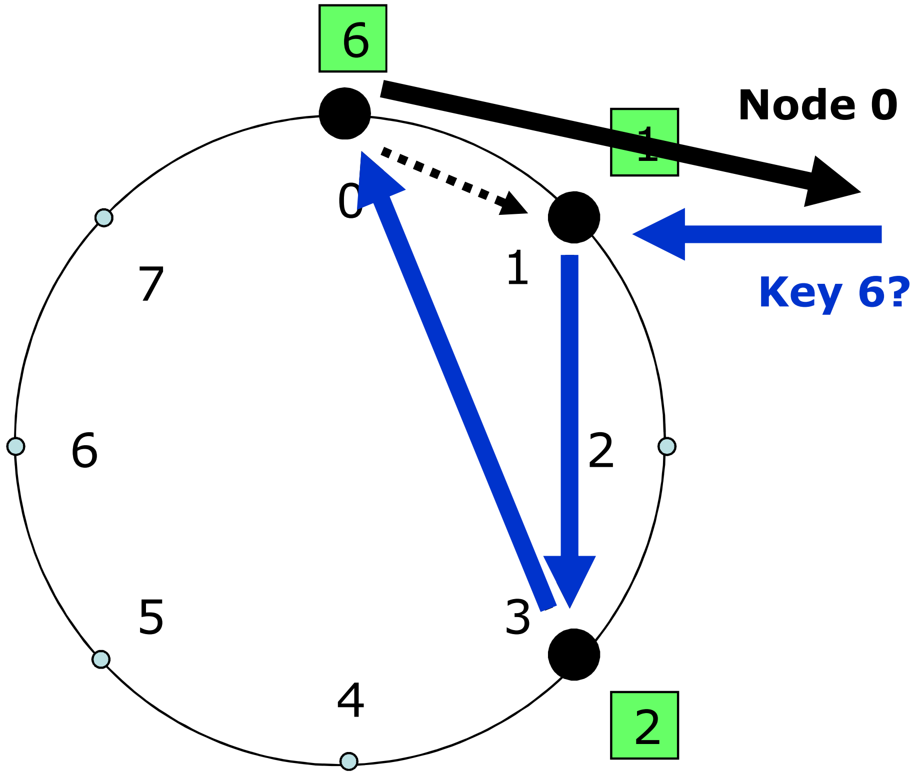
TECHNISCHE UNIVERSITÄT DARMSTADT

- Keys and Node IDs
  - l-bit identifiers, i.e. integers in range $0 \ldots 2^l - 1$
  - Derived from hash function (e.g. SHA-1)
- (key, value) pairs
  - Managed by clockwise next node: successor



1.2. Routing (1)

- Simplest topology: circular linked list
  - Each node has link to clockwise next node
- Primitive routing:
  - Forward query for key x until successor(x) is found
  - Return result to source of query
- Pros:
  - Simple
  - Little node state O(1)
- Cons:
  - Poor lookup efficiency:
    O(1/2 * N) hops on average
    (with N nodes)
  - Node failure breaks circle

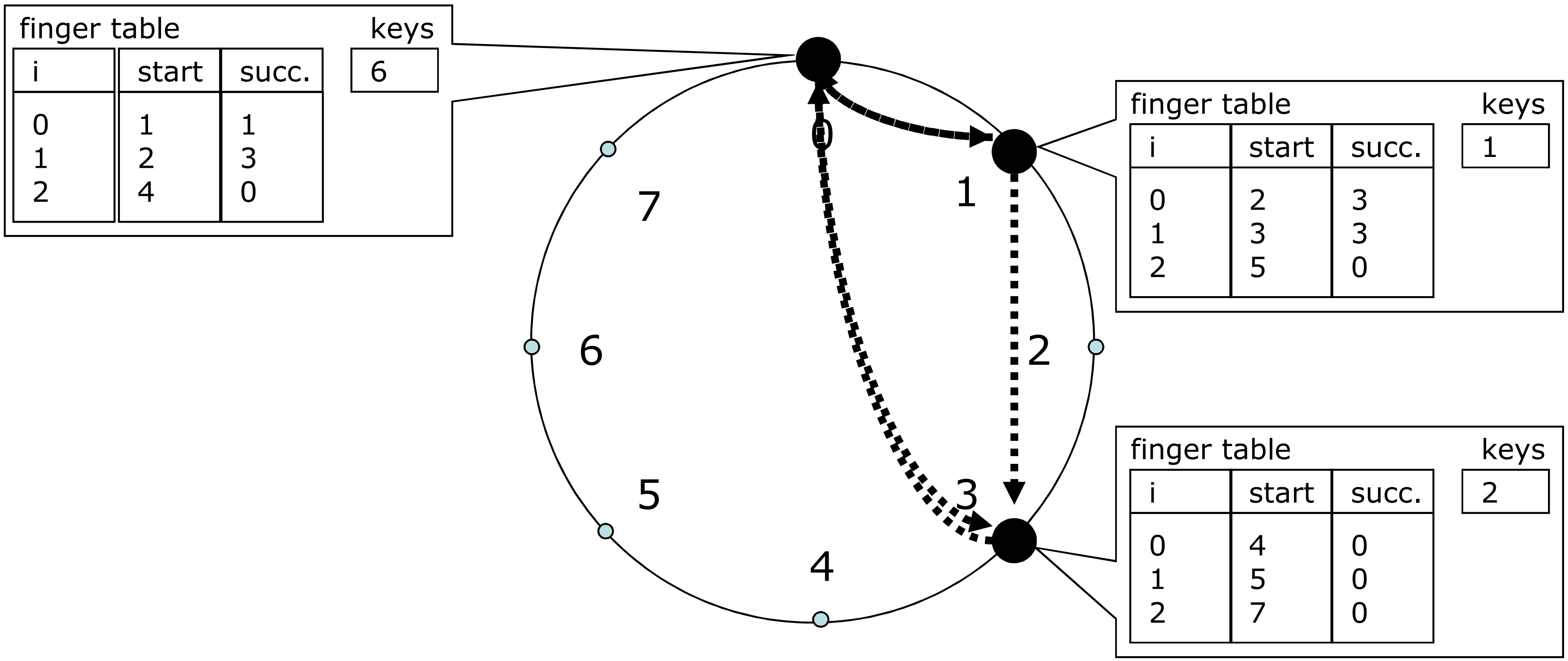


1.2. Routing (2)

- Advanced routing:
  - Store links to z next neighbors
  - Forward queries for k to farthest known predecessor of k
  - For z = N: fully meshed routing system
    - Lookup efficiency: O(1)
    - Per-node state: O(N)
  - Still poor scalability
- Scalable routing:
  - Linear routing progress scales poorly
  - Mix of short- and long-distance links required:
    - Accurate routing in node’s vicinity
    - Fast routing progress over large distances
    - Bounded number of links per node

1.2. Routing (3)

- Chord's routing table: *finger table*
  - Stores log(N) links per node
  - Covers exponentially increasing distances:
    - Node n: entry i points to successor(n + 2^i) (*i-th finger*)

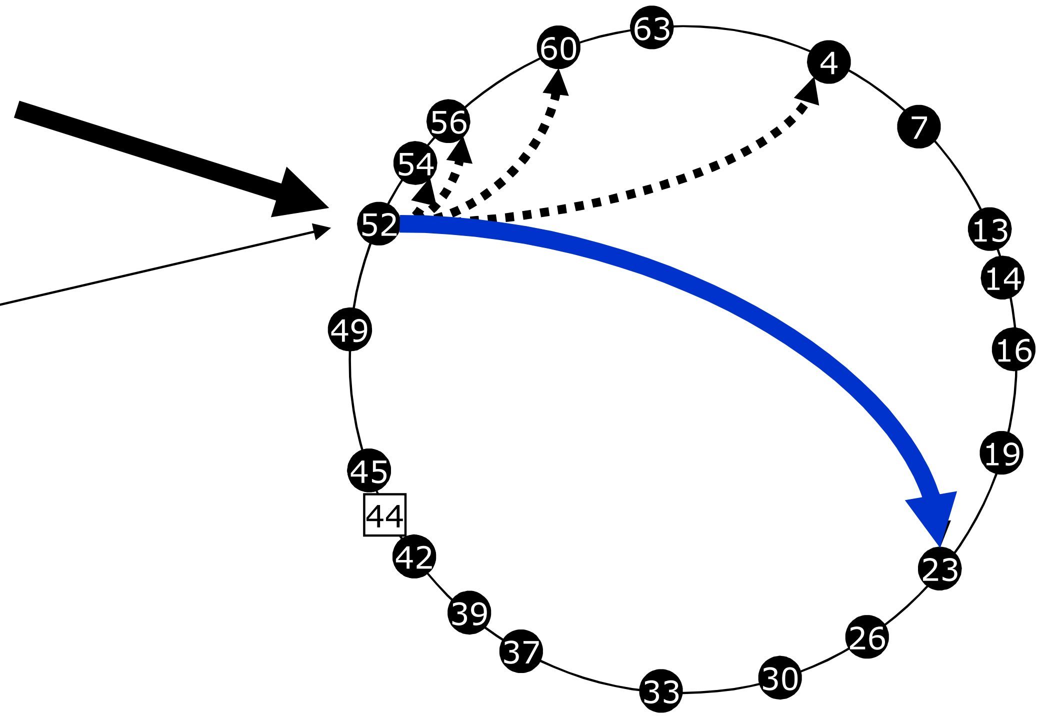


1.2. Routing (4)

- Chord's routing algorithm:
  - Each node n forwards query for key k clockwise
    - To farthest finger preceding k
    - Until n = predecessor(k) and successor(n) = successor(k)
    - Return successor(n) to source of query

lookup (44)

i	2^i	Target	Link
0	1	53	54
1	2	54	54
2	4	56	56
3	8	60	60
4	16	4	4
5	32	20	23

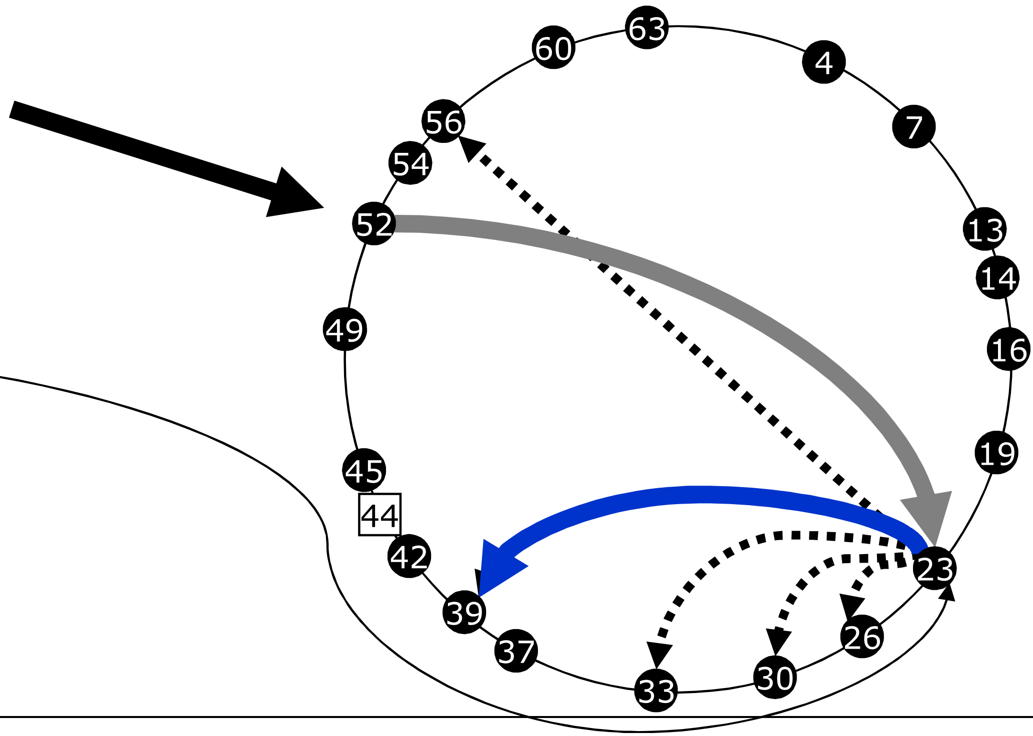


1.2. Routing (4)

- Chord’s routing algorithm:
  - Each node n forwards query for key k clockwise
    - To farthest finger preceding k
    - Until n = predecessor(k) and successor(n) = successor(k)
    - Return successor(n) to source of query

