# 9. Peer-to-Peer Systems

Sistemes Distribuïts en Xarxa (SDX)
Facultat d'Informàtica de Barcelona (FIB)
Universitat Politècnica de Catalunya (UPC)
2024/2025 Q2

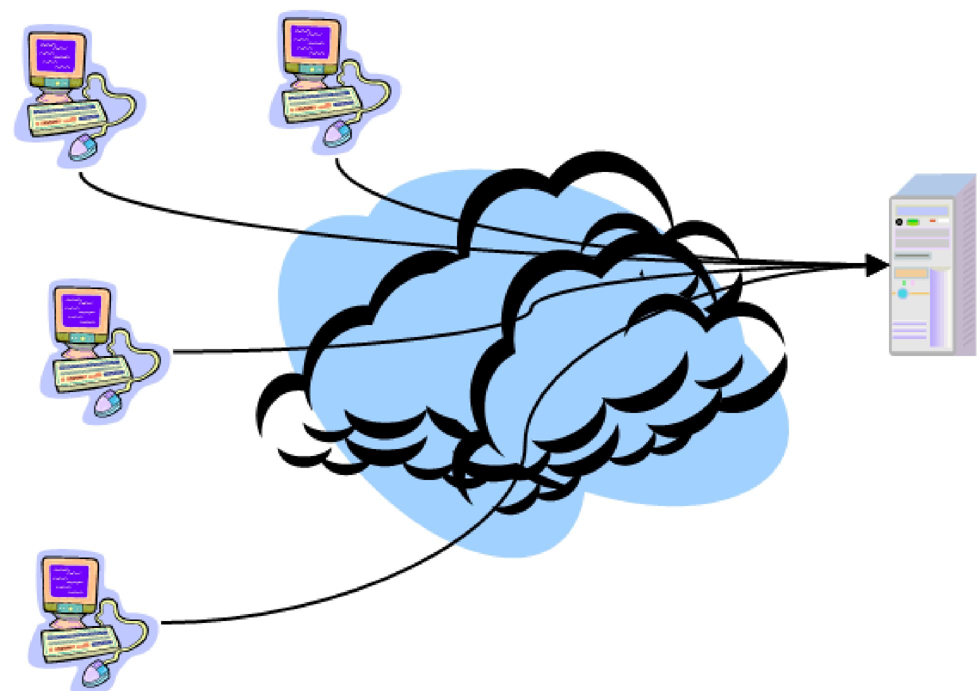
FIB

# Contents

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

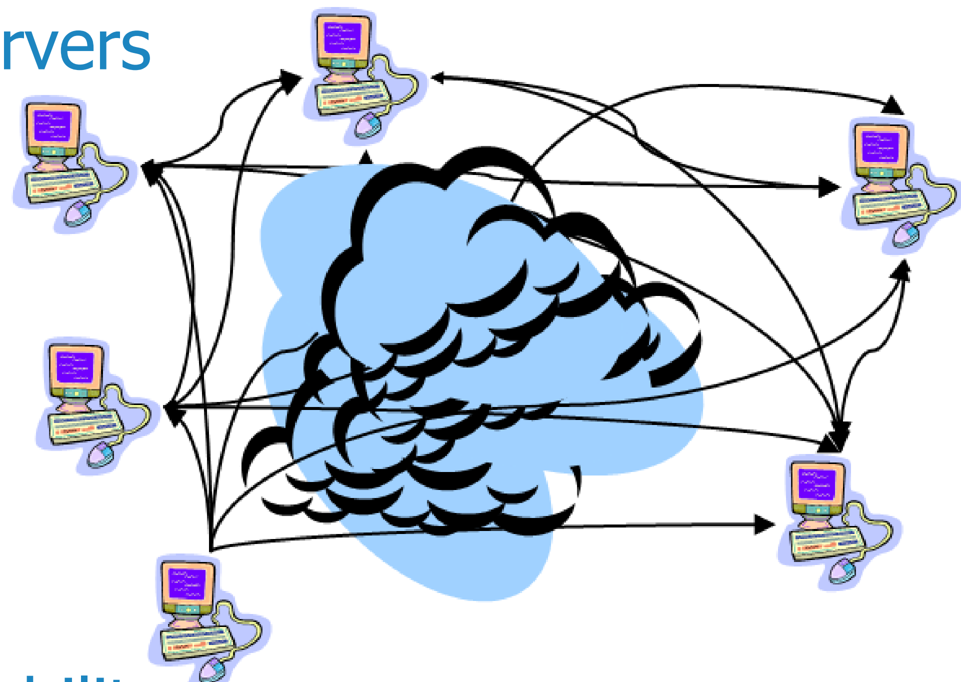
# Introduction

- Client-server systems have been common (and successful) until now, but have limited scalability and robustness in modern scenarios exhibiting …
  - 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
  - Load is balanced among all peers
  - Operation independent of any central node



↑ High scalability & availability
  - Huge amount of computation and storage resources
  - No performance bottlenecks or single points of failure

# 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 directly with each other, messages have to be routed through the available links
- Efficient and fault-tolerant placement of data items across peers and their subsequent access is the main challenge in P2P systems
  - There is a huge number of peers, and they are volatile (they can join/leave/fail at any time)

# Introduction

1. **Unstructured P2P systems**
   - Overlay is built ad hoc (random graph topology)
   - Items end up placed arbitrarily (must be searched)
   - Insert is easy, search is hard
   - 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

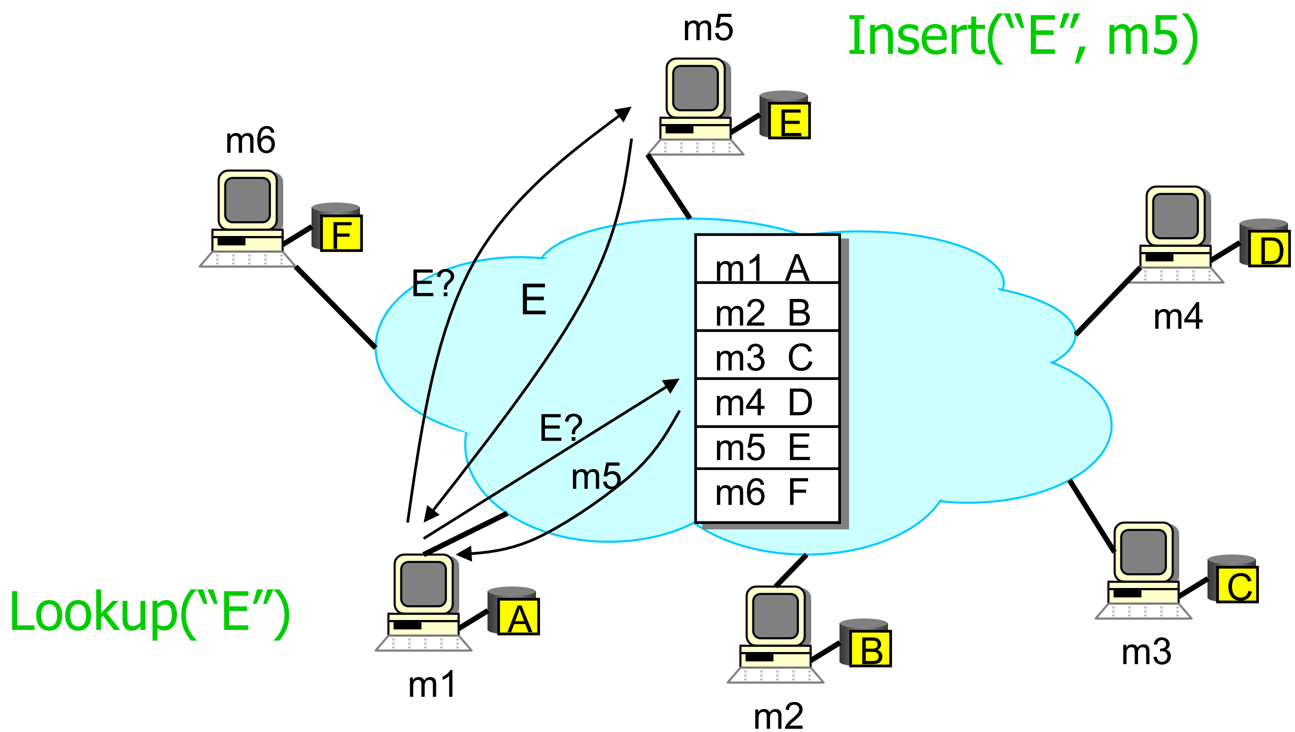
- Introduction
- **Unstructured P2P systems**
- Structured P2P systems

# Contents

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

# Centralized model: Napster

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



# Centralized model: Napster

- Advantages
  - ↑ Simple large-scale service that depends mostly on data and computers owned by ordinary users
  - ↑ Locates files quickly and efficiently (sophisticated search engines can be used to scan the index)
- Disadvantages
  - ↓ Low robustness/scalability: the index can be a single point of failure and performance bottleneck
    - Napster's index was replicated to mitigate this
  - ↓ Resources are distributed, but the index is not
    - Napster was easy to shut down (due to copyright issues)

# READING REPORT

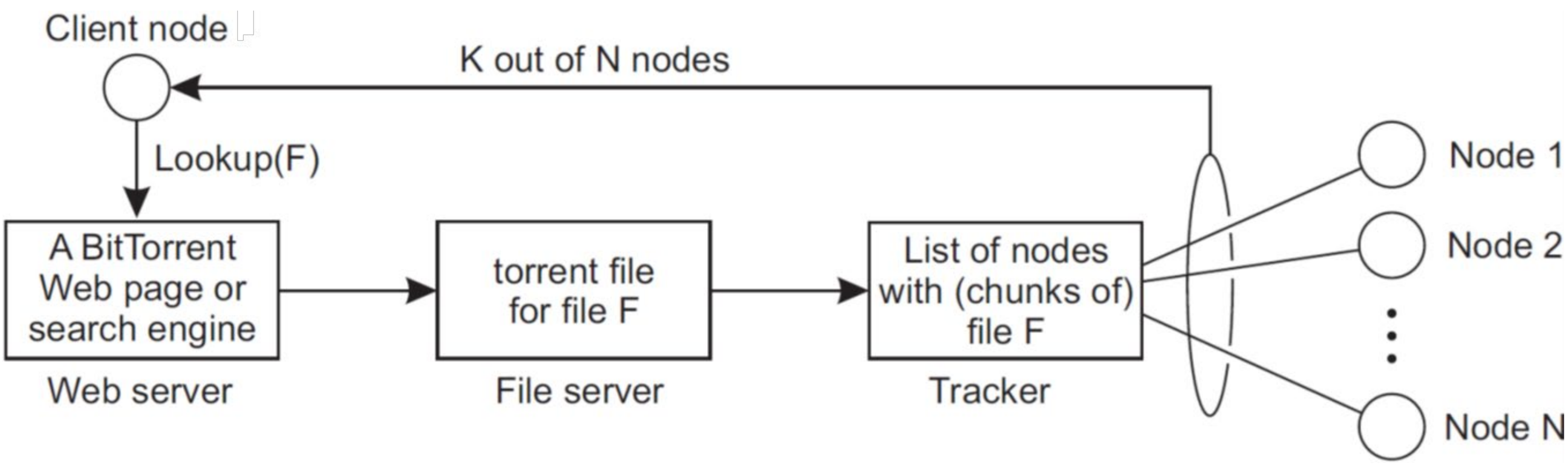
**[Cohen03]** Cohen, B., *Incentives Build Robustness in BitTorrent*, 1st Workshop on Economics of Peer-to-Peer Systems, Berkeley, CA, USA, June 2003

# 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
4. Peers then use this information to connect to each other and distribute file pieces among them



# Centralized model: BitTorrent

- .torrent file contains:
  - Metadata about the file to be shared
    - Name, length, piece length, SHA-1 hash of each piece
      - ↑ Integrity checks done at the piece level
  - URL of the tracker
- Tracker
  - Keeps track of all the peers who have the file (seeds/leechers) and which pieces 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 stay after they get the file
  - Initial seed: a peer that provides the initial copy
    - It must serve at least one complete copy of the file
- Leechers
  - Any peer who does not have the complete file
  - When gets all the pieces, it can become a seed for subsequent downloads

