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

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

SIGCOMM'01, August 2001

Presented by Byung Choi

Outline

- Introduction
- System Model
- The Base Chord Protocol
- Concurrent Operations and Failures
- Simulation and Experimental Results
- Future Work & Conclusion

Introduction

- Peer-to-peer systems and applications are distributed systems without any centralized control or hierarchical organization.
- The core operation in most peer-to-peer systems is efficient location of data items.
- The *Chord protocol* supports just one operation: given a key, it maps the key onto a node.

System Model

- **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.

System Model (cont.)

- The application interacts with Chord in two main ways:
  - Chord provides a lookup(key) algorithm that yields the IP address of the node responsible for the key.
  - The Chord software on each node notifies the application of changes in the set of keys that the node is responsible for.

Distributed Storage System Based on Chord

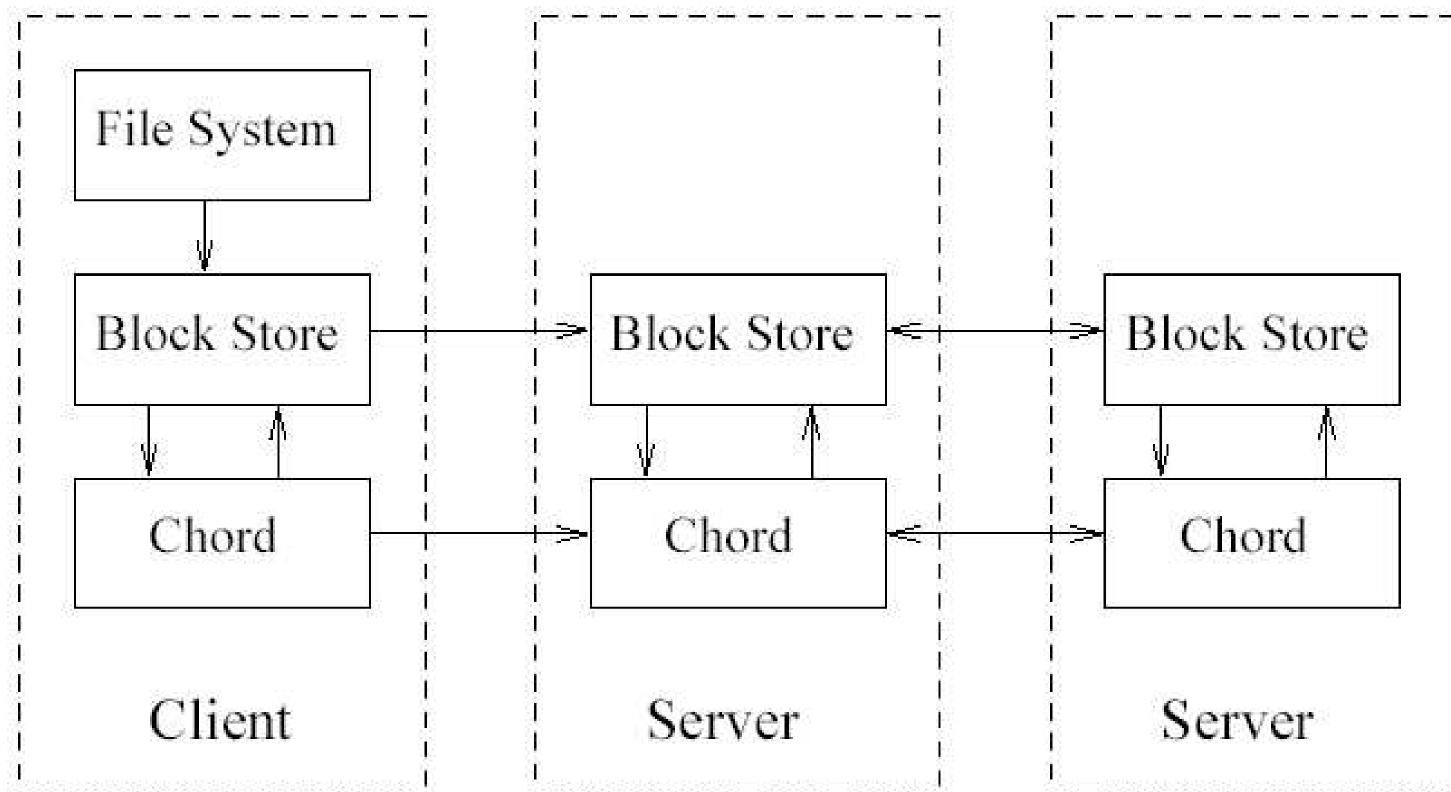

Figure 1: Structure of an example Chord-based distributed storage system.

The Base Chord Protocol

- The Chord protocol specifies how to find the locations of keys.
- It uses consistent hashing, all nodes receive roughly the same number of keys.
- When an Nth node joins (or leaves) the network, only an $O(1/N)$ fraction of the keys are moved to a different location.
- Improves the scalability of consistent hashing by avoiding the requirement that every node know about every other node.
- In an N-node network, each node maintains information only about $O(\log N)$ other nodes, and a lookup requires $O(\log N)$ messages.

Consistent Hashing

- The consistent hash function assigns each node and key an *m*-bit *identifier* using a base hash function such as SHA-1.
- Identifiers are ordered in an *identifier circle* modulo 2^m.
- Key *k* is assigned to the first node whose identifier is equal to or follows *k* in the identifier space. This node is called the *successor node* of key *k*.
- If identifiers are represented as a cycle of numbers from 0 to $2^m - 1$, then *successor*(*k*) is the first node clockwise from *k*.

Consistent Hashing (cont.)

