Content Overlays (continued)

Nick Feamster
CS 7260
March 26, 2007

Administrivia

- Quiz date
- Remaining lectures
- Interim report
- PS 3
  - Out Friday, 1-2 problems

Structured vs. Unstructured Overlays

- Structured overlays have provable properties
  - Guarantees on storage, lookup, performance
- Maintaining structure under churn has proven to be difficult
  - Lots of state that needs to be maintained when conditions change
- Deployed overlays are typically *unstructured*

Structured [Content] Overlays

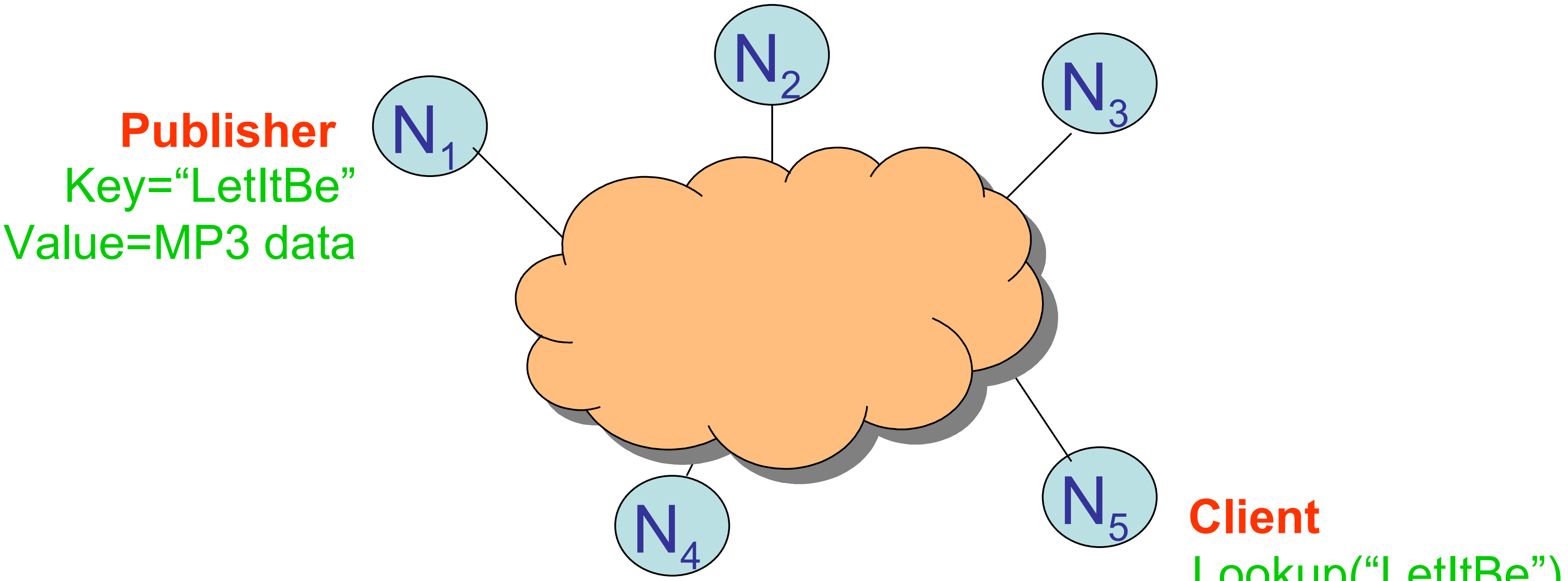
Chord: Overview

- What is Chord?
  - A scalable, distributed “lookup service”
  - **Lookup service:** A service that maps keys to values (*e.g.*, DNS, directory services, etc.)
  - **Key technology:** Consistent hashing

- Major benefits of Chord over other lookup services
  - Simplicity
  - Provable correctness
  - Provable “performance”

Chord: Primary Motivation

Scalable location of data in a large distributed system



Key Problem: Lookup

Chord: Design Goals

- Load balance: Chord acts as a distributed hash function, spreading keys evenly over the nodes.
- Decentralization: Chord is fully distributed: no node is more important than any other.
- Scalability: The cost of a Chord lookup grows as the log of the number of nodes, so even very large systems are feasible.
- Availability: Chord automatically adjusts its internal tables to reflect newly joined nodes as well as node failures, ensuring that, the node responsible for a key can always be found.
- Flexible naming: Chord places no constraints on the structure of the keys it looks up.

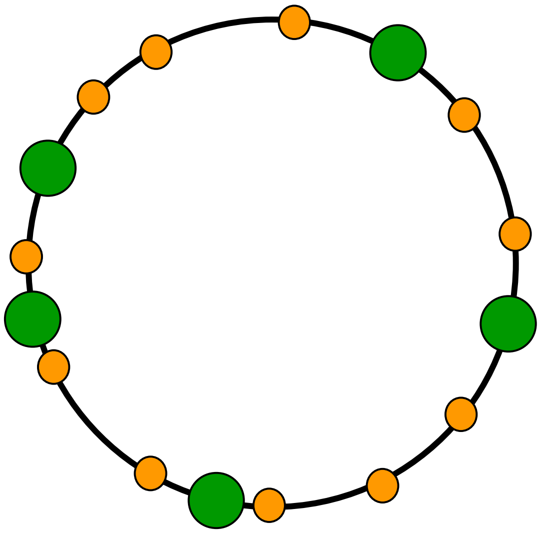
Consistent Hashing

- **Uniform Hash:** assigns values to "buckets"
  - *e.g., H(key) = f(key) mod k,* where *k* is number of nodes
  - Achieves load balance if keys are randomly distributed
- **Problems with uniform hashing**
  - How to perform consistent hashing in a distributed fashion?
  - What happens when nodes join and leave?

