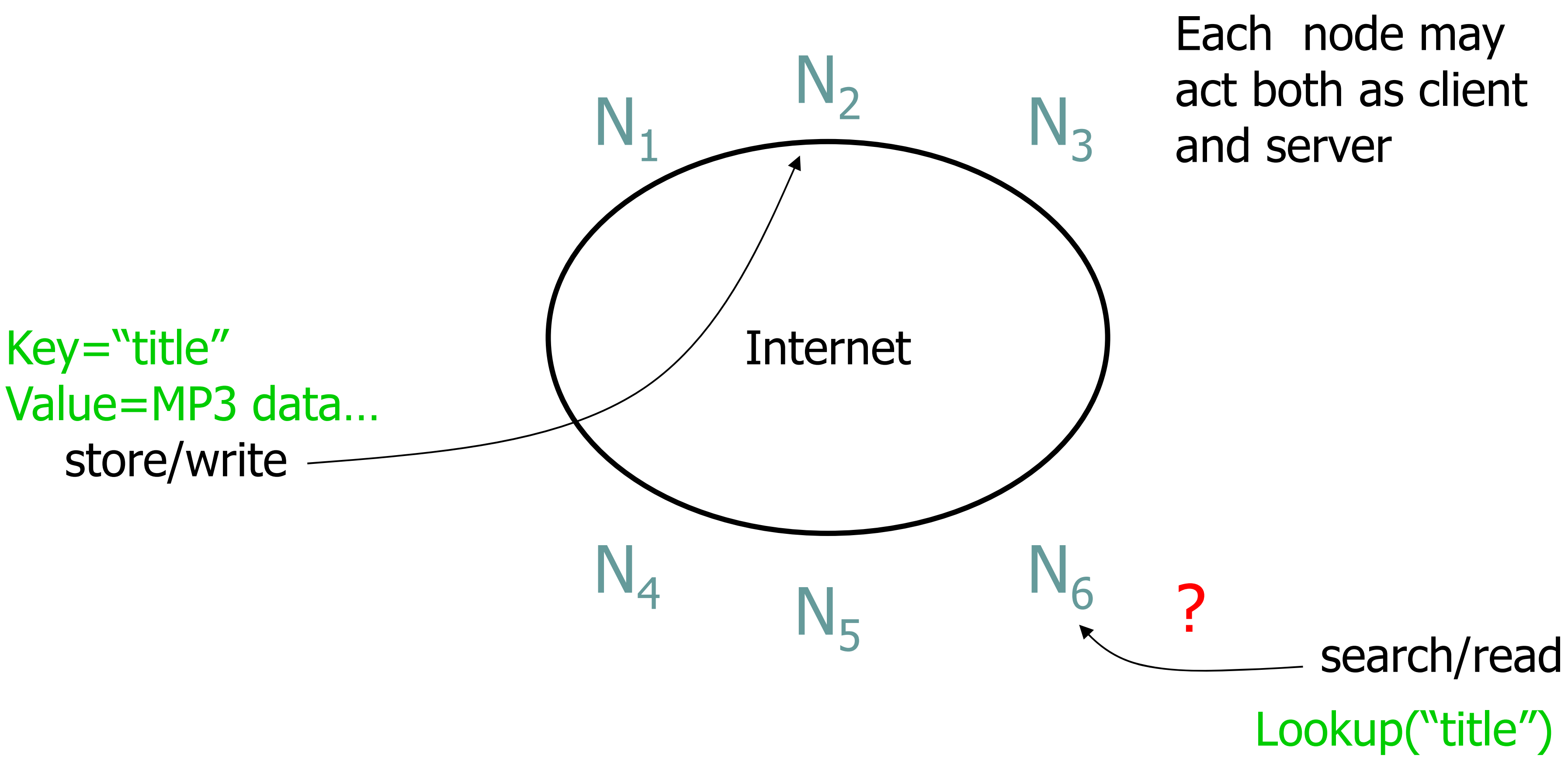
Chord:
A scalable peer-to-peer look-up protocol for internet applications

Acknowledgement
Some slide s are taken from University of California, University Of Berkeley and Max planck institute

What is Chord?

- In short: a peer-to-peer lookup/mapping service
- Solves problem of locating a data item in a collection of distributed nodes, considering frequent node arrivals and departures
- Core operation in most p2p systems is efficient location of data items
- Supports just one operation: given a key, it maps the key onto a *node*

The distributed store problem



Chord characteristics

- Simplicity, *provable* correctness, and *provable* performance
- Each Chord node needs routing information about only a few other nodes
- Resolves lookups via messages to other nodes (iteratively or recursively)
- Maintains routing information as nodes join and leave the system

Addressed Difficult Problems

- **Load balance**: distributed hash function, spreading keys evenly over nodes
- **Decentralization**: chord is fully distributed, no node more important than other, improves robustness
- **Scalability**: logarithmic growth of lookup costs with number of nodes in network, even very large systems are feasible
- **Availability**: chord automatically adjusts its internal tables to ensure that the node responsible for a key can always be found
- **Flexible naming**: no constraints on the structure of the keys – key-space is flat, flexibility in how to map names to Chord keys

The Base Chord Protocol

Specifies

- How to find the locations of keys
- How new nodes join the system
- How to recover from the failure or planned departure of existing nodes

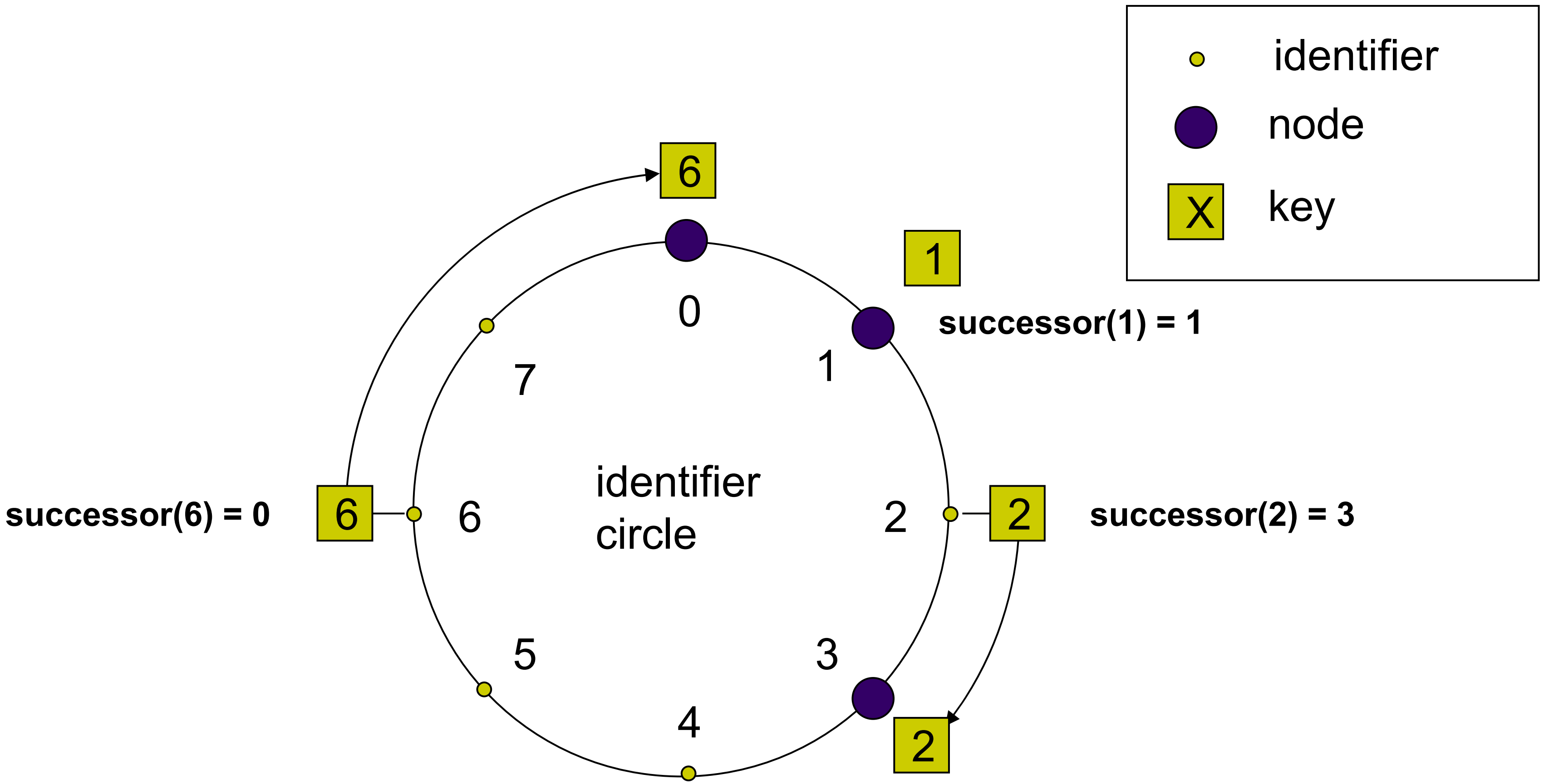
The Chord algorithm – Construction of the Chord ring

- Hash function assigns each node *and* key an m-bit *identifier* using a base hash function such as SHA-1
    - ID(node) = hash(IP, Port)
    - ID(key) = hash(key)
    - Both are uniformly distributed
    - Both exist in the same ID space
- Properties of consistent hashing:
    - Function balances load: 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

