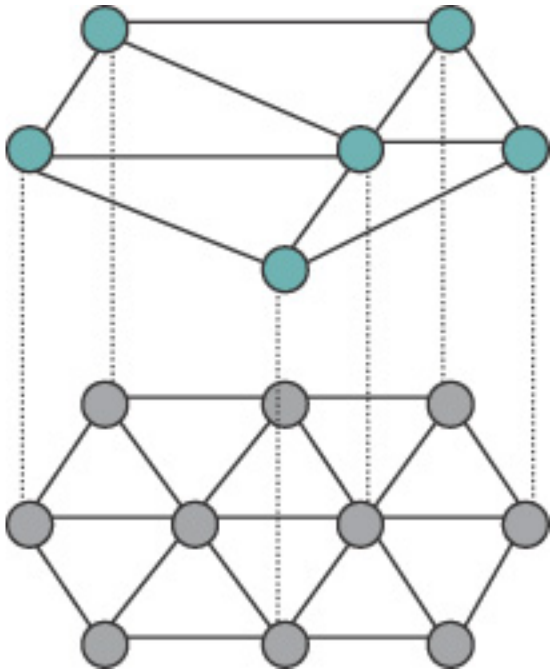
CompSci 356: Computer Network Architectures
Lecture 24: Overlay Networks
Chap 9.4

Xiaowei Yang
xwy@cs.duke.edu

Overview

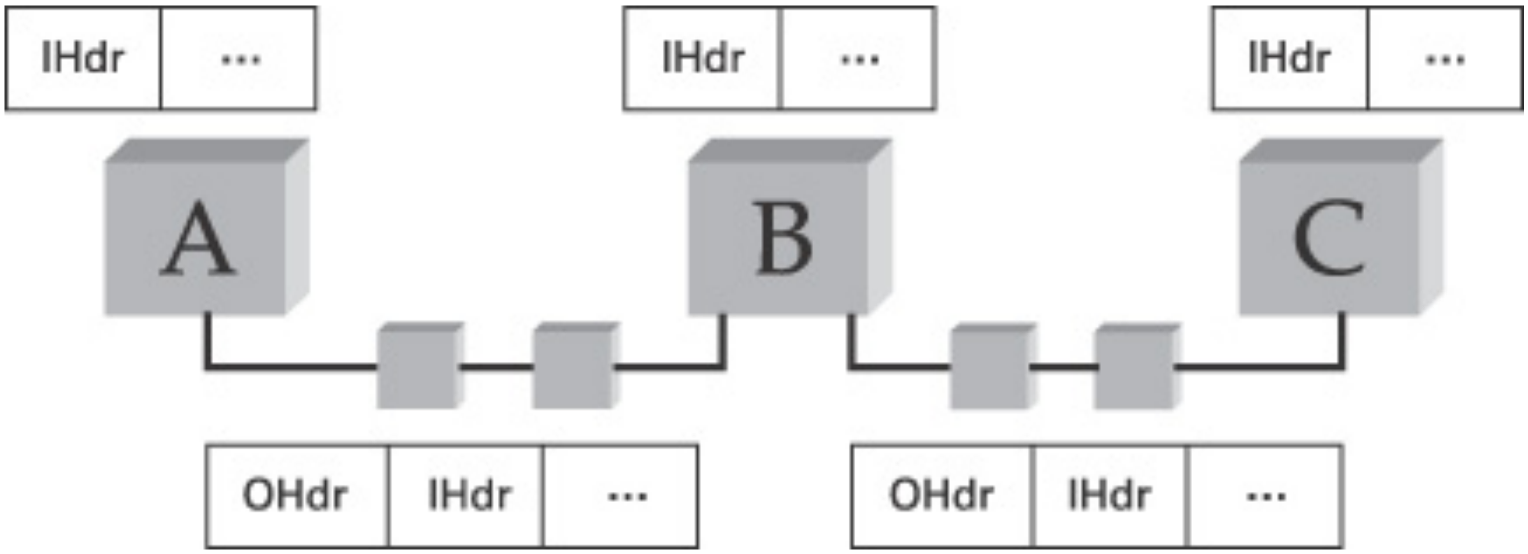
- What is an overlay network?
- Examples of overlay networks
  - End system multicast
  - Unstructured
    - Gnutella, BitTorrent
  - Structured
    - DHT

What is an overlay network?



- A logical network implemented on top of a lower-layer network
- Can recursively build overlay networks
- An overlay link is defined by the application
- An overlay link may consist of multi hops of underlay links

Ex: Virtual Private Networks



- Links are defined as IP tunnels
- May include multiple underlying routers

Other overlays

- The Onion Router (Tor)
- Resilient Overlay Networks (RoN)
  - Route through overlay nodes to achieve better performance
- End system multicast

Unstructured Overlay Networks

- Overlay links form random graphs
- No defined structure
- Examples
  - Gnutella: links are peer relationships
    - One node that runs Gnutella knows some other Gnutella nodes
  - BitTorrent
    - A node and nodes in its view

Peer-to-Peer Cooperative Content Distribution

- Use the client's upload bandwidth
  - infrastructure-less
- Key challenges
  - How to find a piece of data
  - How to incentivize uploading

Data lookup

- Centralized approach
  - Napster
  - BitTorrent trackers

- Distributed approach
  - Flooded queries
    - Gnutella
  - Structured lookup
    - DHT

Gnutella

- All nodes are true peers
  - A peer is the publisher, the uploader, and the downloader
  - No single point of failure
- A node knows other nodes as it neighbors
- How to find an object
  - Send queries to neighbors
  - Neighbors forward to their neighbors
  - Results travel backward to the sender
  - Use query IDs to match responses and to avoid loops

Gnutella

- Challenges
  - Efficiency and scalability issue
    - File searches span across many nodes → generate much traffic
  - Integrity (content pollution)
    - Anyone can claim that he publishes valid content
    - No guarantee of quality of objects
  - Incentive issue
    - No incentive for cooperation → free riding

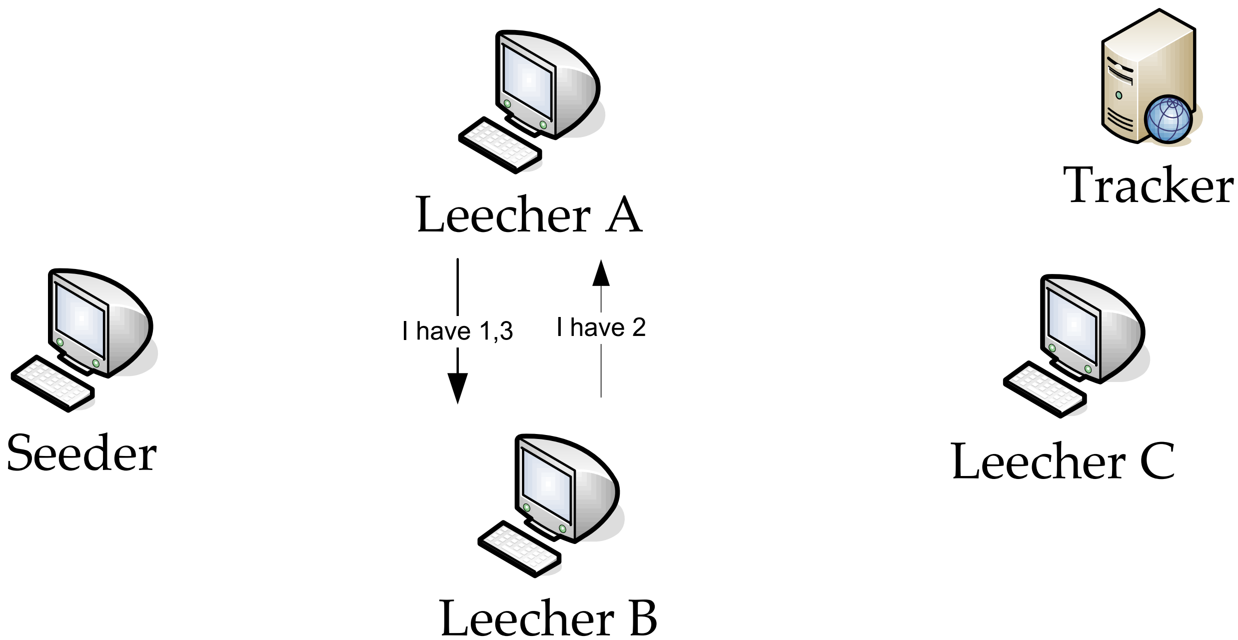
BitTorrent

- Designed by Bram Cohen
- Tracker for peer lookup
  - Later trackerless
- Rate-based Tit-for-tat for incentives