# 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 all peers having the same set of available pieces, and none of the missing ones
  - If the initial seed is prematurely taken down, then the file could not be completely downloaded

# BitTorrent: Piece selection

- **Rarest Piece First**
  - This is the general rule: Request first the pieces that are most rare among the peers
    - Peers will have pieces which all of the others want so they can easily upload
    - Reduces the load on the initial seed as only rare pieces will be downloaded from it
    - Increases the likelihood that all pieces are still available even if the initial seed leaves
- The behavior of the *Rarest Piece First* policy can be modified by three additional policies
  - Strict Priority, Random First Piece, Endgame Mode

# BitTorrent: Piece selection

- **Strict Priority**
  - Once a sub-piece has been requested, request the other sub-pieces from that piece before any other
    - Aims to minimize partially-received pieces as only complete pieces can be uploaded to other peers
- **Random First Piece**
  - Request pieces at random until the first complete piece is assembled and switch to *Rarest Piece First*
    - Special case at the start of download: Peer has nothing to upload so it must get a complete piece promptly
    - *Rarest Piece First* is not suitable: Rare pieces are present on few peers, so they would be downloaded slower

# BitTorrent: Piece selection

- **End Game Mode**
  - <u>When all missing sub-pieces have been requested, those not received yet are requested from every peer containing them</u>
    - When a sub-piece arrives, the pending requests for that sub-piece are cancelled
  - Special case at the end of download: Completion of a download could be delayed due to a single peer with a slow transfer rate
  - This mode wastes some bandwidth, but it is not too much in practice since the end period is short

# 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 some peer you can download from it

# 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
    - Parallel download from multiple sources
    - Partially downloaded files can be uploaded
  - ↑ Tit-for-tat encourages all peers to contribute and avoids ‘free riders’
    - Peers reciprocate to peers which upload to them
  - ↑ ‘Rarest Piece First’ extends the lifetime of swarm
    - Increases the likelihood that all pieces are available

# Centralized model: BitTorrent

- Disadvantages
  - ↓ All interested peers should be active at same time
    - Performance deteriorates if the swarm 'cools off'
  - ↓ A centralized tracker is a single point of failure
    - New nodes cannot enter the swarm if tracker goes down

⇒ BitTorrent without a centralized tracker

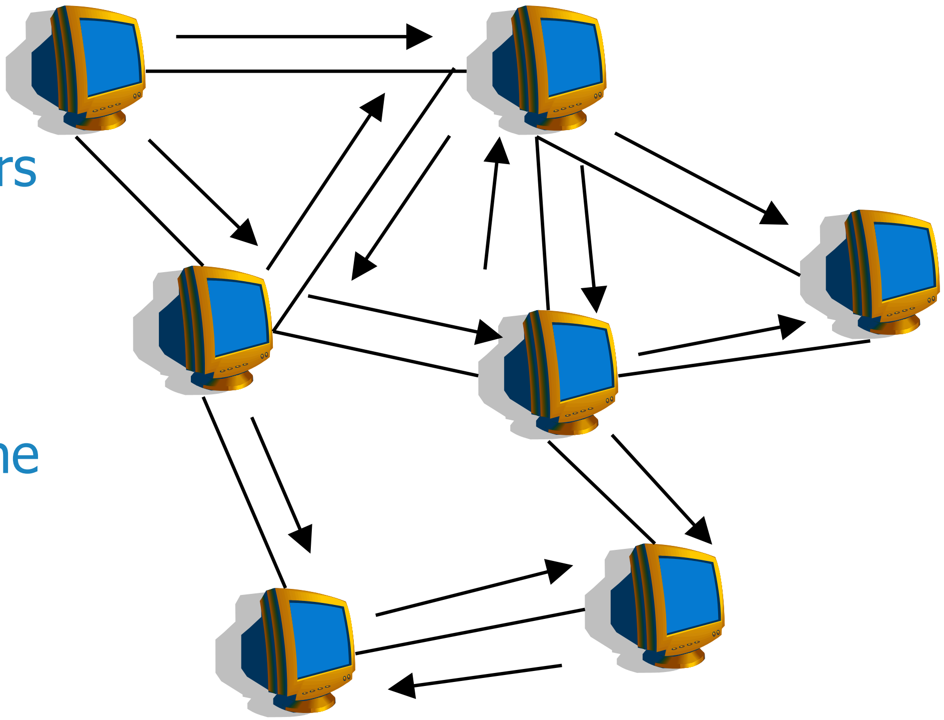
- e.g. Vuze (former Azureus)
- Official client also supports distributed tracking
  - Mainline DHT (based on **Kademlia**)

# Contents

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

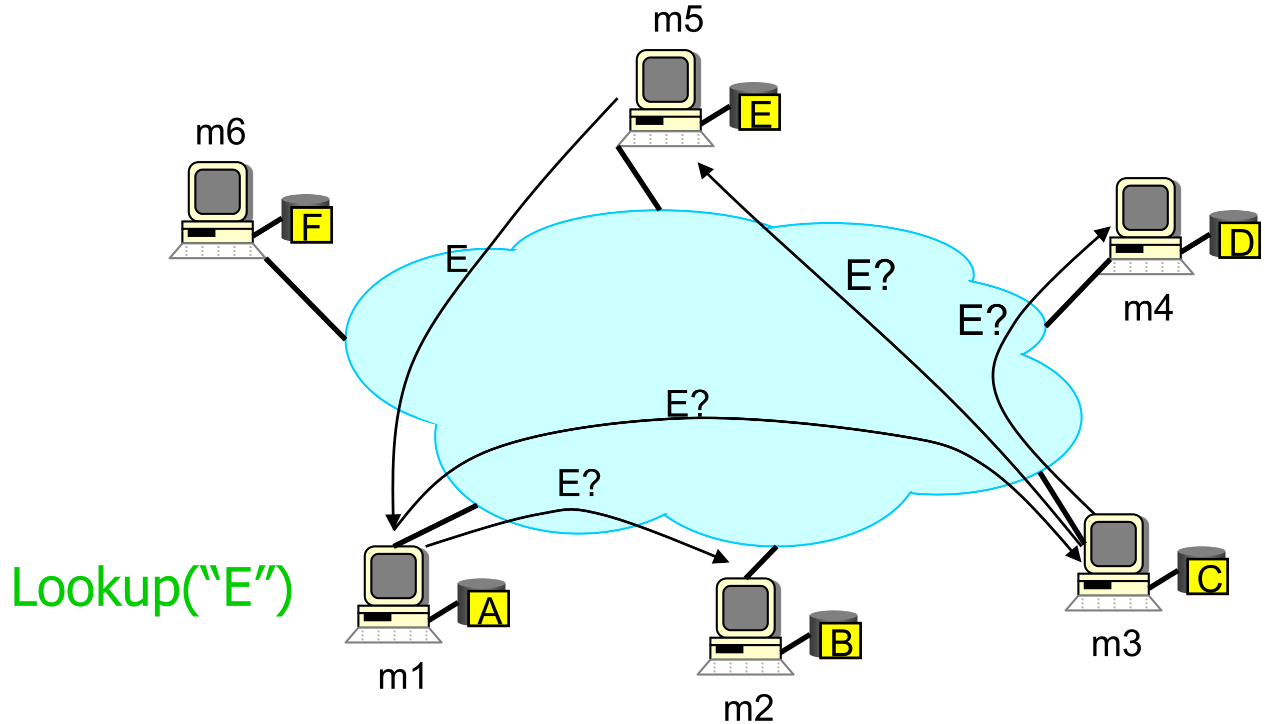
# Decentralized model: Gnutella

- Based on query **flooding**
- How to find a file:
  1. Peer sends **Query** descriptor to all neighbors
     - All peers connected to it
  2. Neighbors recursively multicast the descriptor
  3. Eventually a peer with the file receives descriptor, and sends back a **QueryHit** (retracing the path of Query)
  4. Peer requests file to this peer (direct connection)



# Decentralized model: Gnutella

- Example
  - Assumption: m1's neighbors are m2 and m3; m3's neighbors are m4 and m5;…



# Decentralized model: Gnutella

- Advantages
  - ↑ Highly decentralized, self-organizing, and naturally resilient to node failures
  - ↑ No centralized index: directory is distributed across peers (they have similar responsibilities)
- Disadvantages
  - ↓ Slow information discovery
    - Cannot offer absolute guarantees on locating objects
  - ↓ 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

- Needs 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
   - Keep increasing TTL until search succeeds

# Decentralized model: Gnutella

- Needs some mechanisms to control flooding

C. Random walk: Ask one neighbor randomly, who asks one neighbor, and so on
   - 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
   - Avoid re-broadcasting Query messages
   - Choose another neighbor if using random walk

# Contents

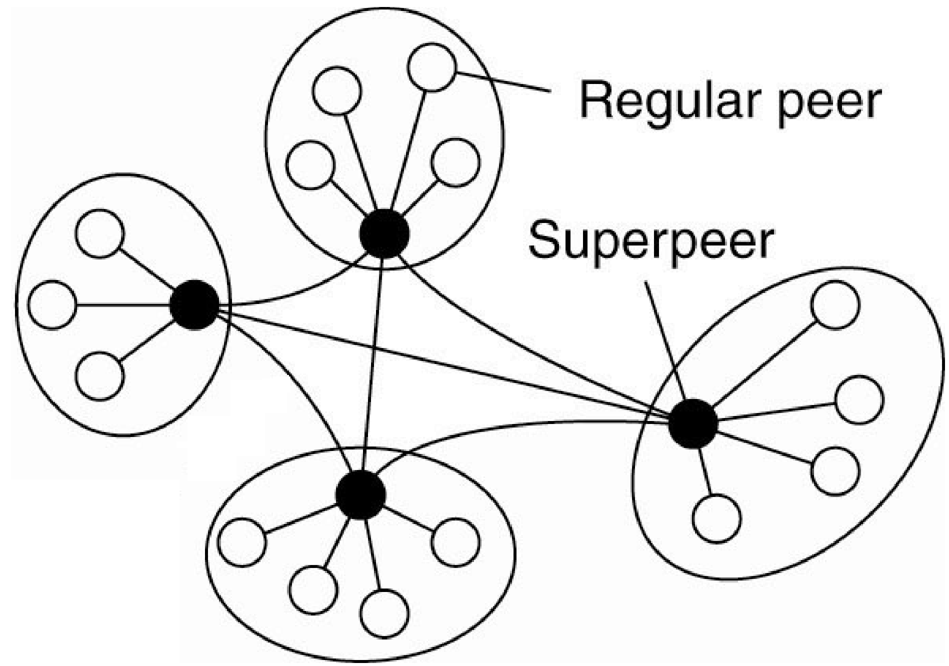
- Introduction

- **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

- Combines the advantages and disadvantages of the centralized and decentralized models
  - ↑ Directory is still decentralized (across super-peers)
  - ↑ Information discovery involves only super-peers
  - ↓ Super-peers can suffer scalability and robustness problems (because they are centralized and communicate through flooding)
  - ↓ A mechanism to elect the super-peers is needed
- 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

