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

Motivation

- Advent of peer-to-peer (P2P) systems where resources are distributed and all nodes are equally important to provide a single application
- First popular P2P systems include Napster, Freenet, Gnutella, BitTorrent
- Core operation in P2P systems is efficient location of data items

Centralized P2P Systems

- Napster was one of the first P2P file sharing application with an emphasis on digital audio file distribution.
- Napster used a central index server that had all the information about which node contains which data

Any issues?

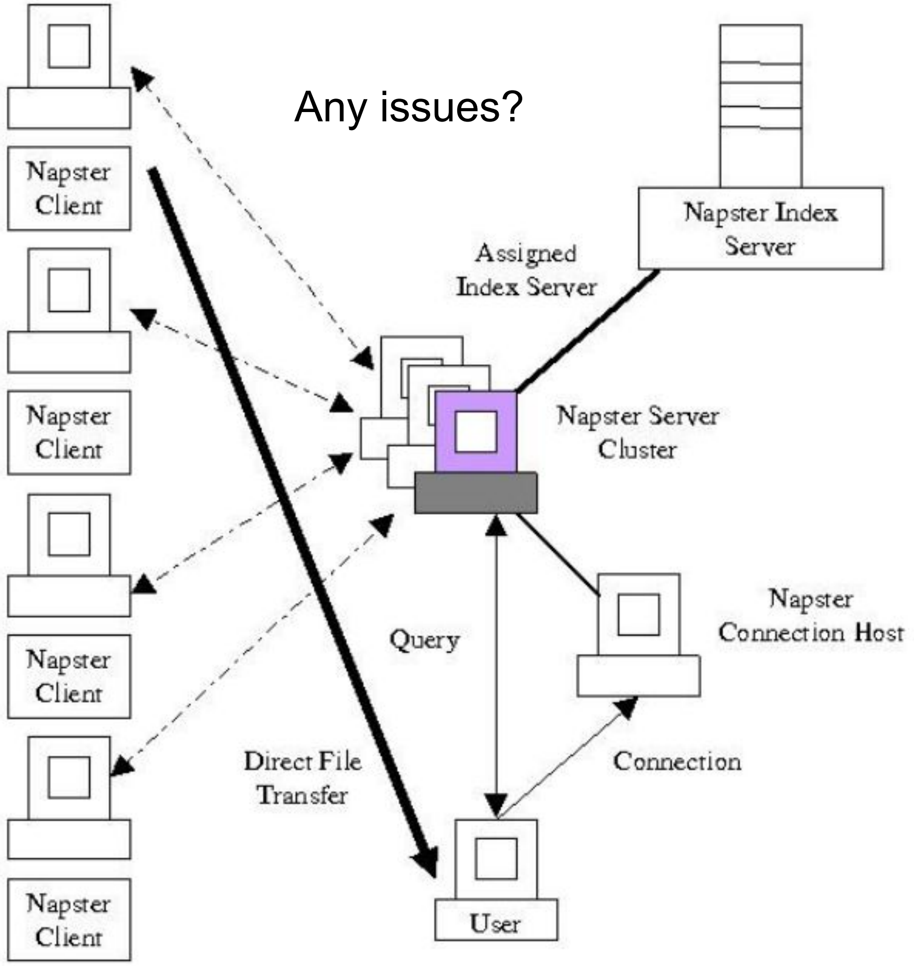
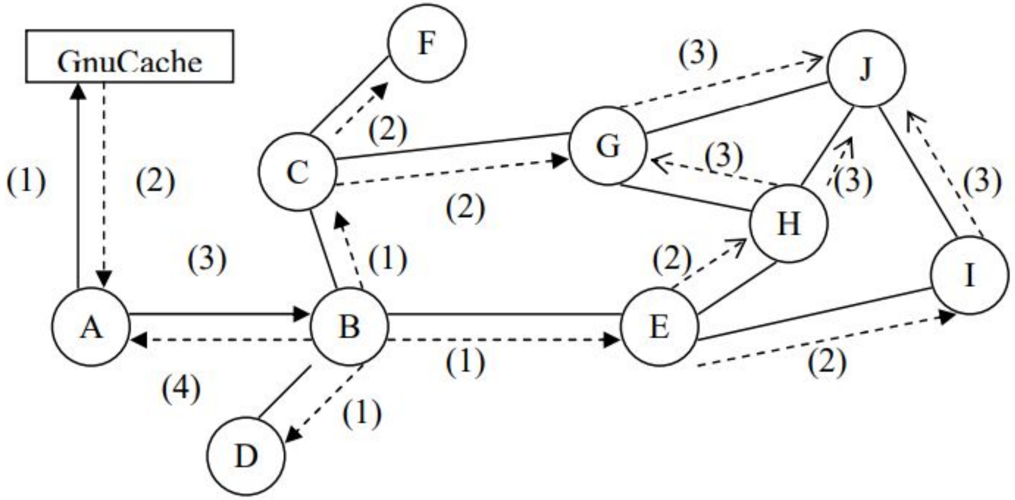


Figure 8: Illustration of the Napster Architecture

More Distributed P2P system

- Gnutella is a P2P system with a flat topology composed of peers(servent) and a Host(GnuCache) that contains the list of current servent
- Query flooding model



(1) User A connects to the GnuCache to get the list of available servents already connected in the network
(2) GnuCache sends back the list to the user A
(3) User A sends the request message GNUTELLA CONNECT to the user B
(4) User B replies with the GNUTELLA OK message granting user A to join the network

Figure 17: Example of Gnutella Protocol

Distributed P2P Systems

- Freenet is fully decentralized and symmetric and automatically adapts when hosts leave and join.
- Freenet document search involves a starting node sending request to another node, which in turn may send more request to other nodes, and so on until the document is found, or the max number of hops is reached.
- Prevents it from guaranteeing retrieval of existing documents or from providing low bounds on retrieval costs.