Terminology

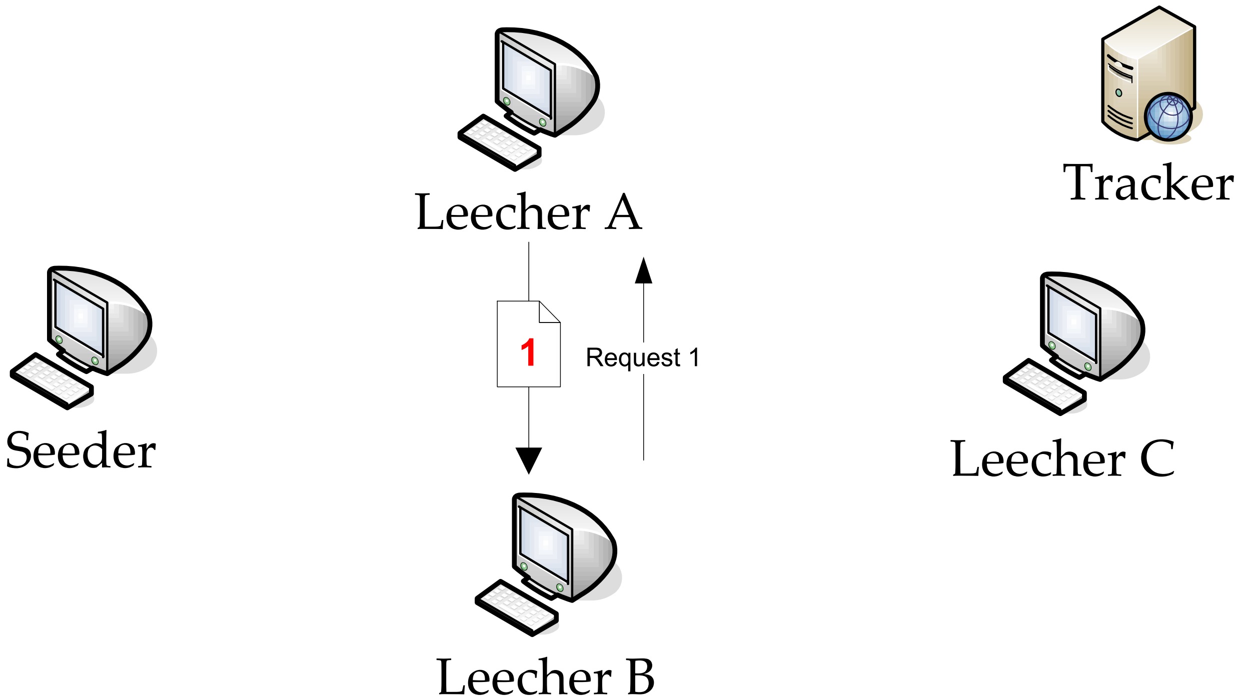
- Seeder: peer with the entire file
  - Original Seed: The first seed
- Leecher: peer that's downloading the file
  - Fairer term might have been "downloader"
- Piece: a large file is divided into pieces
- Sub-piece: Further subdivision of a piece
  - The "unit for requests" is a sub piece
  - But a peer uploads only after assembling complete piece
- Swarm: peers that download/upload the same file

BitTorrent overview



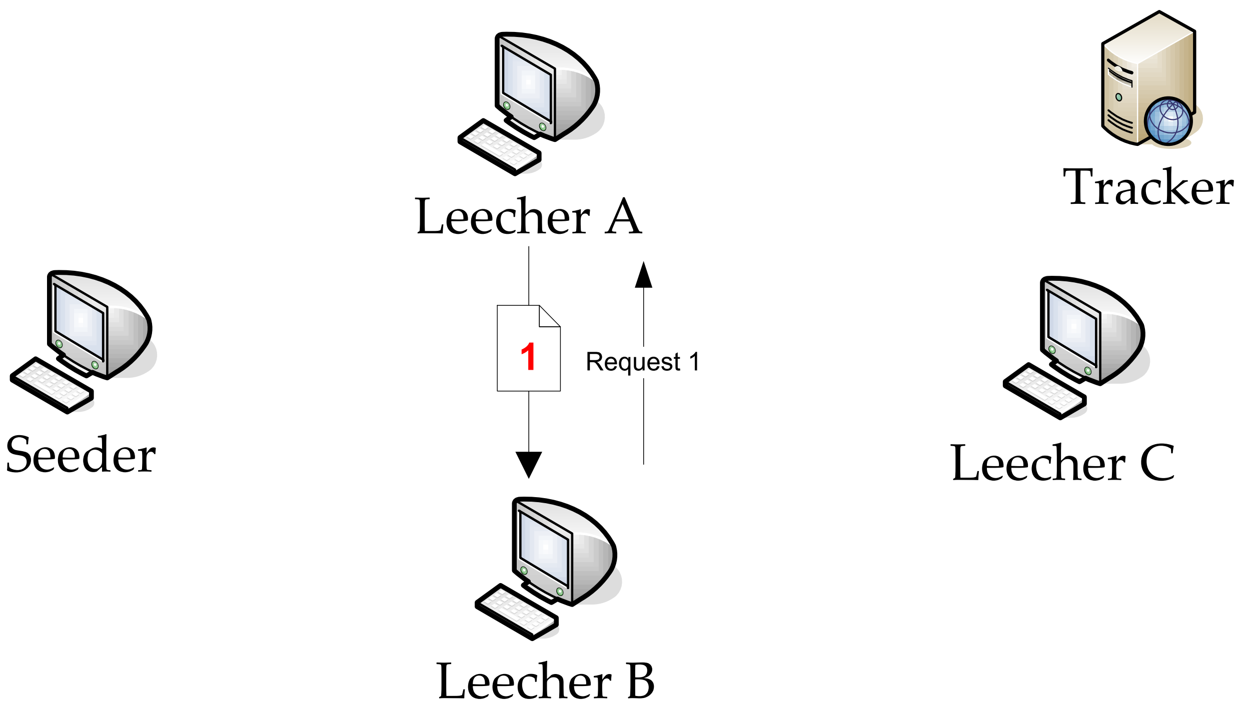
- A node announces available chunks to their peers
- **Leechers** request chunks from their peers (**locally rarest-first**)

BitTorrent overview



- **Leechers** request chunks from their peers (**locally rarest-first**)

BitTorrent overview



- **Leechers** request chunks from their peers (**locally rarest-first**)
- **Leechers** choke slow peers (tit-for-tat)
    - Keeps at most four peers. Three fastest, one random chosen (optimistic unchoke)

Optimistic Unchoking

- Discover other faster peers and prompt them to reciprocate
- Bootstrap new peers with no data to upload

Scheduling: Choosing pieces to request

- Rarest-first: Look at all pieces at all peers, and request piece that's owned by fewest peers
  1. Increases diversity in the pieces downloaded
     - avoids case where a node and each of its peers have exactly the same pieces; increases throughput
  2. Increases likelihood all pieces still available even if original seed leaves before any one node has downloaded the entire file
  3. Increases chance for cooperation

- Random rarest-first: rank rarest, and randomly choose one with equal rareness

Start time scheduling

- Random First Piece:
  - When peer starts to download, request random piece.
    - So as to assemble first complete piece quickly
    - Then participate in uploads
    - May request sub pieces from many peers
  - When first complete piece assembled, switch to rarest-first

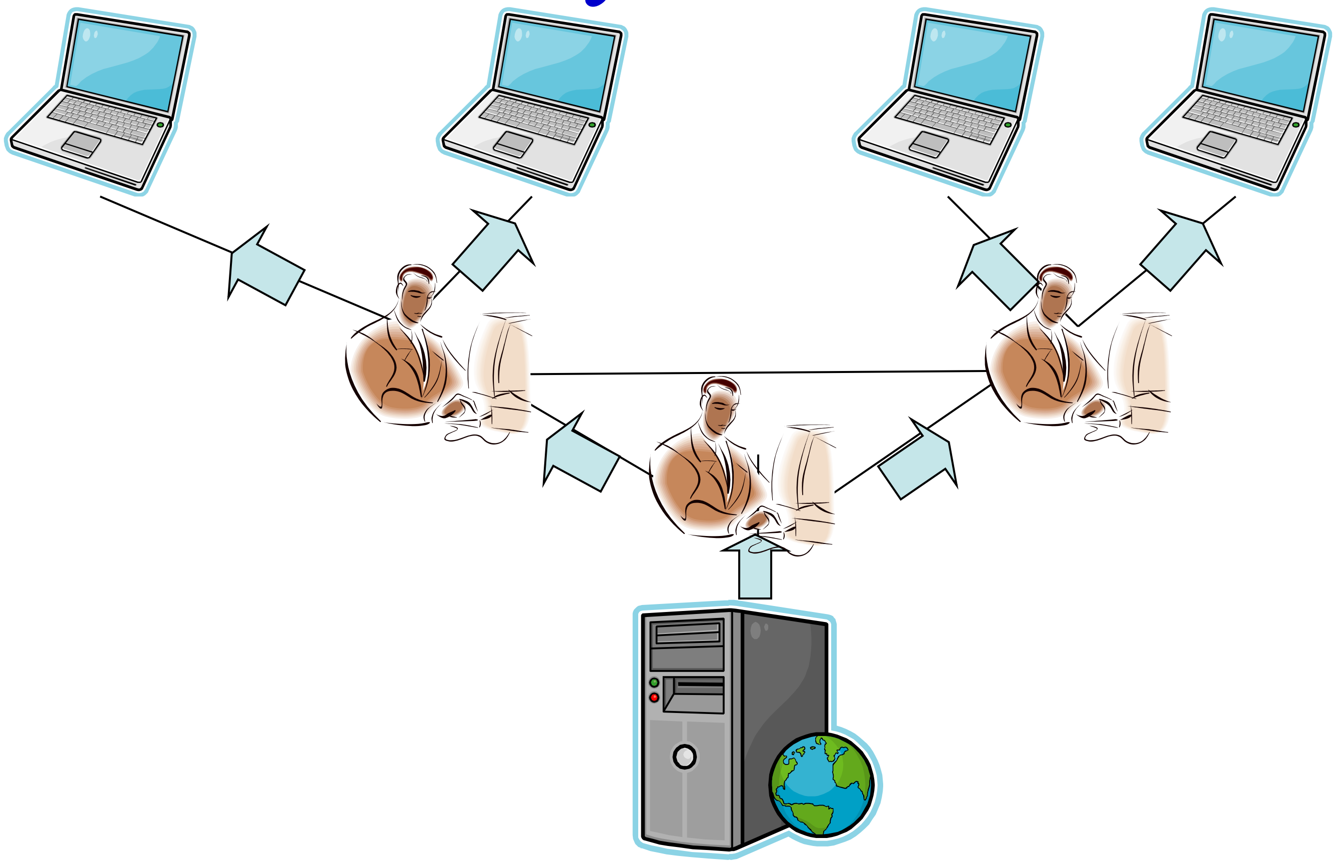
Choosing pieces to request

- End-game mode:
  - When requests sent for all sub-pieces, (re)send requests to all peers.
  - To speed up completion of download
  - Cancel requests for downloaded sub-pieces

Overview

- Overlay networks
  - Unstructured
  - Structured
    - End systems multicast
    - Distributed Hash Tables

End system multicast



- End systems rather than routers organize into a tree, forward and duplicate packets
- Pros and cons

Structured Networks

- A node forms links with specific neighbors to maintain a certain structure of the network

- Pros
  - More efficient data lookup
  - More reliable

- Cons
  - Difficult to maintain the graph structure

- Examples
  - Distributed Hash Tables
  - End-system multicast: overlay nodes form a multicast tree

DHT Overview

- Used in the real world
  - BitTorrent tracker implementation
  - Content distribution networks
  - Many other distributed systems including botnets
- What problems do DHTs solve?
- How are DHTs implemented?