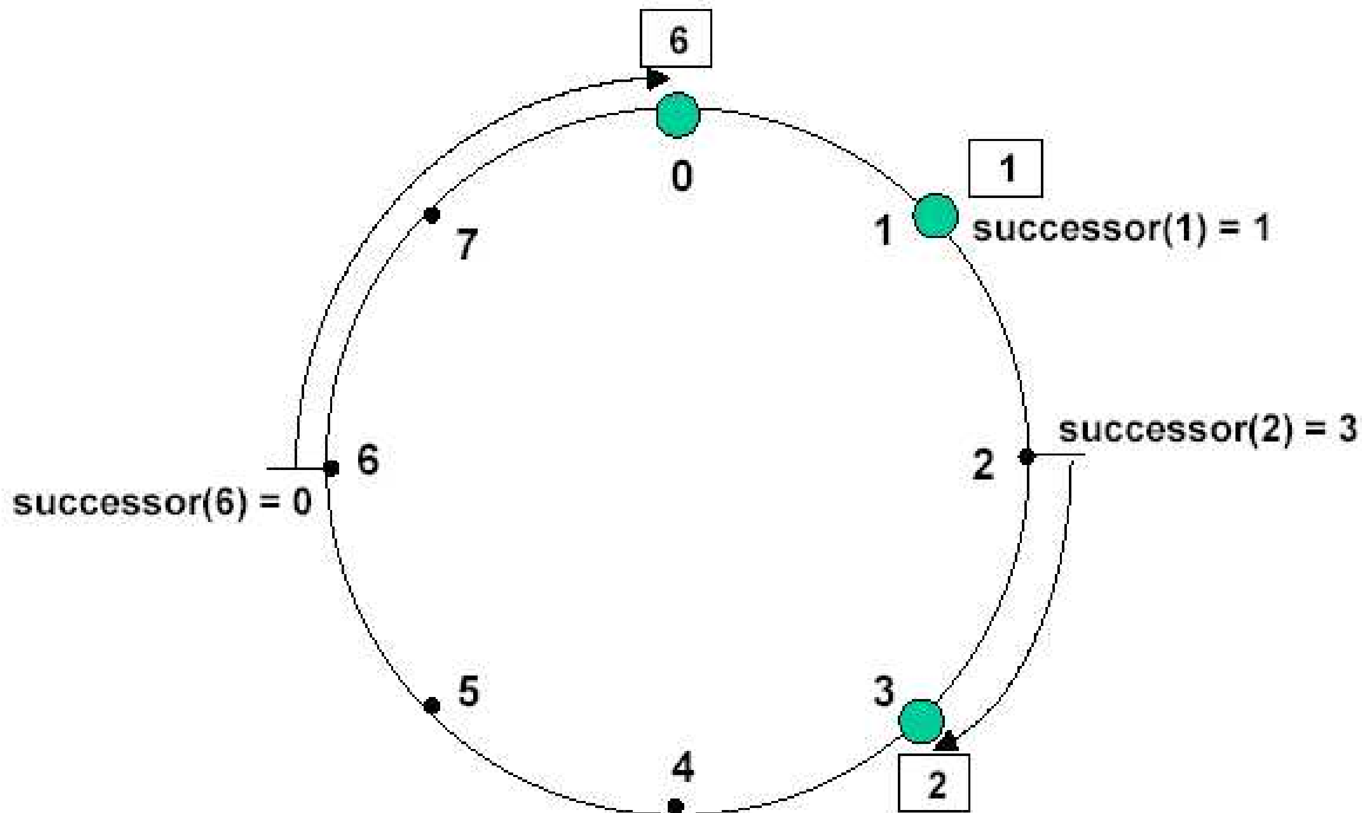

Figure 2. An identifier circle consisting of the three nodes 0, 1, and 3. In this example, key 1 is located at node 1, key 2 at node 3, and key 6 at node 0.

Consistent Hashing (cont.)

THEOREM 1. For any set of N nodes and K keys, with high probability:

1. Each node is responsible for at most $(1+\ )K/N$ keys.
2. When an $(N+1)^{st}$ node joins or leaves the network, responsibility for $O(K/N)$ keys changes hands (and only to or from the joining or leaving node).

Scalable key Location

- **Let *m* be the number of bits in the key/node identifiers.**
- **Each node, *n*, maintains a routing table with (at most) *m* entries, called the *finger table*.**
- **The ith entry in the table at node *n* contains the identity of the *first* node, *s*, that succeeds *n* by at least $2^{i}-1$ on the identity circle.**

Scalable key Location (cont.)

Notation	Definition
finger$[k]$.*start*	$(n + 2^{k-1}) \bmod 2^m$, $1 \le k \le m$
.*interval*	$[$*finger*$[k]$.*start*, *finger*$[k+1]$.*start*$)$
.*node*	first node $\ge n$.*finger*$[k]$.*start*
successor	the next node on the identifier circle; *finger*$[1]$.*node*
predecessor	the previous node on the identifier circle

Table 1: Definition of variables for node *n*, using *m*-bit identifiers.

Scalable key Location (cont.)

```
// 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;
```

Figure 4: The pseudo code to find the successor node of an identifier *id*. Remote procedure calls and variable lookups are preceded by the remote node.

Scalable key Location (cont.)

