Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications

Ion Stoica, Robert Morris, David Karger,
M. Frans Kaashoek, Hari Balakrishnan

MIT and Berkeley

presented by Daniel Figueiredo

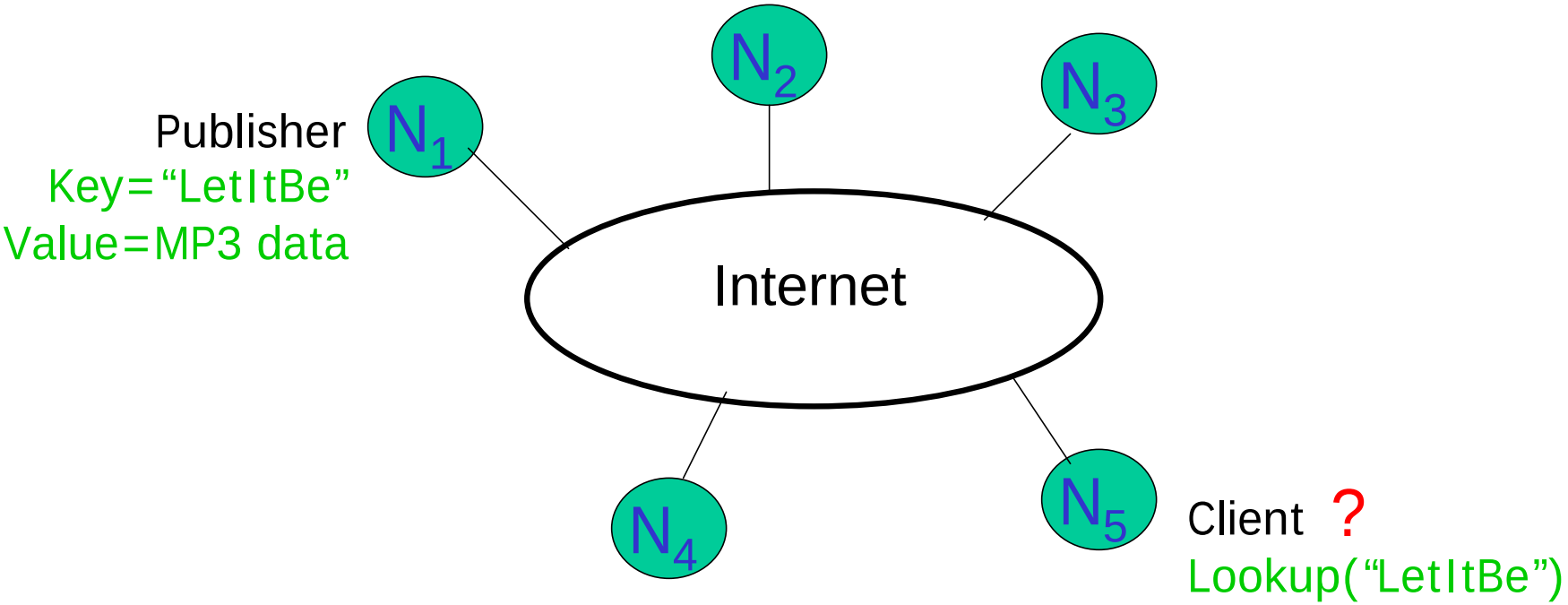
- presentation based on slides by Robert Morris (SIGCOMM'01)

Outline

- Motivation and background
- Consistency caching
- Chord
- Performance evaluation
- Conclusion and discussion

Motivation

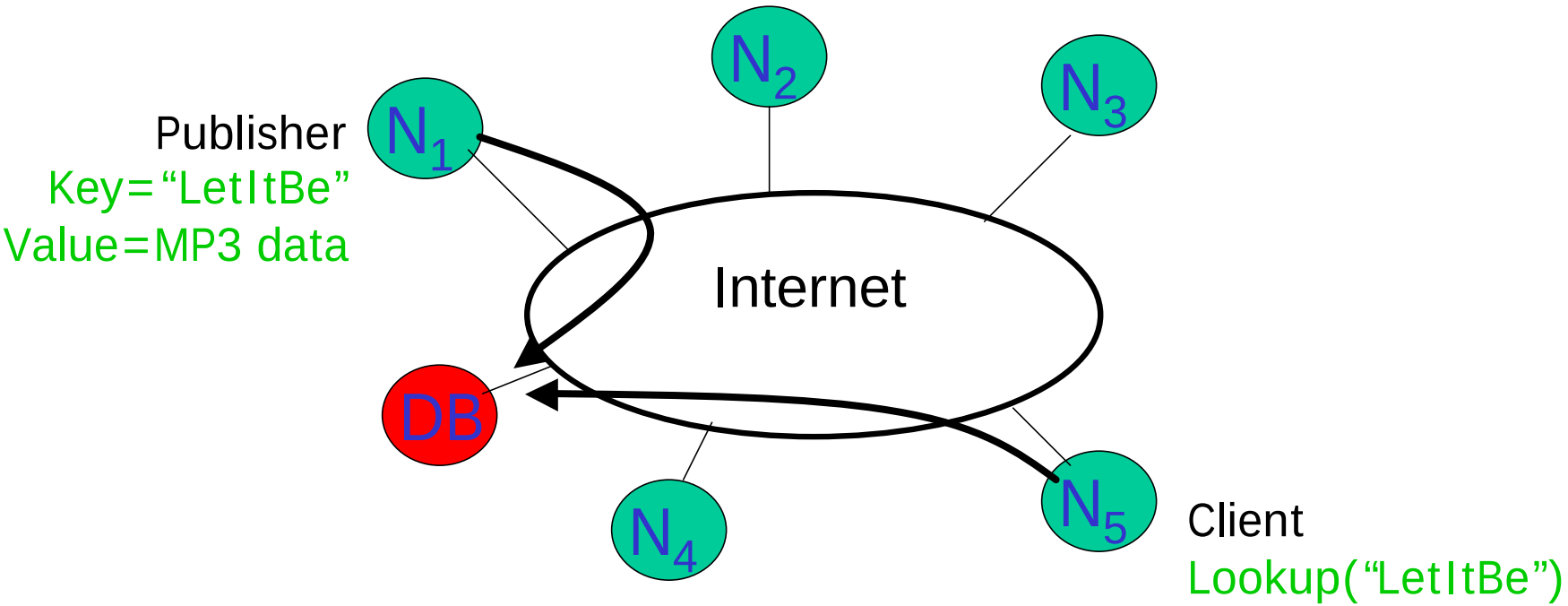
How to find data in a distributed file sharing system?