Background

- A hash table is a data structure that stores (key, object) pairs.
- Key is mapped to a table index via a hash function for fast lookup.
- Content distribution networks
  - Given an URL, returns the object

Example of a Hash table: a web cache

http://www.cnn.com	Page content
http://www.nytimes.com	…….
http://www.slashdot.org	…..
…	…
…	…

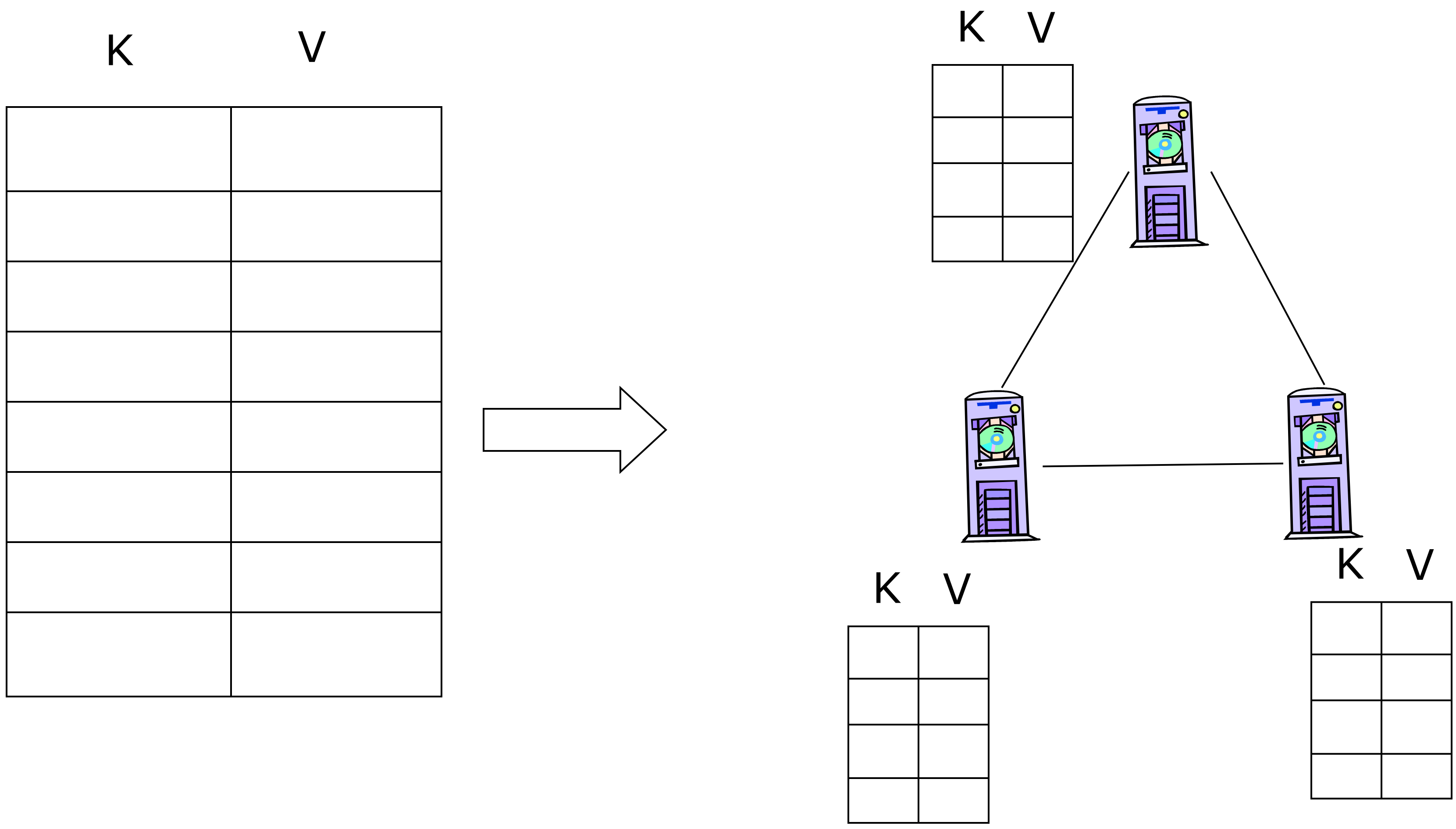
- Client requests http://www.cnn.com
- Web cache returns the page content located at the 1st entry of the table.

DHT: why?

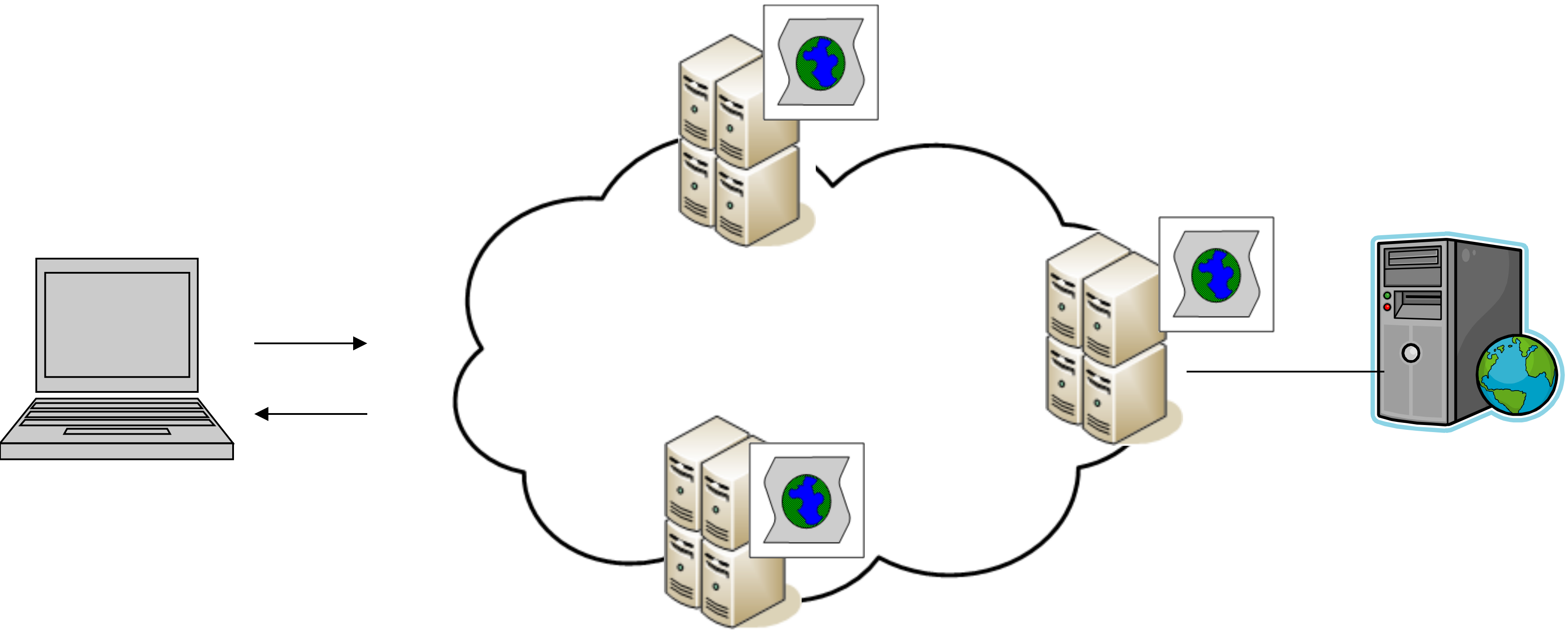
- If the number of objects is large, it is impossible for any single node to store it.