Consistent hashing addresses these problems

Consistent Hashing

- **Main idea:** map both keys and nodes (node IPs) to the same (metric) ID space



Ring is one option.
Any metric space will do

Initially proposed for relieving Web cache hotspots [Karger97, STOC]

Consistent Hashing

- The consistent hash function assigns each node and key an *m*-bit identifier using SHA-1 as a base hash function

- **Node identifier:** SHA-1 hash of IP address

- **Key identifier:** SHA-1 hash of key

Chord Identifiers

- *m* bit identifier space for both keys and nodes
- **Key identifier:** SHA-1(key)

Key="LetItBe" —SHA-1→ ID=60

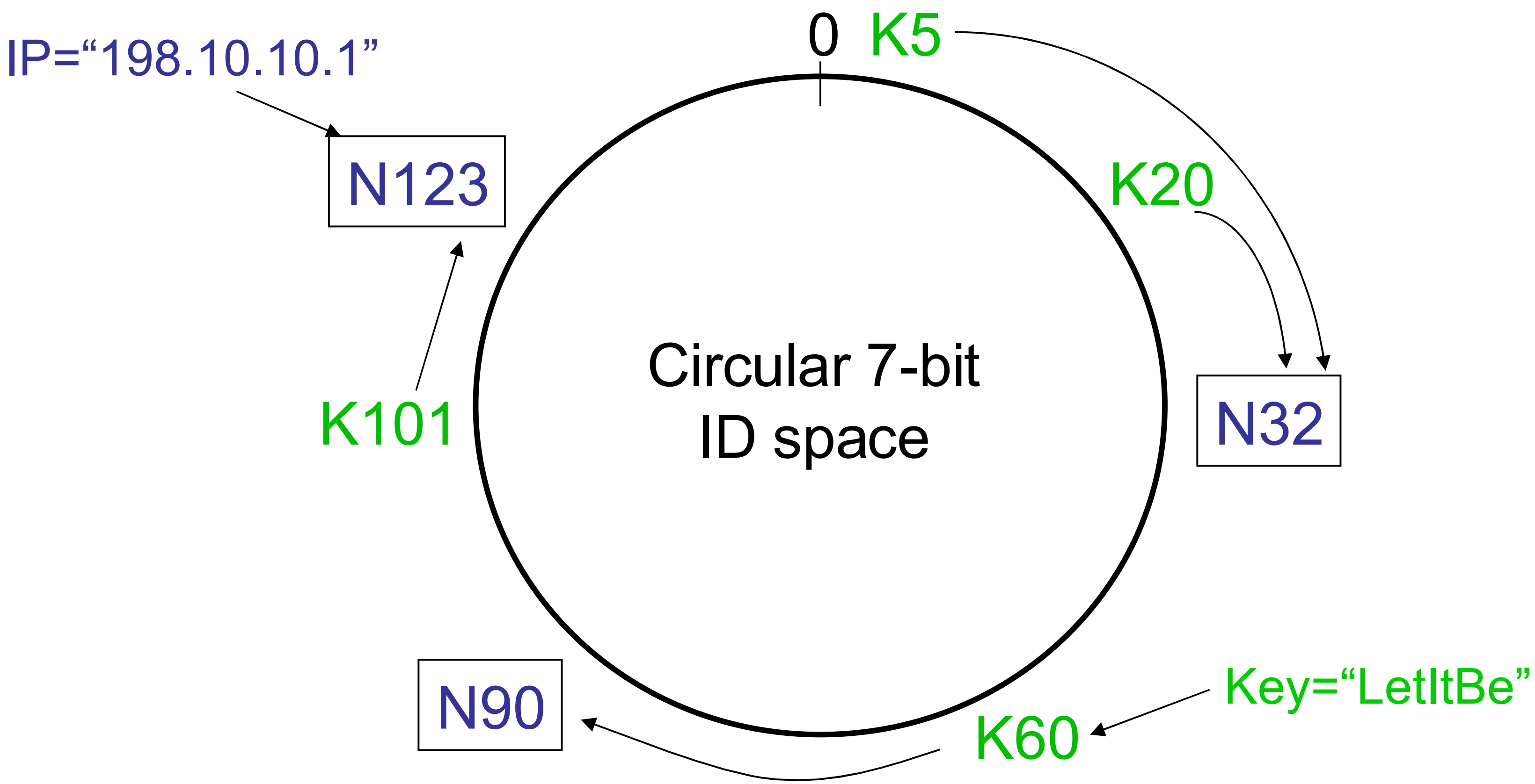
- **Node identifier:** SHA-1(IP address)

IP="198.10.10.1" —SHA-1→ ID=123

- Both are uniformly distributed
- How to map key IDs to node IDs?

