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

Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek Hari Balakrishnan

About Chord

- Protocol and algorithm for distributed hash table (DHT)
- Structured-decentralized P2P overlay network

About Chord

- Supports just one operation: lookup(key)
  - Given a key, it maps the key onto a node
  - Returns the IP-address of the node
- Associate a value with each key and store the pair on the resulting node:
  - Distributed hash table
- Uses *consistent hashing* for this operation

Chord properties (1)

- Load balancing:
  - spreads keys evenly over nodes
- Decentralized:
  - fully distributed, all nodes equally important
  - does not take into account heterogeneity of nodes

Chord properties (2)

- Scalability:
  - lookup cost: O(log(N))
- Availability:
  - automatically updates lookup tables as nodes join, leave, fail
- Flexible naming:
  - Flat key-space, no constraint on structure of keys

Traditional hashing

- Given an object, assign it to a bin from a set of N bins
- Each bin: roughly same amount of objects
- Problem:
  - If N changes, all objects need to be reassigned

Consistent hashing

- Evenly distributes K objects across N bins, about K/N each
- When N changes:
  - Not all objects need to be moved
  - Only O(K/N) need to
- Uses a base hash function, such as SHA-1

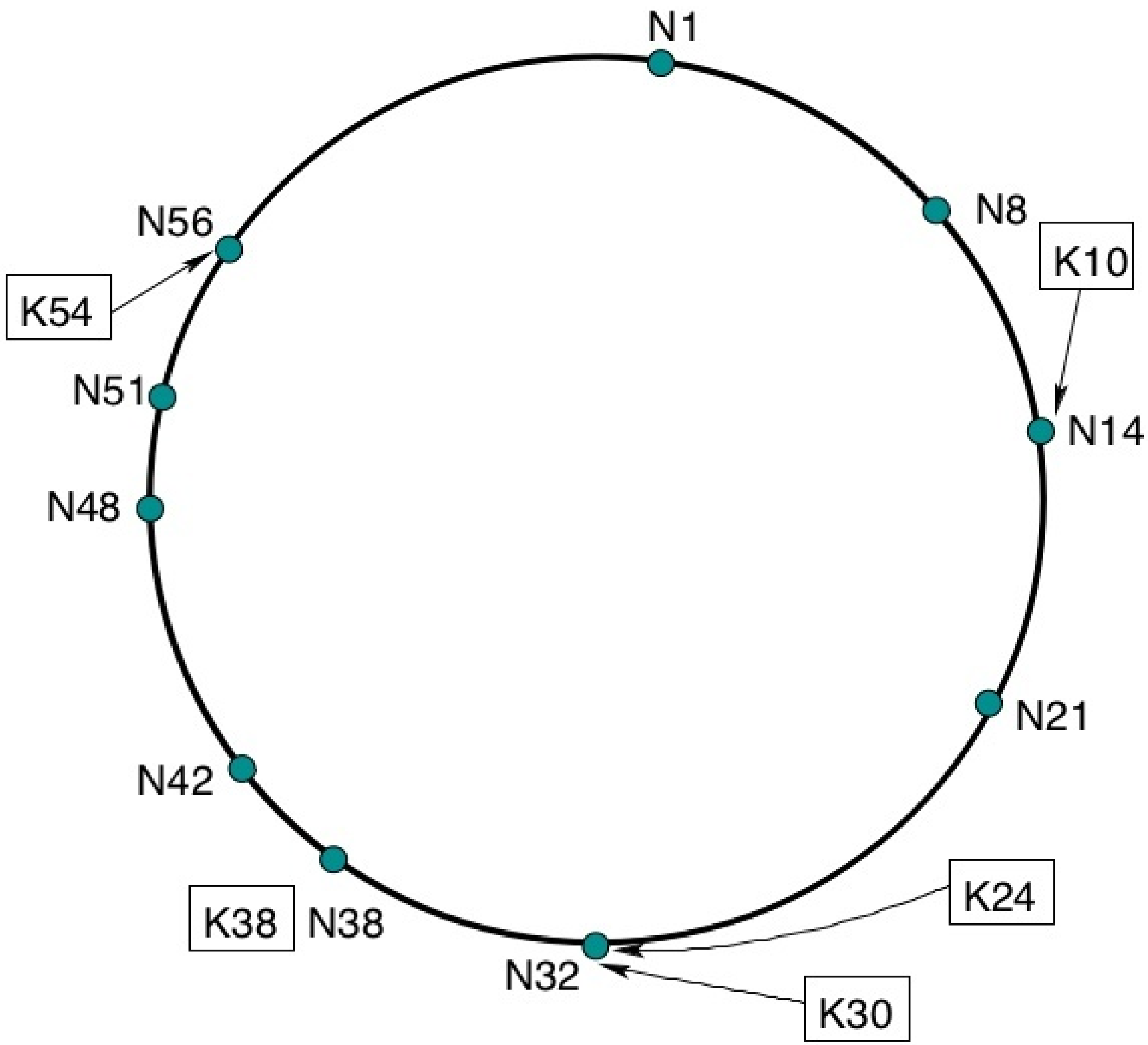
Chord protocol

- Assign each *node* and *key* an m-bit *identifier* using SHA-1:
