Content Overlays

Nick Feamster
CS 7260
March 12, 2007

Content Overlays

- Distributed content storage and retrieval
- Two primary approaches:
  - Structured overlay
  - Unstructured overlay

- Today's paper: *Chord*
  - Not strictly a content overlay, but one can build content overlays on top of it (e.g., Dabek *et al.* "CFS")

Goals and Examples

- Goals
  - File distribution/exchange
  - Anonymous storage and communication

- Examples
  - **Directory-based:** Napster
  - **Unstructured overlays:** Freenet and Gnutella
  - **Structured overlays:** Chord, CAN, Pastry, etc.
  - Content-distribution: Akamai
  - Bittorrent (overview and economics)

Directory-based Search, P2P Fetch

- Centralized Database
  - **Join**: on startup, client contacts central server
  - **Publish**: reports list of files to central server
  - **Search**: query the server

- Peer-to-Peer File Transfer
  - **Fetch**: get the file directly from peer

History: Freenet (circa 1999)

- Unstructured overlay (compare to Gnutella)
  - No hierarchy; implemented on top of existing networks (*e.g.,* IP)

- First example of key-based routing
  - Freenet's legacy
  - Unlike Chord, no provable performance guarantees

- Goals
  - **Censorship-resistance**
  - **Anonymity:** for producers and consumers of data
    - Nodes don't even know what they are storing
  - **Survivability:** no central servers, etc.
  - **Scalability**

- **Current status:** redesign

Big Idea: Keys as First-Class Objects

Keys name both the objects being looked up and the content itself

- **Keyword-signed Key (KSK)**
  - Key is based on human-readable description of the file
  - **Problem:** flat, global namespace (possible collisions)

- **Signed Subspace Key**
  - Helps prevent namespace collisions
  - Allows for secure update
  - User can only retrieve and decrypt a document if it knows the SSK

- **Content Hash Key**
  - SHA-1 hash of the file that is being stored
  - Allows for efficient file updates through indirection

Publishing and Querying in Freenet

- Process for both operations is the same
- Keys passed through a chain of proxy requests
  - Nodes make local decisions about routing queries
  - Queries have **hops-to-live** and a unique ID
- Two cases
  - **Node has local copy of file**
    - File returned along reverse path
    - Nodes along reverse path cache file
  - **Node does not have local copy**
    - Forward request to neighbor whose key is closest to the key of the file

