# Distributed Hash Tables

SAD

# Consistent Hashing

- OK.
  - This seems fine for managed clusters
  - How about P2P?
- It turns out, it can be arranged to work on non-managed sets of servers
  - Need to find a way to implement the successor operation
    - When nobody keeps track of the full set of servers
    - In a fully distributed system
    - Distributed Hash Tables (DHT)
- Exercise: Study the Dynamo System from Amazon.
  - Describe the concrete way CH is implemented there
  - Describe performance characteristics for that implementation
  - Prepare to implement a managed consistent hashing cluster

# From Consistent Hashing to DHT: Chord

- What is Chord?
  - A P2P lookup service
    - Find where to go for a resource
  - Problem: Locate a data item in a collection of distributed nodes
    - <u>Frequent node arrivals and departures</u>
- Reminder:
  - Efficient location of resource is core to many P2P systems
- Just one main operation:
  - Map a key to a node

# DHT: Chord

- Chord goodness?
  - Simplicity
  - Provable correctness
  - Small storage overhead for search purposes
    - Each Chord node needs routing information about a small set of other nodes
  - Maintains routing information dynamically
    - As nodes join and leave the system
- Lookups by means of messages to other nodes
  - Iteratively/recursively

# DHT: Chord

- NOTE: Mapping to Nodes, not Values
  - Traditional storage maps keys to values
- Easy to do in Chrod
  - Each (key, value) pair can be stored at the key's node
- Depends on the application
  - Chord is multi-purpose
    - Only location of node responsible for a key
  - Application can store a key-value store on each node
    - Finally resolving the value for a key, after finding the node where the key-value is stored

# DHT: Chord

- We saw other approaches:
- Napster
  - Centralized Directory
    - SPoF/Dictatorship
- Gnutella et al
  - Flooding on an unstructured overlay network
    - Difficult scaling proposition
- Consistent-hashing based DHT (like Chord)
  - Scales well
  - Fully distributed: independent of any single point of control or failure

# DHT: Chord

- Let us compare to DNS

| DNS | Chord |
| --- | --- |
| provides a host name to IP address mapping | can provide same service: Name = key, value = IP |
| relies on a set of special root servers | requires no special servers |
| names reflect administrative boundaries | imposes no naming structure |
| is specialized to finding named hosts or services | can also be used to find data objects that are not tied to certain machines |

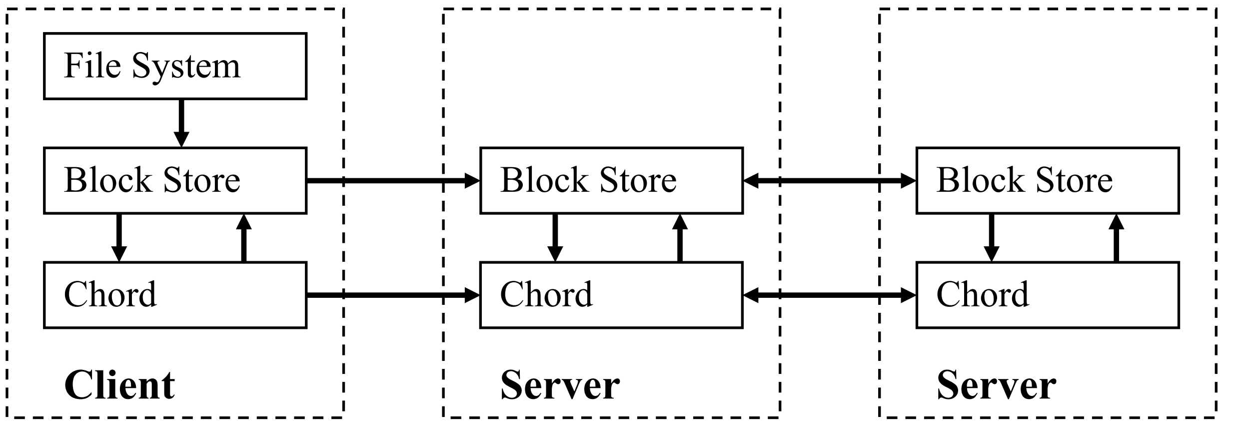
# DHT: Chord

- Addressed difficulties
  - Load Balancing
    - Spreading keys efficiently among nodes
      - Consistent hashing
  - Decentralization
    - Fully distributed algorithm
      - Non-managed
  - Scalability
    - Logarithmic growth of lookups (with number of nodes in network)
      - Feasible for large systems

# DHT: Chord

- Addressed difficulties (cont)
  - Availability
    - Automatic adjustment of internal tables
    - Goal: Ensure node responsible for a key is always found
  - Flexible naming
    - No structure imposed on keys
      - Key space is flat
    - Mapping application "names" to Chord keys is up to the application
      - Chord is just enabling middleware

# DHT: Chord: Example




- Highest layer frovides a filesystem-like interface
  - User-friendly names, authentication, etc
- Middle layer converts to block operations
- Block storage uses Chord
  - Identify node responsible for storing block
    - Then Block Store can talk to storage server on the holding node

# DHT: Chord

- Base protocol main tasks:
  - Specify how to find the locations of keys
    - E.g., how lookup, the main operation, is performed
  - Specify how to join new nodes to the system
    - How the structures are updated to maintain correct lookup
  - Specify what to do on planned node departures
    - Same as before: how are structures modified
  - Specify what to do on failures
    - Unplanned departures

# DHT: Chord

- Consistent hashing is used as a base, with its properties

# DHT: Chord

- Consistent hashing is used as a base, with its properties
- SUMMARY of benefits