  - node: hash the IP-address
  - key: hash the key
- Identifiers are ordered on an *identifier circle* module 2^m, the *Chord ring*:
  - circle of numbers from 0 to 2^{m-1}

Assigning keys to nodes

- Assigning key *k* to a node:
  - first node whose identifier is equal to or follows the identifier of key *k* in the identifier space
  - successor node of *k*: *successor(k)*
  - first node clockwise from *k* on *Chord ring*

Chord ring (m=6)



Minimal disruption

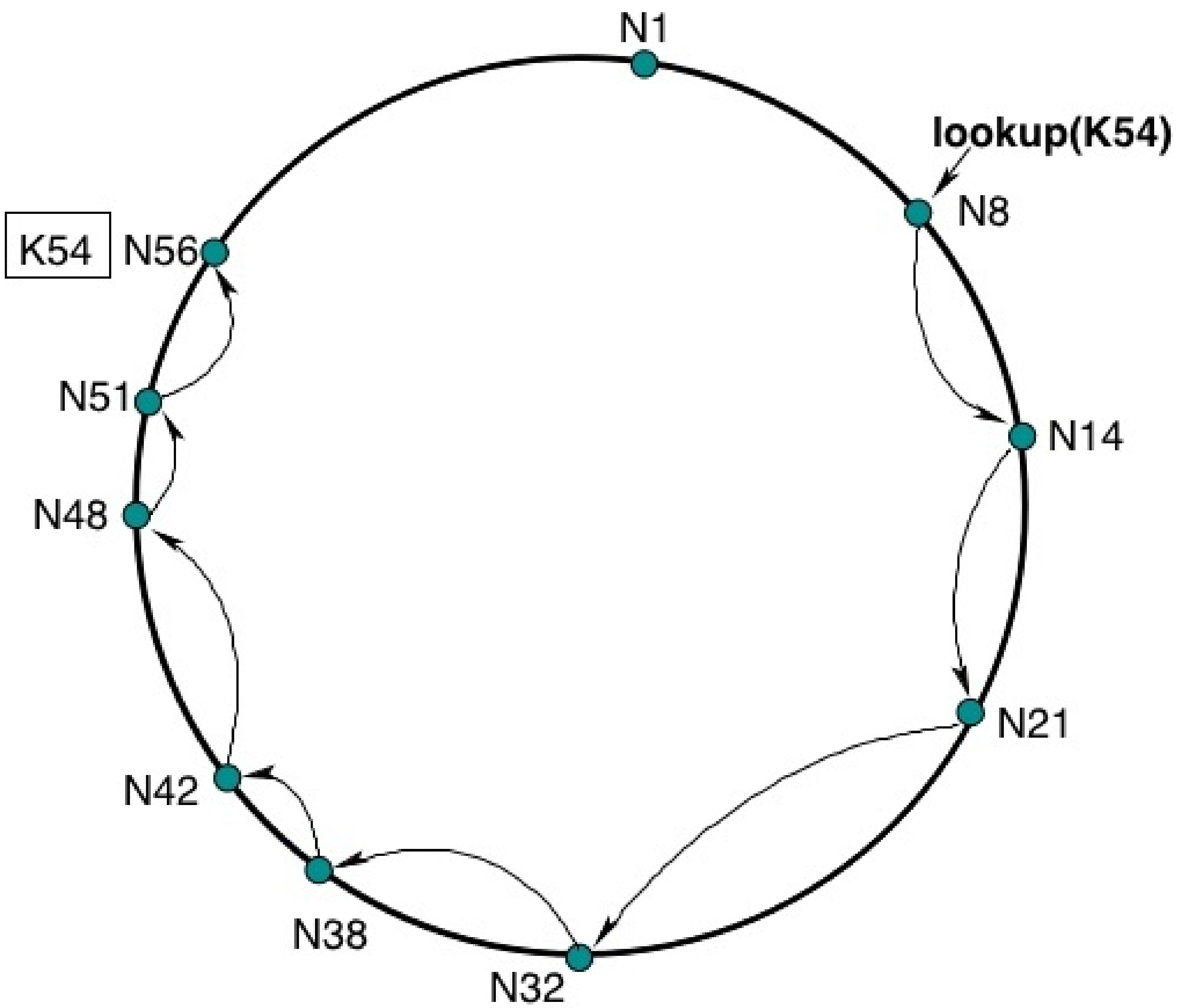
- Minimal disruption when node joins/leaves
- *n* joins:
  - certain keys assigned to *successor(n)* are now assigned to *n*
- *n* leaves:
  - all keys assigned to *n* are now assigned to *successor(n)*