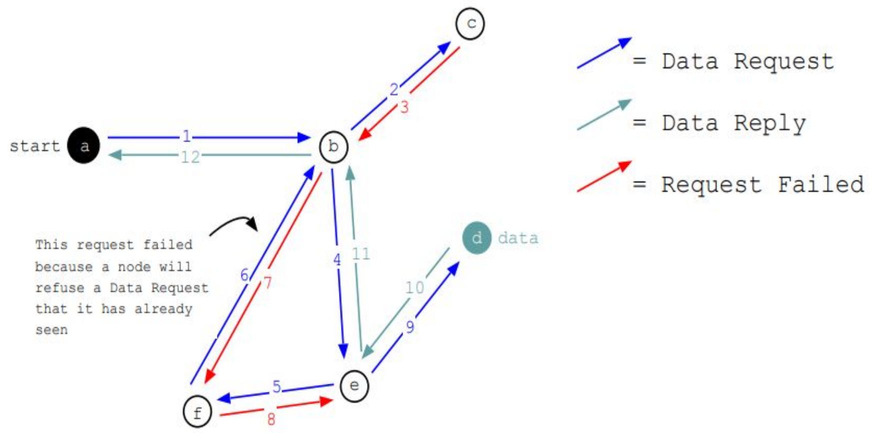

start
a
b
c
d
data
e
f
1
2
3
4
5
6
7
8
9
10
11
12
This request failed because a node will refuse a Data Request that it has already seen
= Data Request
= Data Reply
= Request Failed

Fig. 1. A typical request sequence.

Motivation - continued

- Core operation in P2P systems is efficient location of data items
- These P2P systems motivated research for efficient distribution of data items and retrieval.
- First protocols for efficient data items storage and retrieval include CAN, Chord, Pastry, and Tapestry.

Chord Protocol

- Distributed protocol that provides efficient lookup of data items
- It uses consistent hashing
  - Given a key -> map the key to a node
- Provide foundation for distribution of data. Some examples of application built on top of Chord can include
  - Cooperative mirroring
  - Time-shared storage
  - Distributed indexes
  - Combinatorial search

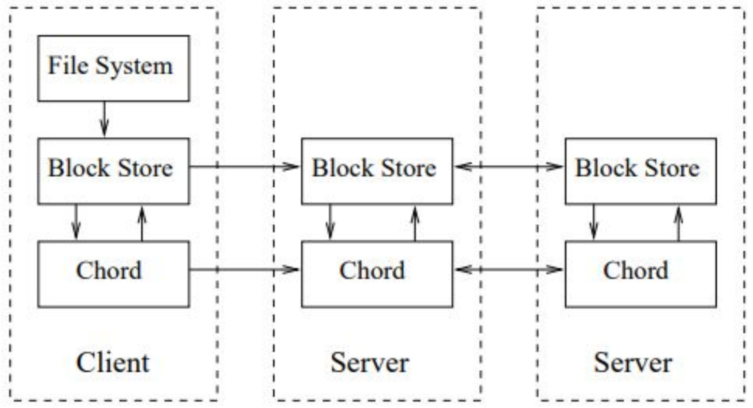


Fig. 1. Structure of an example Chord-based distributed storage system.

System Model

Simplify design of peer-to-peer systems by addressing following issues:

- **Load Balance**
  - Spread keys evenly over nodes
- **Decentralization**
  - Fully distributed, all nodes are equally important
- **Scalability**
  - Lookup cost grows as the log of the number of nodes
- **Availability**
  - Lookups work even when internal state changes (e.g. nodes joining or exiting)
- **Flexible naming**
  - No constraints in the structure of the keys

Consistent Hashing

- Assign each nodes and key an *m*-bit identifier using a hash function such as SHA-1
  - For nodes, hash the node’s IP address
  - For keys, hash the key itself
- *m* must be large enough such that the probability of nodes having the same hash is low
- Nodes are ordered in a circle (Chord ring) based on their identifier modulo 2^m
- A key *k* is assigned to the first node (successor node) whose identifier (hash) is equal to or follows *k* in the circle.