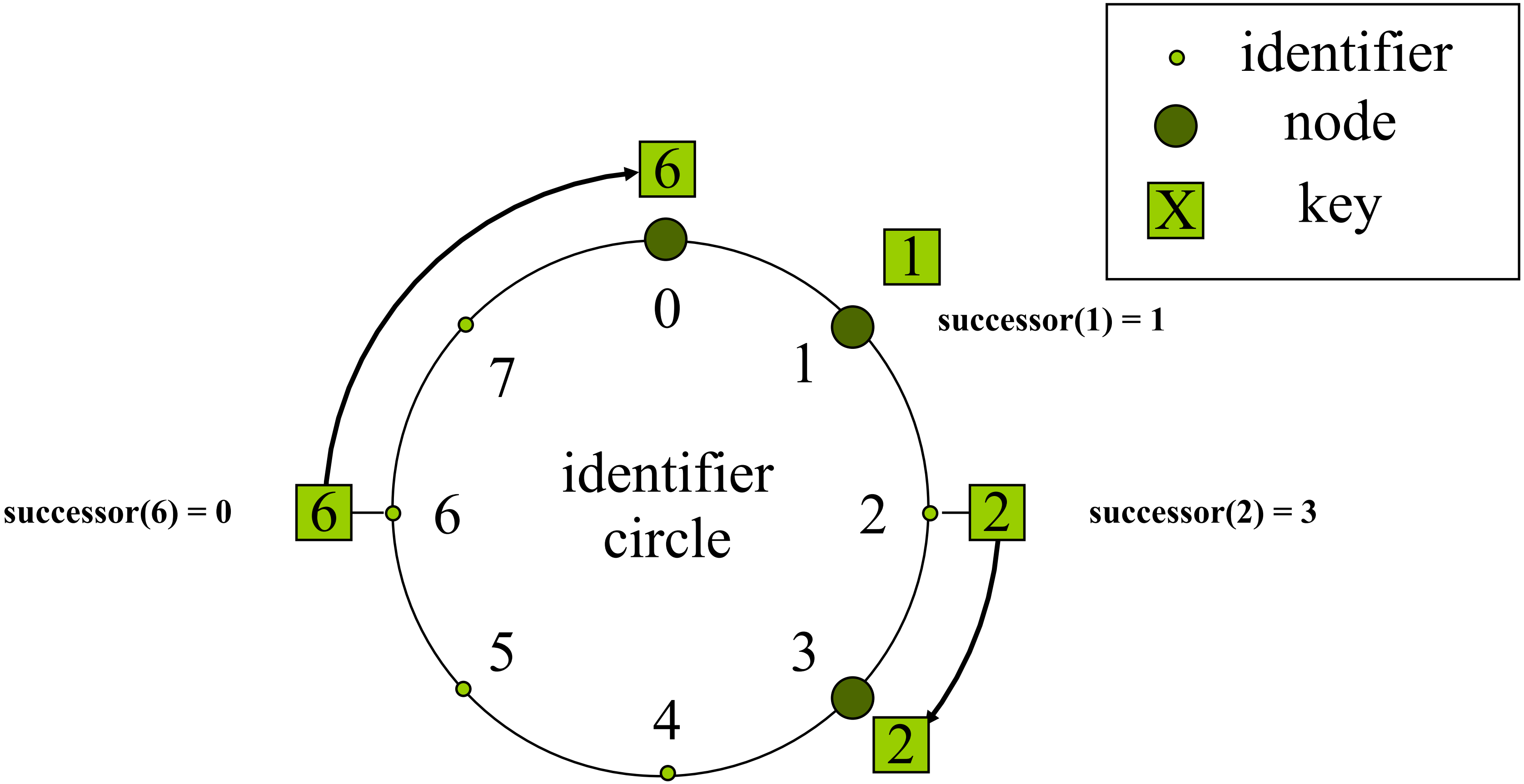
- Hash function assigns each node *and* key an m-bit *identifier* using a base hash function such as SHA-1
  - ID(node) = hash(IP, Port)
  - ID(key) = hash(key)
- Properties of consistent hashing:
  - Function balances load: all nodes receive roughly the same number of keys – good?
  - When an Nth node joins (or leaves) the network, only an O(1/N) fraction of the keys are moved to a different location

# DHT: Chord

- Consistent hashing is used as a base, with its properties

# DHT: Chord

- Consistent hashing is used as a base, with its properties



# DHT: Chord

- In a distributed setting we need to route messages:
  - Need to store extra information to do so
    - A very small amount of routing information is necessary to implement consistent hashing
- Each node need only be aware of its successor node in the ring
- Queries for a given identifier can be passed around the circle via these pointers
- Is this efficient?

# DHT: Chord

- In a distributed setting we need to route messages:
  - Need to store extra information to do so
    - A very small amount of routing information is necessary to implement consistent hashing
- Each node need only be aware of its successor node in the ring
- Queries for a given identifier can be passed around the circle via these pointers
- What order of complexity does this have?