Simple Chord lookup

- Simplest Chord lookup algorithm:
  - Each node only needs to know its successor on Chord ring
  - Query for identifier *id* is passed around Chord ring until *successor(id)* is found
  - Result returned along reverse path
- Easy but not efficient: # messages O(N)

Simple Chord lookup

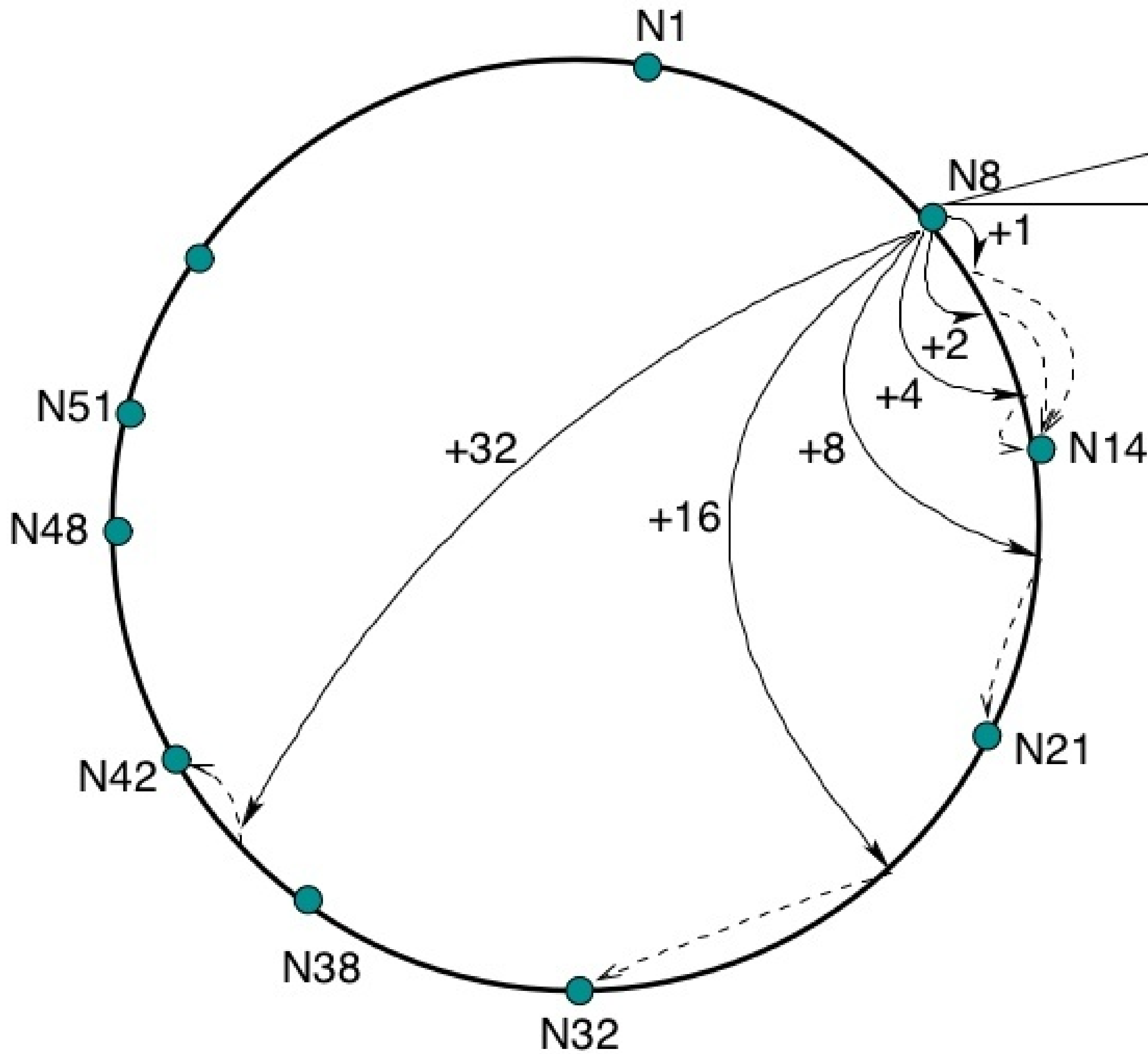
```
// ask node n to find the successor
// of id
n.find_successor(id)
  if (id ∈ (n, successor])
    return successor;
  else
    // forward the query around the
    // circle
    return successor.find_successor(id);
```



