Distributed Hash Tables

Advanced Computer Networks

Summer Semester 2012

P2P Systems

- We saw unstructured systems:
  - Napster (still uses some client/server)
  - Gnutella
  - BitTorrent (Swarming, but again uses trackers)

- Structured systems:
  - Routing & Lookup
  - DHTs

- These slides are based on a lecture by Prof. Roscoe, ETH Zürich, and provided with his kind permission.

Problem Space

- Challenge: spread lookup database among P2P participants
- Goals:
  - Scalable – operates with millions of nodes
  - Self-organized – no central, external control
  - Load-distributing – every member should contribute (at least ideally)
  - Fault tolerant – robust against node leaves or failures
  - Robustness – resiliance against malicious activity

Idea

- Distributed Hash Tables
  - Hash content identifiers to machines
  - Hash IP addresses
  - Store content (or content locator) at machine with closest hash value
- Originally 4 papers submitted to SIGCOMM 2001:
  - CAN, Chord, Pastry, Tapestry

Background: Hash Functions

- Hash function maps arbitrary input sequence to fixed length output:
  - H(m) = x, x of fixed length
- Crypto-Hashes:
  - Small input changes result in large output changes (Avalanche criterium)
  - If H(m1) = x is known, it is hard to find another m2 giving H(m2) (collision resistant)
- Inheritly hash functions span whole 2^k space (k bits hash length)

MD5 / SHA-1

- Message Digest Algorithm 5
  - 128 bit hash values
  - Weak collisions found

- SHA-1 (similar to MD4)
  - 160 bit hash values
  - Stronger than MD5, but „under researcher‘s attack“: find collisions in 2^{69}

- But: Both algorithms efficiently map input homogeniously to 2^k space

DHTs

- Index data by hash value
- Assign each node in the network a portion of the hash address space
- DHT provides the lookup function

Example: Chord

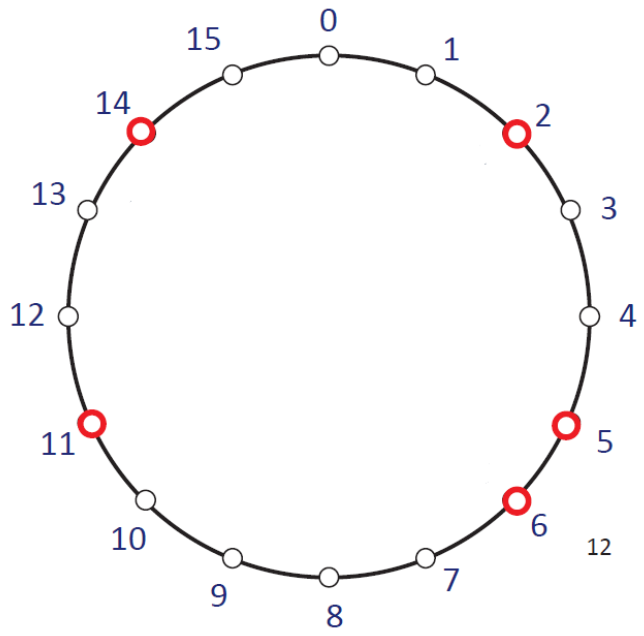
- Published 2001 at SIGCOMM by Stoica et al. „Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications”

- Keys are SHA-1 hashes – 160 bit identifiers
- Key: Identifier of a data item
- Value: Identifier of a node

- Host (key,value) pair at node with ID larger or equal to key – successor(key)

Identifier Space

- Identifier in 2^4 space
  - Space from 0..15
  - Nodes pick IDs:
    - 2,5,6,11,14 covered by nodes
    - Remaining values are not directly covered by a node

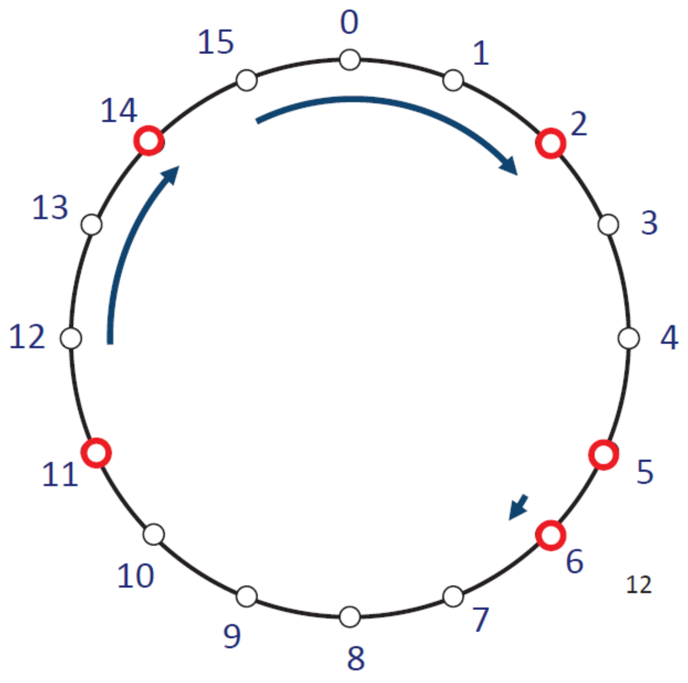


12

Successor

- First node in clockwise direction with ID larger or equal the key

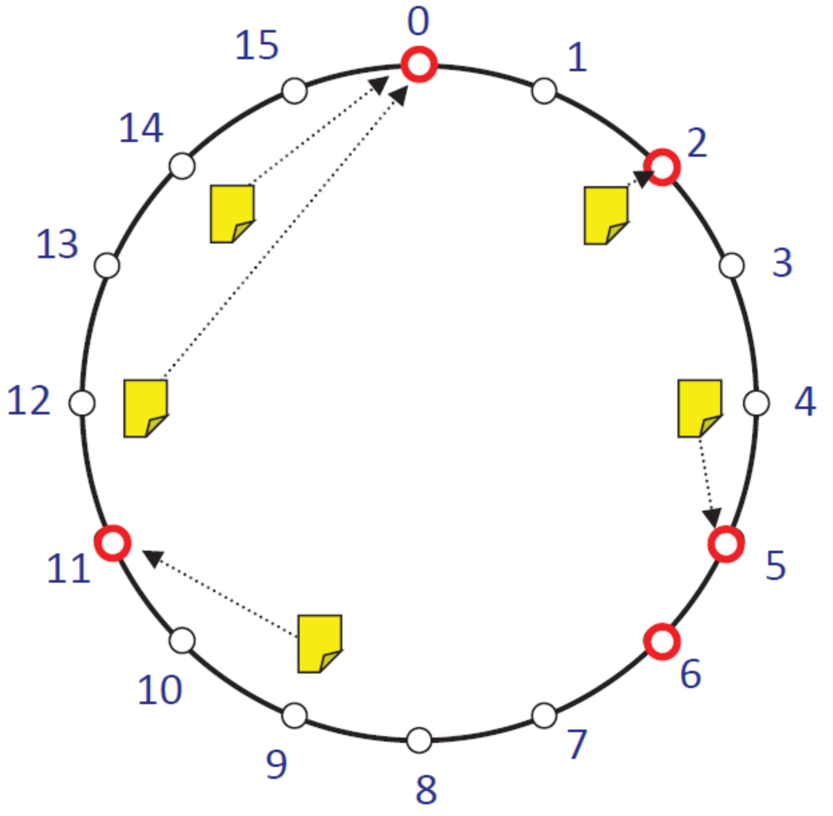
- Examples:
  - succ(6) = 6
  - succ(12) = 14
  - succ(15) = 2



How to store and locate data?

