9. Peer-to-Peer Systems

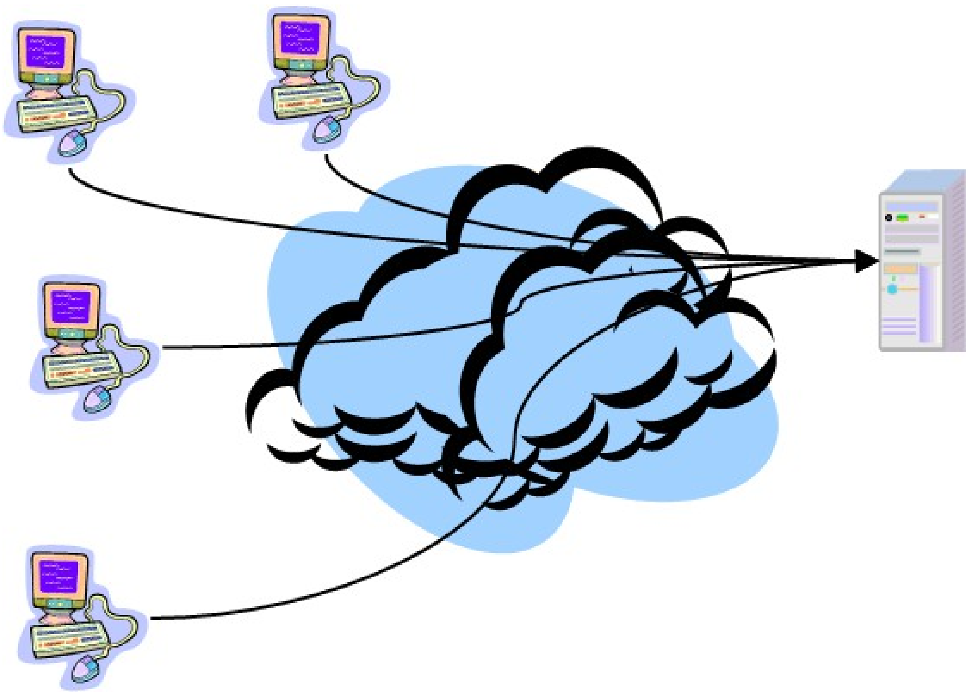
Sistemes Distribuïts en Xarxa (SDX)
Facultat d'Informàtica de Barcelona (FIB)
Universitat Politècnica de Catalunya (UPC)
2017/2018 Q2

Contents

- **Introduction**
- Unstructured P2P systems
- Structured P2P systems

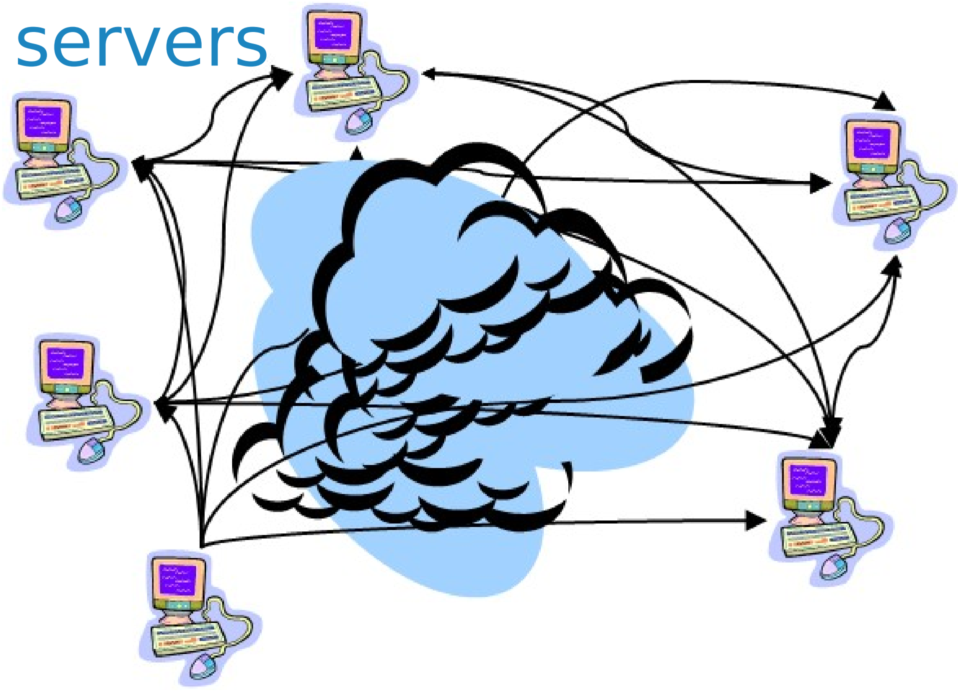
Introduction

- Client-server systems have been common (and successful) until now, but incur some scalability and robustness problems in modern scenarios
  - Need for data and resource sharing on <u>a very large scale</u>
  - Sudden spikes in demand for data and resources
  - Instability of nodes and links (can come and go at any time)



Introduction

- Peer-to-Peer (P2P) systems do not distinct between clients and servers
  - Peer functionality is symmetric, all peers contribute resources
  - Loads are balanced between all peers
  - Operation independent of any central node



- High scalability & availability
  - Huge amount of computation and storage resources

Introduction

- Peers are organized in an **overlay network**
  - The nodes are formed by the peers and the links represent the possible communication channels
  - When peers cannot communicate <u>directly</u> with each other, messages have to be <u>routed</u> through the available links
- Efficient and fault-tolerant placement of data items across peers and their subsequent access is the <u>main challenge</u> in P2P systems
  - There is a huge number of peers, and they

Introduction

- Two types of overlay networks exist:
  1. **Unstructured P2P systems**
     - Overlay topology is constructed ad hoc (random graph)
     - Items end up placed arbitrarily (must be searched)
     - Insert is easy, search is hard or imprecise
     - Examples: Napster, Gnutella, KaZaA
  2. **Structured P2P systems**
     - Overlay follows a specific deterministic topology
     - Items are stored deterministically at specific peers
     - Insert is hard, search is easy
     - Examples: DHT-based systems
       - Kademlia, Chord, Pastry, Tapestry, CAN

Contents

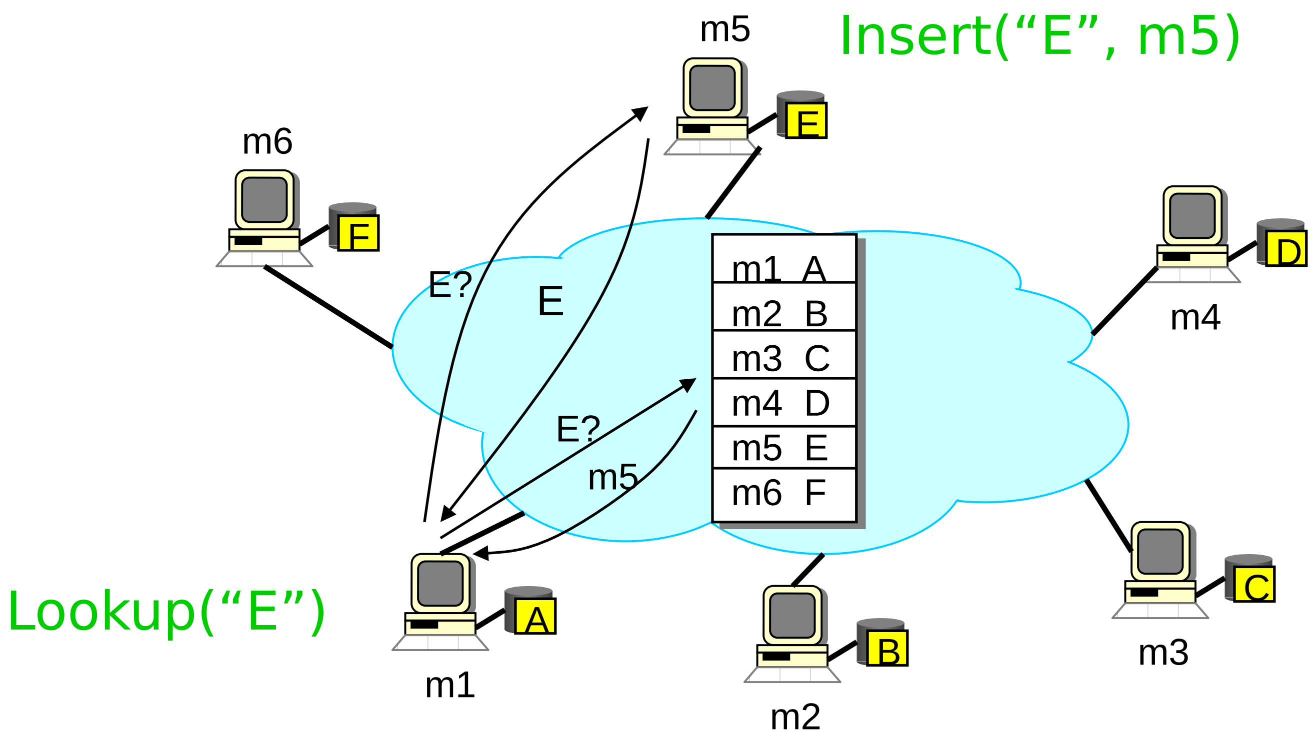
- Introduction
- **Unstructured P2P systems**
- Structured P2P systems

Contents

- Introduction
- **Unstructured P2P systems**
  - **Centralized model**
  - Decentralized model
  - Hierarchical model
- Structured P2P systems

Centralized model: Napster

A. Query a **centralized** index system that returns the peer/s that stores the required file
B. Transfer the file from the given peer/s



Centralized model: Napster

- Advantages
  - Simple and locates files quickly and efficiently
  - Easy to implement sophisticated search engines on top of the index system
- Disadvantages
  - ↓ Lack of robustness: single point of failure
  - ↓ Scalability / performance bottleneck
  - ↓ Resources are distributed, but index service is not
    - Easy to detect copyright infringements
    - For this reason, Napster could be shut down

READING REPORT

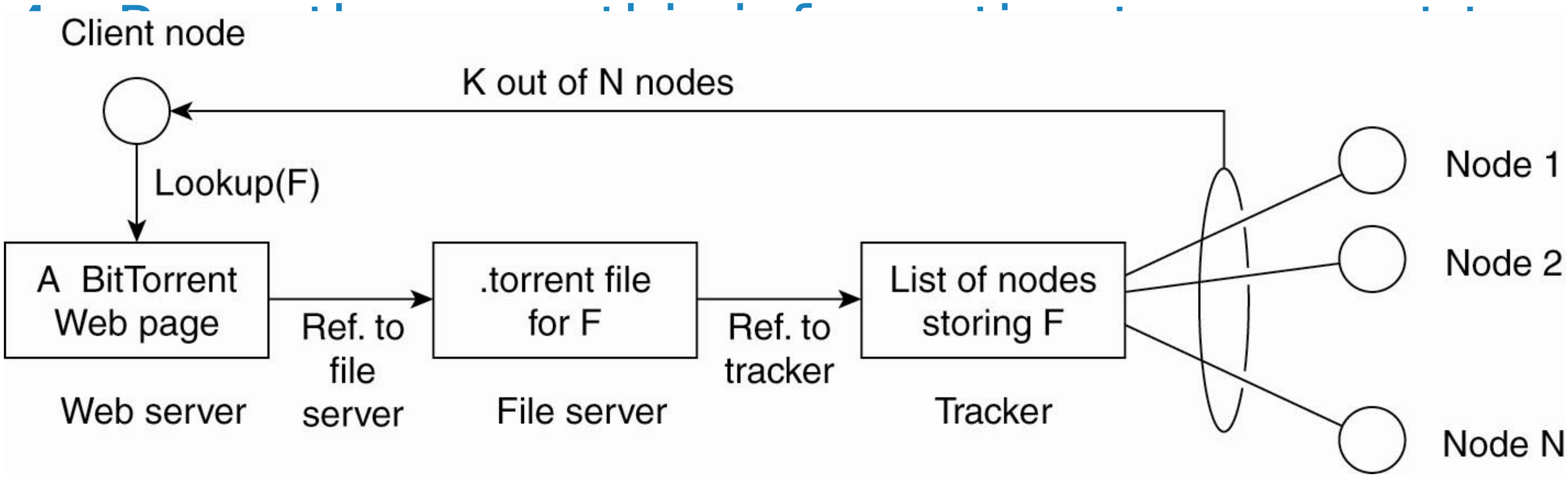
- **[Cohen03]** Cohen, B., *Incentives Build Robustness in BitTorrent*, 1st Workshop on Economics of Peer-to-Peer Systems, Berkeley, CA, USA, June 2003

Centralized model: BitTorrent

- Collaborative system providing efficient content distribution using file swarming
  - Each file split into smaller pieces
  - Nodes request desired pieces from neighbors
  - Encourages contribution by all nodes
- Combines a client-server model for locating the pieces with a P2P download protocol
  - Usually does not perform all the functions of a typical P2P system, like searching
- Written by Bram Cohen (in Python) in 2001

Centralized model: BitTorrent

1. Peers obtain a .torrent file, hosted in a web server
2. Peers connect to the tracker specified there
3. Tracker responds with contact information about the peers that are downloading the same file



Centralized model: BitTorrent

- .torrent file contains:
  - Metadata about the file to be shared
    - Name, length, piece length, SHA-1 hash of each piece
  - URL of the tracker
- Tracker
  - Keeps track of all the peers who have the file (seeds/leechers) and which chunks each peer has
  - When asked for a list of peers, the tracker returns a random subset of the peers, typically 50 peers
  - Can receive statistics about what peers are doing

Centralized model: BitTorrent

- Seeds
  - Peers that have a complete copy of the file
  - They are usually selfish and do not want to wait after they get the file
  - Initial seed: a peer that provides the initial copy
  - At least the initial seed has to stay to serve one complete copy of the file
- Leechers
  - Any peer who does not have a complete file
  - He stays a leecher and eventually becomes a seed