Consistent Hashing - Example

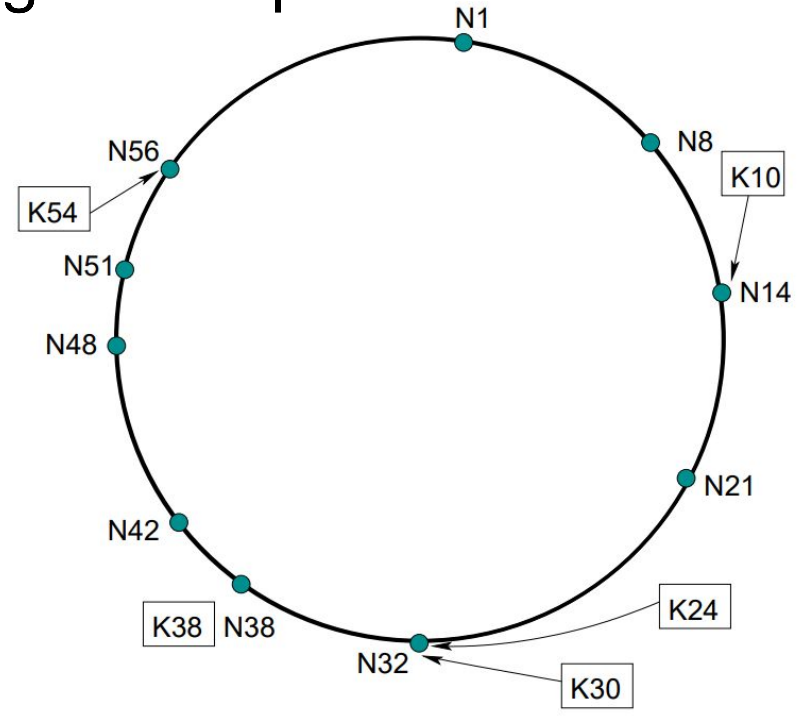
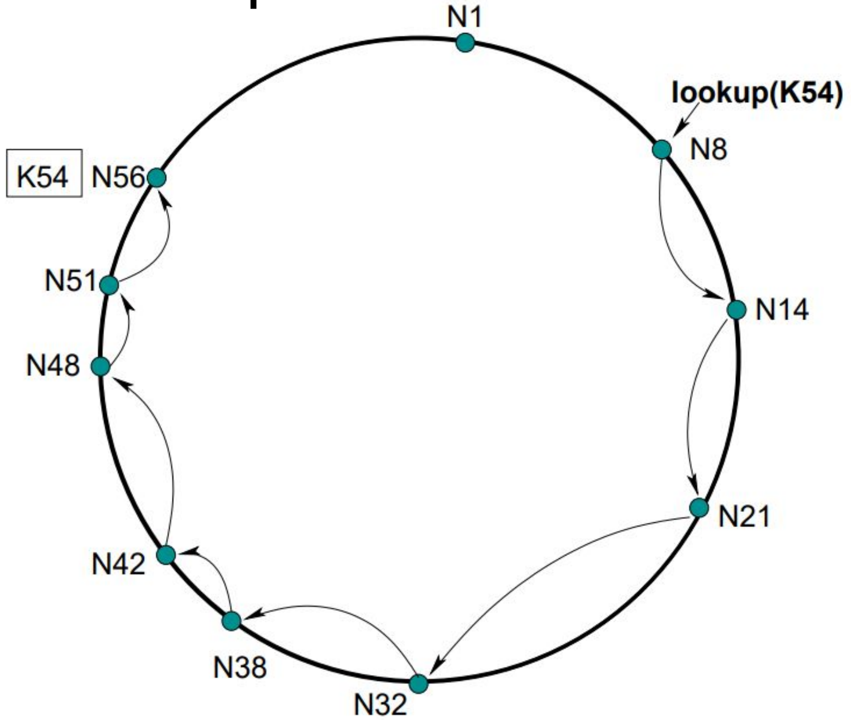


Fig. 2. An identifier circle (ring) consisting of 10 nodes storing five keys.

Consistent Hashing - Simple Lookup

```
// ask node $n$ to find the successor of $id$
$n$.find_successor($id$)
    if ($id \in (n, successor]$)
        return $successor$;
    else
        // forward the query around the circle
        return $successor.find\_successor(id)$;
```

(a)



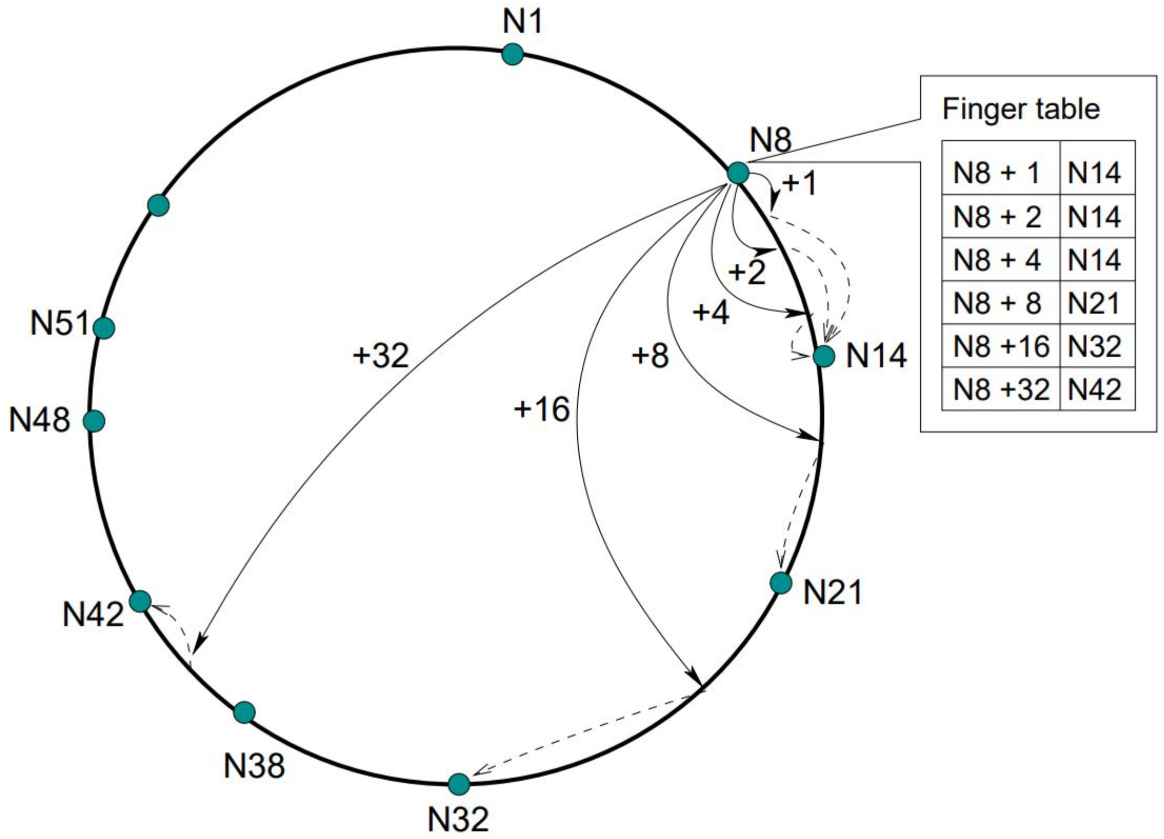
(b)