Consistent Hashing in Chord

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

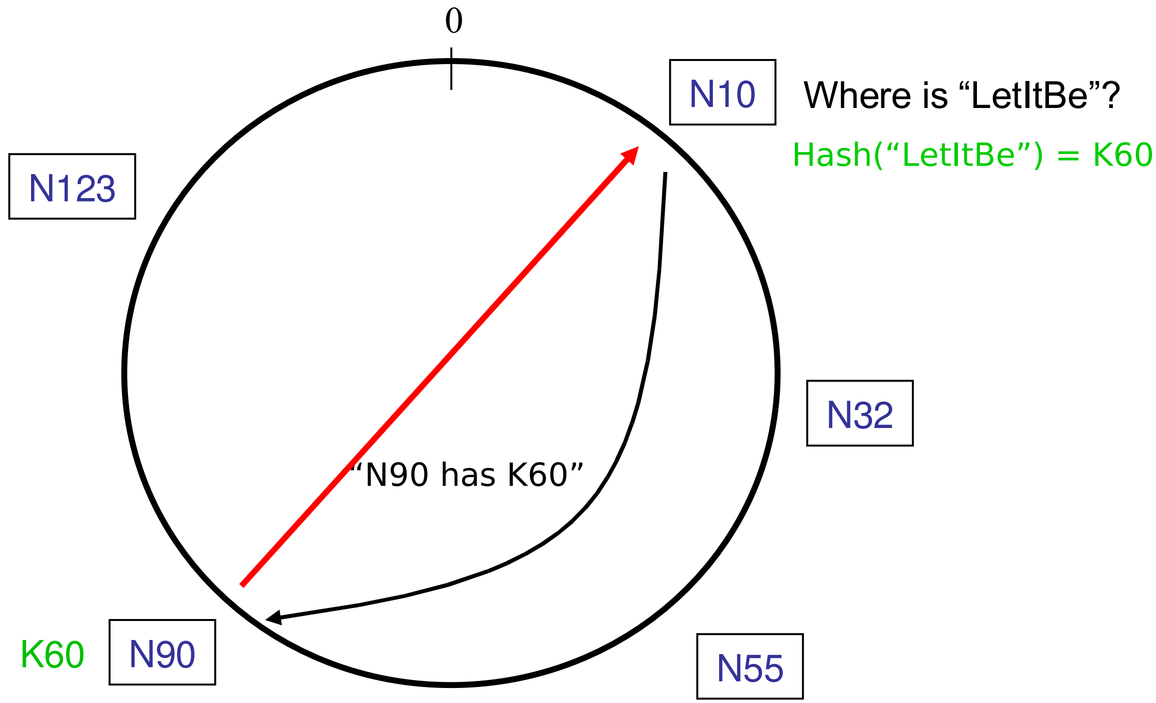


Consistent Hashing Properties

- **Load balance:** all nodes receive roughly the same number of keys
- **Flexibility:** when a node joins (or leaves) the network, only an fraction of the keys are moved to a different location.
  - This solution is **optimal** (*i.e.*, the minimum necessary to maintain a balanced load)

Consistent Hashing

- Every node knows of every other node
  - requires global information
- Routing tables are large: O(N)
- Lookups are fast: O(1)

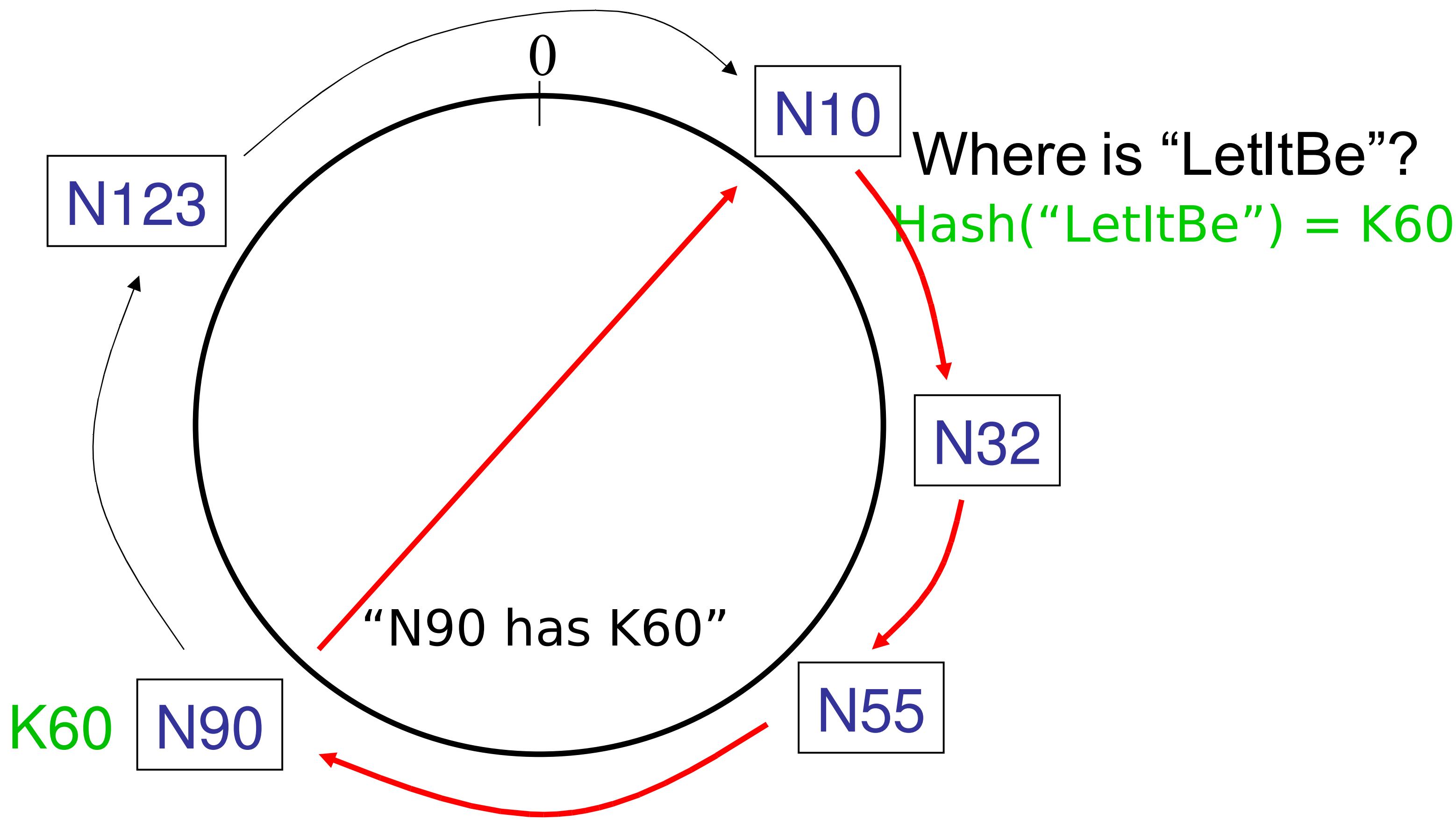


Load Balance Results (Theory)