BitTorrent: Piece selection

- Each file is split into smaller pieces
  - This is the unit for uploading, typically 256Kb
- Pieces are further divided in sub-pieces
  - This is the unit for downloading, typically 16Kb
- The order in which pieces are selected is critical for good performance
  - Avoid ending with peers having all the same set of available pieces, and none of the missing ones
  - If the initial seed is prematurely taken down, then the file cannot be completely

BitTorrent: Piece selection

- **Rarest Piece First**
  - This is the general rule: download first the pieces that are most rare among your peers
    - Most commonly available pieces are left until the end to download
  - Increases diversity in the pieces downloaded
    - You are going to be useful to others
    - Avoids the case where all peers have exactly the same pieces
  - Increases likelihood that all pieces are still available even if the initial seed leaves

BitTorrent: Piece selection

- The behavior of the 'Rarest Piece First' policy can be modified by three additional policies:
  - Strict Priority, Random First Piece, Endgame Mode

- **Strict Priority**
  - Once a single sub-piece has been requested, the other sub-pieces from that piece are requested before sub-pieces from any other piece
    - As only complete pieces can be sent, it is important to minimize the number of partially-received pieces

BitTorrent: Piece selection

- **Random First Piece**
  - Special case at the beginning of the download
  - A peer has nothing to trade so it is important to get a complete piece as soon as possible
  - ‘Rarest Piece First’ is not a good option
    - Rare pieces are present on few peers, so they would be downloaded slower
  - <u>Select a random piece</u> of the file and download it
  - When first complete piece is assembled, switch to ‘Rarest Piece First’

BitTorrent: Piece selection

- **End Game Mode**
  - Special case at the end of the download
  - The completion of a download can be delayed due to a single peer with a slow transfer rate
  - When all missing sub-pieces have been requested, those not received yet are requested <u>from every peer containing them</u>
  - When a sub-piece arrives, the pending requests for that sub-piece are cancelled
  - Some bandwidth is wasted, but in practice, this is not too much

BitTorrent: Choking

- Want to encourage all peers to contribute
  - Guarantee a reasonable level of upload and download reciprocation
  - Avoid ‘free riders’
- **Choking** is a temporary refusal to upload
  - A chokes B if A decides not to upload to B
- A given peer can unchoke only 4 remote peers at a given time
  - Not because you are connected to somebody you can download from him

BitTorrent: Choking

- Unchoking decision reconsidered periodically
  - Every 10 seconds, the interested remote peers are ordered according to their upload rate to the local peer and the 3 fastest peers are unchoked
    - i.e. initially they should have unchoked the local peer
  - Every 30 seconds, one additional interested remote peer from the remaining connections is unchoked at random
    - This is called **optimistic unchoke**
    - Allows to replace peers with better upload capacity and bootstrap new peers that do not have any piece to share

Centralized model: BitTorrent

- Advantages
  - Very good download performance
    - Slow nodes do not slow down other nodes
    - Proficient in utilizing partially downloaded files
  - Discourages ‘freeloading’ by rewarding fastest uploaders
  - ‘Rarest Piece First’ extends the lifetime of swarm, increasing the likelihood all pieces are available
  - Users use well-known, trusted sources to locate content, as there is not search feature
    - Avoids the pollution problem, where garbage is passed off as authentic content

Centralized model: BitTorrent

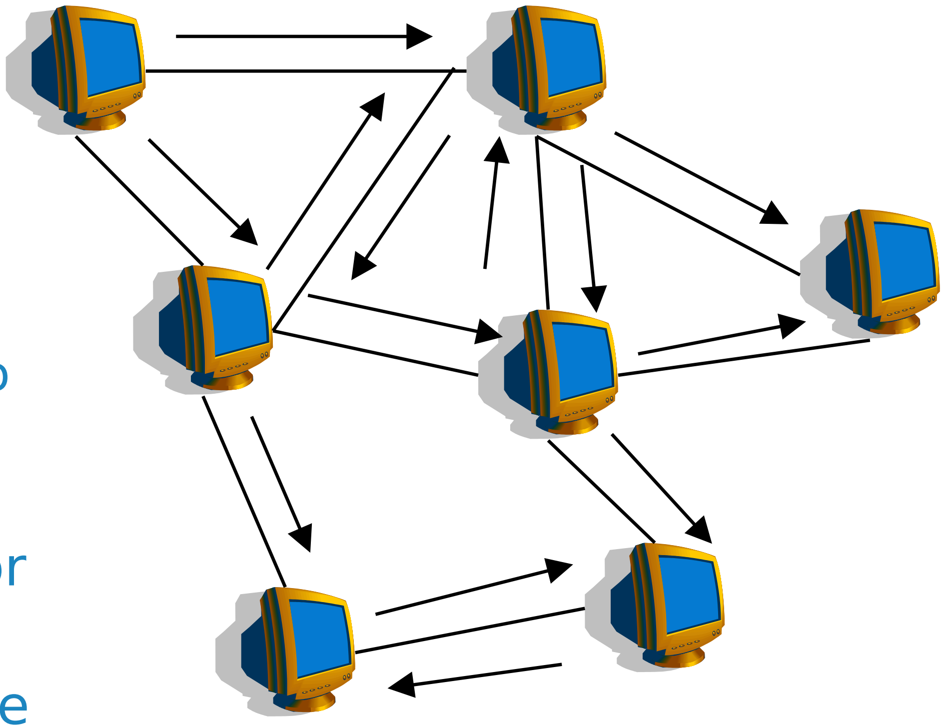
- Disadvantages
  - ↓ All interested peers should be active at same time: performance deteriorates if swarm "cools off"
  - ↓ The centralized tracker is a single point of failure
    - New nodes cannot enter the swarm if tracker goes down
    - BitTorrent variants without a centralized tracker
      - e.g. Vuze (former Azureus)
      - Official client also supports distributed tracking
        - » Mainline DHT (based on Kademlia)
  - ↓ Lack of search feature
    - Users need to resort to out-of-band search

Contents

- Introduction
- **Unstructured P2P systems**
  - Centralized model
  - **Decentralized model**
  - Hierarchical model
- Structured P2P systems

Decentralized model: Gnutella

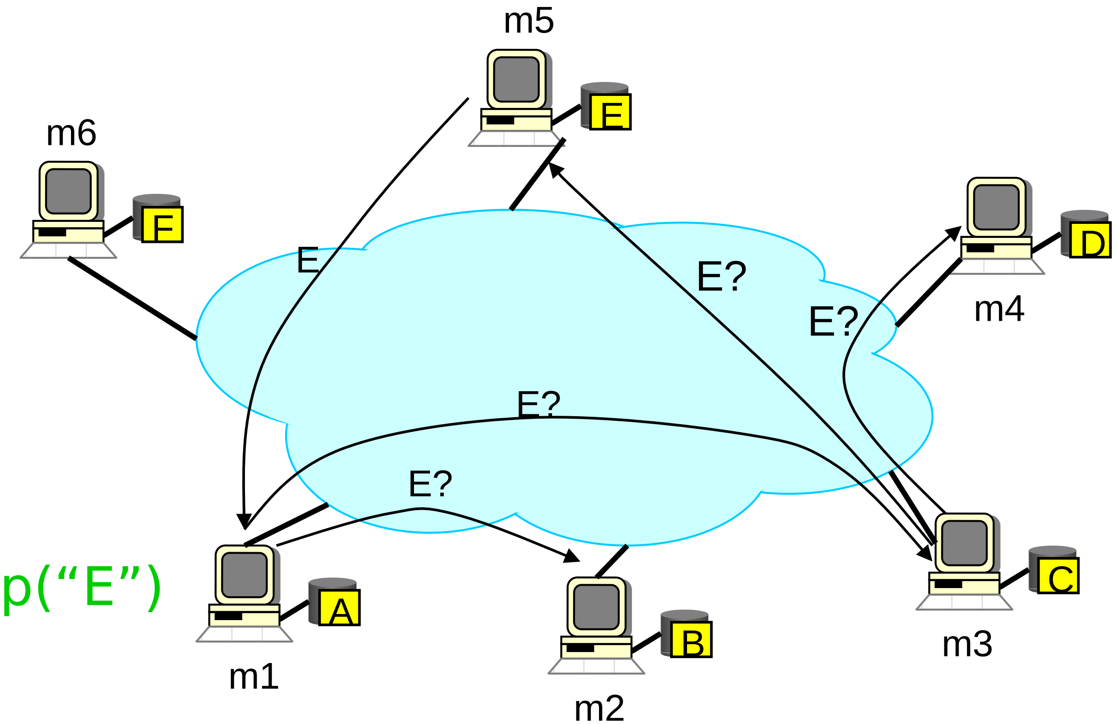
- Based on query **flooding**
- Hot to find a file:
  - Peer sends **Query** descriptor to all neighbors
    - All peers connected to it
  - Neighbors recursively multicast the descriptor
  - Eventually a peer that has the file receives the descriptor, and sends back a **QueryHit** (retracing the path of Query)
  - Peer requests file to this



Decentralized model: Gnutella

- Example
  - Assumption: m1’s neighbors are m2 and m3; m3’s neighbors are m4 and m5;...

Lookup(“E”)



Decentralized model: Gnutella

- Advantages
  - Highly decentralized
  - Location service distributed over peers
    - Peers have similar responsibilities
  - No centralized directory
    - System harder to shut down
- Disadvantages
  - ↓ Slow information discovery
  - ↓ Scalability problem due to <u>excessive query traffic</u>
    - Queries for contents that are not widely replicated must be sent to a large fraction of peers

Decentralized model: Gnutella

- Need some mechanisms to control flooding

A. Add **TTL (Time-to-Live)** to each message
  - Number of hops that message may travel before giving up (default in Gnutella is 7)
  - How to adjust properly TTL value?
    - For popular objects, small TTLs suffice
    - For rare objects, large TTLs are necessary
  - Number of messages grow exponentially with TTL

B. Expanding ring
  - Start querying with TTL=1

Decentralized model: Gnutella

- Need some mechanisms to control flooding

C. Random walk: Ask one neighbor randomly, who asks one neighbor, etc.
   - Reduces the number of flooded messages
     - Tradeoff: longer delay, fewer messages
   - Multiple-walker random walk
     - Initiate several random walks in parallel

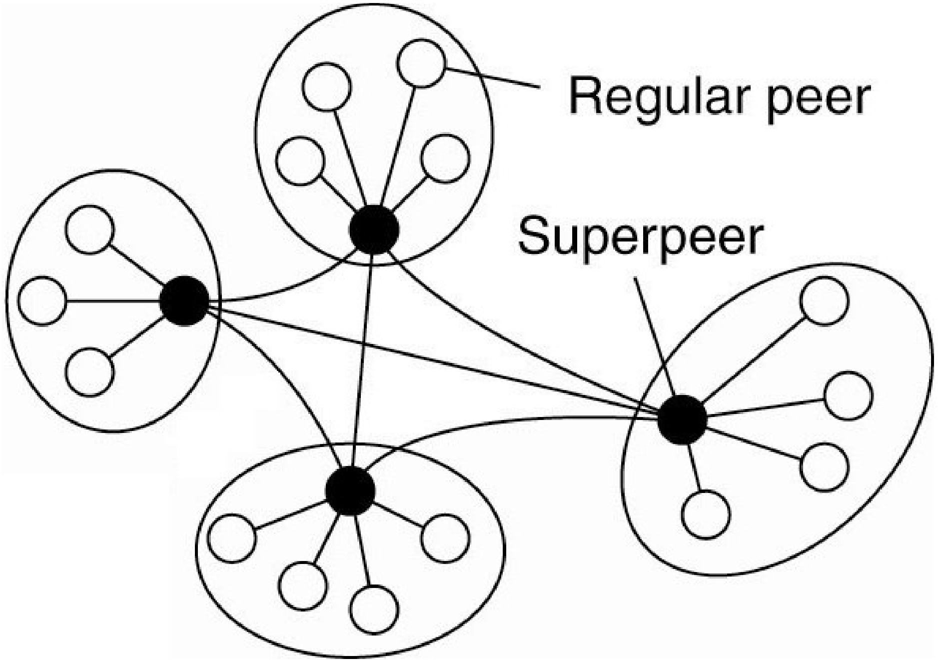
D. State keeping: Keep track of recently routed messages
   - Avoids re-broadcasting Query messages
   - Can choose another neighbor if random walk used

Contents

- Introduction
- **Unstructured P2P systems**
  - Centralized model
  - Decentralized model
  - **Hierarchical model**
- Structured P2P systems

Hierarchical model: FastTrack

- Few special nodes (**super-peers**)
- A peer is either a super-peer or assigned to one
- Super-peer knows the files in all its peers
- Peer queries its super-peer, which may query other super-peers using flooding
- Download among peers



Hierarchical model: FastTrack

- Advantages
  - Combines advantages of centralized and decentralized models
  - Directory is decentralized
  - Less flooding than decentralized model
- Disadvantages
  - ↓ Super-peers can suffer the same problems that the centralized directory
  - ↓ Scalability problems by using flooding
- Used in KaZaA, Skype

Contents

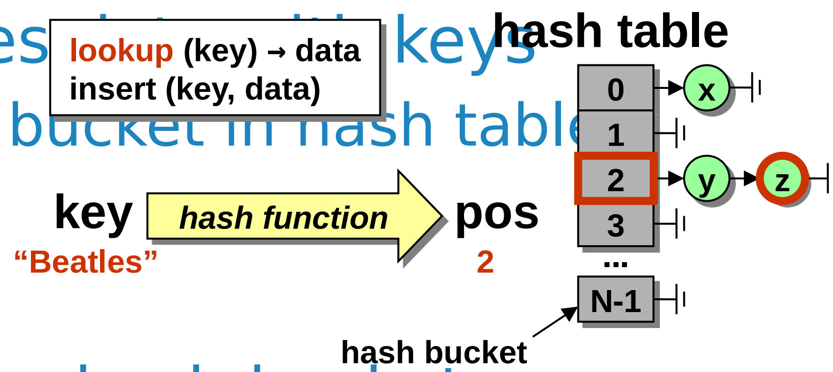
- Introduction
- Unstructured P2P systems
- **Structured P2P systems**