lookup (44)

i	2^i	Target	Link
0	1	24	26
1	2	25	26
2	4	27	30
3	8	31	33
4	16	39	39
5	32	55	56



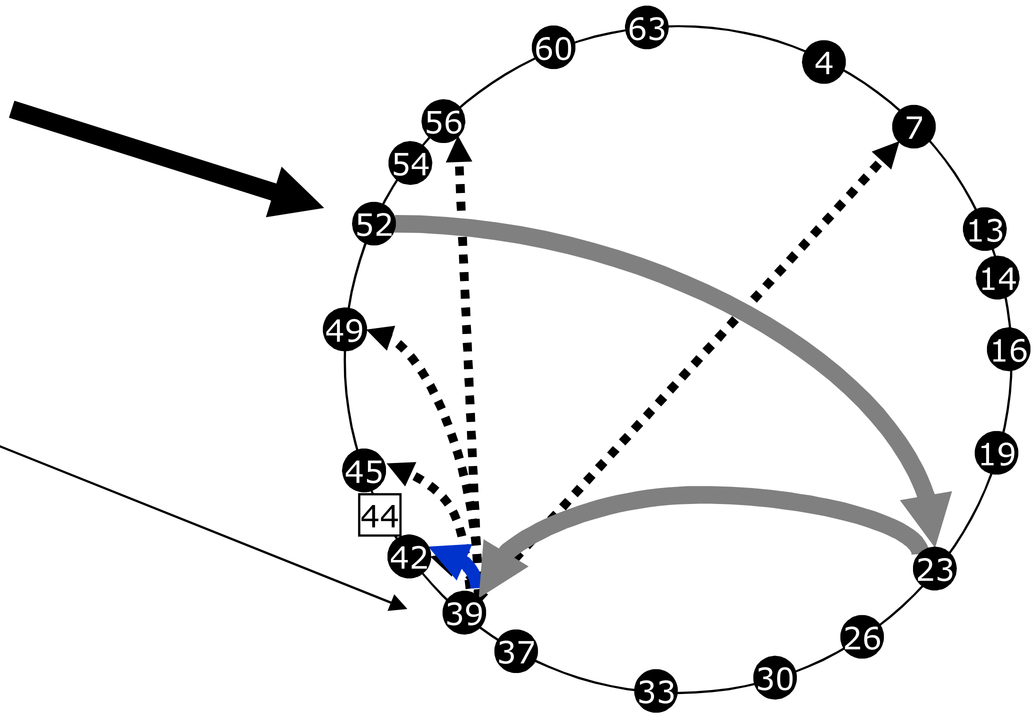
1.2. Routing (4)

TECHNISCHE UNIVERSITÄT DARMSTADT

- Chord's routing algorithm:
  - Each node n forwards query for key k clockwise
    - To farthest finger preceding k
    - Until n = predecessor(k) and successor(n) = successor(k)
    - Return successor(n) to source of query

lookup (44)

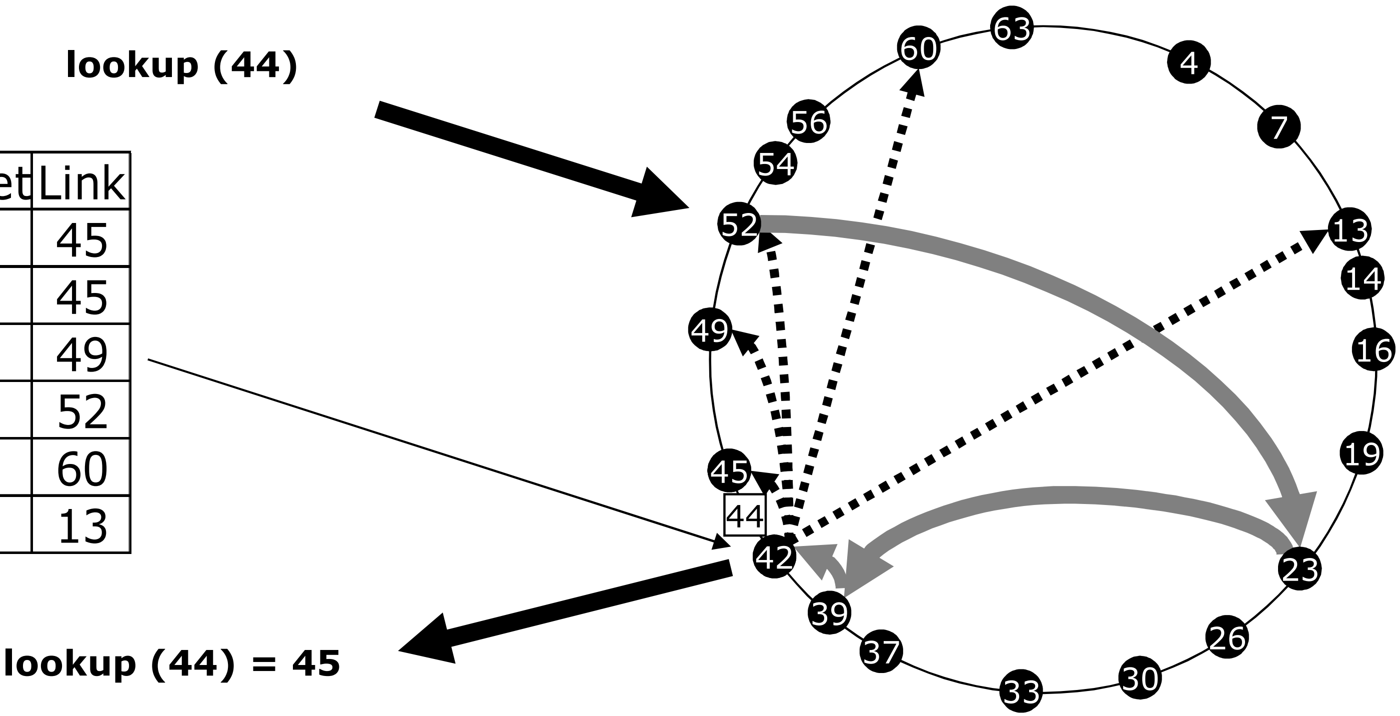
i	2^i	Target	Link
0	1	40	42
1	2	41	42
2	4	43	45
3	8	47	49
4	16	55	56
5	32	7	7



1.2. Routing (4)

- Chord's routing algorithm:
  - Each node n forwards query for key k clockwise
    - To farthest finger preceding k
    - Until n = predecessor(k) and successor(n) = successor(k)
    - Return successor(n) to source of query

i	2^i	Target	Link
0	1	43	45
1	2	44	45
2	4	46	49
3	8	50	52
4	16	58	60
5	32	10	13



1.3. Self-Organization

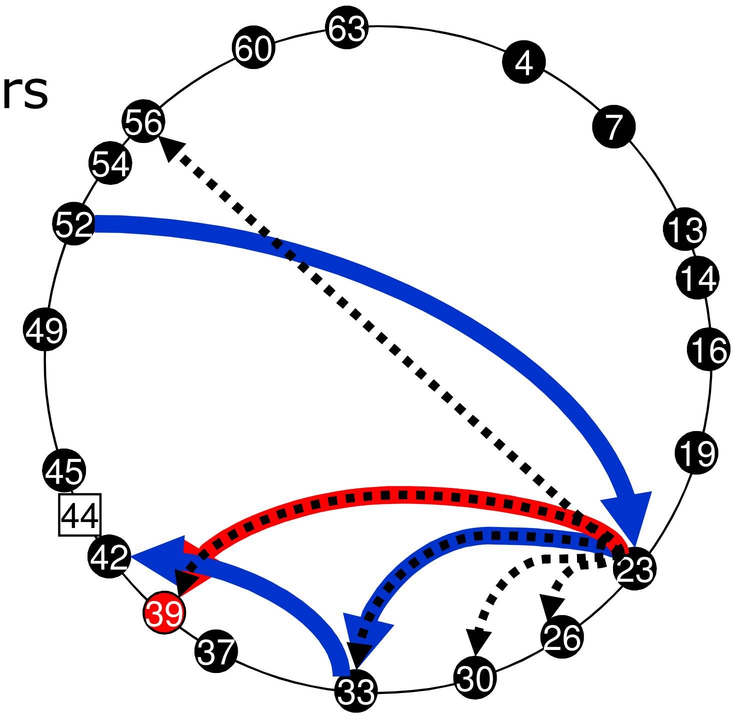
TECHNISCHE UNIVERSITÄT DARMSTADT

- Handle changing network environment
  - Failure of nodes
  - Network failures
  - Arrival of new nodes
  - Departure of participating nodes
- Maintain consistent system state for routing
  - Keep routing information up to date
    - Routing correctness depends on correct successor information
    - Routing efficiency depends on correct finger tables
  - Failure tolerance required for all operations

1.3. Failure Tolerance: Routing (1)

TECHNISCHE UNIVERSITÄT DARMSTADT

- Finger failures during routing
  - Query cannot be forwarded to finger
  - Forward to previous finger (do not overshoot destination node)
  - Trigger repair mechanism: replace finger with its successor
- Active finger maintenance
  - Periodically check liveliness of fingers
  - Replace with correct nodes on failures
  - Trade-off: maintenance traffic vs. correctness & timeliness



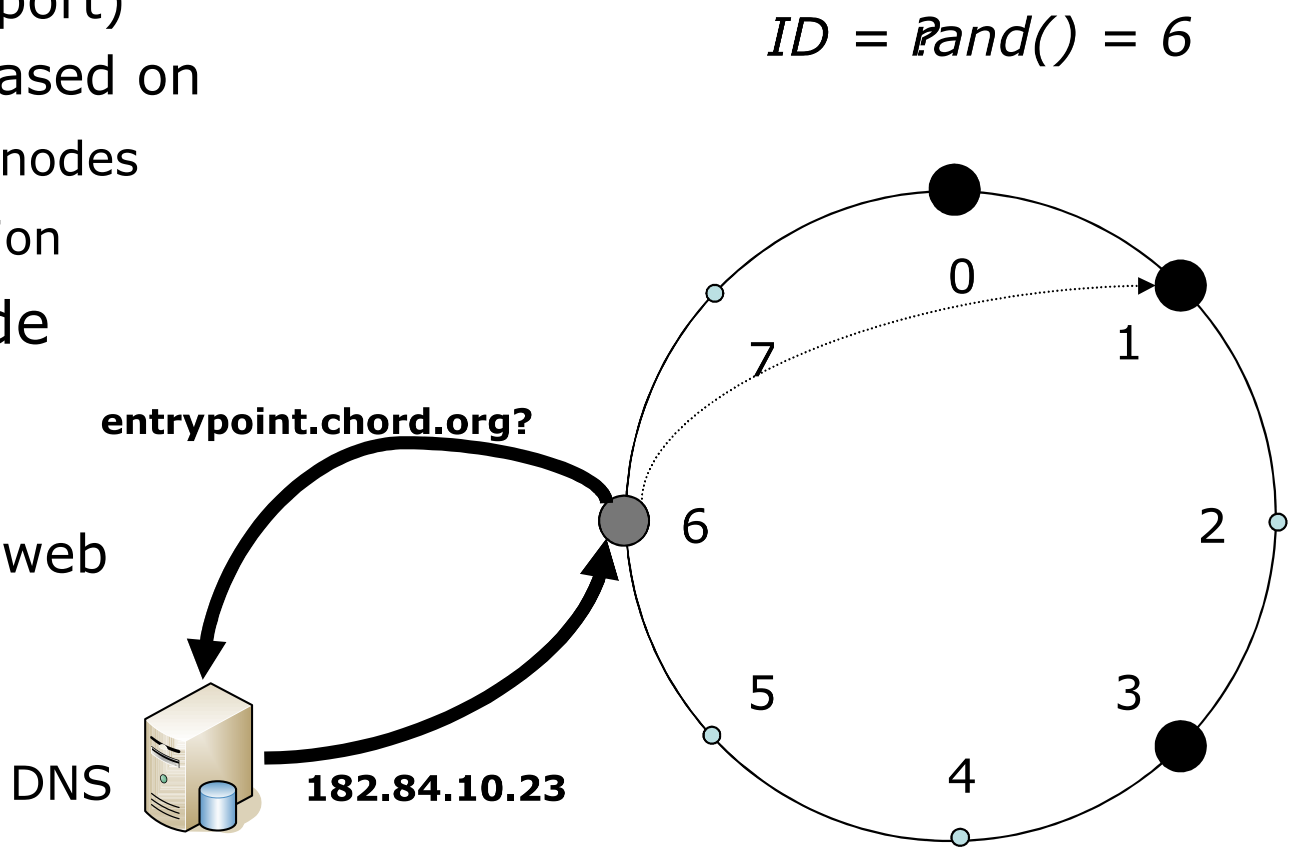
1.3. Failure Tolerance: Routing (2)

