Chord: A scalable Peer-to-Peer Lookup Protocol for Internet Applications

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

MIT Laboratory for Computer Science

Chord@ics.mit.edu

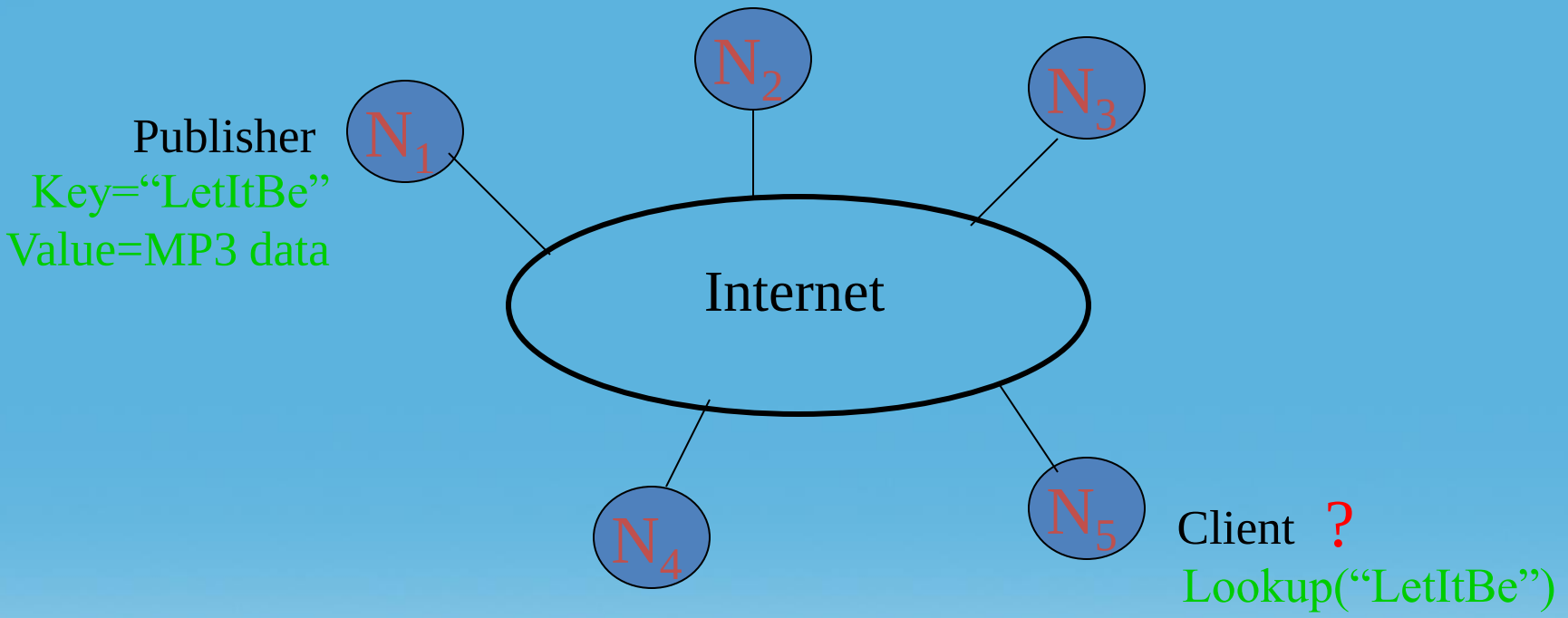
http://pdos.lcs.mit.edu/chord/

Outline

- Introduction
- System Model
- Chord Protocol
- Simulation and Experimental Results
- Future Work
- Weakness

Motivation

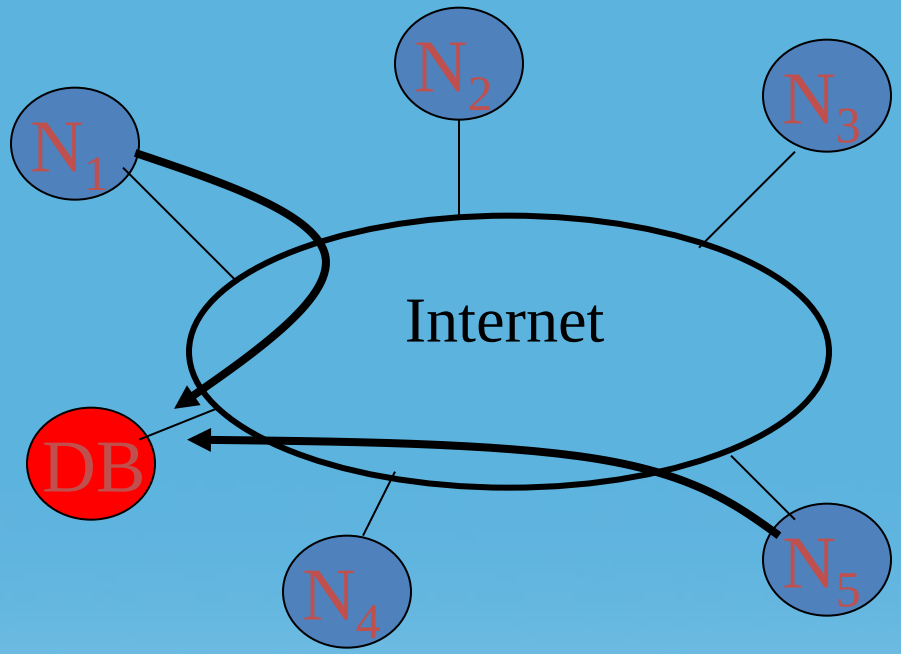
How to locate a data item in a dynamic peer-to-peer system?



Lookup is the key problem

Centralized Solution

Central server (Napster)