Distributed Hash Tables (DHT)

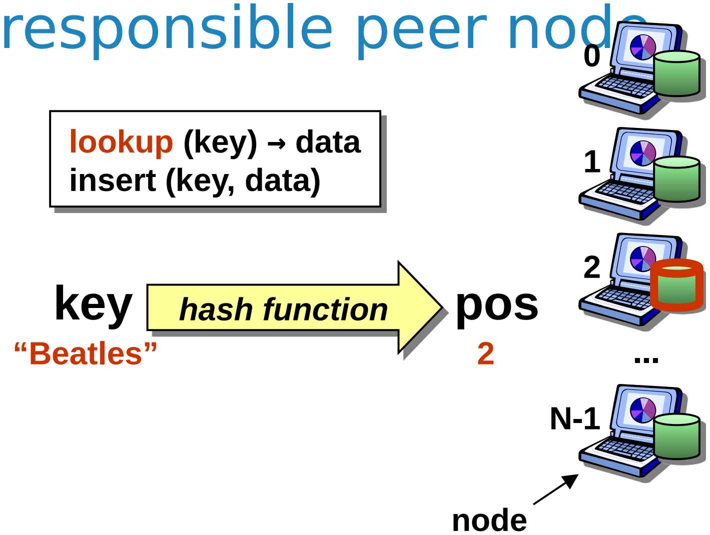
- Goals
  - Make sure that an item identified is always found
    - Directed search: Organize peers following a specific distributed data structure + deterministic mapping of data items to peers based only on their ID
  - Distribute the responsibilities ‘evenly’ among the existing peers
  - Adapt to peers joining or leaving (or failing)
  - Scale to large number of peers
- Examples
  - Chord, Kademlia, Pastry, Tapestry, CAN

Distributed Hash Tables (DHT)

- A hash table associates [illegible] keys
  - Key is hashed to find bucket in hash table



- In a DHT, nodes are the hash buckets
  - Key is hashed to find responsible peer node



Contents

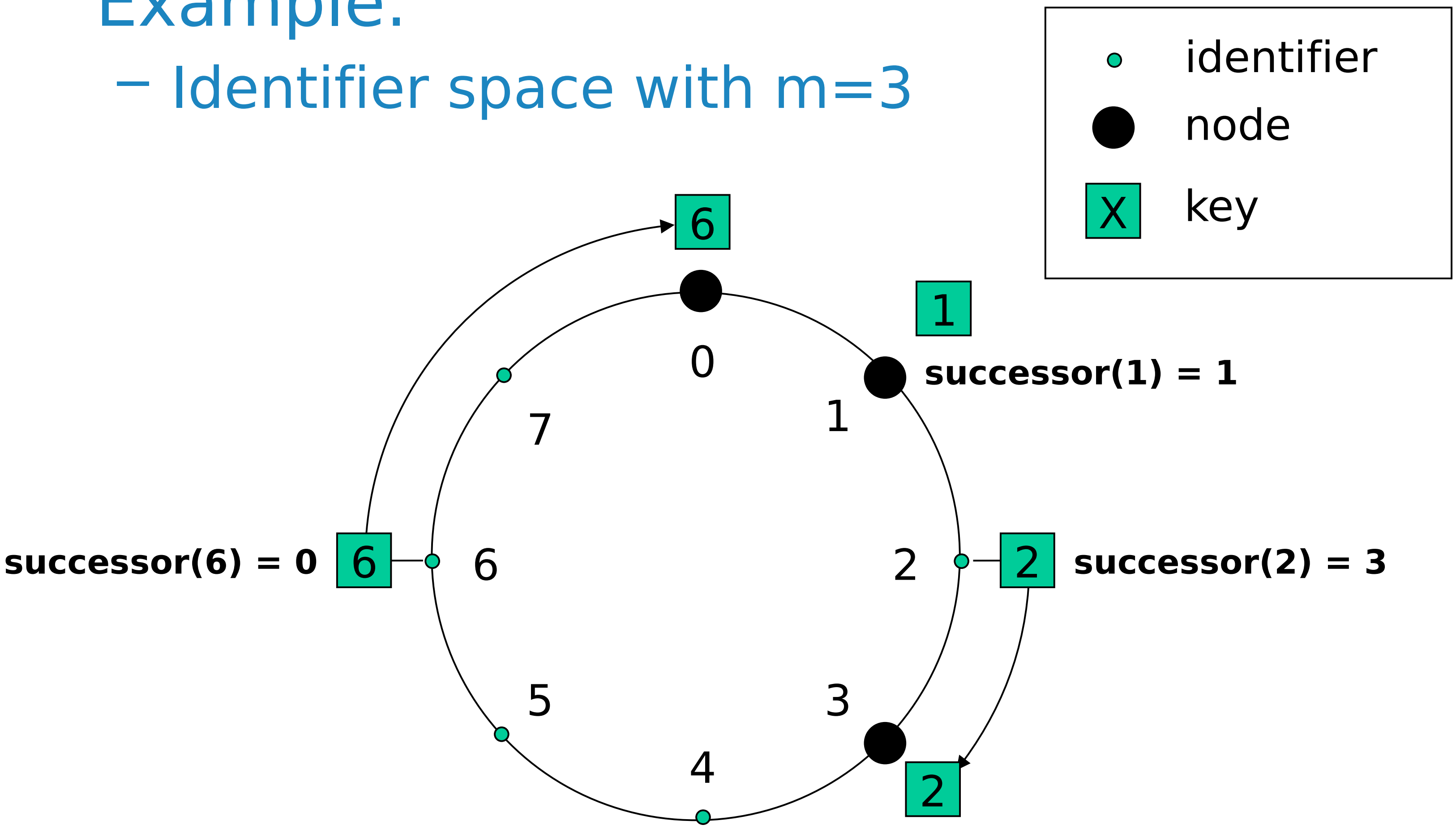
- Introduction
- Unstructured P2P systems
- **Structured P2P systems**
  - **Chord**
  - Kademlia

Chord

- m bit identifier space for both keys and nodes
  - Identifiers are ordered on an identifier **circle** modulo 2^m (they go from 0 to 2^m-1)
  - Map nodes & keys to identifiers with SHA-1 hash function
- How to map key IDs to node IDs?
  - Use <u>consistent hashing</u>
    - Map key IDs to nodes with 'closest' IDs
  - A key identified by ID is stored on the **successor** node of ID (node with next higher ID)
    - Each node is responsible for $O(K/N)$ keys

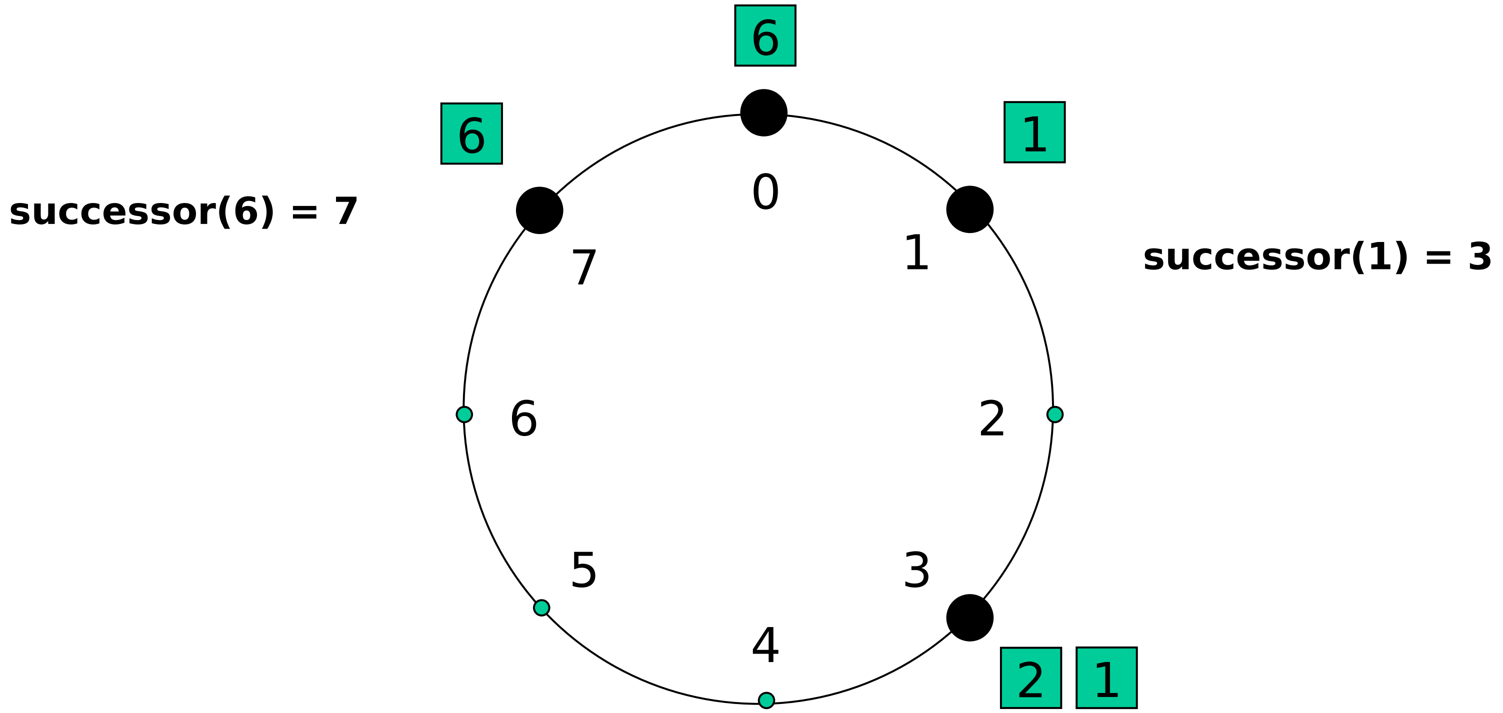
Chord: Item insertion

- Example:
  - Identifier space with m=3



Chord: Item insertion

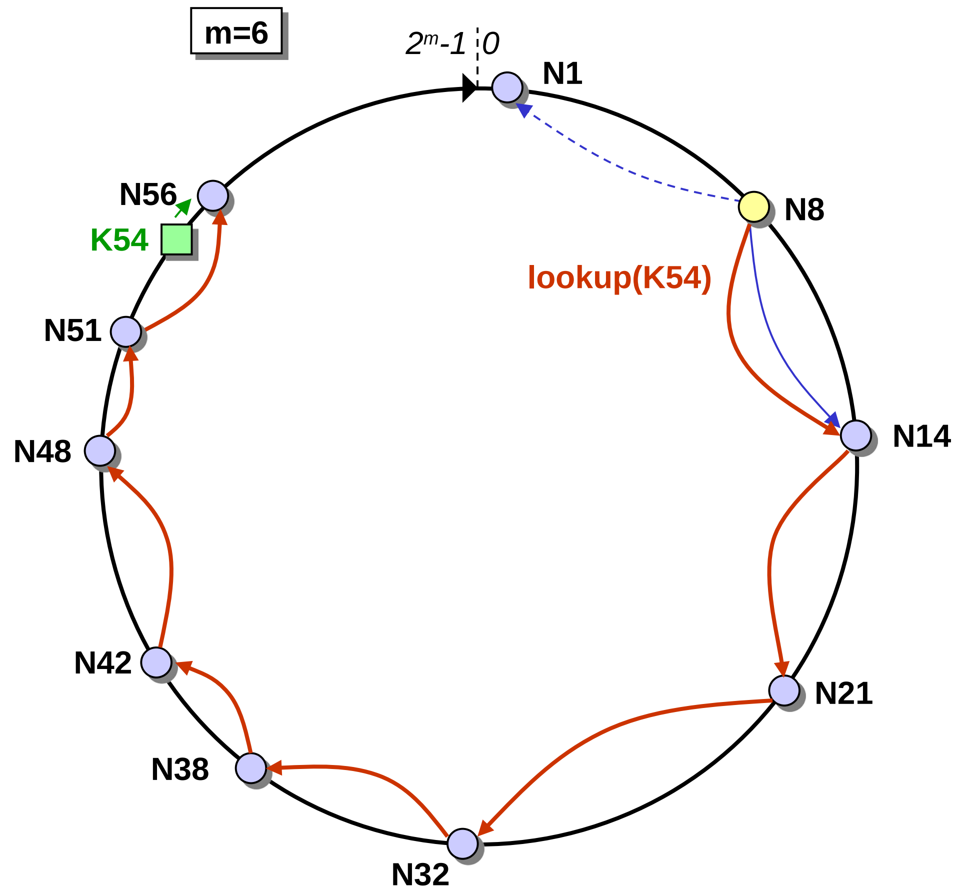
- Item management with node joins and departures



Chord: Item lookup

A. Basic Chord

- Each node knows only two other nodes
  - Successor
  - Predecessor (for ring management)
- Lookup by forwarding requests around the ring through successor pointers
- Requires *O(N)* hops

