CHORD: A Scalable Peer-to-Peer Lookup Service for Internet Applications

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

MIT Laboratory for Computer Science

Adopted for Presented in 527-07 UBC

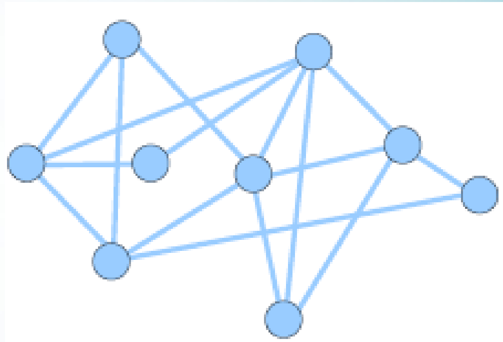
Outline

- Background
- Chord
  - Naming
  - Searching
  - Storing data
  - Node Join
  - Node Leave
  - Fault Tolerance
  - Load Balancing
  - Simulation and Experimental Results
- Summary

P2P Background

What is P2P?

- An application-level Internet on top of the Internet
- Every participating node acts as both a client and a server (“servent”)
- •Every node “pays” its participation by providing access to (some of) its resources
- Properties:
  - No central coordination
  - No central database
  - No peer has a global view of the system. Global behavior emerges from local interactions
  - All existing data and services are accessible from any peer
  - Dynamic
  - Peers and connections are unreliable



P2P applications

- File sharing & storage Systems
- Peer information retrieval and P2P web search
- Peer data management, Peer query reformulation

P2P Network Requirements

- Efficient data location & Routing (Scalability)
- Adaptable to changes (data and query)
- Self-Organizing, Ad hoc participation
- Robust and fault tolerant

Question: How to find the data?

- An efficient index mechanism: DHT
  - Hash Table
    - data structure that maps “keys” to “buckets”
    - essential building block in software systems
  - Distributed Hash Table (DHT)
    - similar, but spread across the Internet
    - Rely on the hashing to achieve load balancing
  - Interface--Every DHT node supports operation:
    - lookup(key)--Given *key* as input; route messages toward node holding *key*
    - insert(key, value)

DHT Design Goals

- **Low diameter:** Theoretical search bound
- **Low degree:** Limited number of neighbours
- **Local routing decision**: Greedy. Nodes incrementally calculate a path to the target
- **Load balance**: distributed hash function, spreading keys evenly over nodes
- **Robustness**: Operational even in partial failure
- **Availability**: Can automatically adjusts its internal tables to ensure that the node responsible for a key can always be found