- Each (key,value) pair is assigned the identifier H(key)
- Each item is stored at its succ(H(key))

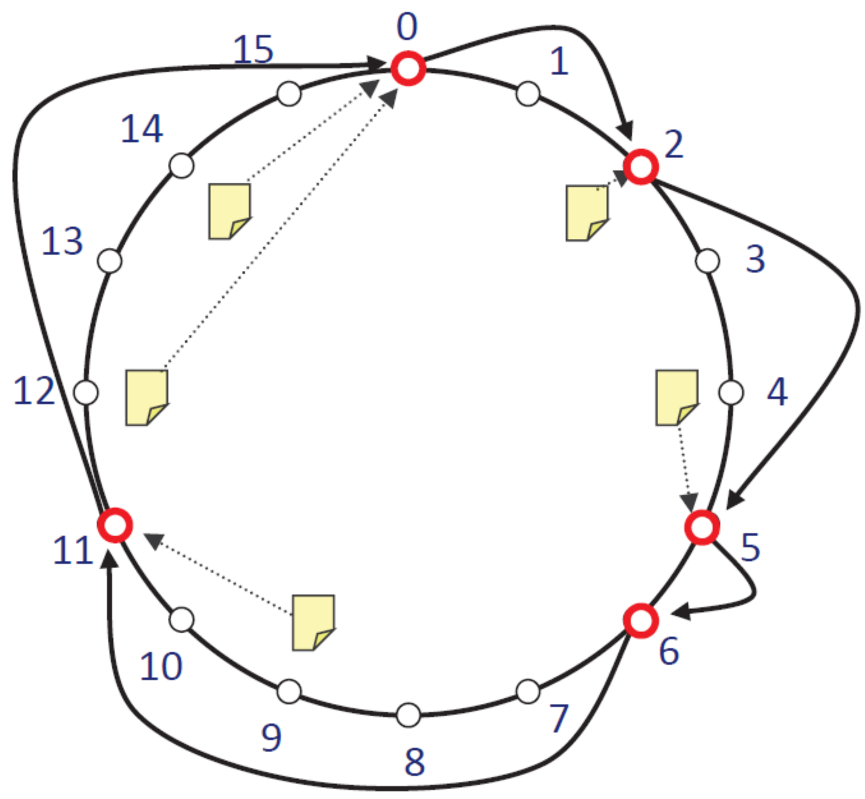
Drink	Location	H(Drink)
Beer	Göttingen	12
Wine	France	2
Whisky	Scotland	9
Wodka	Russia	14



Successor Pointer

- Each node points to its successor
  - Known as node‘s succ pointer
  - Successor of n is succ(n+1)

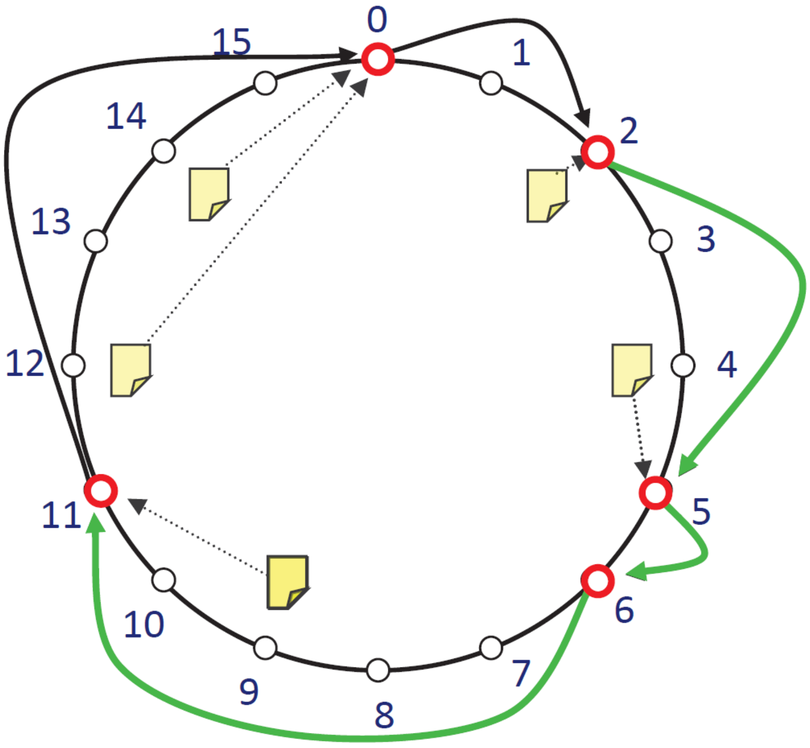
- Example:
  - 0‘s succ = succ(1) = 2
  - 2‘s succ = suss(3) = 5
  - ...



Basic Lookup of Data

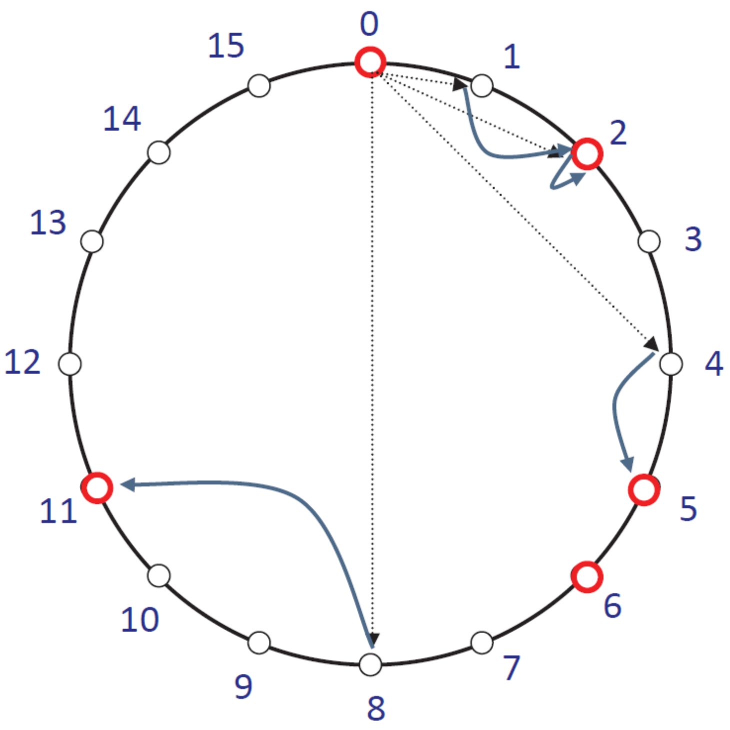
- Lookup key:
  - Calculate H(key)
  - Follow succ pointers until key is found
  - Lookup time: O(n)

- Example:
  - „Where can I drink Whisky?“
  - Calculate H(Whisky) = 9
  - Traverse nodes:
    - 2,5,6,11
  - Return „Scotland“



Scalable Lookup

- Each node maintains finger table (max k entries)
- for i in 0..k-1: finger[i] = succ($n+2^{i-1}$)
  - Point to succ(n+1)
  - Point to succ(n+2)
  - Point to succ(n+4)
  - ...
  - Point to succ($n+2^{i-1}$)

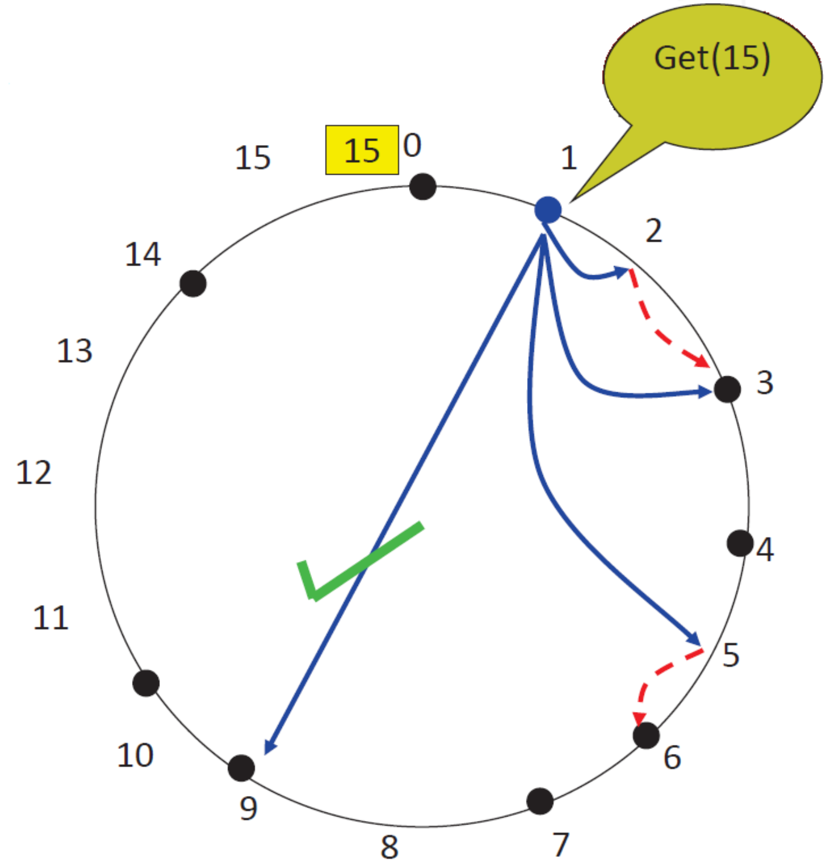


- Makes lookup time logarithmic!
  - O(log n)

Routing

- Determines the next hop
- Each node n knows $succ(n+2^{i-1})$ for all i=1..k
- Forward queries for key to then highest predecessor of key