# DHT: Chord

- In a distributed setting we need to route messages:
  - Need to store extra information to do so
    - A very small amount of routing information is necessary to implement consistent hashing
- Each node need only be aware of its successor node in the ring
- Queries for a given identifier can be passed around the circle via these pointers
- What order of complexity does this have?
  - Resolution is correct but inefficient
    - May require traversing all nodes
      - O(N)

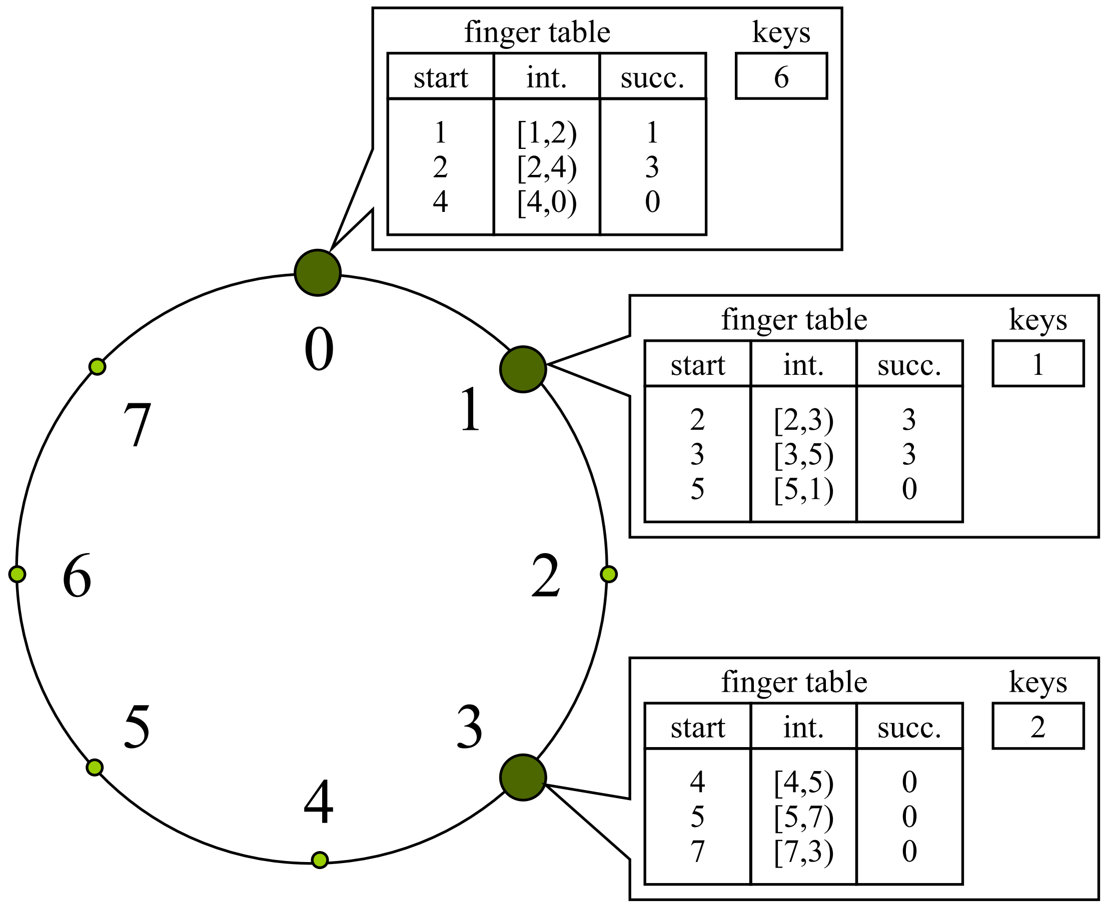
# DHT: Chord

- Will need extra information if we want to speedup the lookups
  - Didn’t we say we could use some sort of a tree?...
  - How to do so in a distributed setting?

# DHT: Chord

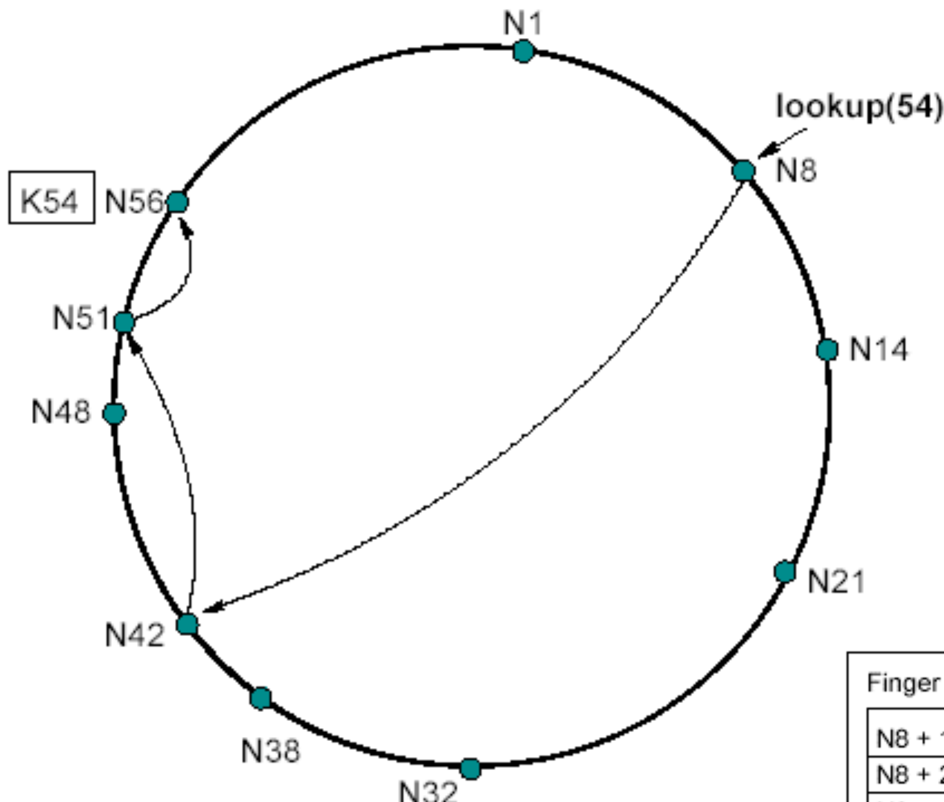
- Will need extra information if we want to speedup the lookups
  - Didn't we say we could use some sort of a tree?...
  - How to do so in a distributed setting?
- Each node maintains a routing table with $\leq m$ entries
  - **Finger Table**
  - $N = 2^m$ or $m = \log N$
- Small data structure even for large N
- For any node n, and entry i of its Finger Table, we have
  - $FT_i^n = successor(n + 2^{2-1}) = s$
  - S is called the $i^{th}$ finger of node n.
    - n.finger(i).node