TECHNISCHE UNIVERSITÄT DARMSTADT

- Successor failure during routing
  - Last step of routing can return failed node to source of query
    -> All queries for successor fail
  - Store n successors in *successor list*
    - Successor[0] fails -> use successor[1] etc.
    - Routing fails only if n consecutive nodes fail simultaneously
- Active maintenance of successor list
  - Periodic checks similar to finger table maintenance
  - Crucial for correct routing

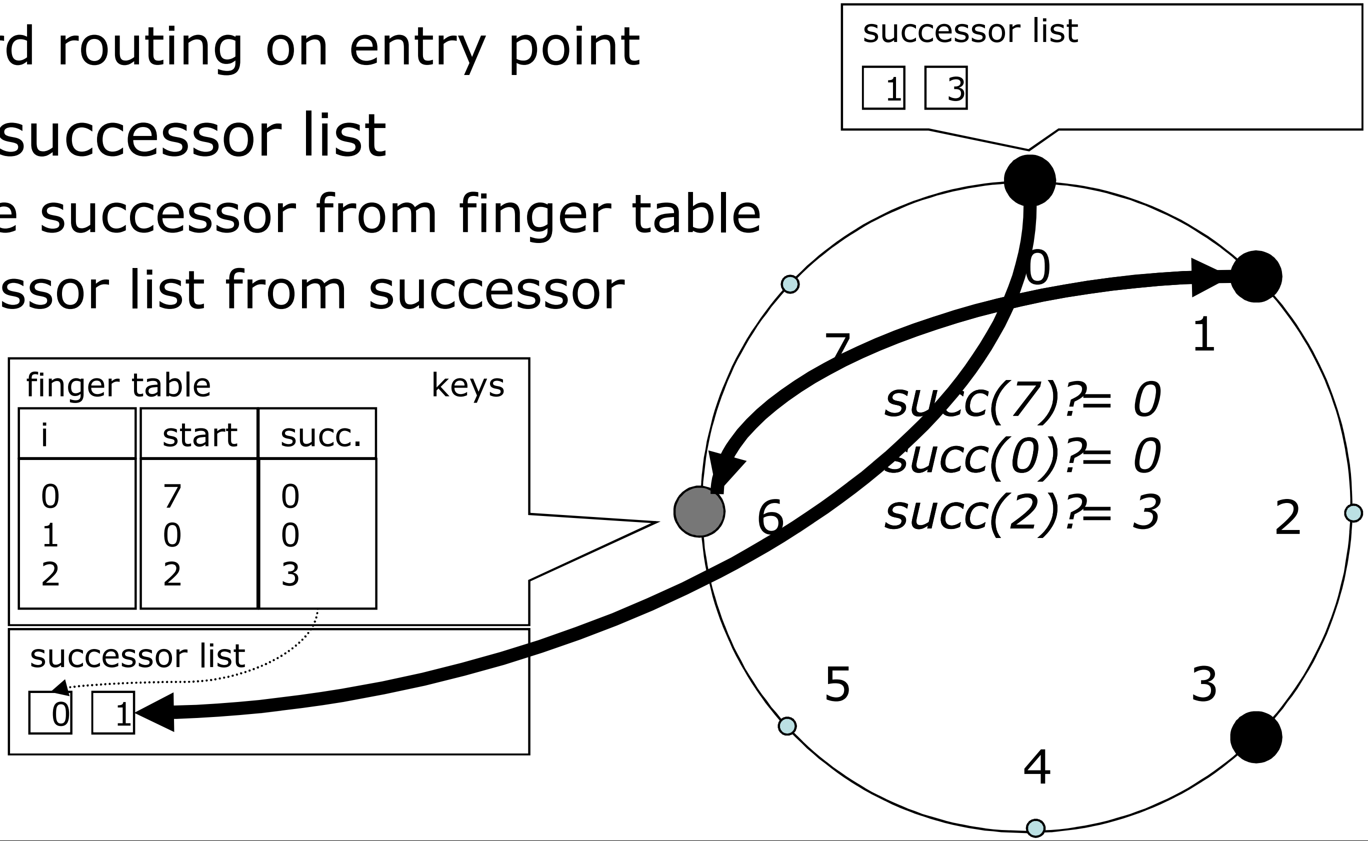
1.3. Node Arrival (1)

- New node picks ID
  - Random ID: equal distribution assumed but not guaranteed
  - Hash (IP address, port)
  - Place new nodes based on
    - Load on existing nodes
    - Geographic location
- Contact existing node
  - Controlled flooding
  - DNS aliases
  - Published through web



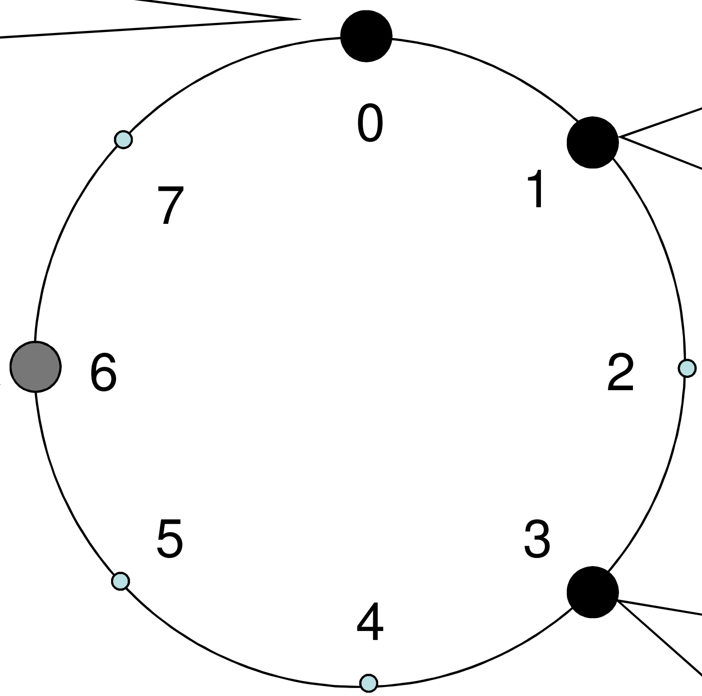
1.3. Node Arrival (2)

- Construction of finger table
  - Iterate over finger table rows
  - For each row: query entry point for successor
  - Standard Chord routing on entry point
- Construction of successor list
  - Add immediate successor from finger table
  - Request successor list from successor



1.3. Node Arrival (3)

- Retrieve (key, value) pairs from successor



Node 0 — finger table (keys: 6)

i	start	succ.
0	1	1
1	2	3
2	4	0

Node 1 — finger table (keys: 1)

i	start	succ.
0	2	3
1	3	3
2	5	0

Node 3 — finger table (keys: 2)

i	start	succ.
0	4	0
1	5	0
2	7	0

Node 6 — finger table (keys:)

i	start	succ.
0	7	0
1	0	0
2	2	3

1.4. Summary

TECHNISCHE UNIVERSITÄT DARMSTADT

- 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 about complexity
  - Simple & flexible
- Disadvantages
  - No notion of node proximity and proximity-based routing optimizations
  - Chord rings may become disjoint in realistic settings
- Many improvements published
  - E.g. proximity, bi-directional links, load balancing, etc.

2. Storage Load Balancing in Distributed Hash Tables

Chord without Load Balancing, Algorithms for Load Balancing in DHTs, Comparison

2. Storage Load Balancing in DHTs

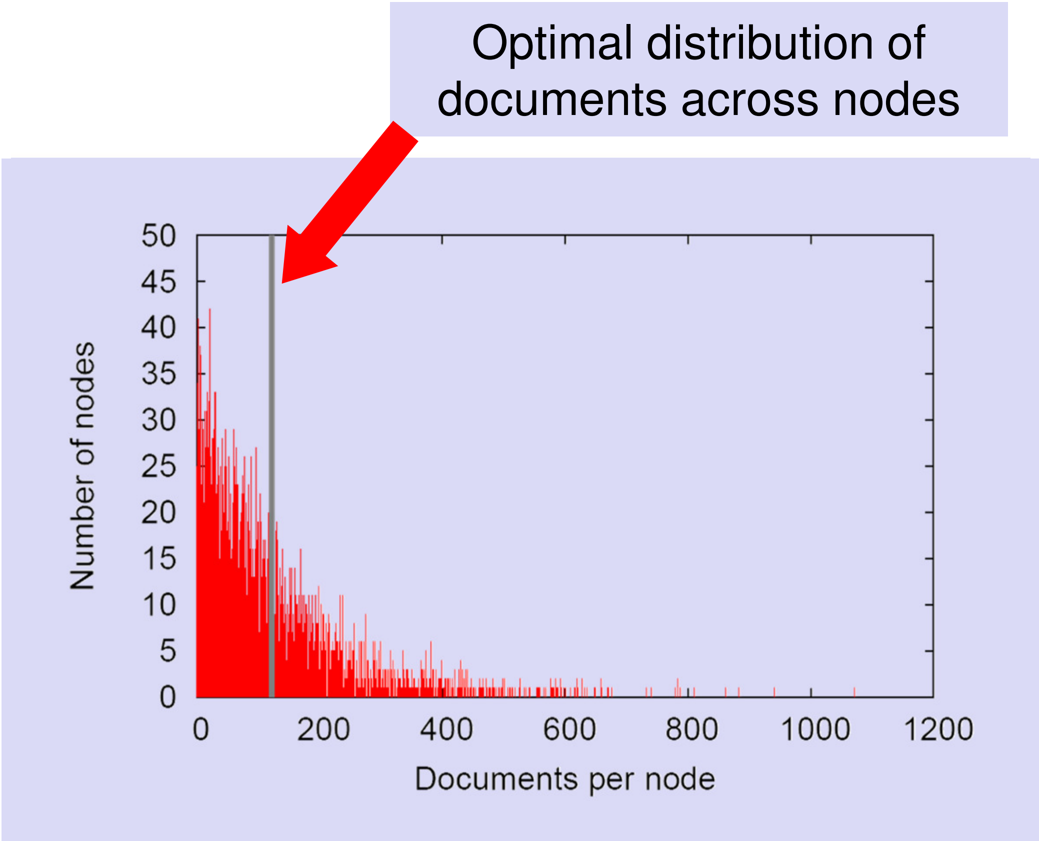
- Standard assumption: uniform key distribution
  - Hash function
  - Every node with equal load
  - No load balancing is needed