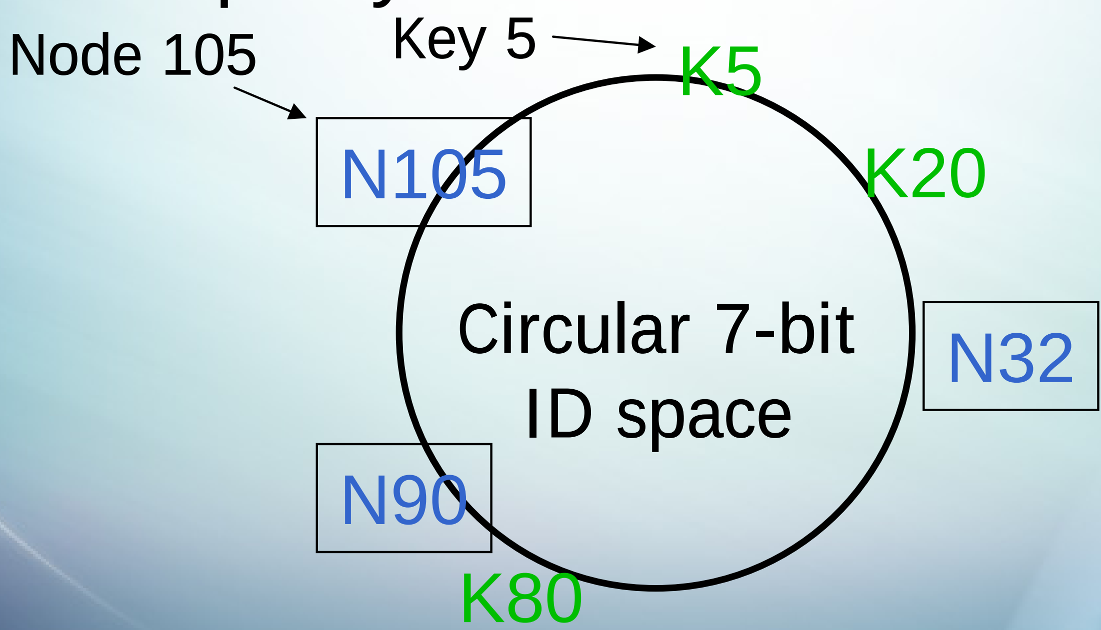
An Example DHT: Chord

Consistent Hashing

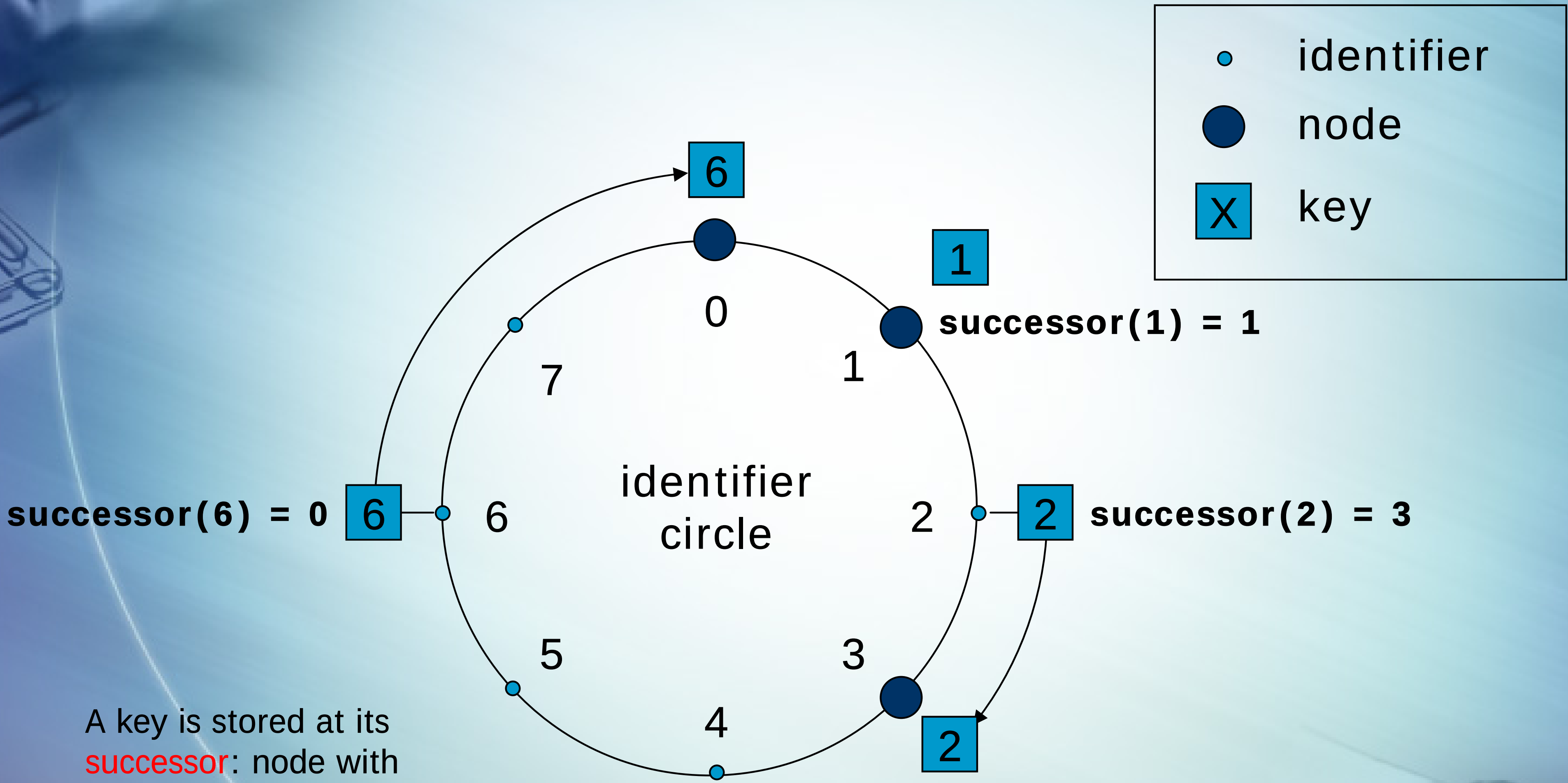
- A scheme that provides hash table functionality in a way that the addition or removal of one bucket does not significantly change the mapping of keys to buckets
- A consistent hashing function:
  - Map both the buckets and keys into a range. The consistent hash of a key is defined as the bucket whose image is closest* to the key's.
  - When buckets join or leave, only nearby buckets are affected
- CHORD is a distributed consistent hashing table.

Chord Naming

- Key identifier = SHA-1(key)
- Node identifier = SHA-1(IP address)
- Both are uniformly distributed
- Both exist in the same ID space
- How to map key IDs to node IDs?