Routing Queries in Freenet

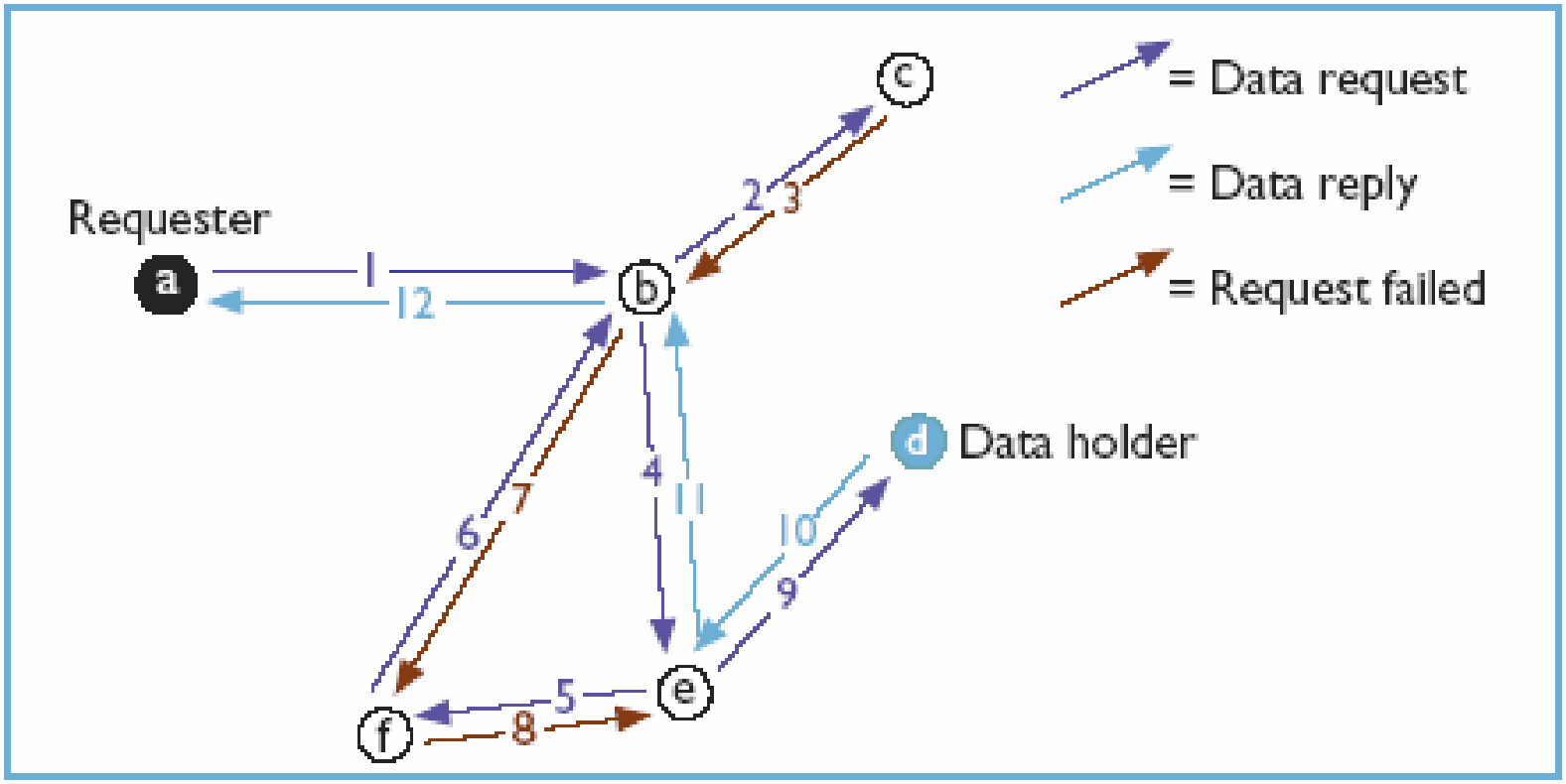


Figure 1. Typical request sequence. The request moves through the network from node to node, backing out of a dead-end (step 3) and a loop (step 7) before locating the desired file.

Small World Network Property

- The majority of the nodes have a few local connections to other nodes
- Few nodes have large wide ranging connections
- Resulting properties
  - Fault tolerance
  - Short average path length

Freenet Design

- Strengths
  - Decentralized
  - Anonymous
  - Scalable

- Weaknesses
  - **Problem:** how to find the names of keys in the first place?
  - No file lifetime guarantees
  - No efficient keyword search
  - No defense against DoS attacks
  - Bandwidth limitations not considered

Freenet Security Mechanisms

- Encryption of messages
  - Prevents eavesdropping
- Hops-to-live
  - prevents determining originator of query
- Hashing
  - checks data integrity
  - prevents intentional data corruption

Structured [Content] Overlays

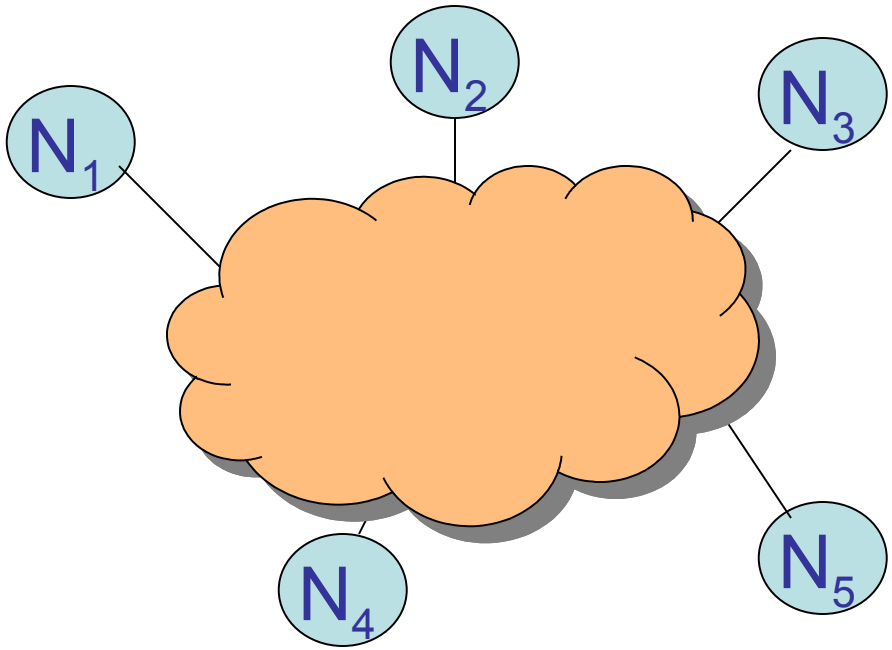
Chord: Overview

- 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



Publisher
Key="LetItBe"
Value=MP3 data

Client
Lookup("LetItBe")

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

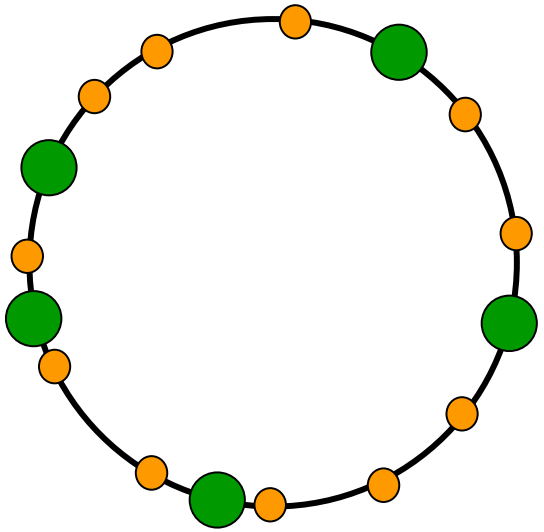
- **Problems with uniform hashing**
  - How to perform consistent hashing in a distributed fashion?
  - What happens when nodes join and leave?