What is the issue?

Inefficient, might need to visit all nodes for an answer

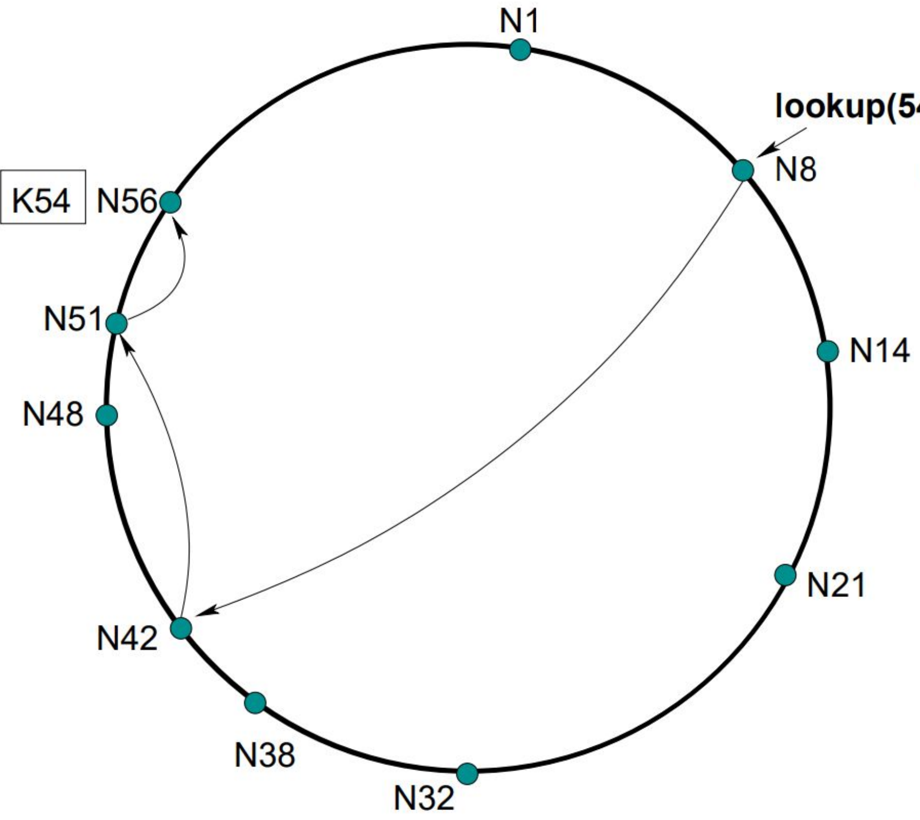
Consistent Hashing - Scalable lookup

- Store additional routing information (e.g. node identifier with IP address and port number) in a “Finger table”
- Each finger *k* in the table is defined as the first node that succeeds the current node *n*: *(n + 2^(k-1)) mod 2^m, 1 <= k <= m*.
- In the example, first finger of node 8 is 14, because (8 + 1) mod 2^6 = 9



N8 + 1	N14
N8 + 2	N14
N8 + 4	N14
N8 + 8	N21
N8 +16	N32
N8 +32	N42

Consistent Hashing - Scalable lookup



Finger table

N8 + 1	N14
N8 + 2	N14
N8 + 4	N14
N8 + 8	N21
N8 +16	N32
N8 +32	N42

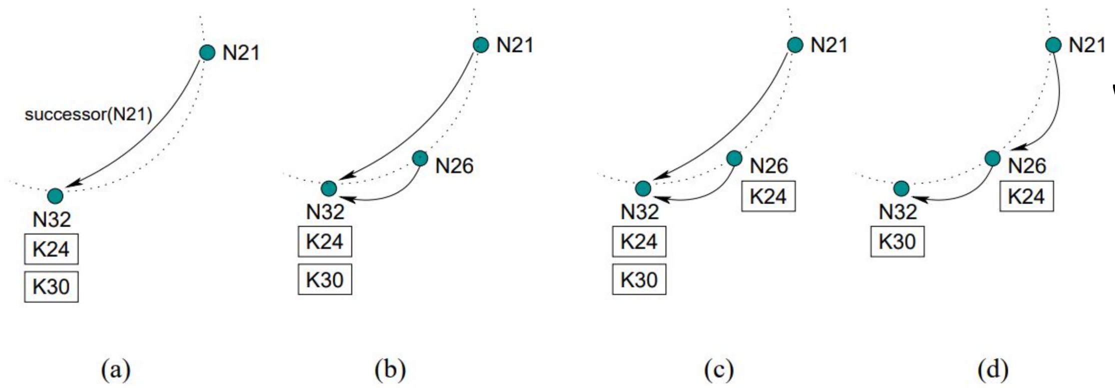
```
// ask node $n$ to find the successor of id
$n$.find_successor($id$)
  if ($id \in (n, successor]$)
    return $successor$;
  else
    $n' = closest\_preceding\_node(id)$;
    return $n'.find\_successor(id)$;

// search the local table for the highest predecessor of id
$n$.closest_preceding_node($id$)
  for $i = m$ downto 1
    if ($finger[i] \in (n, id)$)
      return $finger[i]$;
  return $n$;
```