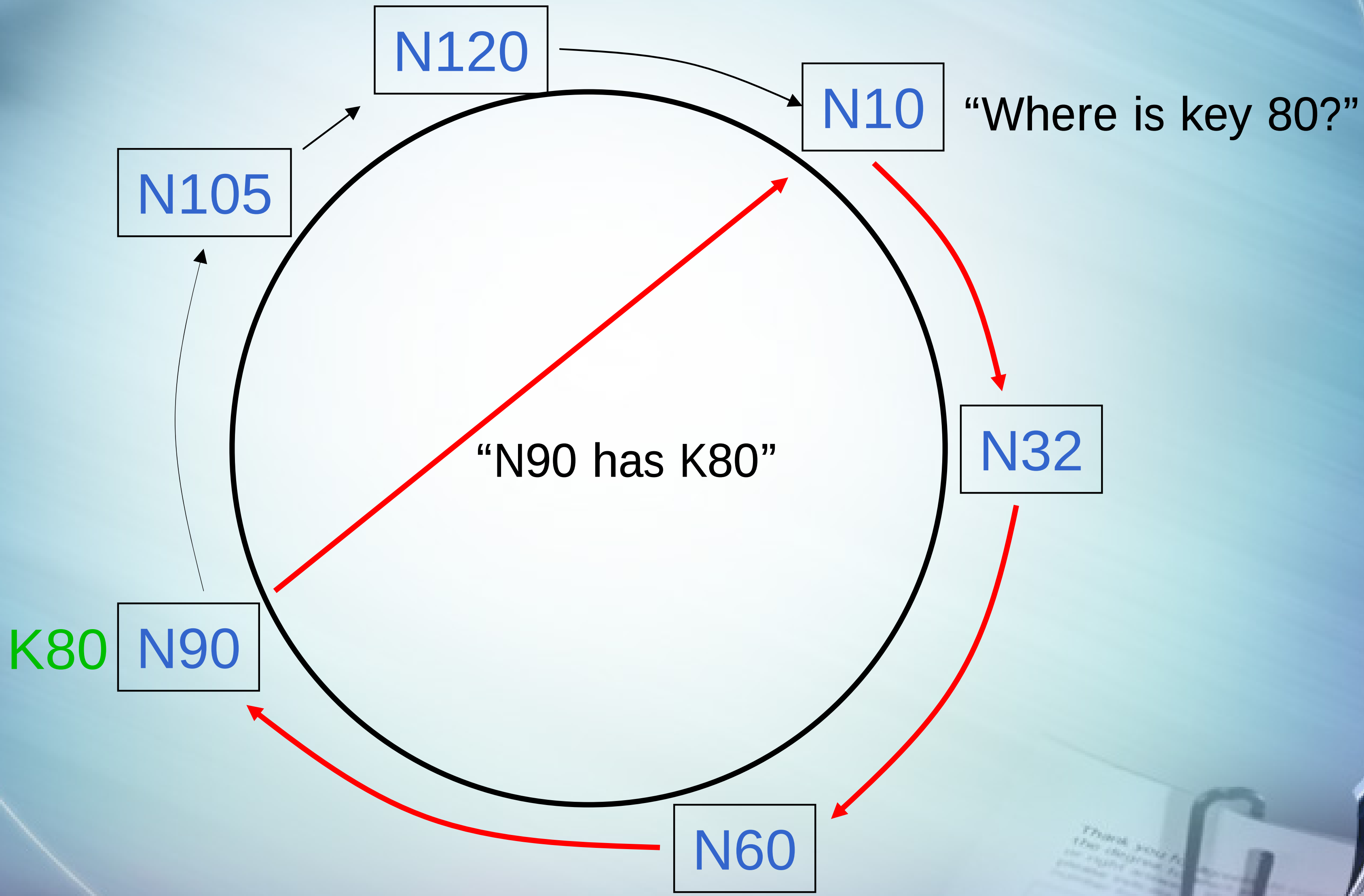
Successor Nodes



A key is stored at its successor: node with the same id or next higher ID

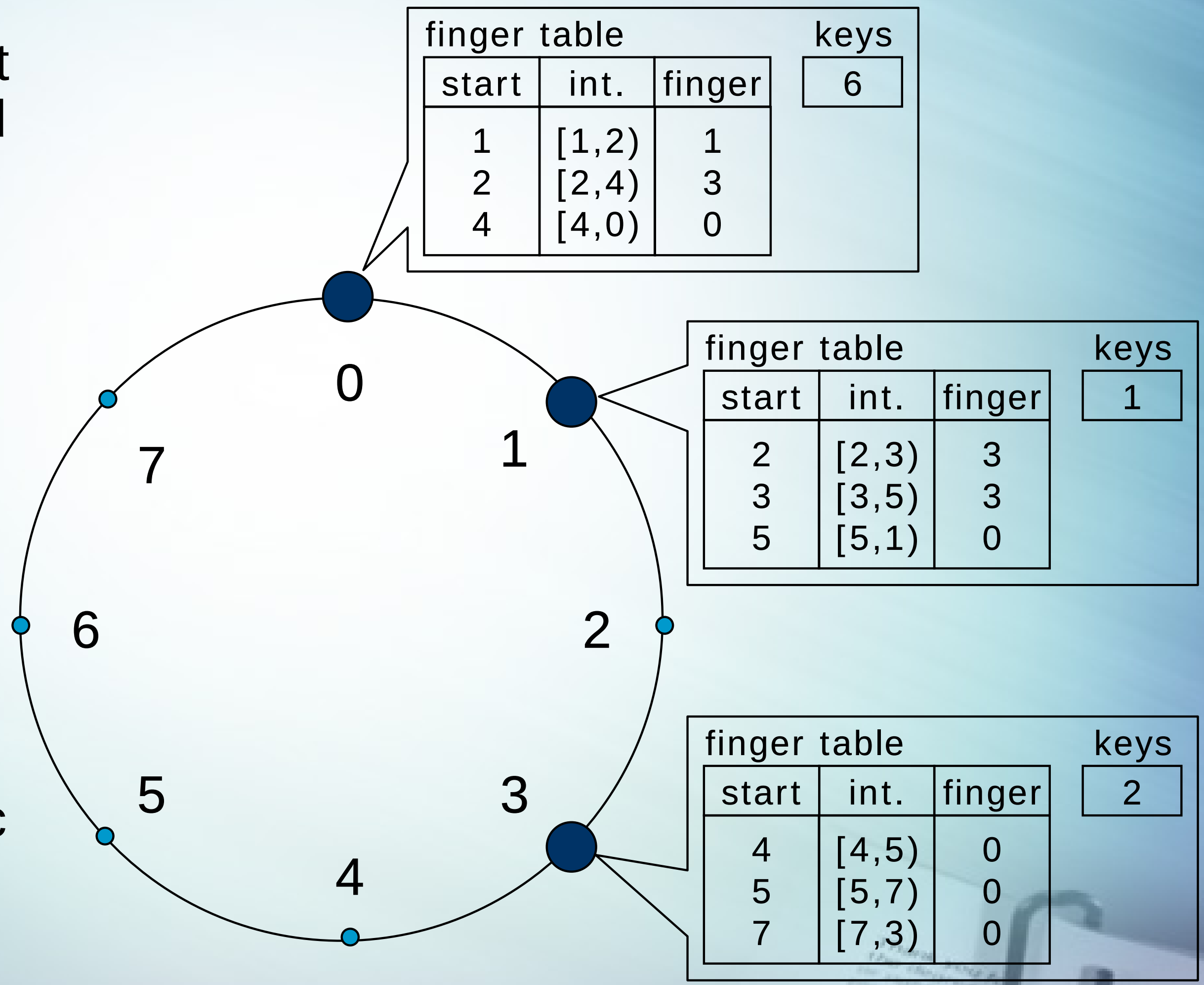
Note: successor(key) is different from successor(node)!