- Lookup is the key problem

Centralized Solution

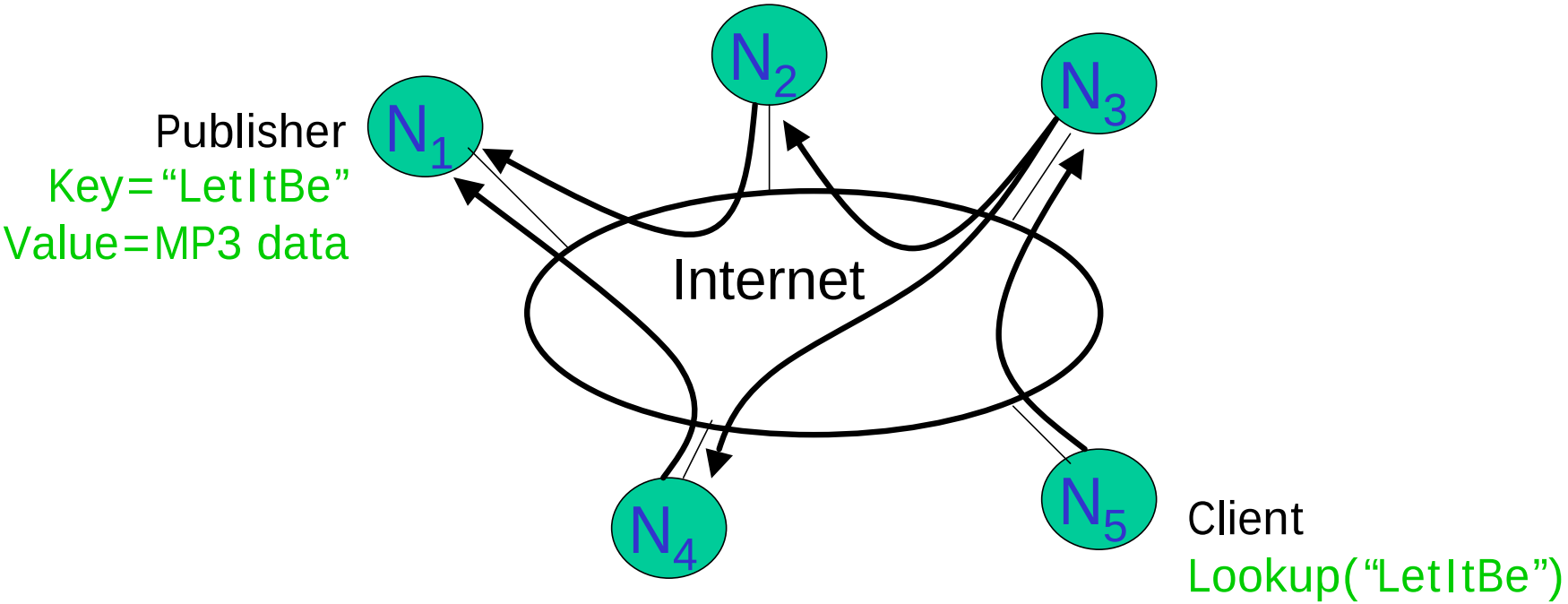
- Central server (Napster)



- Requires O(M) state
- Single point of failure

Distributed Solution (1)

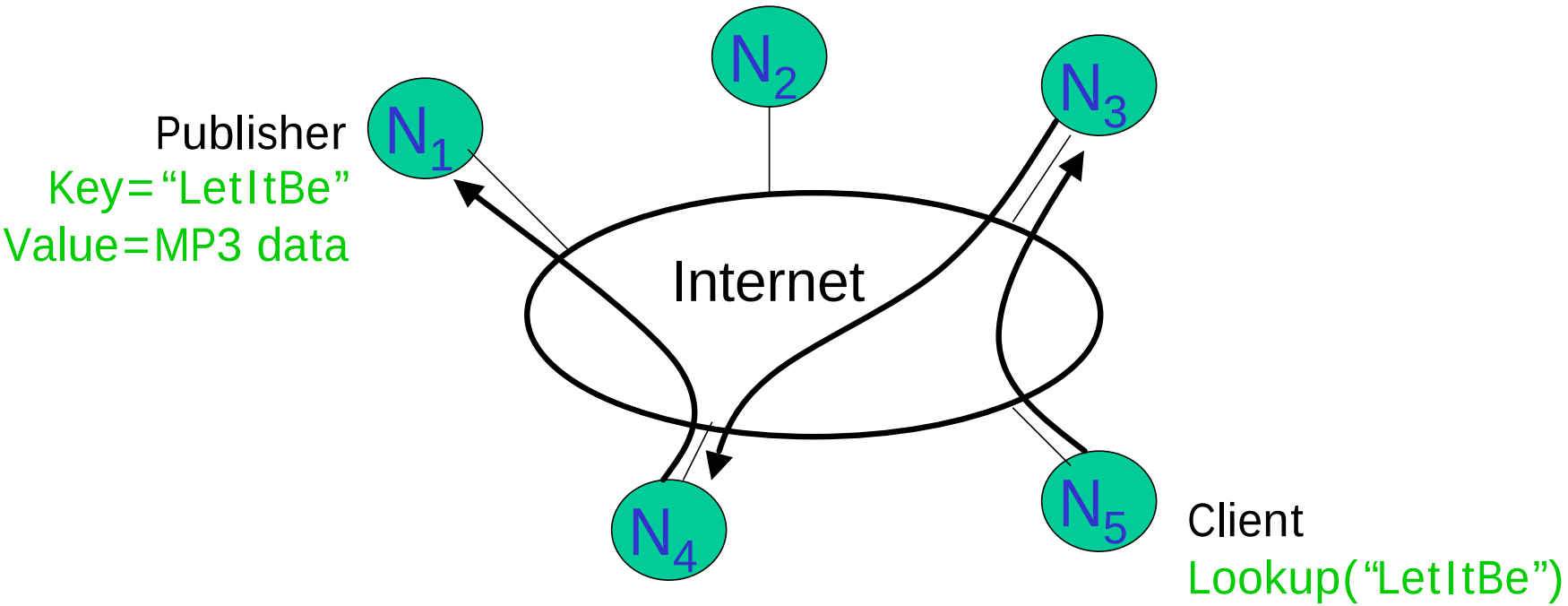
- Flooding (Gnutella, Morpheus, etc.)