- Solution: distributed hash tables.
  - Split one large hash table into smaller tables and distribute them to multiple nodes

DHT



A content distribution network

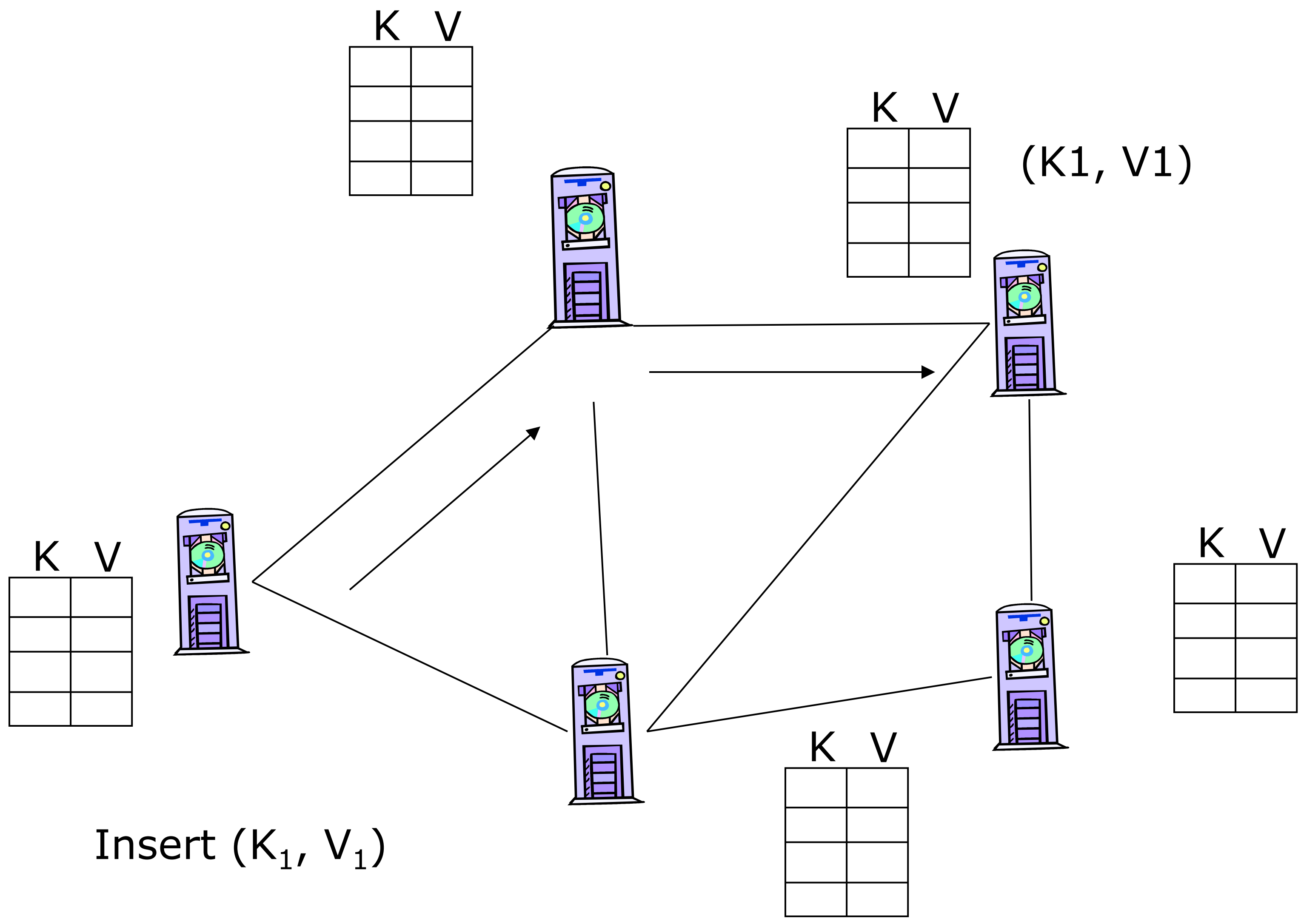


- A single provider that manages multiple replicas
- A client obtains content from a close replica

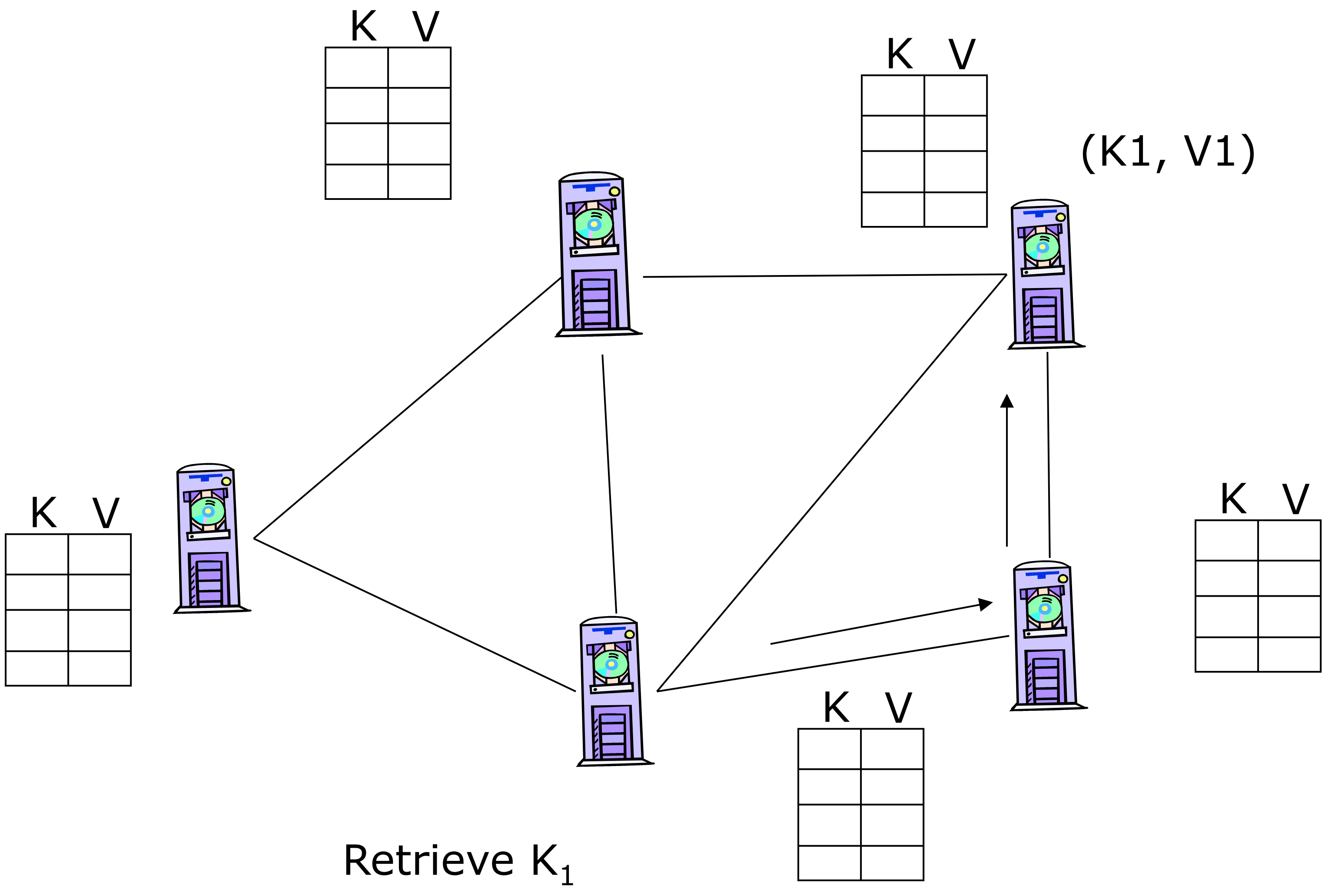
Basic function of DHT

- DHT is a “virtual” hash table
  - Input: a key
  - Output: a data item
- Data Items are stored by a network of nodes
- DHT abstraction
  - Input: a key
  - Output: the node that stores the key
- Applications handle key and data item association

DHT: a visual example



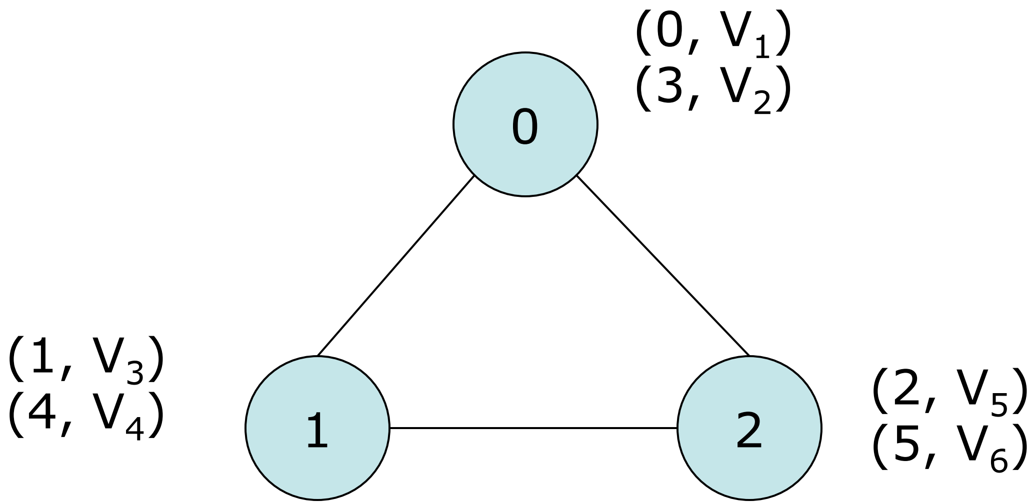
DHT: a visual example



Desired goals of DHT

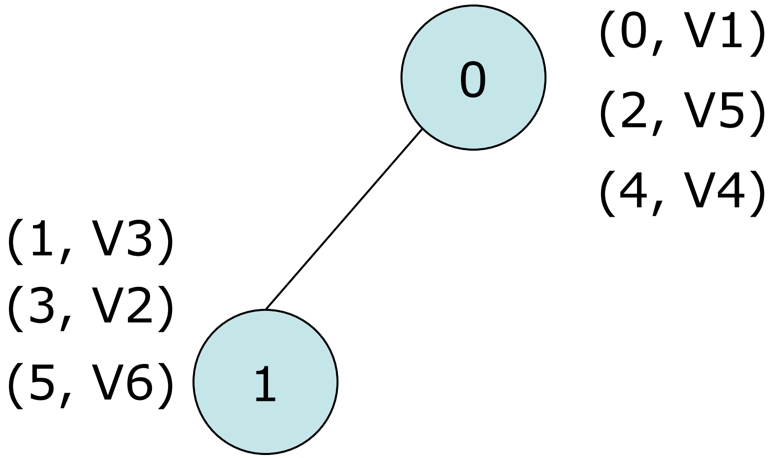
- Scalability: each node does not keep much state
- Performance: small look up latency
- Load balancing: no node is overloaded with a large amount of state
- Dynamic reconfiguration: when nodes join and leave, the amount of state moved from nodes to nodes is small.
- Distributed: no node is more important than others.