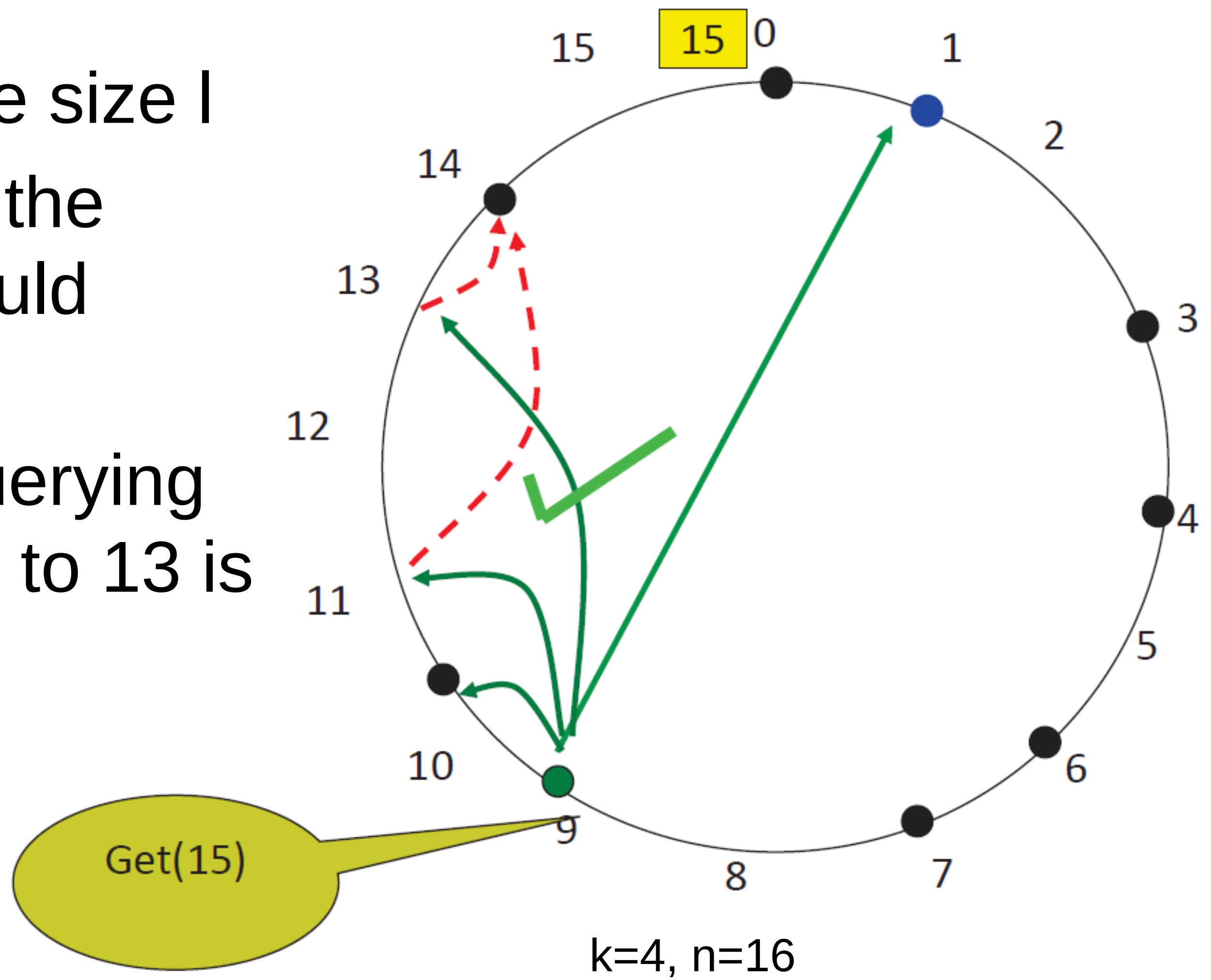
- Routing entries = $\log_2(n)$



k=4, n=16

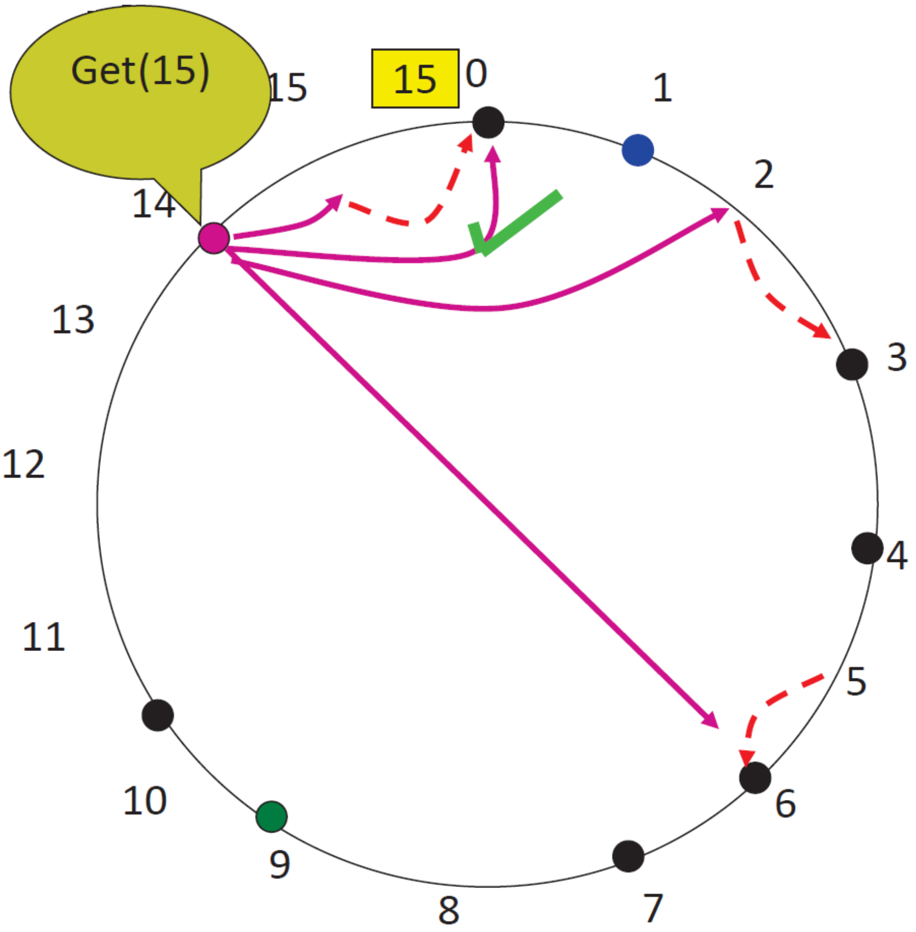
Routing cont‘d

- Routing table size l
- Node 9 was the highest 1 could reach
- Node 9 is querying again, finger to 13 is best



Routing cont‘d

- 13 is handled by 14
- 14 completes the route:
  - 15 is found at 0



Routing cont'd

- From node 1, 3 hops to node 0 where item 15 is stored
- k=4 equals an ID space of 16, therefore the maximum number of hops is:
  - $\log_2(16) = 4$
- Average complexity is ½ log(n)

Routing cont‘d

- Such routing algorithms solve the lookup problem
- General concept:
  - Each node has only a limited view on the network
  - A node that receives a message containing a destination ID that is not managed by that node, it just forwards the request to the closest hop
- Here, algorithm is based on numeric closeness

- In Gummadi et al., „*The Impact of DHT Routing Geometry on Resilience and Proximity*“, SIGCOMM 2003, implications are discussed

Recursive vs. Iterative Lookup

- Recursive: Each node forwards the request (as shown) to the next hop
  - Fast, efficient
  - Each node can optimize forwarding

- Iterative: The requesting client queries the next hop iteratively from the nodes
  - Allows the lookup client to keep in control
  - Lookup client detects and localizes failures

Achieved goals

- The DHT is scalable, as operations are performed in log(n)
- It is self-organized as each node directly knows its position (thanks to the hash function) and learns about the next hops
- On average load-distributing

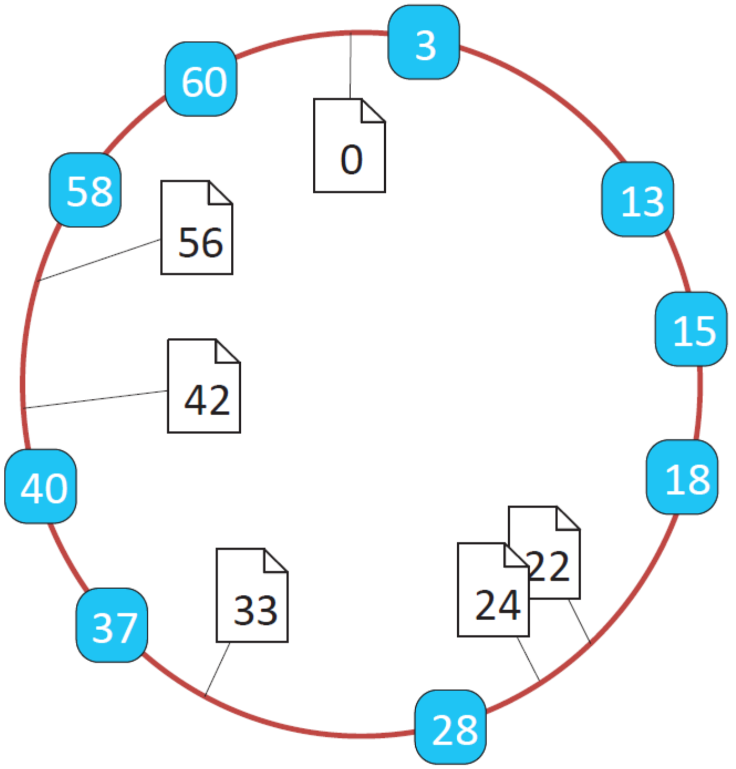
- What about joins and especially leaves?