# DHT: Chord


finger table
keys
start
int.
succ.
6
1 2 4
[1,2) [2,4) [4,0)
1 3 0
finger table
keys
start
int.
succ.
1
2 3 5
[2,3) [3,5) [5,1)
3 3 0
finger table
keys
start
int.
succ.
2
4 5 7
[4,5) [5,7) [7,3)
0 0 0
0
1
2
3
4
5
6
7

21

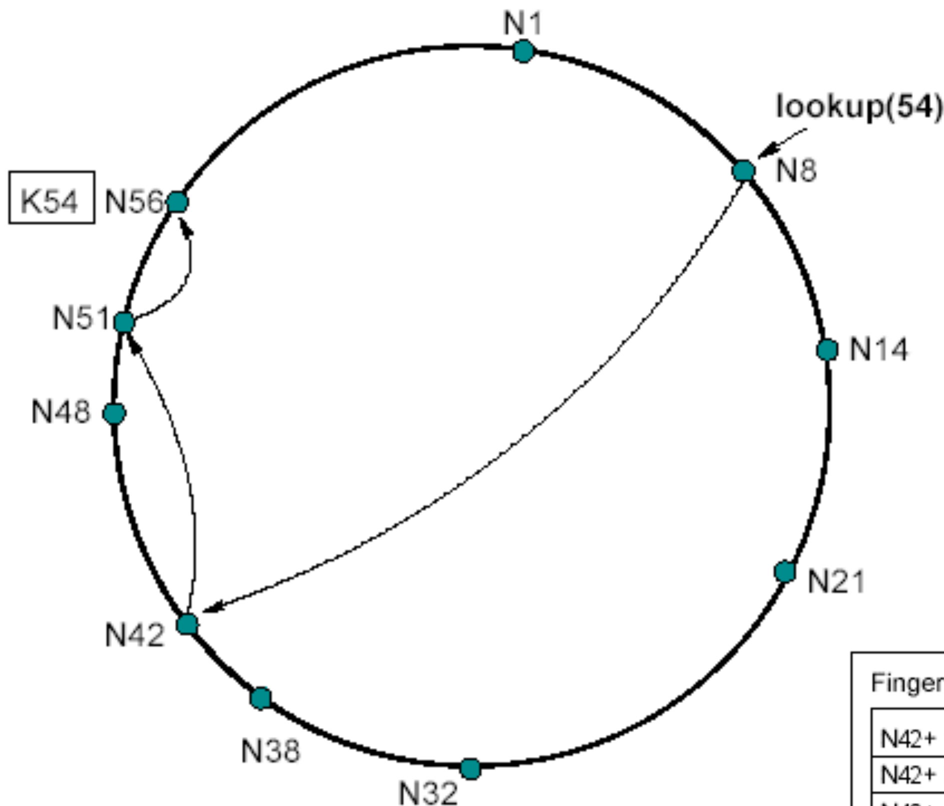
# DHT: Chord location




Finger table

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

# DHT: Chord location



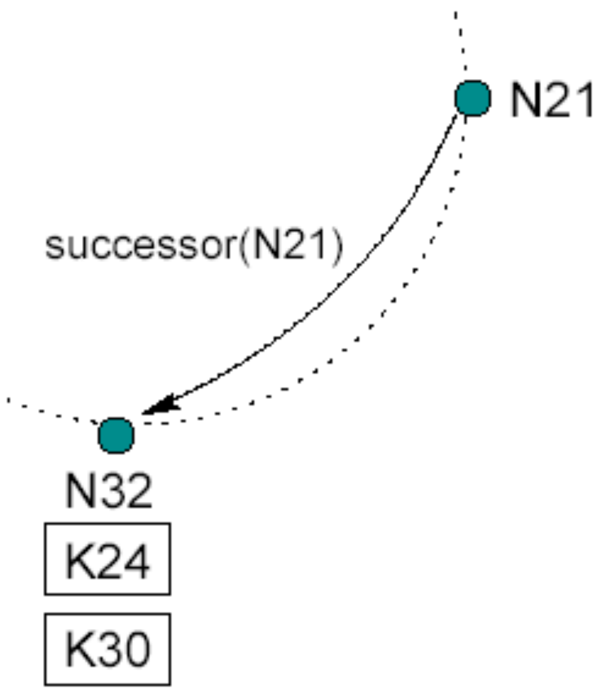

Finger table

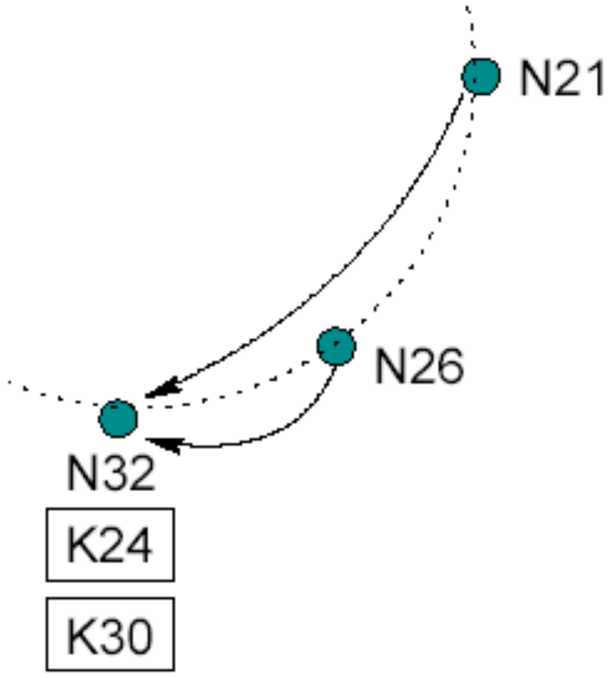
| | |
|---|---|
| N42+ 1 | N48 |
| N42+ 2 | N48 |
| N42+ 4 | N48 |
| N42+ 8 | N51 |
| N42+16 | N1 |
| N42+32 | N14 |

# DHT: Chord

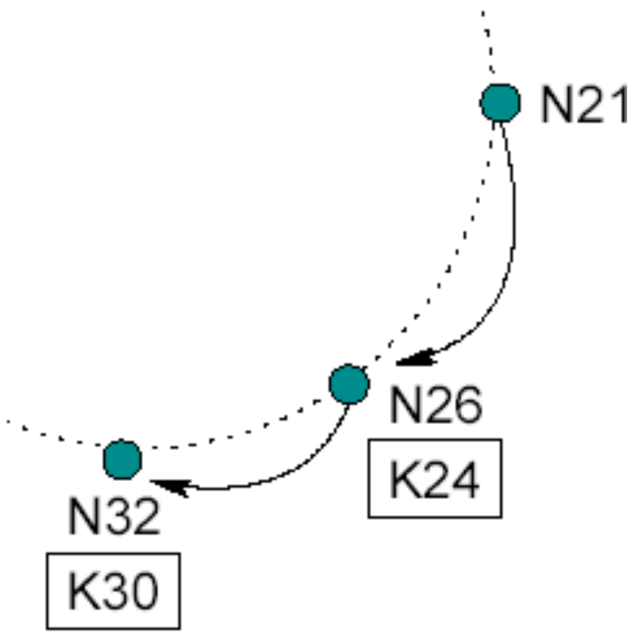
- Thus…

- Each node stores info about a small number of other nodes
  - Knows more about nearby nodes than about far away nodes
- Finger Table in a node has insufficient info to determine a successor of a key by itself
  - Repetitive queries are the key (no  pun intended)
    - To nodes that immediately precede the given key
    - Eventually reach successor(key)