- Worst case O(N) messages per lookup

Distributed Solution (2)

- Routed messages (Freenet, Tapestry, Chord, CAN, etc.)



- Only exact matches

Routing Challenges

- Define a useful key nearness metric
- Keep the hop count small
- Keep the routing tables “right size”
- Stay robust despite rapid changes in membership

Authors claim:

- Chord: emphasizes efficiency and simplicity

Chord Overview

- Provides peer-to-peer hash lookup service:
  - Lookup(key) ® IP address
  - Chord does not store the data
- How does Chord locate a node?
- How does Chord maintain routing tables?
- How does Chord cope with changes in membership?

Chord properties

- Efficient: O(Log N) messages per lookup
  - N is the total number of servers
- Scalable: O(Log N) state per node
- Robust: survives massive changes in membership

- Proofs are in paper / tech report
  - Assuming no malicious participants

Chord IDs

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

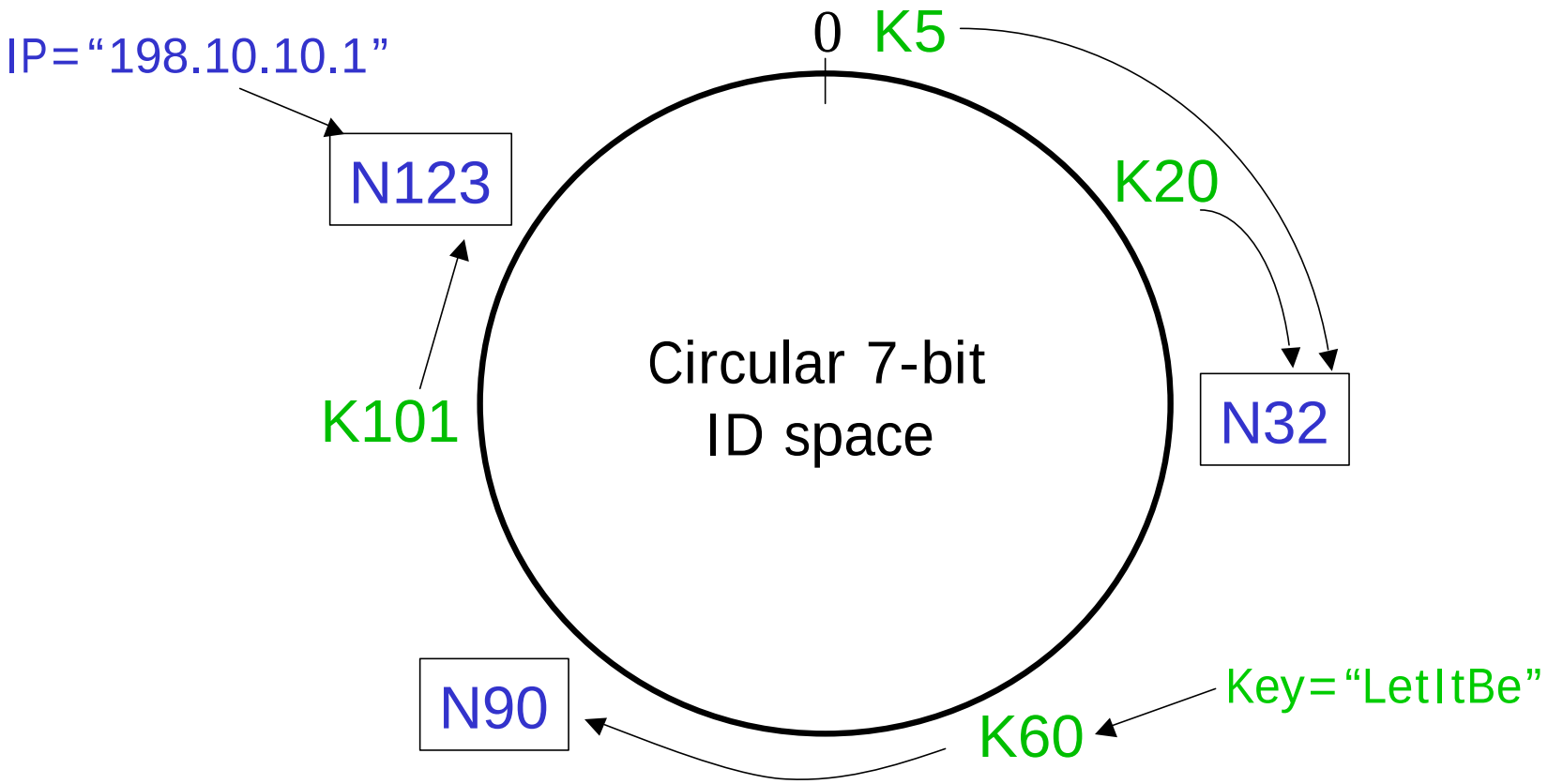
  Key="LetItBe" —SHA-1→ ID=60

- 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?

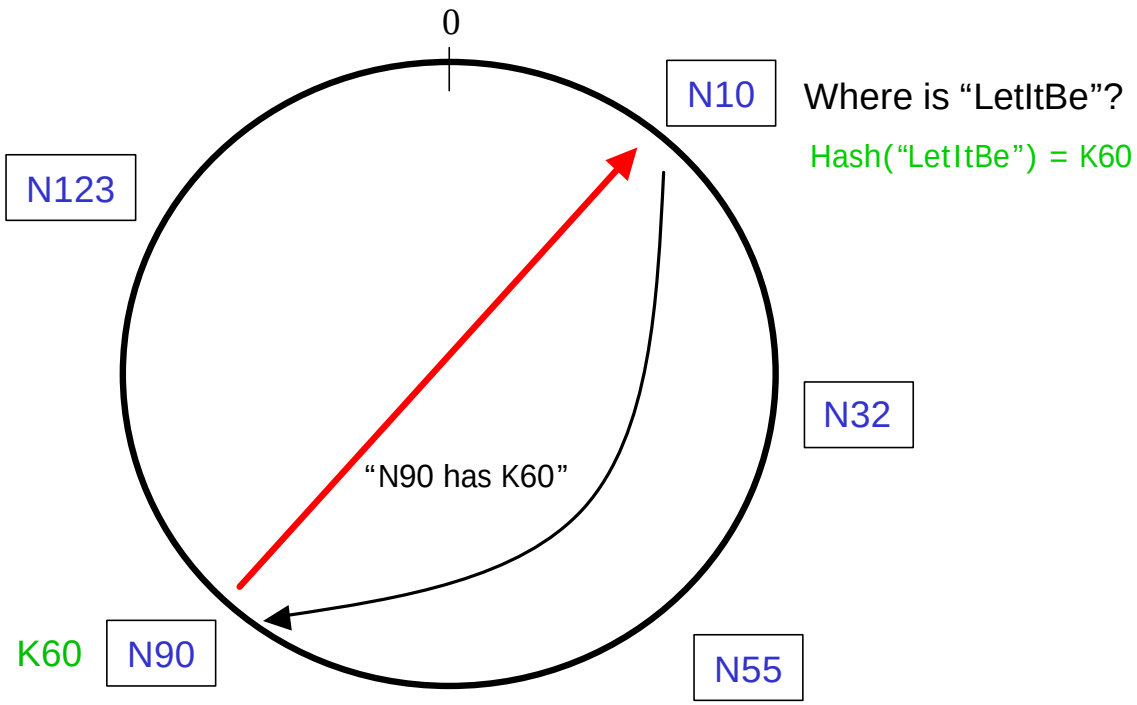
Consistent Hashing [Karger 97]