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

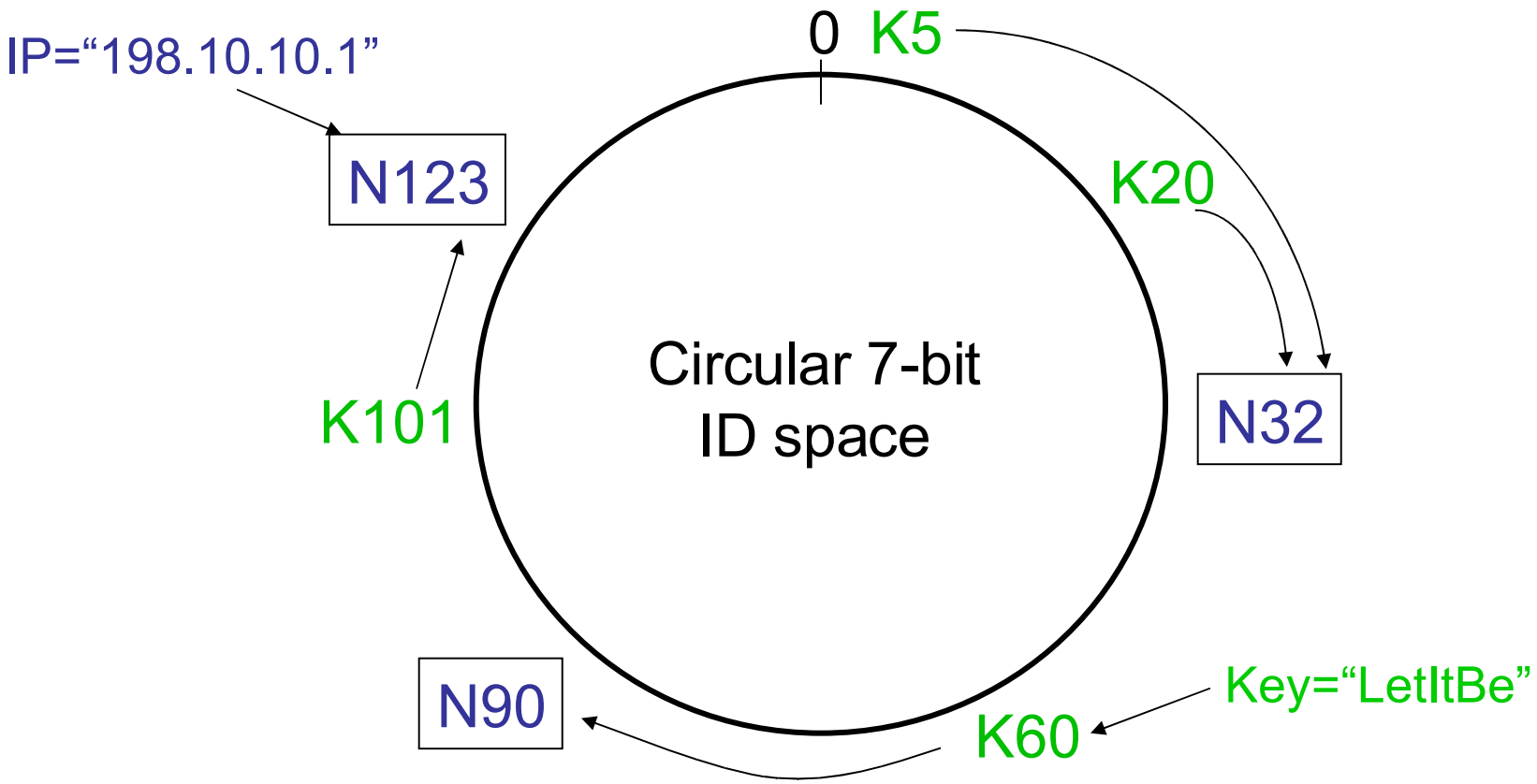
- **Node identifier:** SHA-1(IP address)