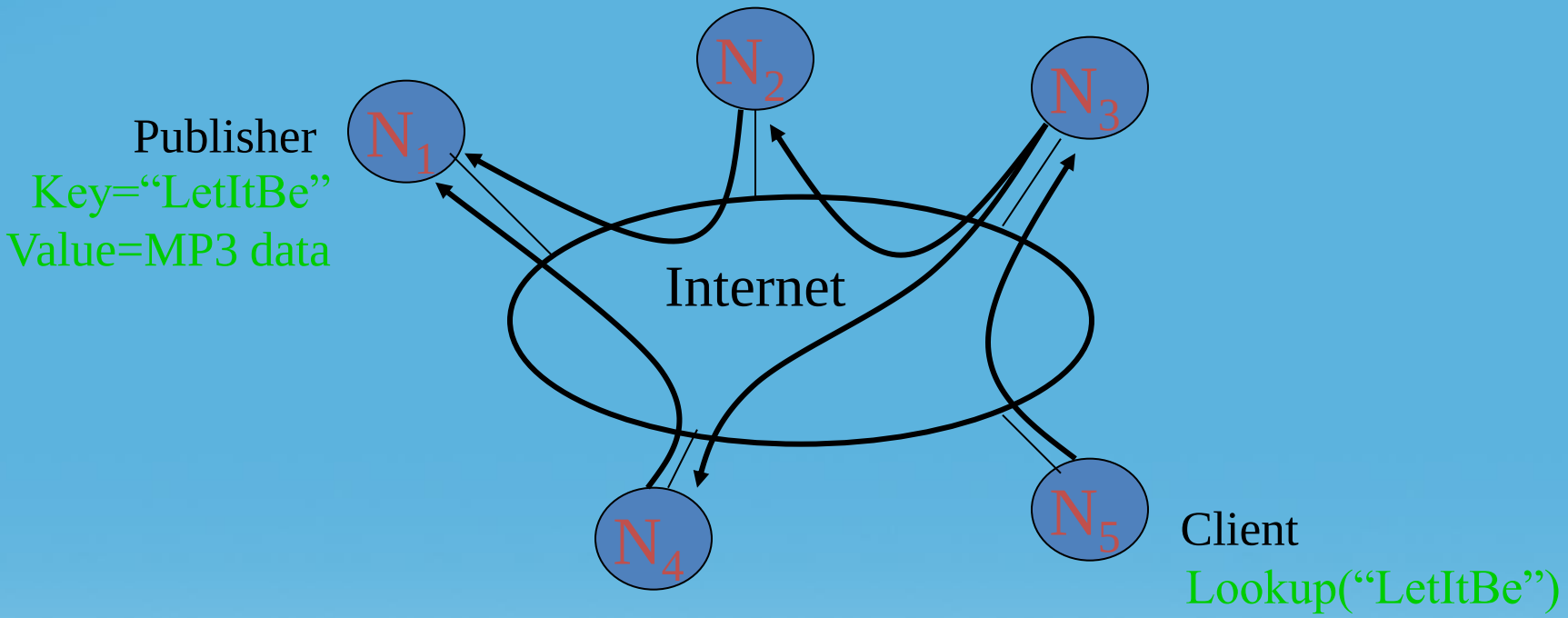
Publisher
Key="LetItBe"
Value=MP3 data

Client
Lookup("LetItBe")

Requires O(M) state

Distributed Solution (1)

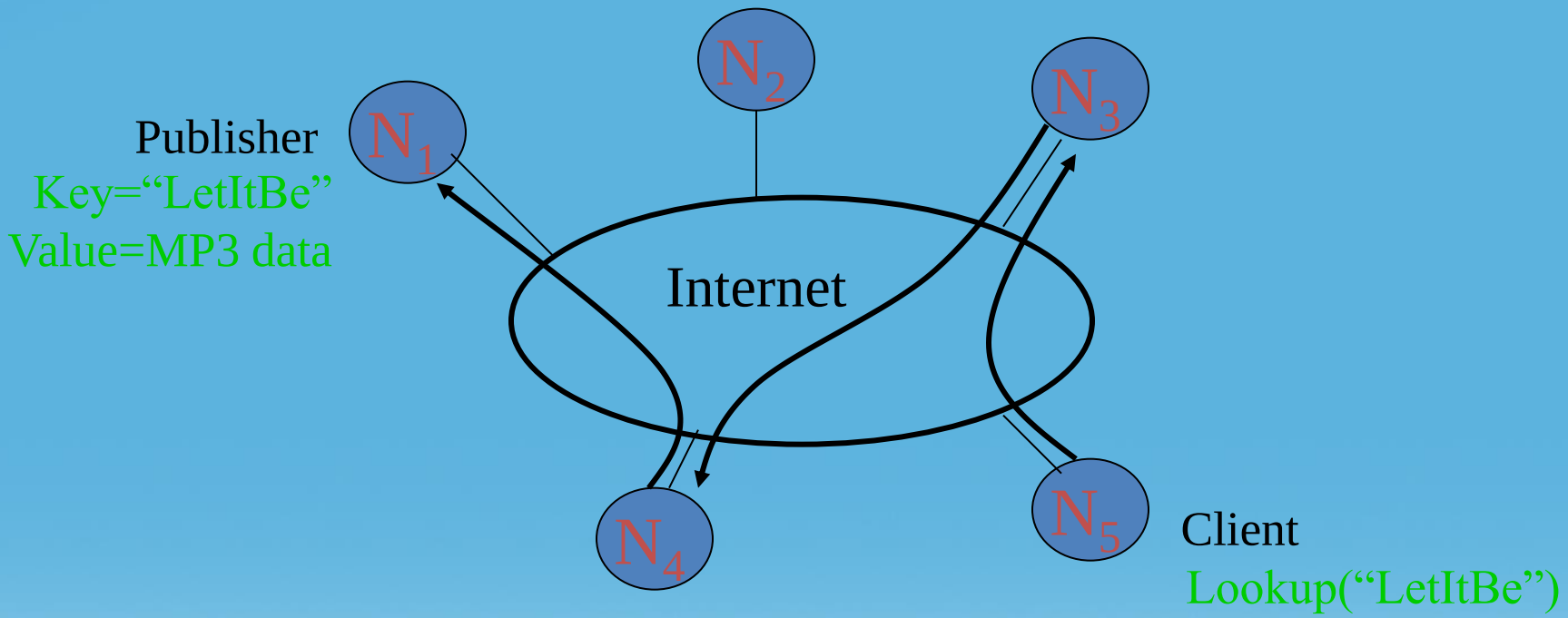
Flooding (Gnutella, Morpheus, etc.)



Worst case O(N) messages per lookup

Distributed Solution (2)

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



Only exact matches

Introduction

- Chord provides peer-to-peer hash lookup service:
  - Lookup(key) → IP address
- Features:

  Simplicity
  provable correctness
  provable performance

- How does Chord distribute files?
- How does Chord build routing tables?
- How does Chord locate a node?
- How does Chord maintain routing tables?
- How does Chord cope with changes in membership?

System Mode

- Load balance
  - Chord acts a distributed hash function
- Decentralization: fully distributed
- Scalability with high probability
  - O(log N) routing tables
  - O(log N) lookup
  - $O(\log^2 N)$ join/leave
- Availability
- Flexible naming

Chord Protocol
-- overview

- Fast distributed computation of a hash function mapping keys to nodes.
- Using *consistent hashing*

  -- load balance

  --minimum necessary to maintain a balanced load
- Scalability: A node needs a small amount of information

Chord Protocol
-- Consistent Hashing

- Each node and key has *an m-bit identifier*
- Node's identifier
  - hashing the node's IP address
- Key's identifier
  - hashing the key
- Key k is assigned to the *successor(k)*
  - identifier of successor(k) is equal to or follows k's identifier

Chord IDs

m bit identifier space for both keys and nodes

Key identifier = SHA-1(key)