N21
successor(N21)
N32
K24
K30



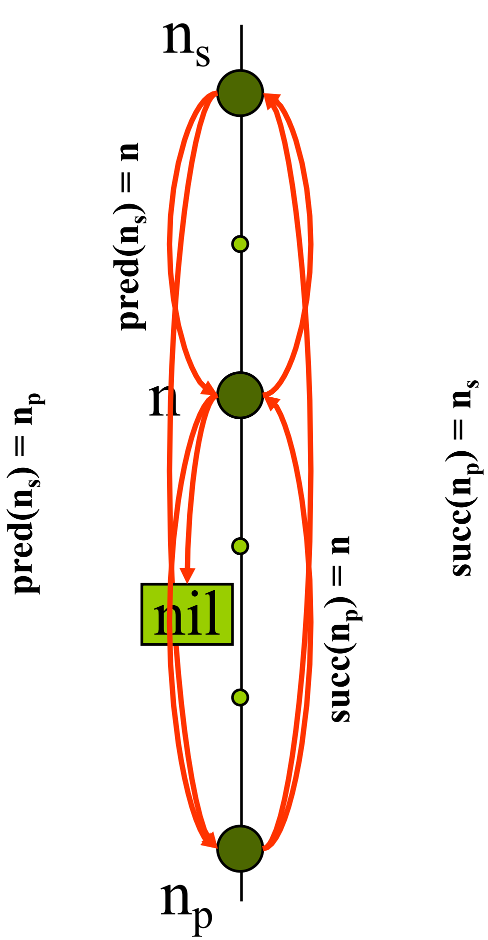
N21
N26
N32
K24
K30


N21
N26
K24
N32
K30

# DHT: Chord

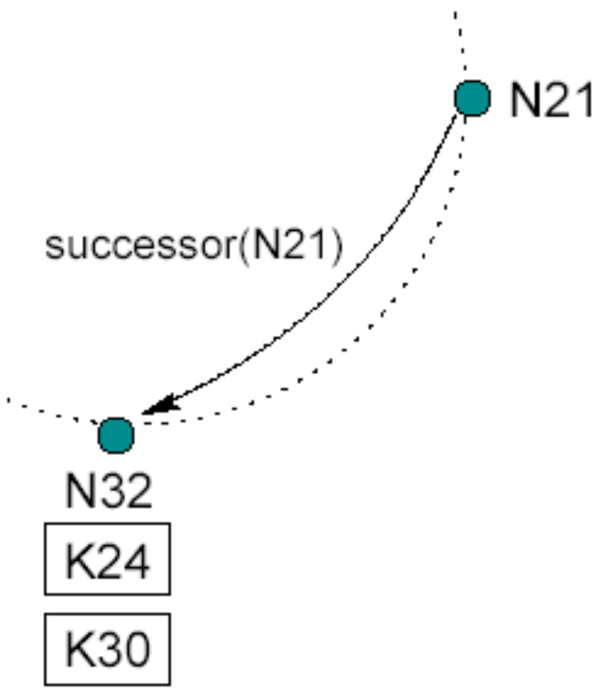
- Basic “stabilization” protocol
  - Used to keep nodes’ successor pointers up to date
    - Sufficient to guarantee correctness of lookups
- Successor pointers can be used to verify finger table entries
- Every node runs *stabilize*
  - Periodically
  - To find new joined nodes

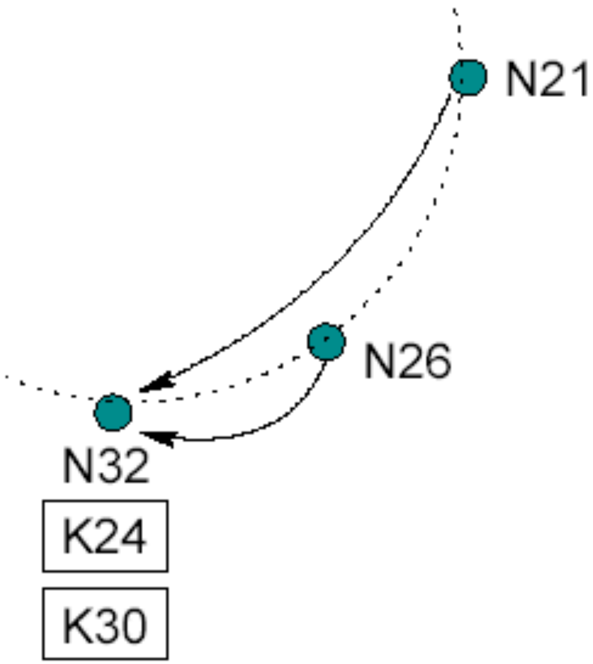
# Chord: Stabilization after Join



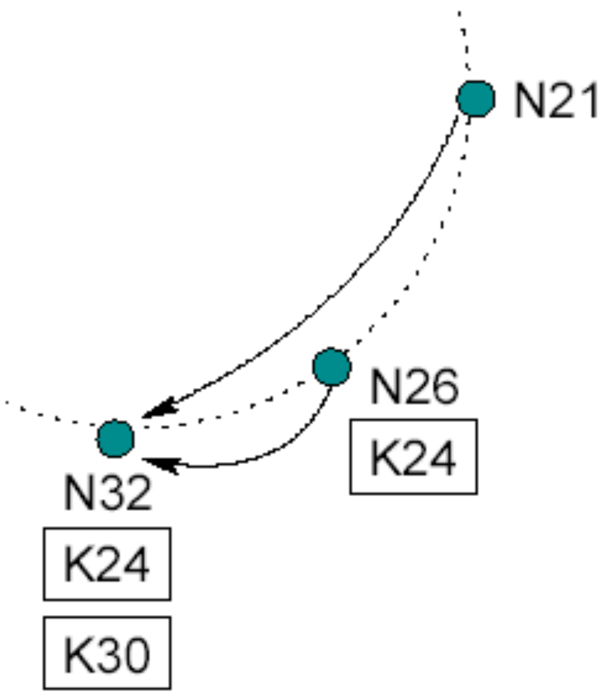

- **n joins**
  - predecessor = nil