# Contents

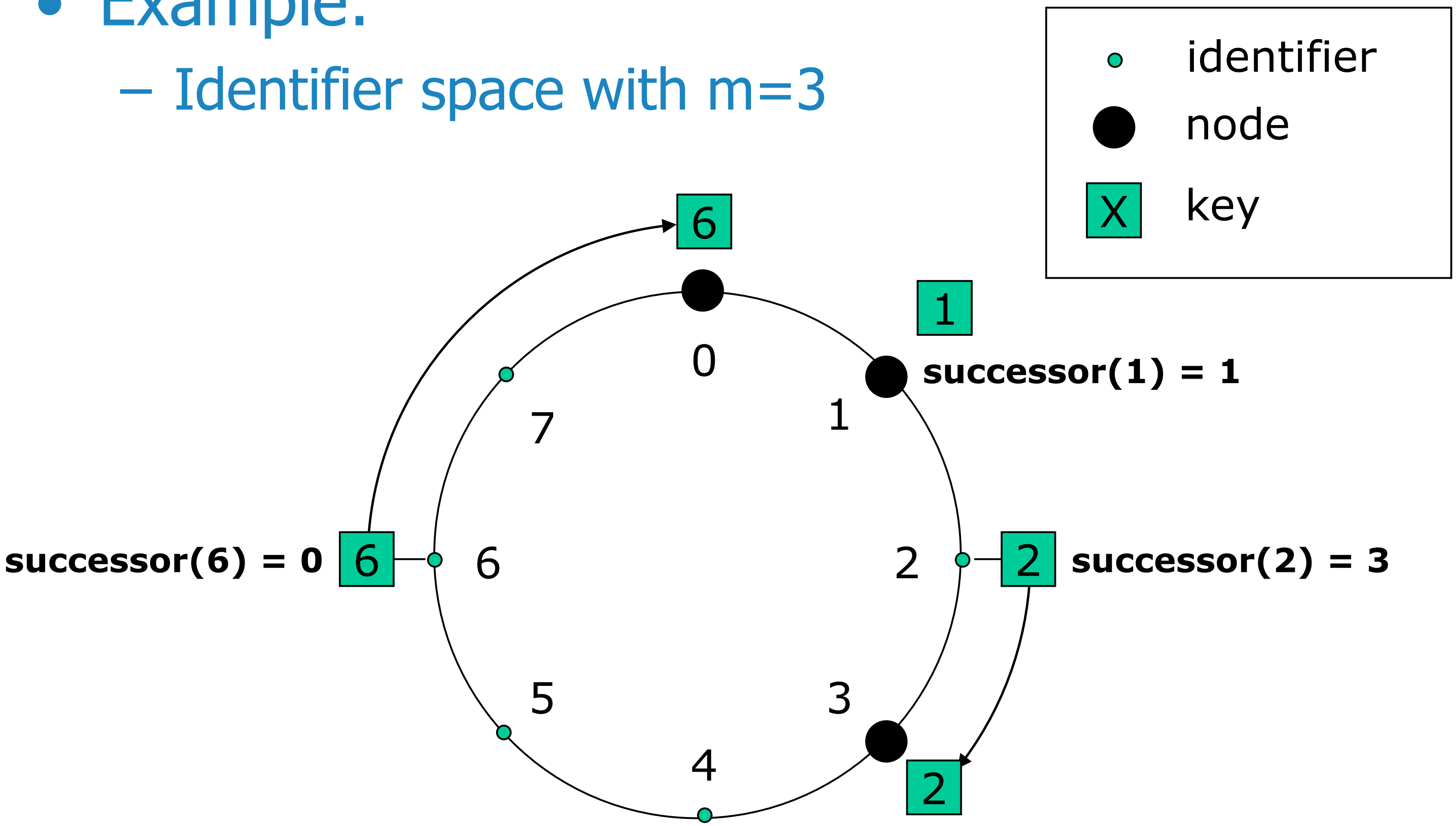
- Introduction
- Unstructured P2P systems
- **Structured P2P systems**
  - **Chord**
  - Kademlia

# 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

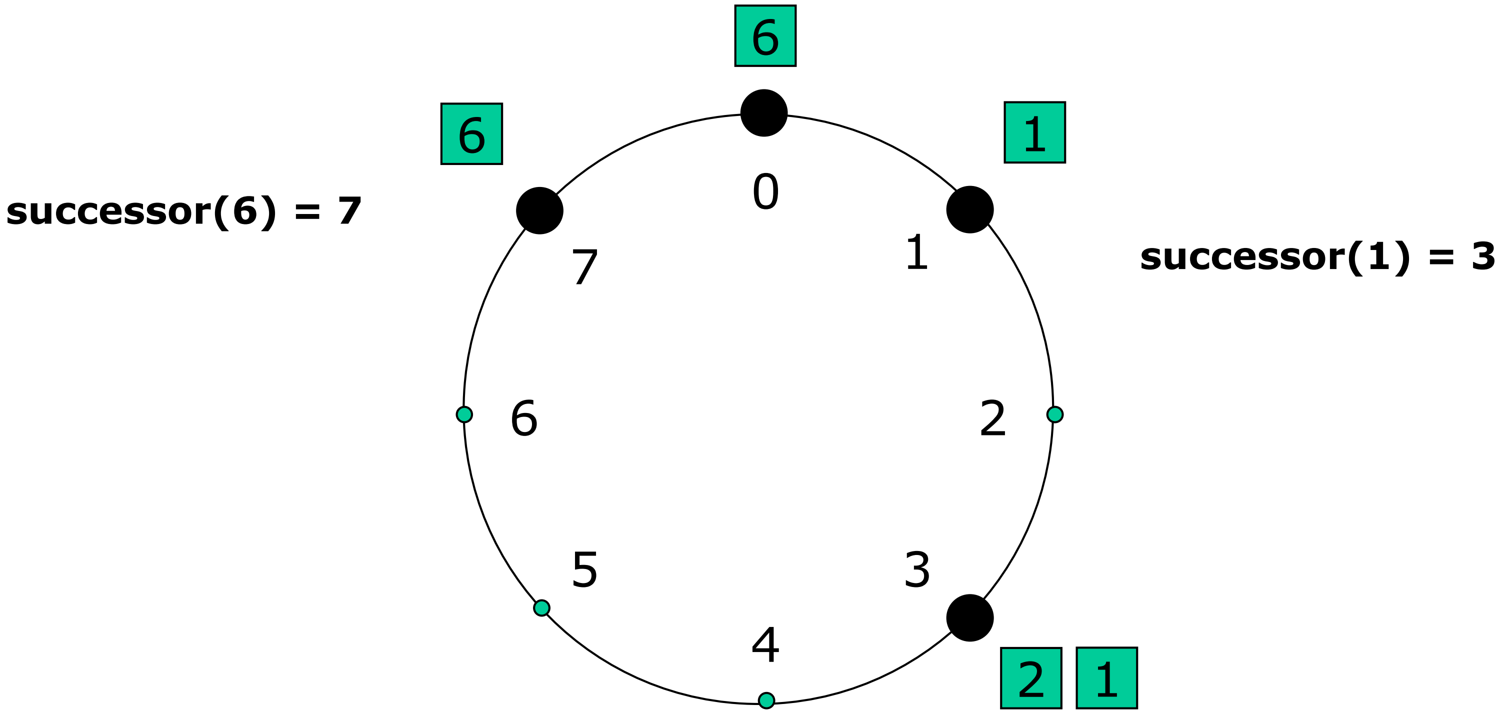
# Chord: Item insertion

- Example:
  - Identifier space with m=3



# Chord: Item insertion

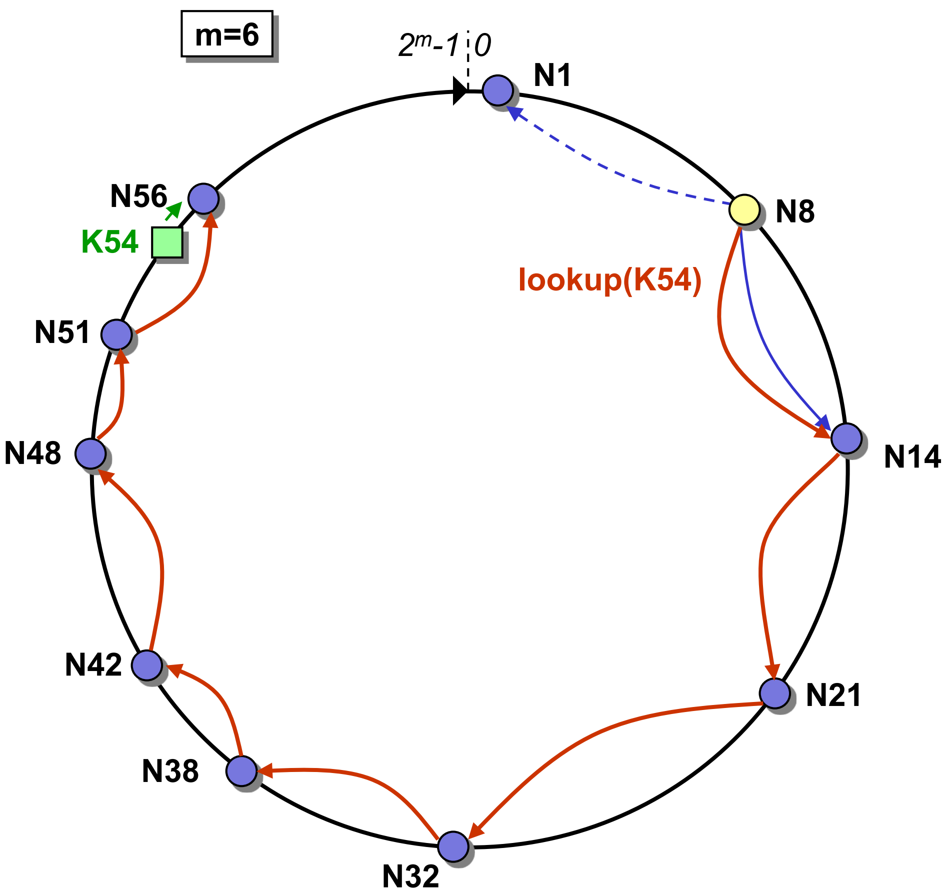
- Item management with node joins and departures



# Chord: Item lookup

## A. Basic Chord

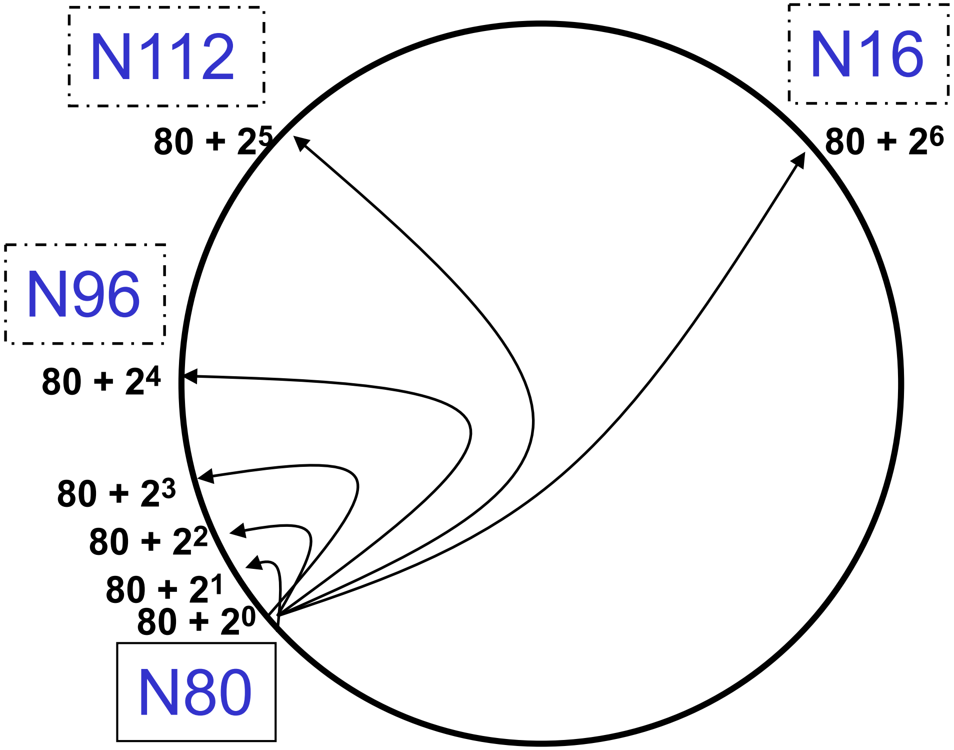
- Each node knows only two other nodes
  - Successor
  - Predecessor (for ring management)
- Lookup by forwarding requests around the ring <u>through successor pointers</u>
- Requires $O(N)$ hops