Node Join and Leave

- Node join:
  1. Bootstrap: a new node contacts a known node in the DHT
  2. The new node gets a partion of the address space
  3. Routing information is updated
  4. The new node retrieves all tuples for which it is responsible
- Node departure:
  - Replication and load balancing
- Node failure:
  - Reactive or proactive recovery
  - Maintenance, load balancing, redistribution of tuples
  - Data is lost if not replicated!

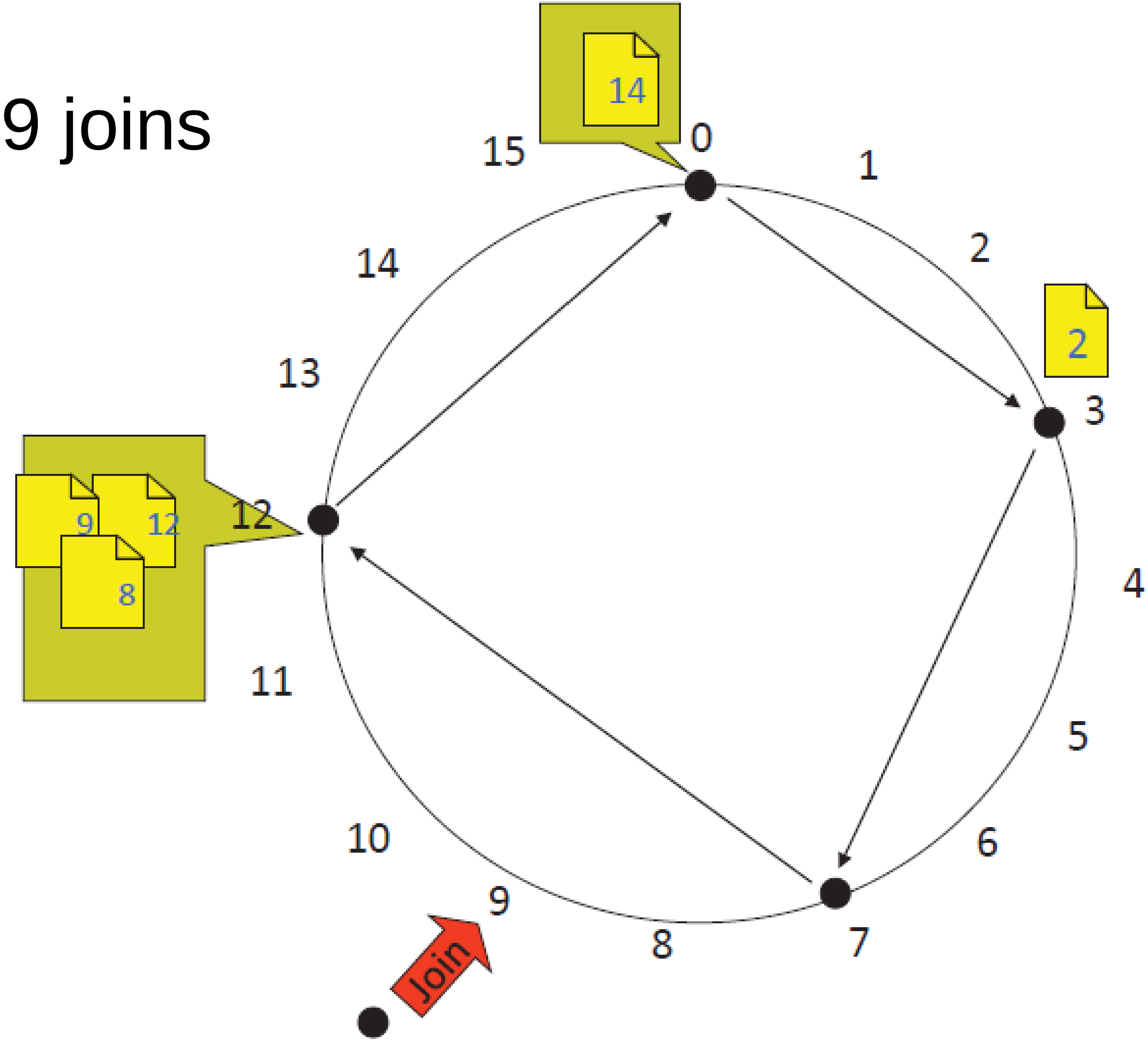
Node Join and Leave

- Join:
  - Lookup of own ID‘s successor
  - Contact that to get successors and predecessor
- Leaves:
  - Ping successors regularly
  - Always ensure x live nodes in successor set
- Thereby, failures are treated as „normal“