- Equal distribution
  - Nodes across address space
  - Data across nodes

- But is this assumption justifiable?
  - Analysis of distribution of data using simulation

2.1. Chord without Load Balancing (1)

- Analysis of distribution of data
- Example
  - Parameters
    - 4,096 nodes
    - 500,000 documents
  - Optimum
    - ~122 documents per node

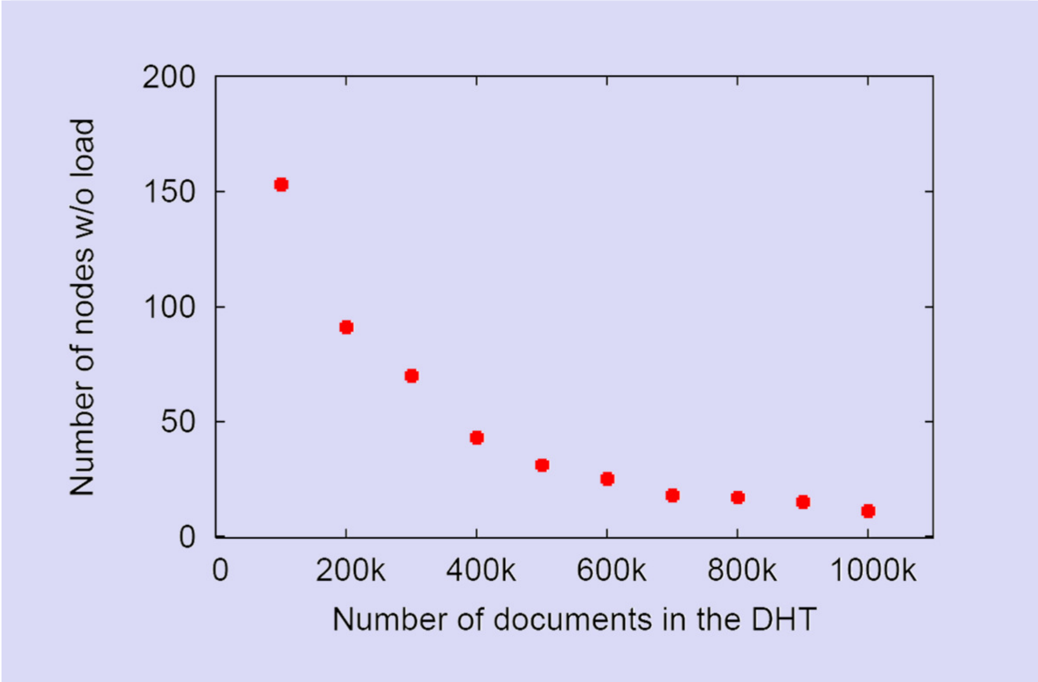


→ No optimal distribution in Chord w/o load balancing

2.1. Chord without Load Balancing (2)

TECHNISCHE UNIVERSITÄT DARMSTADT

- Number of nodes without storing any document
  - Parameters
    - 4,096 nodes
    - 100,000 to 1,000,000 documents
  - Some nodes w/o any load



→ Load Balancing is needed to keep the complexity of DHT management low

2.2. Load Balancing Algorithms

TECHNISCHE UNIVERSITÄT DARMSTADT

- Problem
  - Significant difference in the load of nodes

- Several techniques to ensure an equal data distribution
  - Power of Two Choices (Byers et. al, 2003)
  - Virtual Servers (Rao et. al, 2003)
  - Thermal-Dissipation-based Approach (Rieche et. al, 2004)
  - A Simple Address-Space and Item Balancing (Karger et. al, 2004)

2.2.1. Power of Two Choices (1)

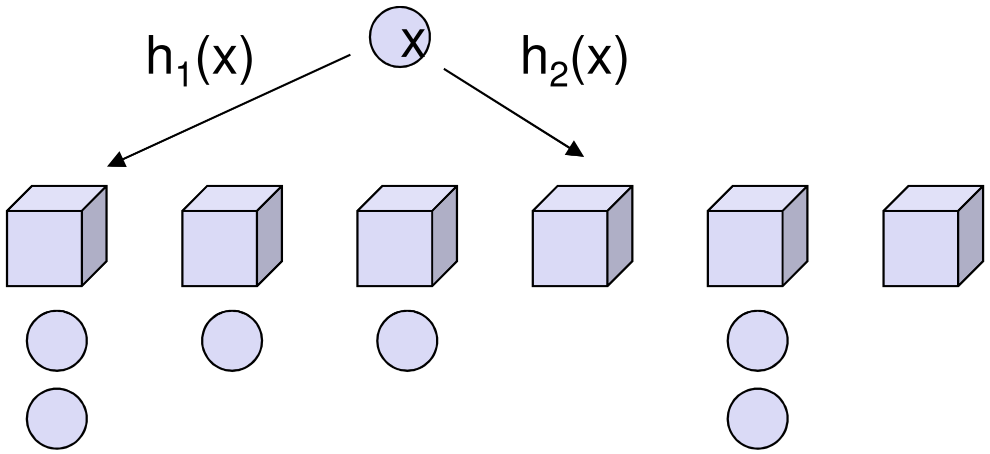
- John Byers, Jeffrey Considine, and Michael Mitzenmacher (2003)
  - "Simple Load Balancing for Distributed Hash Tables"

- Idea
  - One hash function for all nodes
    - h_0
  - Multiple hash functions for data
    - $h_1, h_2, h_3, \ldots h_d$

- Two options
  - Data is stored at one node
  - Data is stored at one node & other nodes store a pointer

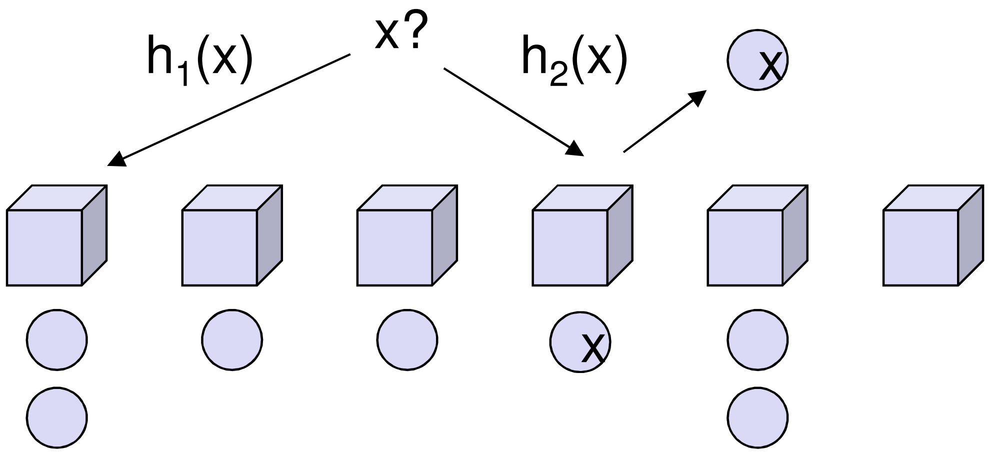
2.2.1. Power of Two Choices (2)

- Inserting Data
  - Results of all hash functions are calculated
    - $h_1(x)$, $h_2(x)$, $h_3(x)$, …$h_d(x)$
  - Data is stored on the retrieved node with the lowest load
  - Alternative
    - Other nodes stores pointer



2.2.1. Power of Two Choices (3)

- Retrieving
  - Without pointers
    - Results of all hash functions are calculated
    - Request all of the possible nodes in parallel
    - One node will answer
  - With pointers
    - Request only one of the possible nodes.
    - Node can forward the request directly to the final node



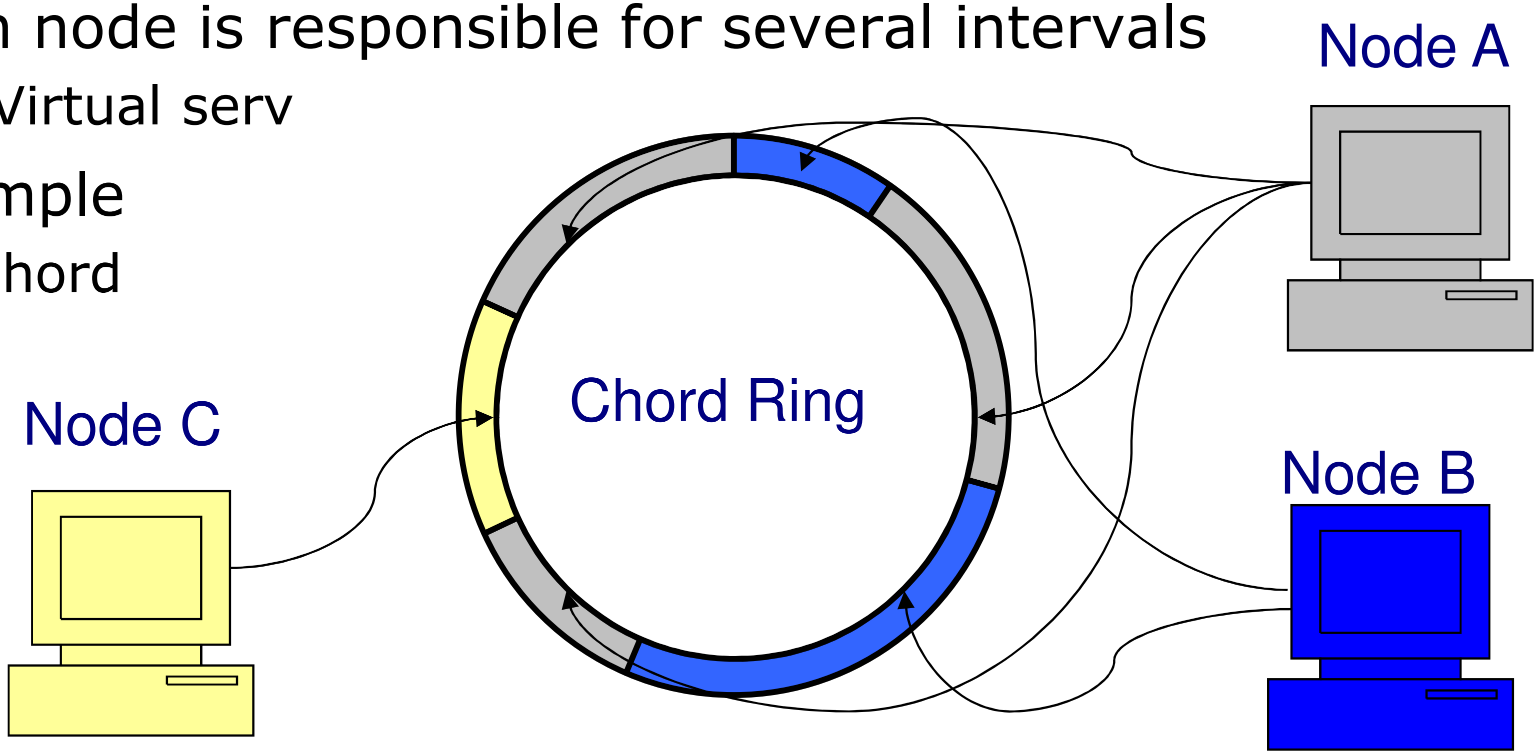
2.2.1. Power of Two Choices (4)

- Advantages
  - Simple

- Disadvantages
  - Message overhead at inserting data
  - With pointers
    - Additional administration of pointers
      - More load
  - Without pointers
    - Message overhead at every search