A straw man design



- Suppose all keys are integers
- The number of nodes in the network is n
- id = key % n

When node 2 dies



- A large number of data items need to be rehashed.

Fix: consistent hashing

- A node is responsible for a range of keys
  - When a node joins or leaves, the expected fraction of objects that must be moved is the minimum needed to maintain a balanced load.
  - All DHTs implement consistent hashing
  - They differ in the underlying “geometry”

Basic components of DHTs

- Overlapping key and node identifier space
  - Hash(www.cnn.com/image.jpg) → a n-bit binary string
  - Nodes that store the objects also have n-bit string as their identifiers
- Building routing tables
  - Next hops (structure of a DHT)
  - Distance functions
  - These two determine the geometry of DHTs
    - Ring, Tree, Hybercubes, hybrid (tree + ring) etc.
  - Handle nodes join and leave
- Lookup and store interface

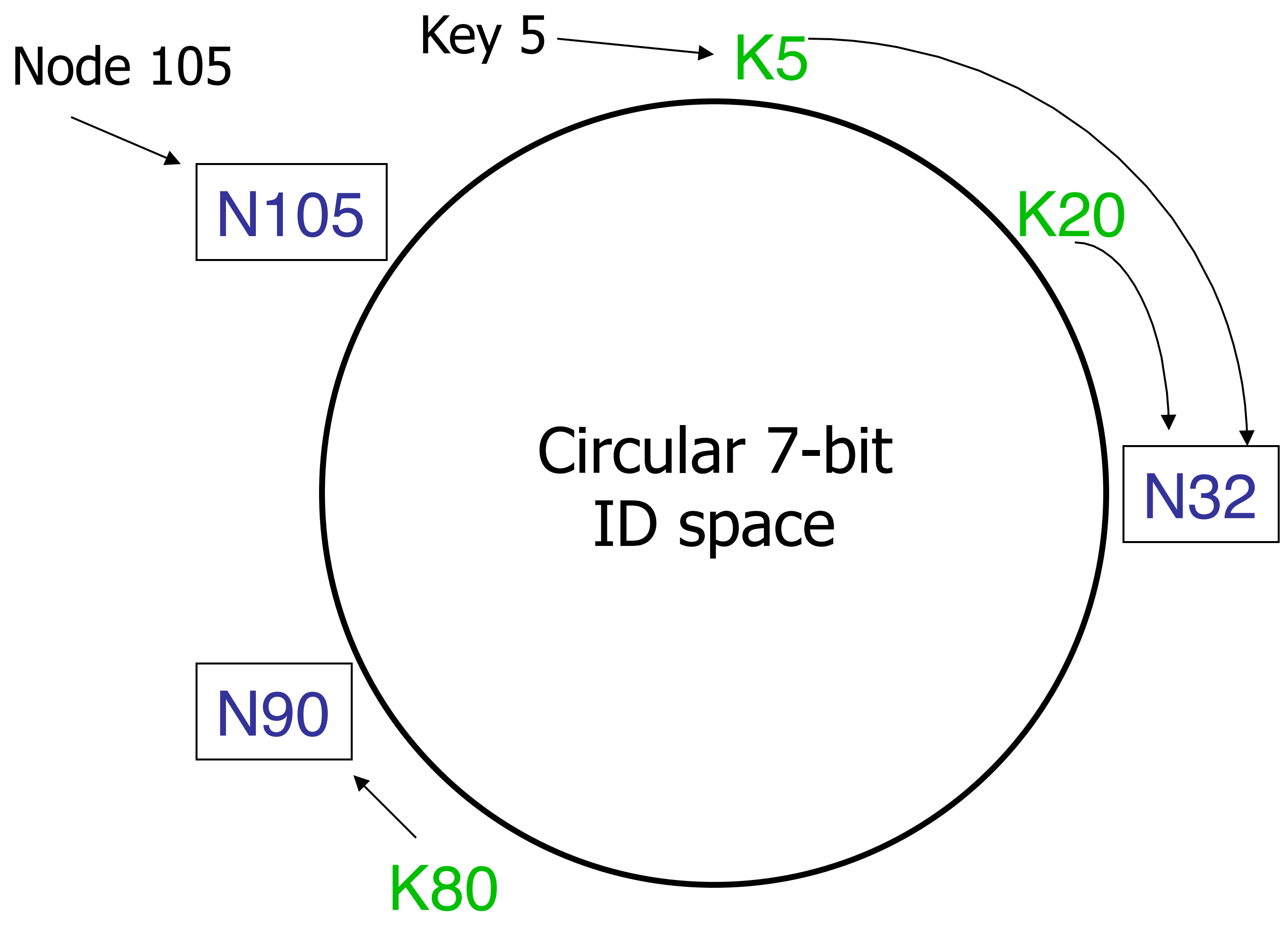
Case study: Chord

Note: textbook uses Pastry

Chord: basic idea

- Hash both node id and key into a m-bit one-dimension circular identifier space
- Consistent hashing: a key is stored at a node whose identifier is closest to the key in the identifier space
  - Key refers to both the key and its hash value.

Chord: ring topology

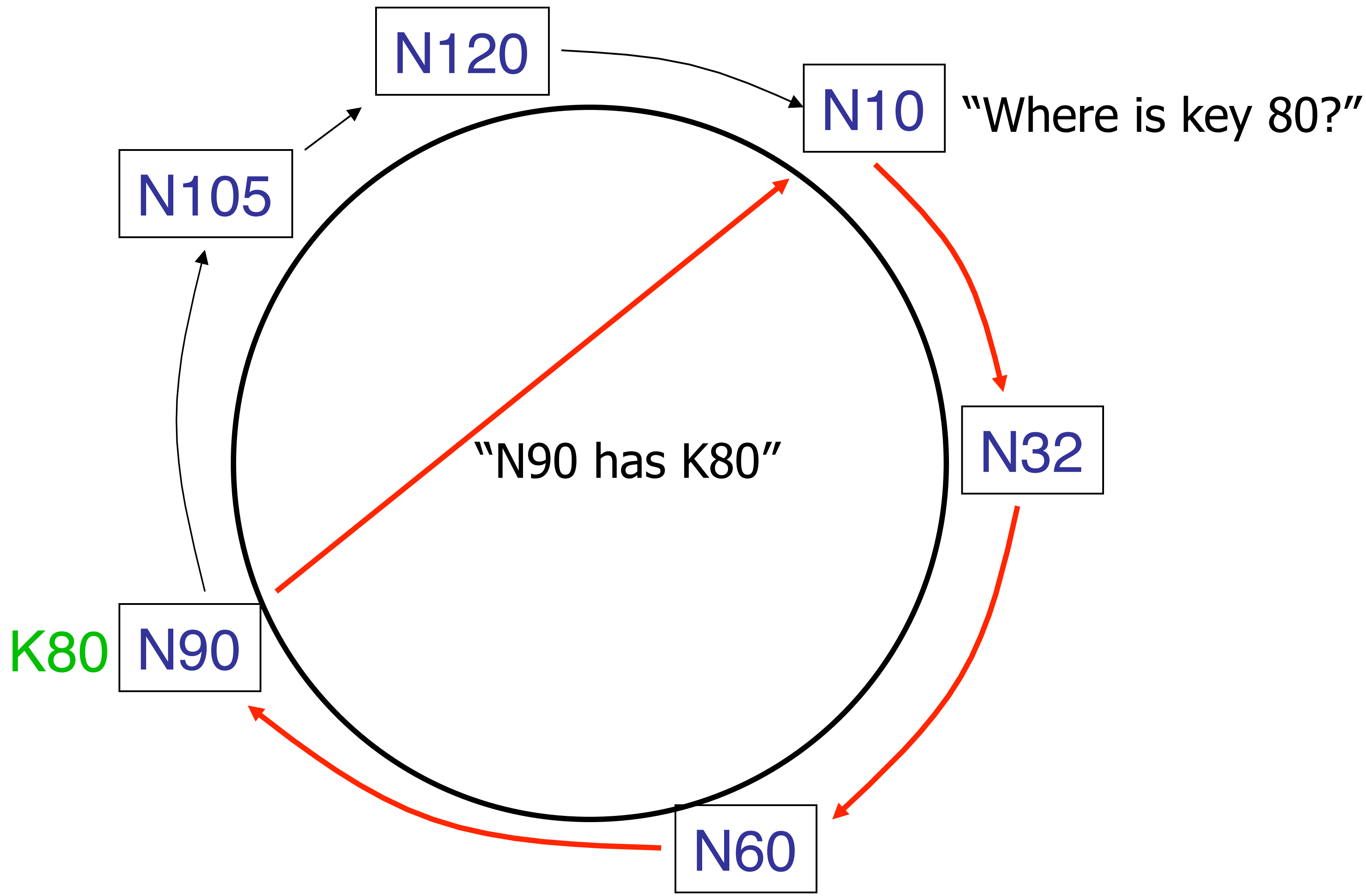


A key is stored at its successor: node with next higher ID

Chord: how to find a node that stores a key?