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



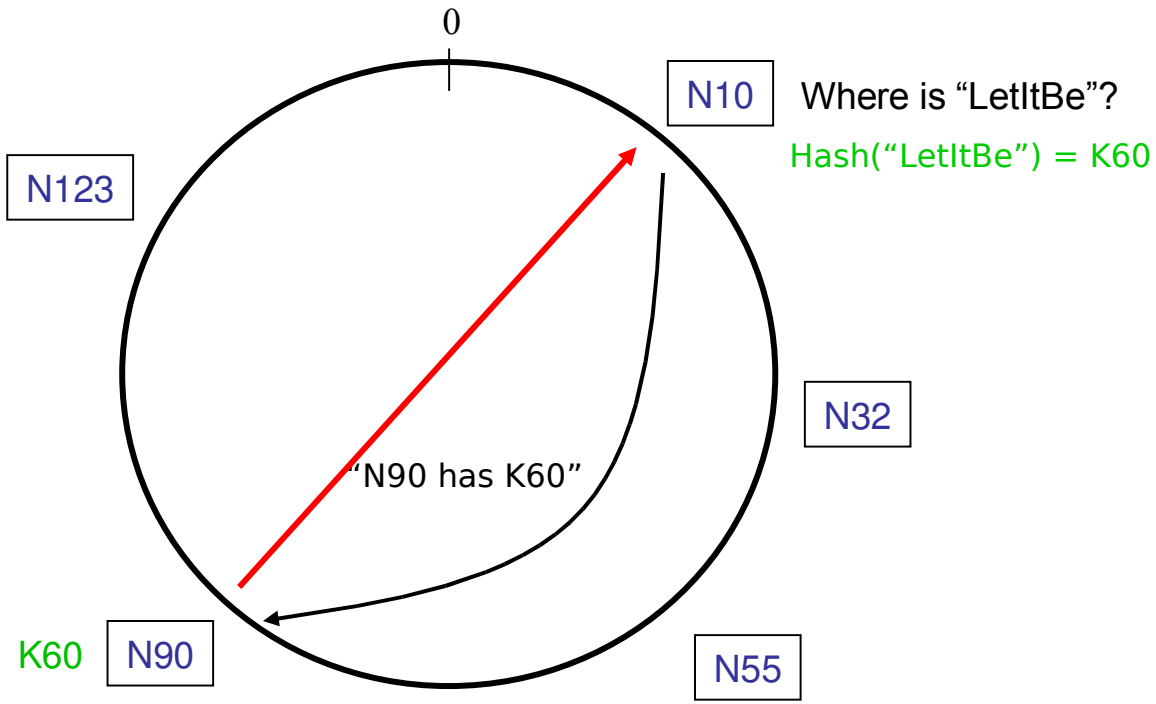
Consistent Hashing Properties

- **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)

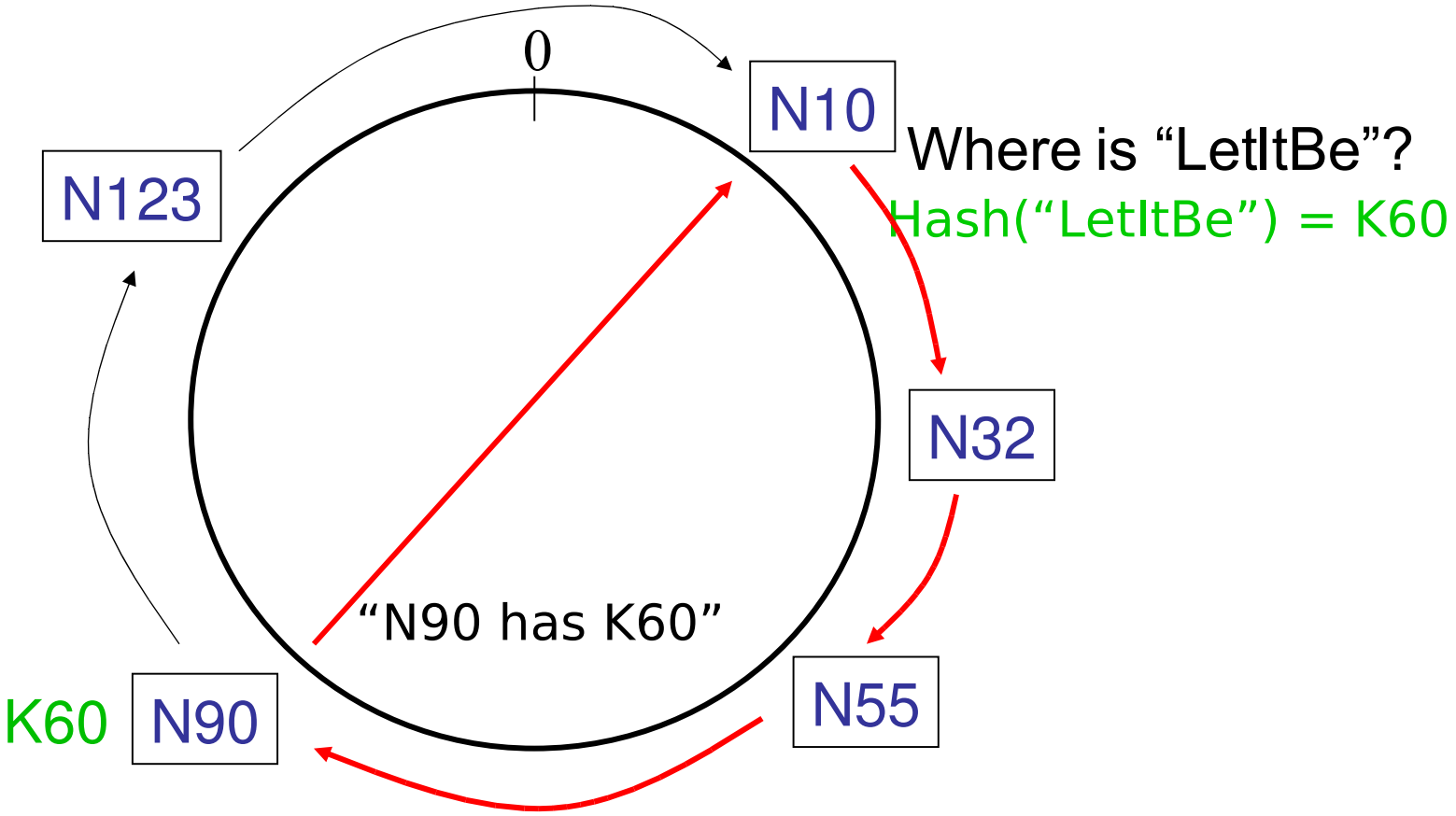


Load Balance Results (Theory)