2.2.2. Virtual Server (1)

- Ananth Rao, Karthik Lakshminarayanan, Sonesh Surana, Richard Karp, and Ion Stoica (2003)
  - "Load Balancing in Structured P2P Systems"
- Each node is responsible for several intervals
  - "Virtual serv
- Example
  - Chord



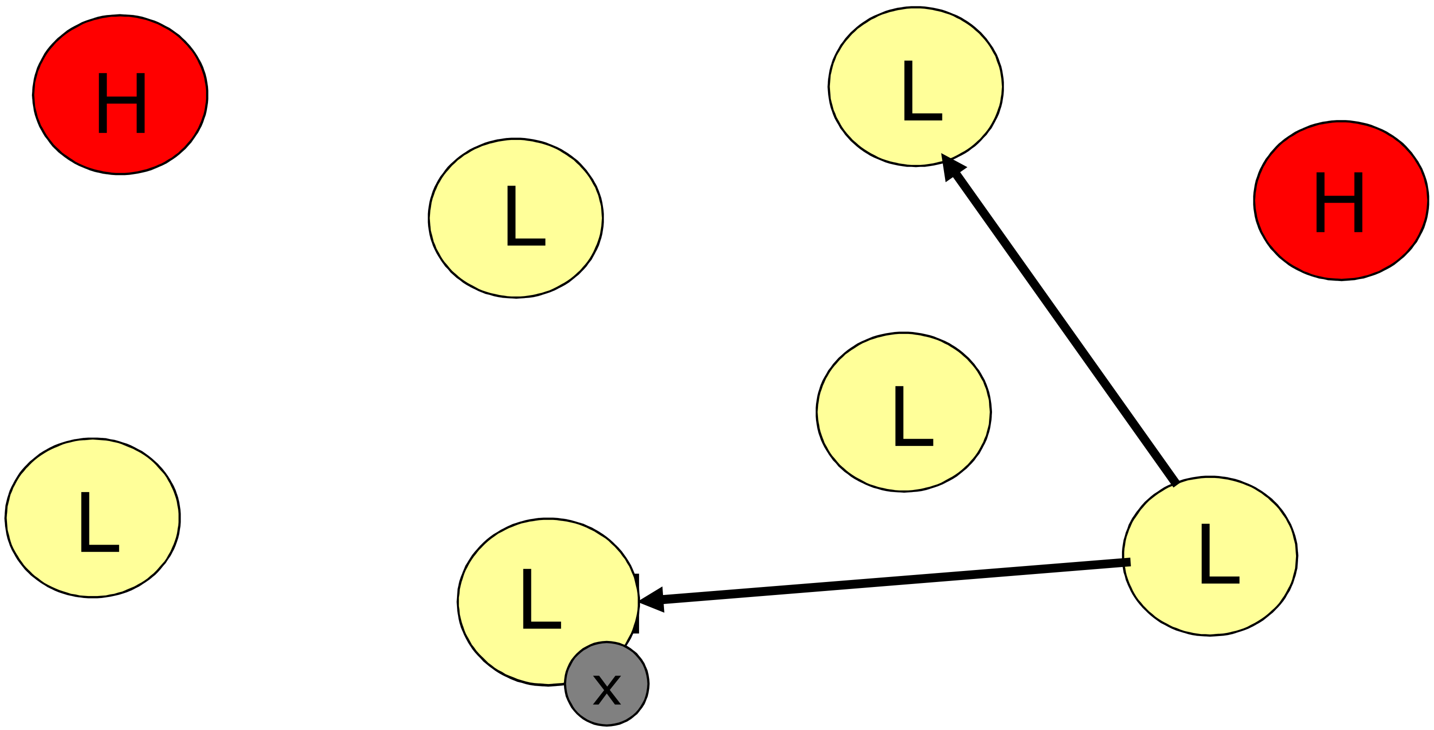
2.2.2. Virtual Server (2)

- Rules for transferring a virtual server
  - From heavy node to light node
  1. The transfer of an virtual server makes the receiving node not heavy
  2. The virtual server is the lightest that makes the heavy node light
  3. If there is no virtual server whose transfer can make a node light, the heaviest virtual server from this node would be transferred

- Different possibilities to transfer virtual servers
  - One-to-one
  - One-to-many
  - Many-to-many

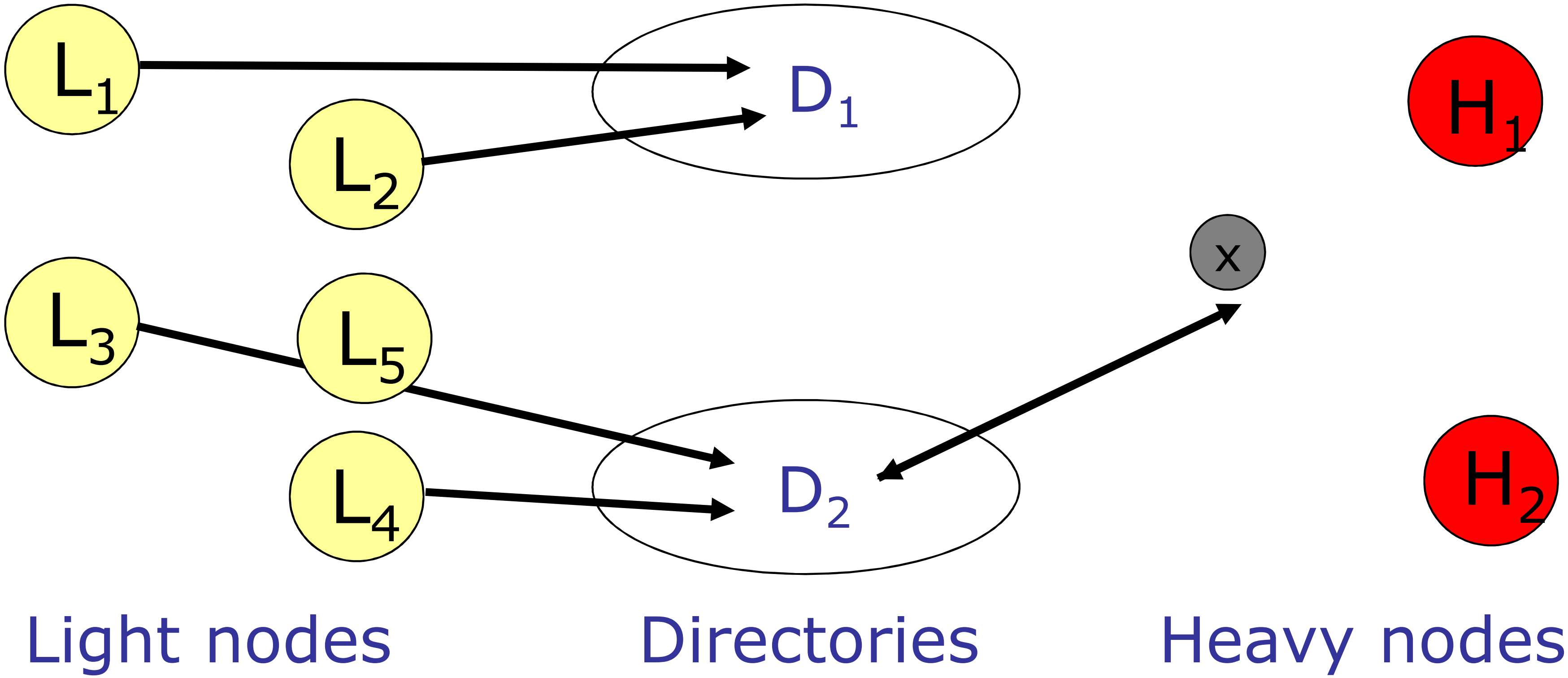
2.2.2. Scheme 1: One-to-One

- One-to-One
  - Light node picks a random ID
  - Contacts the node x responsible for it
  - Accepts load if x is heavy



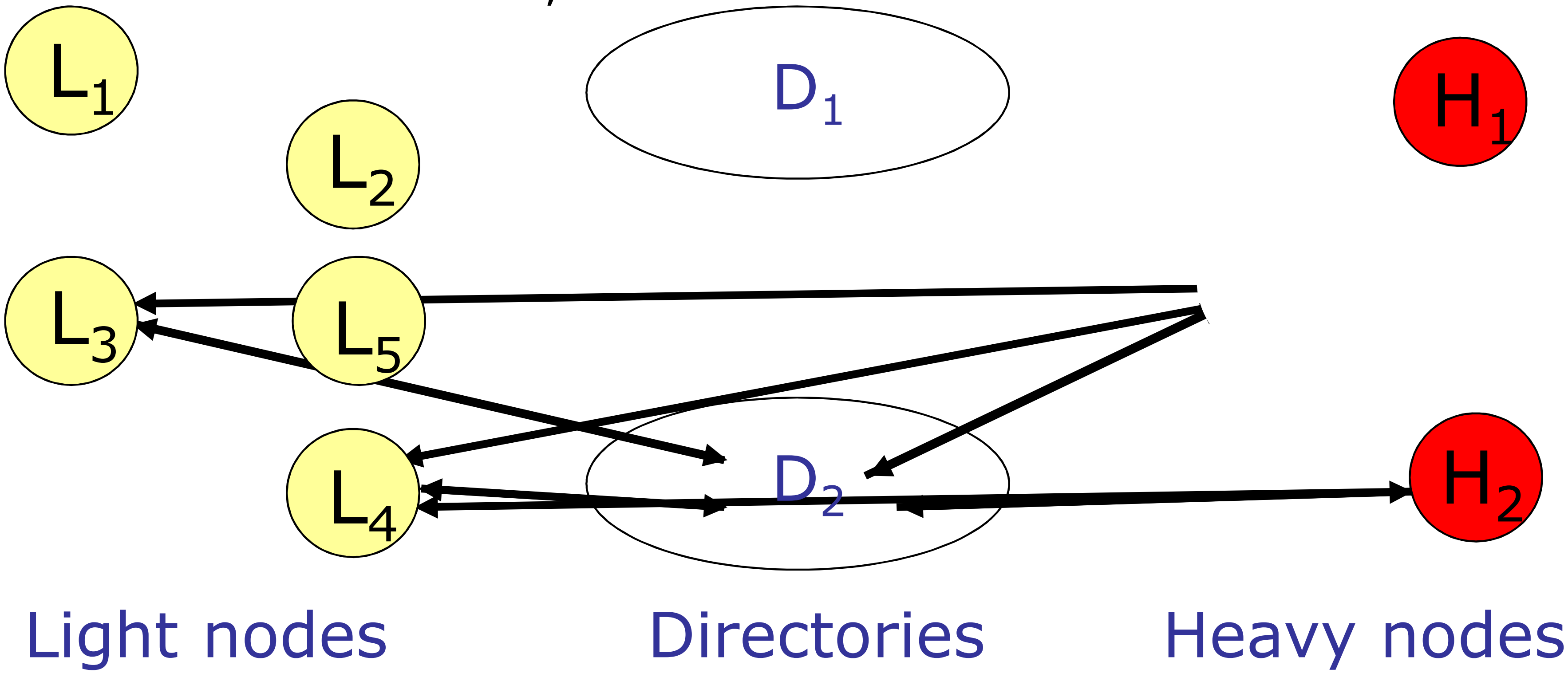
2.2.2. Scheme 2: One-to-Many

- One-to-Many
  - Light nodes report their load information to directories
  - Heavy node H gets this information by contacting a directory
  - H contacts the light node which can accept the excess load



2.2.2. Scheme 3: Many-to-Many

- Many-to-Many
  - Many heavy and light nodes rendezvous at each step
  - Directories periodically compute the transfer schedule and report it back to the nodes, which then do the actual transfer