Fig. 5. Scalable key lookup using the finger tab

Nodes Joining

- This handles a stable system, but how can we extend it to allow new nodes to join?
    - Remember: each node needs to know its successor
- We need new node *n* to join the ring and learn its successor. How can we do this?

- We can use the *find_successor* operation with our node key!
    - We can call this on any other node

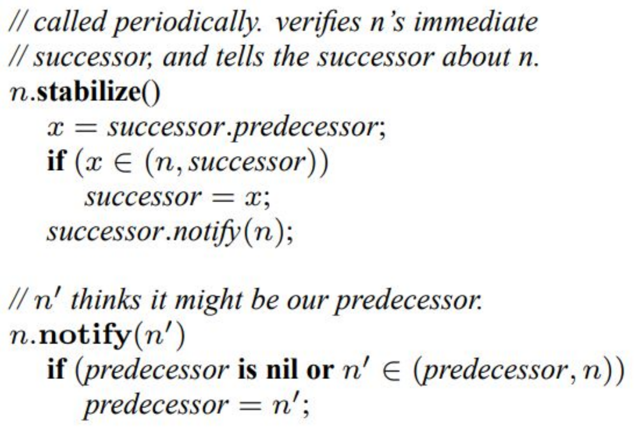


Will correctness be maintained?

What else is missing?

Node Stabilization

- Nodes need to learn about other nodes
    - They need some access point into the ring – this is why they must have a successor!
        - It doesn’t need to be right
- Asks successor about its predecessor
    - A new node may have been added in between, might need to update successor
    - Notify new successor to let it know we are the new predecessor (if we are actually right!)



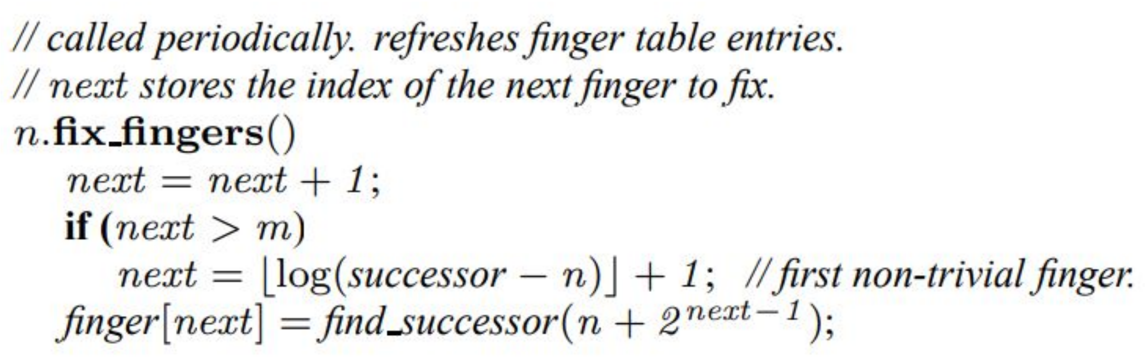
```
// called periodically. verifies n's immediate
// successor, and tells the successor about n.
n.stabilize()
    $x = successor.predecessor$;
    if ($x \in (n, successor)$)
        $successor = x$;
    $successor.notify(n)$;

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

Is correctness maintained?

What else should we update?

Fixing “Fingers”

- Nodes need to update their finger tables

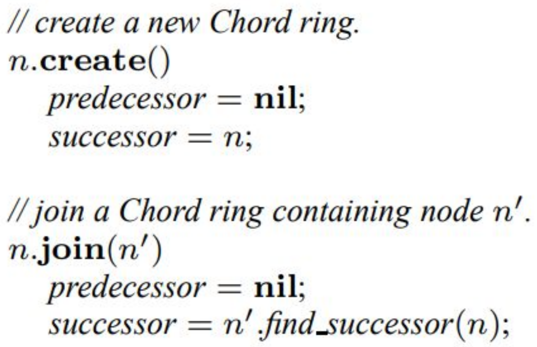