# Chord: Item lookup

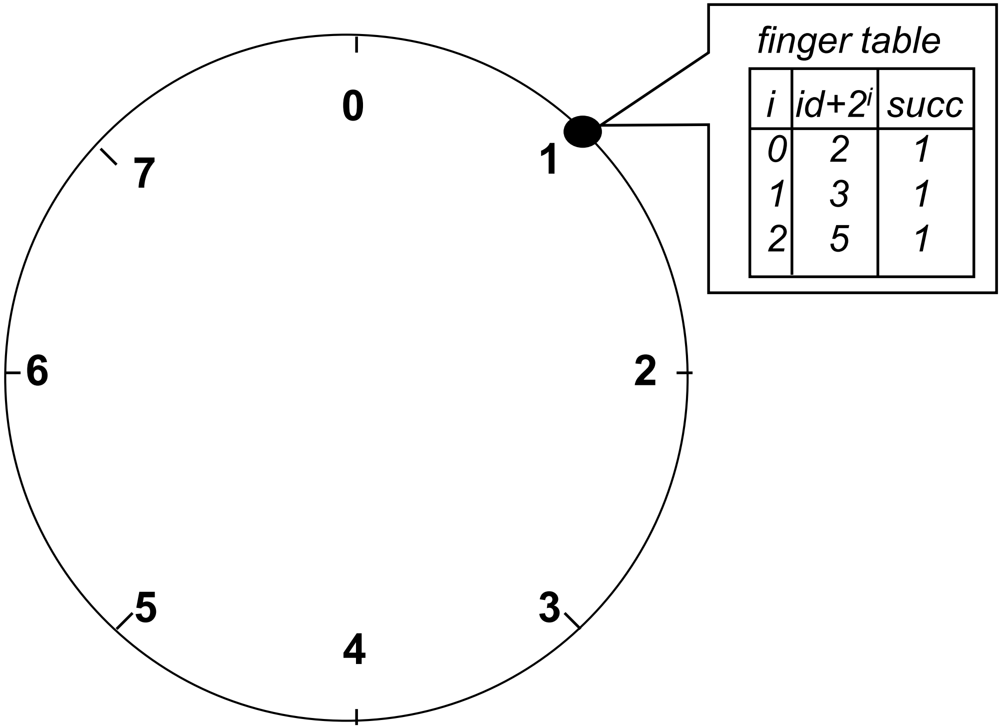
**B. Finger tables** for accelerating lookup

- 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
- Requires $O(Log\ N)$ hops



# Chord: Managing finger tables

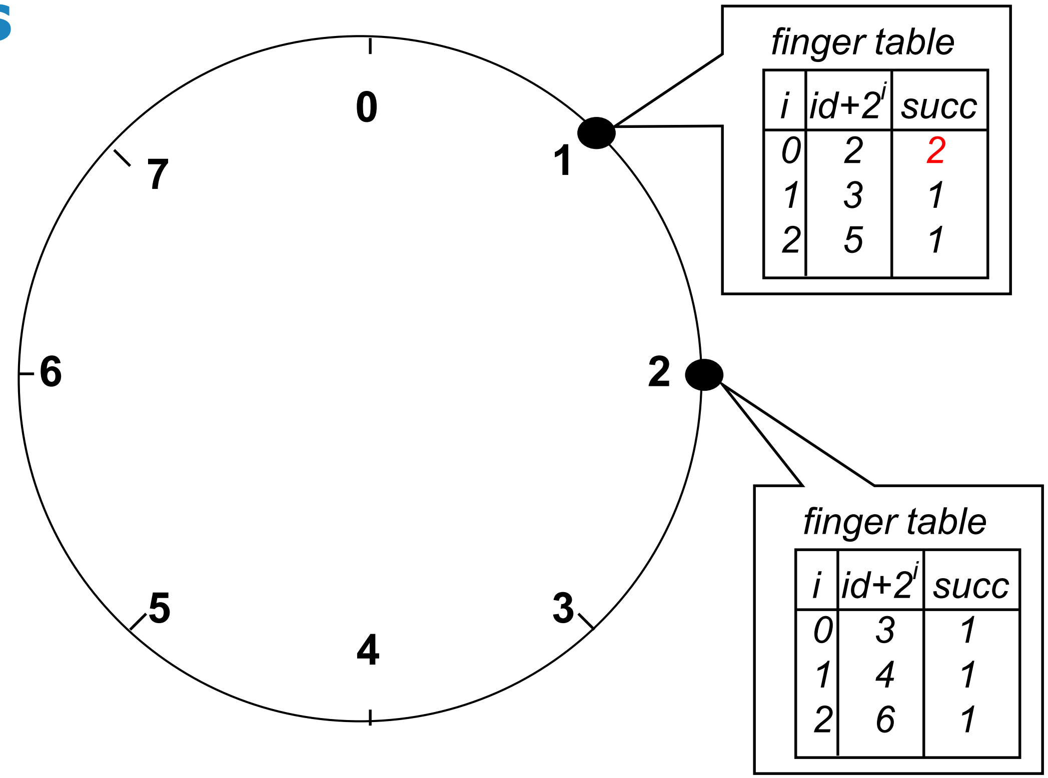
- Example:
  - Node (1) **joins**
    - All entries in its finger table are initialized to itself




| $i$ | $id+2^i$ | succ |
|---|---|---|
| 0 | 2 | 1 |
| 1 | 3 | 1 |
| 2 | 5 | 1 |

# Chord: Managing finger tables

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




finger table (node 1)

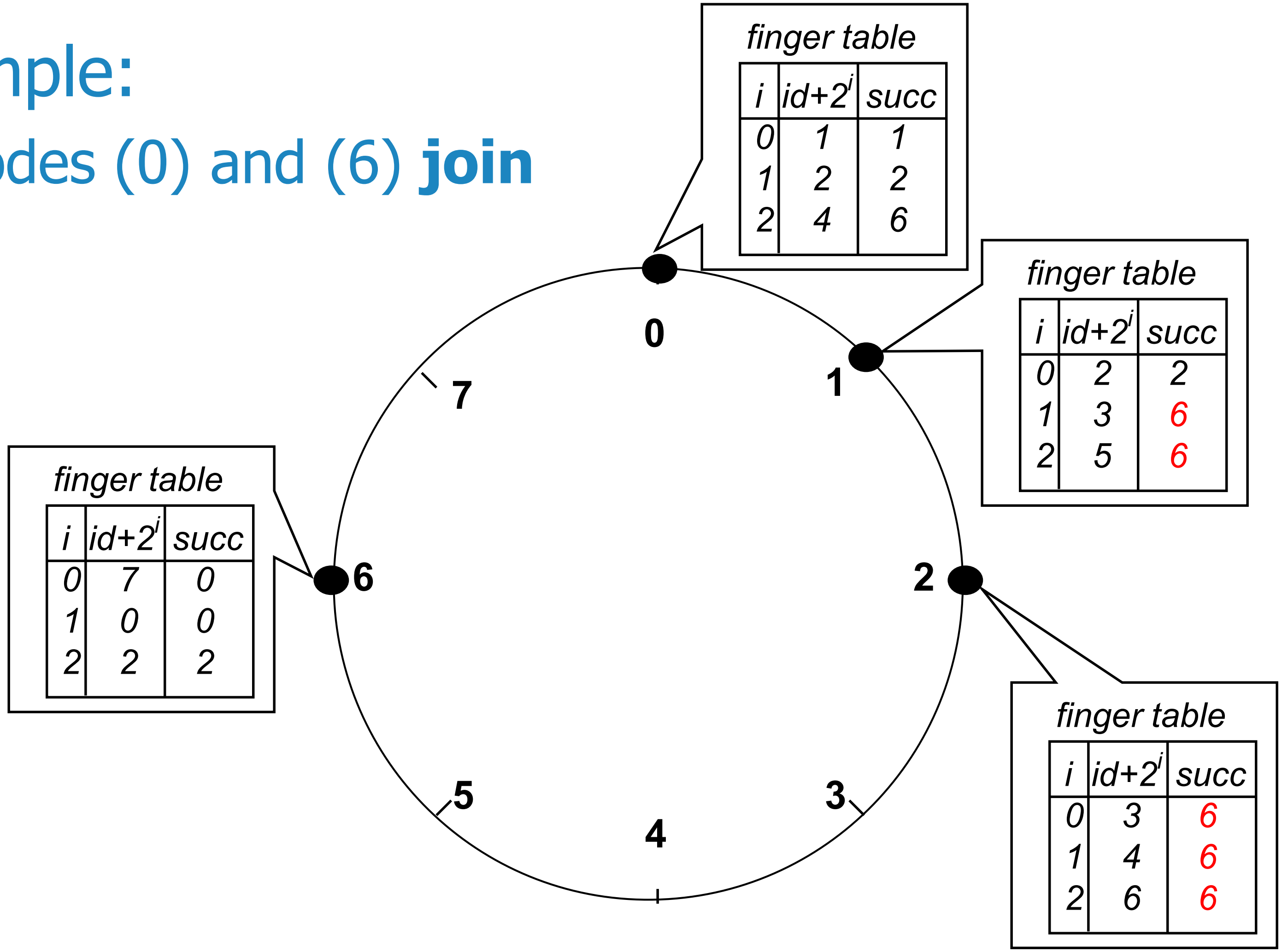
| i | id+$2^i$ | succ |
|---|---|---|
| 0 | 2 | 2 |
| 1 | 3 | 1 |
| 2 | 5 | 1 |

finger table (node 2)