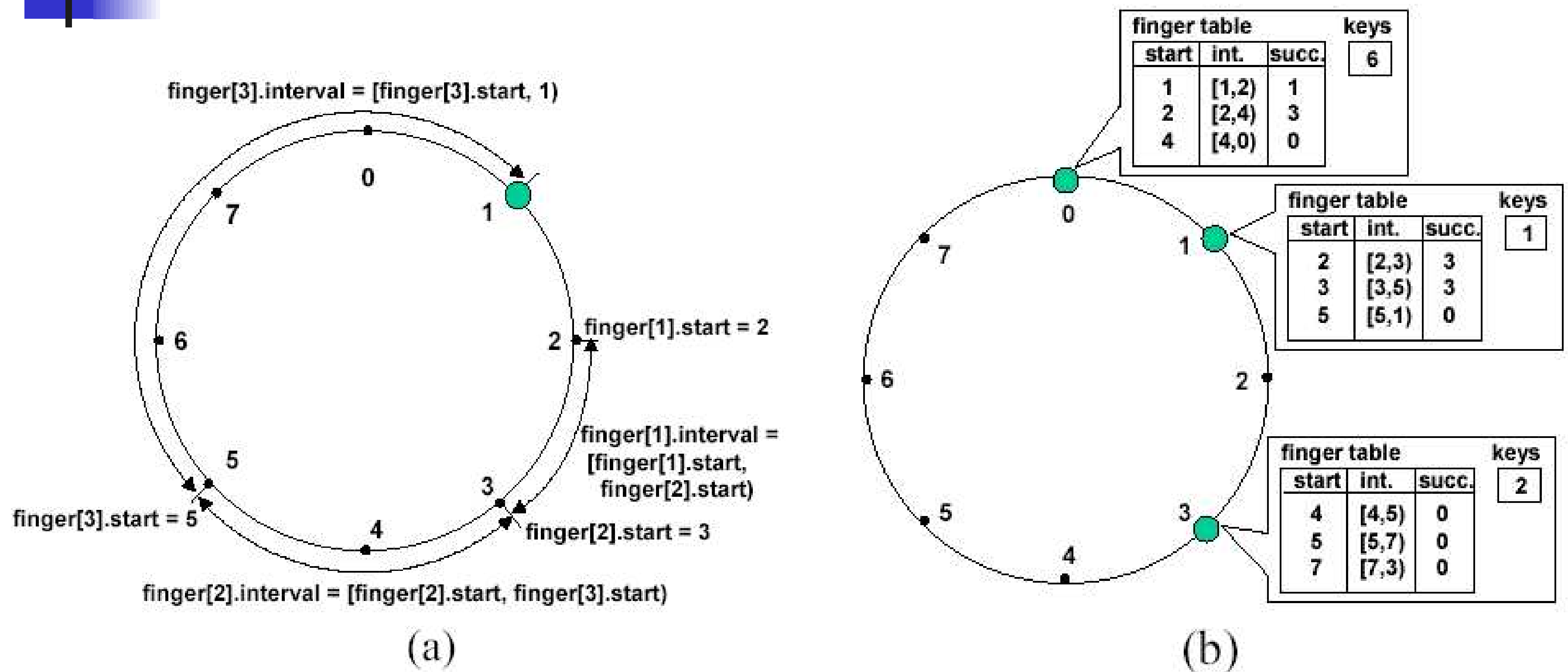

Figure 3: (a) The finger intervals associated with node 1. (b) Finger tables and key locations for a net with nodes 0, 1, and 3, and keys 1, 2, and 6.

Scalable key Location (cont.)

- THEOREM 2. With high probability (or under standard hardness assumption), the number of nodes that must be contacted find a successor in an *N*-node network is $O(\log N)$.

Node Joins

- Each node in Chord maintains a *predecessor pointer*, and can be used work counterclockwise around the identifier circle.
- When a node *n* joins the network:
  1. Initialize the predecessor and fingers of node *n*.
  2. Update the fingers and predecessors of existing nodes to reflect the addition of *n*.
  3. Notify the higher layer software so that it can transfer state (e.g. values) associated with keys that node *n* is now responsible for.

Node Joins (cont.)

```
#define successor finger[1].node

// node n joins the network;
// n' is an arbitrary node in the network
n.join(n')
  if (n')
    init_finger_table(n');
    update_others();
    // move keys in (predecessor, n] from successor
  else // n is the only node in the network
    for i = 1 to m
      finger[i].node = n;
    predecessor = n;

// initialize finger table of local node;
// n' is an arbitrary node already in the network
n.init_finger_table(n')
  finger[1].node = n'.find_successor(finger[1].start);
  predecessor = successor.predecessor;
  successor.predecessor = n;
  for i = 1 to m - 1
    if (finger[i + 1].start ∈ [n, finger[i].node))
      finger[i + 1].node = finger[i].node;
    else
      finger[i + 1].node =
          n'.find_successor(finger[i + 1].start);

// update all nodes whose finger
// tables should refer to n
n.update_others()
  for i = 1 to m
    // find last node p whose i^th finger might be n
    p = find_predecessor(n - 2^(i-1));
    p.update_finger_table(n, i);

// if s is i^th finger of n, update n's finger table with s
n.update_finger_table(s, i)
  if (s ∈ [n, finger[i].node))
    finger[i].node = s;
    p = predecessor; // get first node preceding n
    p.update_finger_table(s, i);
```

Figure 6: Pseudo code for the node join operation.

Node Joins (cont.)