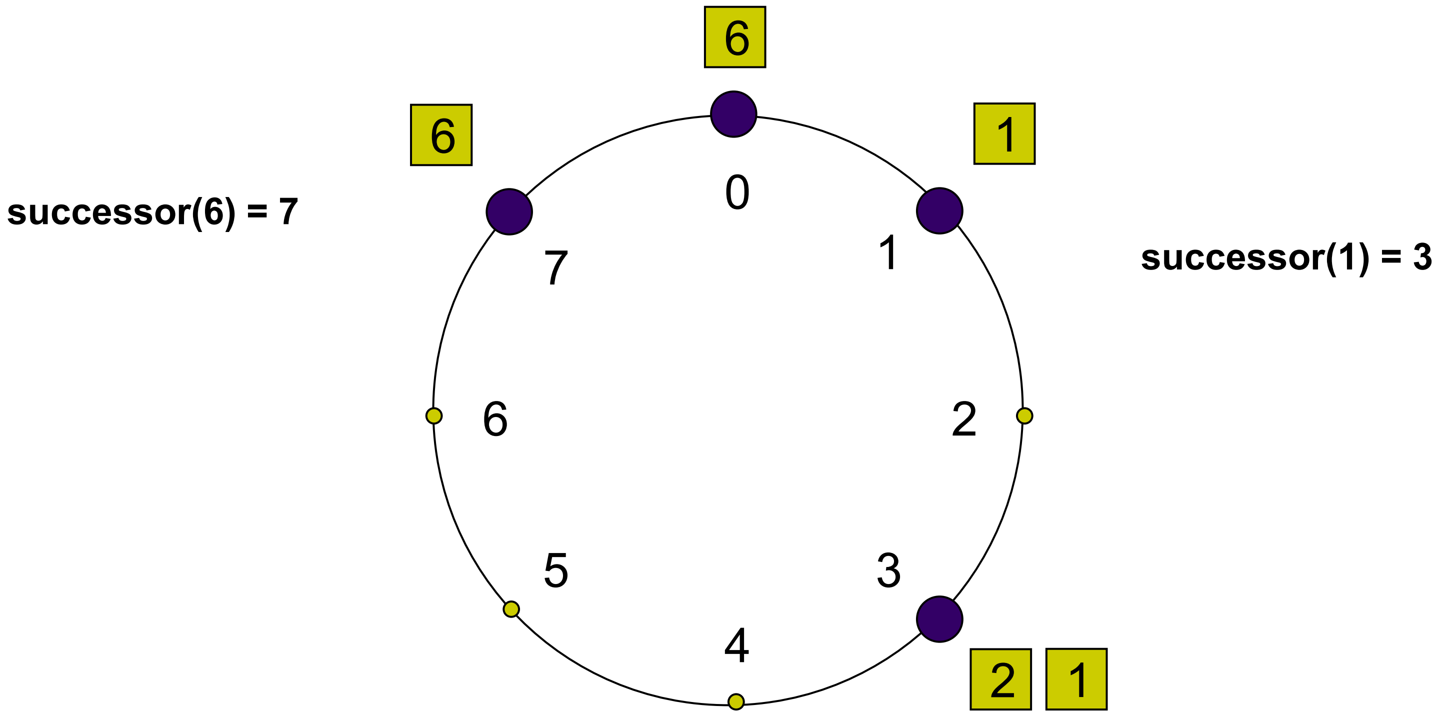
The Chord algorithm – Construction of the Chord ring

- identifiers are arranged on a identifier circle modulo 2^m => **Chord ring**
- a key k is assigned to the node whose identifier is **equal to or greater** than the key‘s identifier
- this node is called successor(k) and is the first node clockwise from k.

The Chord algorithm – Construction of the Chord ring



Node Joins and Departures



The Chord algorithm – Simple node localization

// ask node n *to find the successor of* id

n.find_successor(id)

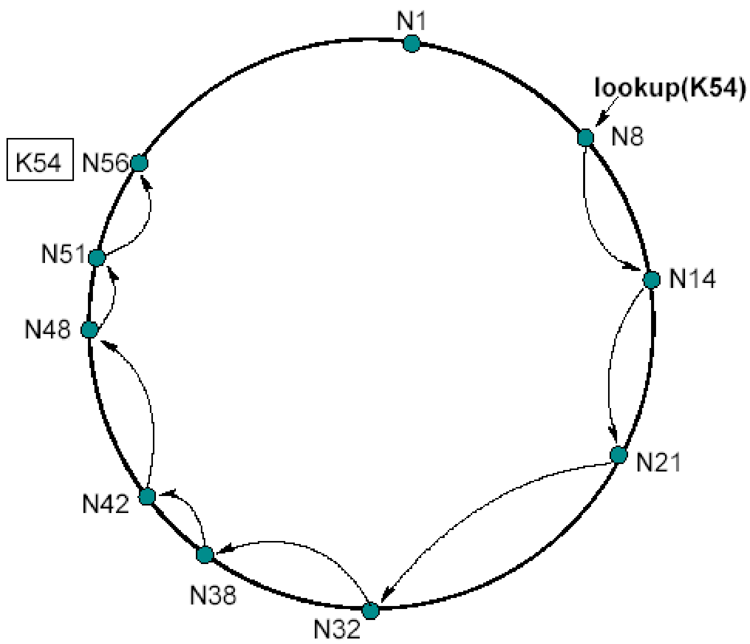
if (id $\in$ {id mapped onto n})

return *n*;

else

// forward the query around the circle

return *successor.find_successor*(id);



=> Number of messages linear in the number of nodes !

The Chord algorithm – Scalable node localization

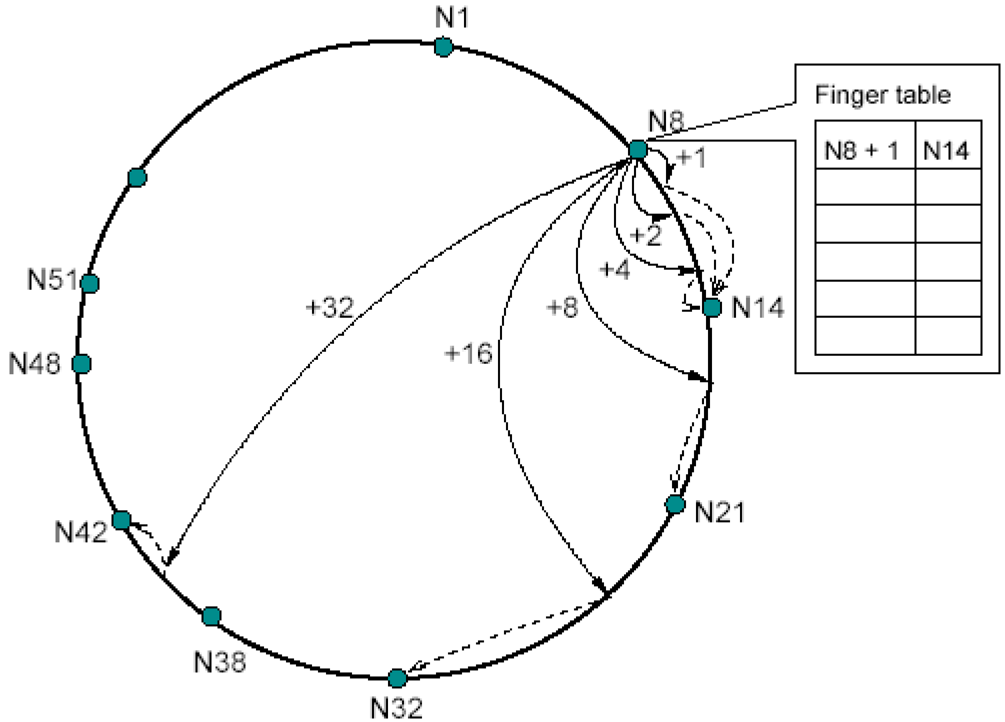
- Additional routing information to accelerate lookups
- Each node n contains a *routing table* with up to m entries (m: number of bits of the identifiers) => *finger table*
- i^{th} entry in the table at node n contains the first node s that succeds n by at least 2^{i-1}
- s = successor $(n + 2^{i-1})$
- s is called the i^{th} finger of node n

The Chord algorithm – Scalable node localization

Finger table:

finger[i] =

successor $(n + 2^{i-1})$

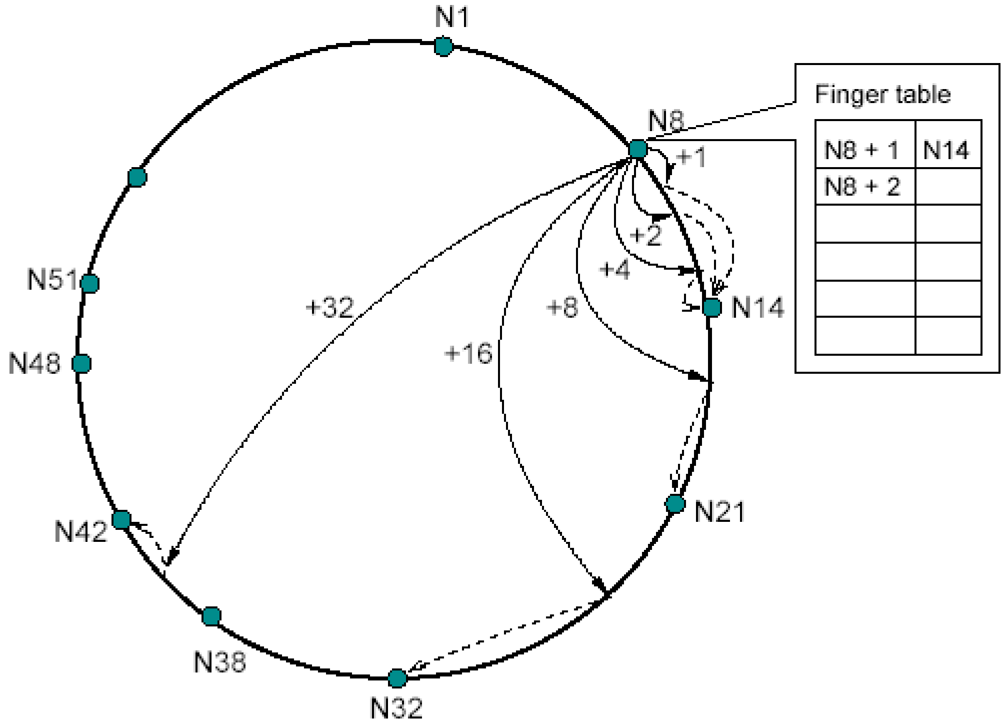


The Chord algorithm – Scalable node localization

Finger table:

finger[i] =

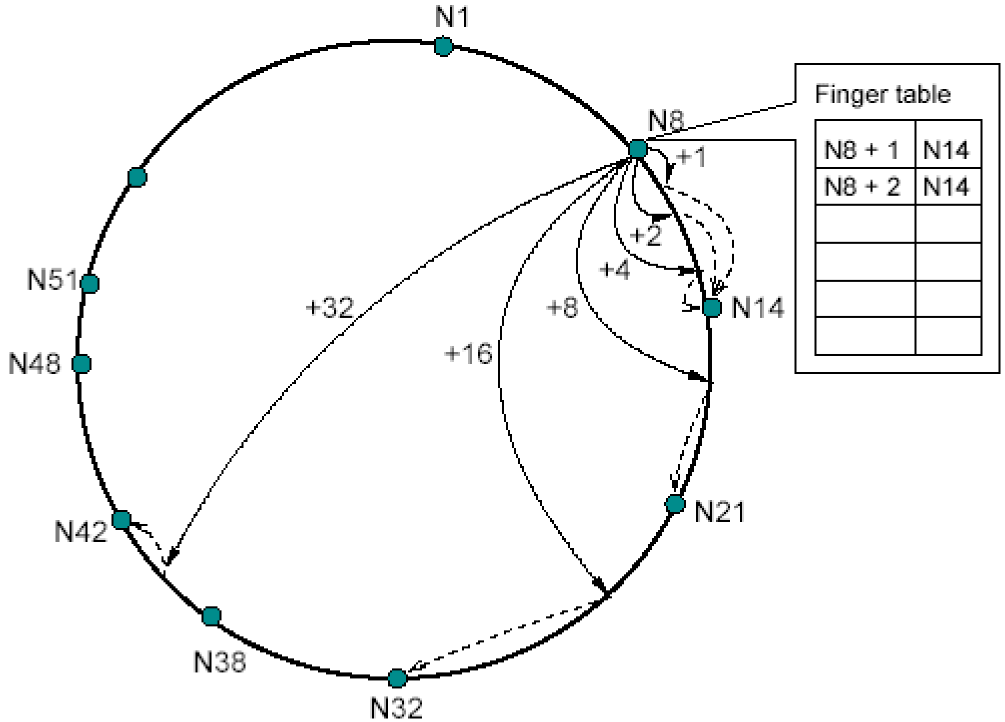
successor $(n + 2^{i-1})$



The Chord algorithm – Scalable node localization

Finger table:

finger[i] = successor (n + 2^{i-1})

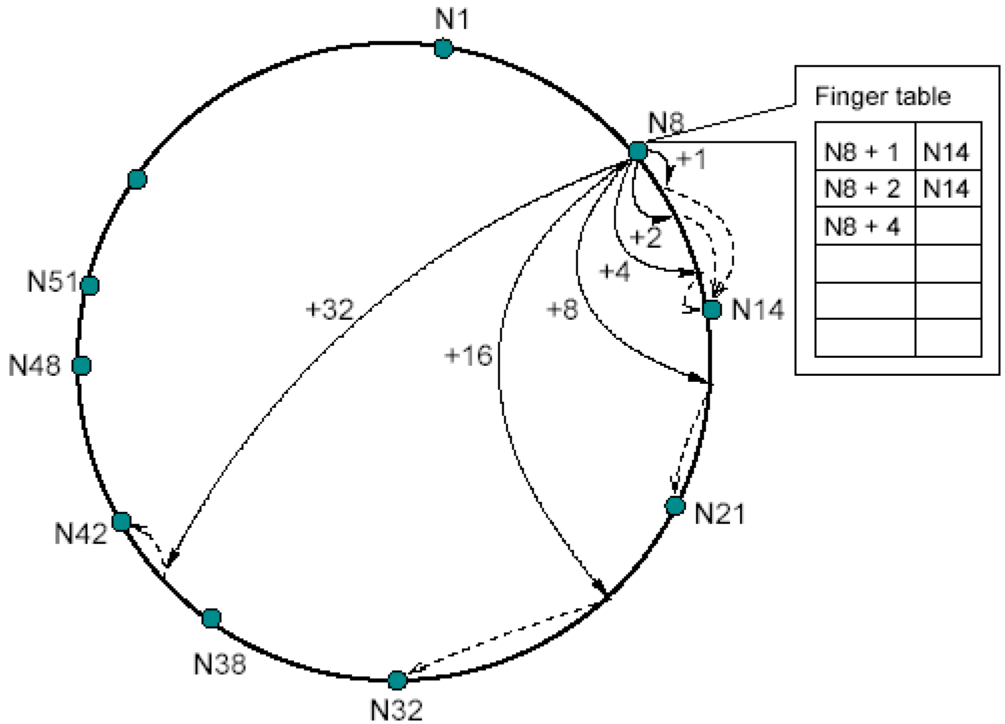


The Chord algorithm – Scalable node localization

Finger table:

finger[i] =

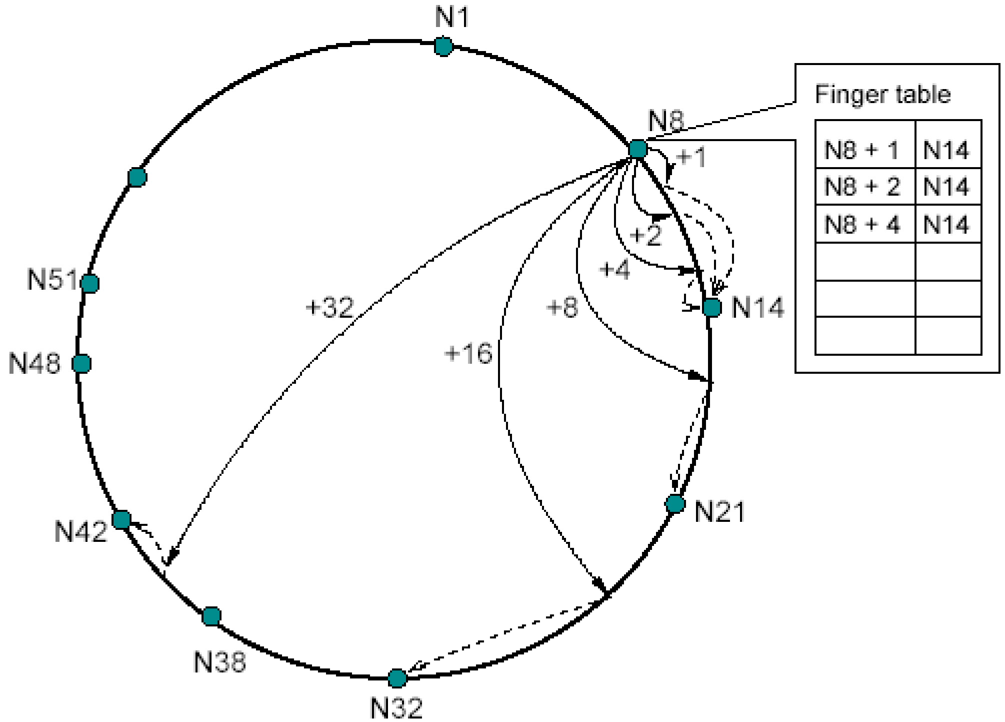
successor (n + 2^{i-1}*)*



The Chord algorithm – Scalable node localization

Finger table:

finger[i] = successor (n + 2^{i-1})

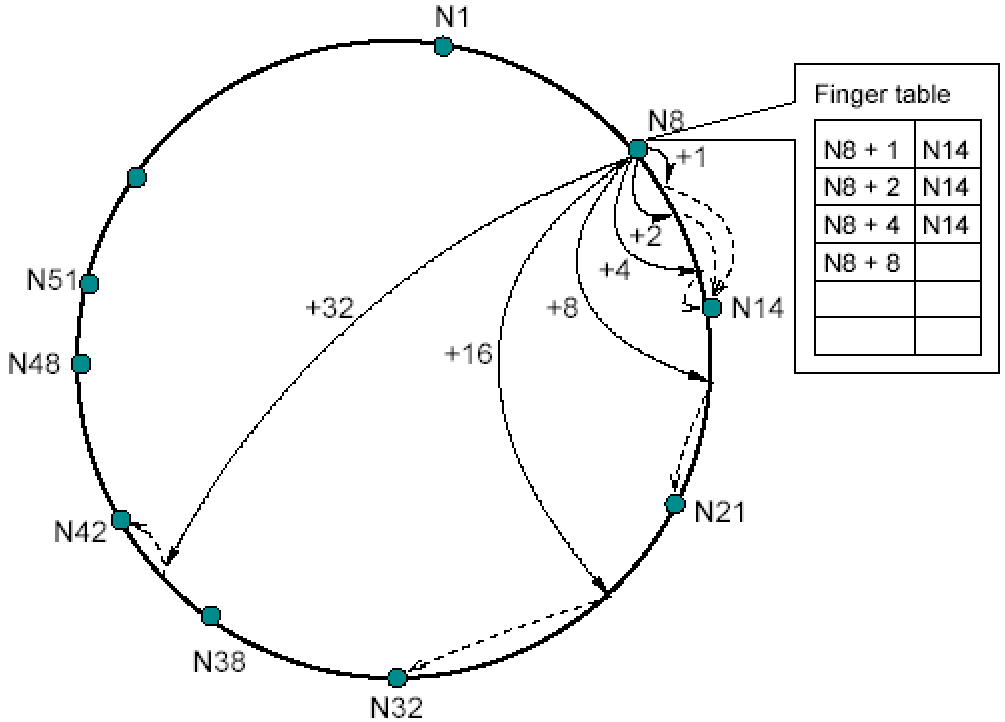


The Chord algorithm – Scalable node localization

Finger table:

finger[i] =

successor $(n + 2^{i-1})$



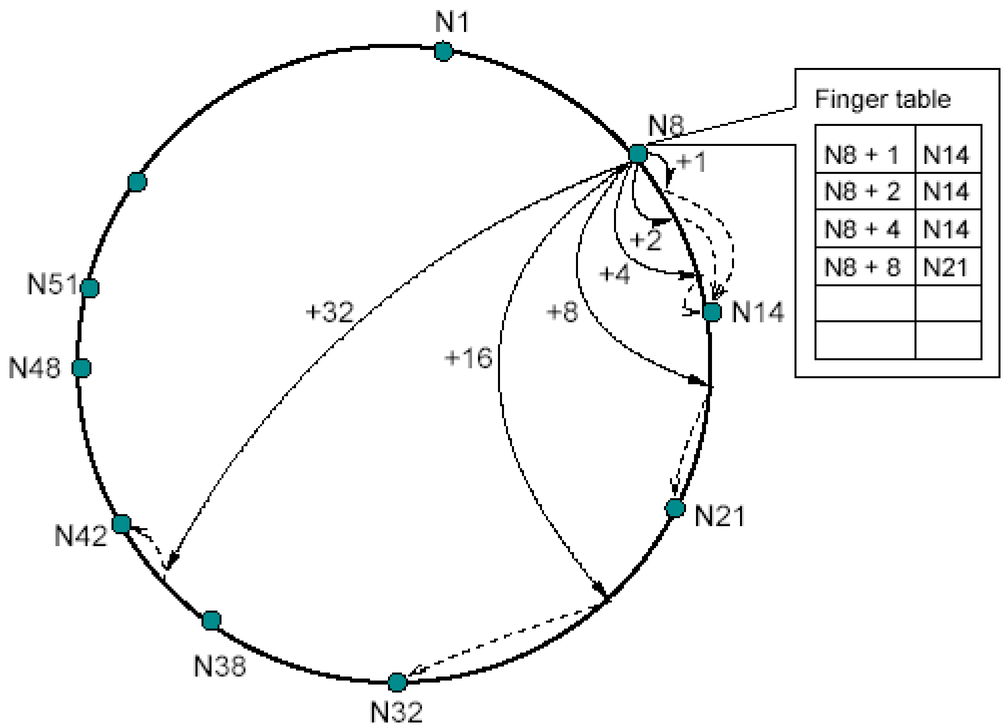
N8 + 1	N14
N8 + 2	N14
N8 + 4	N14
N8 + 8	

The Chord algorithm – Scalable node localization

Finger table:

finger[i] =

successor $(n + 2^{i-1})$

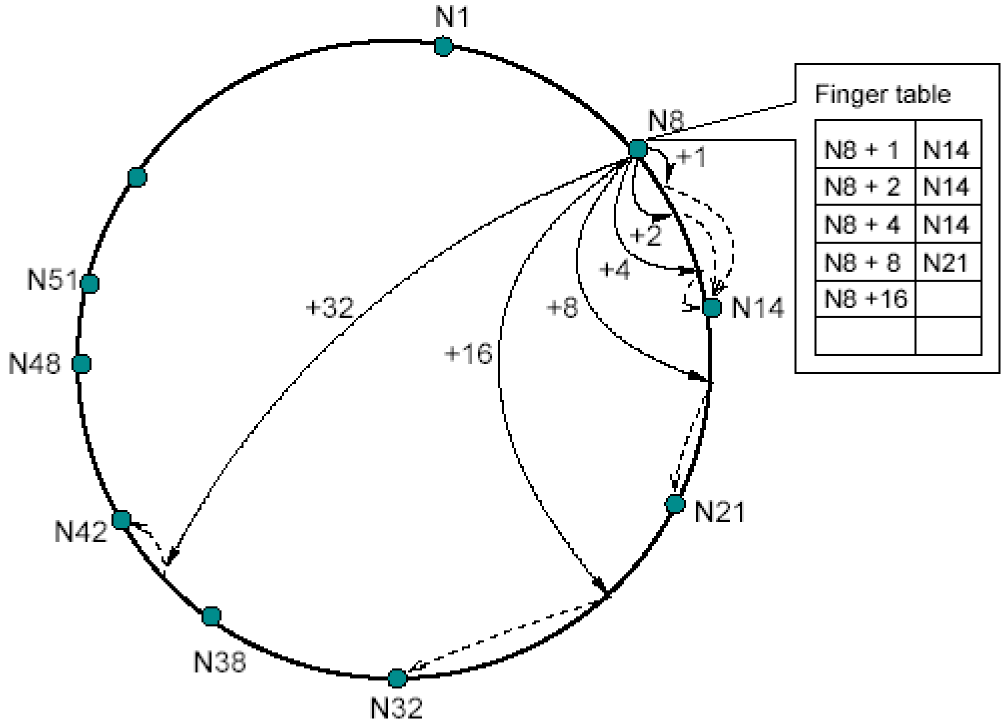


The Chord algorithm – Scalable node localization

Finger table:

finger[i] =

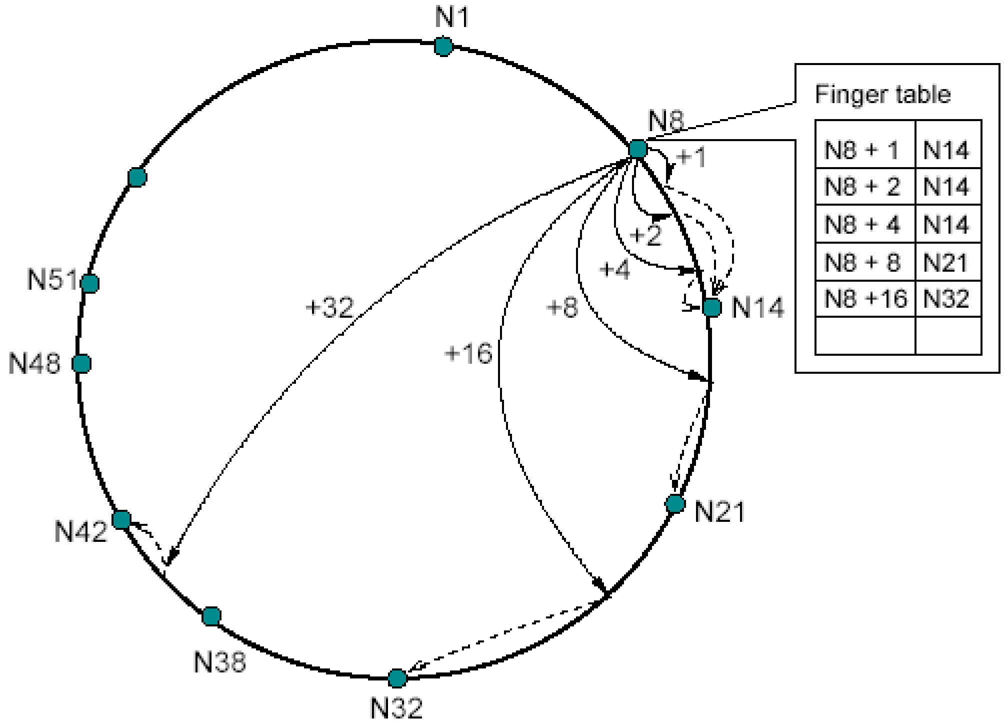
successor $(n + 2^{i-1})$



The Chord algorithm – Scalable node localization

Finger table:

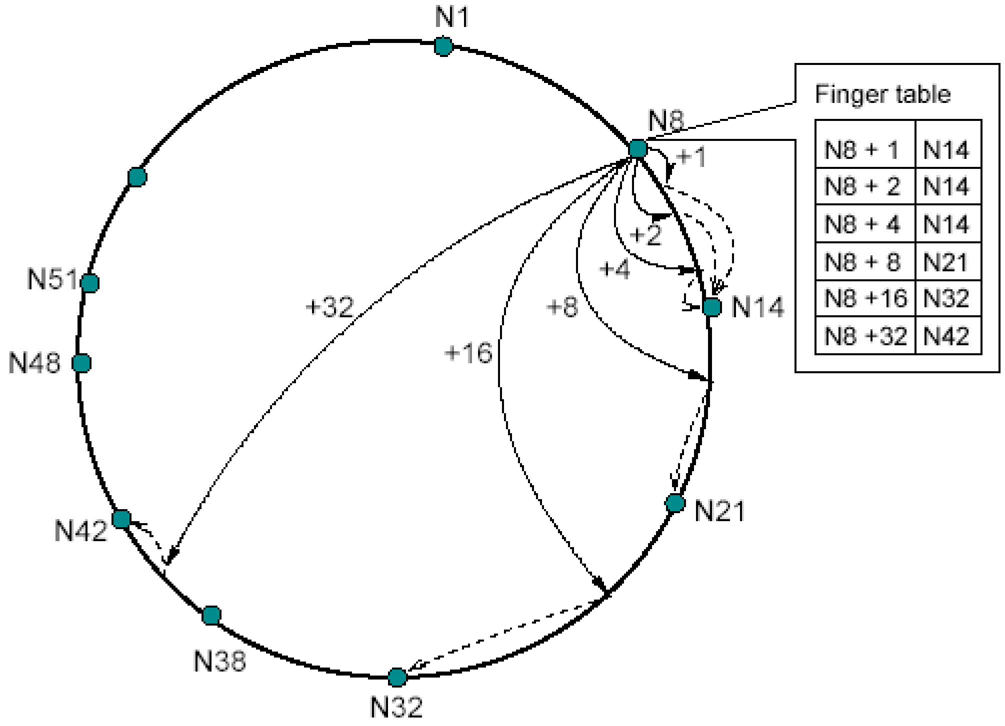
finger[i] = successor (n + 2^{i-1})



The Chord algorithm – Scalable node localization

Finger table:

finger[i] = successor (n + 2^{i-1})