| i | id+$2^i$ | succ |
|---|---|---|
| 0 | 3 | 1 |
| 1 | 4 | 1 |
| 2 | 6 | 1 |

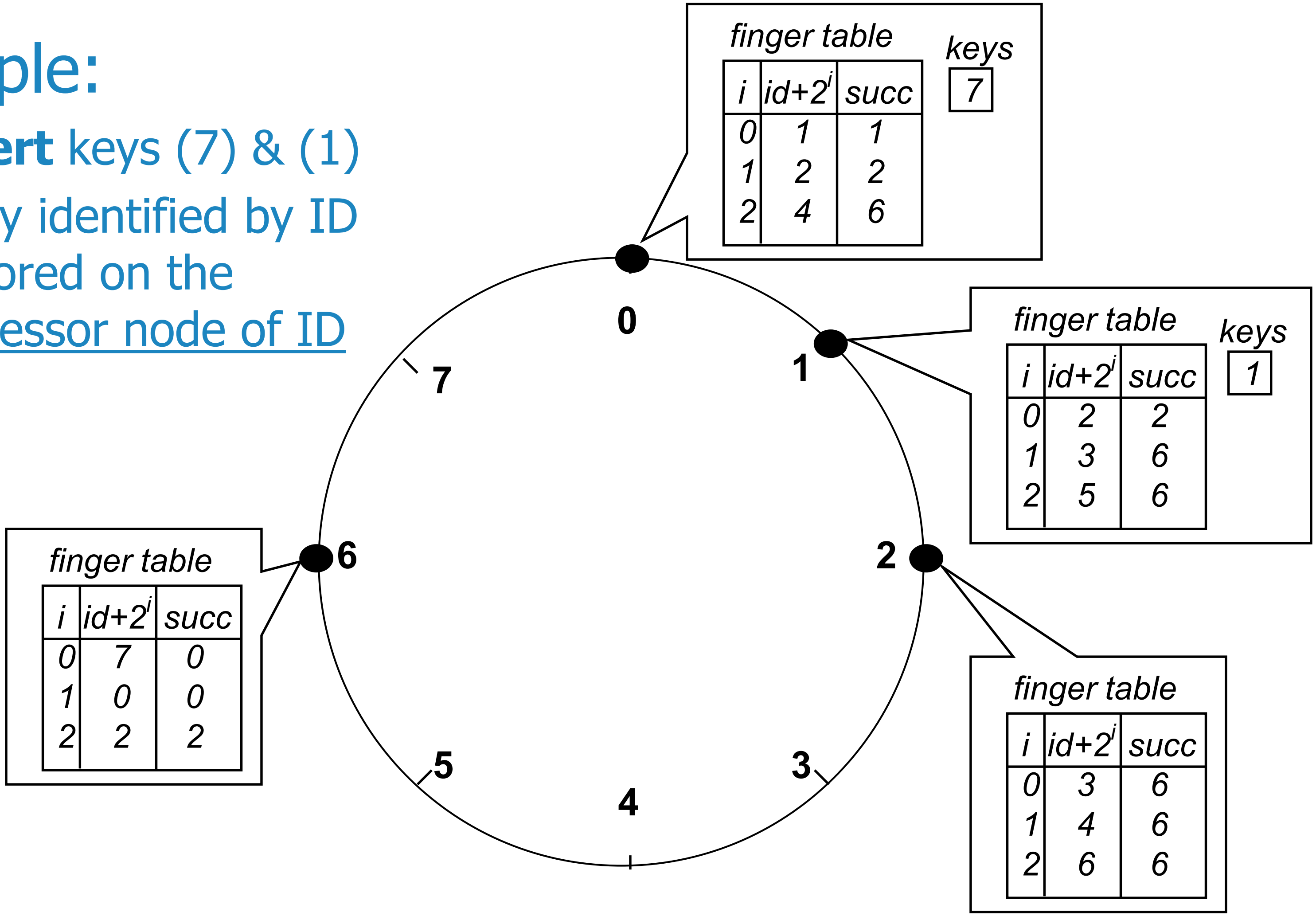
# Chord: Managing finger tables

- Example:
  - Nodes (0) and (6) **join**



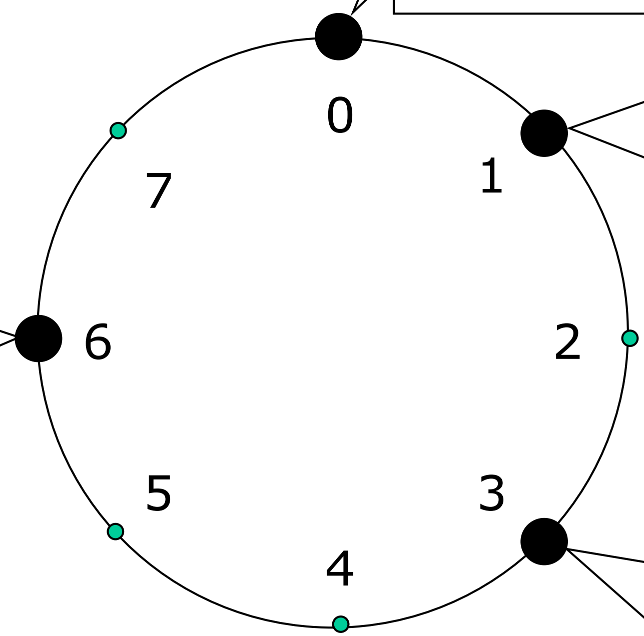
# Chord: Managing finger tables

- Example:
  - **Insert** keys (7) & (1)
  - A key identified by ID is stored on the successor node of ID



# Chord: Managing finger tables

- Example 2:
  - Item management when node (6) joins




Node 0:

*finger table*

| *i* | *id+2^i* | *succ* |
|---|---|---|
| 0 | 1 | 1 |
| 1 | 2 | 3 |
| 2 | 4 | **6** |

*keys*: 6

Node 1:

*finger table*

| *i* | *id+2^i* | *succ* |
|---|---|---|
| 0 | 2 | 3 |
| 1 | 3 | 3 |
| 2 | 5 | **6** |

*keys*: 1

Node 3:

*finger table*

| *i* | *id+2^i* | *succ* |
|---|---|---|
| 0 | 4 | **6** |
| 1 | 5 | **6** |
| 2 | 7 | 0 |

*keys*: 2

Node 6:

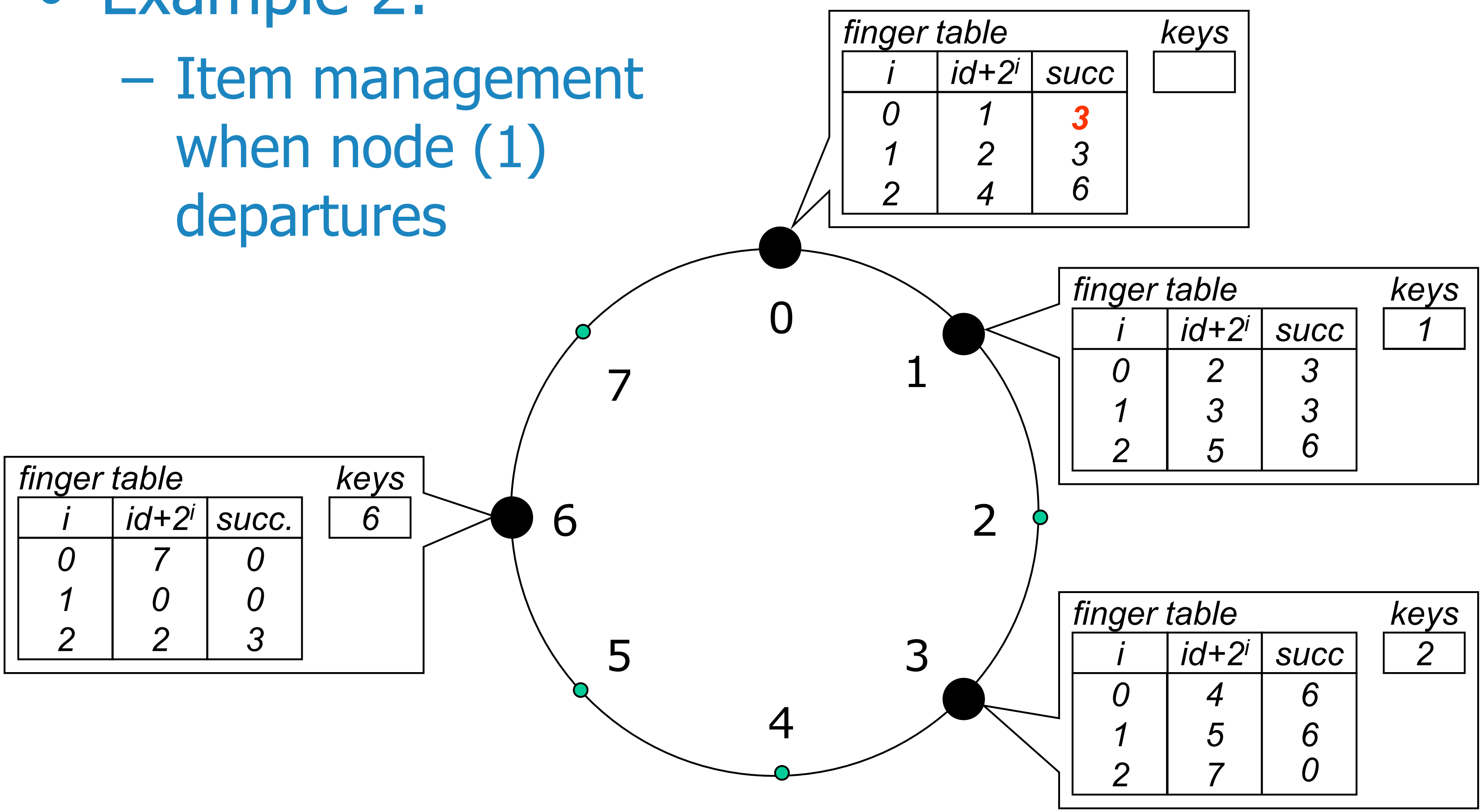
*finger table*

| *i* | *id+2^i* | *succ* |
|---|---|---|
| 0 | 7 | 0 |
| 1 | 0 | 0 |
| 2 | 2 | 3 |

*keys*:

# Chord: Managing finger tables

- Example 2:
  - Item management when node (1) departures

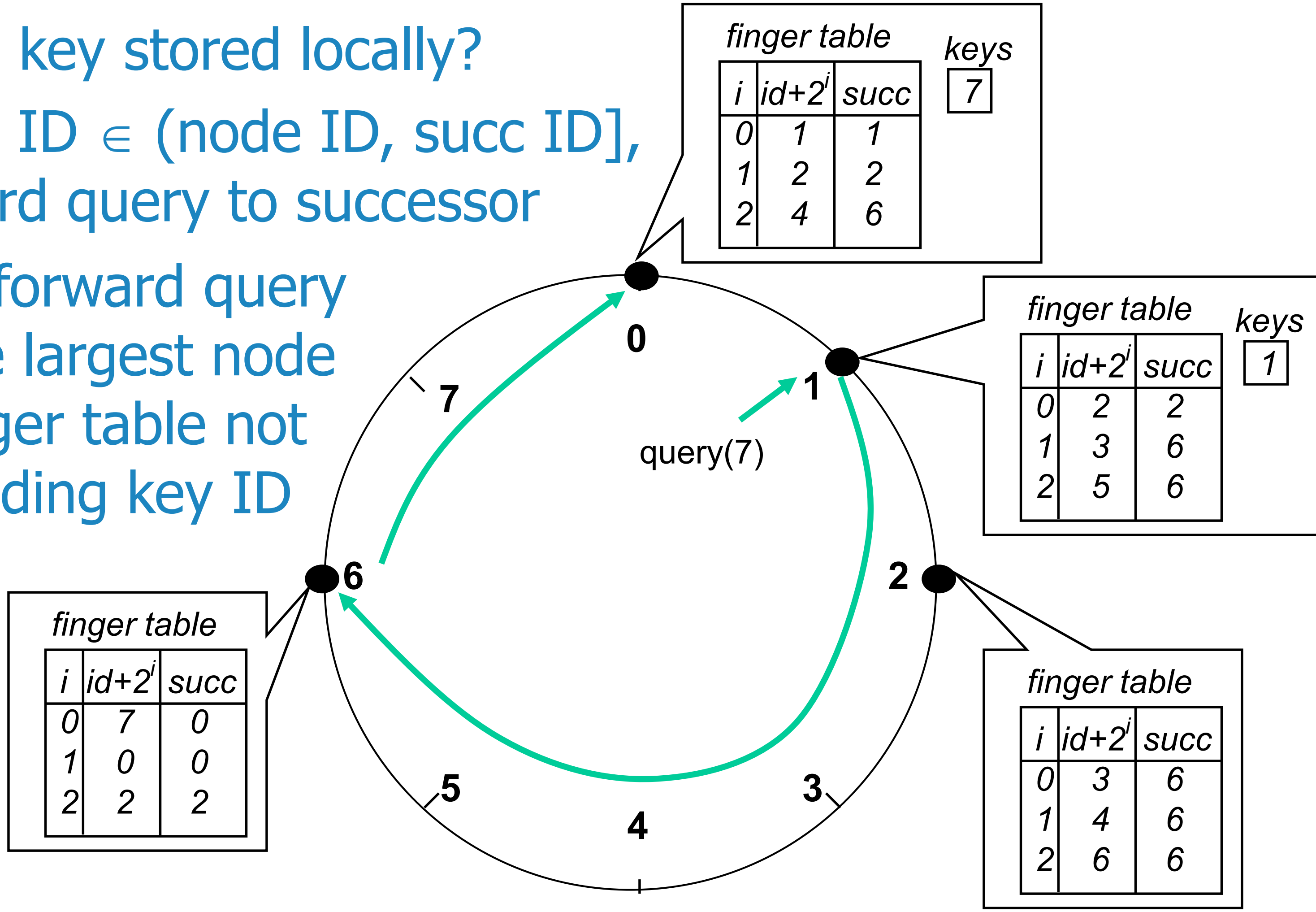


# Chord: Item lookup with finger tables

a) Is the key stored locally?