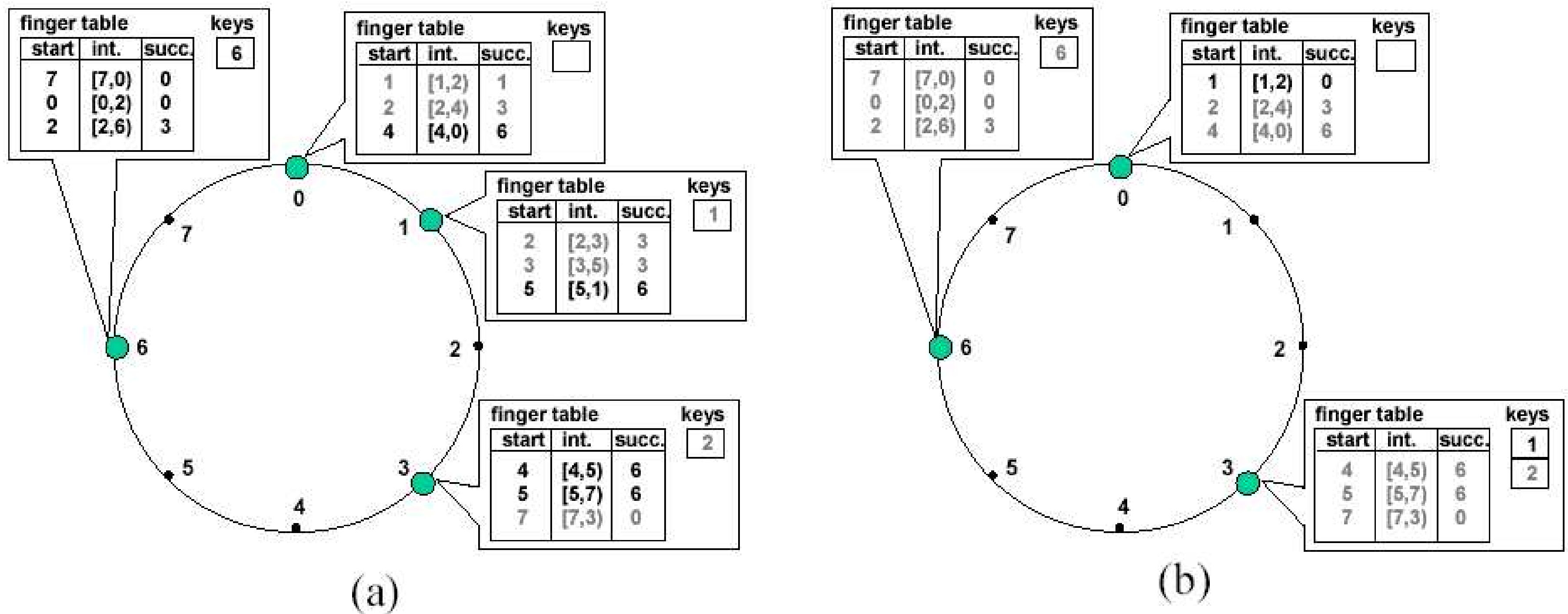

Figure 5: (a) Finger tables and key locations after node 6 joins. (b) Finger table and key locations after node 1 leaves. Changed entries are shown in black , and unchanged in gray.

Concurrent Operations : Stabilization

- A basic "stabilization" protocol is used to keep nodes' successor pointers up to date, which is sufficient to guarantee correctness of lookups.
- If joining nodes have affected some region of the Chord ring, a lookup that occurs before stabilization has finished can exhibit one of three behaviors.
  - All the finger table entries involved in the lookup are reasonably current, and the lookup finds the correct successor in O (log N) steps.
  - Successor pointers are correct, but fingers are inaccurate.
  - The nodes in the affected region have incorrect successor pointers, or keys may not yet have migrated to newly joined nodes, and the lookup may fail.
- These cases could be detected and repaired by periodic sampling of the ring topolgy.

Stabilization

```
n.join(n')
    predecessor = nil;
    successor = n'.find_successor(n);

// periodically verify n's immediate successor,
// and tell the successor about n.
n.stabilize()
    x = successor.predecessor;
    if (x ∈ (n, successor))
        successor = x;
    successor.notify(n);

// n' thinks it might be our predecessor.
n.notify(n')
    if (predecessor is nil or n' ∈ (predecessor, n))
        predecessor = n';

// periodically refresh finger table entries.
n.fix_fingers()
    i = random index > 1 into finger[];
    finger[i].node = find_successor(finger[i].start);
```

Figure 7: Pseudo code for stabilization.

Failures and Replication

- When a node *n* fails, nodes whose finger tables include *n* must find *n*'s successor.
- Each Chord node maintains a "successor-list" of its r nearest successor on the Chord ring.
- THEOREM 7. 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 8. then expected time to execute find_successor in the failed network is $O(\log N)$
- A node's *r* successors all fail with probability $2^{-r} = 1/N$.
- A typical application using Chord might store replicas of the data associated with key at the *k* nodes succeeding the key.

Simulation: Load Balance

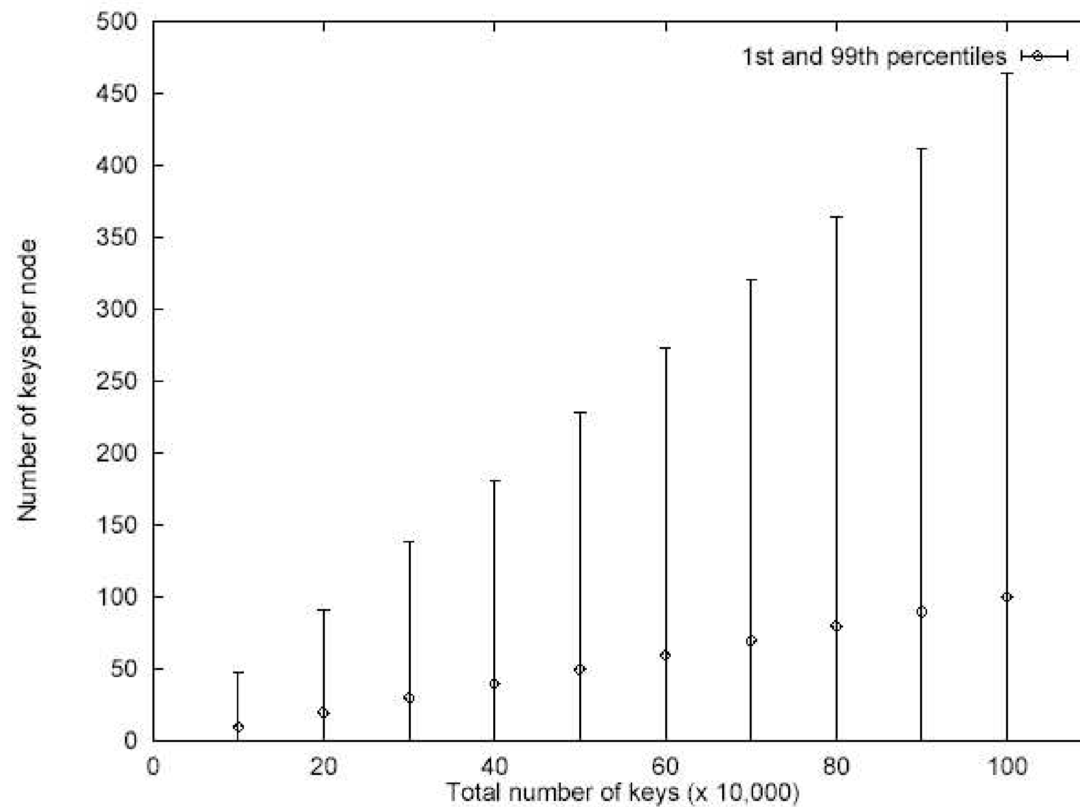

Figure 8: (a) The mean and 1st and 99th percentiles of the number of keys stored per node in a 10^4 node network.

Load Balance (cont.)