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

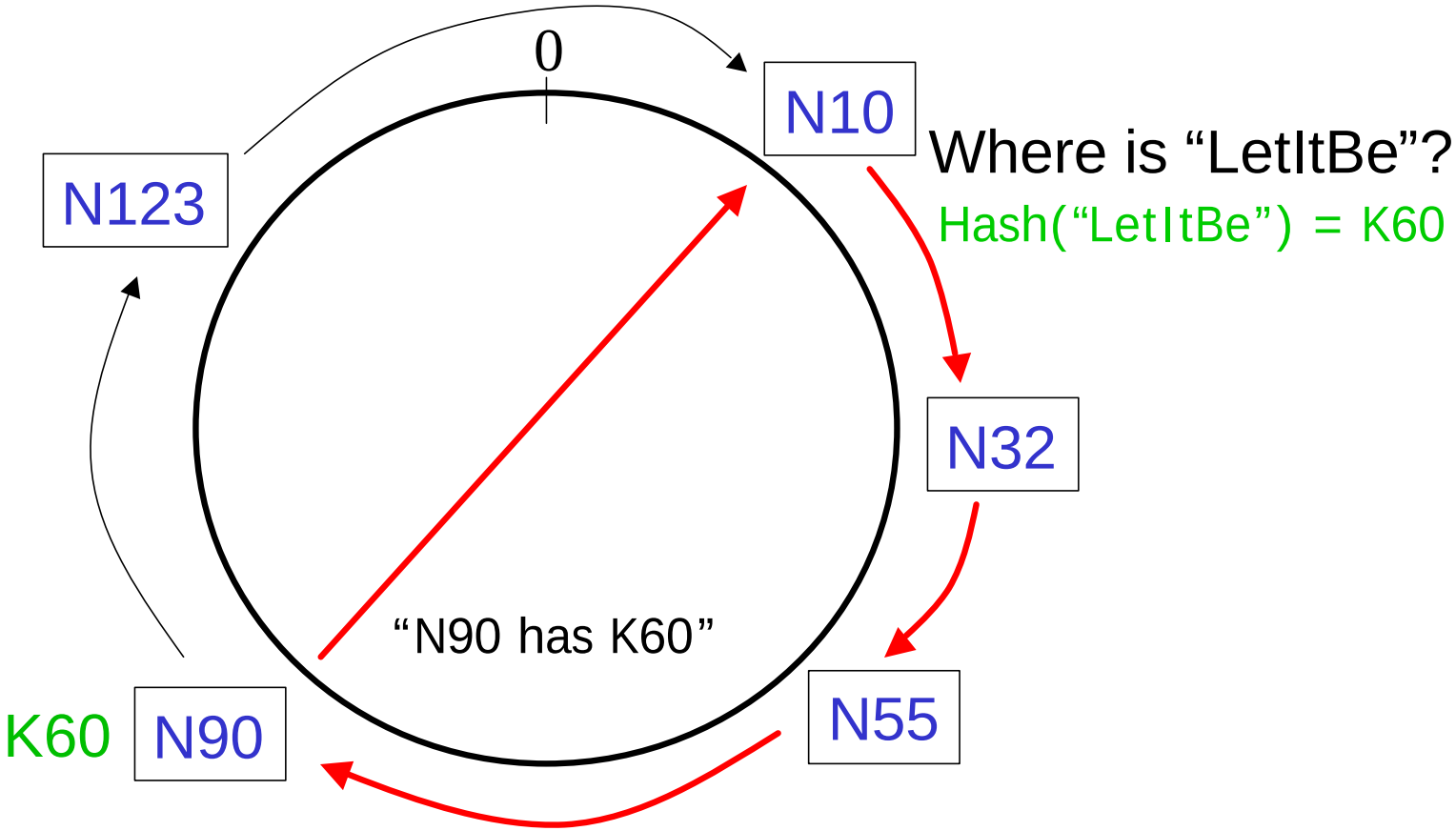
Consistent Hashing

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



Chord: Basic Lookup

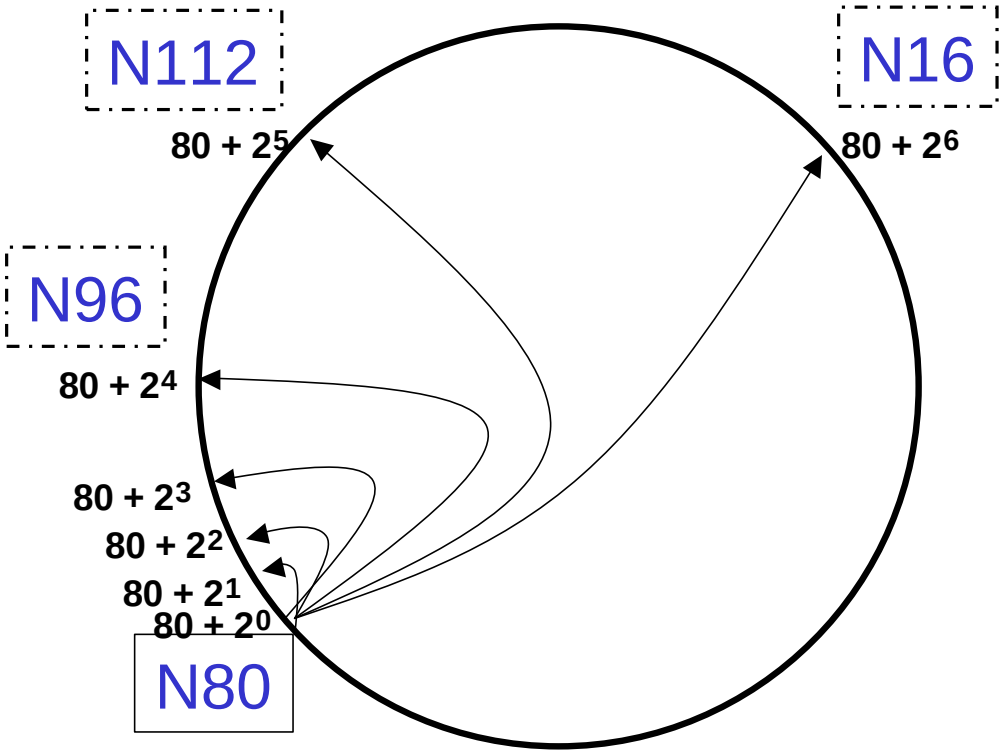
- Every node knows its successor in the ring