Finger table

- Chord maintains additional routing info:
  - not essential for correctness
  - but allows acceleration of lookups
- Each node maintains *finger table* with *m* entries:
  - n.*finger*[i] = *successor*(n + 2^{i-1})

Finger table



Finger table

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

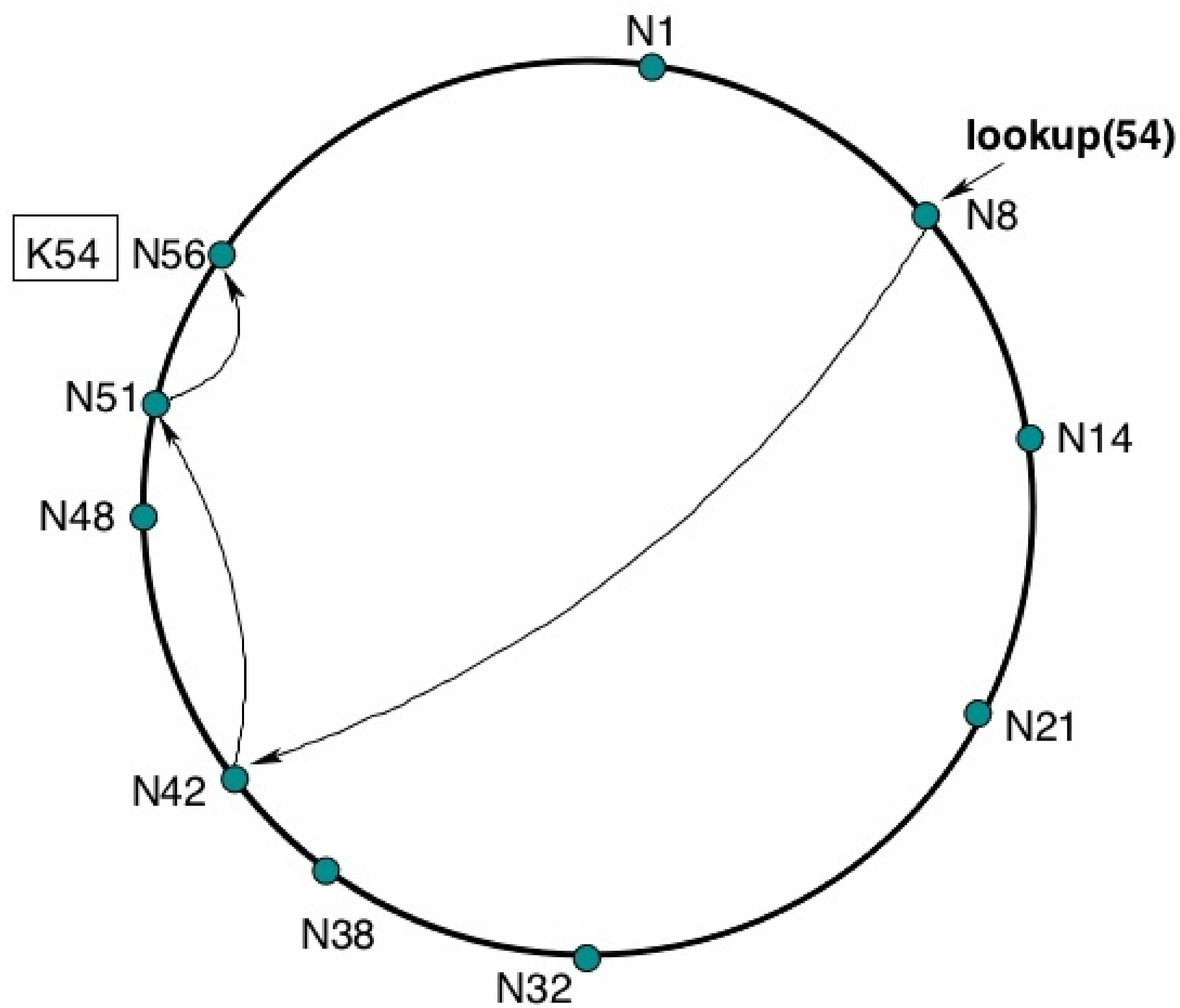
Improved lookup

- Lookup for *id* now works as follows:
  - If id falls between *n* and *successor(n)*, *successor(n)* is returned
  - Otherwise, lookup is performed at node *n'*, which is the node in the finger table of *n* that most immediately precedes *id*
- Since each node has finger entries at power of two intervals around the identifier circle, each node can forward a query at least halfway along the remaining distance between the node and the target identifier.
- Thus O(log(N)) nodes need to be contacted

Chord protocol

```
// ask node n to find the successor
// of id
n.find_successor(id)
  if (id ∈ (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] ∈ (n, id))
      return finger[i];
  return n;
```



Coping with joining/leaving

- Essential that successor pointers stay up to date
- Each node periodically runs *stabilization protocol* which updates Chord's successor pointers and finger tables
- For node *n* to join, it must know a node *n'* of the Chord ring

Coping with joining/leaving

- To join, n performs **join**()
  - node *n* asks *n'* to **find_successor**(n), which *n* sets as its sucessor
- Each node periodically performs **stabilize**() to learn about newly joined nodes
  - *n* asks his successor *s* for *s*' predecessor *p*
  - decides whether *p* should be *n*'s successor instead

Coping with joining/leaving

- **stabilize**() also notifies *n*’s successor *s* of *n*’s existence
  - *s* might change it’s predecessor to *n*
- Each node periodically calls **fix_fingers**()
  - makes sure fingers are up to date
  - is how new nodes acquire finger table
  - is how old nodes add new nodes to their tables