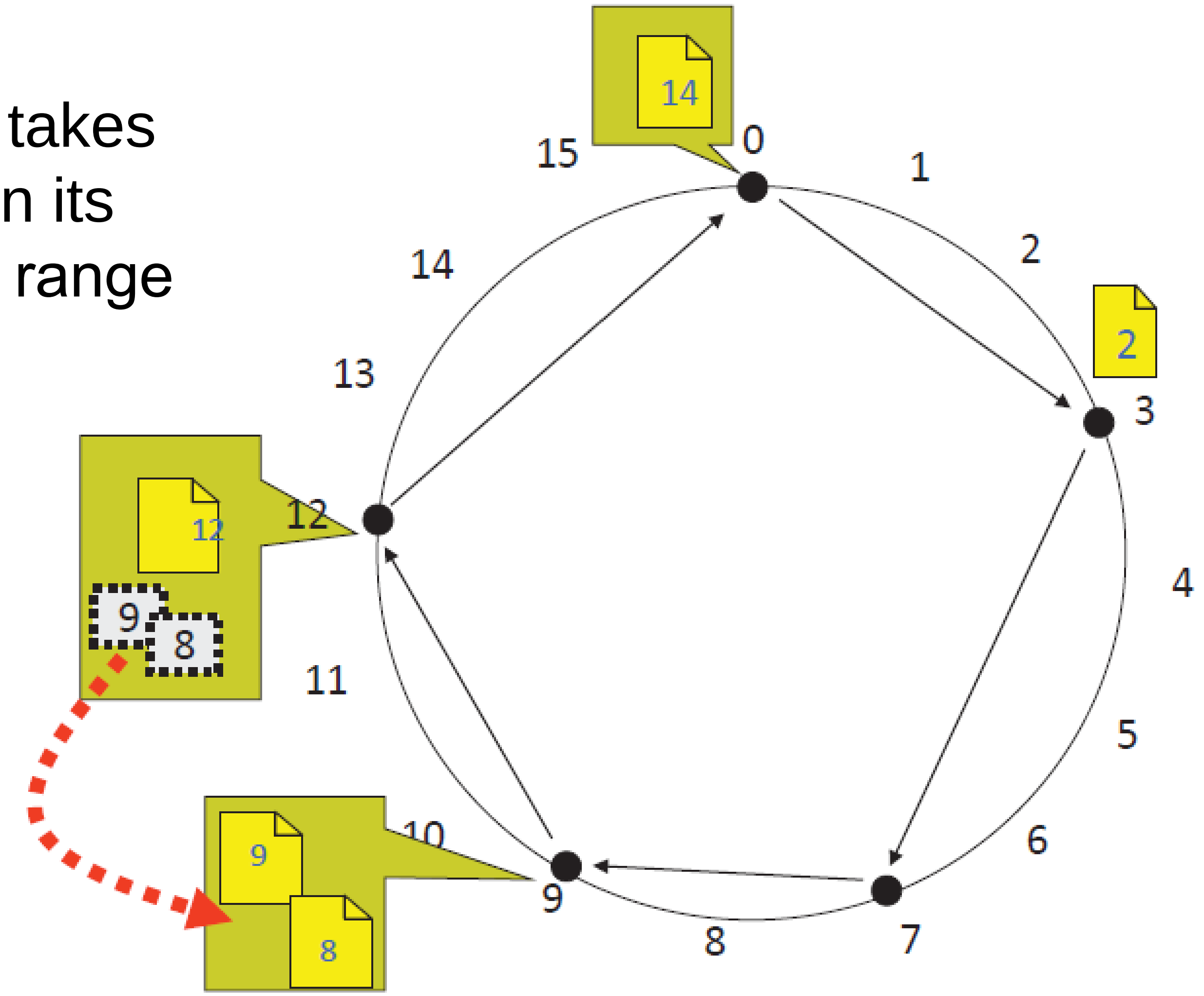
Node Join Example

- Assume node 9 joins



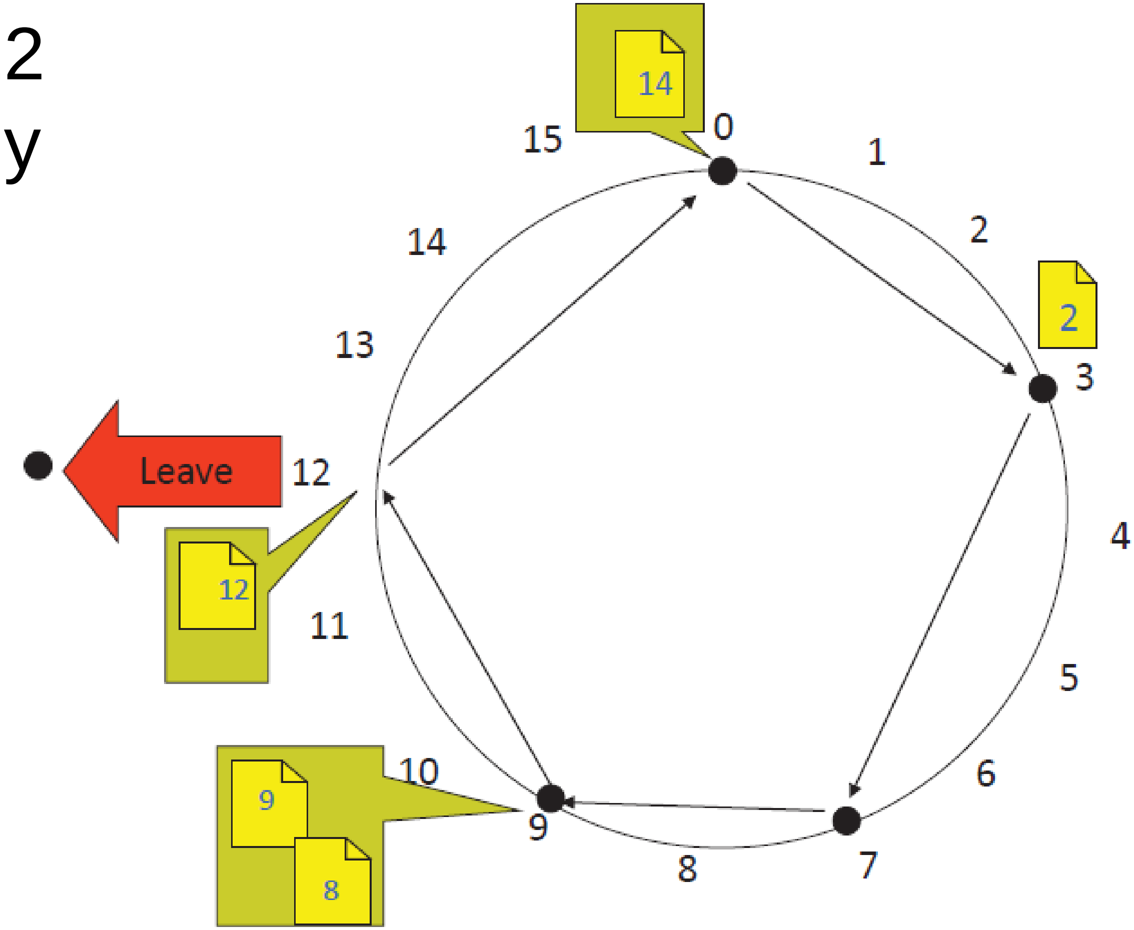
Node Join Example cont'd

- The new node takes over the docs in its "responsibility" range
- Docs 9,8 from its successor