- requires O(N) time

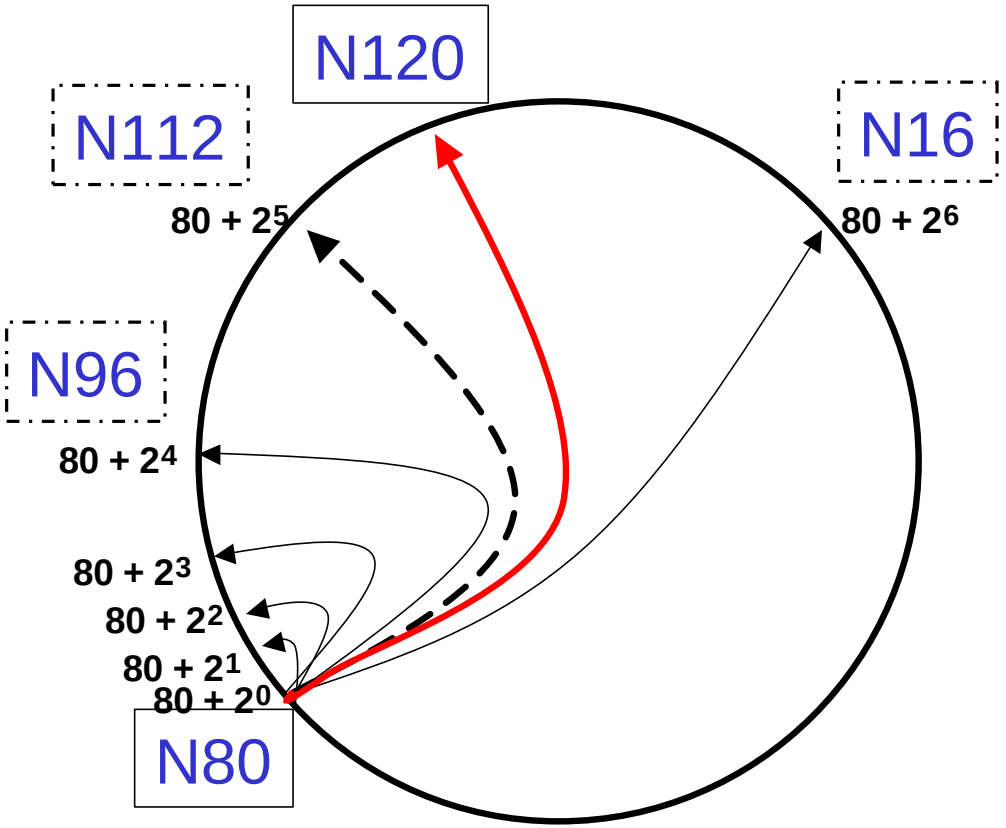
“Finger Tables”

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



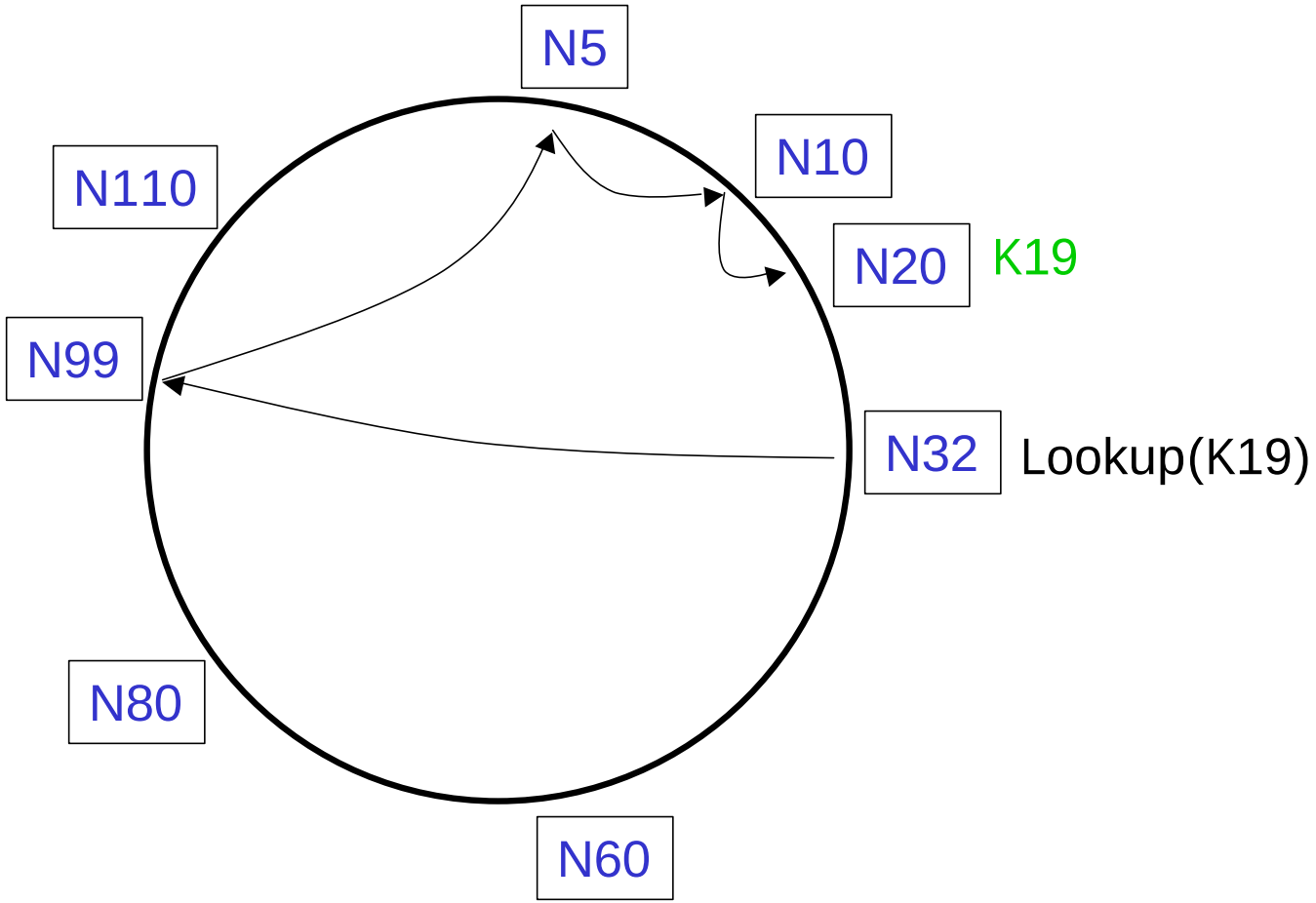
“Finger Tables”