- Solution 1: every node keeps a routing table to all other nodes
  - Given a key, a node knows which node id is successor of the key
  - The node sends the query to the successor
  - What are the advantages and disadvantages of this solution?

Solution 2: every node keeps a routing entry to the node's successor (a linked list)

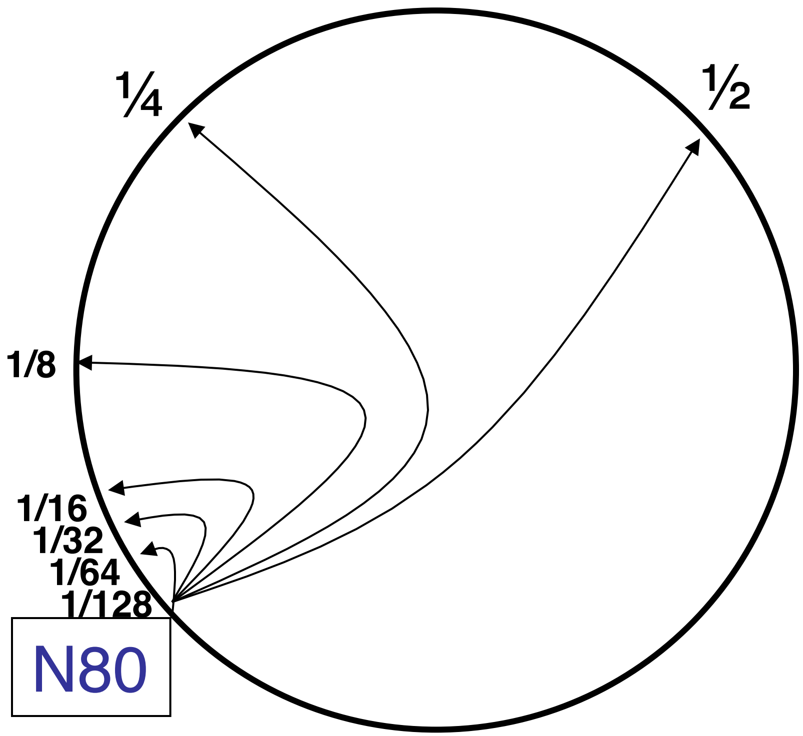


Simple lookup algorithm

```
Lookup(my-id, key-id)
    n = my successor
    if my-id < n < key-id
        call Lookup(key-id) on node n   // next hop
    else
        return my successor             // done
```

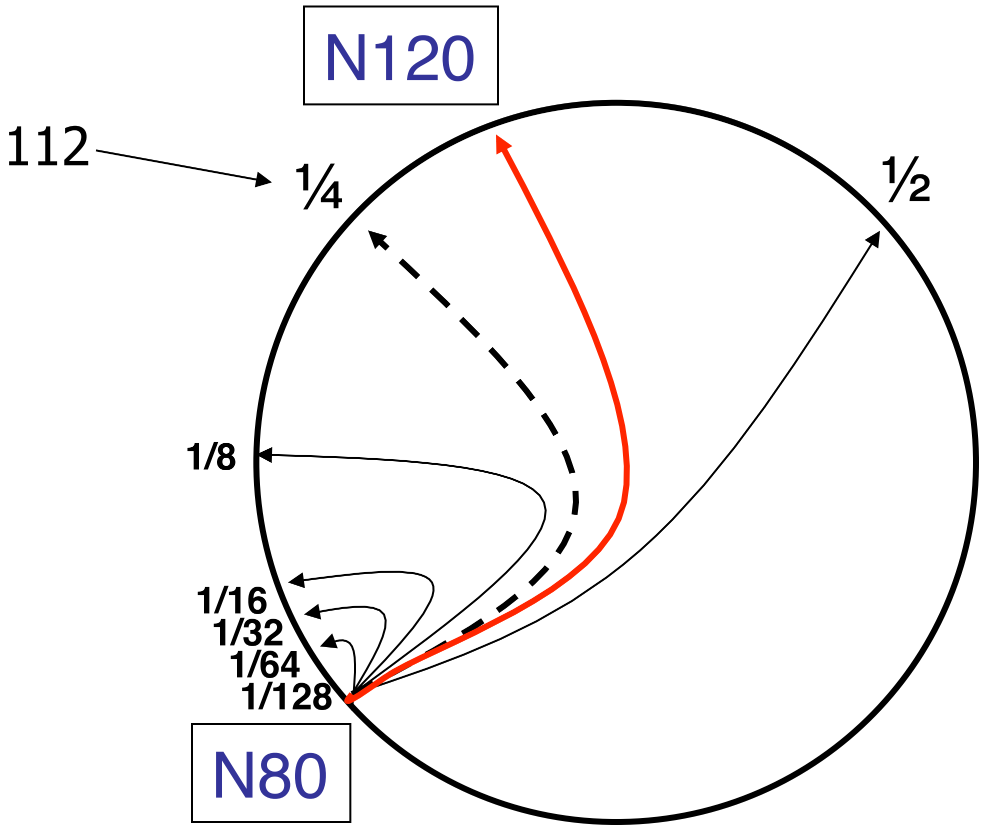
- Correctness depends only on successors
- Q1: will this algorithm miss the real successor?
- Q2: what's the average # of lookup hops?

Solution 3: "Finger table" allows log(N)-time lookups



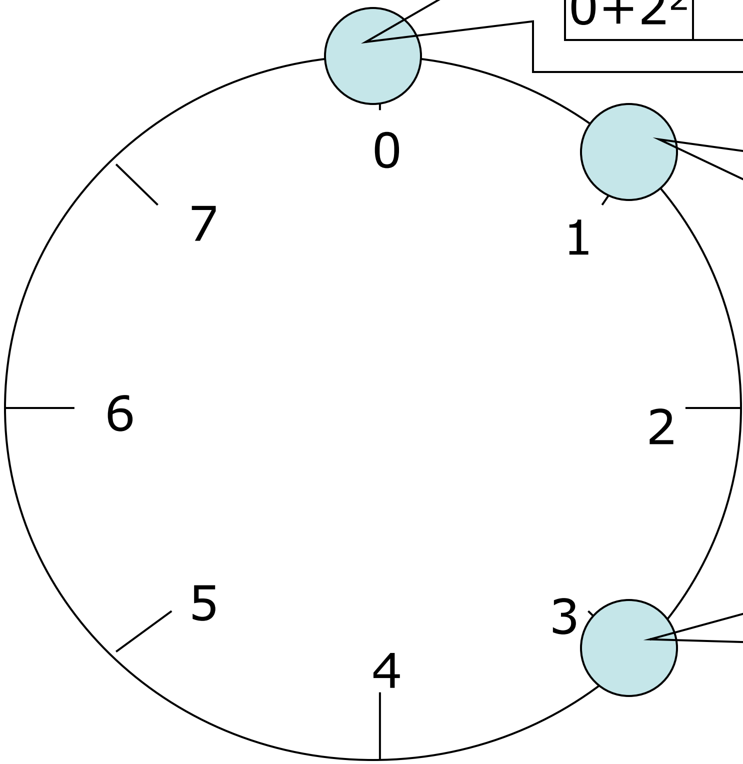
- Analogy: binary search

Finger i points to successor of $n+2^{i-1}$



- The ith entry in the table at node n contains the identity of the *first* node s that succeeds n by at least 2^{i-1}
- A finger table entry includes Chord Id and IP address
- Each node stores a small table log(N)