b) If key ID $\in$ (node ID, succ ID], forward query to successor

c) Else, forward query to the largest node in finger table not exceeding key ID




Node 0 — finger table

| i | id+$2^i$ | succ |
|---|---|---|
| 0 | 1 | 1 |
| 1 | 2 | 2 |
| 2 | 4 | 6 |

Node 1 — finger table

| i | id+$2^i$ | succ |
|---|---|---|
| 0 | 2 | 2 |
| 1 | 3 | 6 |
| 2 | 5 | 6 |

Node 2 — finger table

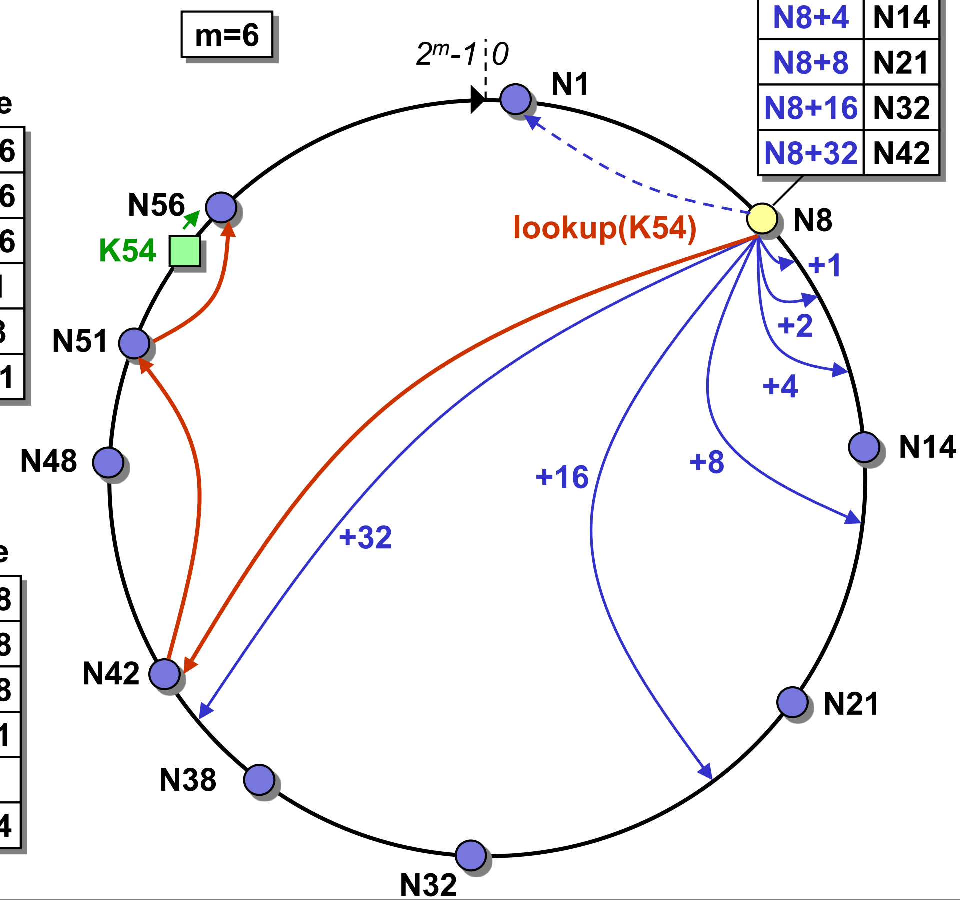
| i | id+$2^i$ | succ |
|---|---|---|
| 0 | 3 | 6 |
| 1 | 4 | 6 |
| 2 | 6 | 6 |

Node 6 — finger table

| i | id+$2^i$ | succ |
|---|---|---|
| 0 | 7 | 0 |
| 1 | 0 | 0 |
| 2 | 2 | 2 |

# Chord: Item lookup with finger tables

- Example 2:




Finger table

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

Finger table

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

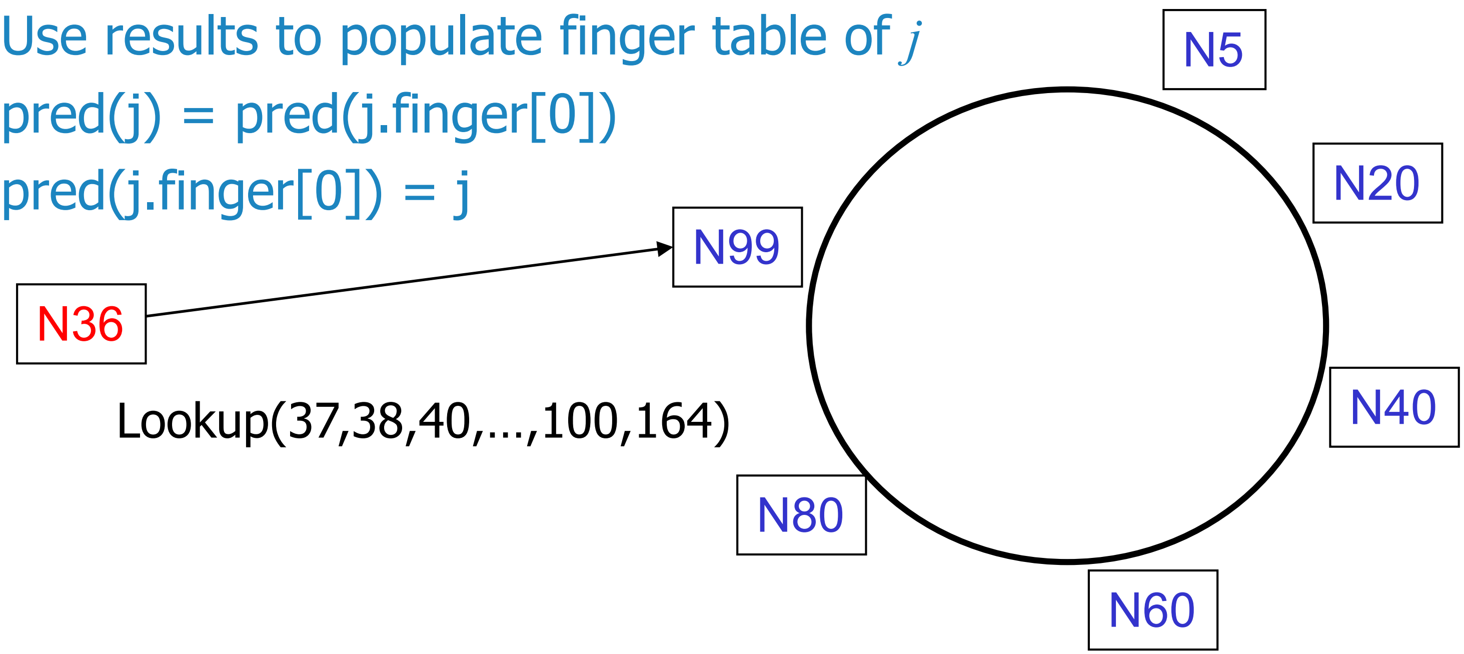
Finger table

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

# Chord: Node joining

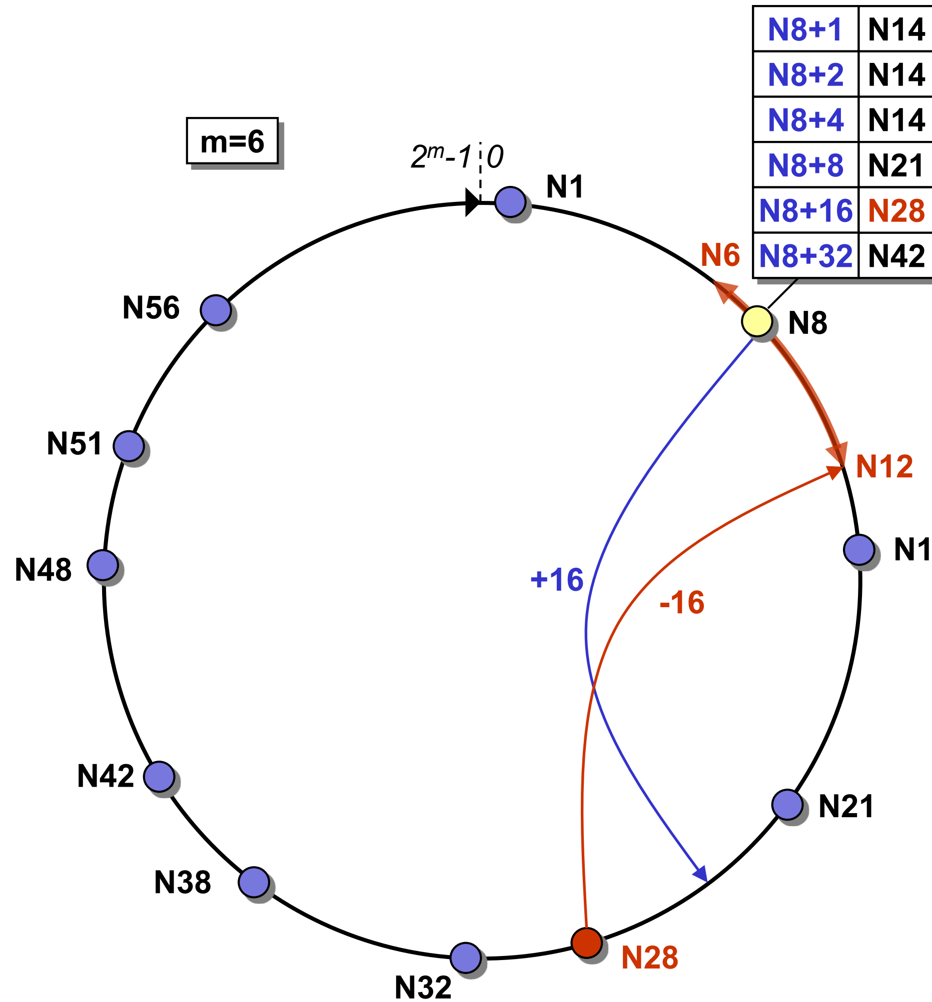
## A. **Basic** process for joining the ring (3 steps):

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



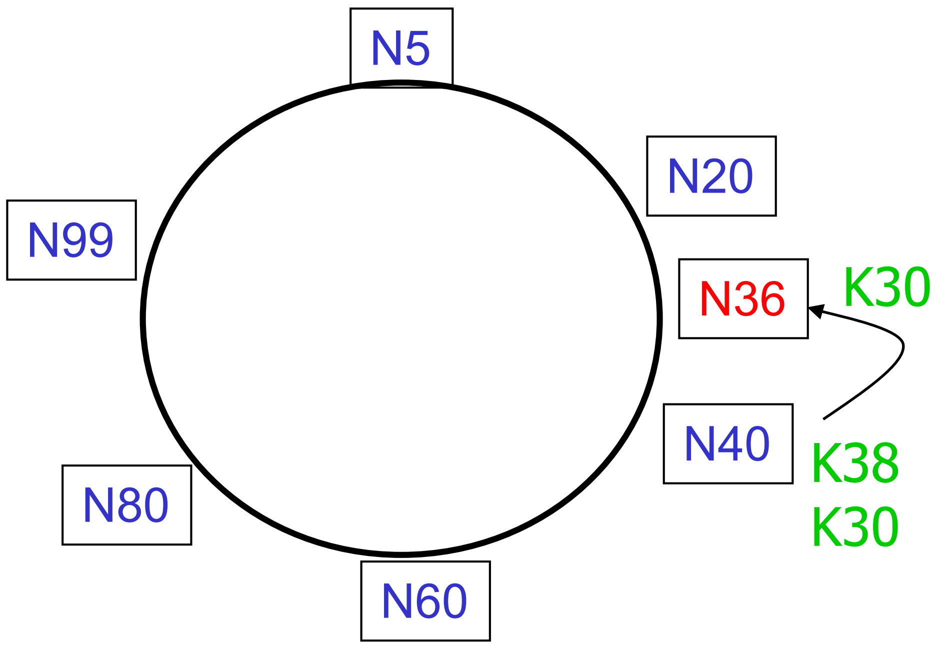
# Chord: Node joining

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 (pred ID, node ID] from successor to new node