- 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

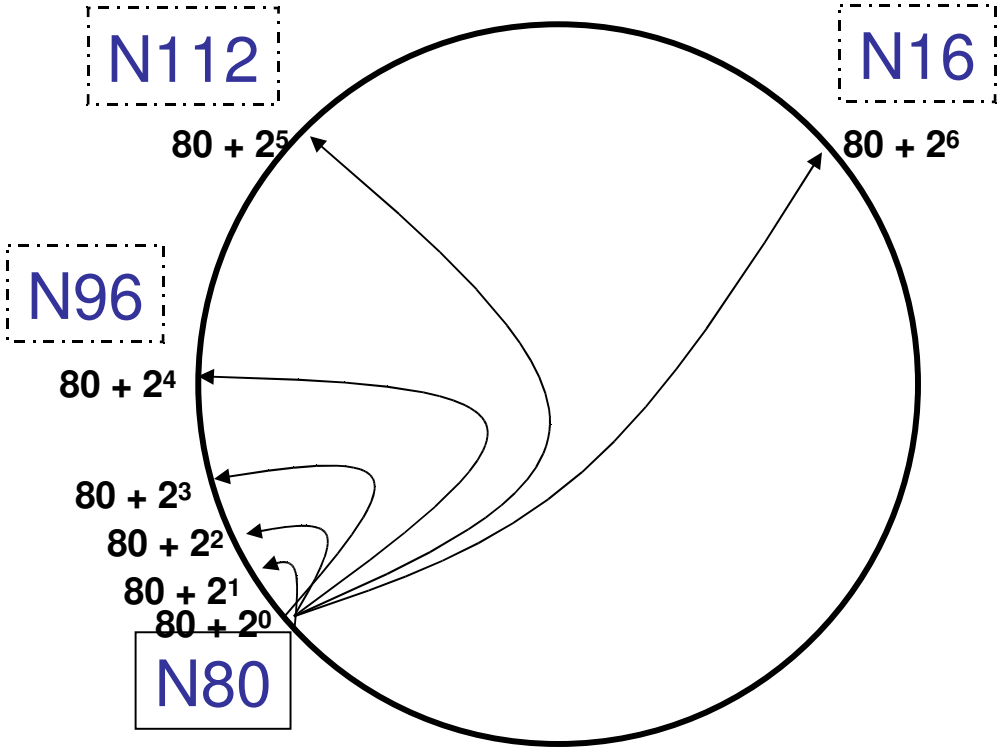
Lookups in Chord

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



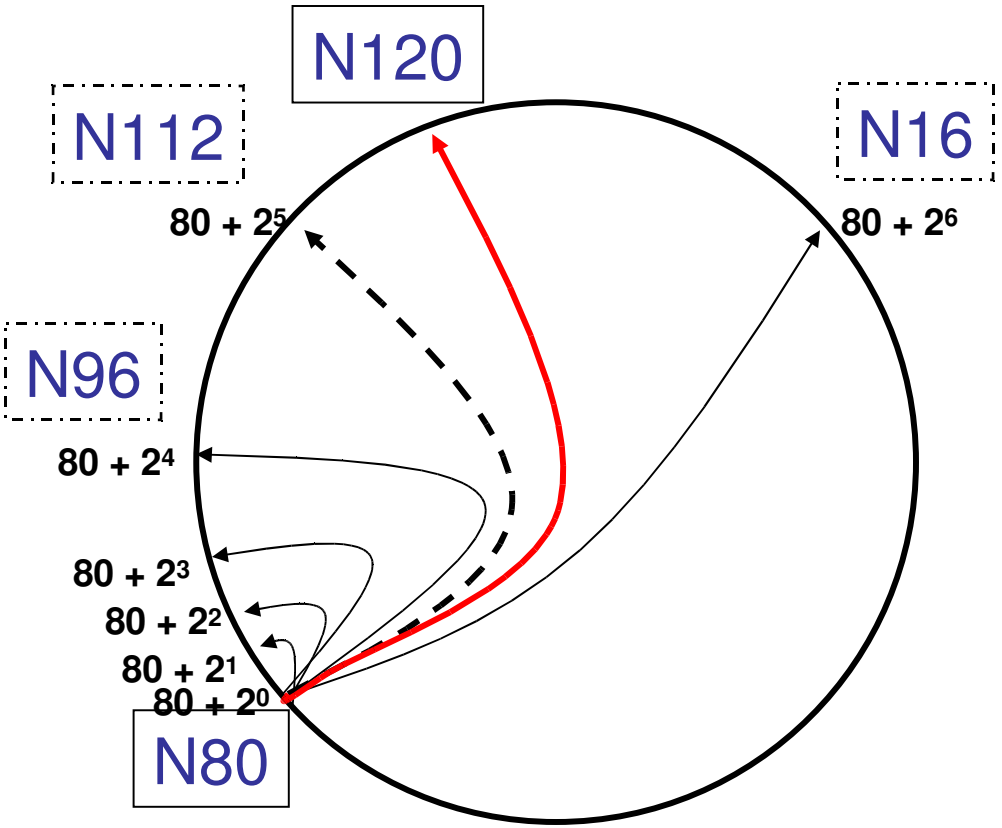
Reducing Lookups: Finger Tables

- 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

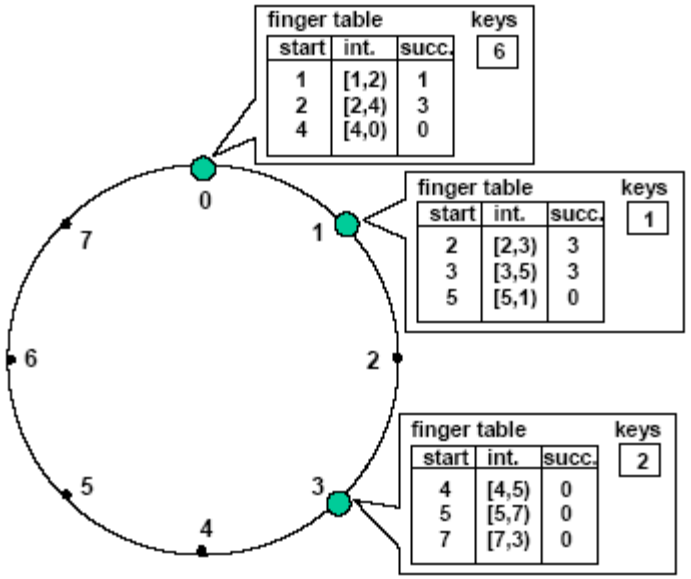