The Chord algorithm – Scalable node localization

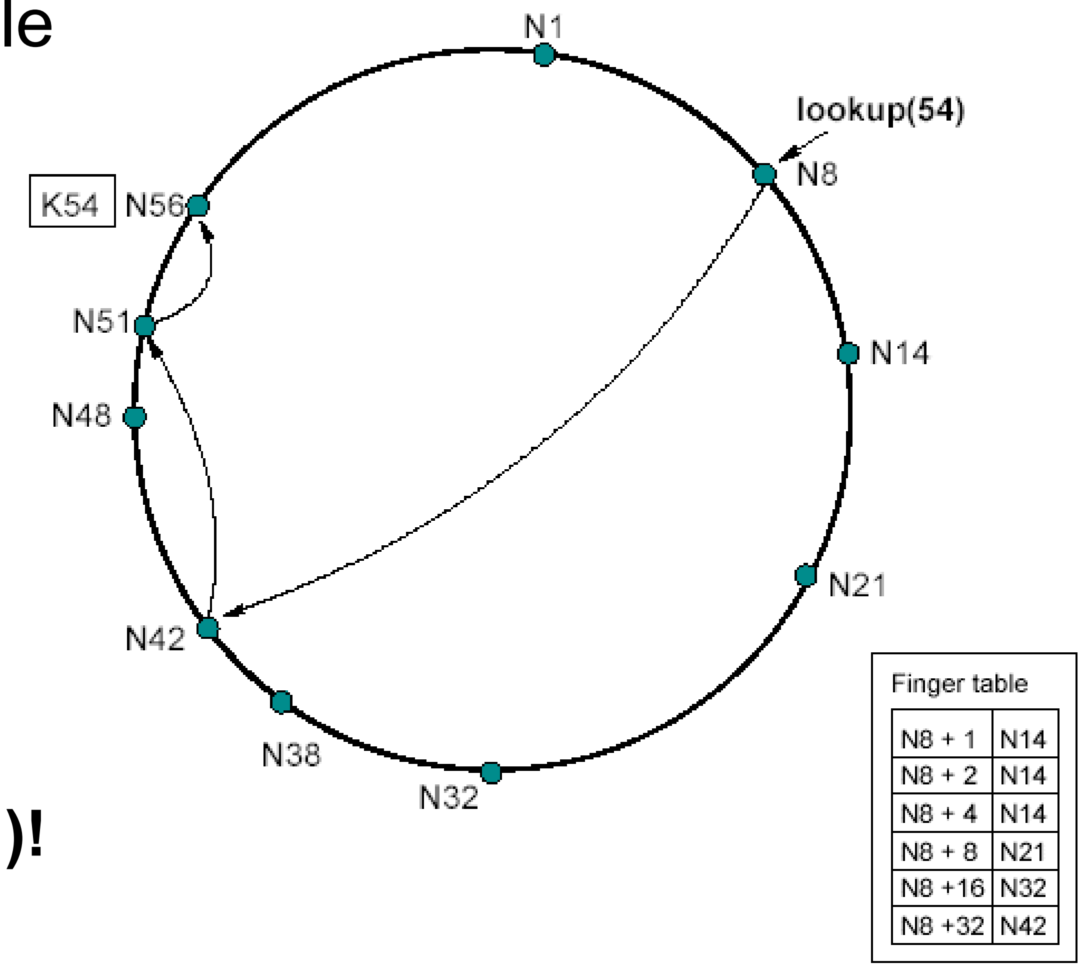
Important characteristics of this scheme:

- Each node stores information about only a small number of nodes (m)
- Each nodes knows more about nodes closely following it than about nodes far away
- A finger table generally does not contain enough information to directly determine the successor of an arbitrary key k

The Chord algorithm – Scalable node localization

- Search in finger table for the nodes which most immediatly precedes id
- Invoke find_successor from that node

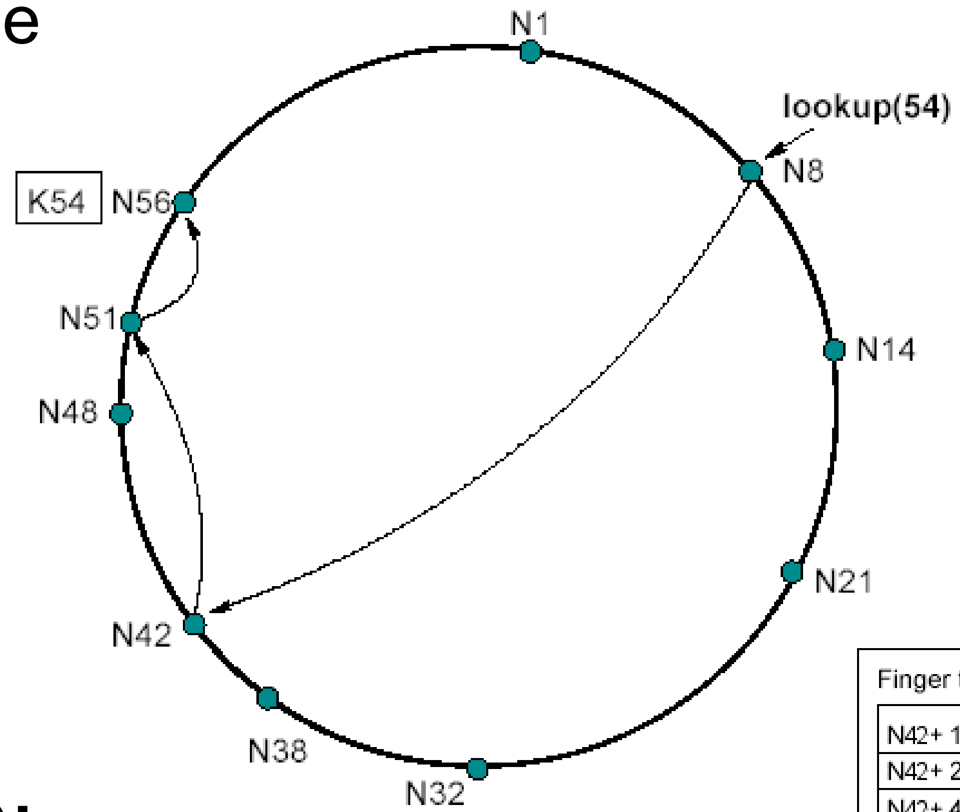
=> Number of messages O(log N)!



The Chord algorithm – Scalable node localization

- Search in finger table for the nodes which most immediatly precedes id
- Invoke find_successor from that node

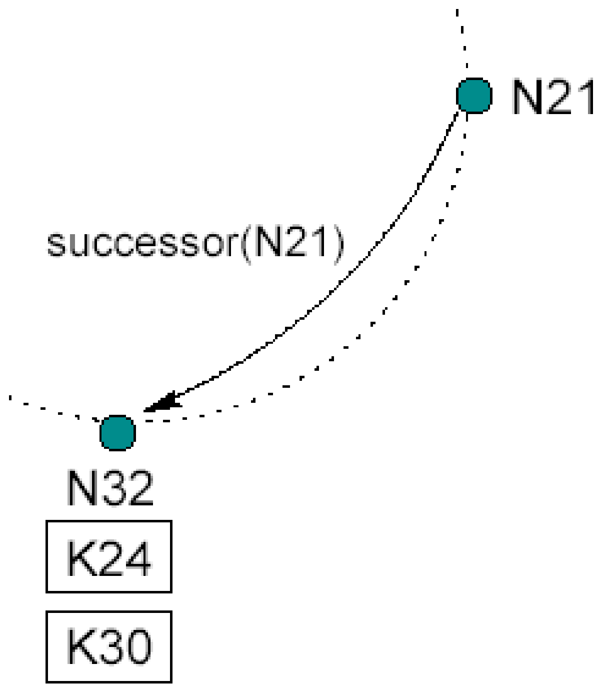
=> Number of messages O(log N)!



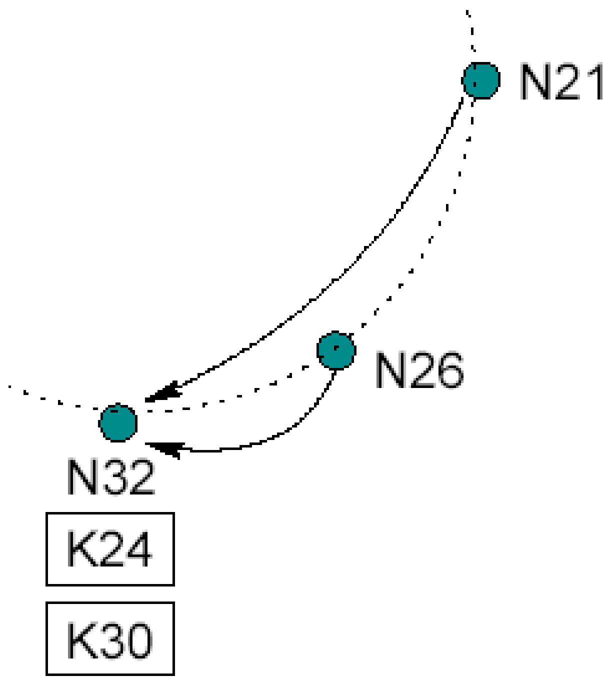
Finger table

N42+ 1	N48
N42+ 2	N48
N42+ 4	N48
N42+ 8	N51
N42+16	N1
N42+32	N14

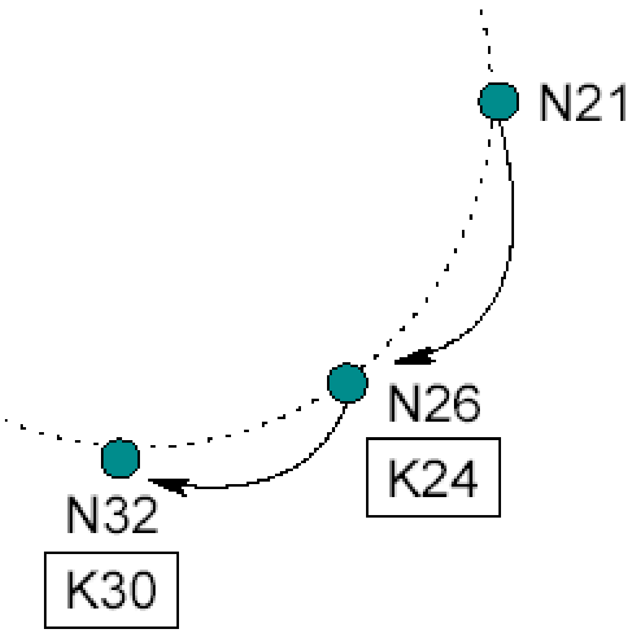
The Chord algorithm – Node joins and stabilization