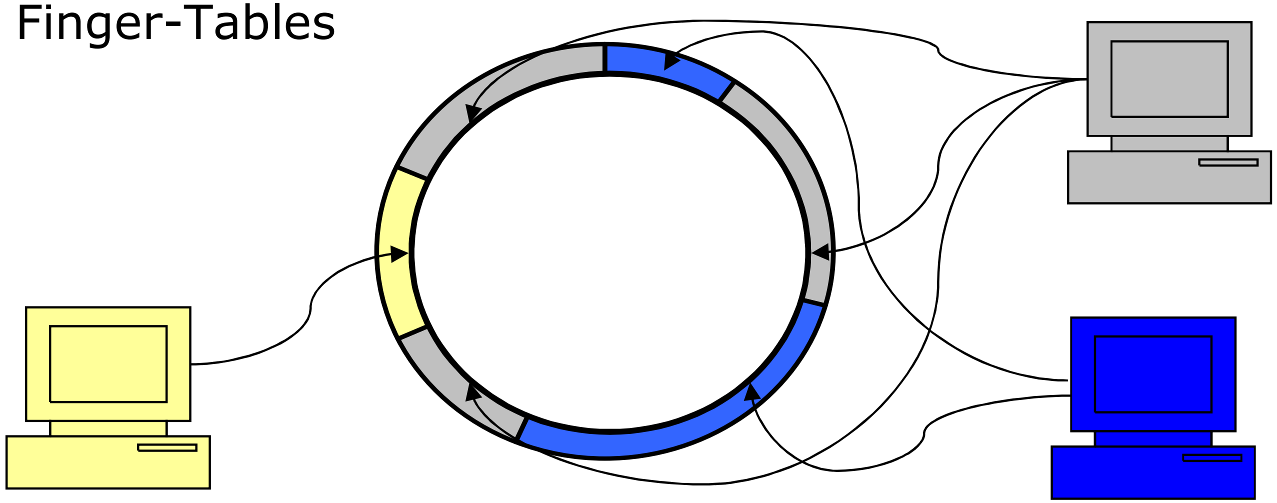


2.2.2. Virtual Server

TECHNISCHE UNIVERSITÄT DARMSTADT

- Advantages
  - Easy shifting of load
    - Whole Virtual Servers are shifted

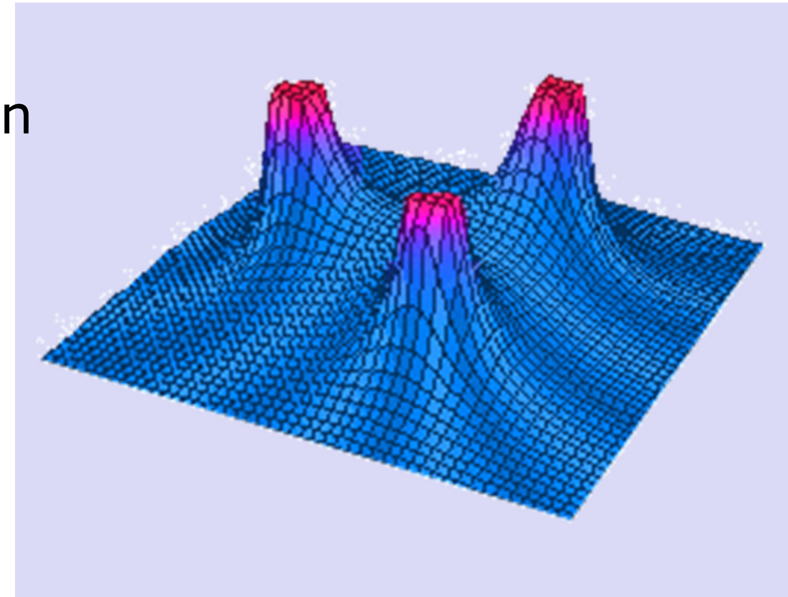
- Disadvantages
  - Increased administrative and messages overhead
    - Maintenance of all Finger-Tables
  - Much load is shifted



2.2.3. Thermal-Dissipation-based Approach (1)

TECHNISCHE UNIVERSITÄT DARMSTADT

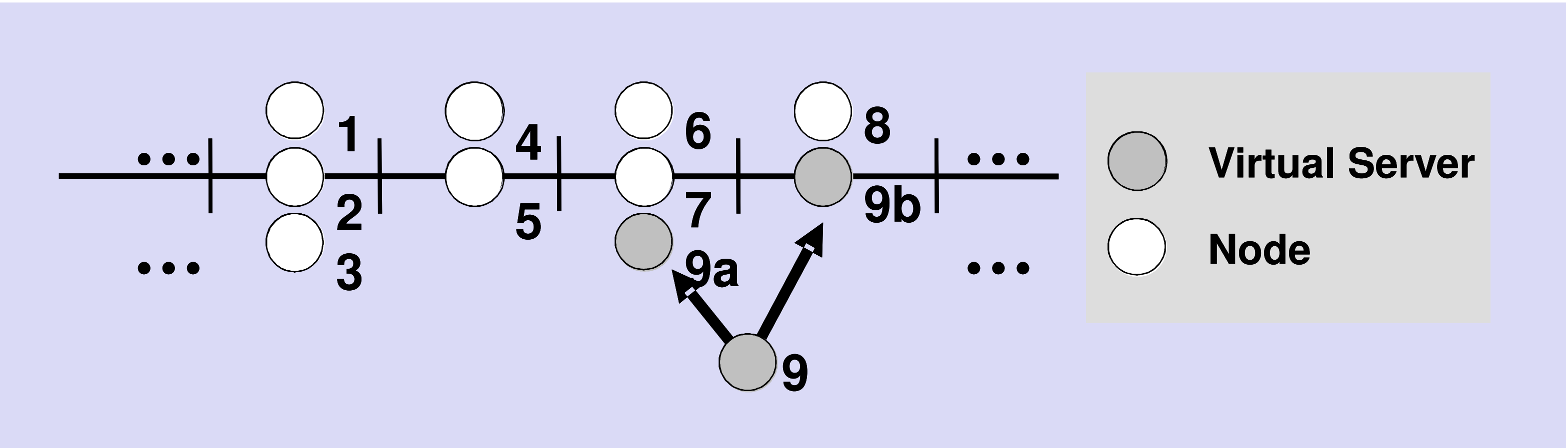
- Simon Rieche, Leo Petrak, and Klaus Wehrle (2004)
  - "A Thermal-Dissipation-based Approach for Balancing Data Load in Distributed Hash Tables"
- Content is moved among peers
  - Similar to the process of heat expansion
  - Several nodes in one interval
    - DHT more fault tolerant



- Fixed positive number f
  - Indicates how many nodes have to act within one interval at least.

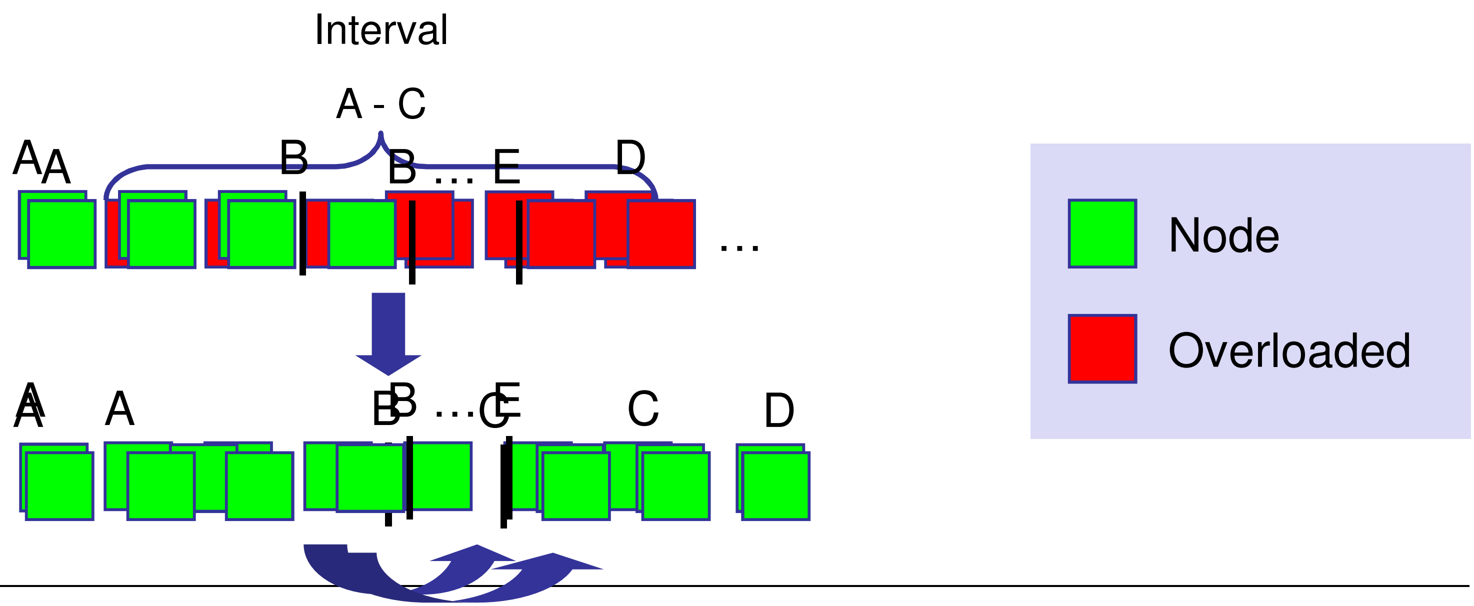
2.2.3. Thermal-Dissipation-based Approach (2)

- Procedure
  - First node takes random position
  - A new node is assigned to any existing node
  - Node is announced to all other nodes in same interval
  - Copy of documents of interval
    - More fault tolerant system



2.2.3. Algorithm

- Nodes can balance the load with other intervals
  - Three various methods
    1. 2f different nodes in same interval and nodes are overloaded
       - Interval is divided
    2. More than f but less than 2f nodes
       - Release some nodes to other intervals
    3. Interval borders may be shifted between neighbors



2.2.4. Address-Space Balancing (1)

TECHNISCHE UNIVERSITÄT DARMSTADT

- David Karger, and Matthias Ruhl (2004)
  - „Simple, Efficient load balancing algorithms for peer-to-peer systems"

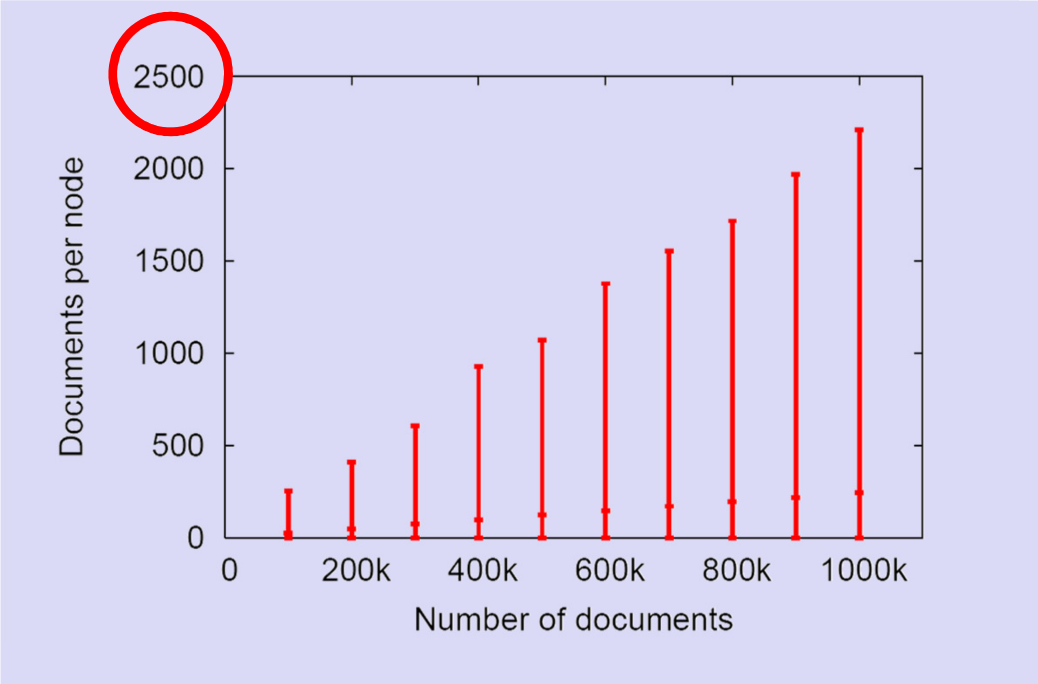
- Each node
  - Has a fixed set of $O(\log n)$ possible positions
    - "virtual nodes"
  - Chooses exactly one of those virtual nodes
    - *this position become active*
    - This is the only position that it actually operates