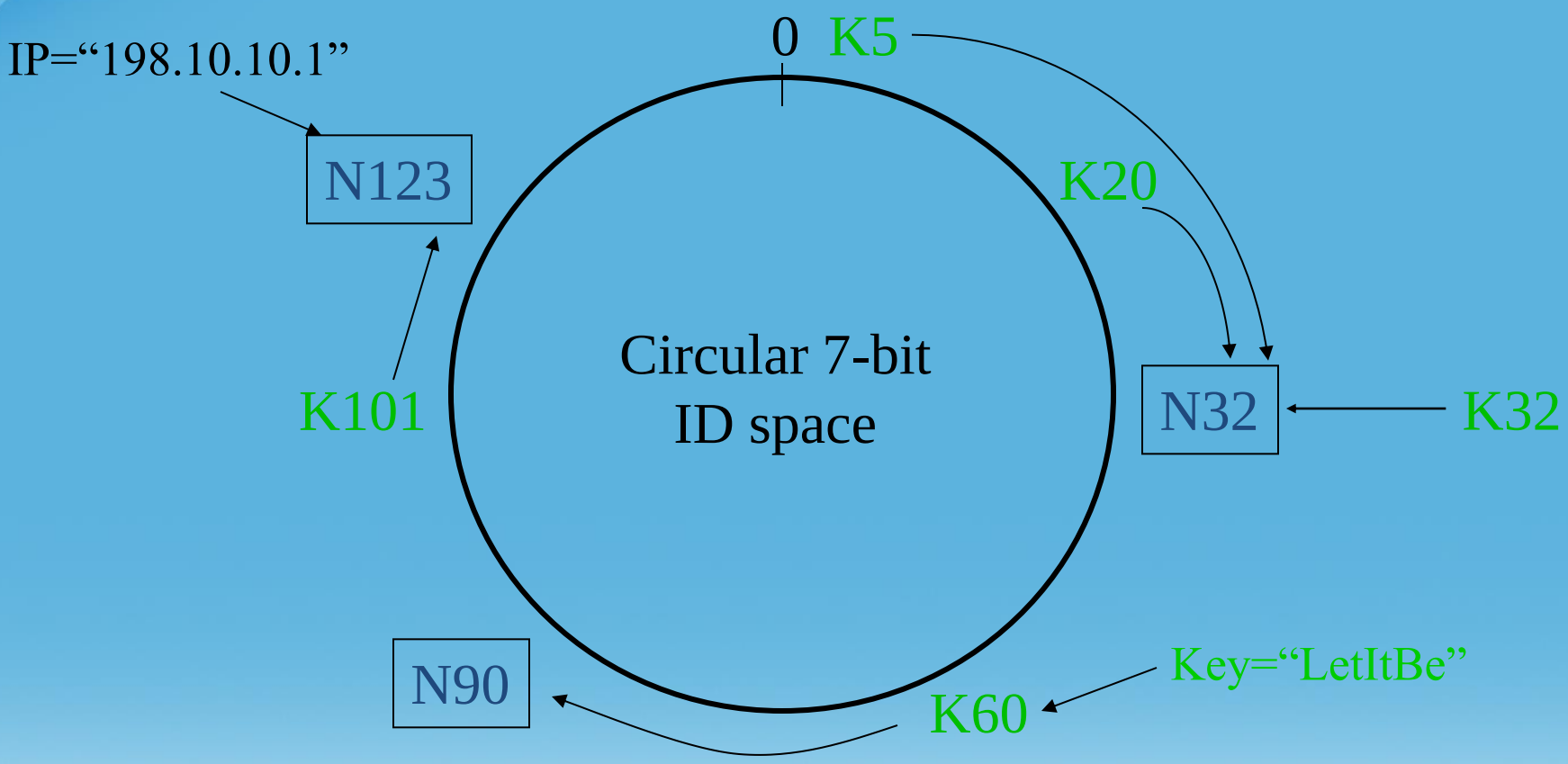
Key="LetItBe" —SHA-1→ ID=60

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?



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

THEOREM 1.

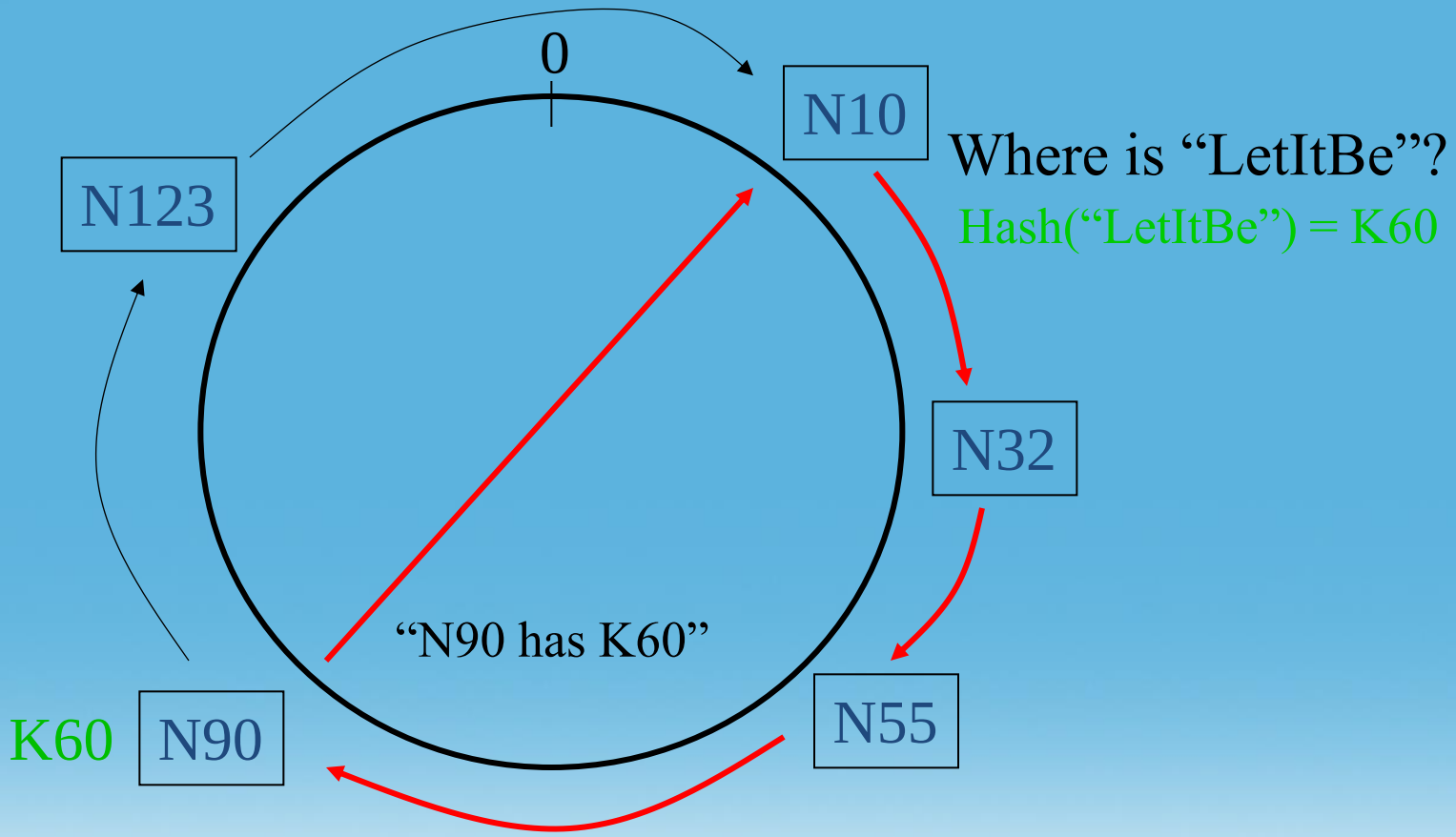
For any set of N nodes and K keys, with high probability:

1. Each node is responsible for at most $(1+\epsilon)K/N$ keys ($\epsilon=O(\log N)$)

2. When an $(N+1)^{st}$ node joins or leaves the network, responsibility for $O(K/N)$ keys changes hands.

Simple Key Location

Every node knows its successor in the ring



Where is "LetItBe"?