```
// ask node n to find id's successor
n.find_successor(id)
  n' = find_predecessor(id);
  return n'.successor;

// 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;
```

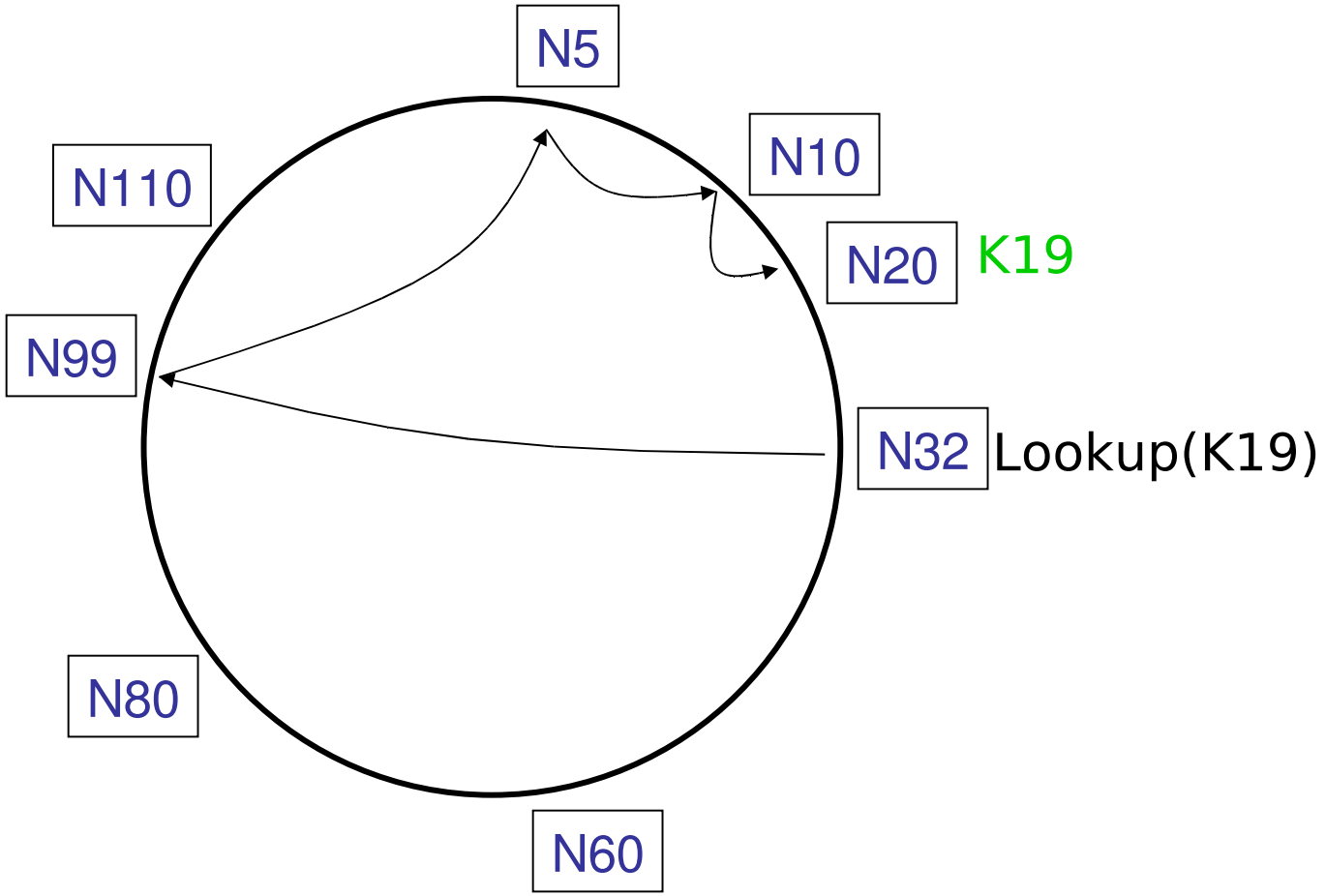
Each node knows its immediate successor. Find the predecessor of *id* and ask for its successor.