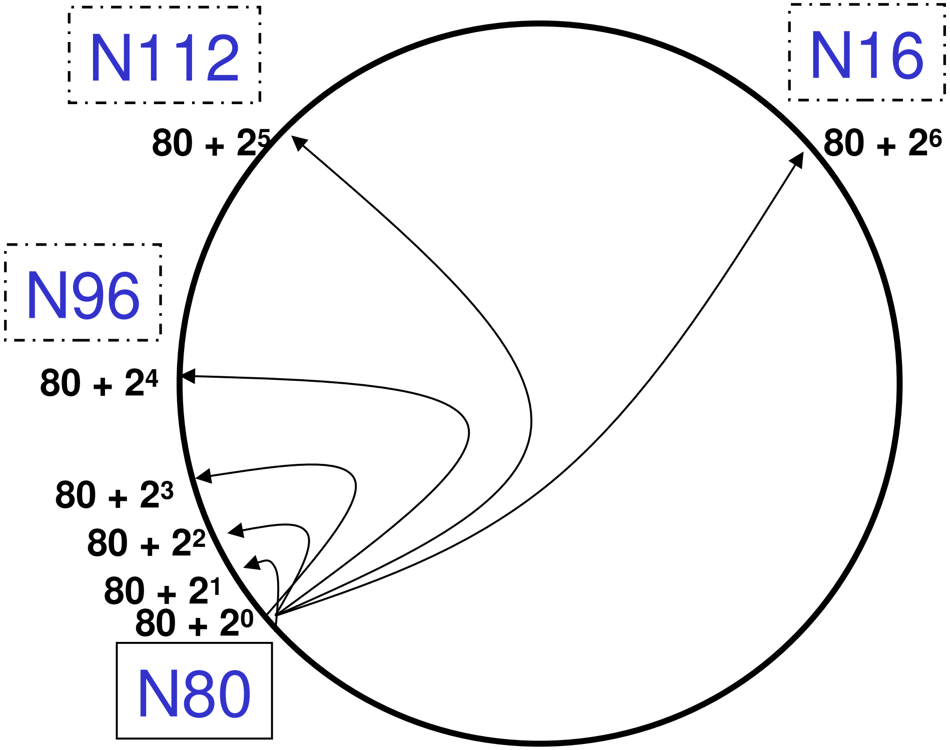


Chord: Item lookup

B. Finger tables

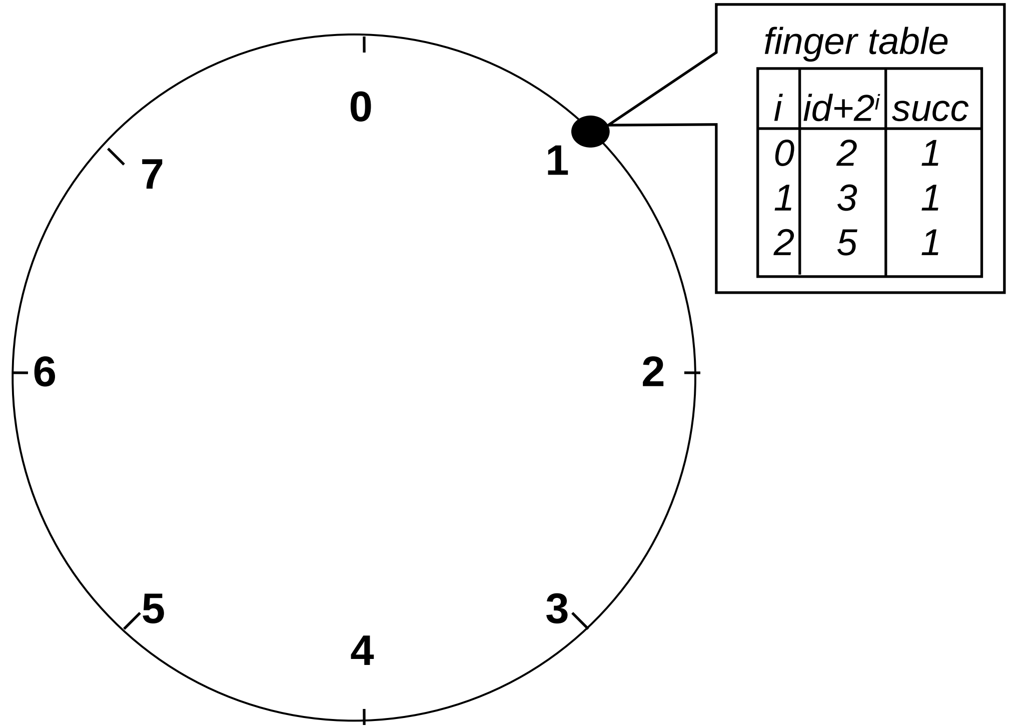
for accelerating lookup

- Every node knows m other nodes
- Entry i in the finger table of node n points to node $n + 2^i$ (if any) or to its successor
- Increase hop distance exponentially
- Lookups take $O(Log\ N)$ hops



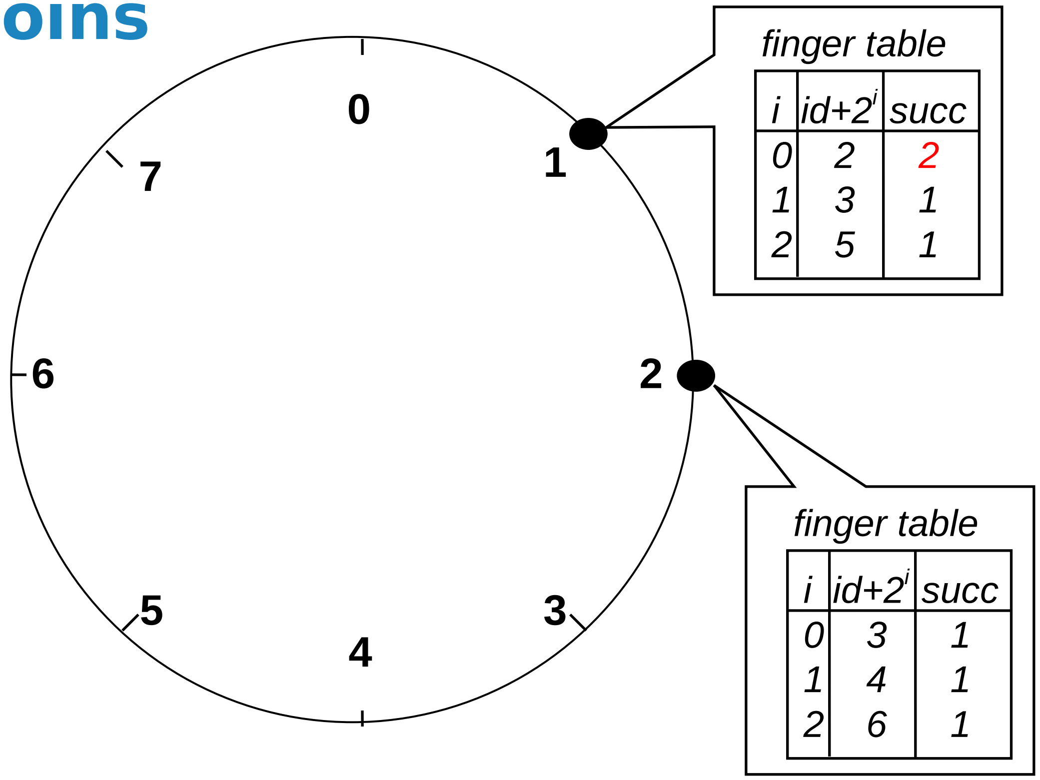
Chord: Managing finger tables

- Example:
  - Node n1:(1) **joins**
    - All entries in its finger table are initialized to itself



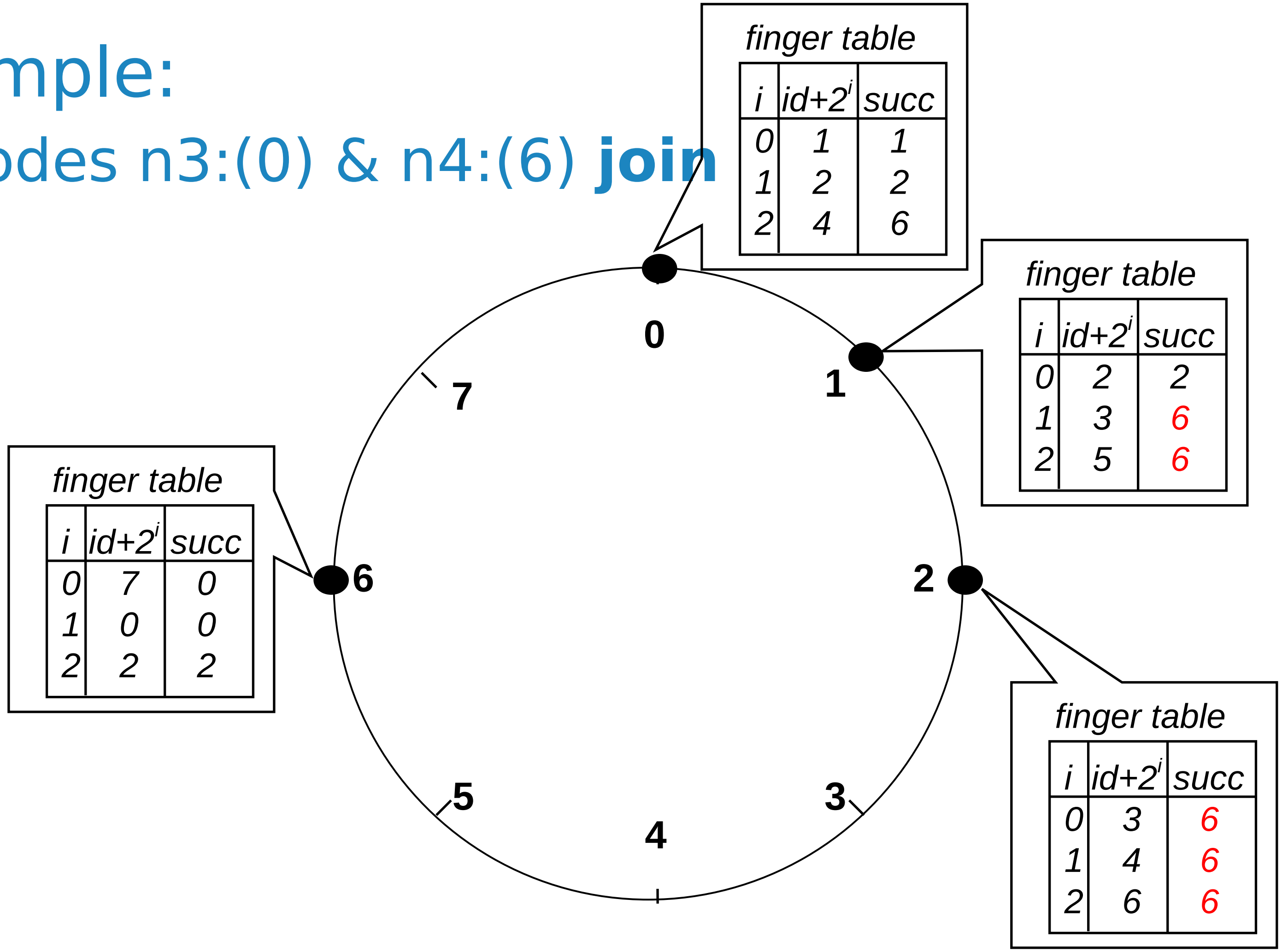
Chord: Managing finger tables

- Example:
  - Node n2:(2) **joins**



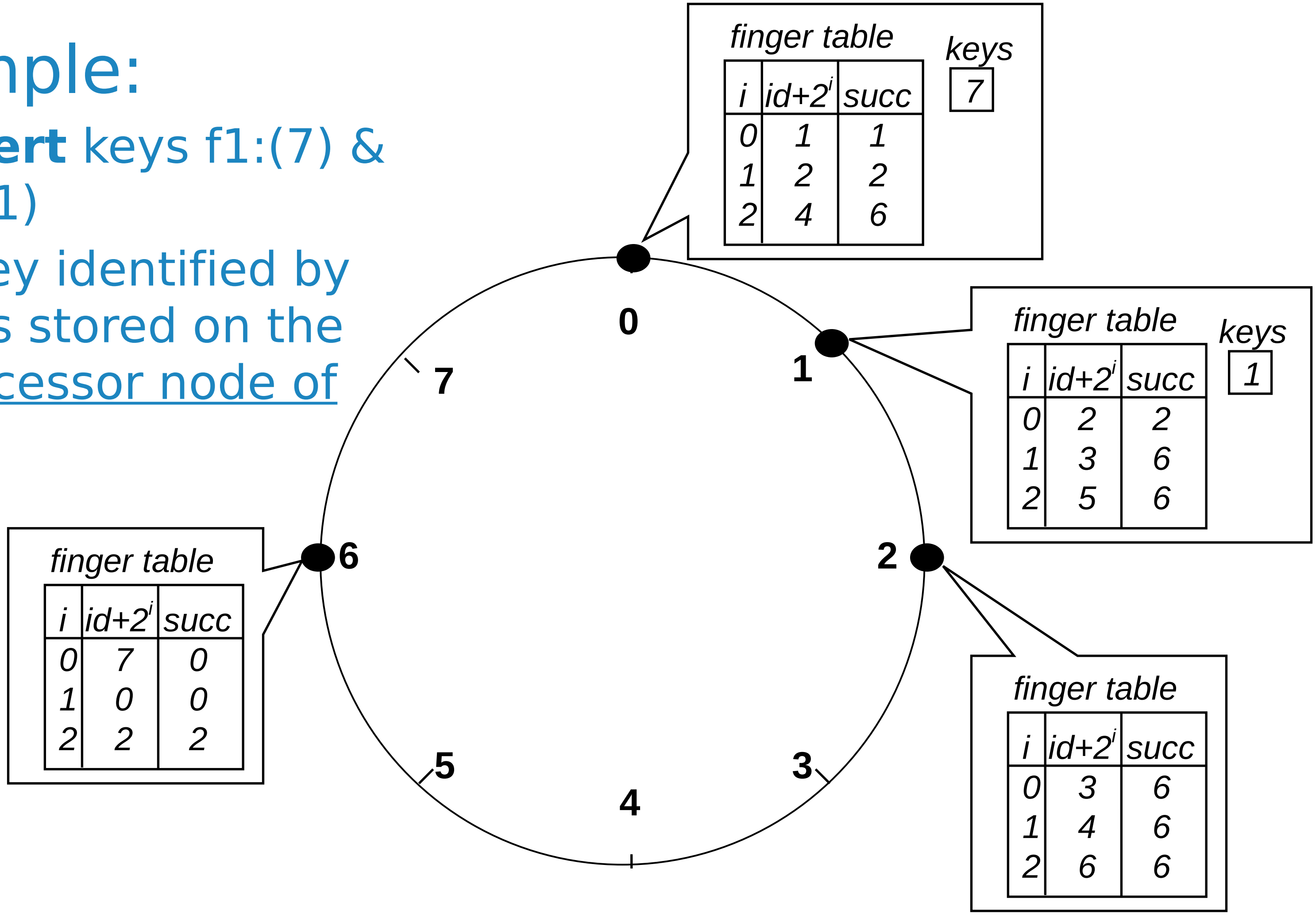
Chord: Managing finger tables

- Example:
  - Nodes n3:(0) & n4:(6) **join**



Chord: Managing finger tables

- Example:
  - **Insert** keys f1:(7) & f2:(1)
  - A key identified by ID is stored on the <u>successor node of ID</u>