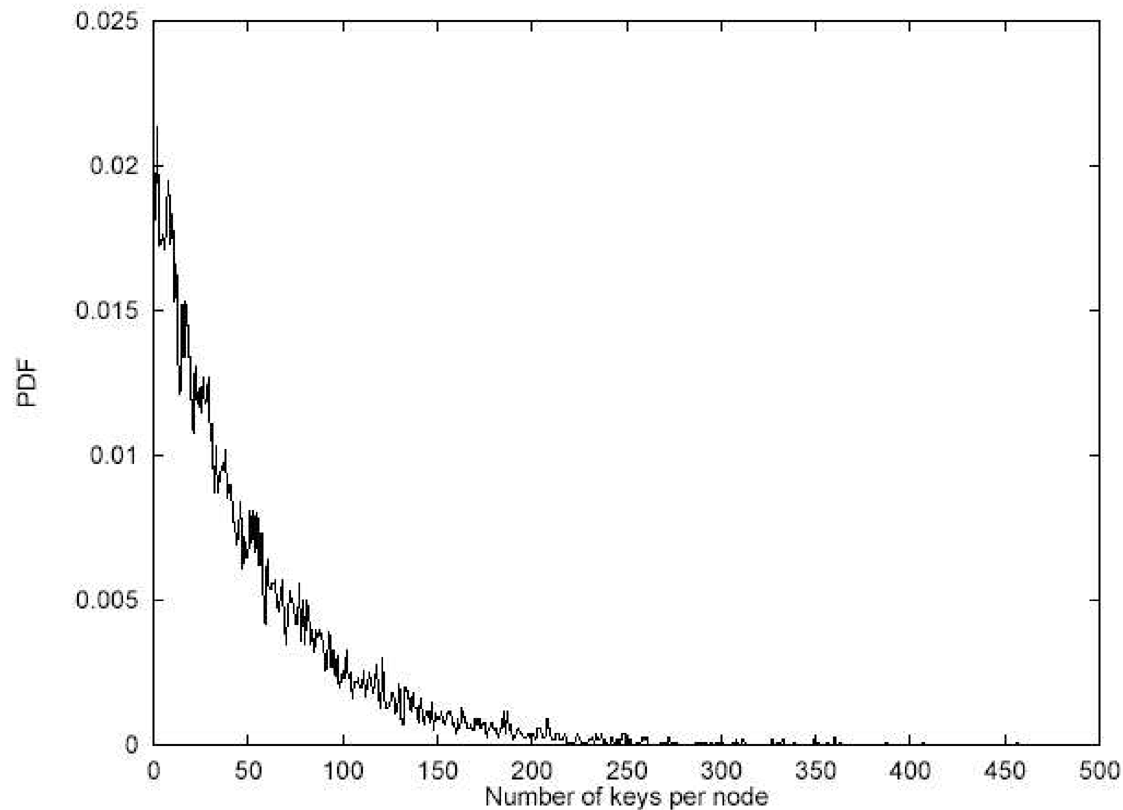

Figure 8: (b) The probability density function (PDF) of the number of keys per node. The total number of keys is 5 x 10^5.

Load Balance (cont.)

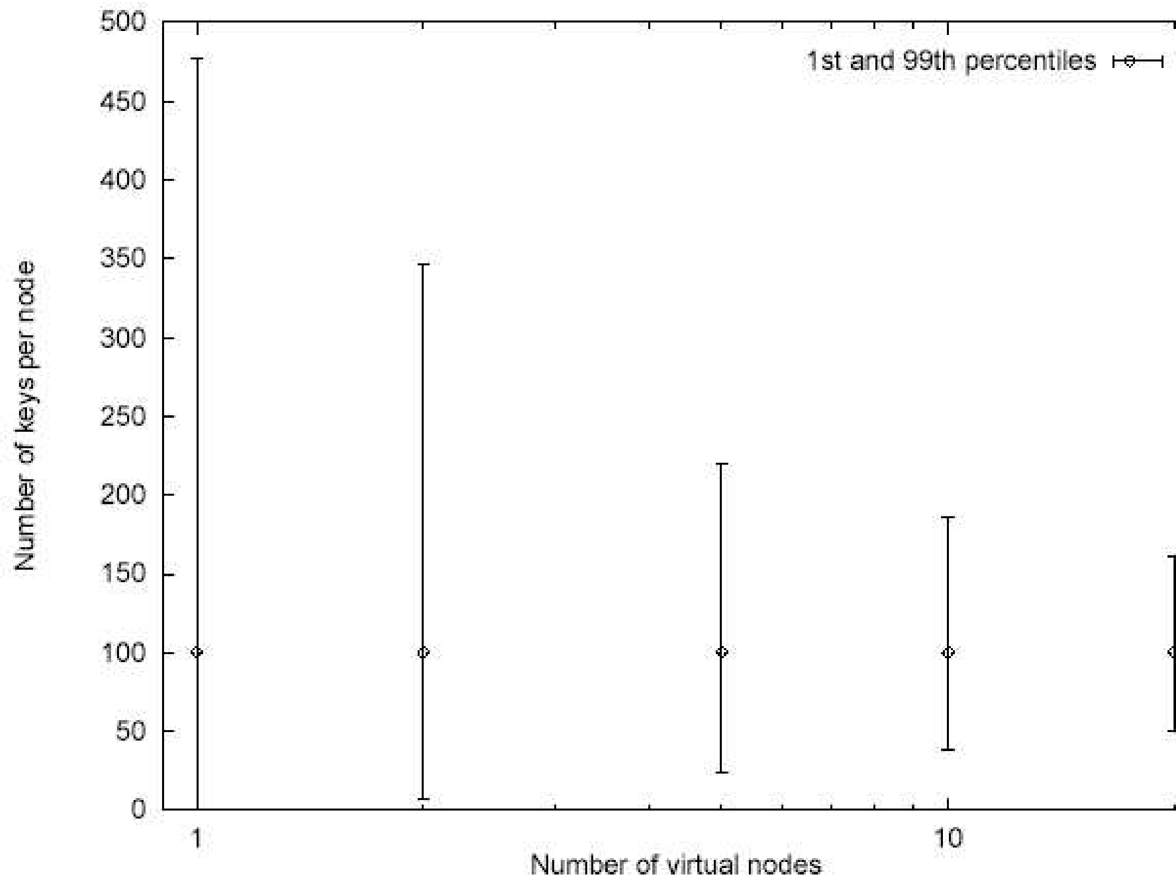

Figure 9: The 1st and the 99th percentiles of the number of keys per node as a function of virtual nodes mapped to a real node. The network has 10^4 real nodes and stores 10^6 keys.