Node Leave

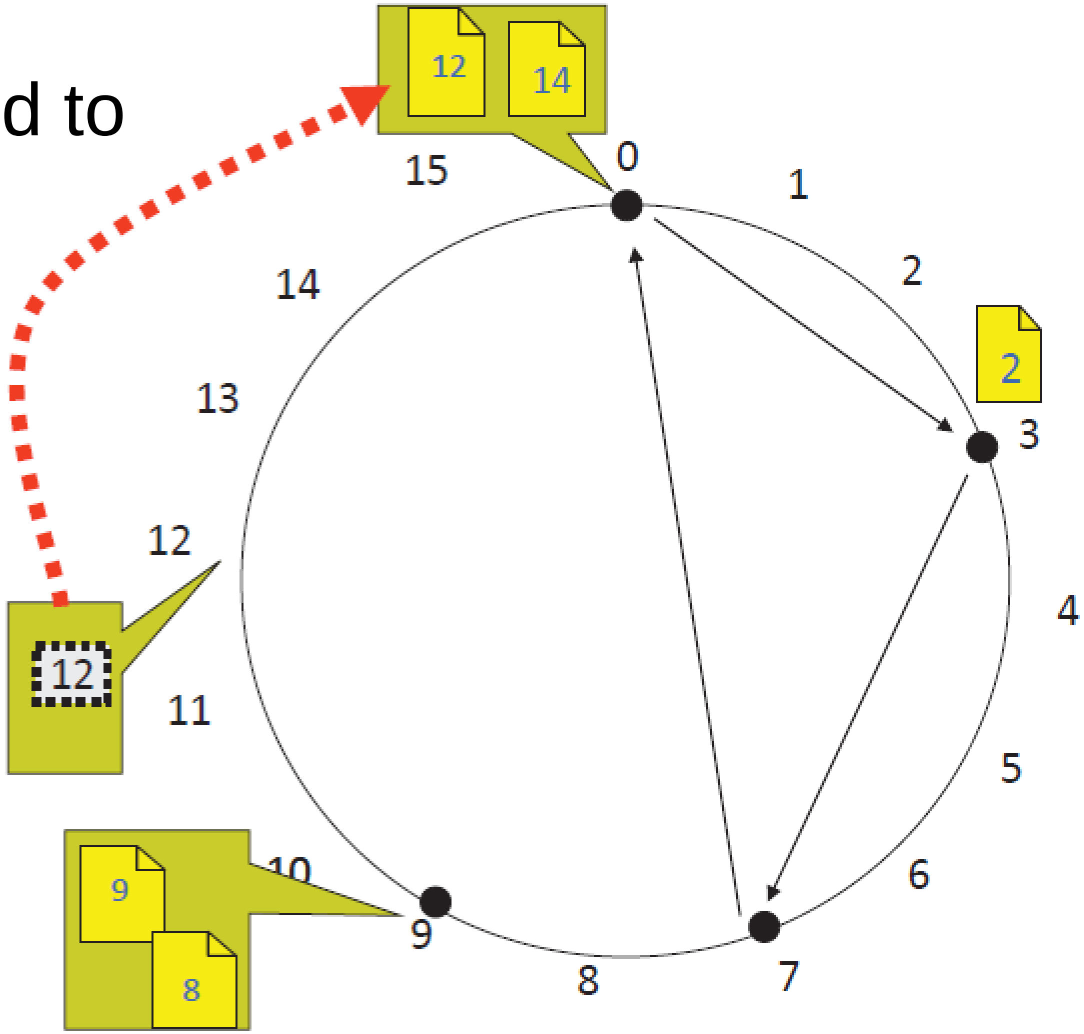
- Assume node 12 leaves gracefully



Node Leave cont'd

- Data is transferred to succ(12) = 14

- Node 12 informs predecessor and successor, who update their finger tables



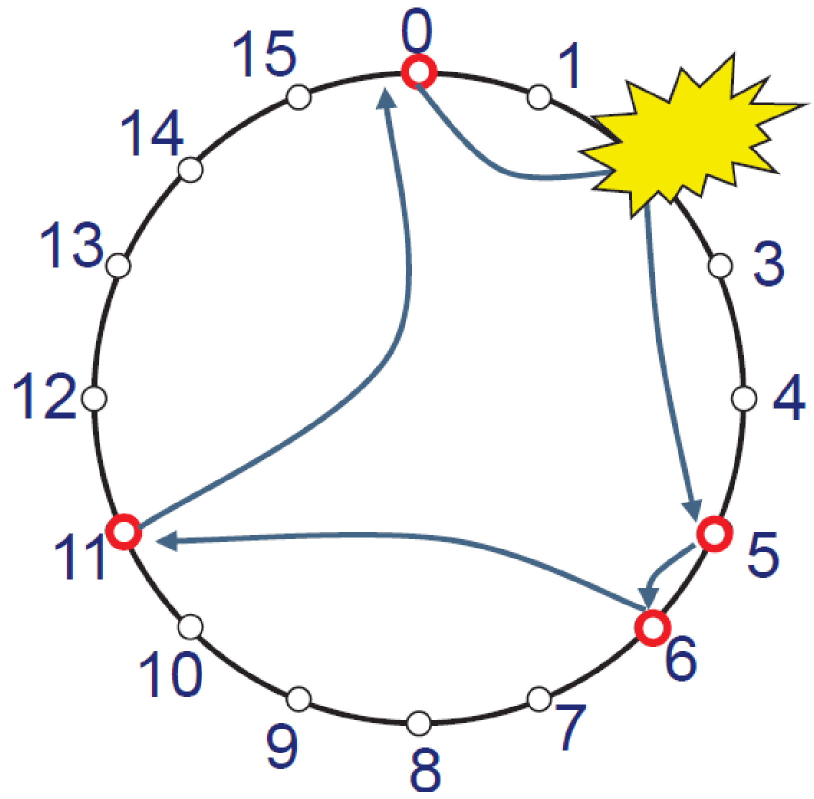
Direct vs. Indirect Storage

- Direct storage:
  - Actual data is stored at the node responsible for it
  - The data is copied towards the responsible node upon node join
  - The node that contributed the data can leave without loss of its data
  - But: High storage and communication overhead!

- Indirect storage:
  - Instead of data, the references to the data are stored
  - The inserting node keeps the data
  - Lower load on the DHT

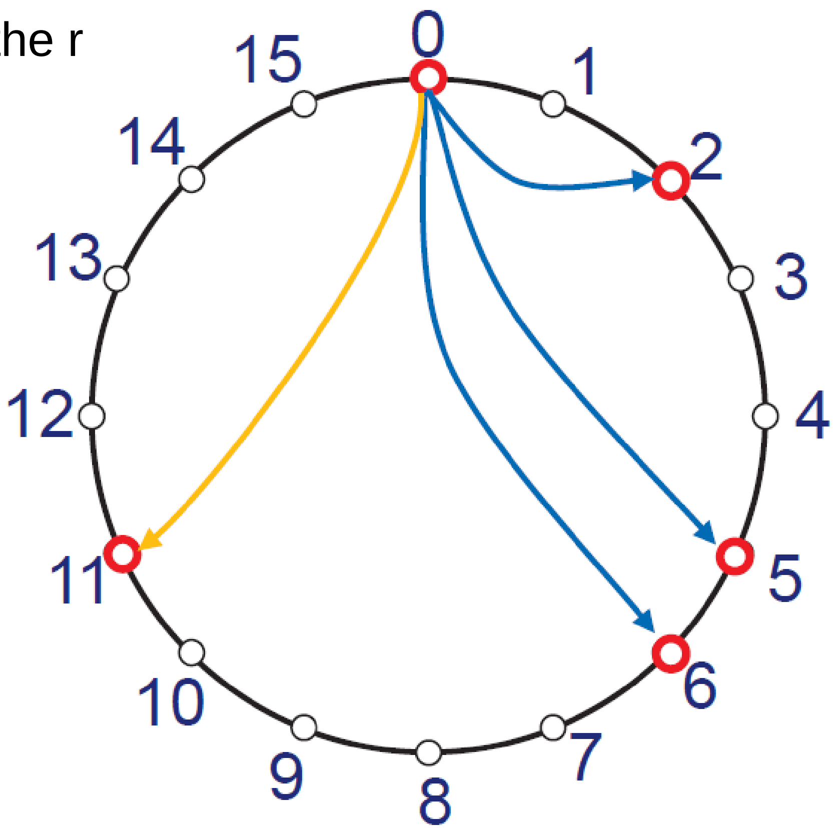
The Fragile Ring

- Problem: Everything is organized in a fragile ring structure
  - Failure of a node breaks the ring and data is lost
  - No way to recover as previous predecessor and successor don't know about each other!



Successor Sets

- As a solution, each node keeps:
  - A Successor set with pointers to the r closest successors
  - Predecessor pointer
- If successor fails, replace with closest alive successor
- If predecessor fails, set pointer to nil
- Replicate objects throughout the successor set



Further Challenges

- How does a node learn its:
  - Predecessors?
  - Fingers?
- What if “better” fingers come along later?
  - How would a node find out?
- How does a node react to failing or leaving fingers?

- All basically the same problem

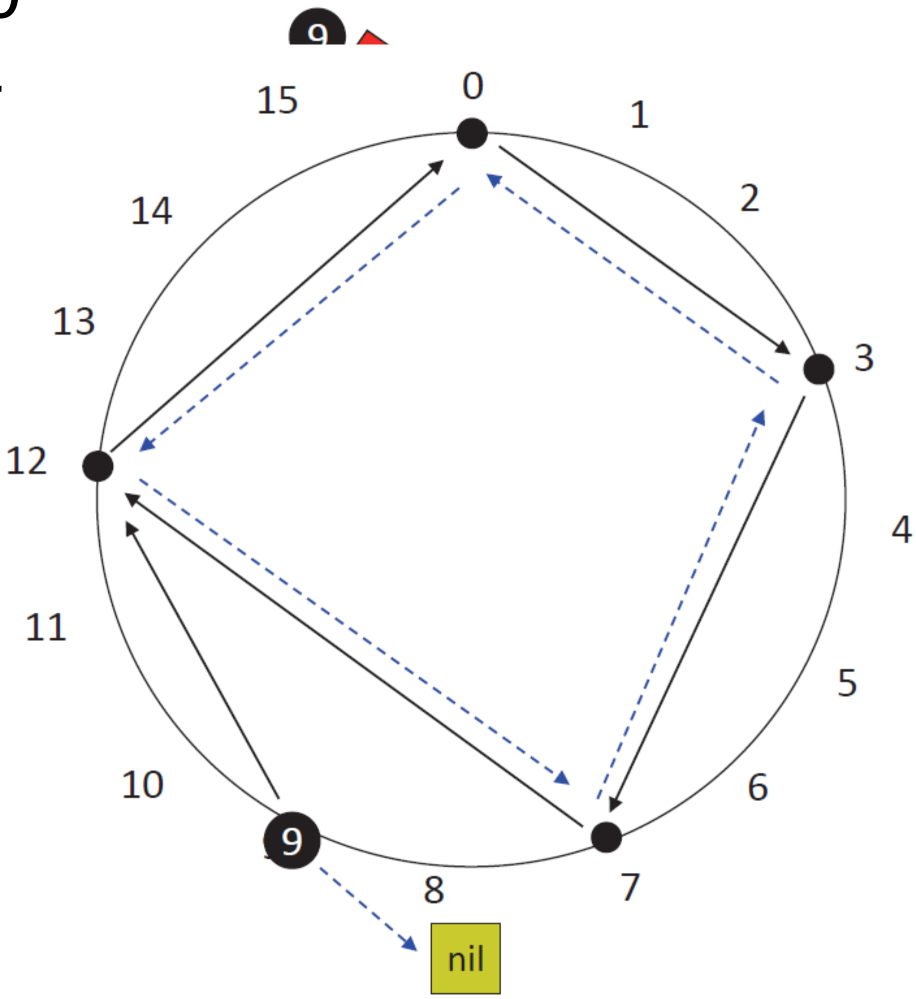
Periodic Stabilization

- Used to make pointers eventually correct
- Requires an additional predecessor pointer
  - First node met in anti-clockwise direction starting at n-1

- A node n joins the DHT through a node o:
  - Find n’s successor by lookup(n)
  - n sets its successor to the found successor
  - Stabilization fixes the rest
    - stabilize() function is run peridically by each node
  - The new node does not determine its predecessor: its predecessor detects and fixes inconsistencies