Chord finger table example



$0+2^0$	1
$0+2^1$	3
$0+2^2$	0

Keys:
5,6

2	3
3	3
5	0

Keys:
1

4	0
5	0
7	0

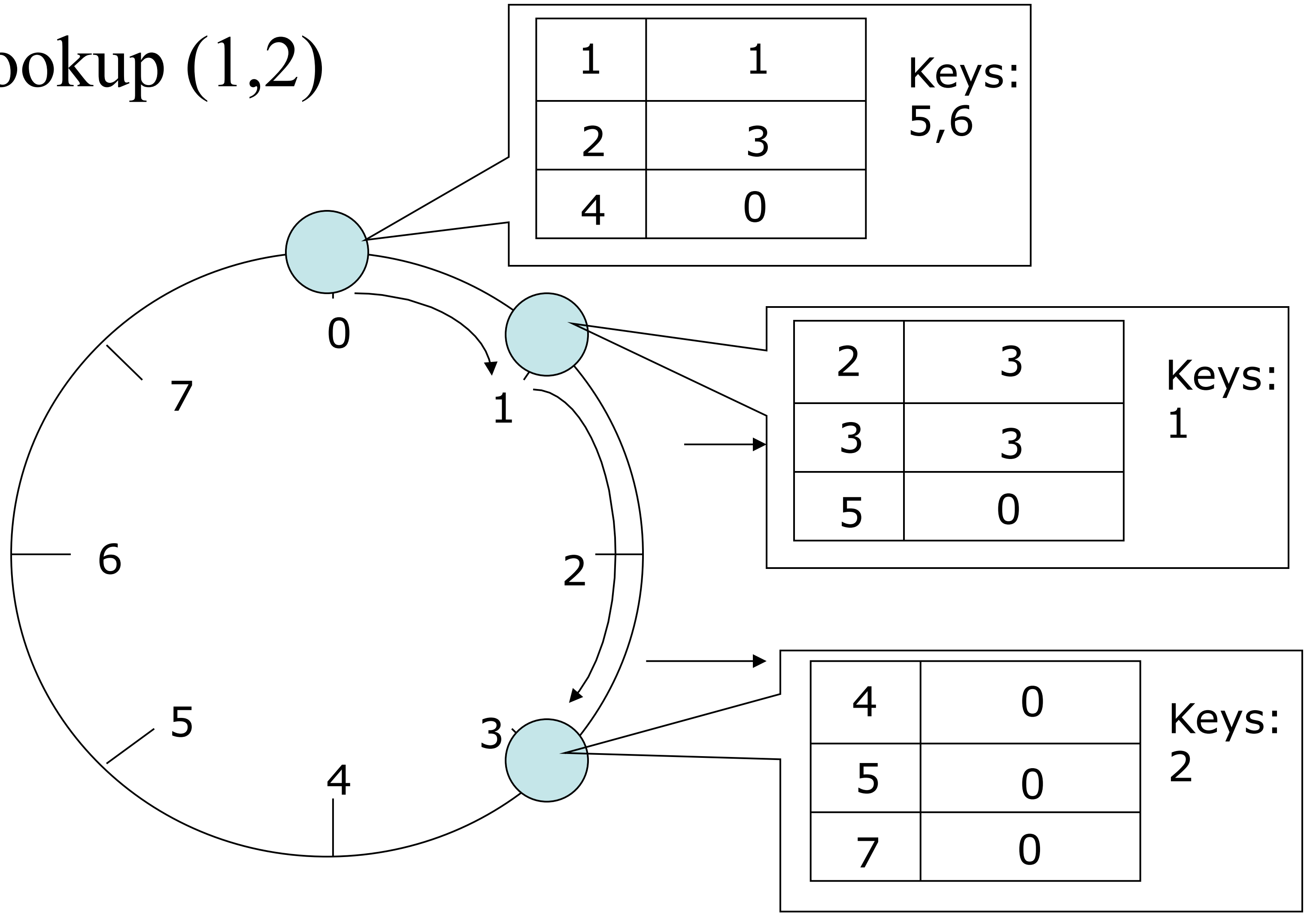
Keys:
2

Lookup with fingers

```
Lookup(my-id, key-id)
    If key-id in my storage
        return my-value;
    else
        look in local finger table for
            highest node n s.t. my-id < n < key-id
        if n exists
        call Lookup(key-id) on node n   // next hop
    else
        return my successor             // done
```

Chord lookup example

- Lookup (1,2)



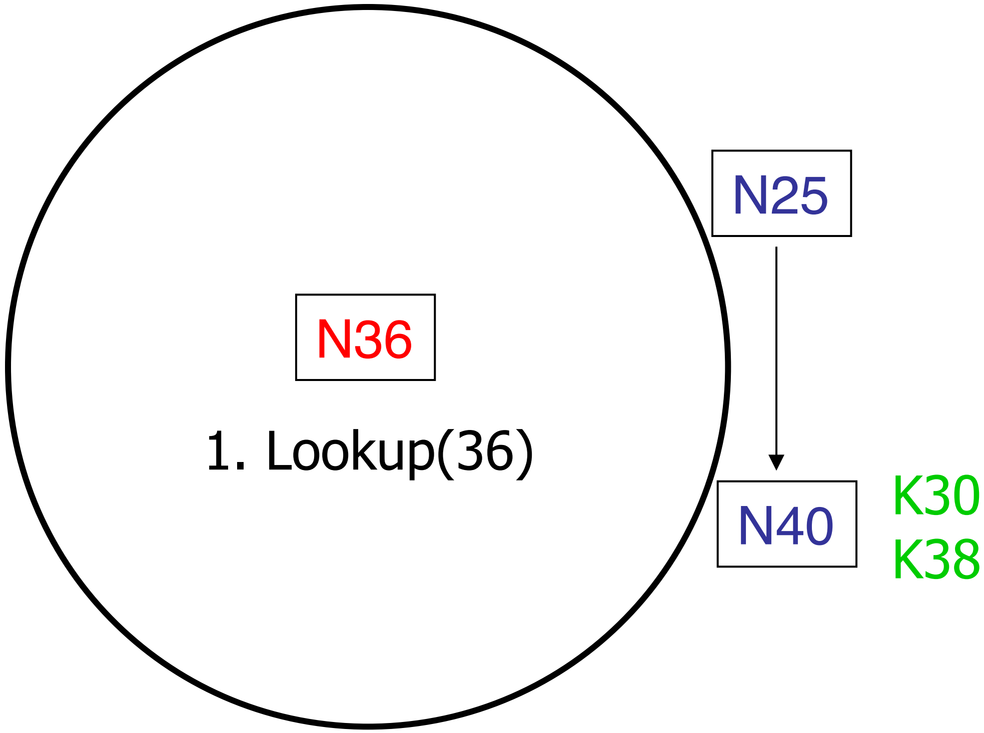
Node join

- Maintain the invariant
  1. Each node’ successor is correctly maintained
  2. For every node k, node successor(k) answers for key k. It’s desirable that finger table entries are correct

- Each nodes maintains a predecessor pointer

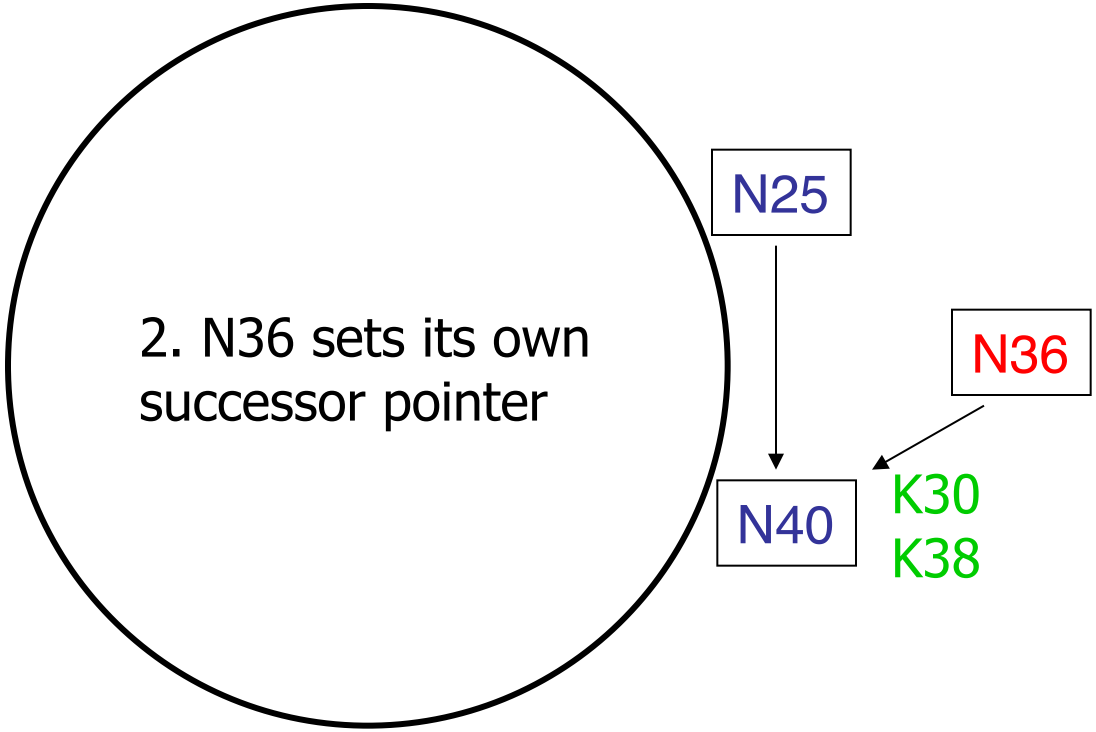
- Tasks:
  - Initialize predecessor and fingers of new node
  - Update existing nodes’ state
  - Notify apps to transfer state to new node