Path Length

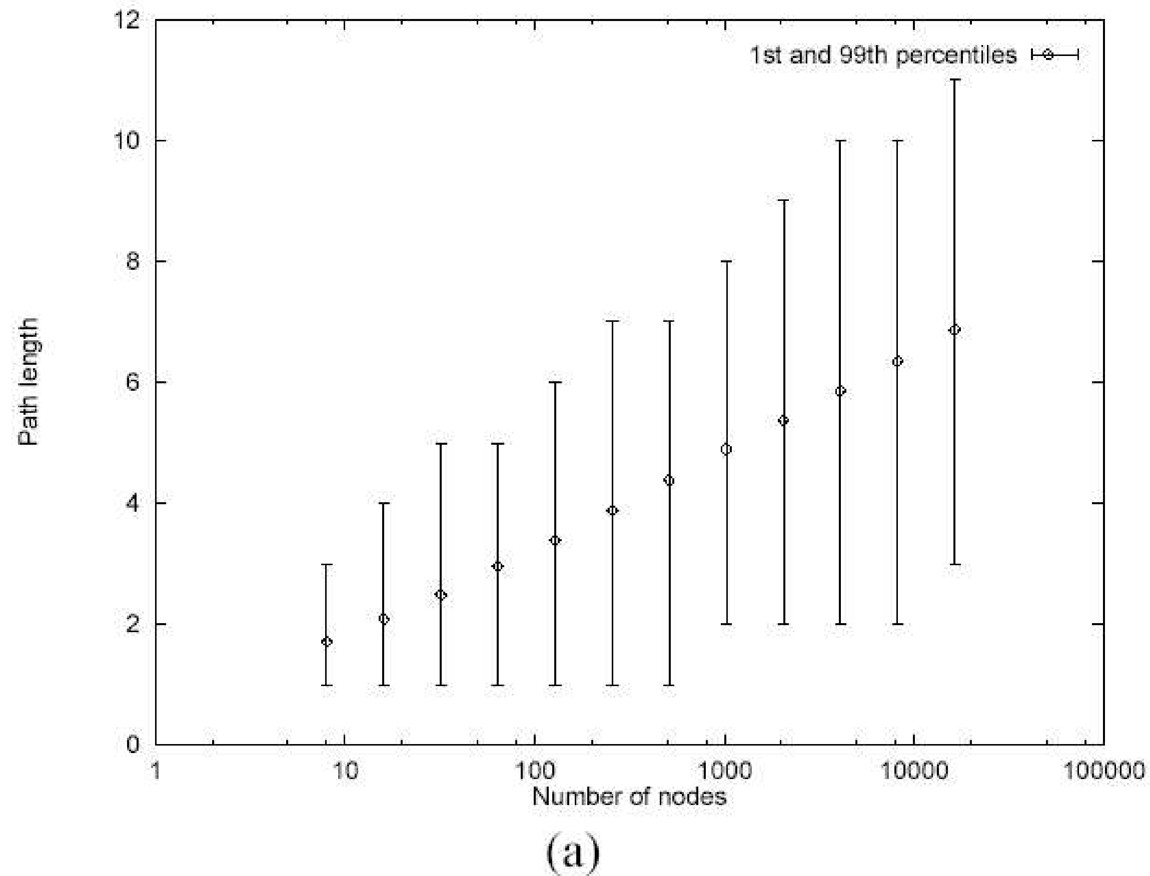

Figure 10: (a) The path length as a function of network size.

Path Length (cont.)