Node 0 — finger table (keys: 7)

i	id+2^i	succ
0	1	1
1	2	2
2	4	6

Node 1 — finger table (keys: 1)

i	id+2^i	succ
0	2	2
1	3	6
2	5	6

Node 2 — finger table

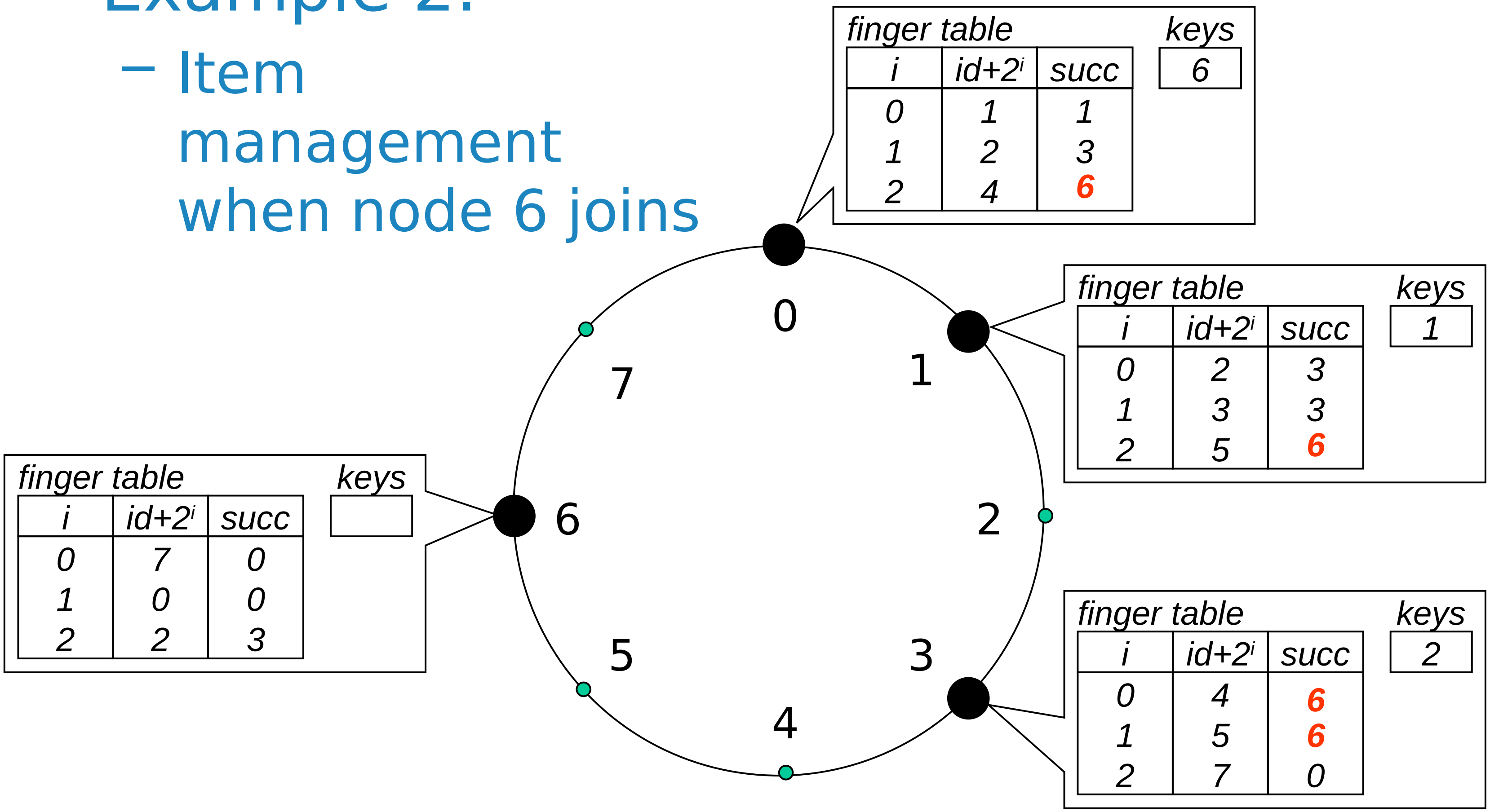
i	id+2^i	succ
0	3	6
1	4	6
2	6	6

Node 6 — finger table

i	id+2^i	succ
0	7	0
1	0	0
2	2	2

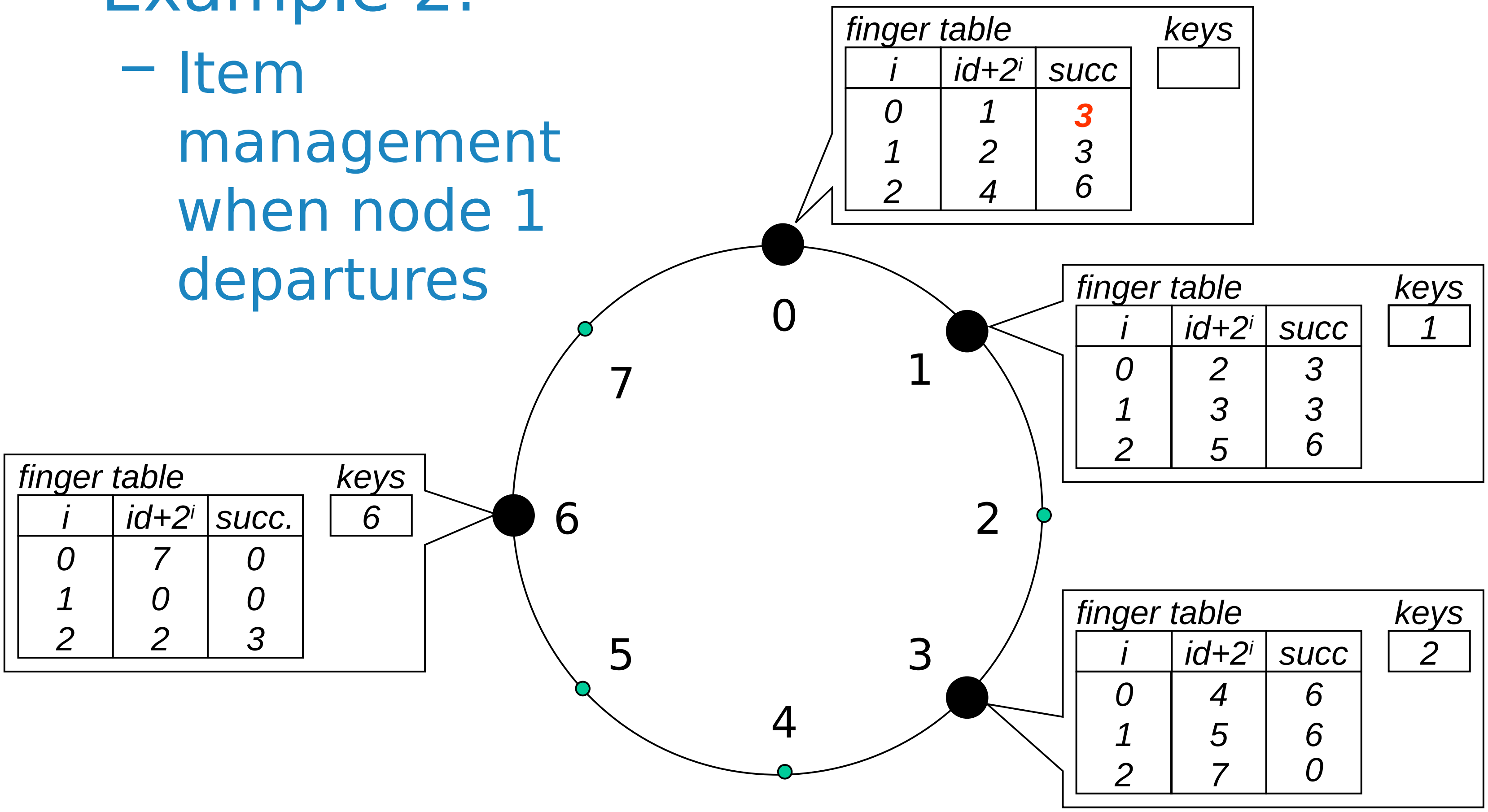
Chord: Managing finger tables

- Example 2:
  - Item management when node 6 joins



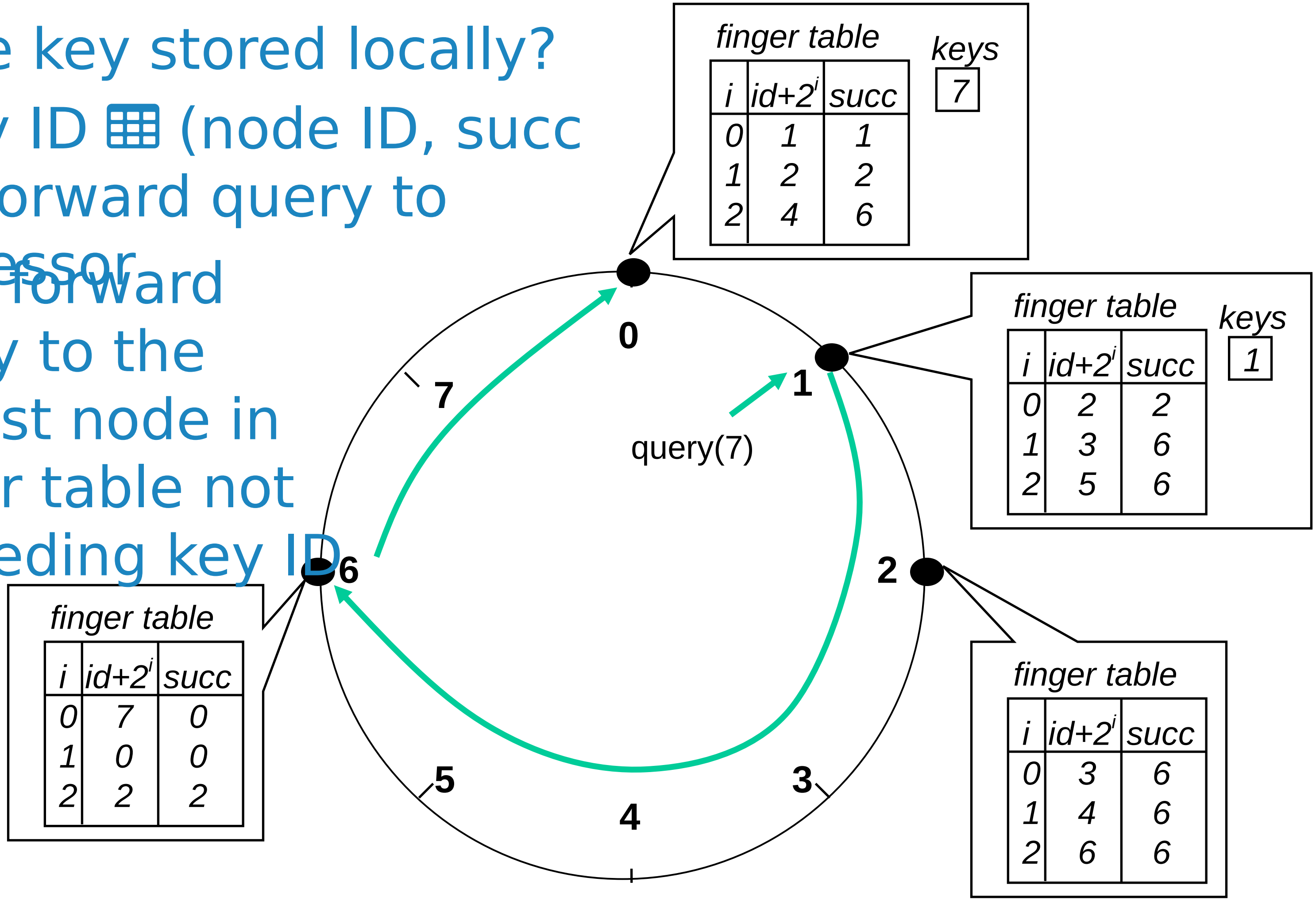
Chord: Managing finger tables

- Example 2:
  - Item management when node 1 departures



Chord: Item lookup with finger tables

a) Is the key stored locally?
b) If key ID ∈ (node ID, succ ID], forward query to successor
c) Else, forward query to the largest node in finger table not exceeding key ID