```
// called periodically. refreshes finger table entries.
// next stores the index of the next finger to fix.
$n$.fix_fingers()
    $next = next + 1$;
    if ($next > m$)
        $next = \lfloor \log(successor - n) \rfloor + 1$;  // first non-trivial finger.
    $finger[next] = find\_successor(n + 2^{next-1})$;
```

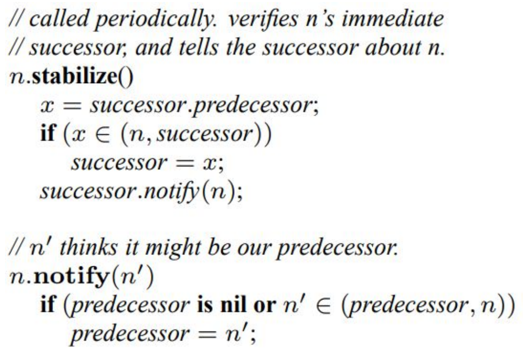
- Note: only updates 1 finger at a time!

Node Failure

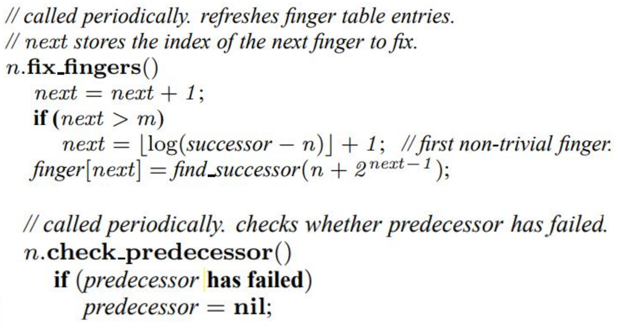
- What happens if a predecessor fails?



```
// create a new Chord ring.
$n$.create()
    $predecessor = \textbf{nil}$;
    $successor = n$;

// join a Chord ring containing node $n'$.
$n$.join($n'$)
    $predecessor = \textbf{nil}$;
    $successor = n'.find\_successor(n)$;
```



```
// called periodically. verifies n's immediate
// successor, and tells the successor about n.
$n$.stabilize()
    $x = successor.predecessor$;
    if ($x \in (n, successor)$)
        $successor = x$;
    $successor.notify(n)$;

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



```
// called periodically. refreshes finger table entries.
// next stores the index of the next finger to fix.
$n$.fix_fingers()
    $next = next + 1$;
    if ($next > m$)
        $next = \lfloor \log(successor - n) \rfloor + 1$;  // first non-trivial finger.
    $finger[next] = find\_successor(n + 2^{next-1})$;

// called periodically. checks whether predecessor has failed.
$n$.check_predecessor()
    if ($predecessor$ has failed)
        $predecessor = \textbf{nil}$;
```

- What happens if a successor fails?

Successor Failure

- Nodes maintain a list of successors.
  - If immediate successor does not respond, assumes that it is unreachable/down and use the second entry
- Must reconcile this list with its successor's successor list
  - Node *n*'s list: [*n1*, *n2*, *n3*]
  - Node *n2's* list: [*n3*, *n4*, *n5*]
  - If *n* cannot reach *n1*, new successor list should be: [*n2*, *n3*, *n4*]
- Unlikely that all successors fail at once
- Must modify the previous operations to work with this

Recursive vs. Iterative

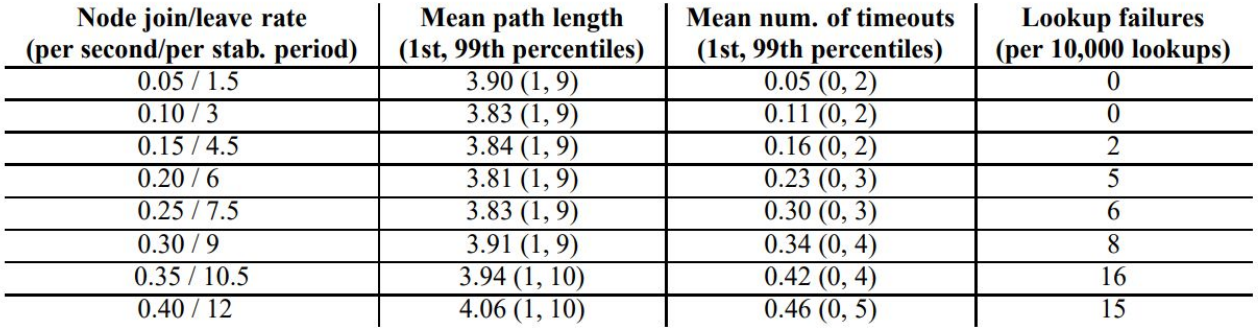
- *Iterative:* initiating node asks each node for information to reach successor
- *Recursive:* intermediate nodes forward request on to the next node
- Which is likely to be faster? How does this affect timeouts?

Other Optimizations

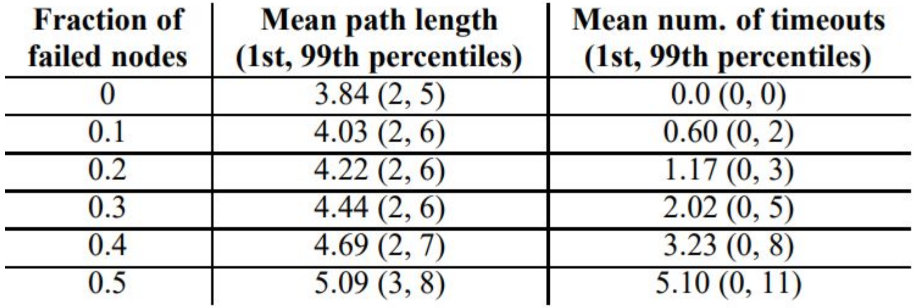
- Use virtual nodes with random identifiers to more evenly distribute nodes across the circle
- Use “alternative nodes” for each node in the finger table that have similar node keys, select the one that has the lowest latency
  - Requires recursive lookup
- Store replicas of the keys at a node in the successor nodes
- Nodes that choose to leave announce their departures

So why Chord?