Chord Joining: linked list insert

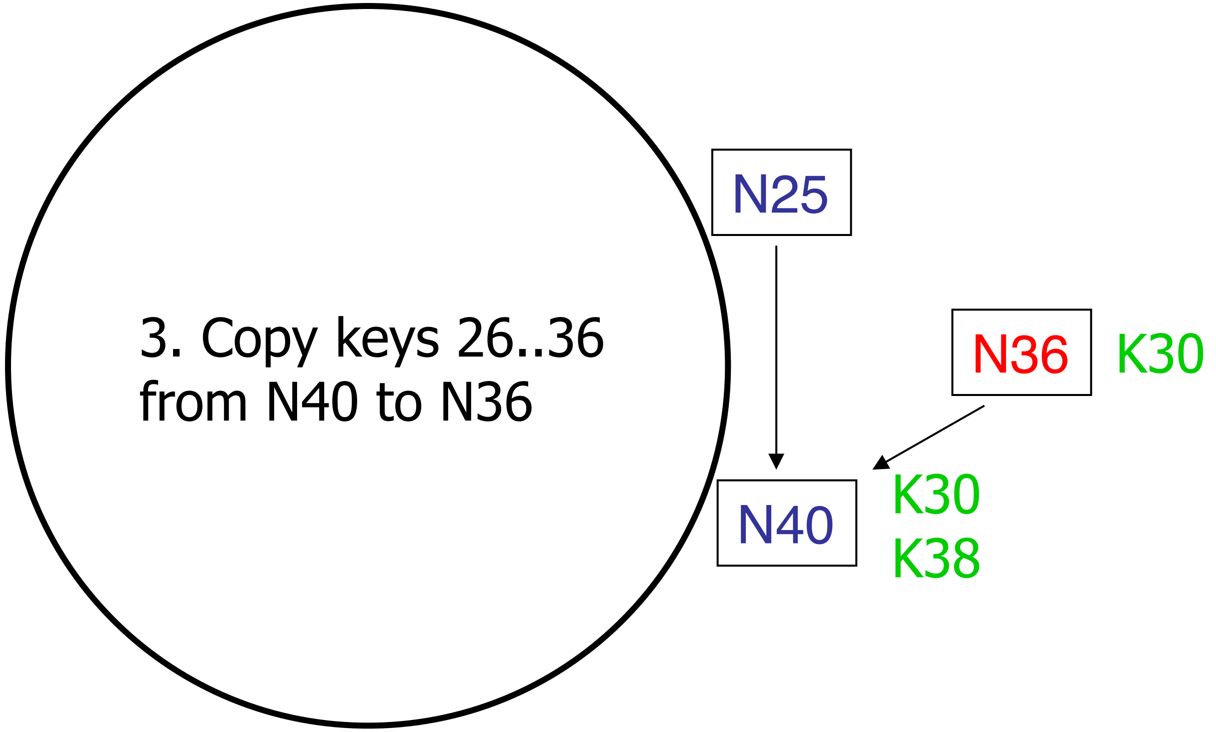


- Node n queries a known node n' to initialize its state
- Look up for its successor: lookup (n)

Join (2)

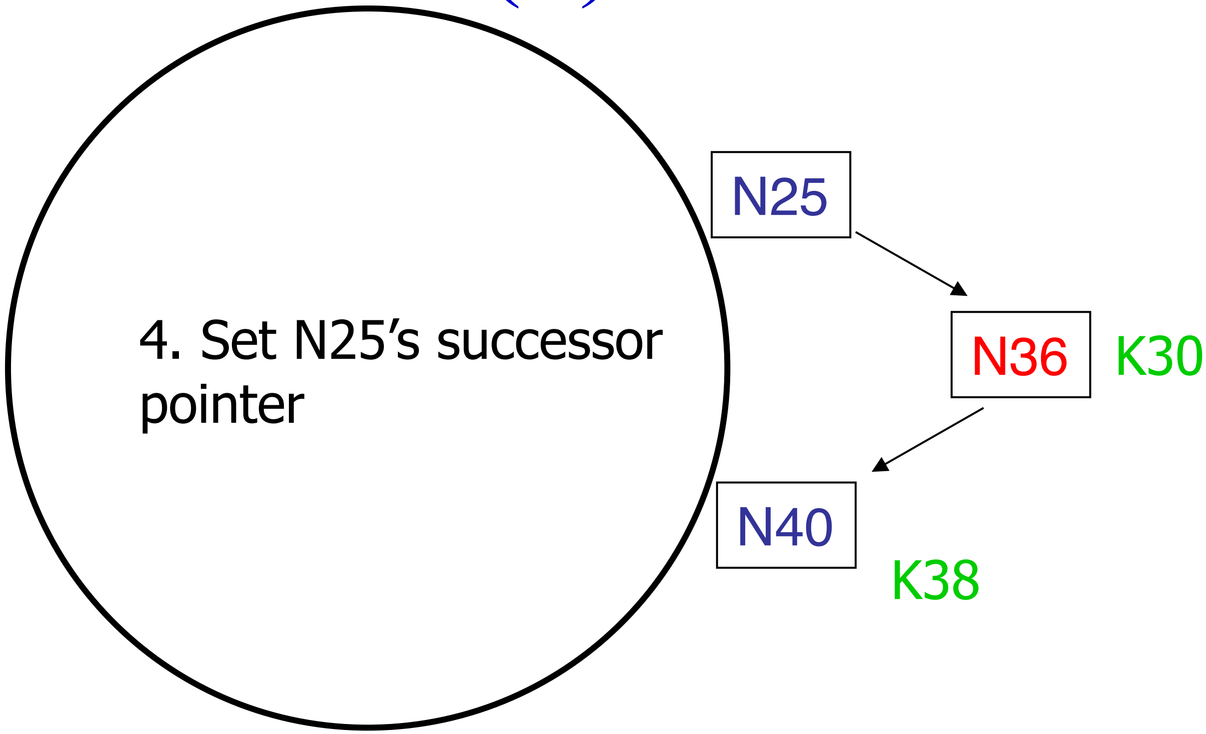


Join (3)



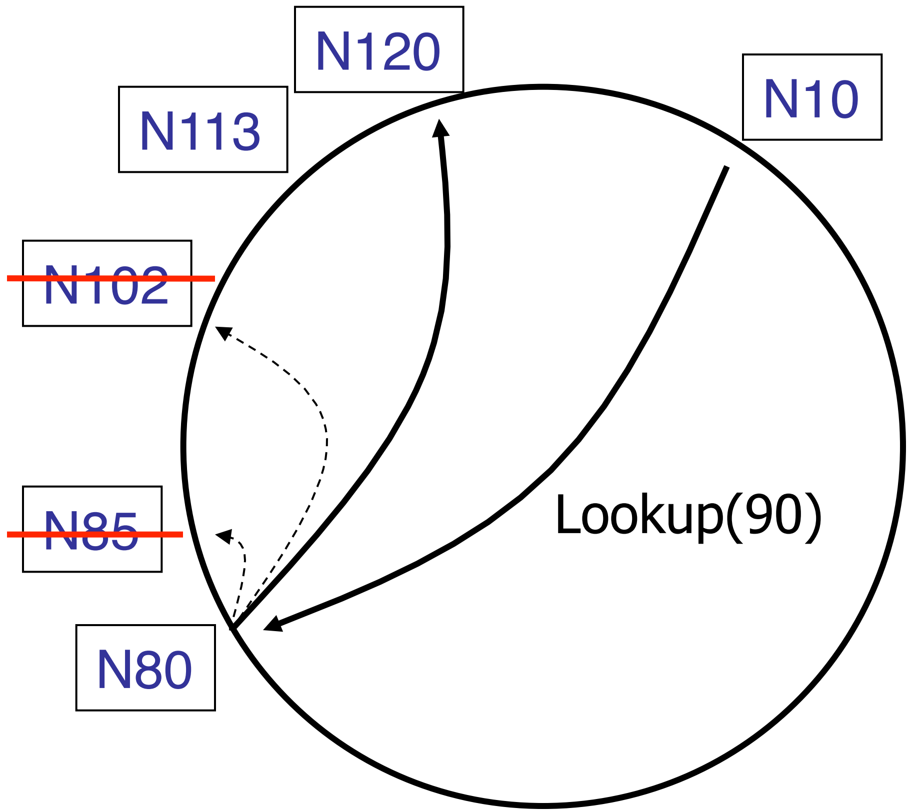
- Note that join does not make the network aware of n

Join (4): stabilize



- Stabilize 1) obtains a node n’s successor’s predecessor x, and determines whether x should be n’s successor 2) notifies n’s successor n’s existence
  - N25 calls its successor N40 to return its predecessor
  - Set its successor to N36
  - Notifies N36 it is predecessor
- Update finger pointers in the background periodically
  - Find the successor of each entry I
- Correct successors produce correct lookups

Failures might cause incorrect lookup



N80 doesn't know correct successor, so incorrect lookup

Solution: successor lists

- Each node knows *r* immediate successors
- After failure, will know first live successor
- Correct successors guarantee correct lookups

- Guarantee is with some probability

- Higher layer software can be notified to duplicate keys at failed nodes to live successors

Choosing the successor list length

- Assume 1/2 of nodes fail
- P(successor list all dead) = $(1/2)^r$
  - I.e. P(this node breaks the Chord ring)
  - Depends on independent failure
- P(no broken nodes) = $(1 - (1/2)^r)^N$
  - $r = 2log(N)$ makes prob. = $1 - 1/N$

Lookup with fault tolerance

```
Lookup(my-id, key-id)
    look in local finger table and successor-list
        for highest node n s.t. my-id < n < key-id
    if n exists
        call Lookup(key-id) on node n   // next hop
        if call failed,
            remove n from finger table
            return Lookup(my-id, key-id)
    else return my successor            // done
```

Chord performance

- Per node storage
  - Ideally: K/N
  - Implementation: large variance due to unevenly node id distribution
- Lookup latency
  - O(logN)

Comments on Chord

- ID distance ≠ Network distance
  - Reducing lookup latency and locality
- Strict successor selection
  - Can't overshoot
- Asymmetry
  - A node does not learn its routing table entries from queries it receives
- Later work fixes these issues

Conclusion

- Overlay networks
  - Structured vs Unstructured

- Design of DHTs
  - Chord

Lab 3 Congestion Control

- This lab is based on Lab 1, you don’t have to change much to make it work.
- You are required to implement a congestion control algorithm
  - Fully utilize the bandwidth
  - Share the bottleneck fairly
  - Write a report to describe your algorithm design and performance analysis
- You may want to implement at least
  - Slow start
  - Congestion avoidance
  - Fast retransmit and fast recovery
  - RTO estimator
  - New RENO is a plus. It handles multiple packets loss very well.

Lab 3 Congestion Control