Periodic Stabilization Example

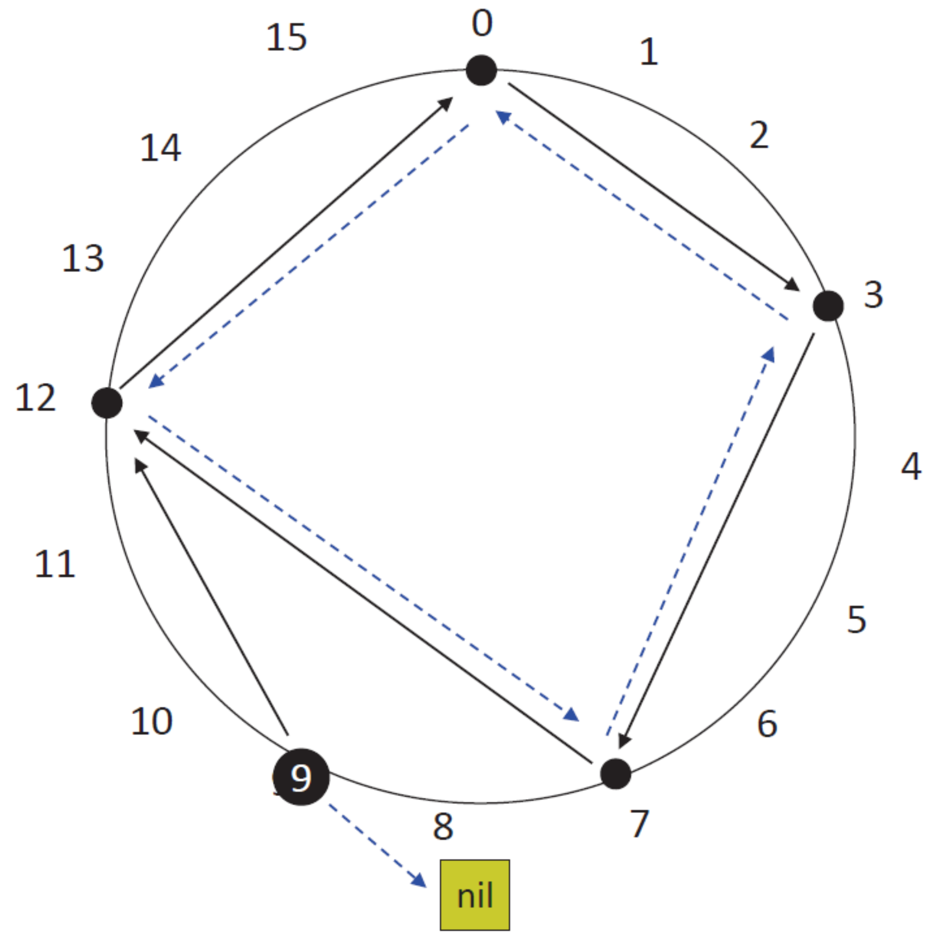
1. 9 joins through node 0
2. 9 sets its predecessor to nil
3. 9 asks 0 for succ(9). Receives “12”
4. 9 sets its succ to 12



Periodic Stabilization Example

- 9 runs stabilize()

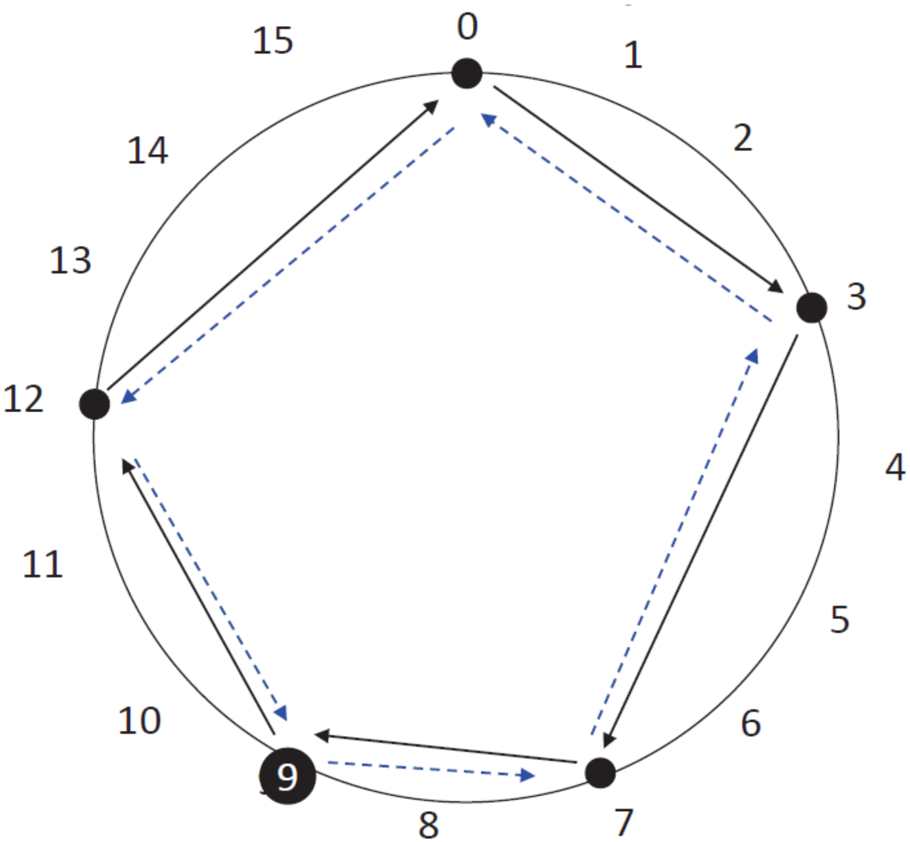
1. 9 asks 12 for its predecessor
2. 12 replies with “7”
3. 9 notifies 12 that 9 is now its predecessor



Periodic Stabilization Example

- 7 runs stabilize()

1. 7 discovers from 12 that pred(12) is now 9
2. 7 sets successor to 9
3. 7 notifies 9
4. 9 sets pred(9) to 7



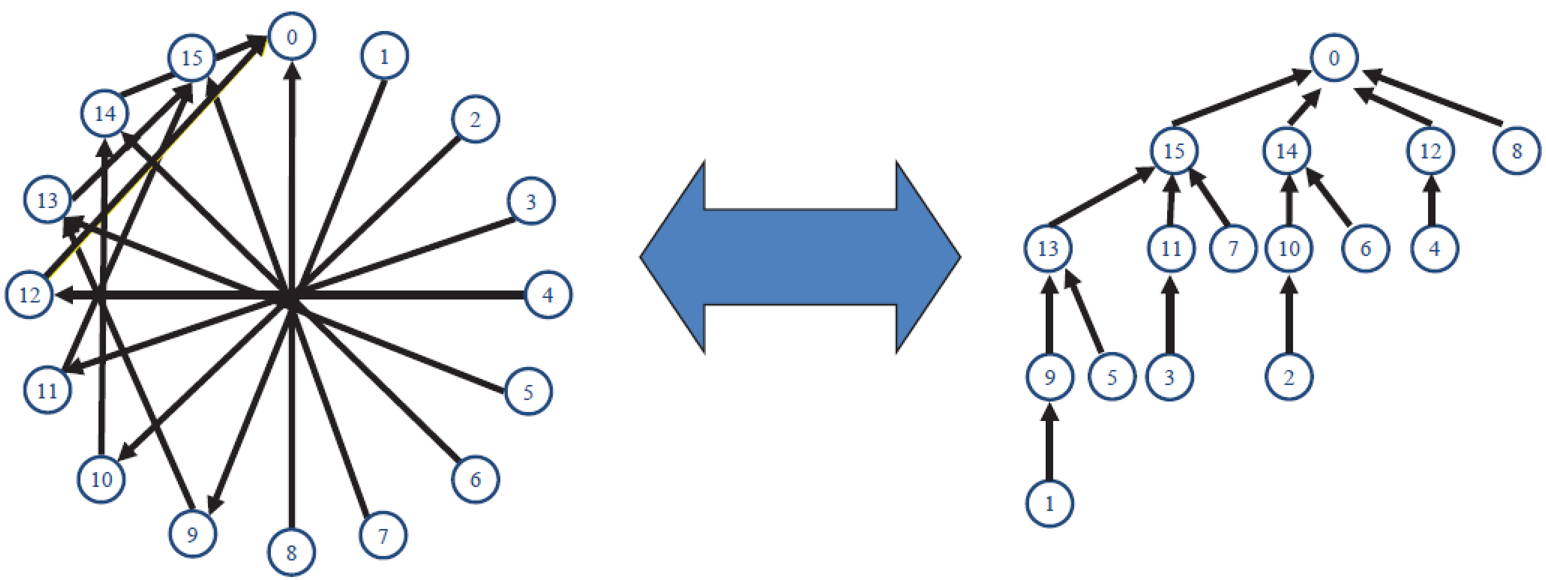
Stabilizing Fingers?