2.2.4. Address-Space Balancing (2)

- Node's set of virtual nodes depends only on the node itself
  - Computed as hashes
    - $h(id,1), h(id,2), \ldots, h(id, \log n)$
- Each (possibly inactive) virtual node "spans" a certain range of addresses
  - Between itself and its succeeding active virtual node
- Each real node has activated the virtual node
  - Which spans the minimal possible address

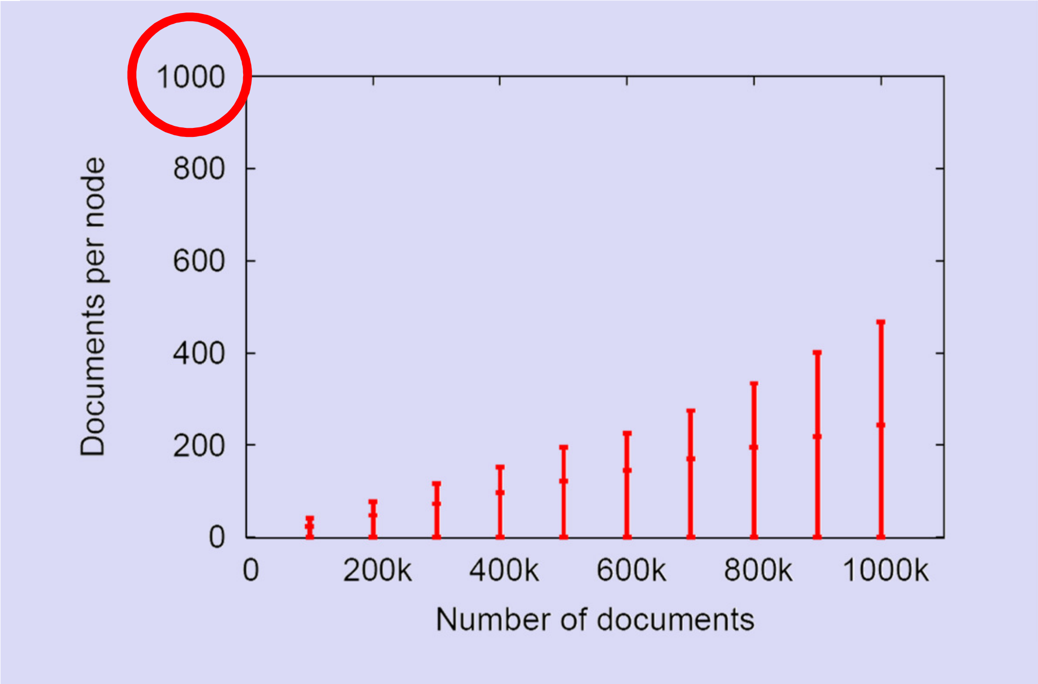
2.3. Comparison (1)

- Without load balancing



+ Simple
+ Original
− Bad load balancing

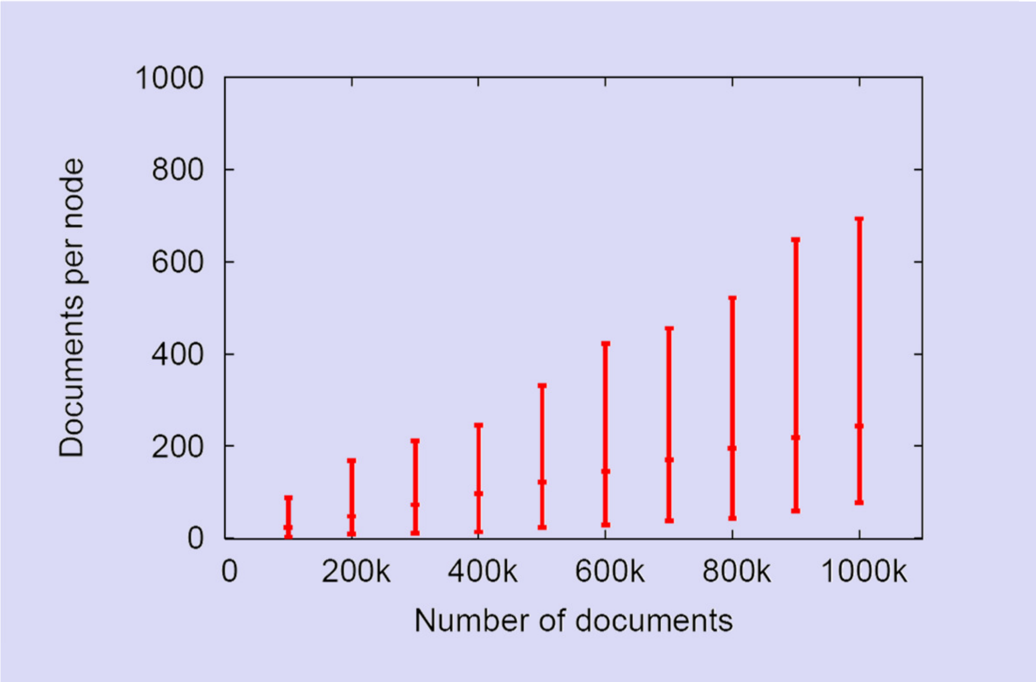
- Power of Two Choices



+ Simple
+ Lower load
− Nodes w/o load

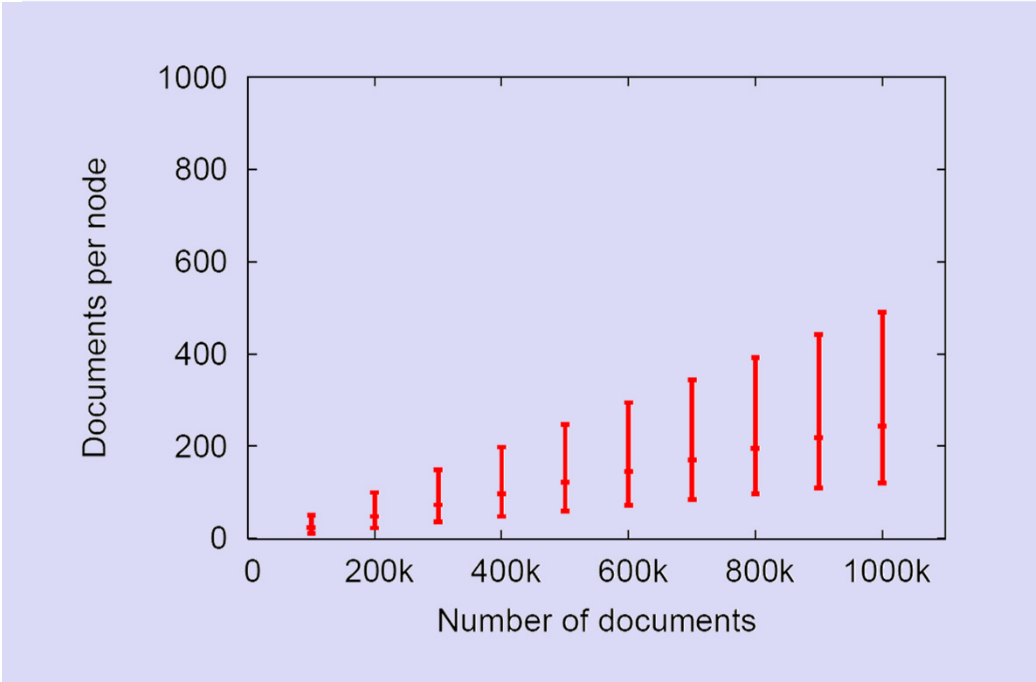
2.3. Comparison (2)

- Virtual server



+ No nodes w/o load
– Higher max. load than Power of Two Choices

- Thermal-Dissipation



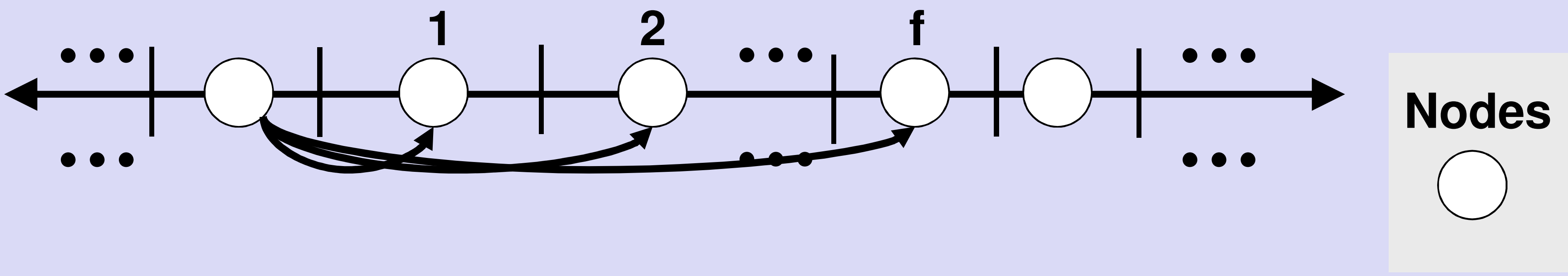
+ No nodes w/o load
+ Best load balancing
– More effort (but redund.)