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

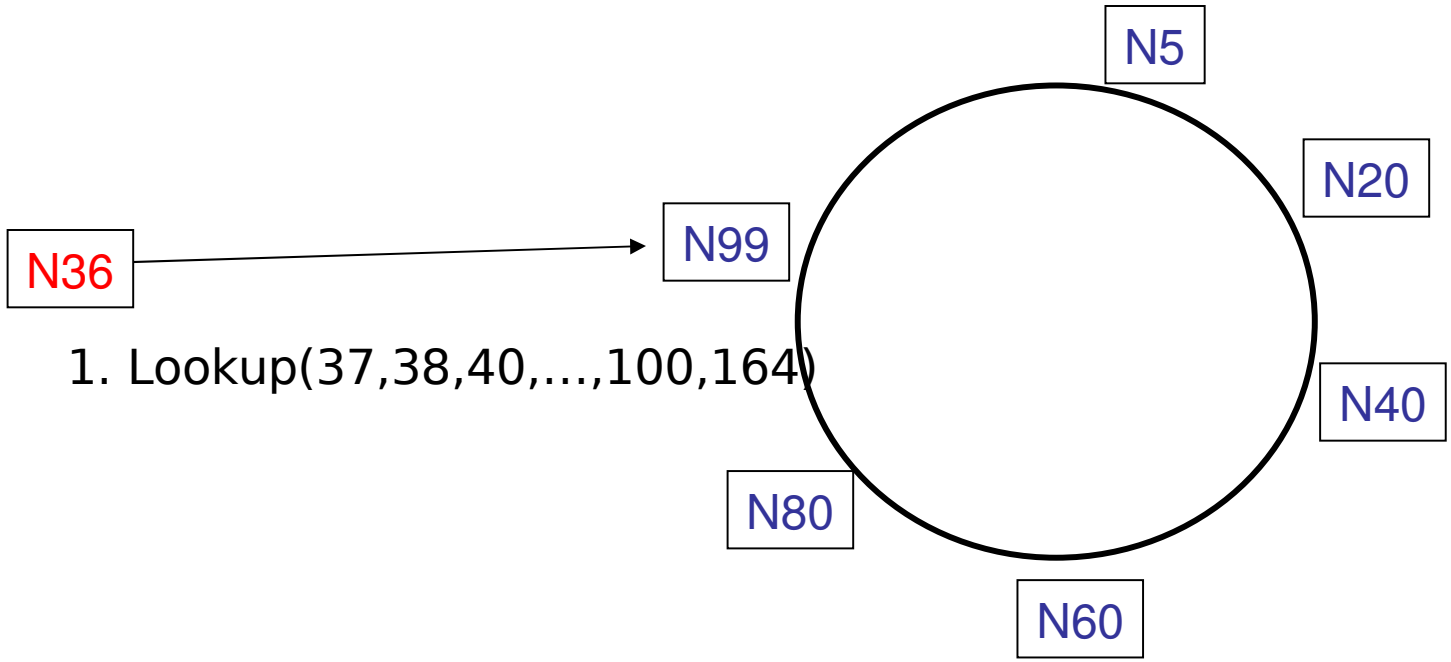
Joining the Ring

- Three step process
  - Initialize all fingers of new node
  - Update fingers of existing nodes
  - Transfer keys from successor to new node

- Two invariants to maintain
  - Each node's successor is maintained
  - *successor(k)* is responsible for *k*