CHORD Search: Basic lookup



CHORD Search: Acceleration of Lookups

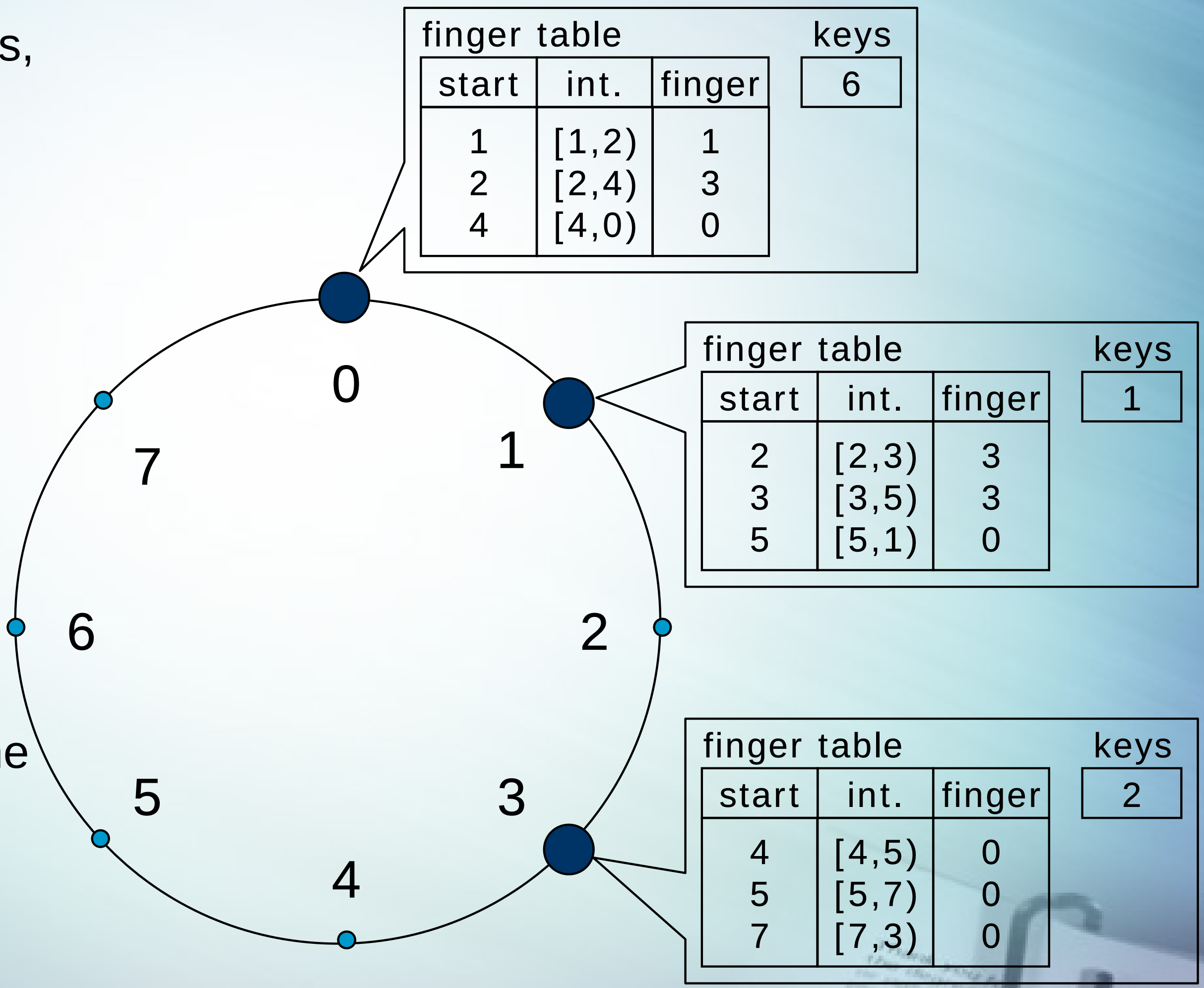
- Each node maintains
  - a routing table with (at most) *m* entries called the **finger table**
    - **Start**: $(n + 2^{i-1}) \bmod 2^m$
    - **Interval**: [finger[i].start, finger[i+1].start)
    - **Successive node (finger)**: First node >= finger[i].start
  - A successor: the next node on the identifier ring, i.e. finger[1].succ
  - A predecessor: the previous node on the identifier ring.



CHORD Search: Acceleration of Lookups

- Each node stores information about only a small number of other nodes, and knows more about nodes *closely* following it than about nodes *farther away*
- A node's finger table generally does not contain enough information to determine the successor of an arbitrary key *k.*

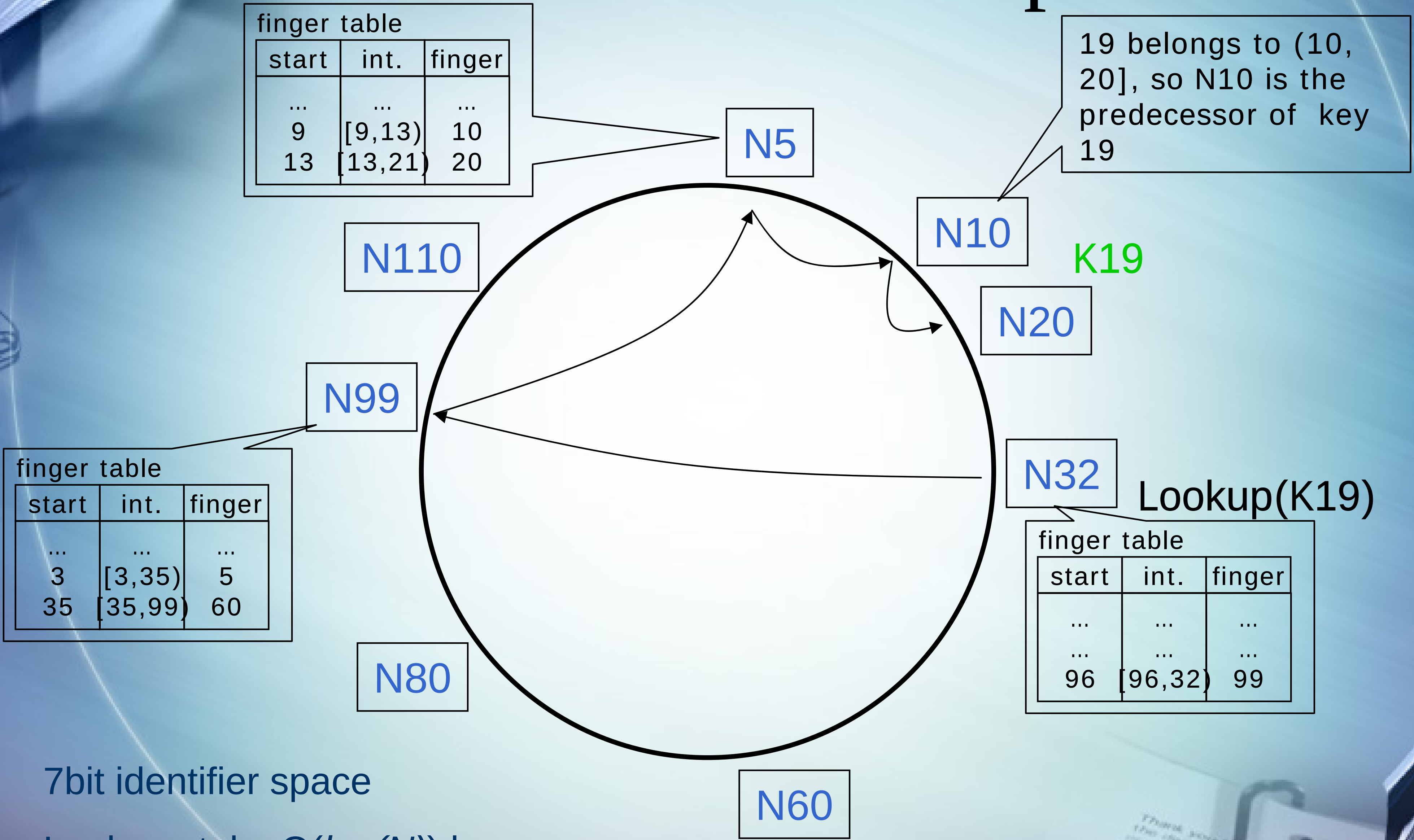
  E.g. Node 3 looks up key 1
- Repetitive queries to nodes that immediately precede the given key will lead to the key's successor eventually