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



Lookups are Faster

- Lookups take O(Log N) hops

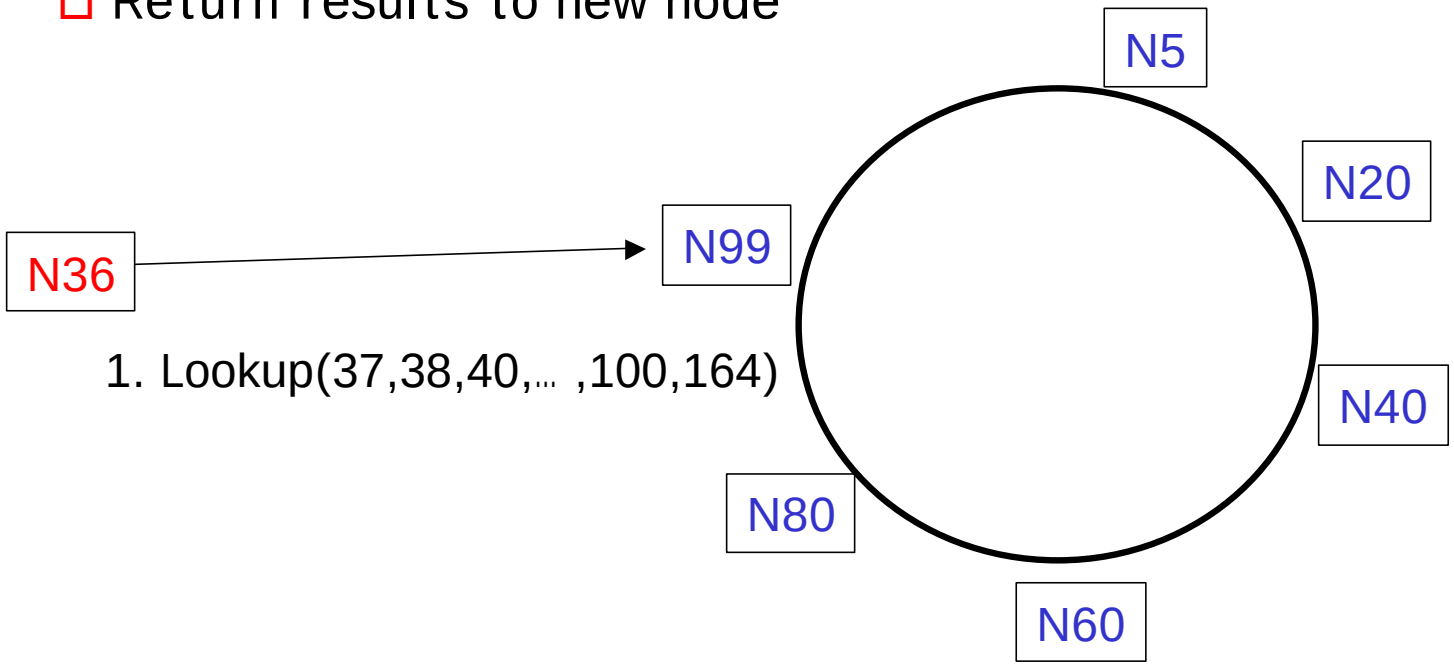


Joining the Ring

- Three step process:
  - Initialize all fingers of new node
  - Update fingers of existing nodes
  - Transfer keys from successor to new node
- Less aggressive mechanism (lazy finger update):
  - Initialize only the finger to successor node
  - Periodically verify immediate successor, predecessor
  - Periodically refresh finger table entries

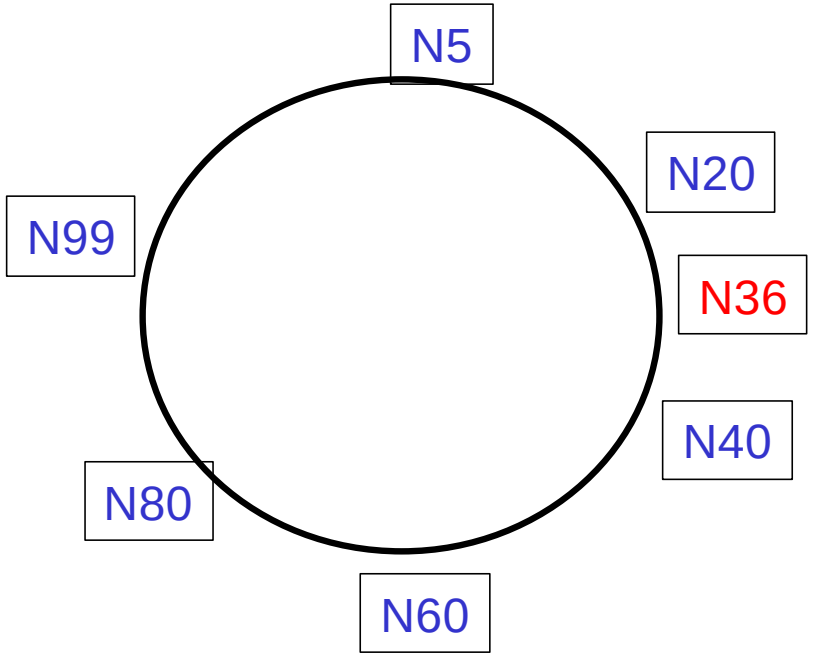
Joining the Ring - Step 1

- Initialize the new node finger table
  - Locate any node *p* in the ring
  - Ask node *p* to lookup fingers of new node N36
  - Return results to new node