Hash("LetItBe") = K60

requires O(N) time

Scalable Key Location

How to maintain the successor information correctly?

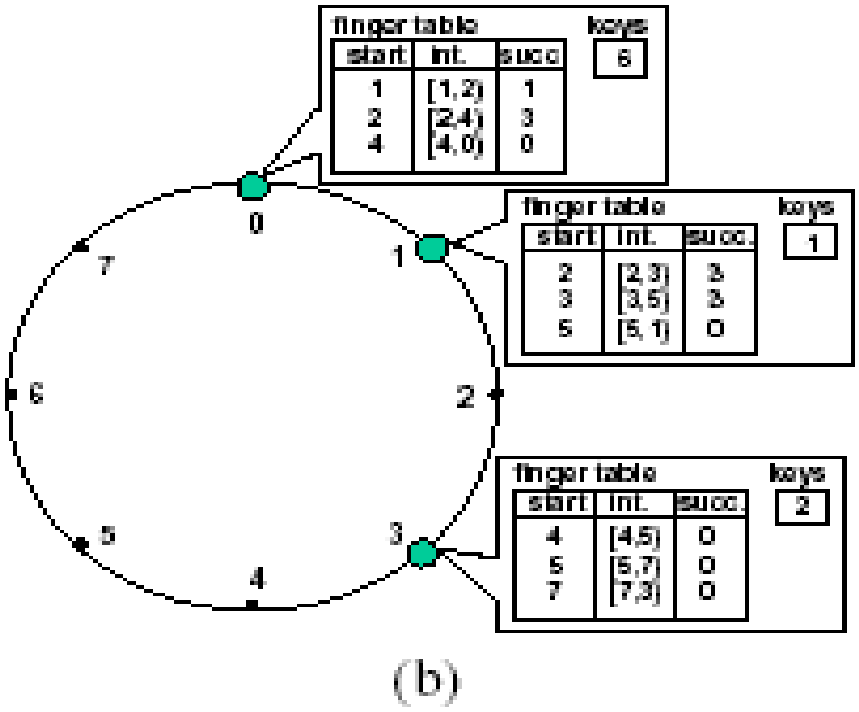
- N, maintains a routing table with m entries, called the *finger table*
- *i^{th} entry is* S = successor($n+2^{i-1}$)

  -- S succeeds n by at least 2^{i-1} on the identifier circle

Scalable Key Location

-- finger table example

Finger tables and key locations for a net with nodes 0, 1, and 3 and keys 1, 2 and 6.



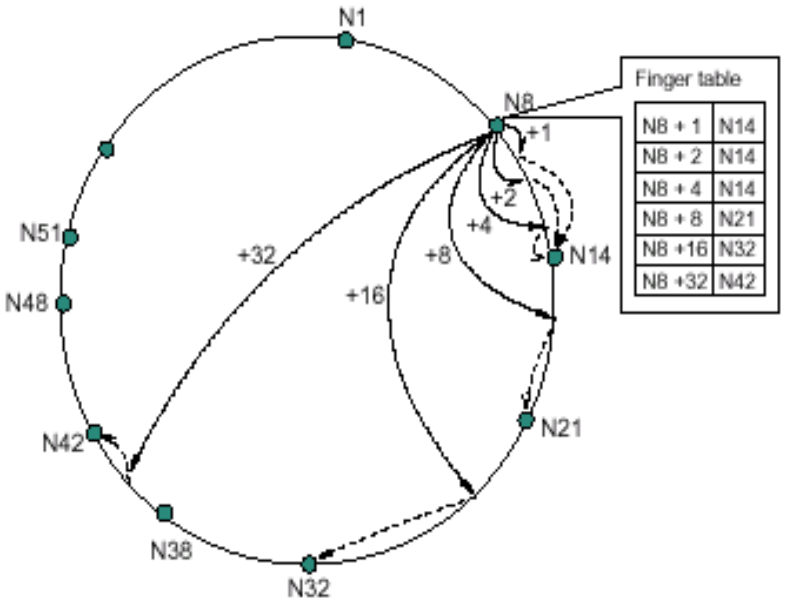
(b)

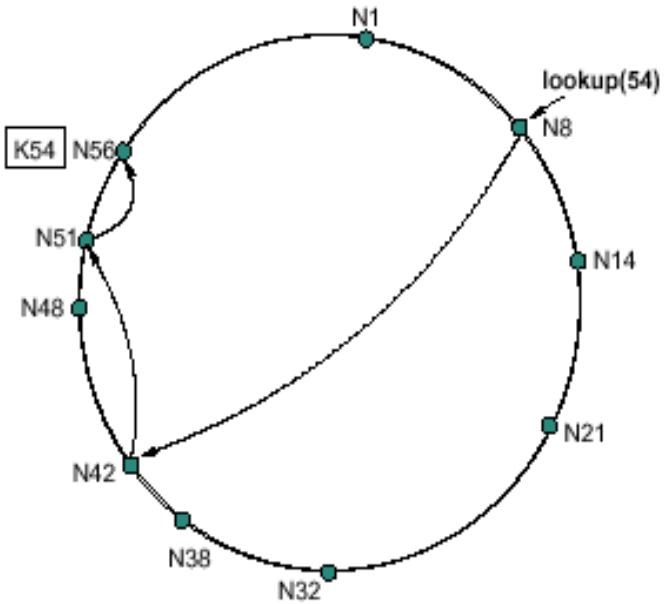
Scalable Key Location

--What happens when a node does not know the successor of a key k?

- N finds a node whose ID is closer than its own to k
- N searches its finger table for the node j, whose ID most immediately precedes k
- N asks j for the node it knows whose ID is closest to k
- Repeat this process, N learn s about nodes with ids closer and closer to k

A faster algorithm uses a “finger” table on each node, somewhat similar to a skip



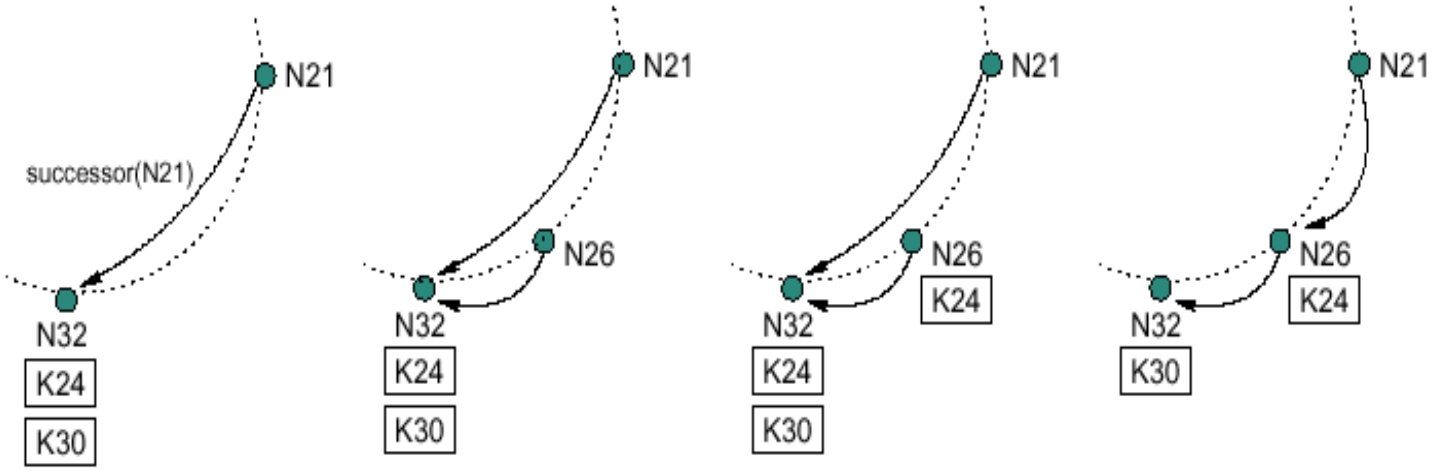


THEOREM 2.