The algorithm

```
// 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;
```

CHORD Search: An example



7bit identifier space

Lookups take O(*log(N)*) hops

CHORD: Storing Data

- Search the node responsible for holding the key
- Insert the key into that node

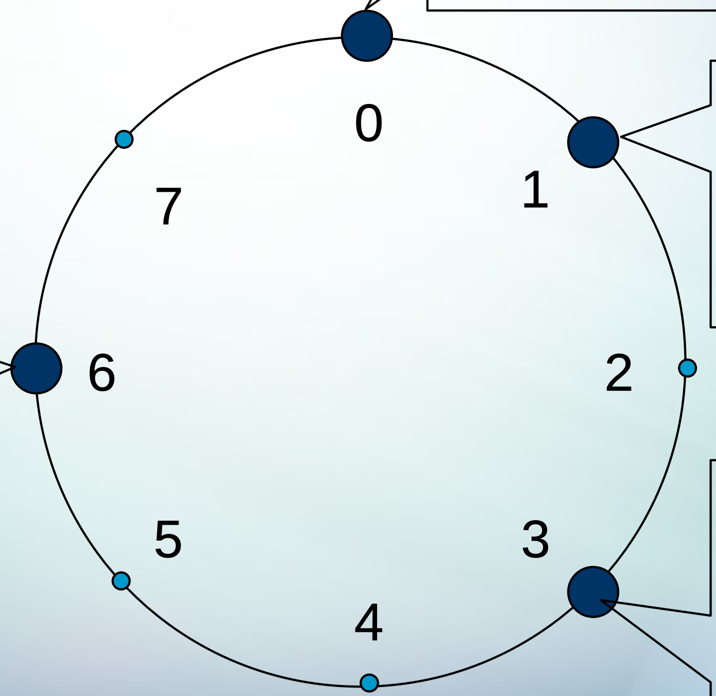
CHORD Node Joins

Node 6 wants to join

1. Initialize fingers and predecessor: $O(\log^2 N)$
2. Transferring keys: $O(1)$
3. Updating fingers of existing nodes: $O(\log^2 N)$

Total Cost: $O(\log^2 N)$

Each node keeps a predecessor pointer



Node 0 — finger table (keys: 6)

start	int.	finger
1	[1,2)	1
2	[2,4)	3
4	[4,0)	**6**

Node 1 — finger table (keys: 1)

start	int.	finger
2	[2,3)	3
3	[3,5)	3
5	[5,1)	**6**

Node 3 — finger table (keys: 2)

start	int.	finger
4	[4,5)	**6**
5	[5,7)	**6**
7	[7,3)	0

Node 6 — finger table (keys:)

start	int.	finger
7	[7,0)	0
0	[0,2)	0
2	[2,6)	3

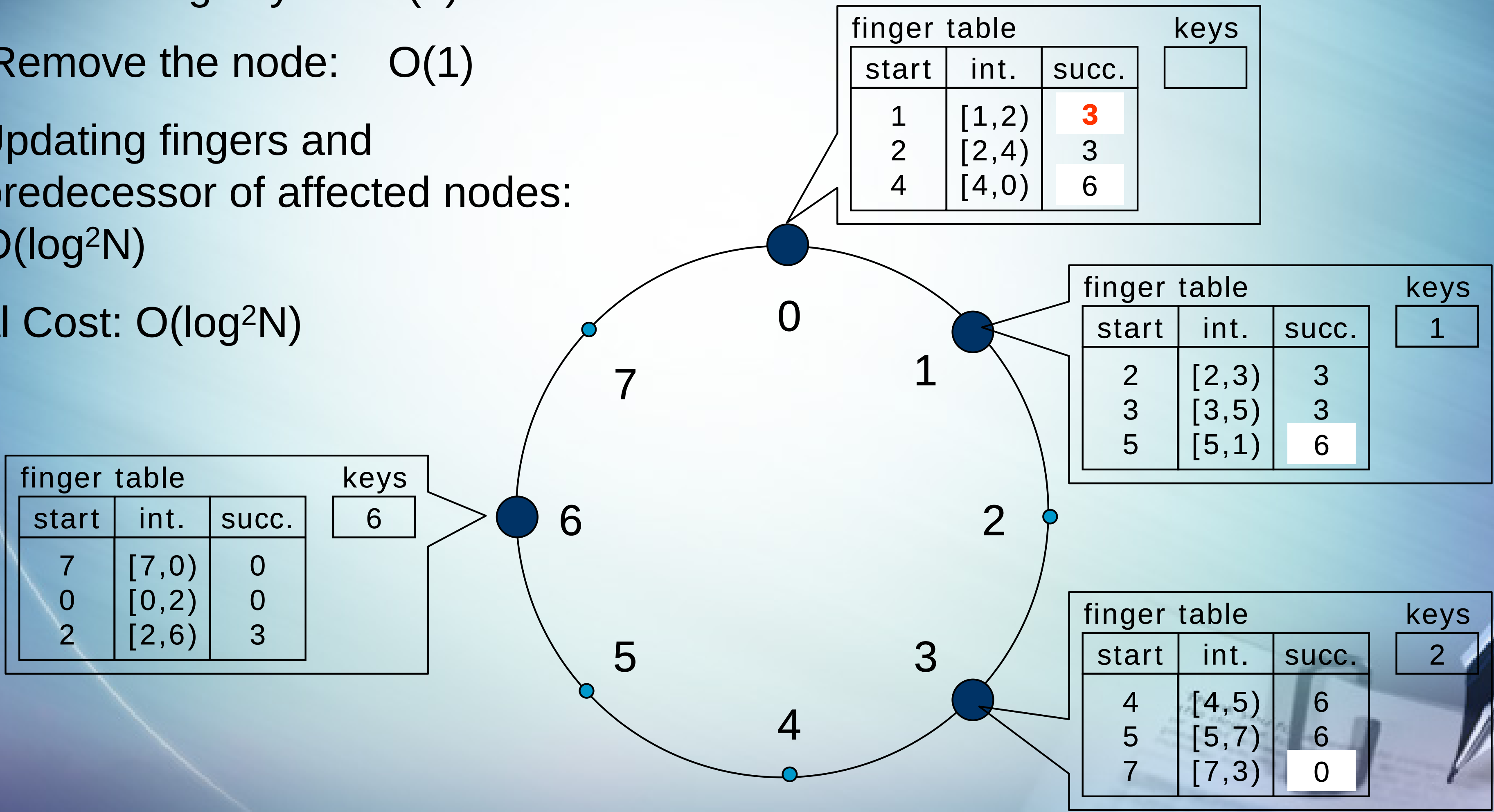
```
// update all nodes whose finger
// tables should refer to n
n.update_others()
  for i = 1 to m
    // find last node p whose i^th finger might be n
    p = find_predecessor(n - 2^(i-1));
    p.update_finger_table(n, i);

// 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);
```