- For *N* nodes and *K* keys, with high probability
  - each node holds at most $(1+\varepsilon)K/N$ keys
  - when node N+1 joins or leaves, $O(N/K)$ keys change hands, and only to/from node N+1

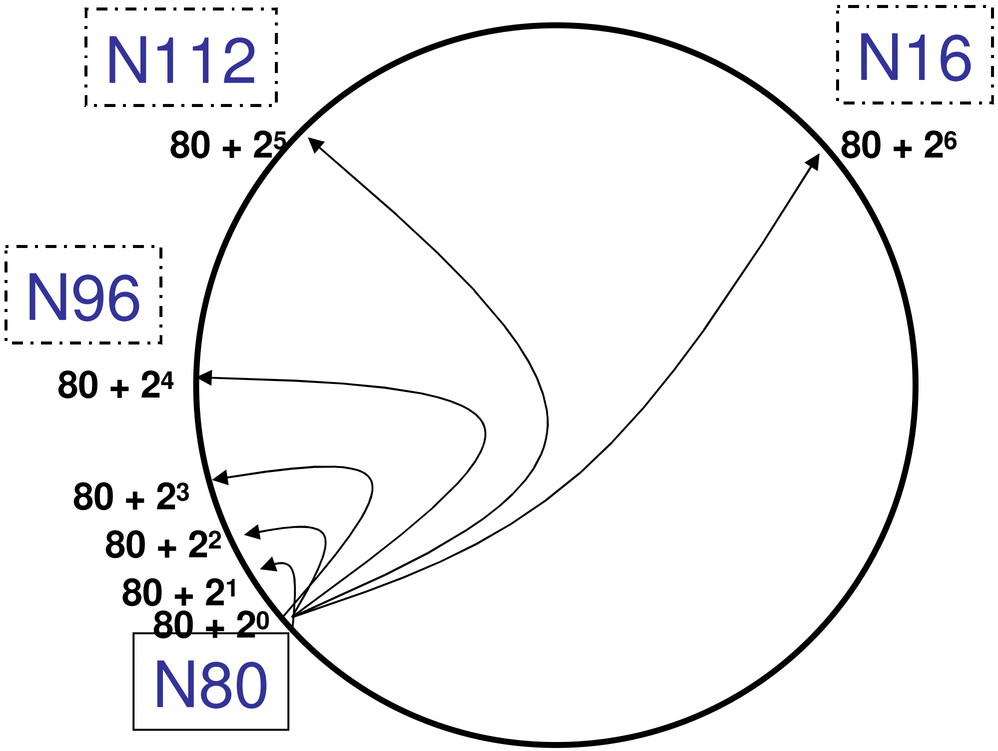
Lookups in Chord

- Every node knows its successor in the ring
- Requires *O(N)* lookups



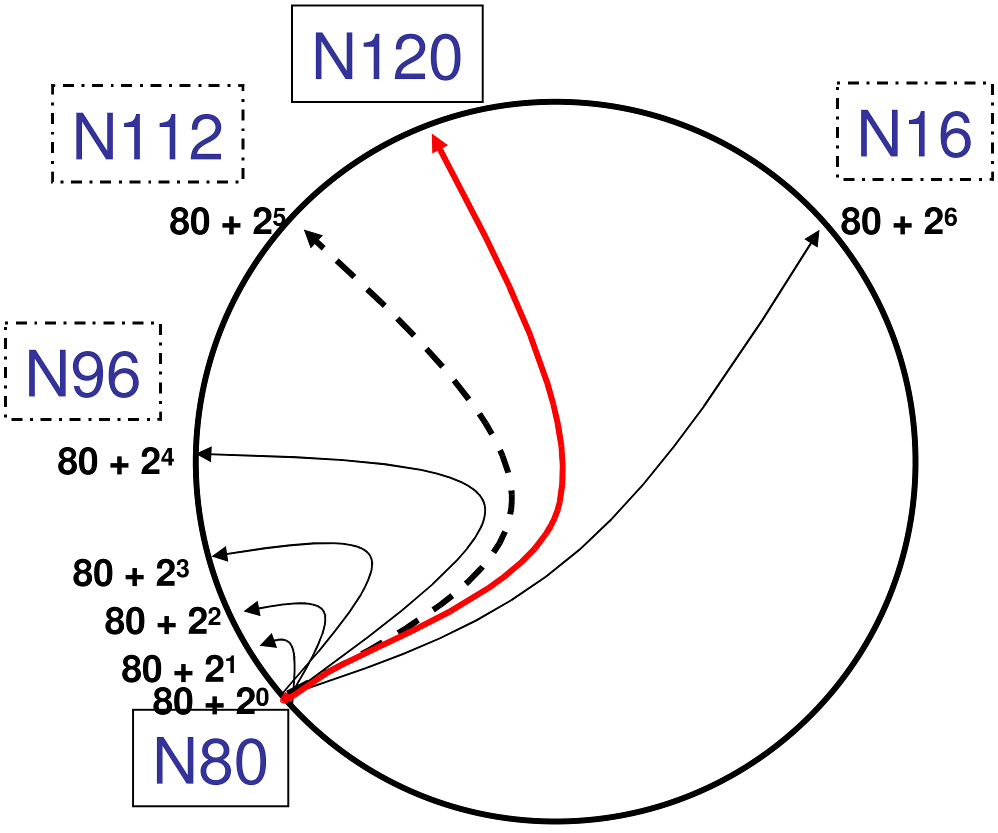
Reducing Lookups: Finger Tables

- Every node knows *m* other nodes in the ring
- Increase distance exponentially



Reducing Lookups: Finger Tables

- Finger *i* points to successor of $n+2^i$



Finger Table Lookups