With high probability, the number of nodes that must be contacted to find a successor in an N-node network is O(logN);

Node Join Implementation

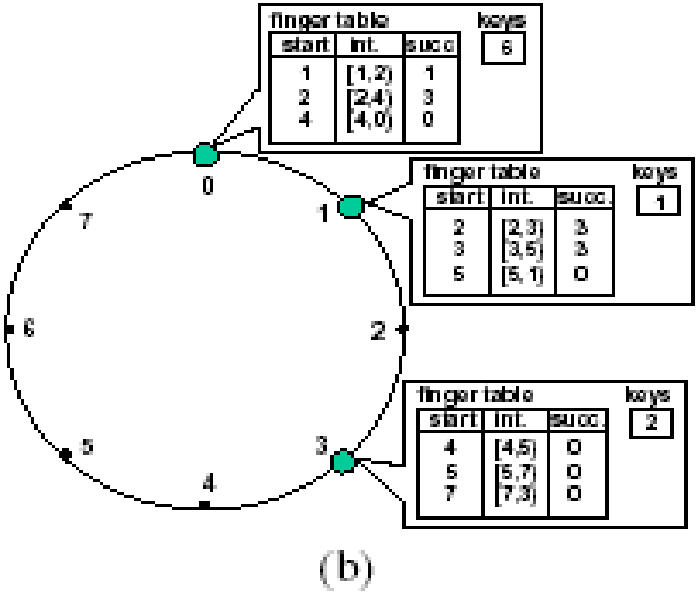
- Three step process:
  - Initialize all fingers of new node
  - Update fingers of existing nodes
  - Transfer keys from successor to new node



Node Join Implementation

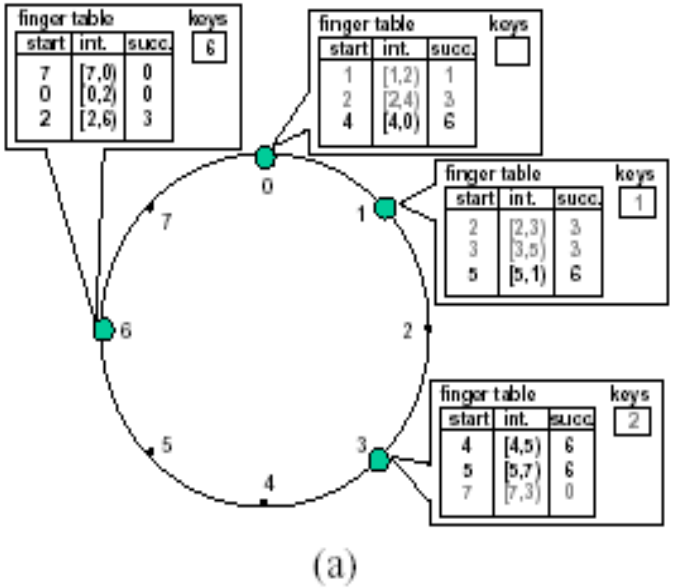
Before Node6 joining

Changed entries are shown in black, and unchanged in gray



(b)

After Node6 joining



(a)

- To ensure locating every key in the network, Chord needs to preserve two invariants:

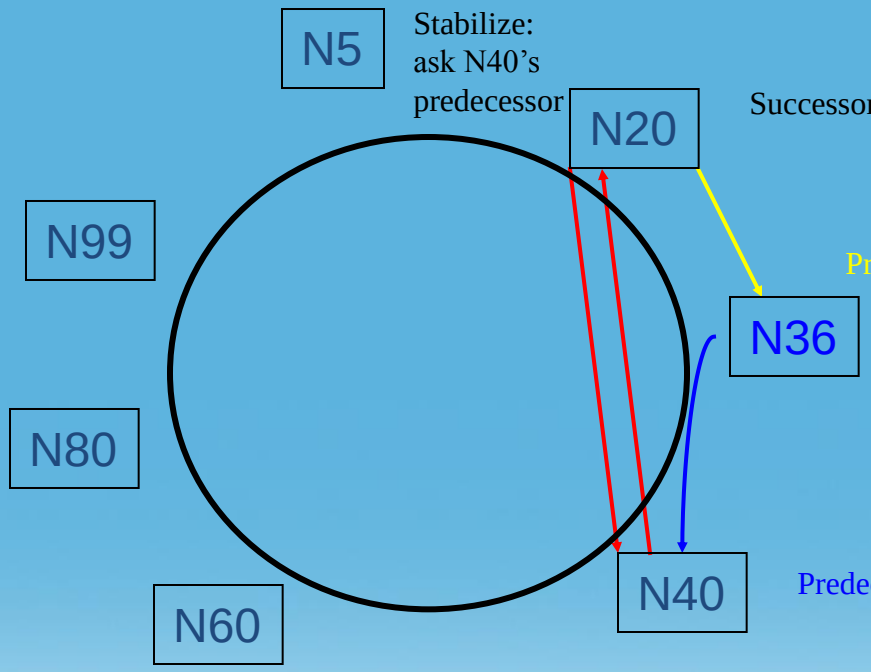
1. Each node's successor is correctly maintained
2. For every key k, node successor(k) is responsible for k

- Less aggressive mechanism (lazy finger update):
  - Initialize only the finger to successor node
  - Periodically verify immediate successor, predecessor
  - Periodically refresh finger table entries

Stabilization

-- to keep nodes' successor pointers up to date

N36 is a newly-joint node.



1. N36 notifies N40 of its existence

2. N40 updates its pre. To N36

3. N20 asks its successor N40 for N40's predecesso

4. N20 and N36 update their successor and predecessor

THEOREM 3.