- Good lookup speed despite reconfigurations



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

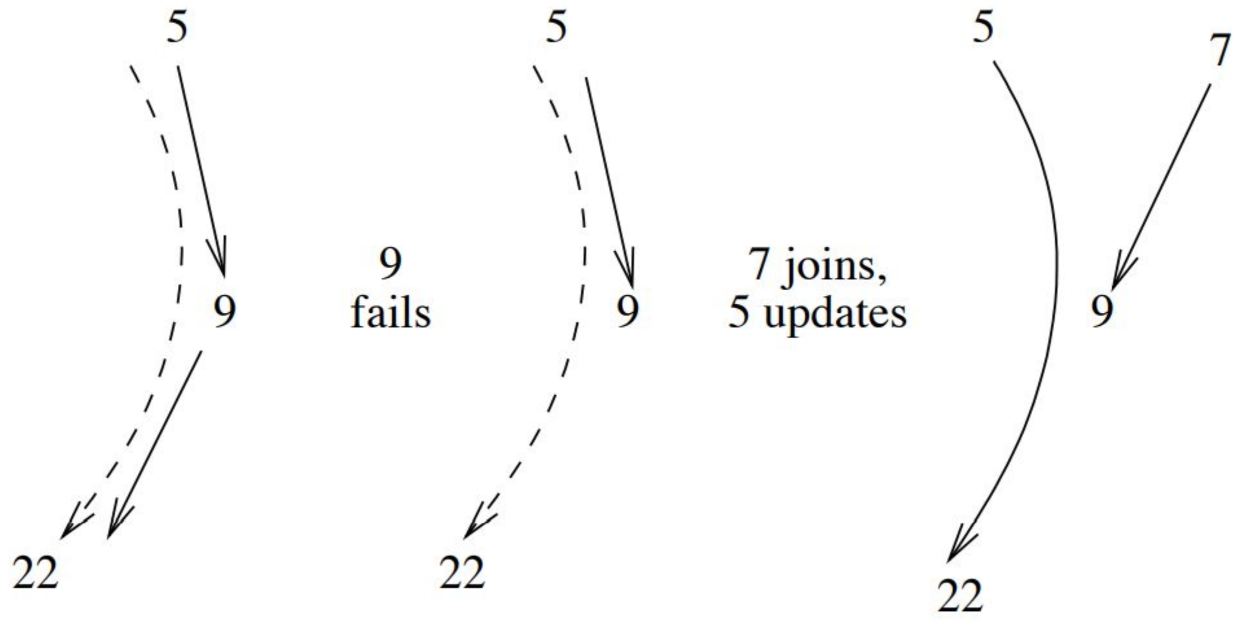


Fraction of failed nodes	Mean path length (1st, 99th percentiles)	Mean num. of timeouts (1st, 99th percentiles)
0	3.84 (2, 5)	0.0 (0, 0)
0.1	4.03 (2, 6)	0.60 (0, 2)
0.2	4.22 (2, 6)	1.17 (0, 3)
0.3	4.44 (2, 6)	2.02 (0, 5)
0.4	4.69 (2, 7)	3.23 (0, 8)
0.5	5.09 (3, 8)	5.10 (0, 11)

Using Lightweight Modeling To Understand Chord

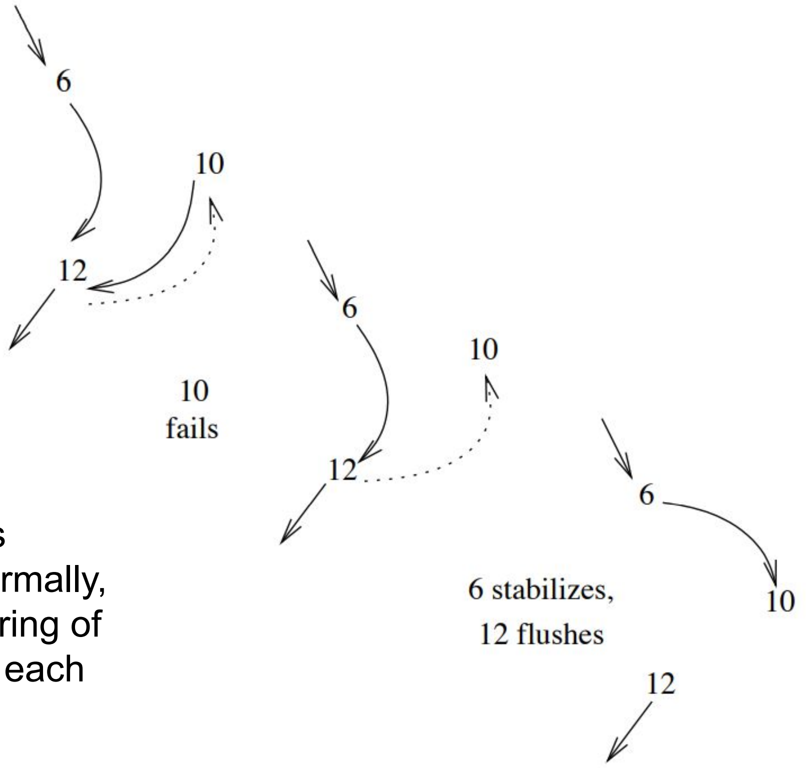
- Published over 10 years after Chord
- “correctness of the ring-maintenance protocol would mean that the protocol can eventually repair all disruptions in the ring structure, given ample time and no further disruptions while it is working. It is a form of “eventual consistency” with respect to reachability.”
    - No published version of Chord is incorrect! (as of 2012)
    - Proves this by showing scenarios where it violates certain invariants