  - n acquires $n_s$ as successor via some n'
  - n notifies $n_s$ being the new predecessor
  - $n_s$ acquires n as its predecessor
- **$n_p$ runs stabilize**
  - $n_p$ asks $n_s$ for its predecessor (now n)
  - $n_p$ acquires n as its successor
  - $n_p$ notifies n
  - n will acquire $n_p$ as its predecessor
- **all predecessor and successor pointers are now correct**
- **fingers still need to be fixed, but old fingers will still work**

# Chord: Stabilization



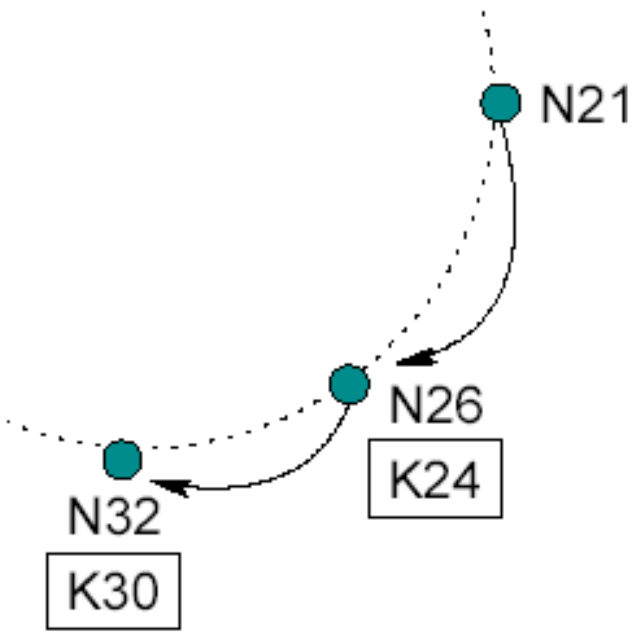



- N26 joins the system
- N26 aquires N32 as its successor
- N26 notifies N32
- N32 aquires N26 as its predecessor