CHORD Node Departures

Node 1 wants to leave

1. Transferring keys: O(1)
2. Remove the node: O(1)
3. Updating fingers and predecessor of affected nodes: $O(\log^2 N)$

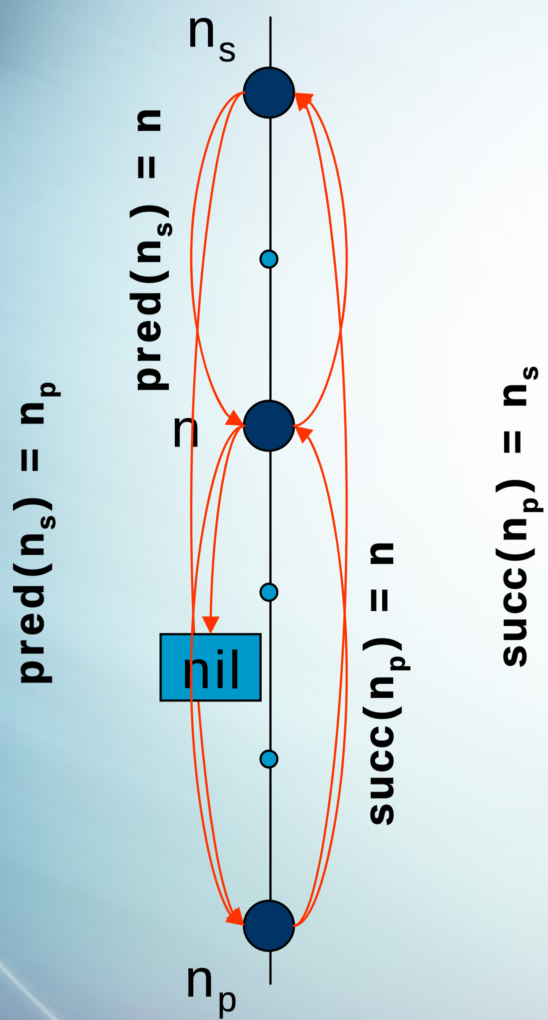
Total Cost: $O(\log^2 N)$



CHORD: Fault Tolerance

- Basic “**stabilization**” protocol is used to keep nodes’ successor pointers up to date, which is sufficient to guarantee correctness of lookups
- Every node runs *stabilize* periodically to find newly joined nodes
- Also fix the fingers：find_successor(finger[i].start)

CHORD: Fault Tolerance -- Stabilization after Join



- **n joins**
  - predecessor = nil
  - n acquires n_s as successor via some n'
  - n notifies n_s being the new predecessor
  - n_s acquires n as its predecessor
- **n_p runs stabilize**
  - n_p asks n_s for its predecessor (now n)
  - n_p acquires n as its successor
  - n_p notifies n
  - n will acquire n_p as its predecessor
- **all predecessor and successor pointers are now correct**

CHORD: Fault Tolerance – Failure Recovery

- Key step in failure recovery is maintaining correct successor pointers (The worst case as in the simple lookup)
- To help achieve this, each node maintains a *successor-list* of its *r* nearest successors on the ring
- If node *n* notices that its successor has failed, it replaces it with the first live entry in the list
- *stabilize* will correct finger table entries and successor-list entries pointing to failed node
- Performance is sensitive to the frequency of node joins and leaves versus the frequency at which the stabilization protocol is invoked

CHORD: Fault Tolerance – Replication

- Chord can store replicas of the data associated with a key at the k nodes succeeding the key
- (+) Increase data availability
- (–) larger size of the <key, value> database

CHORD: Load Balancing

- Even hashing is used, the number of keys stored on each node may vary a lot.
- This is because the node identifiers do not uniformly cover the entire identifier space.
- Solution:
  - Associate keys with a number of virtual nodes
  - Map multiple (e.g. logN as suggested in the consistent hashing paper) virtual nodes to each real node
  - Doesn’t affect worst case performance: O(log(NlogN)) = O(logN)