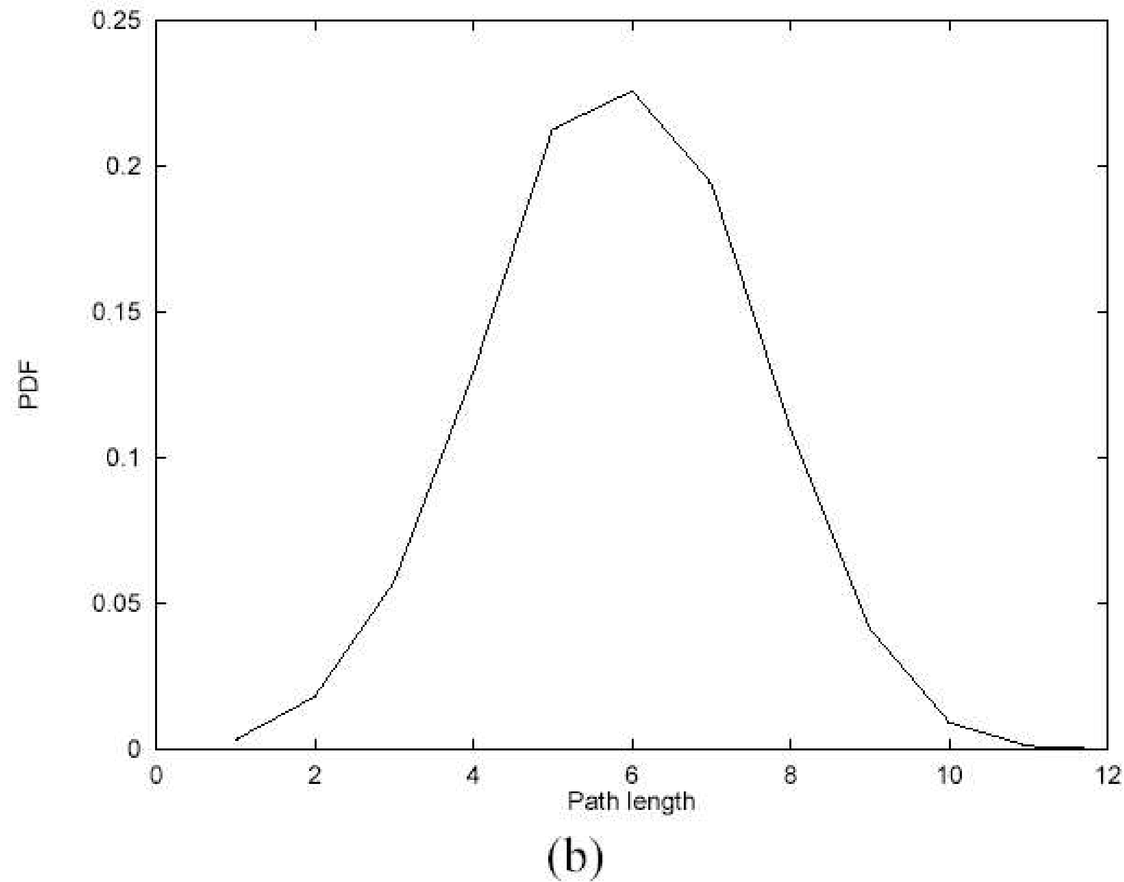

Figure 10: (b) The PDF of the path length in the case of a 2^{12} node network.

Simultaneous Node Failures

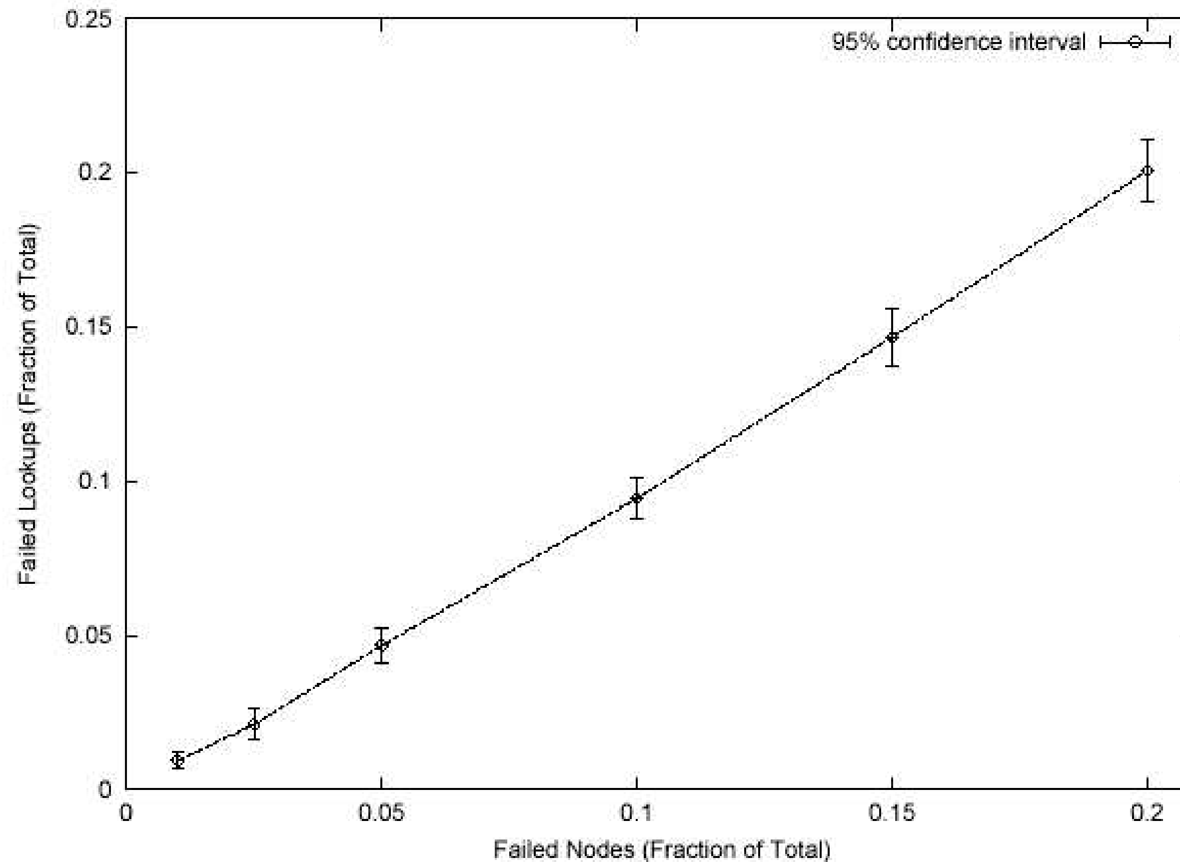

Figure 11: The fraction of lookups that fail as a function of the fraction of nodes that fail.