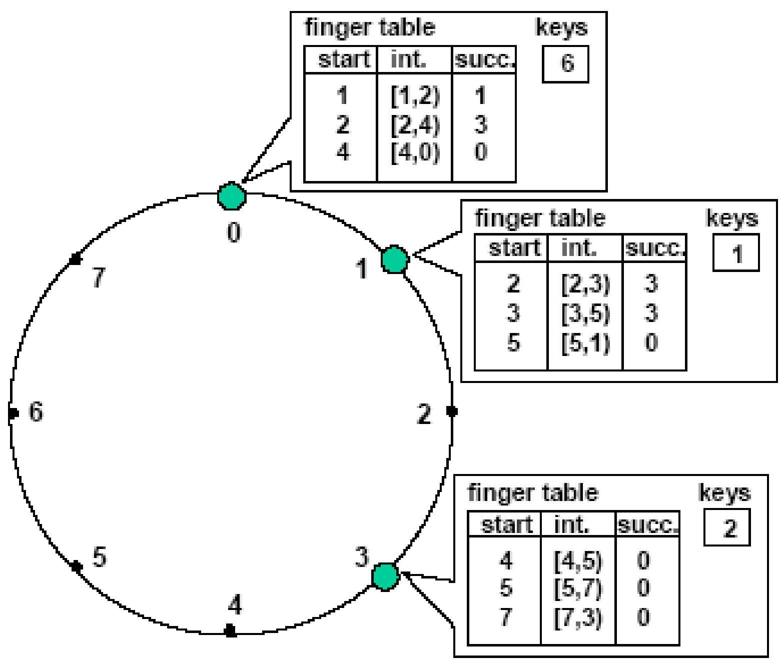
```
// ask node n to find id's successor
n.find_successor(id)
  n' = find_predecessor(id);
  return n'.successor;

// ask node n to find id's predecessor
n.find_predecessor(id)
  n' = n;
  while (id ∉ (n', n'.successor])
    n' = n'.closest_preceding_finger(id);
  return n';

// return closest finger preceding id
n.closest_preceding_finger(id)
  for i = m downto 1
    if (finger[i].node ∈ (n, id))
      return finger[i].node;
  return n;
```

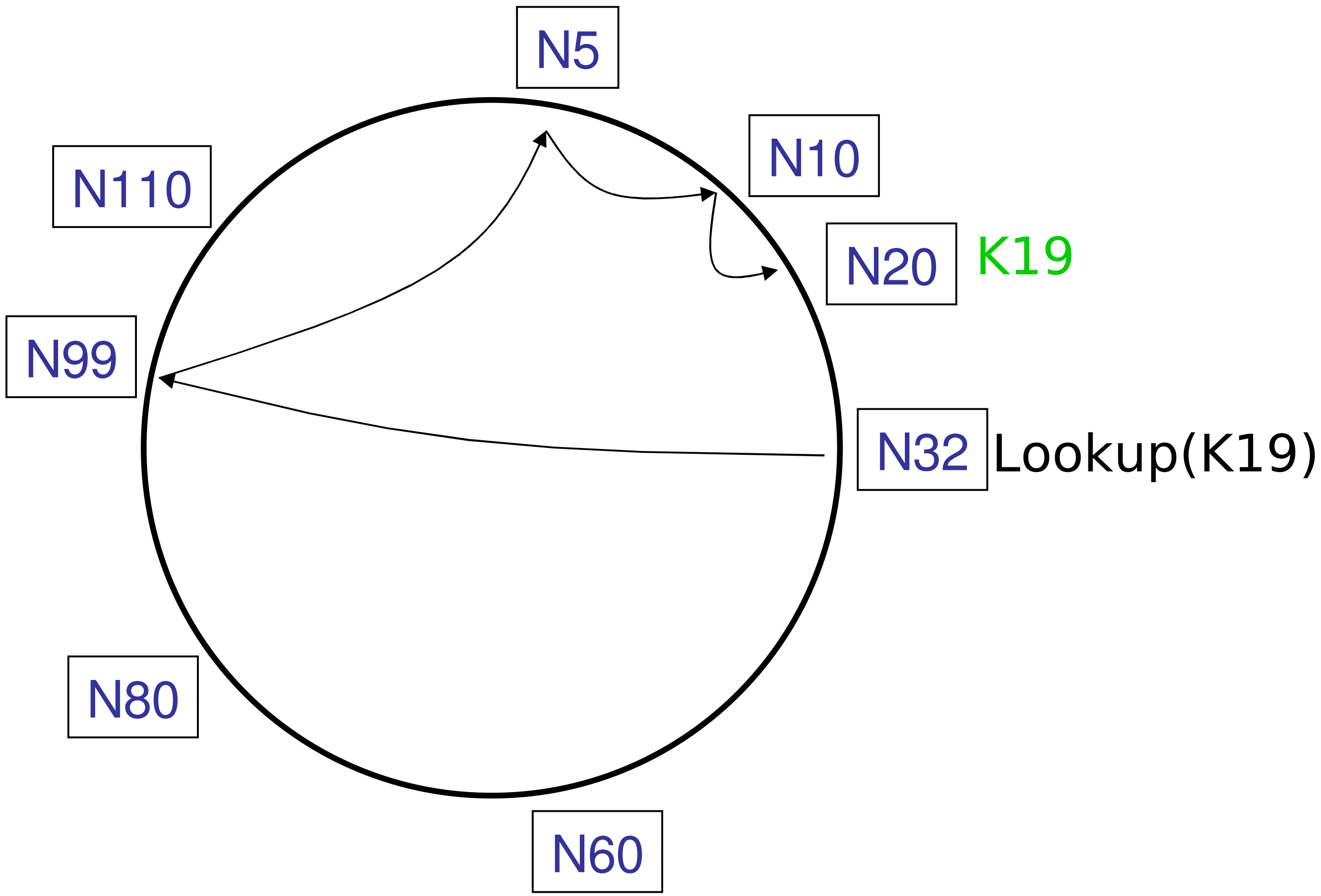
Each node knows its immediate successor. Find the predecessor of *id* and ask for its successor.