If any sequence of join operations is executed interleaved with stabilizations, then at some time after the last join the successor pointers will form a cycle on all the nodes in the network

Impact of Node Joins on Lookups

- Lookup behavior during joins
  - lookup fails if successor/predecessor are incorrect
    - the higher level software needs to retry
  - lookup succeeds, but it is slower if fingers are not yet updated; in most cases still O(log N)

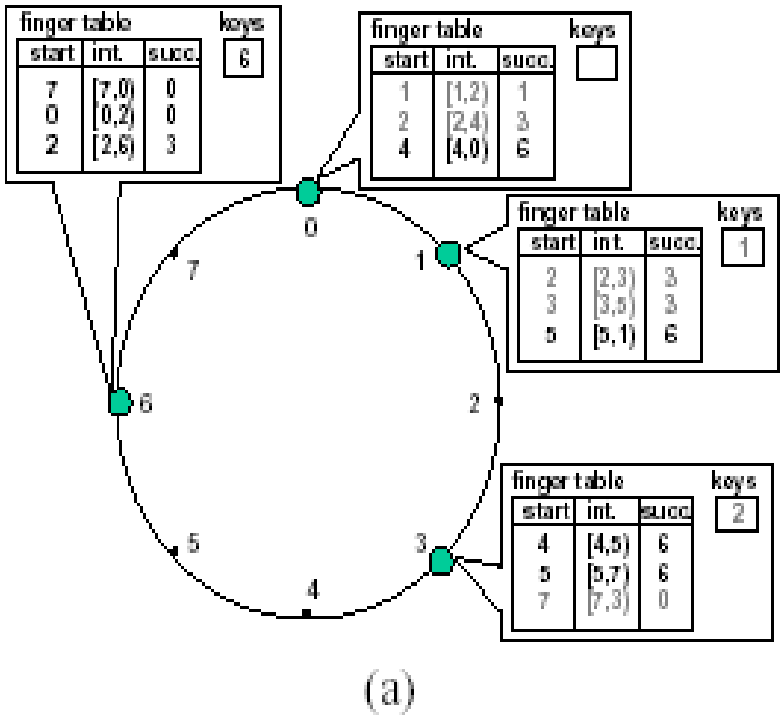
- THEOREM 6.

  If we take a stable network with N nodes, and another set of up to N nodes joins the network with no finger pointers (but with correct successor pointers), then lookups will still take O(log N) time with high probability

Node Leaving example

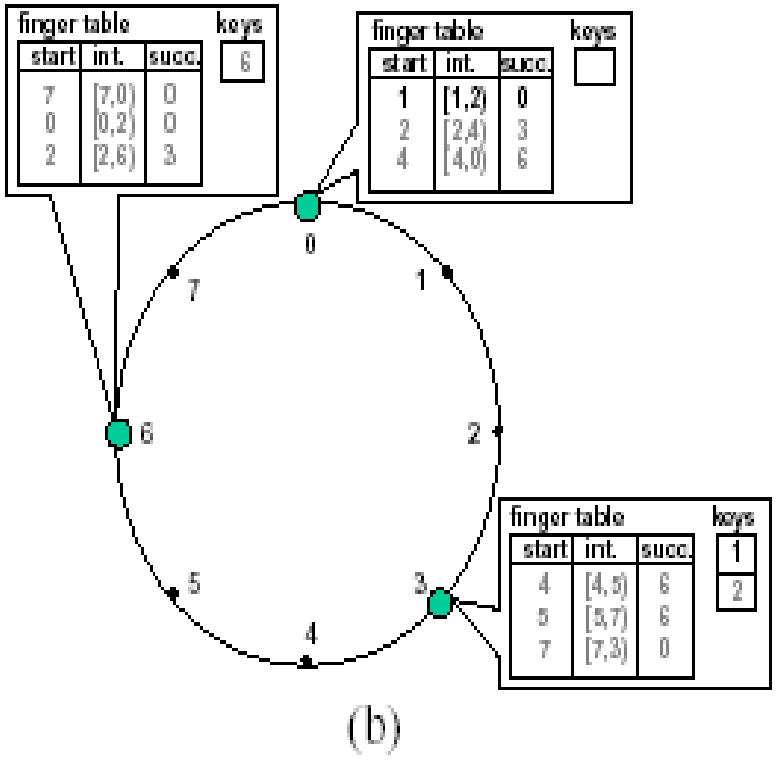
Before Node1 Leaving

Changed entries are shown in black, and unchanged in gray



(a)

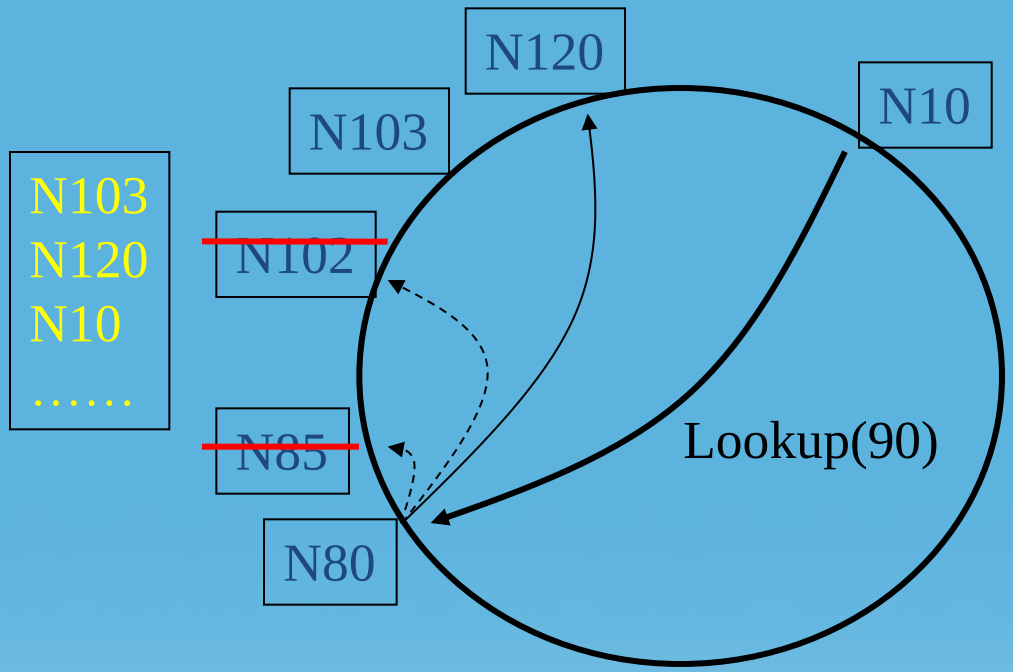
After Node1 Leaving



(b)