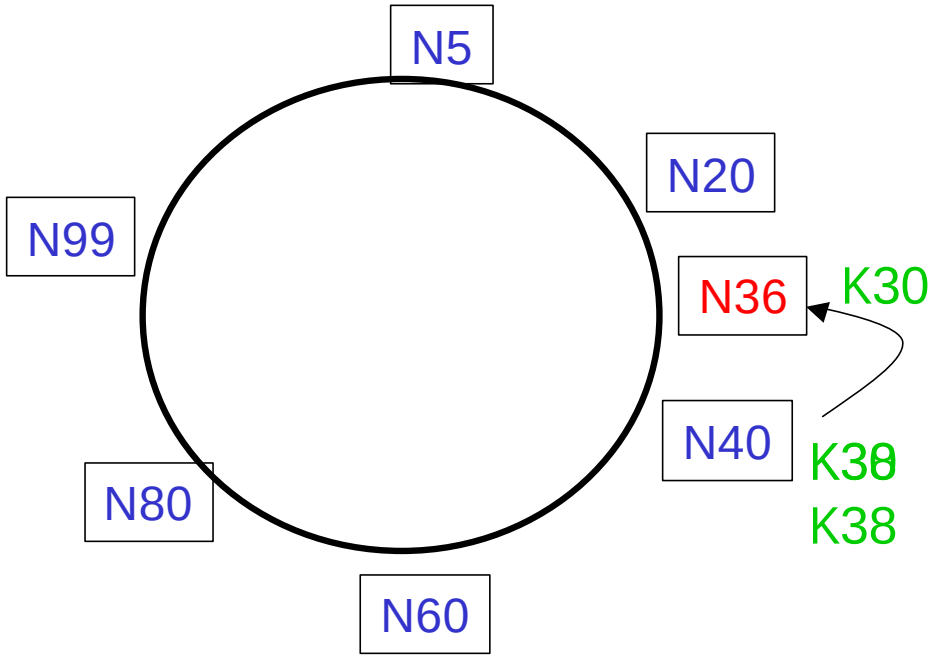
Joining the Ring - Step 2

- Updating fingers of existing nodes
  - new node calls update function on existing nodes
  - existing nodes can recursively update fingers of other nodes



Joining the Ring - Step 3

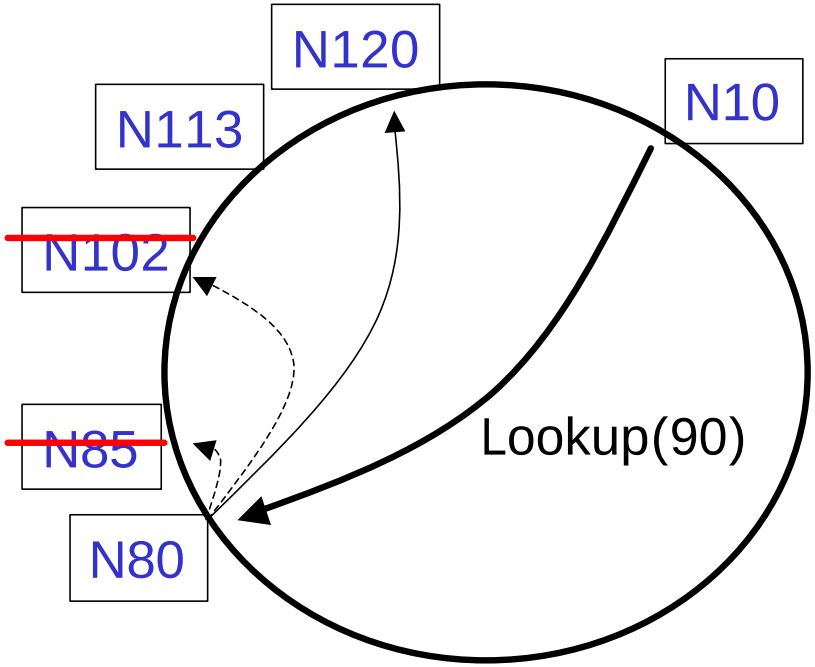
- Transfer keys from successor node to new node
  - only keys in the range are transferred



Copy keys 21..36
from N40 to N36

Handing Failures

- Failure of nodes might cause incorrect lookup



- N80 doesn't know correct successor, so lookup fails
- Successor fingers are enough for correctness

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

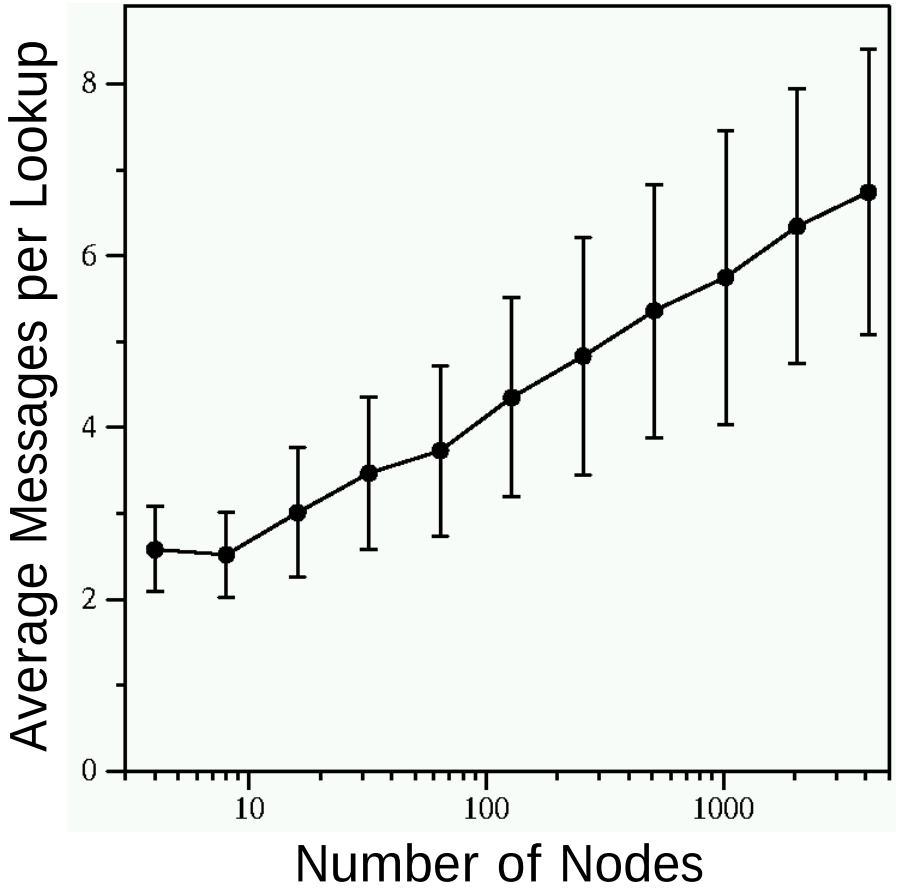
Evaluation Overview

- Quick lookup in large systems
- Low variation in lookup costs
- Robust despite massive failure