Experimental Results—Load balance

Number of Nodes: N= 10^4
Number of Keys: K= 10^5 ~ 10^6

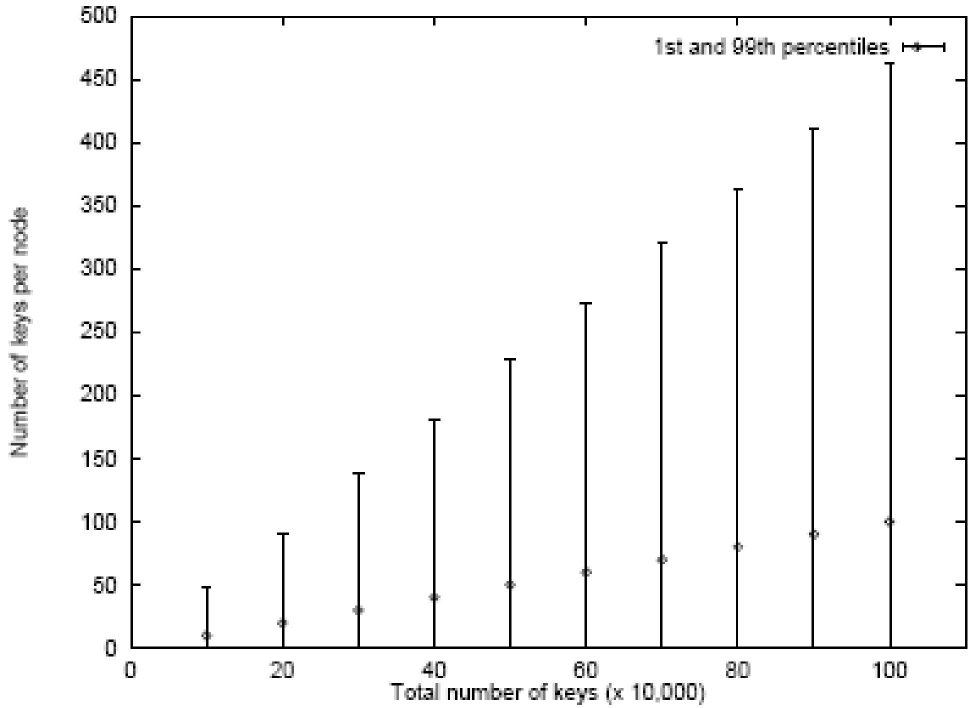


Figure 1: The mean and 1st and 99th percentiles of the number of keys stored per node in a 10^4 node network.

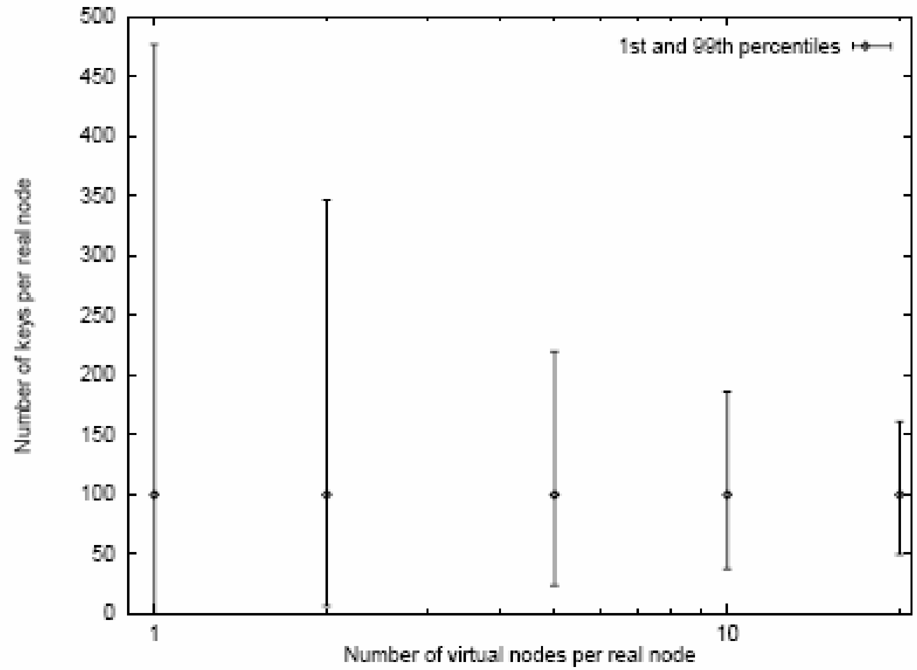
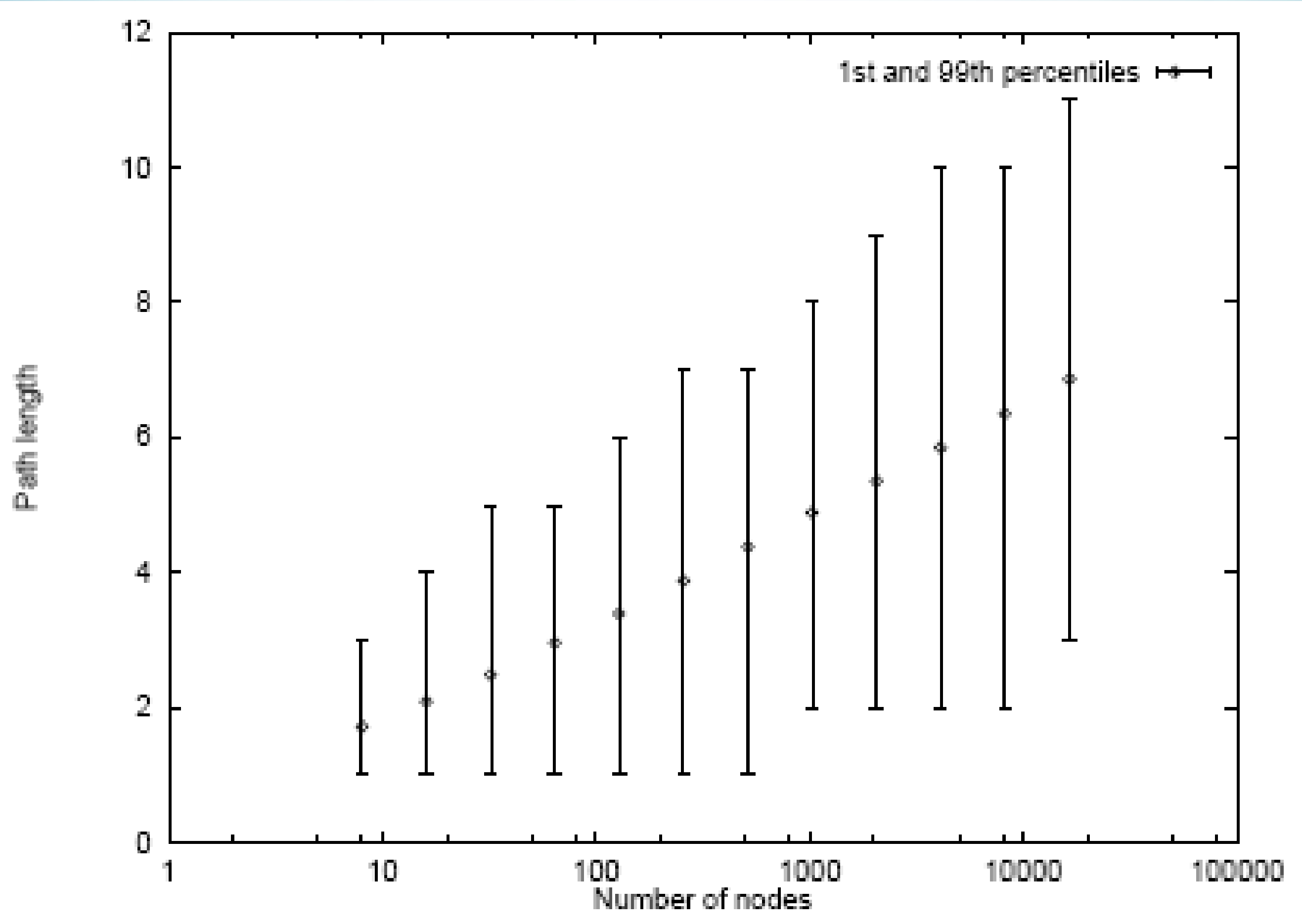