Chord: Item lookup with finger tables

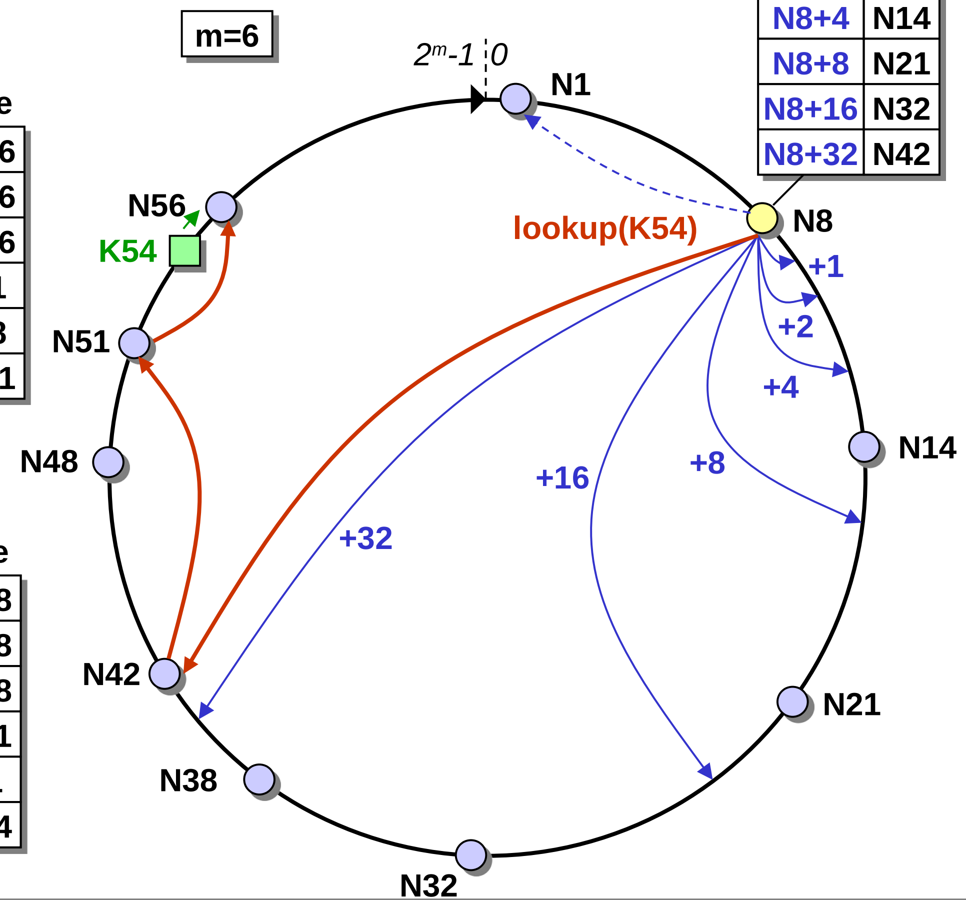
- Example 2:

Finger table

N51+1	N56
N51+2	N56
N51+4	N56
N51+8	N1
N51+16	N8
N51+32	N21

Finger table

N42+1	N48
N42+2	N48
N42+4	N48
N42+8	N51
N42+16	N1
N42+32	N14



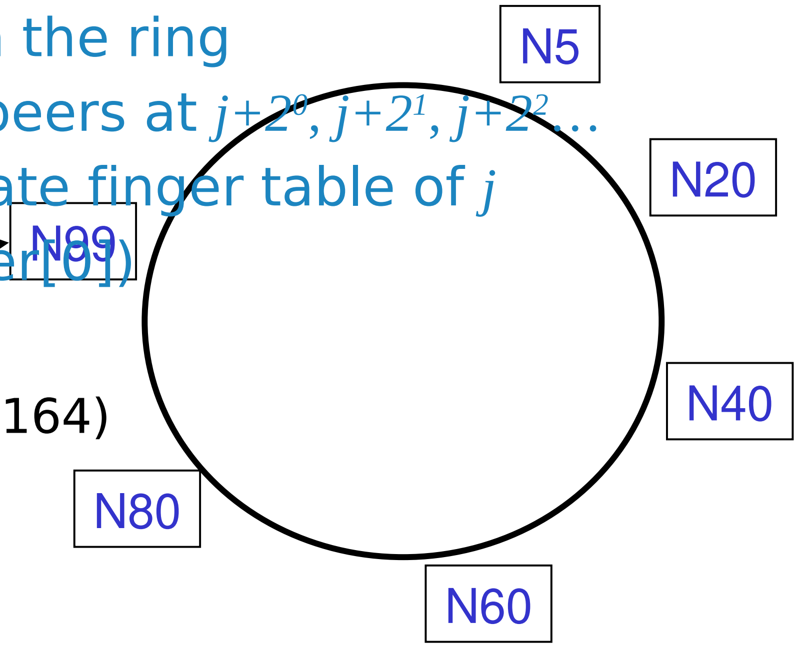
Finger table

N8+1	N14
N8+2	N14
N8+4	N14
N8+8	N21
N8+16	N32
N8+32	N42

Chord: Node joining

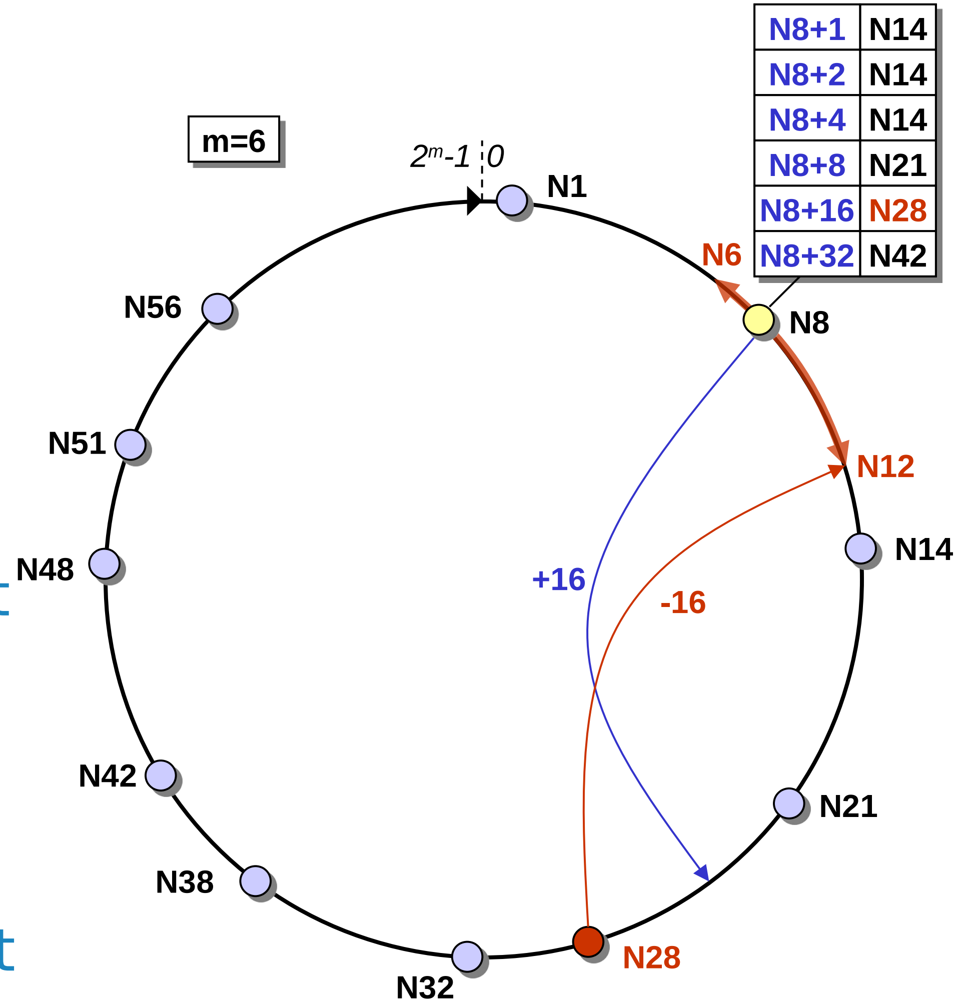
A. **Basic** process for joining the ring (3 steps):

1. Initialize fingers and predecessor of new node j
   - Locate any node n in the ring
   - Ask n to lookup the peers at $j+2^0, j+2^1, j+2^2 \ldots$
   - Use results to populate finger table of j
   - pred(j) = pred(j.finger[0])
   - pred(j.finger[0]) = j



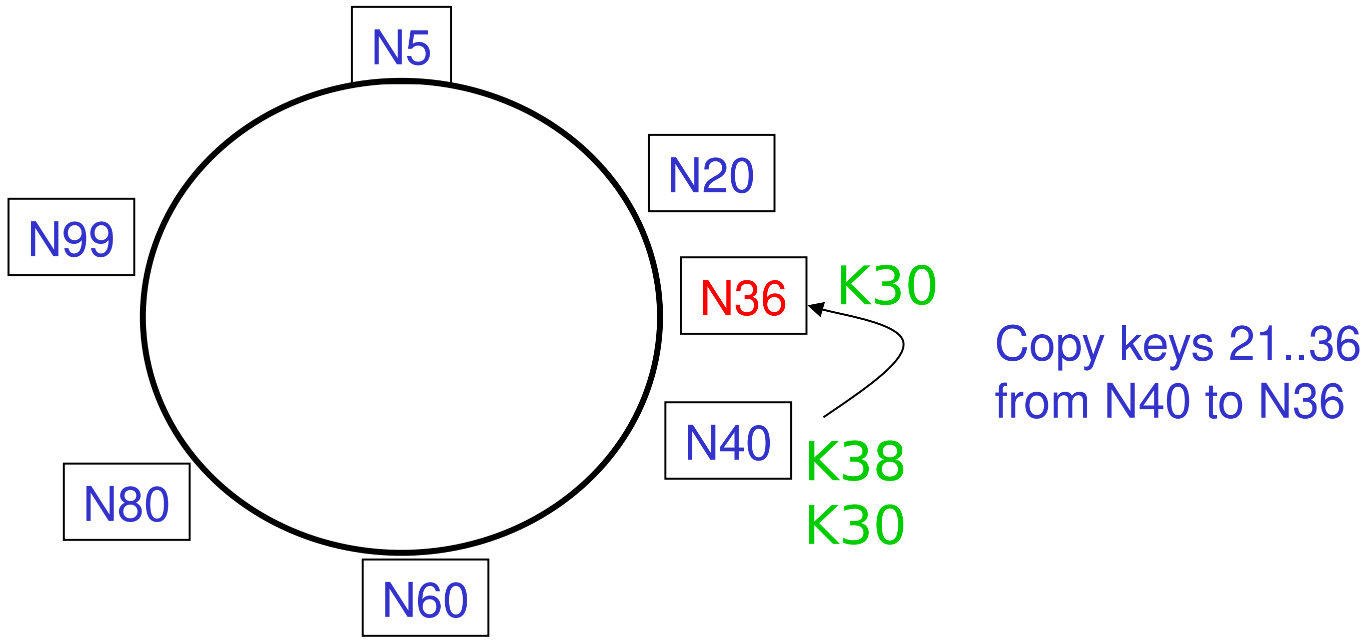
Chord: Node joining

2. Update finger tables of existing nodes
   - For each entry i in the finger table, new node j calls *update* function on existing nodes that must point to j
     - Nodes in the ranges $[pred(j)-2^i+1, j-2^i]$
   - $O(\log N)$ nodes must be updated



Chord: Node joining

3. Transfer keys responsibility
   – Connect to successor and transfer keys in the range from successor to new node



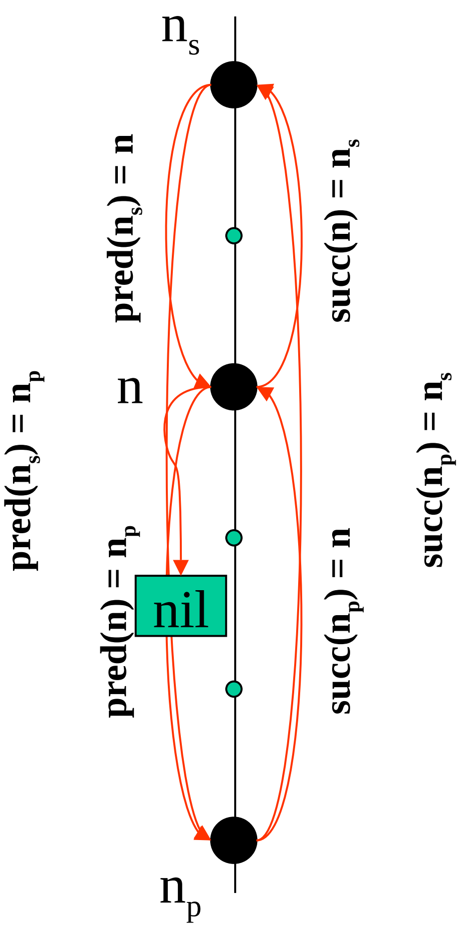
Chord: Node joining

B. Stabilization: Less aggressive mechanism to deal with concurrent joins

- The goal is to keep successor pointers up to date
  - This is sufficient to guarantee correctness of lookups (although they may be slower)
- Finger entries are updated in a lazy fashion

1. Join only initializes the finger to successor node
2. All nodes run a stabilization procedure that periodically verifies successor and predecessor
3. All nodes also refresh periodically finger