Using Lightweight Modeling To Understand Chord



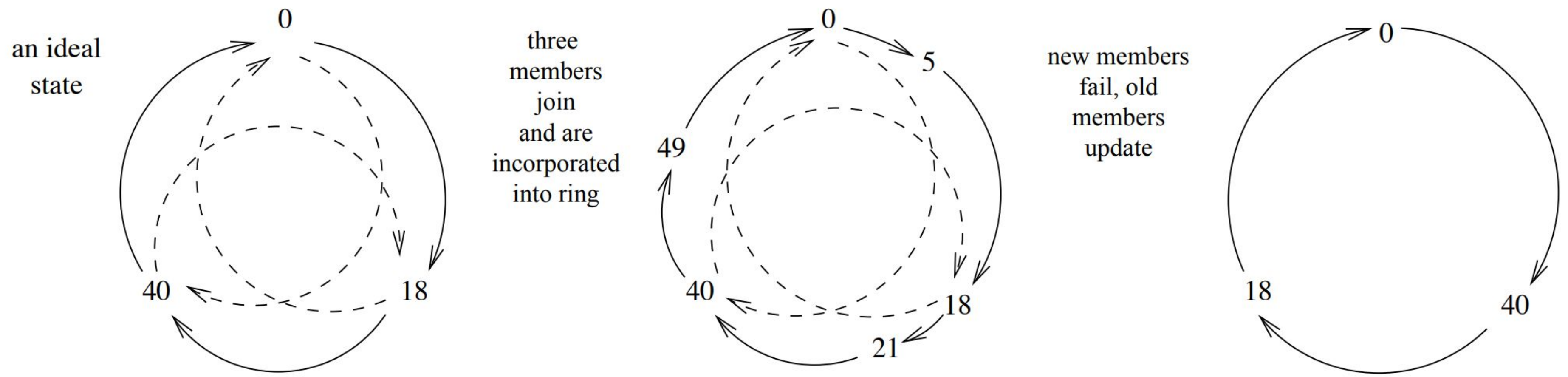
“The first such invariant is ConnectedAppendages. Informally, it says that an appendage to the ring stays connected to the ring.”

Using Lightweight Modeling To Understand Chord



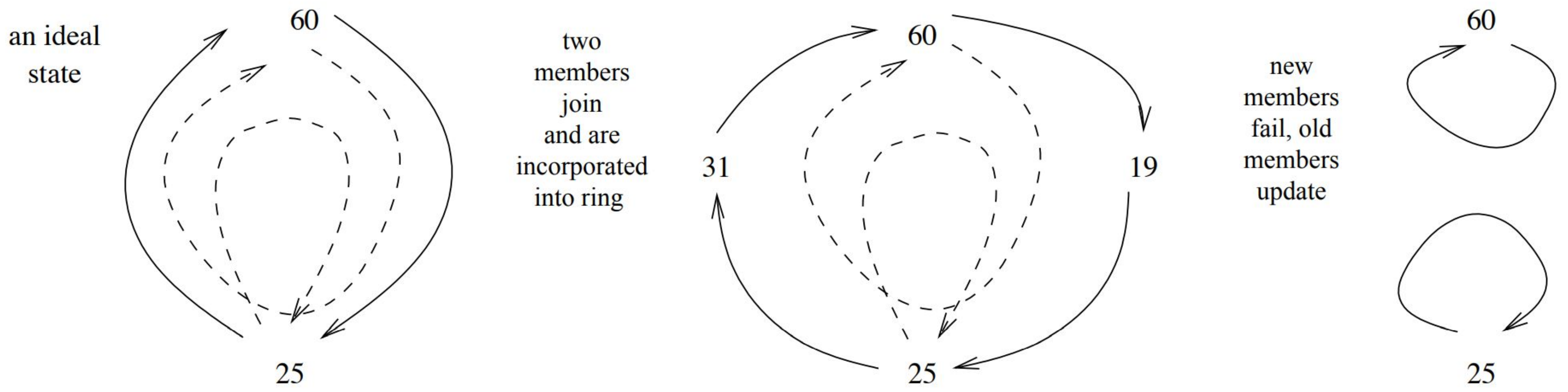
“The next claimed invariant is named AtLeastOneRing. Informally, it says that there is always a ring of members, all reachable from each other.”

Using Lightweight Modeling To Understand Chord


an ideal state
0
40
18
three members join and are incorporated into ring
0
5
49
40
18
21
new members fail, old members update
0
18
40

“The next claimed invariant is OrderedRing. Informally, it says that the ring is always ordered by identifiers.”

Using Lightweight Modeling To Understand Chord


an ideal state
60
25
two members join and are incorporated into ring
60
31
19
25
new members fail, old members update
60
25

“The final claimed invariant is AtMostOneRing. Informally, it says that the network does not break apart into two or more separate rings.”

Discussion

Consider a large network with significant geographical coverage. What can go wrong with node neighbor allocation? Can you propose a way to mitigate this issue and what trade-off is introduced by your solution?

- Large latency in hops – low number of hops isn’t great if each one takes a long time
- How might we address this?
  - Incorporate locality in key generation, so that geographically close nodes are near each other in the circle
  - Other ideas?

Discussion

Chord doesn’t inherently deal with replication of data, just distribution. How can we incorporate replication?

- Make each node a Paxos instance
- Let successors serve as paxos group
- Chain replication with successors
- Other ideas?