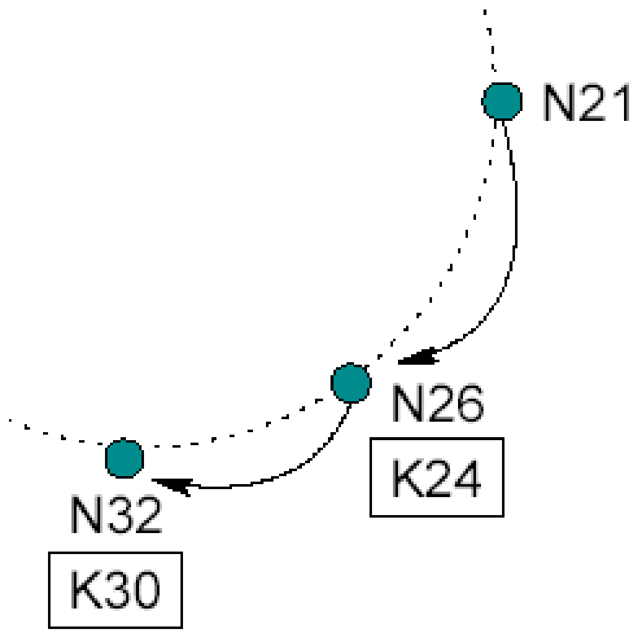
The Chord algorithm – Node joins and stabilization



The Chord algorithm – Node joins and stabilization



The Chord algorithm – Node joins and stabilization



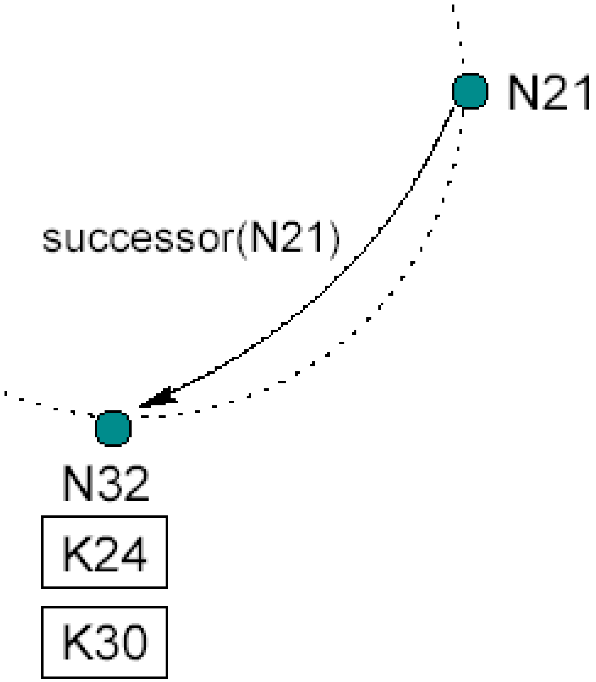
- To ensure correct lookups, all successor pointers must be up to date
- => stabilization protocol running periodically in the background
- Updates finger tables and successor pointers

The Chord algorithm – Node joins and stabilization

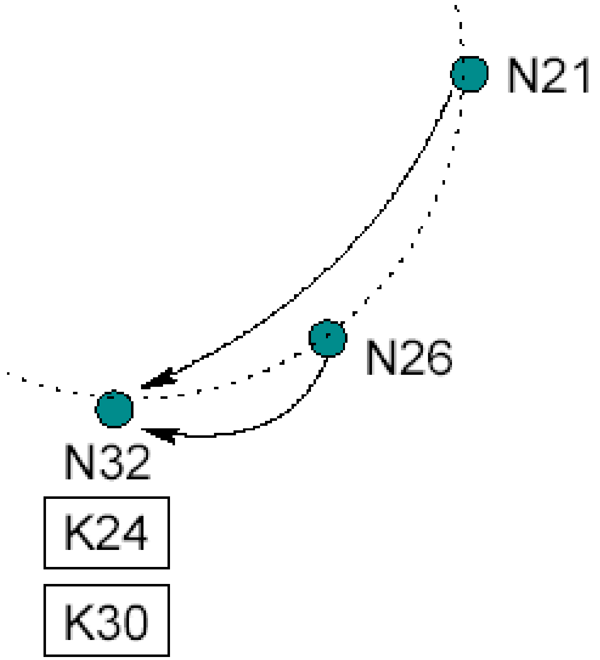
Stabilization protocol:

- Stabilize(): n asks its successor for its predecessor p and decides whether p should be n‘s successor instead (this is the case if p recently joined the system).
- Notify(): notifies n‘s successor of its existence, so it can change its predecessor to n
- Fix_fingers(): updates finger tables

The Chord algorithm – Node joins and stabilization

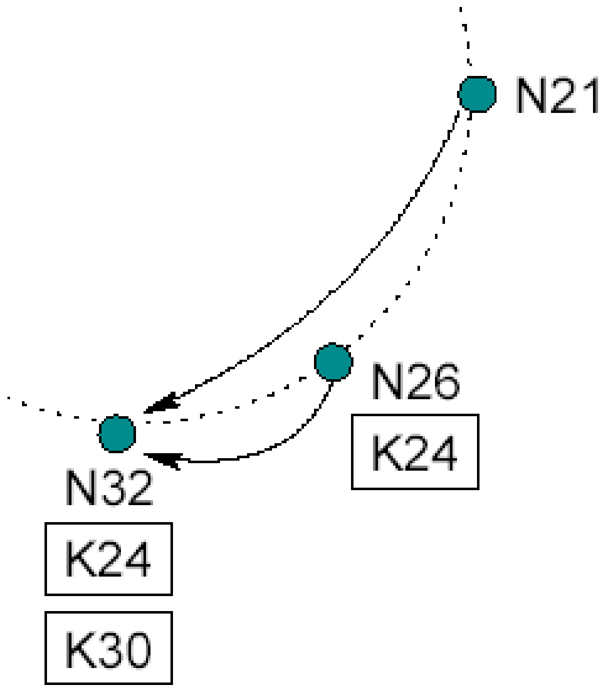


The Chord algorithm – Node joins and stabilization



- N26 joins the system
- N26 aquires N32 as its successor
- N26 notifies N32
- N32 aquires N26 as its predecessor

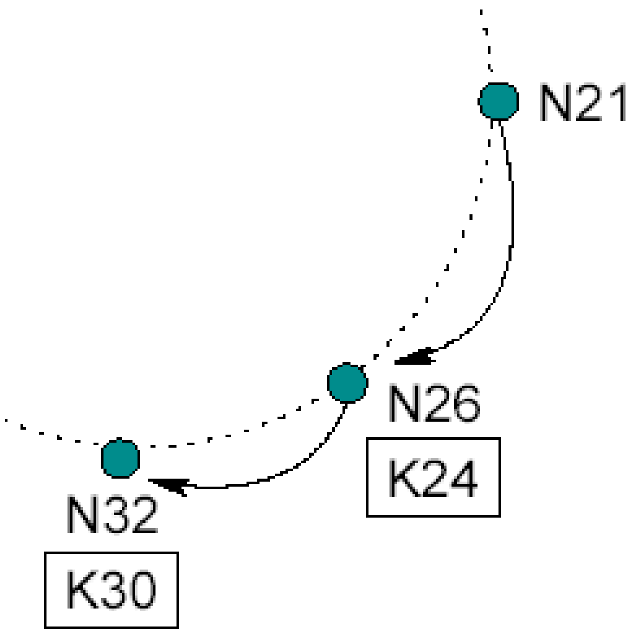
The Chord algorithm – Node joins and stabilization



• N26 copies keys

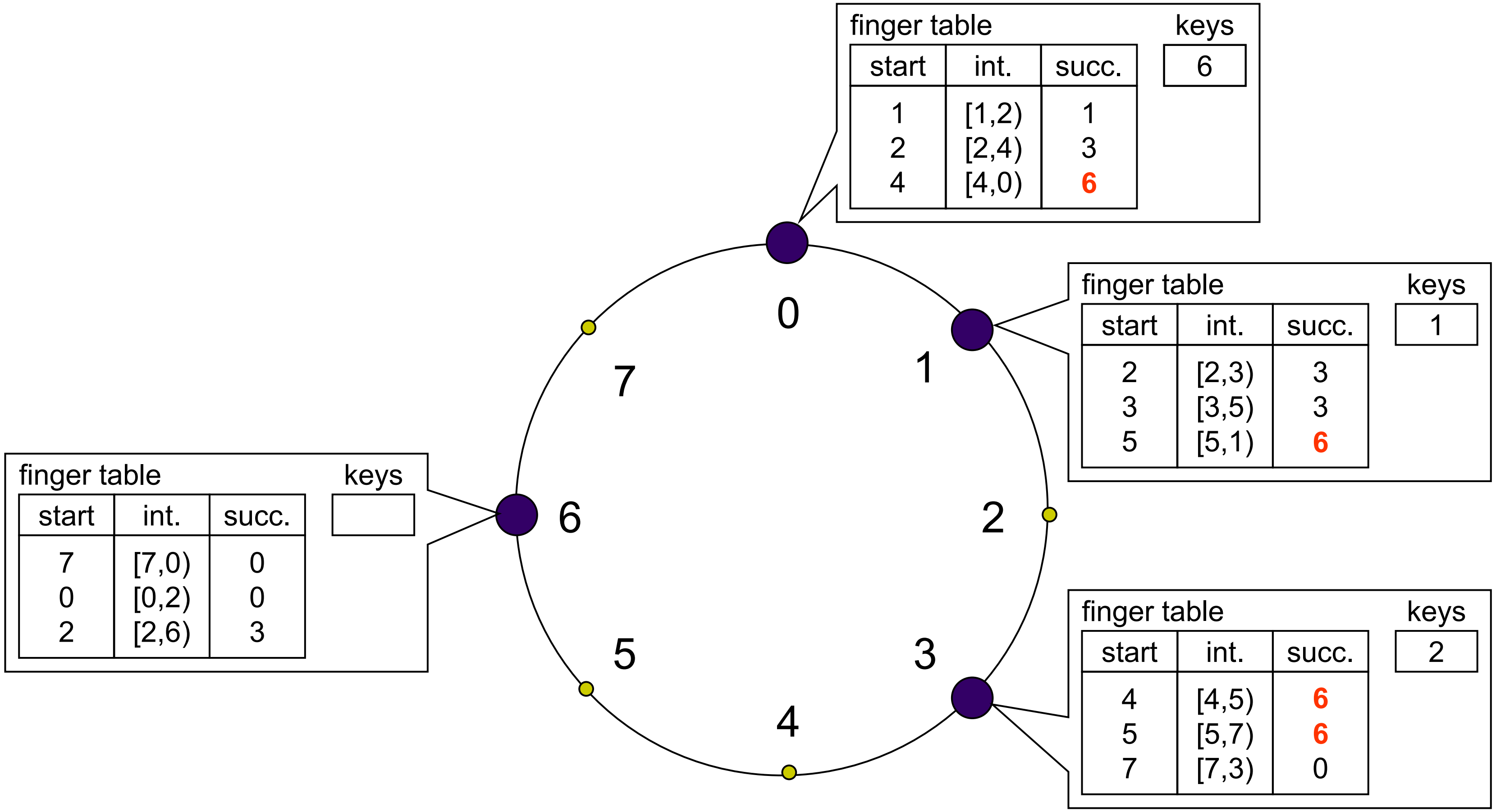
• N21 runs stabilize() and asks its successor N32 for its predecessor which is N26.