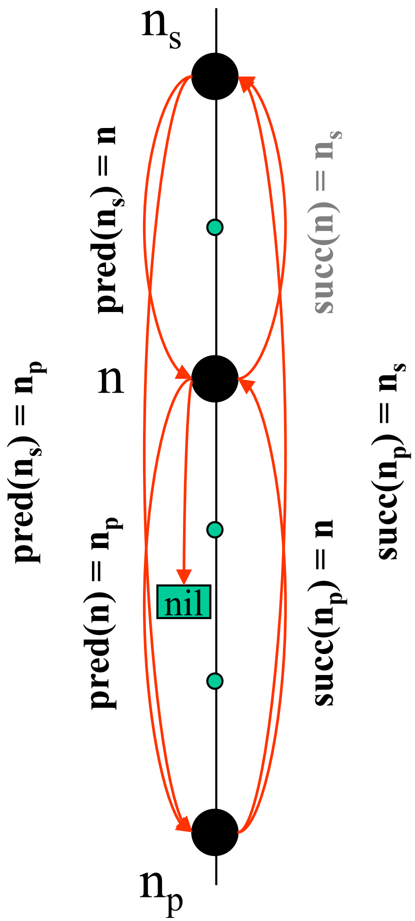

Copy keys 21...36 from N40 to N36

# 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 <u>periodically</u> verifies successor and predecessor
3. All nodes also refresh periodically finger table entries (one entry at a time)

# Chord: Stabilization

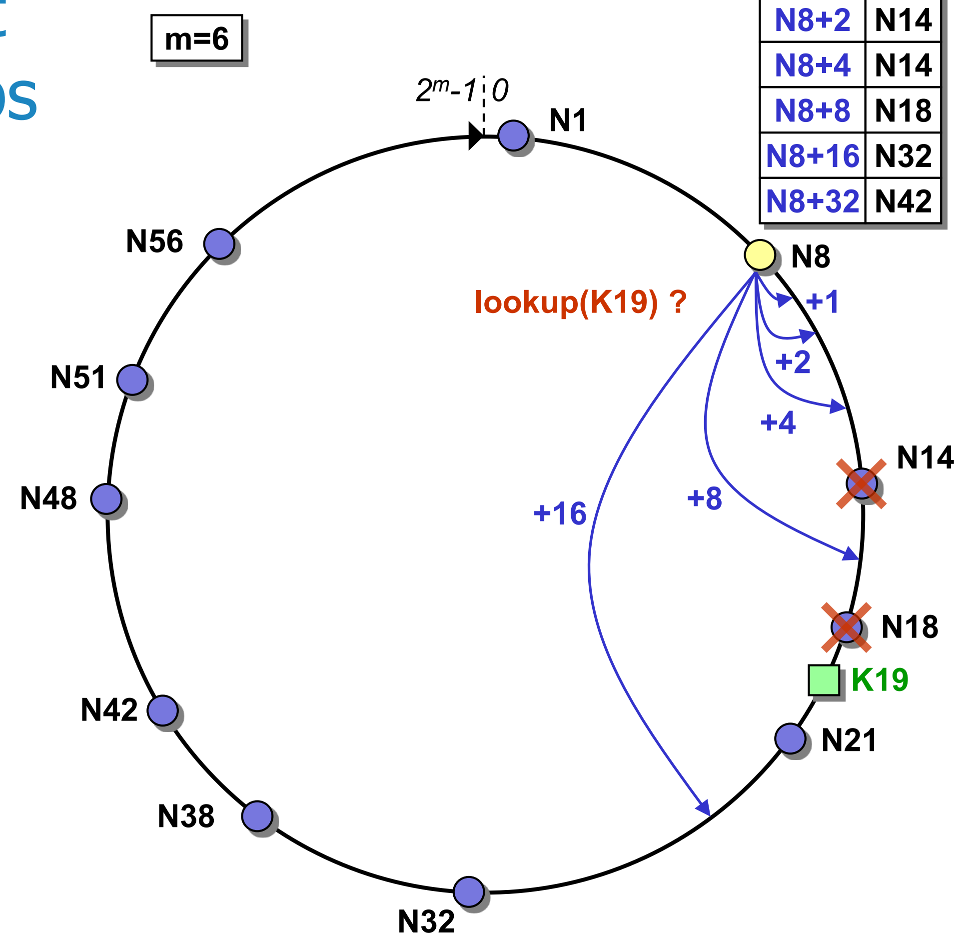



- ***n* joins**
  - $pred(n)$ = nil
  - $n$ schedules the following <u>stabilization procedure</u>, which is repeated until $n$ acquires $n_s$ as successor and $n_s$ learns about $n$
- ***n* runs the stabilization with candidate successor ($n_s$)**
  1. $n$ asks $n_s$ for its predecessor
  2. $n_s$ responds $pred(n_s)=n_p$ to $n$
  3. $n$ checks if $pred(n_s) \in (n, n_s)$
     - *true*: $n$ updates $\mathrm{succ}(n)=pred(n_s)$ and stabilizes again
     - *false*: $n$ confirms $\mathrm{succ}(n)=n_s$
       $n$ notifies $n_s$ that $n$ may be its new predecessor
  4. $n_s$ checks if $n \in (pred(n_s), n_s)$
     - *true*: $n_s$ transfers keys in the range $(pred(n_s), n]$ to $n$
       $n_s$ acquires $n$ as predecessor $\Rightarrow pred(n_s)=n$
- **Eventually, $n_p$ also runs the stabilization (with $n_s$)**
  1. $n_p$ asks $n_s$ for its predecessor
  2. $n_s$ responds with its new predecessor (now $n$)
  3. $n_p$ acquires $n$ as successor and notifies $n$ about its existence
  4. $n$ acquires $n_p$ as its predecessor

# Chord: Dealing with node failures

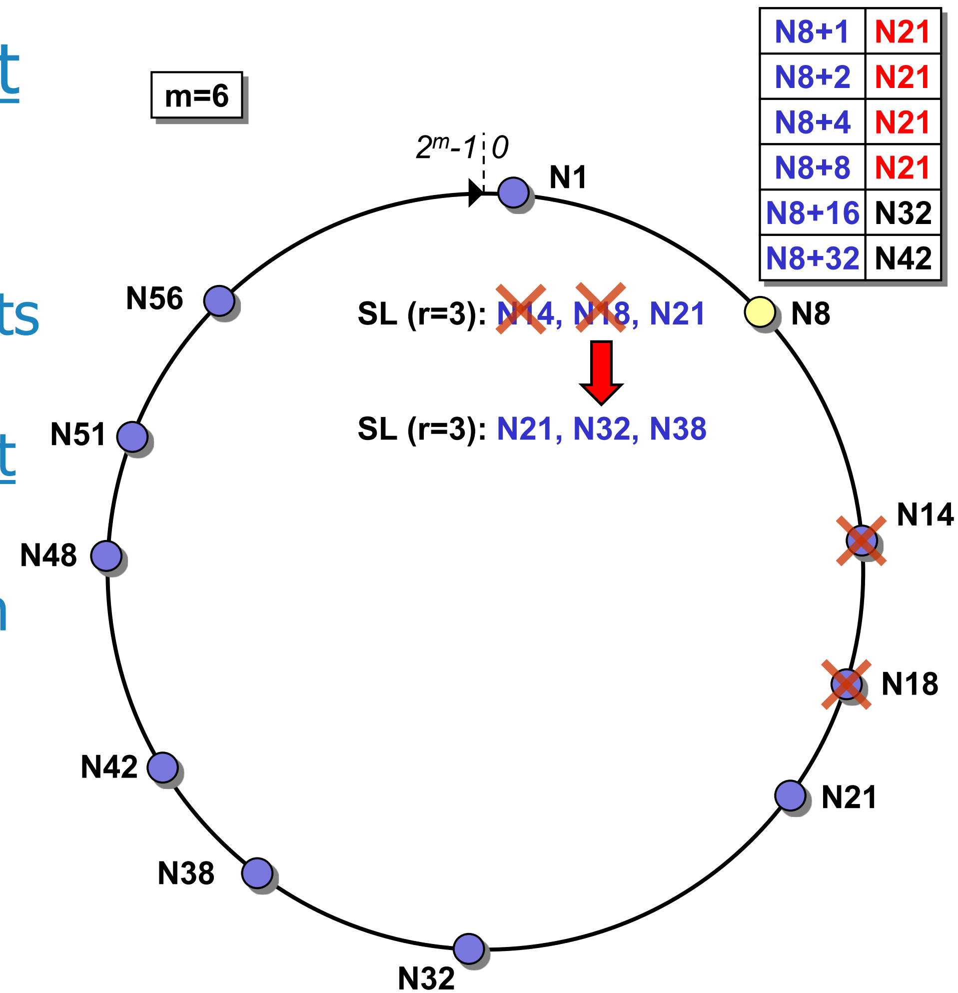
- Failure of nodes might cause incorrect lookups
  - Lookup(K19) fails: N8 returns the first alive node it knows (N32) instead of the correct successor (N21)

| | |
|---|---|
| N8+1 | N14 |
| N8+2 | N14 |
| N8+4 | N14 |
| N8+8 | N18 |
| N8+16 | N32 |
| N8+32 | N42 |



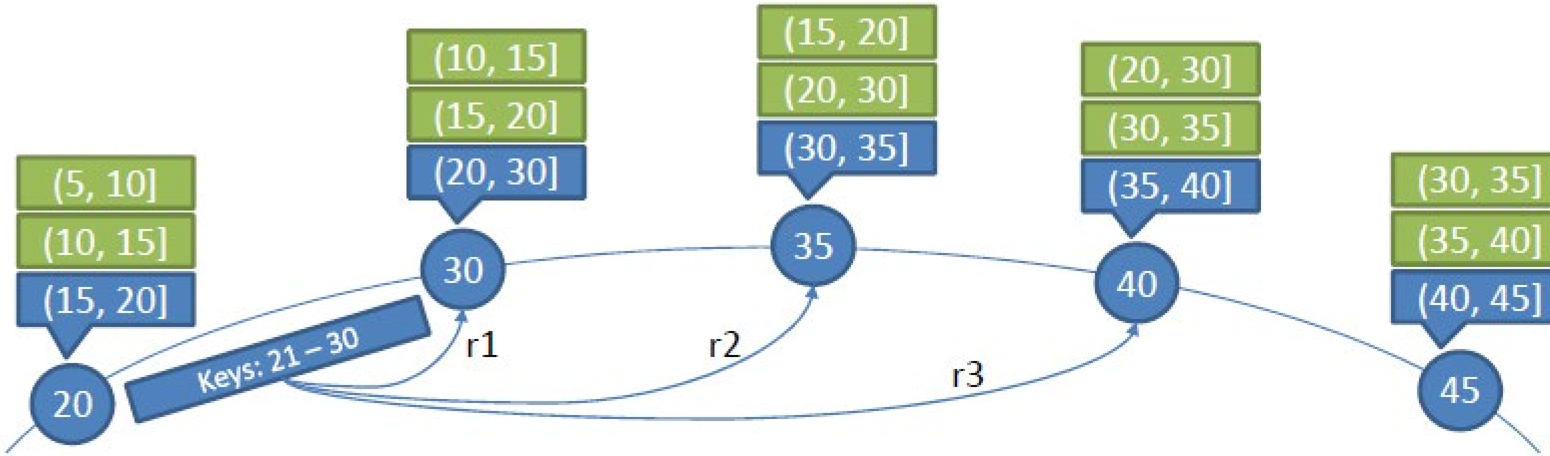
# Chord: Dealing with node failures

- Solution: <u>successor-list</u>
  - Each node knows its **r** immediate successors
  - If a node notices that its successor has failed, it replaces it with the <u>first alive process in the list</u>
  - Eventually, stabilization 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 replication
  - 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 ($d-1 \leq r$)
  - As nodes join/leave, successor-lists are updated, leading to changes in the placement of replicas



# Chord

- Advantages
  - ↑ Efficient: $O(Log\ N)$ messages per lookup
  - ↑ Scalable: $O(Log\ N)$ state per node

  $N$ is the total number of nodes

  - ↑ Robust: survives massive changes in membership
- 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