Figure 2: The 1st and the 99th percentiles of the number of keys per node as a function of virtual nodes mapped to a real node.

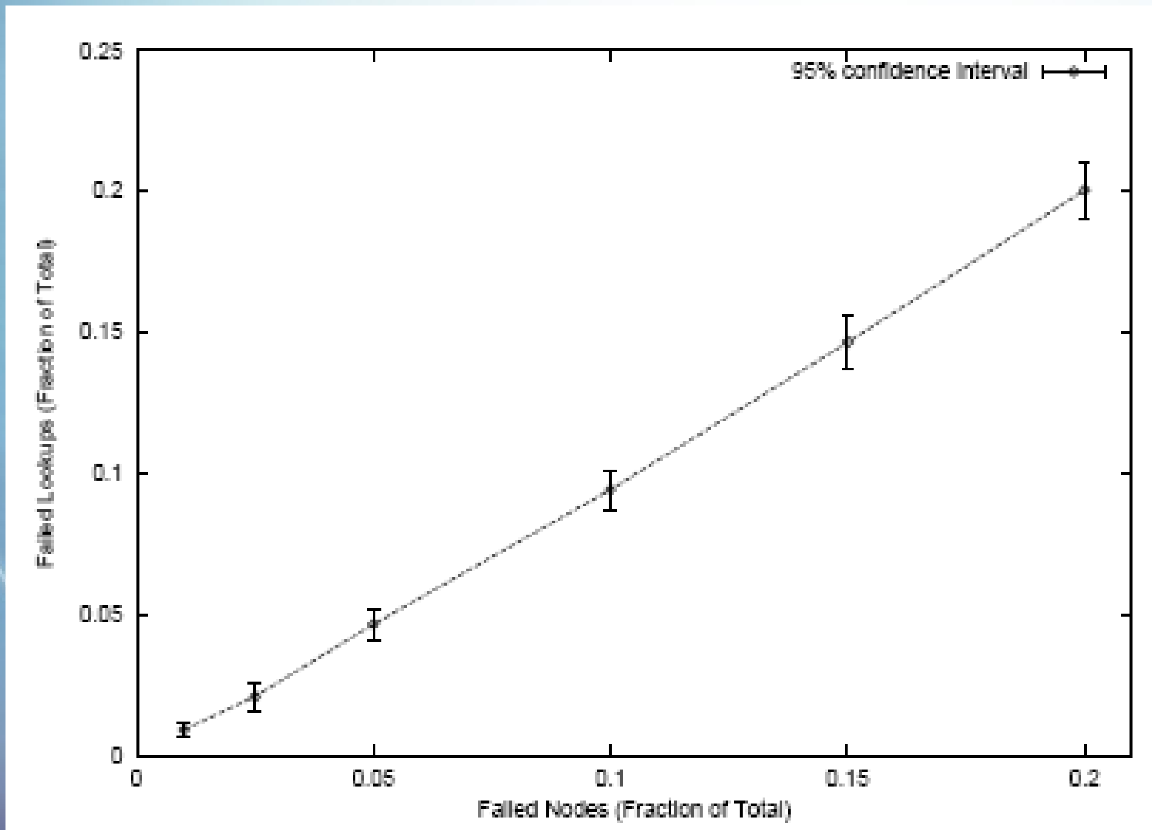
Experimental Results– Path Length

Number of Nodes: N= 2^k
Number of Keys: K= 100 * 2^k
K from 3 to 14



Experimental Results– Simultaneous Node Failures

Number of Nodes: $N= 10^4$
Number of Keys: $K= 10^5 \sim 10^6$



CHORD Summaries

- Scalability
  - Search O(Log n) w.h.p.
  - Update requires search, thus O(Log n) w.h.p.
  - Construction: O(Log^2 n) if a new node joins
- Robustness
  - Replication might be used by storing replicas at successor nodes
- Global knowledge
  - Mapping of IP addresses and data keys to key common key space
- Autonomy
  - Storage and routing: none
  - Nodes have by virtue of their IP address a specific role
- Search types
  - Only equality

Q & A