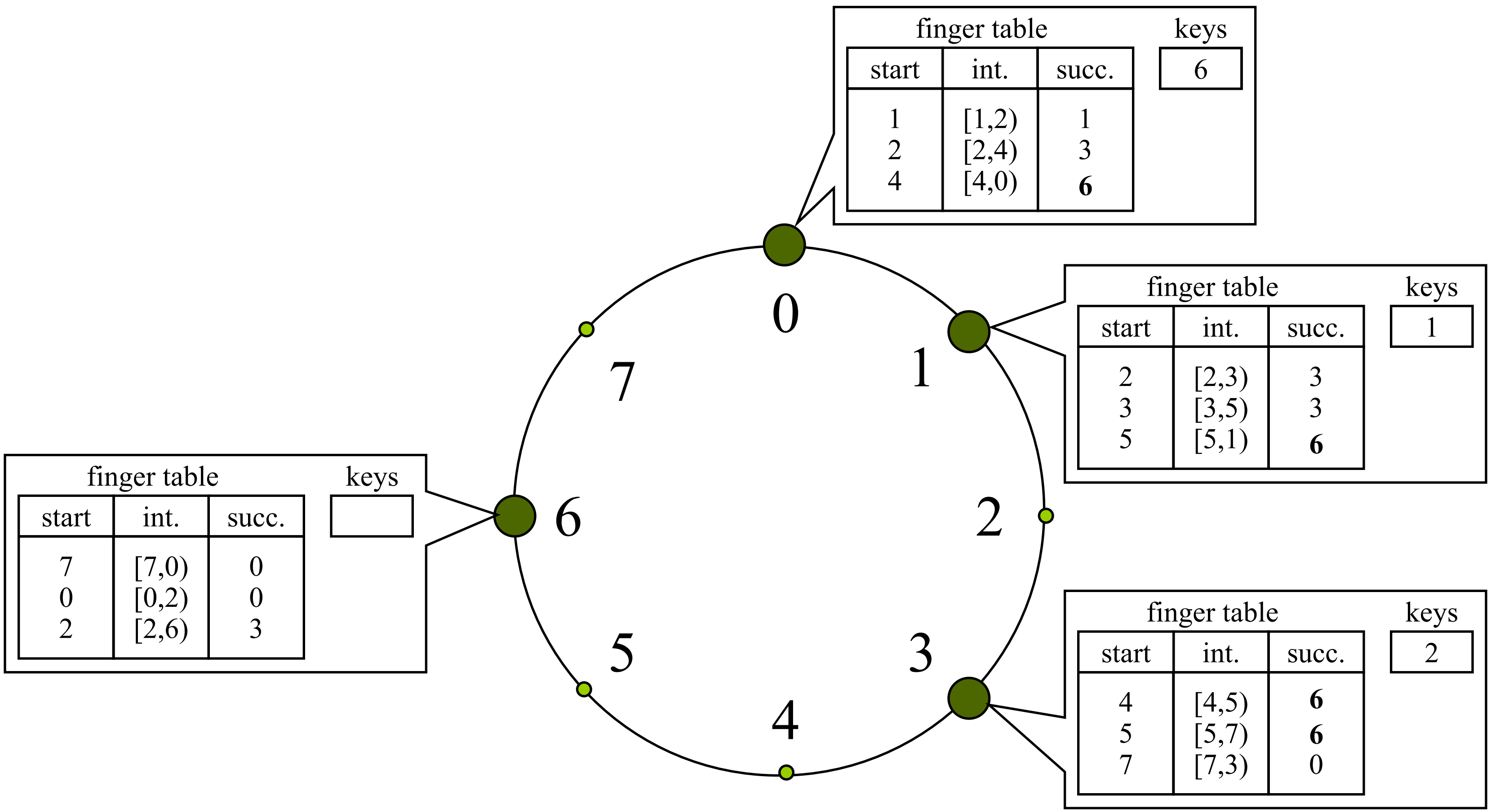

• N26 copies keys

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

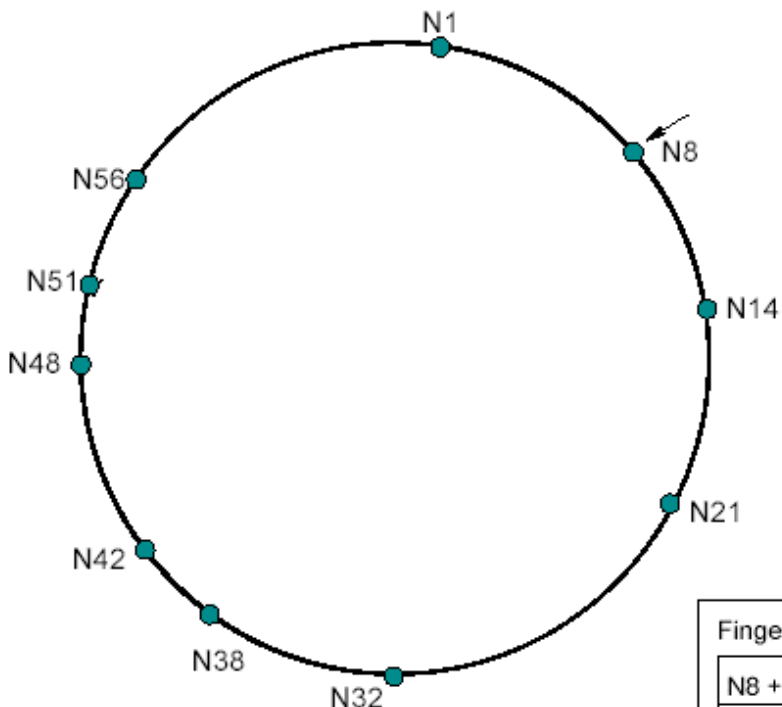



- N21 aquires N26 as its successor
- N21 notifies N26 of its existence
- N26 aquires N21 as predecessor