The Chord algorithm – Node joins and stabilization

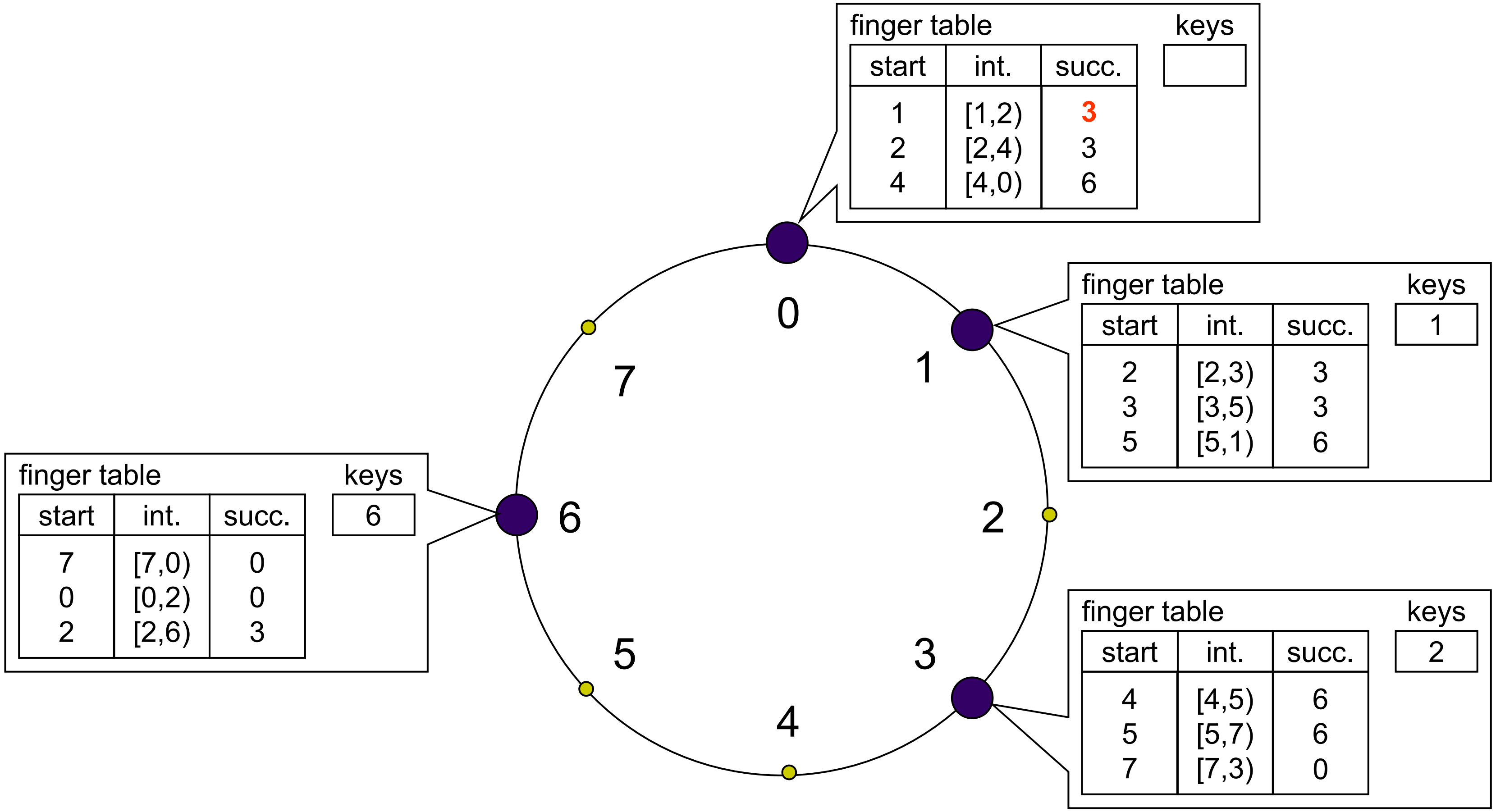


- N21 aquires N26 as its successor
- N21 notifies N26 of its existence
- N26 aquires N21 as predecessor

Node Joins – with Finger Tables

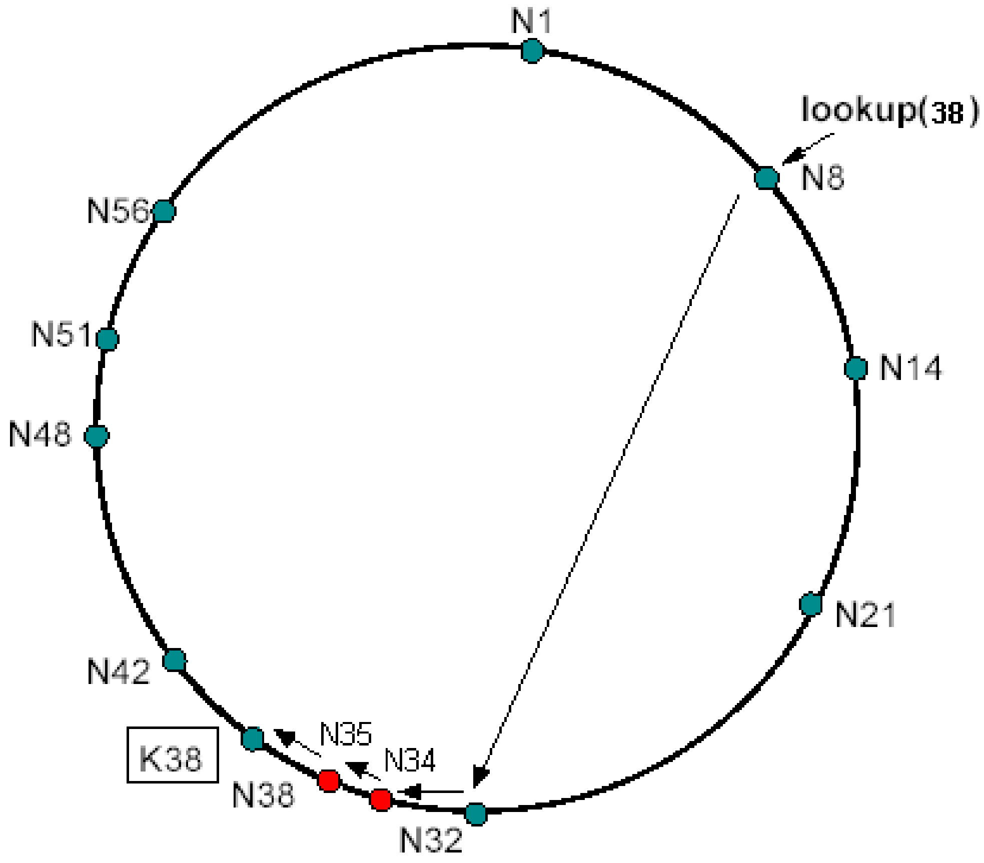


Node Departures – with Finger Tables



The Chord algorithm – Impact of node joins on lookups

- All finger table entries are correct => O(log N) lookups
- Successor pointers correct, but fingers inaccurate => correct but slower lookups



Finger table

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

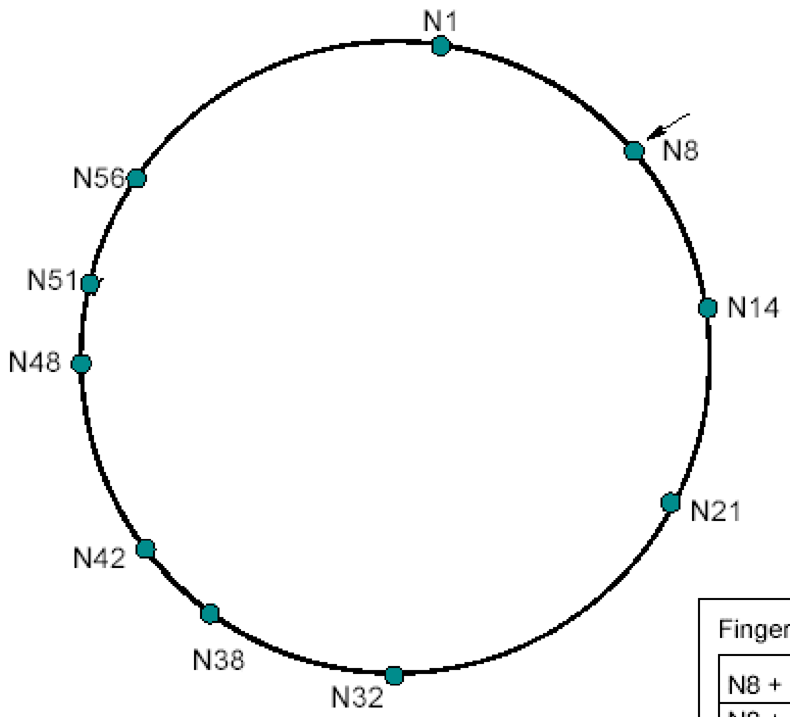
The Chord algorithm –
Impact of node joins on lookups

- Incorrect successor pointers => lookup might fail, retry after a pause
- But still correctness!