Handing Failures

Failure of nodes might cause incorrect lookup



N80 doesn’t know correct successor, so lookup fails

Successor List are enough for correctness

Handling Failures

Use successor list

-- Each node knows *r* immediate successors

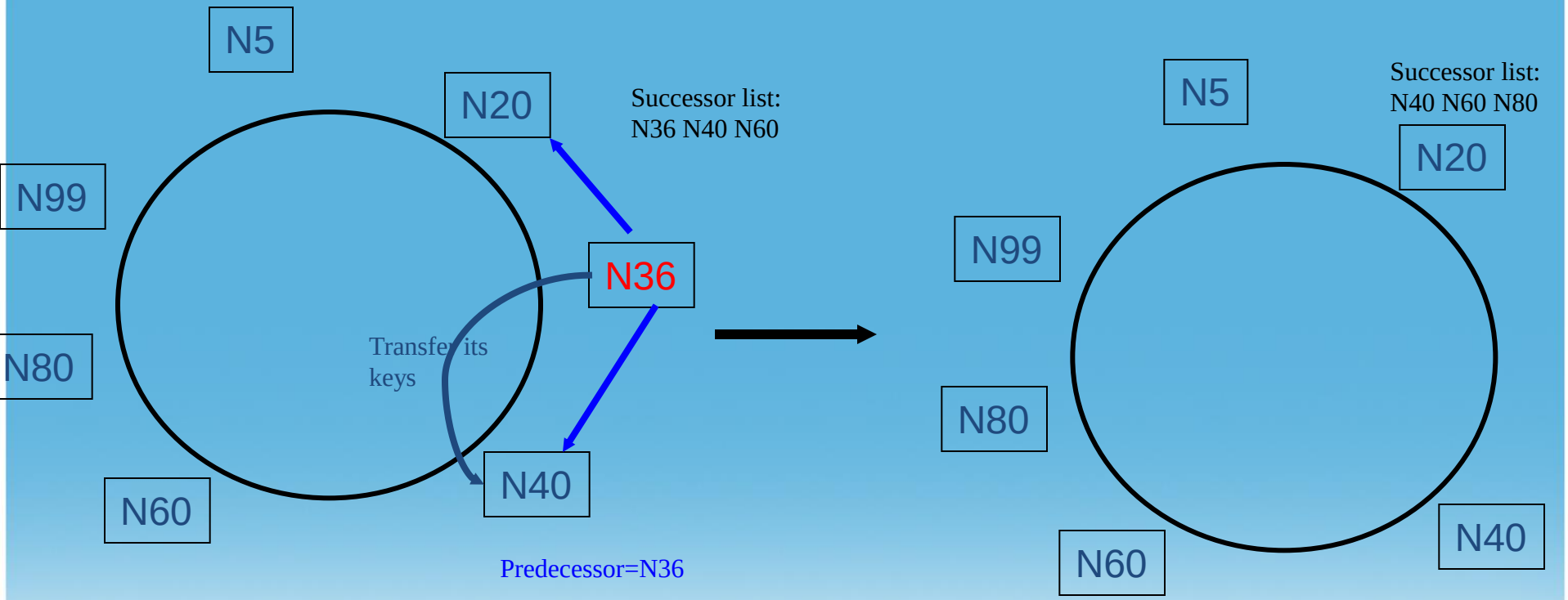
-- After failure, will know first live successor

-- Correct successors guarantee correct lookups

Guarantee is with some probability

choose *r* to make probability of lookup failure arbitrarily small

Voluntary Node Departures



- THEOREM 5.

  If we use a successor list of length r=O(log N) in a network that is initially stable, and then every node fails with probability ½, then with high probability find-successor returns the closest living successor to the query key

- THEOREM 6.

  In a network that is initially stable, if every node then fails with probability ½, then the expected time to execute find-successor is O(logN).

Simulation and Experiment Results

- Load Balance

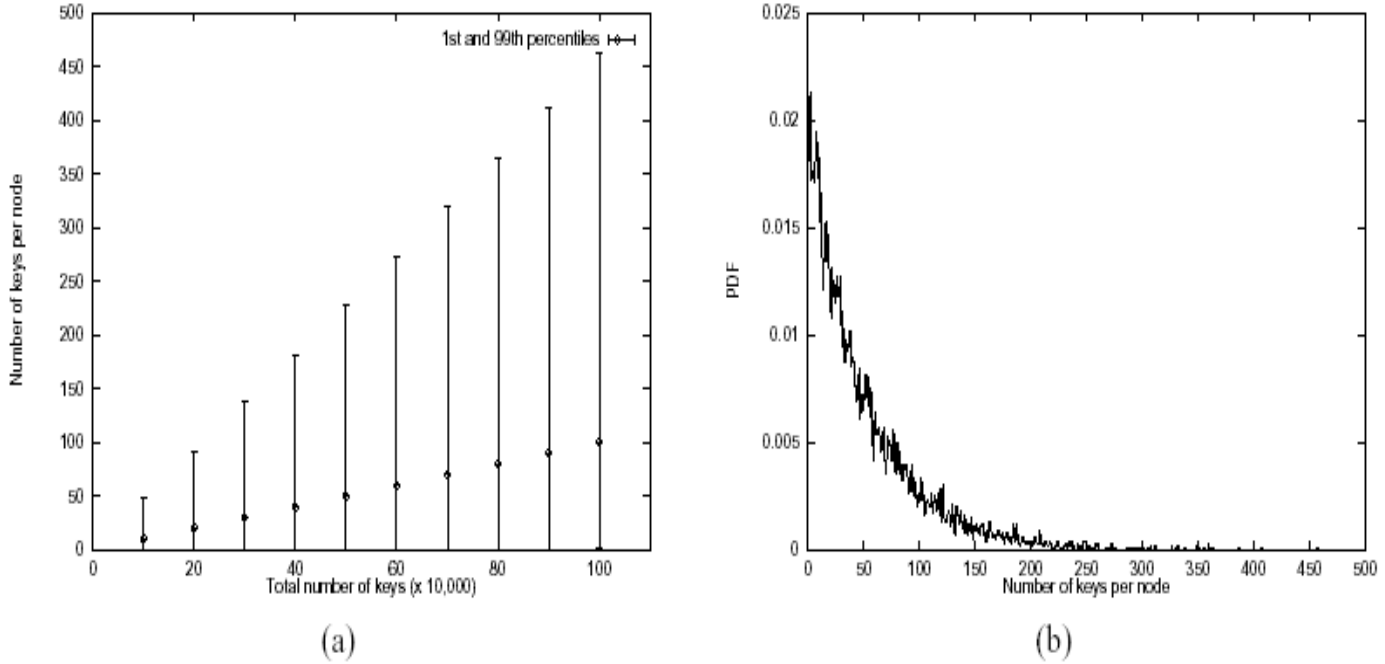


Figure 7: (a) The mean value, the 1st and the 99th percentiles of the number of keys stored by a node in a 10 node network. (b) The probability density function (PDF) of the number of keys per node. The total number of keys is 5×10^5.

Application of Virtual nodes

The 99th percentile decreases from 4.8x to 1.6x the mean value, while the 1st percentile increases from 0 to 0.5 the mean value

– adding virtual nodes as an indirection layer can significantly improve load balance

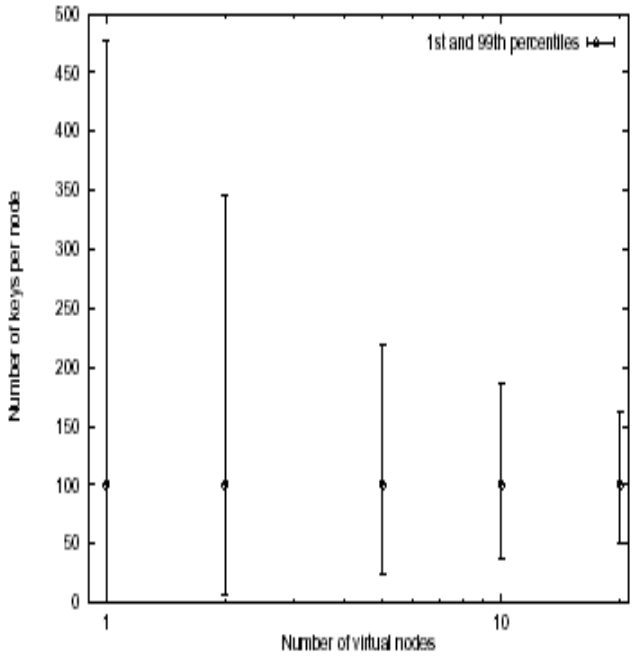


Figure 8: The 1st and the 99th percentiles of the number of keys per node as a function of virtual nodes mapped to a real node. The network has 10^2 real nodes and stores 10^6 keys.

Path Length

N = 2^k, storing 100x 2^k keys in all. K is varied from 3 to 14 and each node picked a random set of keys to query from the system.