Move forward around the ring looking for node whose successor's ID is > *id*



Faster Lookups

- Lookups are *O(log N)* hops



Summary of Performance Results

- **Efficient:** *O(log N)* messages per lookup
- **Scalable:** *O(log N)* state per node
- **Robust:** survives massive membership changes

Possible Applications

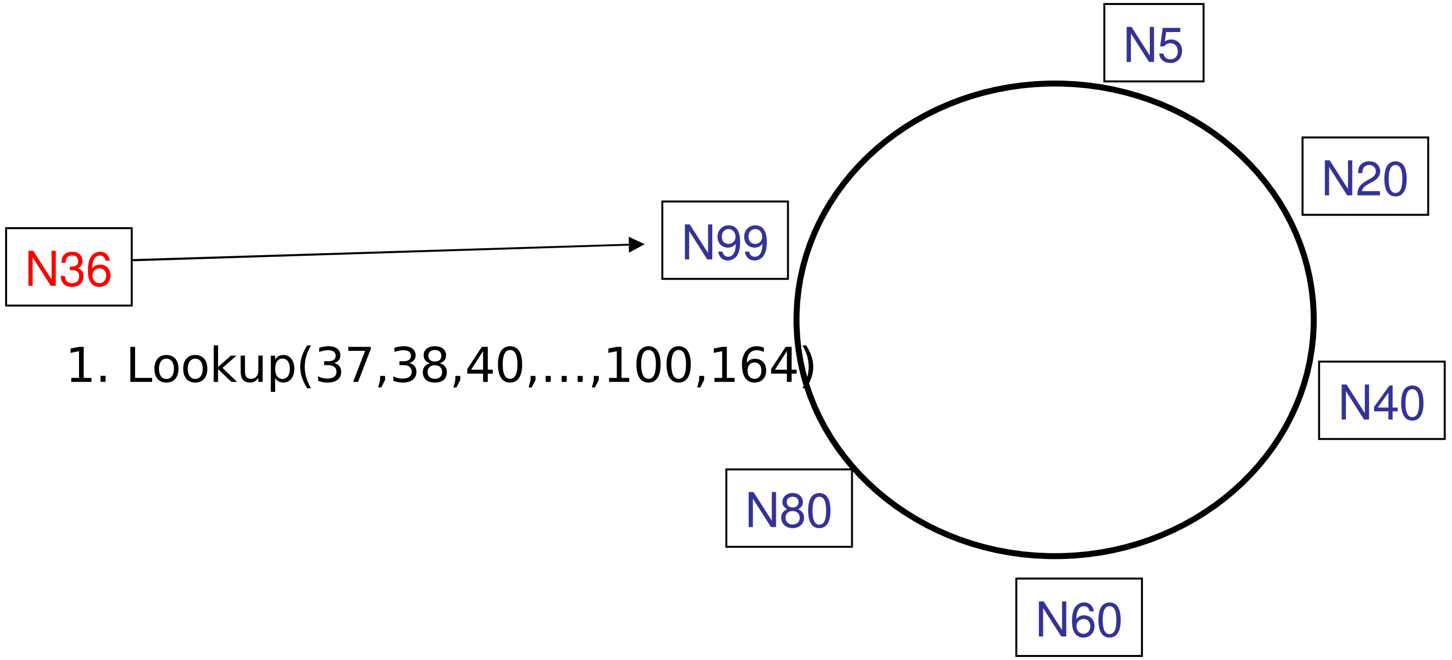
- Distributed indexes
- Cooperative storage
- Distributed, flat lookup services
- …

Joining the Chord Ring

- Nodes can join and leave at any time
  - Challenge: Maintining correct information about every key
- Three step process
  - Initialize all fingers of new node
  - Update fingers of existing nodes
  - Transfer keys from successor to new node
- **Two invariants**
  - Each node’s successor is maintained
  - *successor(k)* is responsible for *k*
  - (finger tables must also be correct for fast lookups)