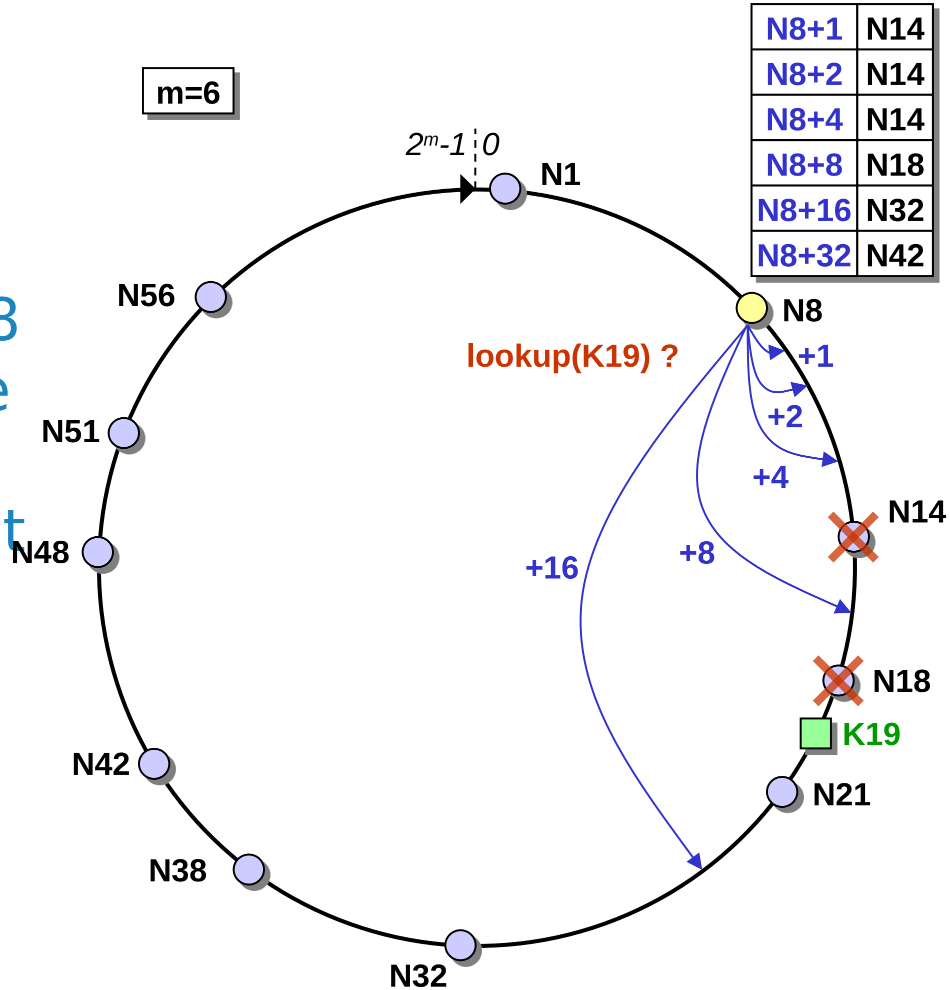
Chord: Stabilization



- **n joins**
  - predecessor = nil
  - n acquires n_s as successor via some node in the ring
- **n runs stabilization procedure**
  - n asks n_s for its predecessor
  - n_s responds ($pred(n_s) = n_p$) to n
  - n confirms n_s as successor and notifies n_s that it is its new predecessor
    - otherwise, n updates its successor and stabilizes again
  - n_s checks and acquires n as its predecessor
  - n_s transfers keys in the range to n
- **n_p also runs stabilization procedure**
  - n_p asks n_s for its predecessor (now n)
  - n_p acquires n as its successor
  - n_p notifies n that it is its new predecessor
  - n acquires n_p as its predecessor

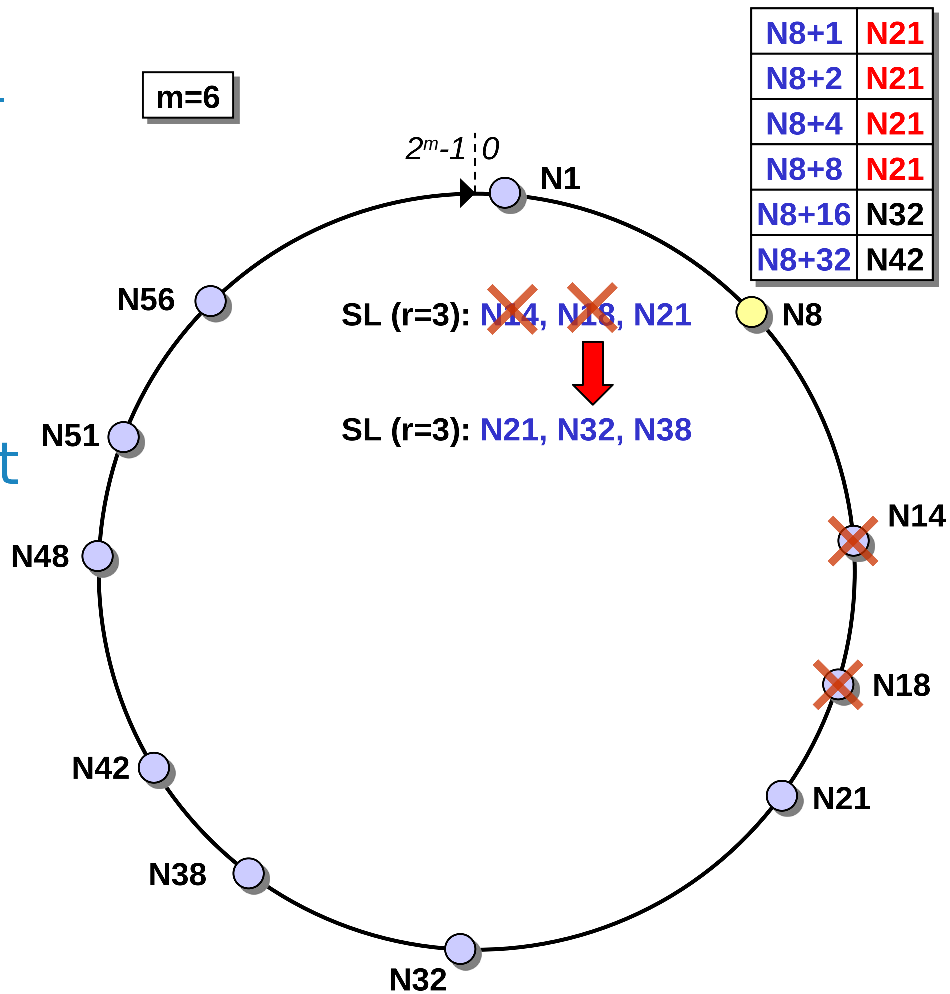
Chord: Dealing with node failures

- Failure of nodes might cause incorrect lookups
  - Lookup(K19) fails: N8 returns the first alive node it knows (N32) instead of the correct successor (N21)



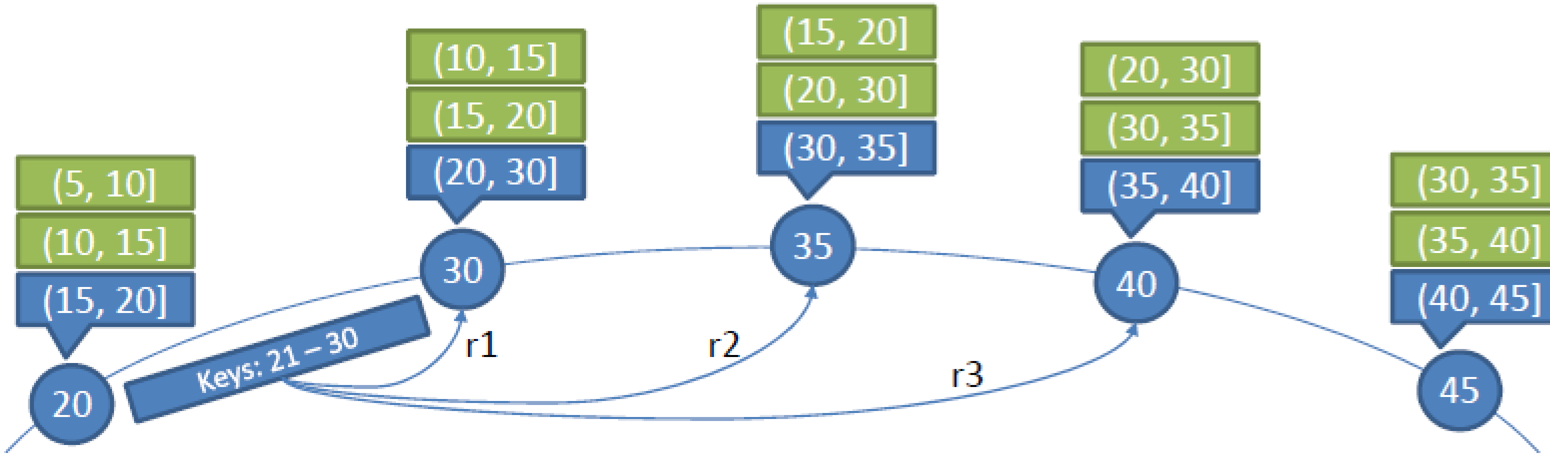
Chord: Dealing with node failures

- Solution: successor-list
  - Each node knows its **r** immediate successors
  - If a node notices that its successor has failed, it replaces it with the first alive process in the list
  - Eventually, stabilize will correct finger table entries and successor-list entries pointing to failed nodes



Chord: Dealing with node failures

- Successor-list can also be used for <u>replication</u>
  - For a replication degree of **d**, a key is stored on the responsible node n and the first d−1 members of n's successor-list (d−1 ≤ r)
  - As nodes join/leave, successor-lists are



Chord

- Advantages
  - Efficient: $O(Log\ N)$ messages per lookup
  - Scalable: $O(Log\ N)$ state per node
  - Robust: survives massive changes in membership

N is the total number of nodes

- Disadvantages
  - ↓ Member joining is complicated
  - ↓ Asymmetric: in- and out- traffic distributions are exactly opposite
    - Incoming traffic cannot be used to reinforce finger table
  - ↓ Finger table rigidity precludes proximity-based routing

SEMINAR PREPARATION – Chordy

- **[Stoica03]** Stoica, I., Morris. R., Liben-Nowell, D., Karger, D.R., Kaashoek, M.F., Dabek, F., Balakrishnan, H., *Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications*, IEEE/ACM Transactions on Networking, Vol. 11, No. 1, pp. 17-32, February 2003

Contents

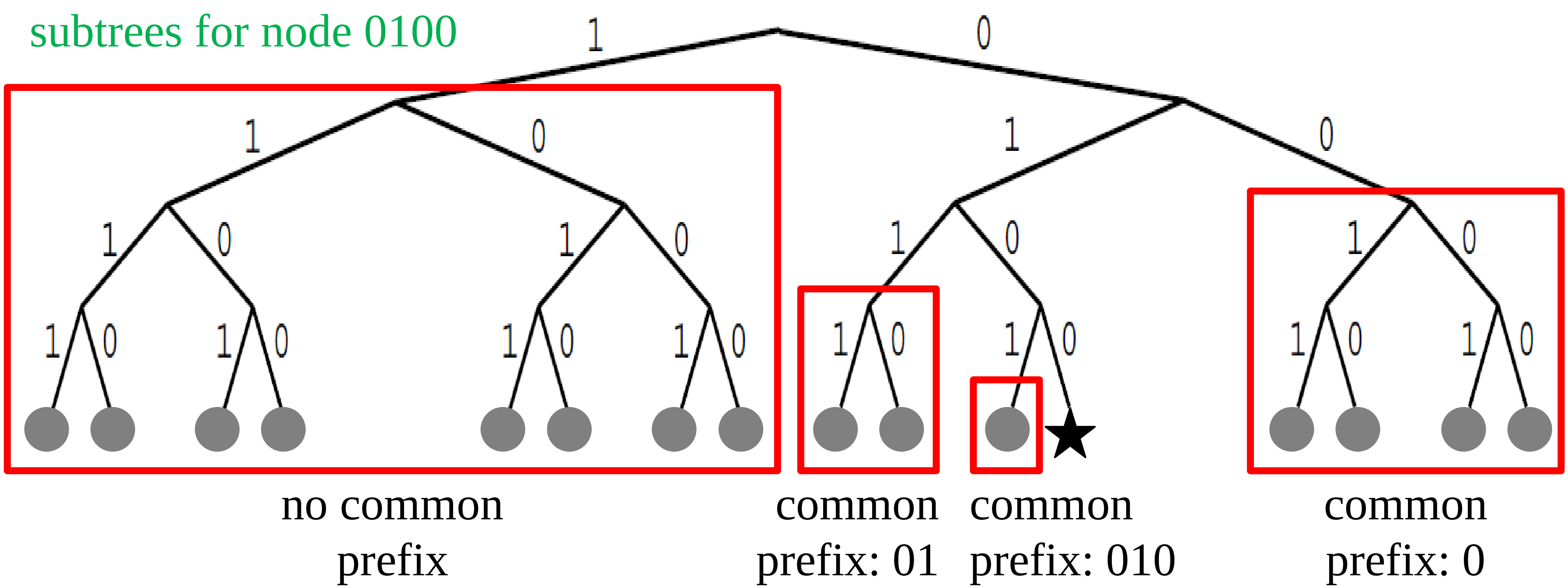
- Introduction
- Unstructured P2P systems
- **Structured P2P systems**
  - Chord
  - **Kademlia**

Kademlia

- 160-bit identifier space for both keys & nodes
  - Map nodes & keys to identifiers with SHA-1 hash
- How to map key IDs to node IDs?
  - Use consistent hashing
    - Map key IDs to nodes with ‘closest’ IDs
  - The closeness between two objects measured as their **bitwise XOR** interpreted as an integer
    - d(x, y) = x XOR y
  - XOR is symmetric (unlike Chord)
    - d(x, y) = d(y, x) x, y