3. Reliability in Distributed Hash Tables

Redundancy, Replication

3. Reliability in Distributed Hash Tables

- Chord
  - Problems
    - Unreliable nodes
    - →Inconsistent connections
    - →Lost of data
  - Successor-List
    - Stored by every node
    - f nearest successors clockwise on the ring



3.1. Redundancy vs. Replication

- Redundancy
  - Each data item is split into M fragments
    - K redundant fragments computed
      - Use of an "erasure-code"
    - Any M fragments allow to reconstruct the original data
  - For each fragment we compute its key
    - M + K different fragments have different keys
- Replication
  - Each data item is replicated K times
  - K replicas are stored on different nodes

3.2. “Stabilize” Function

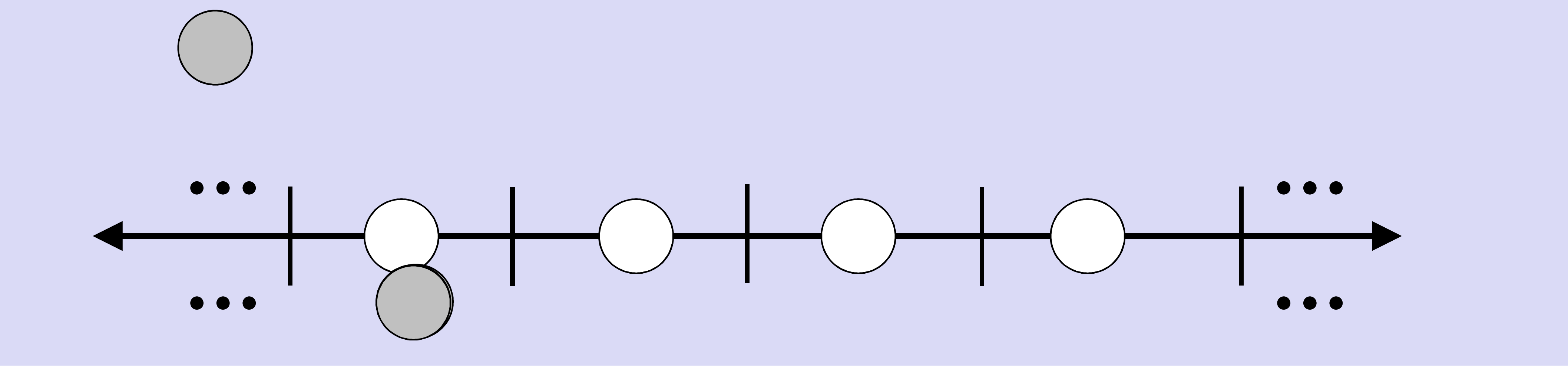
- Stabilize Function to correct inconsistent connections

- Procedure
  - Periodically done by each node n
  - n asks its successor for its predecessor p
  - n checks if p equals n
  - n also periodically refreshes random finger x
    - by (re)locating successor

- Successor-List to find new successor
  - If successor is not reachable use next node in successor-list
  - Start stabilize function

3.2. Reliability of Data in Chord

- Original
  - No Reliability of data
- Recommendation
  - Use of Successor-List
  - The reliability of data is an application task
  - Replicate inserted data to the next f other nodes
  - Chord inform application of arriving or failing nodes

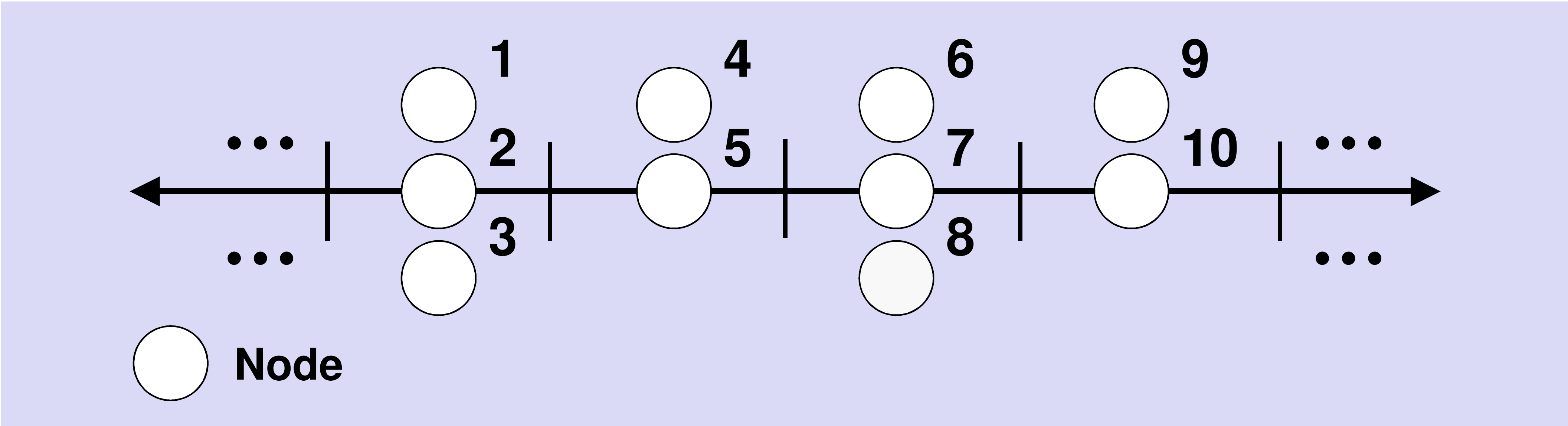


3.2. Properties

- Advantages
  - After failure of a node its successor has the data already stored
- Disadvantages
  - Node stores f intervals
    - → More data load
  - After breakdown of a node
    - Find new successor
    - Replicate data to next node
      - → More message overhead at breakdown
  - Stabilize-function has to check every Successor-list
    - Find inconsistent links
      - → More message overhead

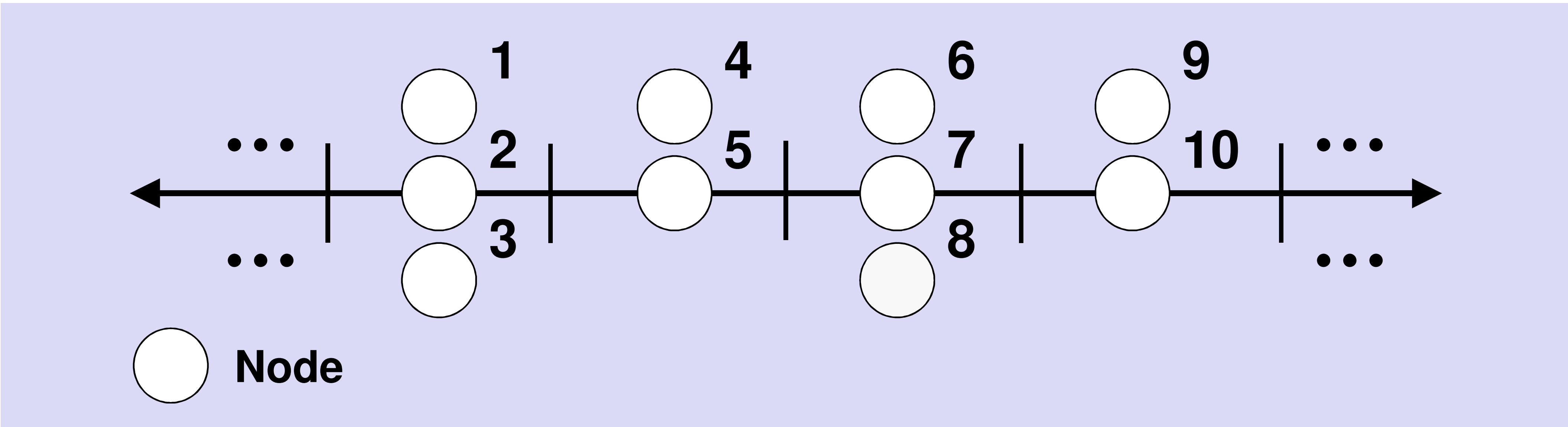
3.2. Multiple Nodes in One Interval (1)

- Fixed positive number f
  - Indicates how many nodes have to act within one interval at least
- Procedure
  - First node takes a random position
  - A new node is assigned to any existing node
  - Node is announced to all other nodes in same interval



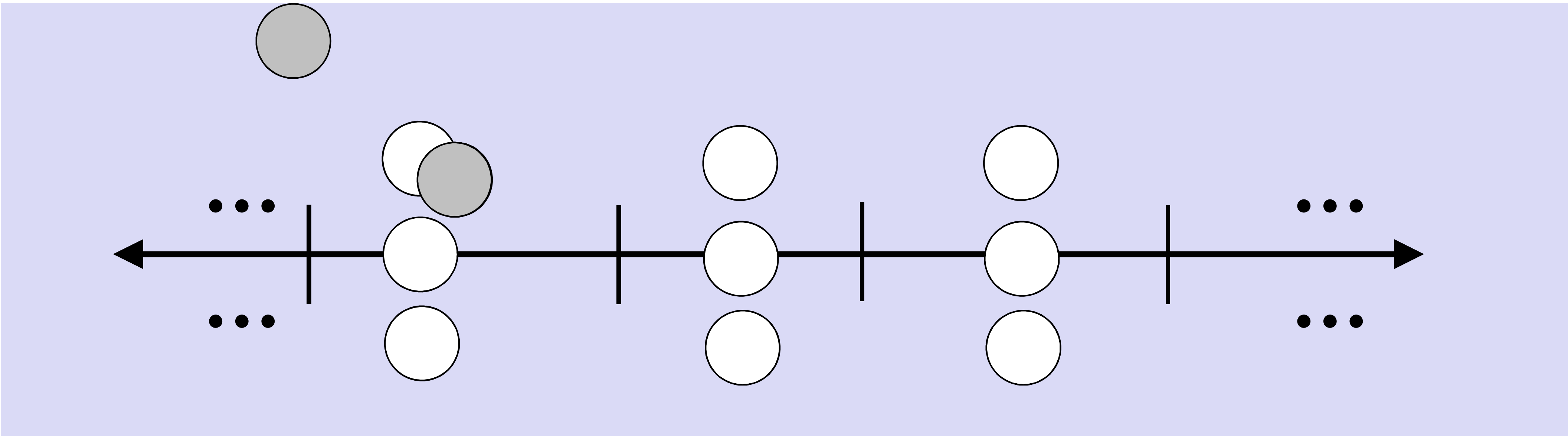
3.2. Multiple Nodes in One Interval (2)

- Effects of algorithm
  - Reliability of data
  - Better load balancing
  - Higher security



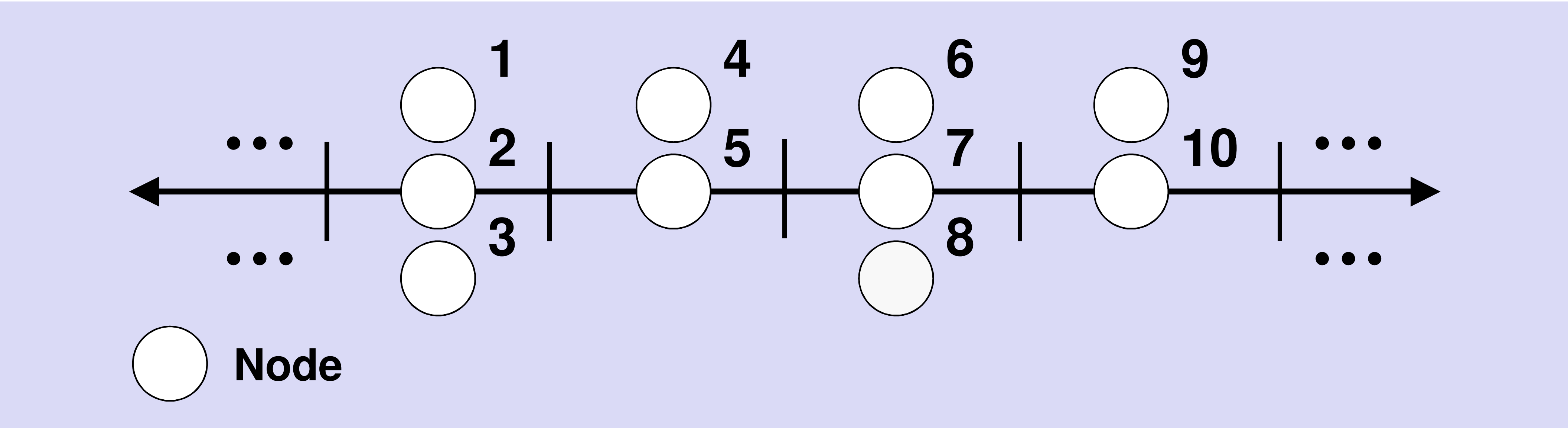
3.2. Reliability of Data (1)

- Insertion
  - Copy of documents
    - Always necessary for replication
  - Less additional expenses
    - Nodes have only to store pointers to nodes from the same interval
  - Nodes store only data of one interval



3.2. Reliability of Data (2)

- Reliability
  - Failure: no copy of data needed
    - Data are already stored within same interval
  - Use stabilization procedure to correct fingers
    - As in original Chord



3.2. Properties

- Advantages
  - Failure: no copy of data needed
  - Rebuild intervals with neighbors only if critical
  - Requests can be answered by f different nodes
- Disadvantages
  - Less number of intervals as in original Chord
    - Solution: Virtual Servers