# Chord: Node joins

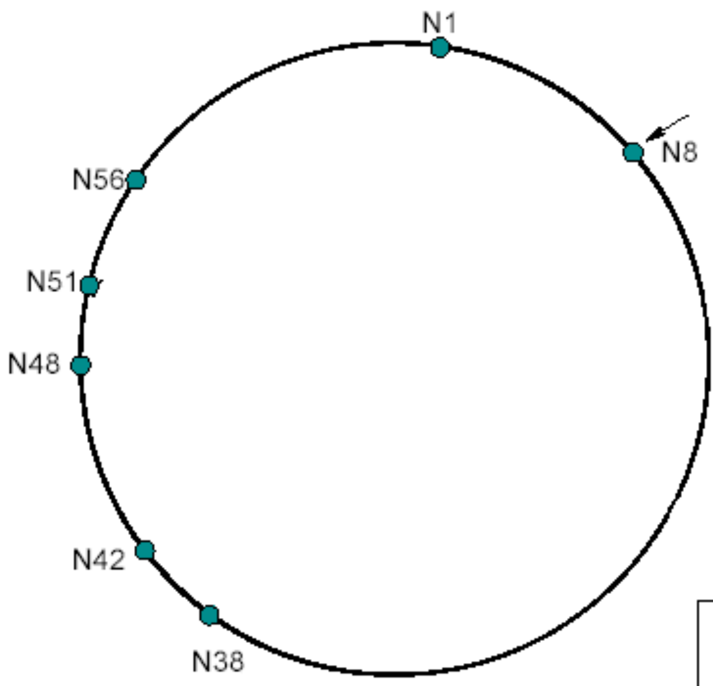


- Correctness relies on correct successor pointers
- What happens, if N14, N21, N32 fail simultaneously?
- How can N8 aquire N38 as successor?




Finger table

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




Finger table

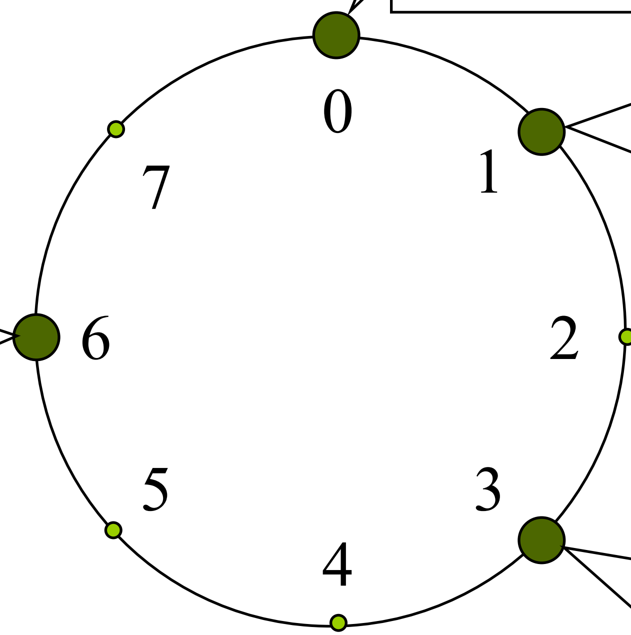
| | |
|---|---|
| N8 + 1 | N14 |
| N8 + 2 | N14 |
| N8 + 4 | N14 |
| N8 + 8 | N21 |
| N8 +16 | N32 |
| N8 +32 | N42 |

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

# DHT: Chord

- Failures. What do we do about them?
  - Key: Maintain correct successor pointers
- Each node maintains a *successor-list* of its r nearest successors on the ring
  - If node *n* notices its successor has failed…
    - It substitutes it with the first live entry in the list
  - *Stabilize* will correct finger table entries pointing to the failed node
    - Also *successor-list* entries
- Performance will depend on ratio of frequencies
  - Node joins/leaves
  - Invocation of stabilization protocol

# Chord: Node departures




finger table

| start | int. | succ. |
|---|---|---|
| 1 | [1,2) | **3** |
| 2 | [2,4) | 3 |
| 4 | [4,0) | 6 |

keys

finger table

| start | int. | succ. |
|---|---|---|
| 2 | [2,3) | 3 |
| 3 | [3,5) | 3 |
| 5 | [5,1) | 6 |

keys: 1

finger table

| start | int. | succ. |
|---|---|---|
| 4 | [4,5) | 6 |
| 5 | [5,7) | 6 |
| 7 | [7,3) | 0 |

keys: 2

finger table

| start | int. | succ. |
|---|---|---|
| 7 | [7,0) | 0 |
| 0 | [0,2) | 0 |
| 2 | [2,6) | 3 |

keys: 6

# DHT: Chord

- Some numbers
  - As per Consistent Hashing
    - Each node takes care of K/N keys on average
    - O(K/N) keys are relocated for joins/leaves
  - Lookups need O(log N) messages
    - Thanks to the finger table
  - Reestablishing routing info and finger tables requires
    - $O(\log^2 N)$ messages