Kademlia: Binary tree

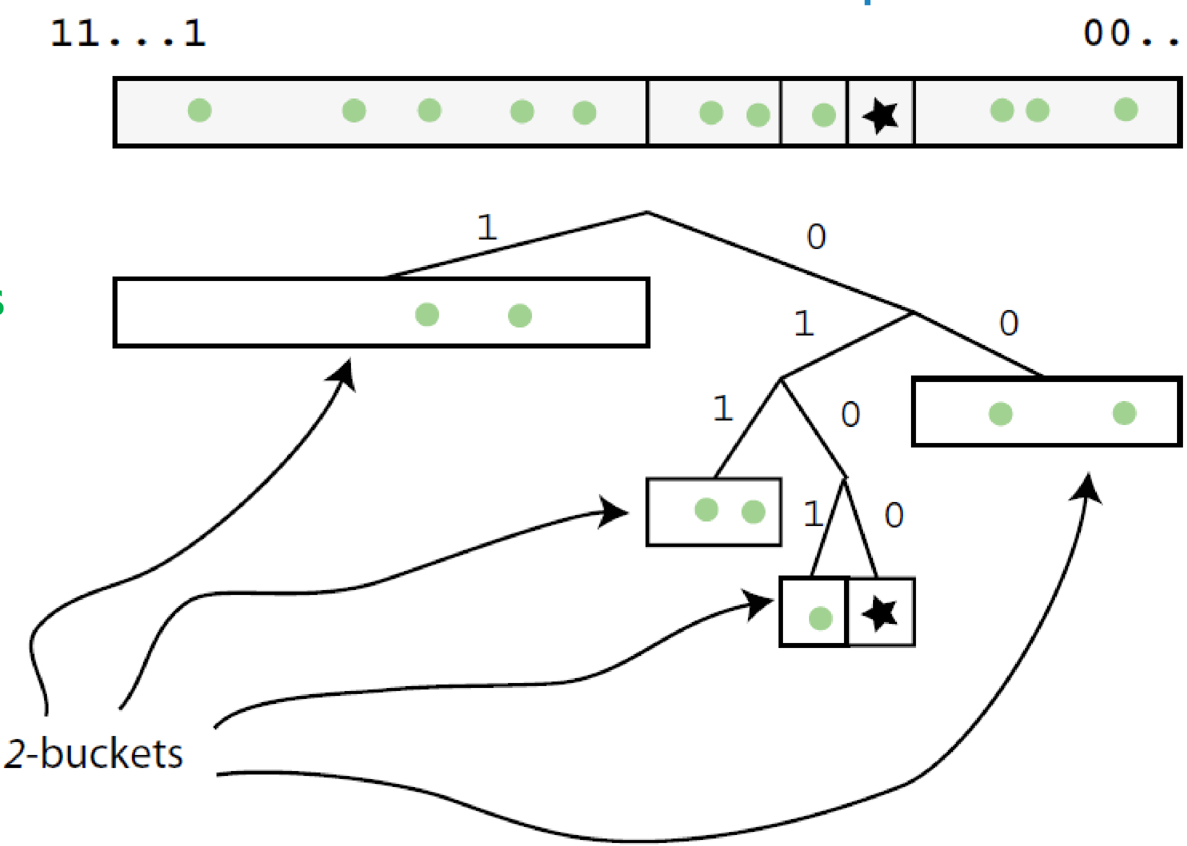
- Treat nodes as leaves in a binary tree
  - The prefix of the node ID determines its position
  - For any given node, the tree is divided into a series of successively lower subtrees that do not contain the node



Kademlia: Node state

- For each of its subtrees, each node keeps a <u>k-bucket</u>: list of references for <u>up to k</u> nodes fror

2-buckets for node 0100



Kademlia: Node state

- Formally, ∀i 0≤i<[#bits in node ID], k-bucket$_i$ in node P keeps [@IP, UDP port, node ID] triples for up to k nodes of distance between $\mathbf{2^i}$ and $\mathbf{2^{i+1}}$ from P
  - k is normally 20
  - k-buckets are kept sorted: most-recently seen at the tail

k-buckets

i	distance	node-reference$_0$		node-reference$_{k-1}$
0	$[2^0,2^1)$	[@IP, UDP port, node ID]	…	[@IP, UDP port, node ID]
1	$[2^1,2^2)$	[@IP, UDP port, node ID]	…	[@IP, UDP port, node ID]
…	…	…	…	…
i	$[2^i,2^{i+1})$	[@IP, UDP port, node ID]	…	[@IP, UDP port, node ID]
…	…	…	…	…
15	$[2^{159},2^{160}$	[@IP, UDP port, node	…	[@IP, UDP port, node

Kademlia: Node state

- When P receives any message from Q, P updates the k-bucket that corresponds to Q:
  1. If Q is already in the k-bucket, move it to the tail
  2. Otherwise, if the k-bucket has fewer than k entries, insert Q at the tail
  3. Otherwise, ping the least-recently seen node in the k-bucket: if responds, move it to the tail and discard Q
  4. Otherwise, evict it from the k-bucket and insert Q at the tail

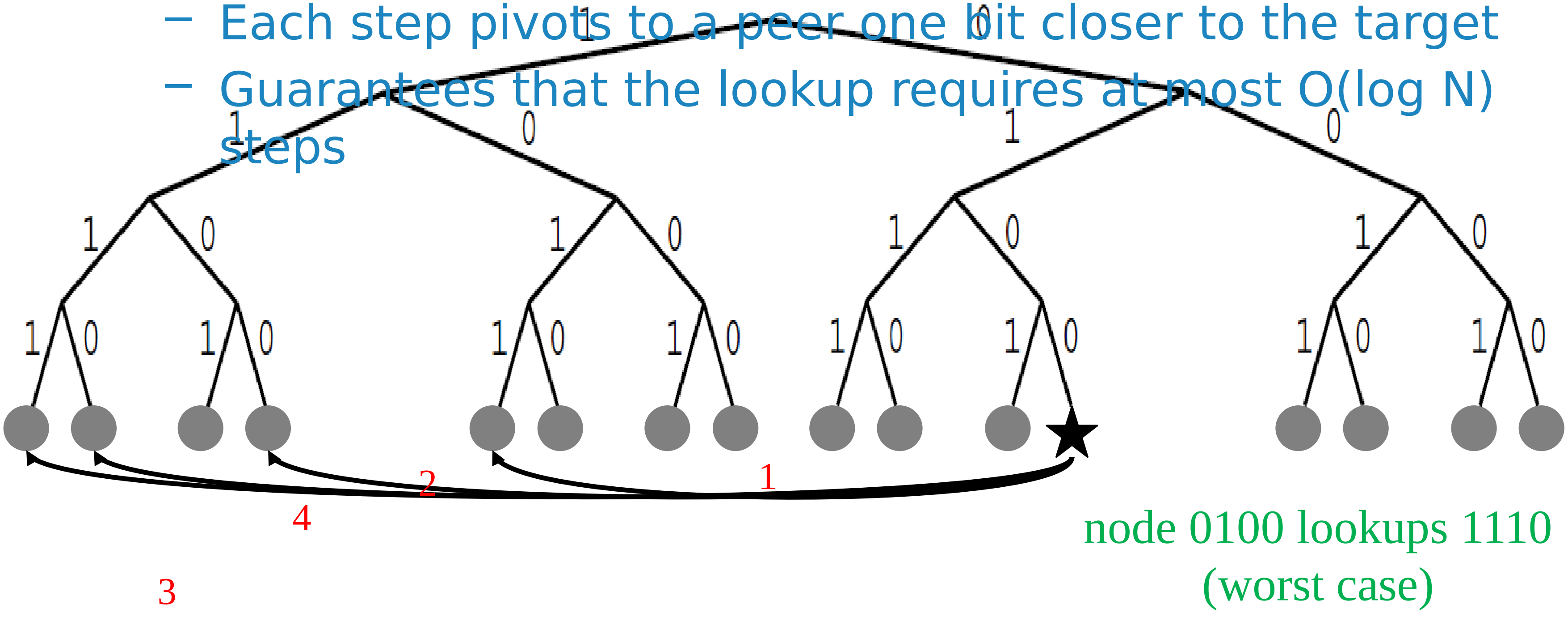
  ⇒ Keeps the oldest live nodes in the k-bucket
    - Maximizes probability that those nodes will remain online
    - Avoids some DoS attacks, as k-buckets cannot be flushed by flooding the system with new nodes

Kademlia: Protocol

- The Kademlia protocol consists of four RPCs:
  1. PING: probes the recipient to see if it is online
  2. STORE: the recipient stores locally the key/value pair contained in the message
  3. FIND_NODE: the recipient returns [@IP, UDP port, node ID] triples for the k nodes it knows closest to the target ID
     - These triples can come from a single k-bucket or multiple k-buckets if the closest k-bucket is not full
  4. FIND_VALUE: behaves like FIND_NODE with one exception: if the recipient has received a STORE RPC for the key, it just returns the

Kademlia: Node lookup

- Basic idea: **Iterative** routing by prefix-matching
  - Origin node is responsible for the entire lookup process
  - Each step pivots to a peer one bit closer to the target
  - Guarantees that the lookup requires at most O(log N) steps



node 0100 lookups 1110
(worst case)

Kademlia: Node lookup

- Use <u>parallel routing</u> to speed up lookups
- Node P performs a lookup of target key T

1. P picks ➔ nodes from its non-empty k-bucket closest to T (if that bucket has fewer than ➔ entries, P just takes the closest nodes it knows)
   - ➔ parameter (=3) configures the degree of concurrency
2. P creates a results list of the k closest nodes to T that it is aware of (k-candidate list) and initially populates it with the first ➔ nodes selected
3. P sends parallel, asynchronous FIND_NODE

Kademlia: Node lookup

- Node P performs a lookup of target key T
  4. Each recipient of FIND_NODE returns to P the k closest nodes to T that it is aware of
  5. On receiving a reply, P uses those nodes to update its k-candidate list so that it holds the k closest nodes to T that it knows at this stage
     - If any of the → nodes fails to reply, it is removed from the k-candidate list
  6. P selects another → nodes from the k-candidate list and sends FIND_NODE to each in parallel
     - The only condition for this selection is that those nodes have not already been contacted

Kademlia: Node lookup

- Node P performs a lookup of target key T
  7. This continues until replied nodes are not closer to T than the k-candidates and P has queried and gotten responses from all the k-candidates
     - They are the k active contacts closest to T
- Messages generated due to lookups keep buckets generally fresh
  – If there is not traffic in the range of a k-bucket within an hour, the node <u>refreshes</u> it by looking up a random key within the k-bucket range

Kademlia: Node join

- To join, node P must know a node Q already in the system (a.k.a. bootstrap node)
  1. P inserts Q into the appropriate k-bucket
  2. P performs a node lookup for its own ID to obtain its closest neighbors
     - Lookup goes through Q, the only other node P knows
  3. P refreshes all k-buckets further away than its closest neighbor k-bucket by looking up a random key within each k-bucket range
     - This populates the k-buckets of P and inserts P into the k-buckets of the other nodes

Kademlia: Key/value pairs

- Storing a key/value pair
  - Perform a node lookup (with FIND_NODE RPCs) to locate the k closest nodes to the key and send each of them a STORE RPC
    - Replicates the pair at the k closest nodes to the key
- Finding a key/value pair
  - Perform a node lookup (with FIND_VALUE RPCs)
    - A node receiving FIND_VALUE returns the value (if it has stored it) or the k closest nodes to the key it knows
  - Procedure halts when any node returns the value
    - key/value pair is cached at the closest node to the

Kademlia: Key/value pairs

- Every key/value pair has an associated expiration time
  - Lifetime of new key/value pairs (from original publishers) is 24 hours
    - Original publishers must republish (store again) them every 24 hours
  - Lifetime of cached key/value pairs is exponentially inversely proportional to the number of nodes between the caching node ID and the node closest to the key
    - The longer the distance between the node closest to the key and the caching node ID, the shorter the lifetime

Kademlia: Key/value pairs

- Each key/value pair must be available in the k closest nodes to the key to ensure lookups correctness, even when nodes join or leave
  1. Each node republishes each key/value pair it contains in k closest nodes to the key every hour
     - This compensates for nodes leaving the system
  2. When joining node P is added to k-buckets of the other nodes, they check whether P is closer to any of the stored key/value pairs, and (if it is) replicate them to P
- Nodes republish key/value pairs keeping their remaining time for expiration

Kademlia: Use cases

- BitTorrent distributed tracker
  - key: info-hash of torrent file metadata
  - value: list of peers currently sharing the file
  - get_peers: get peers associated with an info-hash
    - IT_FIND_VALUE(info-hash)
  - announce_peer: announce that the peer is downloading a torrent on a port
    - IT_FIND_NODE(info-hash) + STORE(info-hash, peer) at the K (=8) closest nodes to info-hash
      - They will add the announcing peer to the peer list stored for that info-hash
  - http://bittorrent.org/beps/bep_0005.html

Kademlia: Use cases

- Kad network (used by eMule)
  - Two different types of keys (MD4 128 bits hash):
    - <u>source key</u>: hash of the content of the file
    - value: list of peers currently sharing the file
    - <u>keyword key</u>: hash of each token of the filename
    - value: list of files {name, hash} containing the keyword
  - Source publication
    - Obtain hashes from file content and each token of the filename + IT_FIND_NODE(hash) + PUBLISH_REQUEST (hash, value) at the K (=10) closest nodes to hash
  - Source or keyword search
    - IT_FIND_NODE(hash) + SEARCH_REQUEST(hash)

Kademlia

- Advantages
  - Efficient: $O(Log\ N)$ messages per lookup
  - Scalable: $k\ O(Log\ N)$ state per node
  - Robust: survives massive changes in membership
  - Symmetric in- and out- traffic distributions
  - Asynchronous parallel lookup: avoids slow links
    - Flexibility to route through closer/faster nodes

N is the total number of nodes

- Disadvantages
  - ↓ Key/value pairs have to be refreshed every 24h
  - ↓ Possible *convoy effects* because of bursts of activity (e.g. key/value republishing)

Summary

- P2P: model for high availability & scalability
  - Removes distinction between clients and servers
    - Peers are symmetric in functionality and responsibilities
    - Operation is independent of any central node
    - Peers are organized in an overlay network
  - Two types of P2P systems
    - Unstructured P2P systems: Gnutella, KaZaA, BitTorrent
    - Structured P2P systems: DHT systems: Chord, Kademlia
- Further details:
  - [Tanenbaum]: chapters 2.2.2 and 2.2.3