Lookups During Stabilization

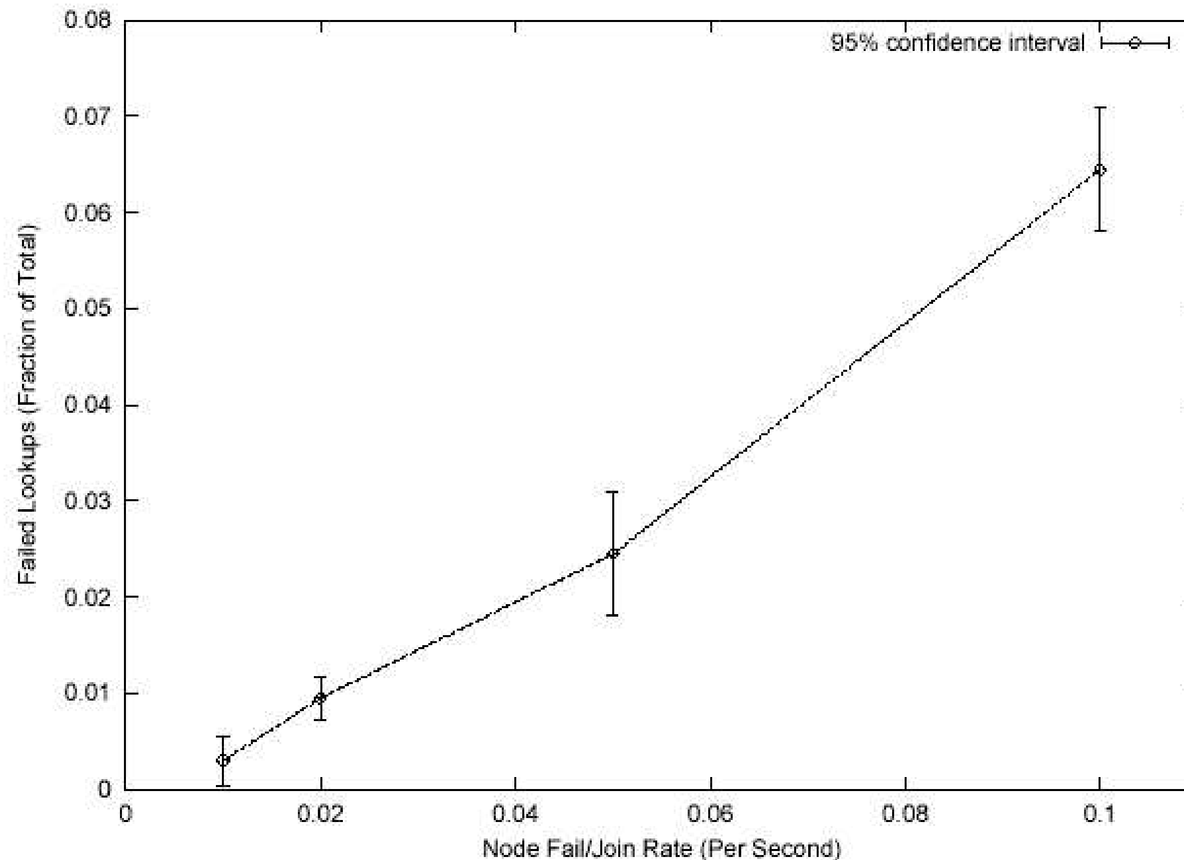

Figure 12: The fraction of lookups that fail as a function of the rate (over time) at which nodes fail and join. Only failures caused by Chord state inconsistency are included, not failures due to lost keys.

Experimental Results

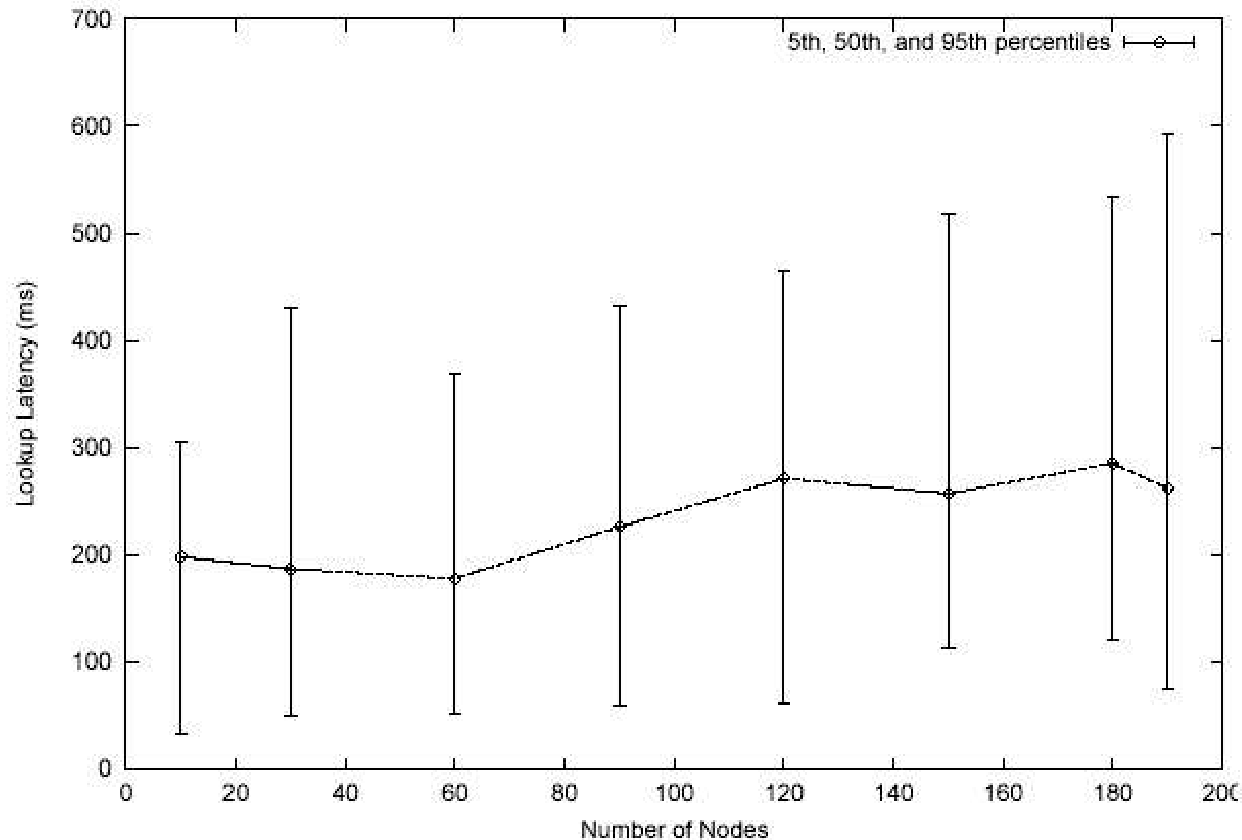

Figure 13: Lookup latency on the Internet prototype, as a function of the total number of nodes. Each of the ten physical sites runs multiple independent copies of the Chord node software.

Future Work

- Chord currently has no specific mechanism to heal partitioned rings; such rings could appear locally consistent to the stabilization procedure.
- Instead of placing its fingers at distances that are all powers of 2, Chord could easily be changed to place its fingers at distances that are all integer powers of $1 + 1/d$.

Conclusion

- The Chord protocol solves that applications need not to determine the node that stores a data item.
- Given a key, it determines the node responsible for storing the key's value, and does so efficiently.
- In the steady state, in an N-node network, each node maintains routing information for only about $O(\log N)$ other nodes,
- and resolves all lookups via $O(\log N)$ messages to other nodes.
- Updates to the routing information for nodes leaving and joining require only $O(\log^2 N)$ messages.
- Chord scales well with the number of nodes, and answers most lookups correctly even during recovery.