Coping with joining/leaving

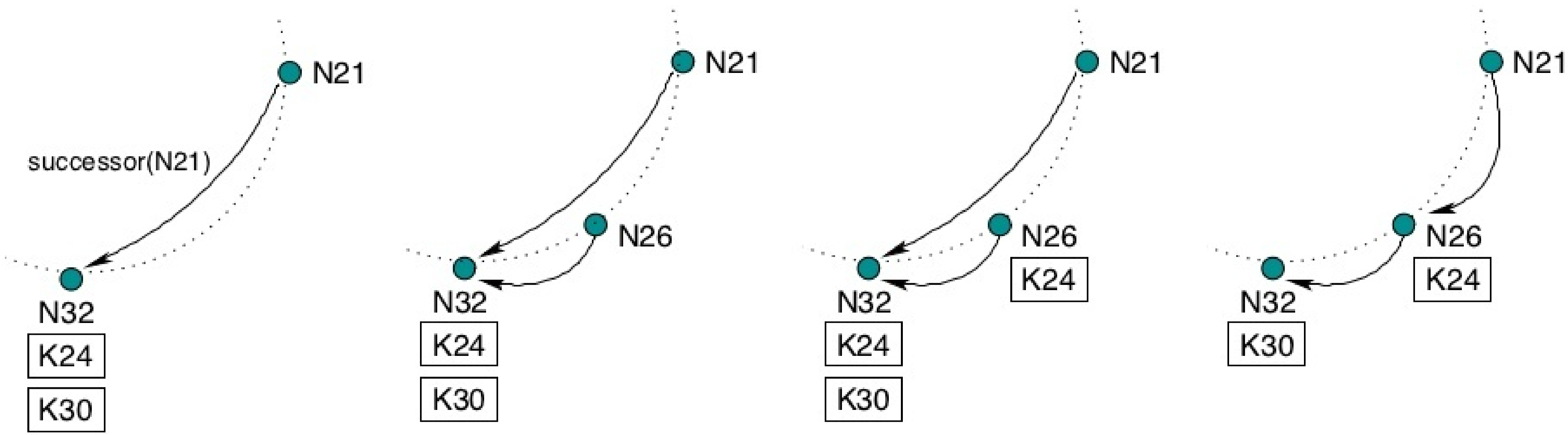
```
//join a Chord ring containing node n'.
n.join(n')
  predecessor = nil;
  successor = n'.find_successor(n);

// called periodically. verifies n's
// immediate successor, and tells the
// successor about n.
n.stabilize()
  x = successor.predecessor;
  if (x ∈ (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';


// 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 = 1;
  finger[next] = find_successor(n+2^(next-1));
```

Coping with joining/leaving



Coping with joining/leaving

- Lookup will only fail if successor pointers are not up to date => retry after a pause
- As soon as successor pointers are correct, lookup will work
  - over time, finger tables will be up to date and linear search is no longer needed
- Old finger tables can still be used and in general lookup will remain O(log(N)) even if not all finger tables are up to date

Successor list

- Lookup fails if a node doesn't know its correct successor
  - can occur when nodes fail
- Solution: each node maintains a *successor list*:
  - contains the node's first *r* successors
  - if first successor does not respond, try the rest of the list

Replication

- *Successor list* can also be used for *replication*
  - application on top of Chord might store replicas of data associated with a key on *k* nodes from *n*'s successor list
  - Chord can inform application when this successor list changes, so application can create new replicas

Chord in practice

- In practice, Chord ring will never be in stable state:
  - Joins and leaves occur continuously
  - Interleaved by stabilization algorithm
  - Not enough time to stabilize before new change
- If stabilization is run at a certain rate:
  - Chord ring will be continuously “almost stable” and lookup will be fast and correct

Virtual nodes

- Chord load balancing can be improved through *virtual nodes:*
  - node *id*s do not uniformly cover entire *id* space
  - to make (# keys / node) more uniform, associate keys with *virtual nodes*
  - map *r virtual nodes*, with unrelated *id*s, to each *real node*
- Tradeoff: each real node needs *r* times as much space to store finger table

Network latency

- Chord path length is O(log(N)), but latency can be quite large
  - Nodes close in *id* space can be far away in the underlying network
- Idea:
  - choose next-hop finger based on progress in *id* space and latency in network
    - maximize progress in *id* space
    - minimize latency

Network latency

- Different approach to improving latency:
  - each entry in finger table can point to list of nodes instead of a single node
  - nodes have similar *id*s and are equivalent for routing purposes
  - latency of the nodes could be different
  - use closest node in terms of network distance in the underlying network

Applications of Chord

- Chord File System (CFS):
  - Chord locates storage blocks
- Distributed indexes:
  - solves Napster's central server issue
- Large-scale combinatorial search:
  - candidate solutions are keys, Chord maps these to machines which test them
- Cooperative mirroring:
  - Uses Chord to provide load balancing
- Time-shared storage:
  - Uses Chord to provide availability

?