- Experiments confirm theoretical results

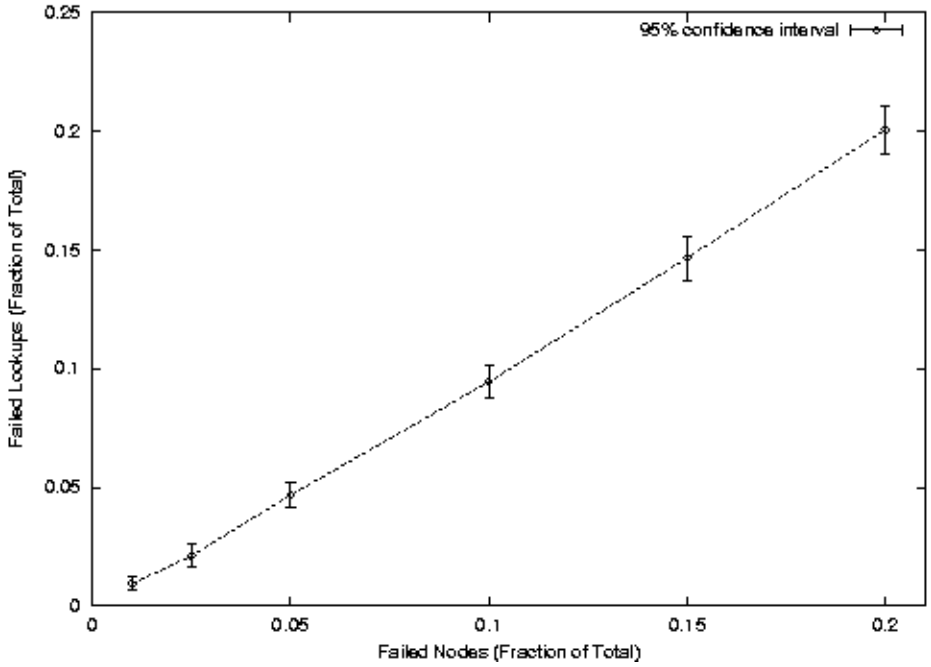
Cost of lookup

- Cost is O(Log N) as predicted by theory
- constant is 1/2



Robustness

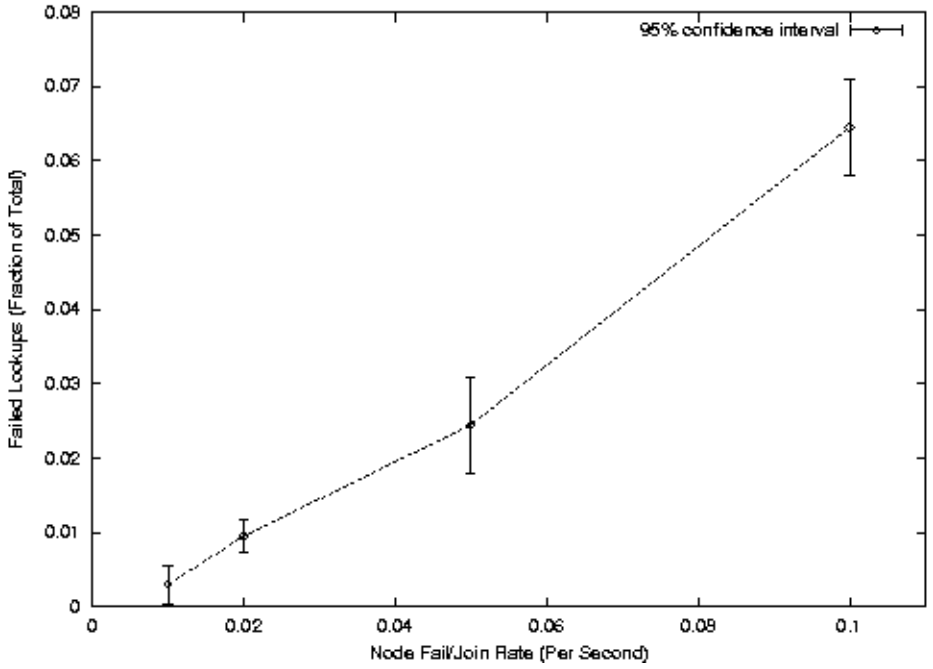
- Simulation results: static scenario
- Failed lookup means original node with key failed (no replica of keys)



- Result implies good balance of keys among nodes!

Robustness

- Simulation results: dynamic scenario
- Failed lookup means finger path has a failed node



- 500 nodes initially
- average *stabilize( )* call 30s
- 1 lookup per second (Poisson)
- x join/fail per second (Poisson)

Current implementation

- Chord library: 3,000 lines of C++
- Deployed in small Internet testbed
- Includes:
  - Correct concurrent join/fail
  - Proximity-based routing for low delay (?)
  - Load control for heterogeneous nodes (?)
  - Resistance to spoofed node IDs (?)

Strengths

- Based on theoretical work (consistent hashing)
- Proven performance in many different aspects
  - “with high probability” proofs
- Robust (Is it?)

Weakness

- **NOT** that simple (compared to CAN)
- Member joining is complicated
  - aggressive mechanisms requires too many messages and updates
  - no analysis of convergence in lazy finger mechanism
- Key management mechanism mixed between layers
  - upper layer does insertion and handle node failures
  - Chord transfer keys when node joins (no leave mechanism!)
- Routing table grows with # of members in group
- Worst case lookup can be slow

Discussions

- Network proximity (consider latency?)
- Protocol security
  - Malicious data insertion
  - Malicious Chord table information
- Keyword search and indexing
- ...