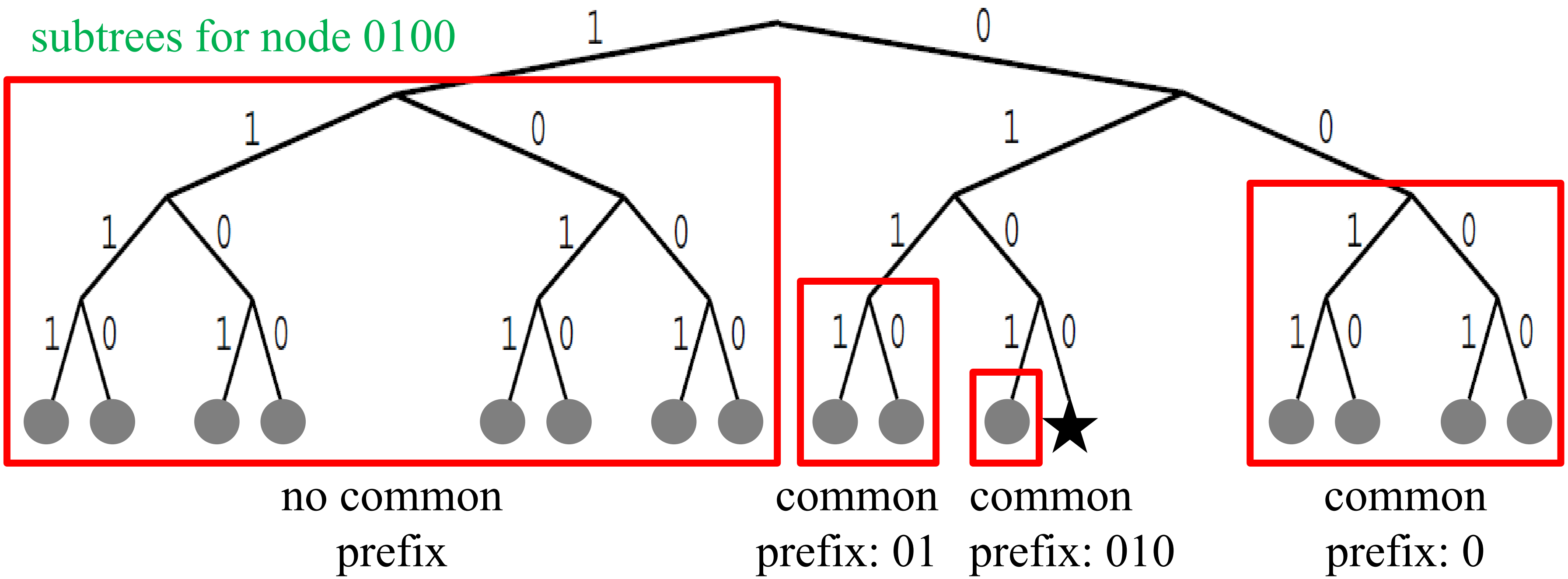
- Introduction
- Unstructured P2P systems
- **Structured P2P systems**
  - Chord
  - **Kademlia**

# 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) $\forall$x, y

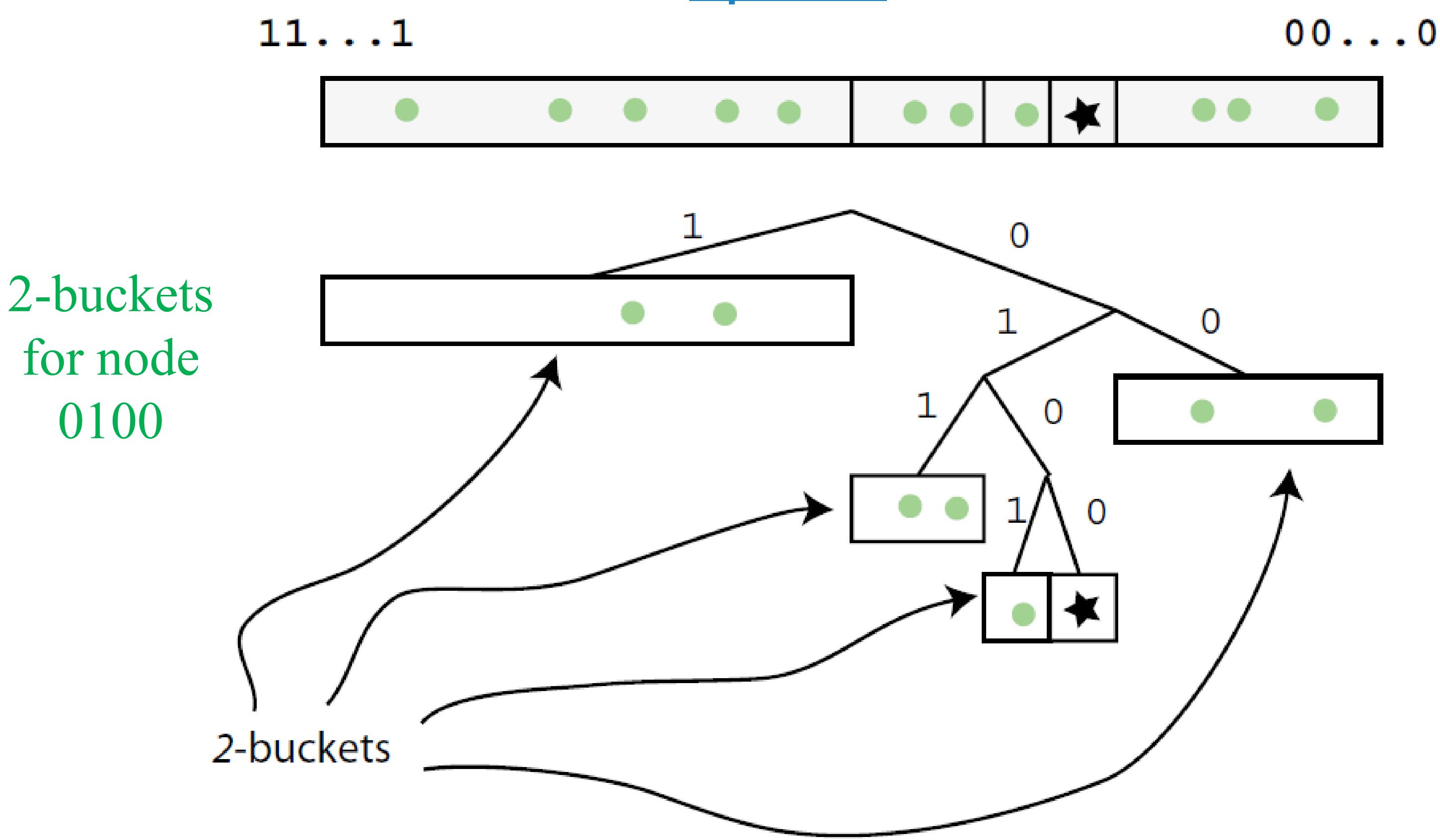
# Kademlia: Binary tree

- 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 not containing the node



# Kademlia: Node state

- For each of its subtrees, each node keeps a k-bucket: list of references for up to k nodes from that subtree



# Kademlia: Node state

- Formally, ∀i $0 \le i <$ [#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] |
| … | … | … | … | … |
| 159 | $[2^{159}, 2^{160})$ | [@IP, UDP port, node ID] | … | [@IP, UDP port, node ID] |

# Kademlia: Node state

- When P receives any message from Q, P updates the k-bucket that corresponds to Q:

  if Q is already in the k-bucket → move it to the tail
  else if the k-bucket is not full → insert Q at the tail
      else ping the least-recently-seen node in the k-bucket
        if it responds → move it to the tail and discard Q
        else evict it from the k-bucket and insert Q at the tail

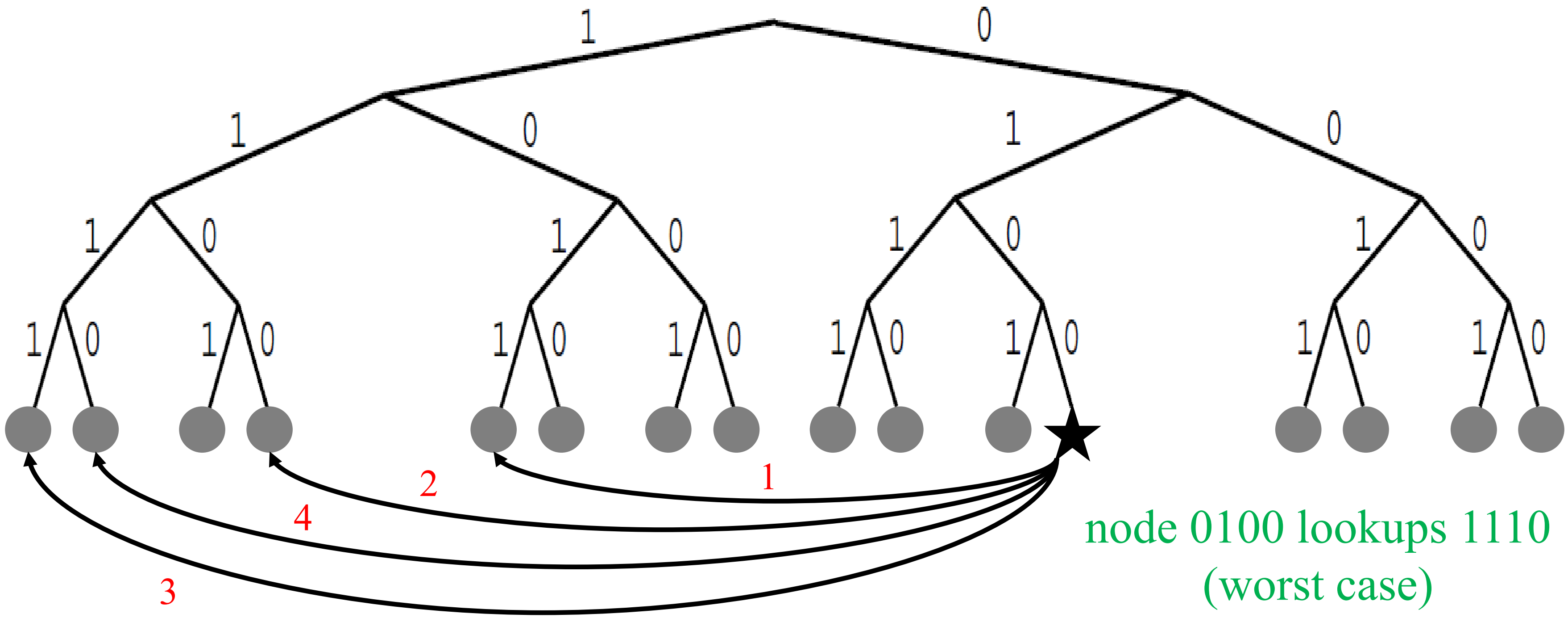
  ⇒ Keeps the oldest live nodes in the k-bucket (they will probably remain online)
    - Also, avoids some DoS attacks, as k-buckets cannot be flushed by flooding the system with new nodes

# 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 stored value

# 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



# Kademlia: Node lookup

- Use parallel routing to speed up lookups
- Node P performs a lookup of target key T
  1. P picks $\alpha$ nodes from its non-empty k-bucket closest to T (if that bucket has fewer than $\alpha$ entries, P picks nodes from other buckets)
     - $\alpha$ 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 $\alpha$ nodes selected
  3. P sends parallel, asynchronous FIND_NODE to the $\alpha$ nodes it has chosen

# 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 $\alpha$ nodes fails to reply, it is removed from the k-candidate list
6. P selects another $\alpha$ 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 the k-buckets generally fresh
  - If there is not traffic in the range of a k-bucket within an hour, the node **refreshes** 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 neighbors 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 key seen during the lookup that did not return the value

# 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 <u>republish</u> (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 <u>republishes</u> 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) <u>republish</u> 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 <u>metadata</u>
  - 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):
    - source key: hash of the content of the file
    - value: list of peers currently sharing the file
    - keyword key: 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

    $N$ is the total number of nodes
  - ↑ 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
- 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]: chapter 2.4
  - [Coulouris]: chapters 10 and 20.6.2