The measured path length is about 1/2logN

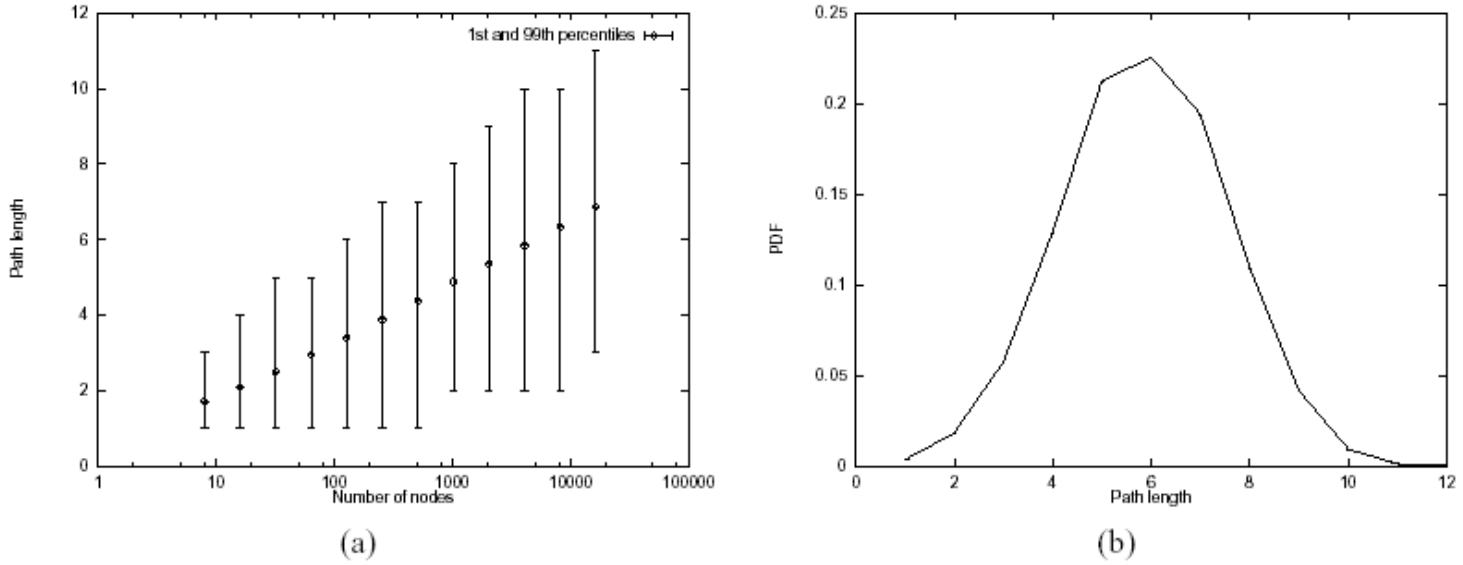


Figure 9: (a) The path length as a function of network size. (b) The PDF of the path length in the case of a 2^{12} node network.

Simultaneous Node Failures

The path length and the number of timeouts experienced by a lookup as function of the fraction of nodes that fail simultaneously. The 1st and the 99th percentiles are in parenthesis. Initially, the network has 1000 nodes.

- Predicted value is a little larger than the measured value because the series is finite in practice
- Timeouts match well the measure number
- All lookups were successfully resolved – robustness

Fraction of failed nodes	Mean path length (1st, 99th percentiles)	Mean num. of timeouts (1st, 99th percentiles)
0	3.84 (2, 5)	0.0 (0, 0)
0.1	4.03 (2, 6)	0.60 (0, 2)
0.2	4.22 (2, 6)	1.17 (0, 3)
0.3	4.44 (2, 6)	2.02 (0, 5)
0.4	4.69 (2, 7)	3.23 (0, 8)
0.5	5.09 (3, 8)	5.10 (0, 11)

TABLE II

Lookups During Stabilization

Key lookups, stabilization are modeled with a certain rate. Change the joins and voluntary leaves rate.

- Measured path length is very close to the predicted value
- Measured timeouts are reasonable close to the predicted value
- Reason for the lookup failures is state inconsistency

Node join/leave rate (per second/per stab. period)	Mean path length (1st, 99th percentiles)	Mean num. of timeouts (1st, 99th percentiles)	Lookup failures (per 10,000 lookups)
0.05 / 1.5	3.90 (1, 9)	0.05 (0, 2)	0
0.10 / 3	3.83 (1, 9)	0.11 (0, 2)	0
0.15 / 4.5	3.84 (1, 9)	0.16 (0, 2)	2
0.20 / 6	3.81 (1, 9)	0.23 (0, 3)	5
0.25 / 7.5	3.83 (1, 9)	0.30 (0, 3)	6
0.30 / 9	3.91 (1, 9)	0.34 (0, 4)	8
0.35 / 10.5	3.94 (1, 10)	0.42 (0, 4)	16
0.40 / 12	4.06 (1, 10)	0.46 (0, 5)	15

TABLE III

Improving Routing Latency

Motivation:

the node identifiers are randomly distributed, and therefore nodes close in the identifier space can be far away in the underlying network.

Solution:

- Each finger maintain a set of alternate nodes.
- Route the queries by selecting the node among the alternate nodes according to some network proximity metirc

Experimental Results

- The lookup stretch of Chord system with 216 nodes and two network topologies are measured (3-d space and Transit stub)
- The *lookup stretch* is defined as the ratio between the

  -- latency of a Chord lookup

  -- latency of an optimal lookup using the underlying network
- Results show that this heuristic is quite effective, the stretch decreases significantly as s increases.

Number of fingers' successors (s)	Stretch (10th, 90th percentiles)			
	Iterative		Recursive	
	3-d space	Transit stub	3-d space	Transit stub
1	7.8 (4.4, 19.8)	7.2 (4.4, 36.0)	4.5 (2.5, 11.5)	4.1 (2.7, 24.0)
2	7.2 (3.8, 18.0)	7.1 (4.2, 33.6)	3.5 (2.0, 8.7)	3.6 (2.3, 17.0)
4	6.1 (3.1, 15.3)	6.4 (3.2, 30.6)	2.7 (1.6, 6.4)	2.8 (1.8, 12.7)
8	4.7 (2.4, 11.8)	4.9 (1.9, 19.0)	2.1 (1.4, 4.7)	2.0 (1.4, 8.9)
16	3.4 (1.9, 8.4)	2.2 (1.7, 7.4)	1.7 (1.2, 3.5)	1.5 (1.3, 4.0)

Strengths

Based on theoretical work (consistent hashing)

Proven performance in many different aspects "with high probability" proofs

Future work

- No specific mechanism to heal partitioned rings
- Find a way to check the malicious or buggy set of Chord participants
  - -- Malicious data insertion
  - -- Malicious Chord table information
- logN messages per lookup many be too many for some applications of Chord
- ……

Weakness

* Hashing both nodes and keys completely destroys locality
  - advantage: resistance to geographic attacks
  - disadvantage: longer network hops

* Chord does not provide a degree of anonymity compared to Freenet whose lookups take the form of searches for cached copies.

* **NOT** that simple (compared to CAN)

* Member joining is complicated

  -- requires too many messages and updates

* Routing table grows with number of members in group

* Worst case lookup can be slow

* ……

Thank you !

Any question?