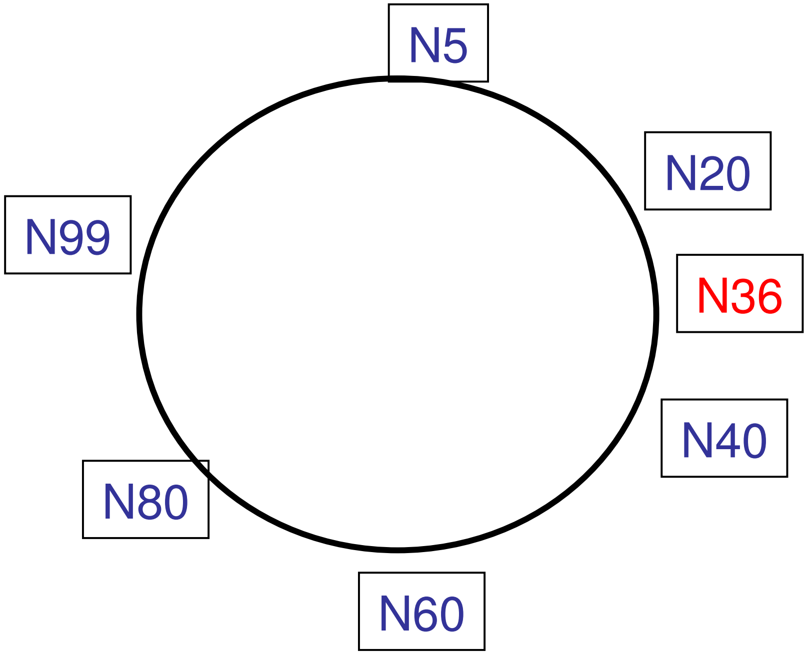
Join: Initialize New Node's Finger Table

- Locate **any** node *p* in the ring
- Ask node *p* to lookup fingers of new node



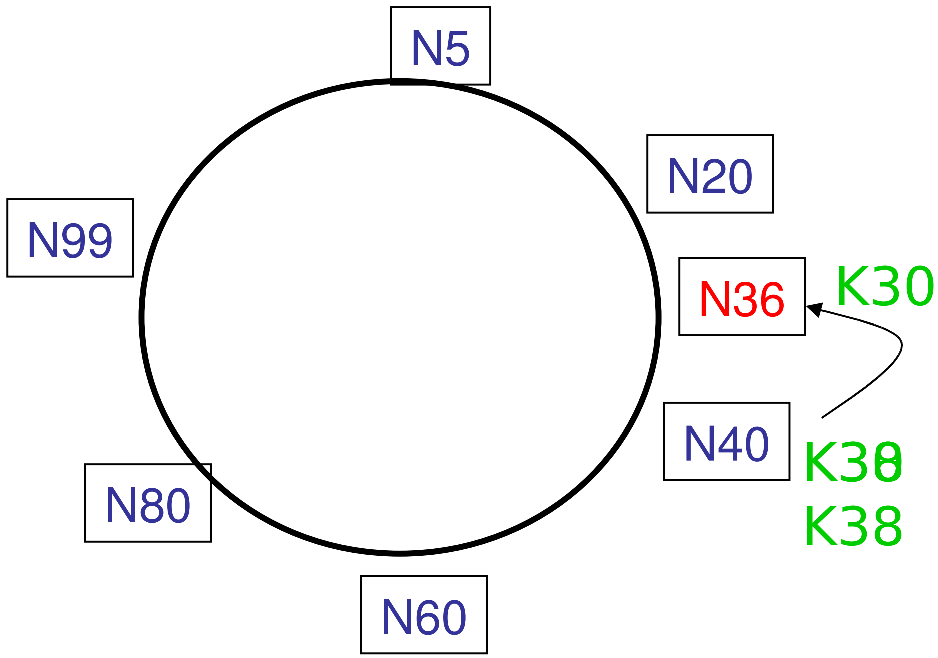
Join: Update Fingers of Existing Nodes

- New node calls update function on existing nodes
  - *N* becomes *ith* finger of *p* if (1) *p* precedes *n* by at least 2^*i-1* (2) *ith* finger of *p* succeeds *n*
- Existing nodes recursively update fingers of predecessors



Join: Transfer Keys

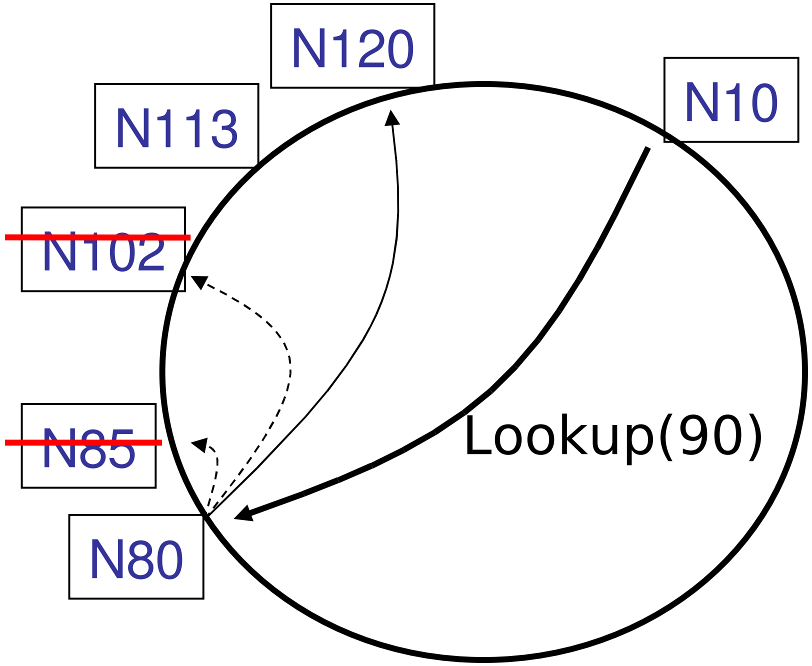
- Only keys in the range are transferred



Copy keys 21..36
from N40 to N36

Handling Failures

- **Problem:** Failures could cause incorrect lookup
- **Solution:** *Fallback:* keep track of successor fingers