- Each node runs fix_fingers() periodically
  - Refresh finger table entries and store the index of the next finger to fix
  - This is also the initialization procedure for the finger table

```
n.fixfingers()
next = next +1;
if (next > k) //check for max size
   next = 1;
finger[next] = find_successor(n+2^(next-1));
```

Chord in a "Tree View"

- Finger tables are Chord's core
  - Providing O(log n) hop routing by at least halving the distance to the target by each hop
  - Forest of binomial trees rooted at each key



Chord - Conclusion

- Lookup time: O(log n)
- Drawbacks:
  - Rigidity
    - Complicates recovery from failed nodes and routing table
    - Precludes proximity-based routing
  - Unidirectional routing
  - Incoming traffic is not used to re-enforce routing tables

- Fault-tolerant, but not very robust.

Kademlia - Goals

- Flexible routing table
  - Benefits from proximity-based routing
  - Minimal maintenance as configuration information automatically spreads together with key lookups

Kademlia: Distance Metric

- The distance between two 160-bit identifiers (e.g., SHA-1 hashes) is defined as their bit-wise XOR interpreted as an integer

- XOR example:
  - `A       = 0 1 0 1 1 0 (22)`
  - `B       = 0 1 1 0 1 1 (27)`
  - `A XOR B = 0 0 1 1 0 1 (13)`

- Intuition: Differences at higher order bits matter more than differences at lower order bits

Advantages of Distance Metric

- The exclusive OR operation shares some properties with “normal” geometric distances:
  - The distance between a node and itself is zero $D(x,x) = 0$
  - The distance function is symmetric: $D(x,y)=D(y,x)$
  - It follows the triangle inequality: $D(x,z) \leq D(x,y) + D(x,z)$
- The distance is not reflecting any topological properties!

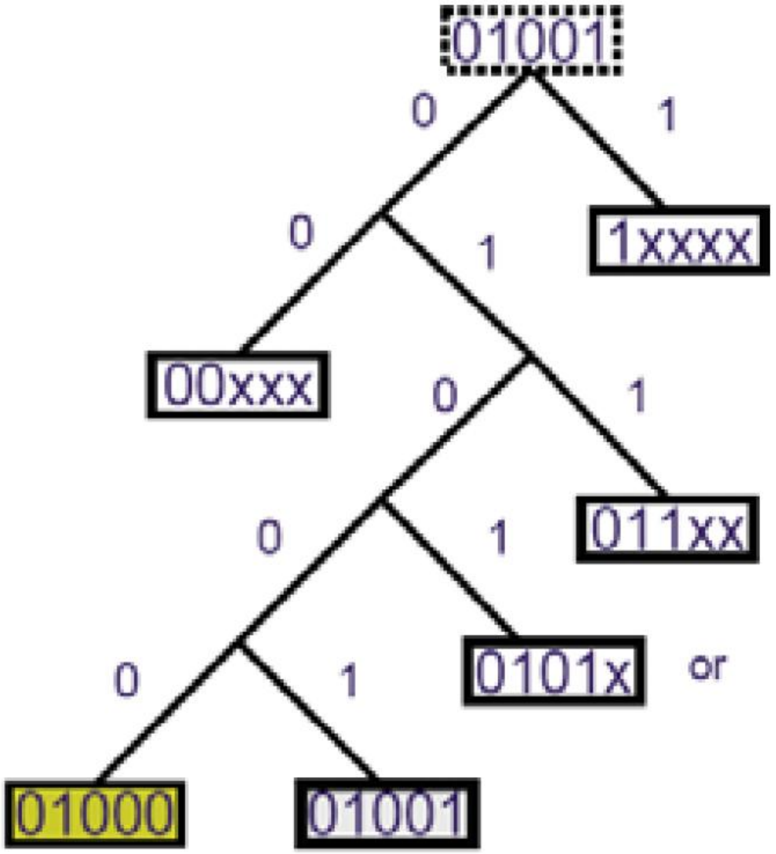
Kademlia: Routing Table

- For each $0 \le i < 160$, each node keeps a list of the triple <IP,port,nodeID> for nodes of distance of 2^i and 2^{i+1} from itself

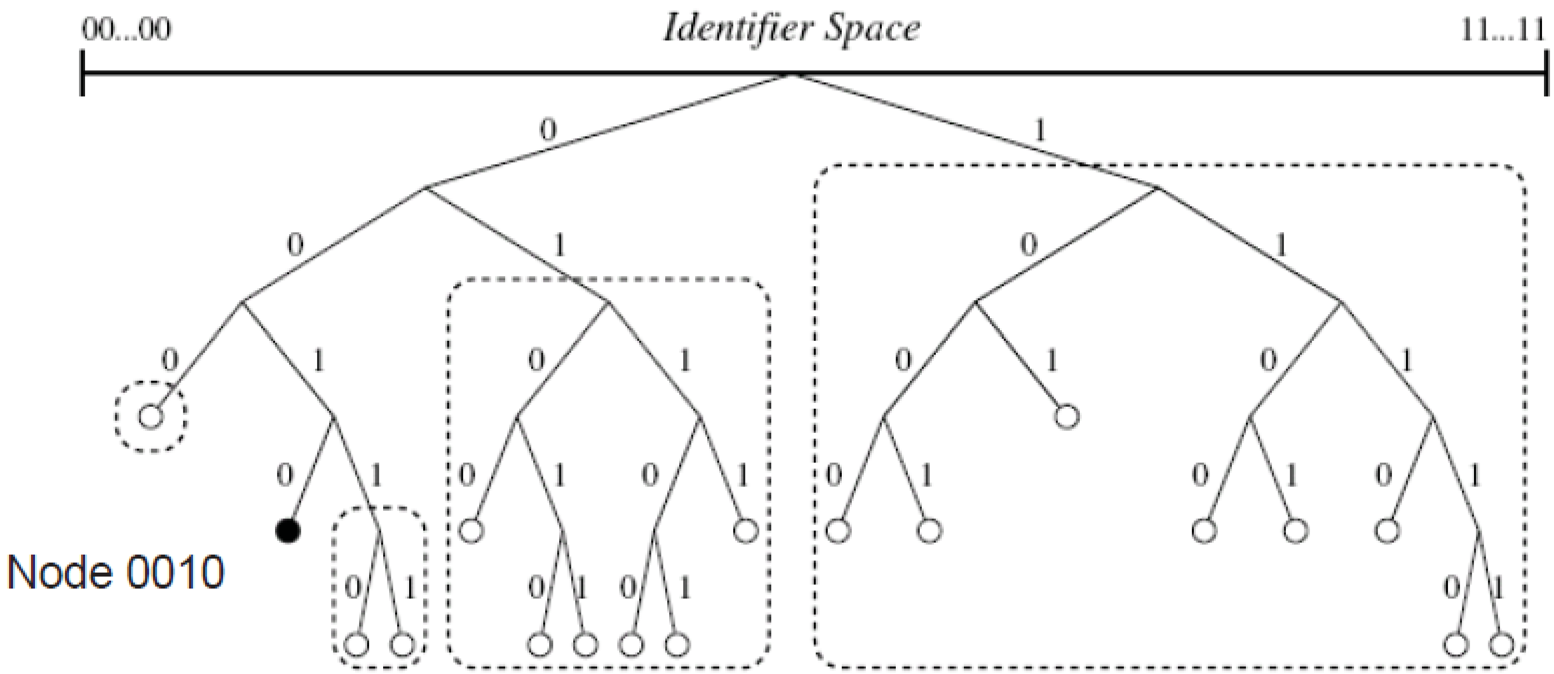
- Each list is called a bucket and stores at most k triples
  - A k-bucket stores at most k nodes that are at distance $[2^i, 2^{i+1}]$
  - Each bucket is kept sorted by time last seen

Example for k=1

- Node 01001
- Distance $[2^0,2^1)$: 01000
- Distance $[2^1,2^2)$: 0101X
- Distance $[2^2,2^3)$: 011XX
- Distance $[2^3,2^4)$: 00XXX
- Distance $[2^4,2^5)$: 1XXXX



Kademlia Topology



- Kademlia treats nodes and keys as leaves of a binary tree
- Each node knows of at least one node in each of the subtrees

Kademlia Routing

- Iterative lookup:
  - Longest matching prefix forwarding: A query is forwarded to the “best” subtree until the destination is reached
  - A node often knows of more than a single node per subtree so that queries can be forwarded in parallel to multiple nodes in a subtree (resliance!)

- Lookup time: O(log n)

Kademlia: Updating Buckets

- Whenever a node receives any message, it updates the appropriate k-bucket based on the sender's information
- If the bucket is full, the oldest entry is removed, if it is not alive
  - Keeping old nodes alive maximizes the probability that the nodes in the bucket will remain online (the long-time persistent nodes)

Kademlia Conclusion

- Easy table maintance
  - Tables are updated when lookups are performed
- Fast lookup by making parallel searches – but at the expense of increased traffic
- Used in many deployed file sharing networks:
  - Kad Network (eMule)
  - BitTorrent when using trackerless BT
  - Gnutella DHT