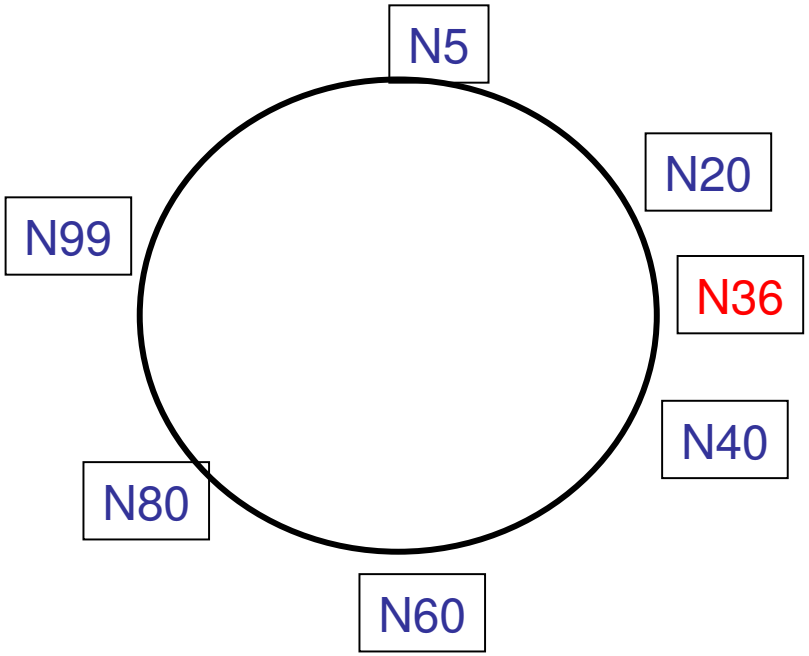
Join: Initialize New Node's Finger Table

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



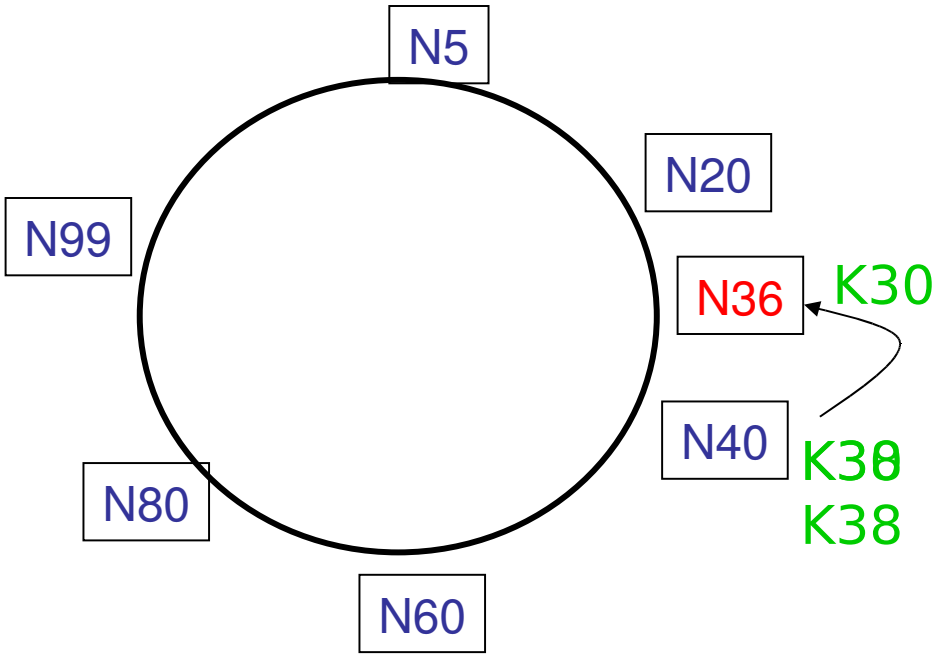
Join: Update Fingers of Existing Nodes

- New node calls update function on existing nodes
- Existing nodes recursively update fingers of other nodes



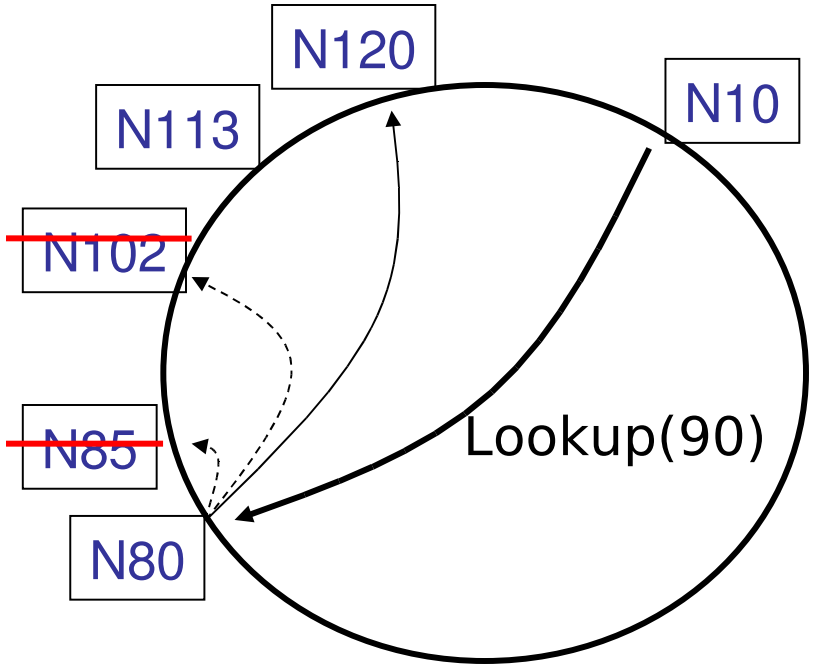
Join: Transfer Keys

- Only keys in the range are transferred



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

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*