Handling Failures

- Use successor list
  - Each node knows *r* immediate successors
  - After failure, will know first live successor
  - Correct successors guarantee correct lookups
- Guarantee is with some probability
  - Can choose *r* to make probability of lookup failure arbitrarily small

Chord: Questions

- Comparison to other DHTs
- Security concerns
- Workload imbalance
- Locality
- Search

Unstructured Overlays

BitTorrent

- Steps for publishing
  - Peer creates torrent: contains metadata about *tracker* and about the *pieces of the file* (checksum of each piece of the time).
  - Peers that create the initial copy of the file are called *seeders*
- Steps for downloading
  - Peer contacts tracker
  - Peer downloads from seeder, eventually from other peers
- Uses basic ideas from game theory to largely eliminate the free-rider problem
  - Previous systems could not deal with this problem

Basic Idea

- Chop file into many pieces
- Replicate *different* pieces on different peers as soon as possible
- As soon as a peer has a complete piece, it can trade it with other peers
- Hopefully, assemble the entire file at the end

Basic Components

- Seed
  - Peer that has the entire file
  - Typically fragmented into 256KB pieces
- Leecher
  - Peer that has an incomplete copy of the file
- Torrent file
  - Passive component
  - The torrent file lists SHA1 hashes of all the pieces to allow peers to verify integrity
  - Typically hosted on a web server
- Tracker
  - Allows peers to find each other
  - Returns a random list of peers

Pieces and Sub-Pieces

- A piece is broken into sub-pieces ... Typically from 64kB to 1MB
- Policy: Until a piece is assembled, only download sub-pieces for that piece
- This policy lets complete pieces assemble quickly

Classic Prisoner's Dilemma

Pareto Efficient Outcome

	Cooperate	Defect
Cooperate	3, 3	0, 5
Defect	5, 0	1, 1

Nash Equilibrium (and the *dominant strategy for both players)*

Repeated Games

- **Repeated game:** play single-shot game repeatedly
- **Subgame Perfect Equilibrium:** Analog to NE for repeated games
  - The strategy is an NE for *every* subgame of the repeated game
- **Problem:** a repeated game has many SPEs
- **Single Period Deviation Principle (SPDP)** can be used to test SPEs

Repeated Prisoner's Dilemma

- Example SPE: Tit-for-Tat (TFT) strategy
  - Each player mimics the strategy of the other player in the last round

	Cooperate	Defect
Cooperate	3, 3	0, 5
Defect	5, 0	1, 1

Question: Use the SPDP to argue that TFT is an SPE.

Tit-for-Tat in BitTorrent: Choking

- Choking is a temporary refusal to upload; downloading occurs as normal
  - If a node is unable to download from a peer, it does not upload to it
  - Ensures that nodes cooperate and eliminates the free-rider problem
  - Cooperation involves uploaded sub-pieces that you have to your peer
- Connection is kept open

Choking Algorithm

- Goal is to have several bidirectional connections running continuously
- Upload to peers who have uploaded to you recently
- Unutilized connections are uploaded to on a trial basis to see if better transfer rates could be found using them

Choking Specifics

- A peer always unchokes a fixed number of its peers (default of 4)
- Decision to choke/unchoke done based on current download rates, which is evaluated on a rolling 20-second average
- Evaluation on who to choke/unchoke is performed every 10 seconds
  - This prevents wastage of resources by rapidly choking/unchoking peers
  - Supposedly enough for TCP to ramp up transfers to their full capacity
- Which peer is the optimistic unchoke is rotated every 30 seconds

Rarest Piece First

- Policy: Determine the pieces that are most rare among your peers and download those first
- This ensures that the most common pieces are left till the end to download
- Rarest first also ensures that a large variety of pieces are downloaded from the seed (*Question:* Why is this important?)

Piece Selection

- The order in which pieces are selected by different peers is critical for good performance
- If a bad algorithm is used, we could end up in a situation where every peer has all the pieces that are currently available and none of the missing ones
- If the original seed is taken down, the file cannot be completely downloaded!

Random First Piece

- Initially, a peer has nothing to trade
- Important to get a complete piece ASAP
- Rare pieces are typically available at fewer peers, so downloading a rare piece initially is not a good idea
- Policy: Select a random piece of the file and download it

Endgame Mode

- When all the sub-pieces that a peer doesn't have are actively being requested, these are requested from every peer
- Redundant requests cancelled when piece arrives
- Ensures that a single peer with a slow transfer rate doesn't prevent the download from completing

Questions

- Peers going offline when download completes
- Integrity of downloads

Distributing Content: Coding

Digital Fountains

- **Analogy:** water fountain
  - Doesn't matter which bits of water you get
  - Hold the glass out until it is full

- **Ideal:** Infinite stream
- **Practice:** Approximate, using erasure codes
  - Reed-solomon
  - Tornado codes (faster, slightly less efficient)

Applications

- Reliable multicast
- Parallel downloads
- Long-distance transmission (avoiding TCP)
- One-to-many TCP
- Content distribution on overlay networks
- Streaming video

Point-to-Point Data Transmission

- TCP has problems over long-distance connections.
  - Packets must be acknowledged to increase sending window (packets in flight).
  - Long round-trip time leads to slow acks, bounding transmission window.
  - Any loss increases the problem.

- Using digital fountain + TCP-friendly congestion control can greatly speed up connections.

- Separates the “what you send” from “how much” you send.
  - Do not need to buffer for retransmission.

Other Applications

- Other possible applications outside of networking
  - Storage systems
  - Digital fountain codes for errors
  - ??