The Chord algorithm – Impact of node joins on lookups

- Stabilization completed => no influence on performance
- Only for the negligible case that a large number of nodes joins between the target‘s predecessor and the target, the lookup is slightly slower
- No influence on performance as long as fingers are adjusted faster than the network doubles in size

The Chord algorithm – Failure of nodes

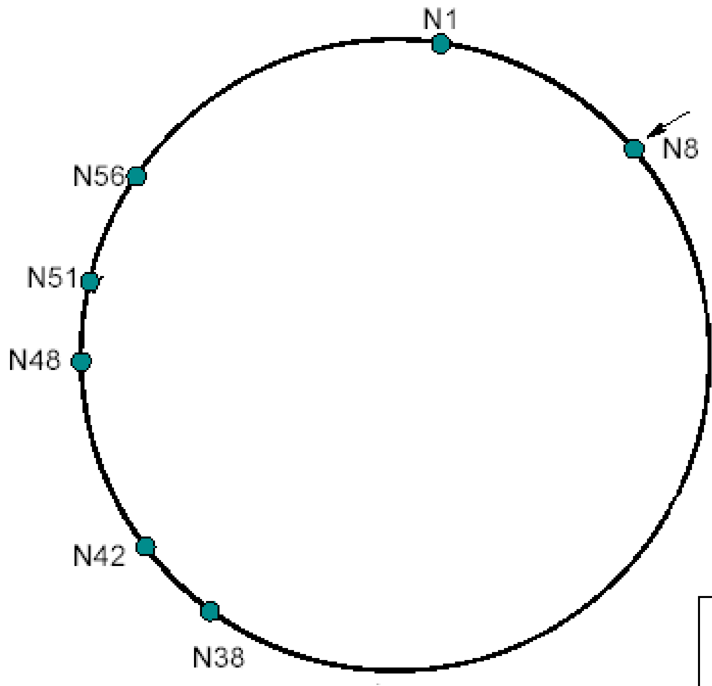


Finger table

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

- Correctness relies on correct successor pointers
- What happens, if N14, N21, N32 fail simultaneously?
- How can N8 aquire N38 as successor?

The Chord algorithm – Failure of nodes



Finger table

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

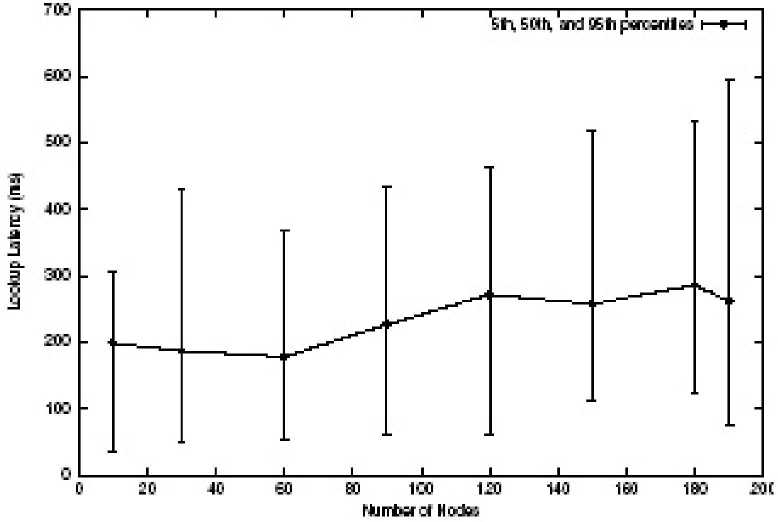
- Correctness relies on correct successor pointers
- What happens, if N14, N21, N32 fail simultaneously?
- How can N8 aquire N38 as successor?

The Chord algorithm – Failure of nodes

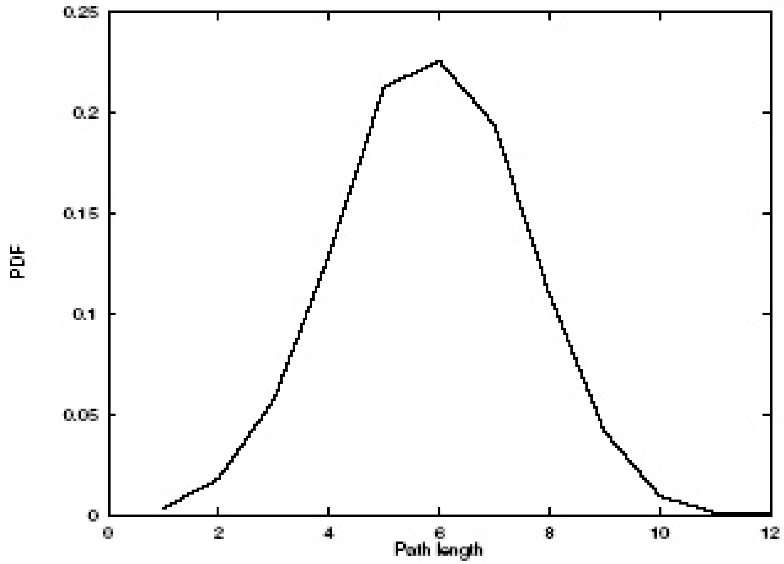
- Each node maintains a successor list of size r
- If the network is initially stable,
  and every node fails with probability ½,
  find_successor still finds the closest living successor to the query key and
  the expected time to execute find_succesor is O(log N)

Experimental Results

- Latency grows slowly with the total number of nodes

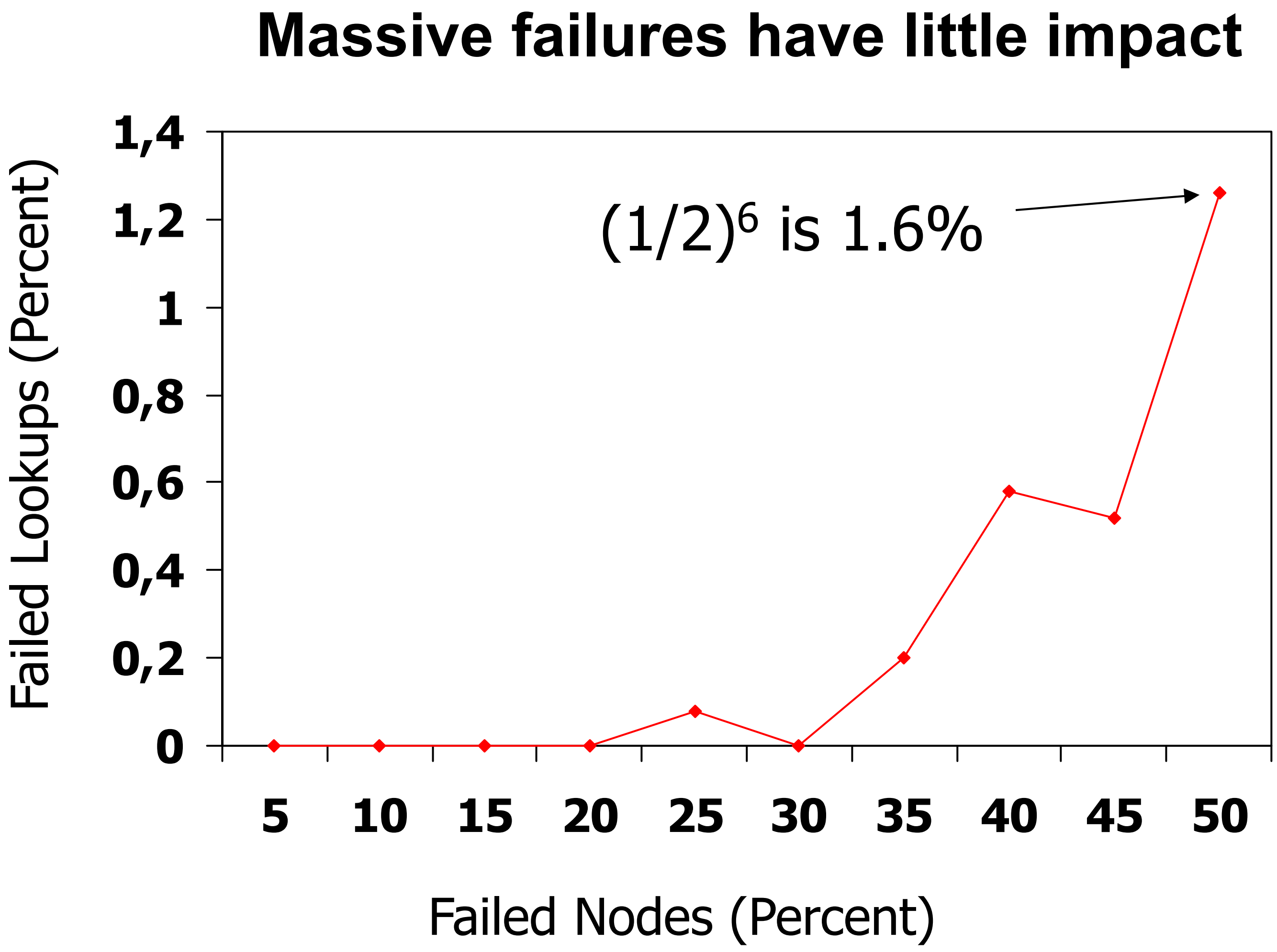


- Path length for lookups is about ½ $\log_2 N$



- Chord is robust in the face of multiple node failures

The Chord algorithm – Failure of nodes



Applications: Time-shared storage

- for nodes with intermittent connectivity (server only occasionally available)
- Store others‘ data while connected, in return having their data stored while disconnected
- Data‘s name can be used to identify the live Chord node (content-based routing)

Applications: Chord-based DNS

- DNS provides a lookup service

  keys: host names values: IP adresses

  Chord could hash each host name to a key
- Chord-based DNS:
  - no special root servers
  - no manual management of routing information
  - no naming structure
  - can find objects not tied to particular machines

Summary

- Simple, powerful protocol
- Only operation: map a key to the responsible node
- Each node maintains information about O(log N) other nodes
- Lookups via O(log N) messages
- Scales well with number of nodes
- Continues to function correctly despite even major changes of the system