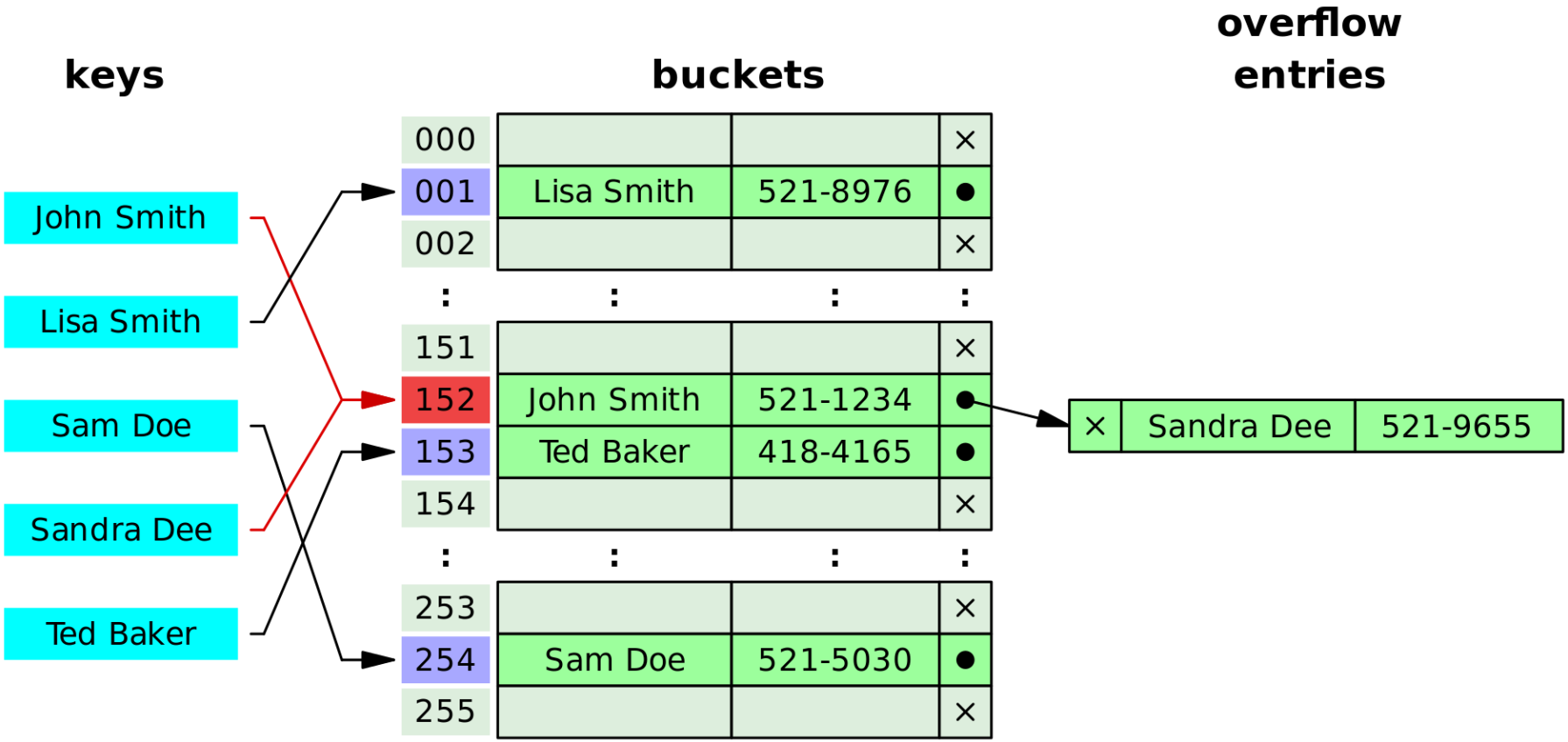
Peer-to-peer (p2p) systems

- Idea: create distributed systems out of individually owned, unreliable machines in possibly different administrative domains
  - Peers make a portion of their resources available in exchange for consuming (usually more) resources
  - Real-world projects like this exist, e.g., SETI@home and Folding@home
    - Trivial programs from a parallel programming viewpoint; mostly computation, very little communication
    - This has been tried with nontrivial parallel programs, often called “Grid Computing”, to little success
- In p2p systems, the primary problem is *lookup*
  - given a data item stored at one or more places, find it

p2p lookup problem

- Possible approaches
  - Centralized server (Napster)
    - Problem: scalability, single point of failure
  - Query Flooding (Gnutella)
    - Problem: inefficiency
  - Unstructured (Freenet)
    - Problem: not finding the object

Review: Hash Tables



By Jorge Stolfi - Own work, CC BY-SA 3.0,
https://commons.wikimedia.org/w/index.php?curid=6472274

DHT: Distributed Hash Tables

- Provides a lookup service that is similar to a hash table
  - Hash table maps a key to a bucket
  - With DHT, we want to translate a key to a node
    - And then to the data (within the node)
  - Each node can communicate with only a small number of nodes
- Goals of DHT:
  - Load Balance, Decentralization, Scalability, Fault Tolerance (nodes leaving and joining)

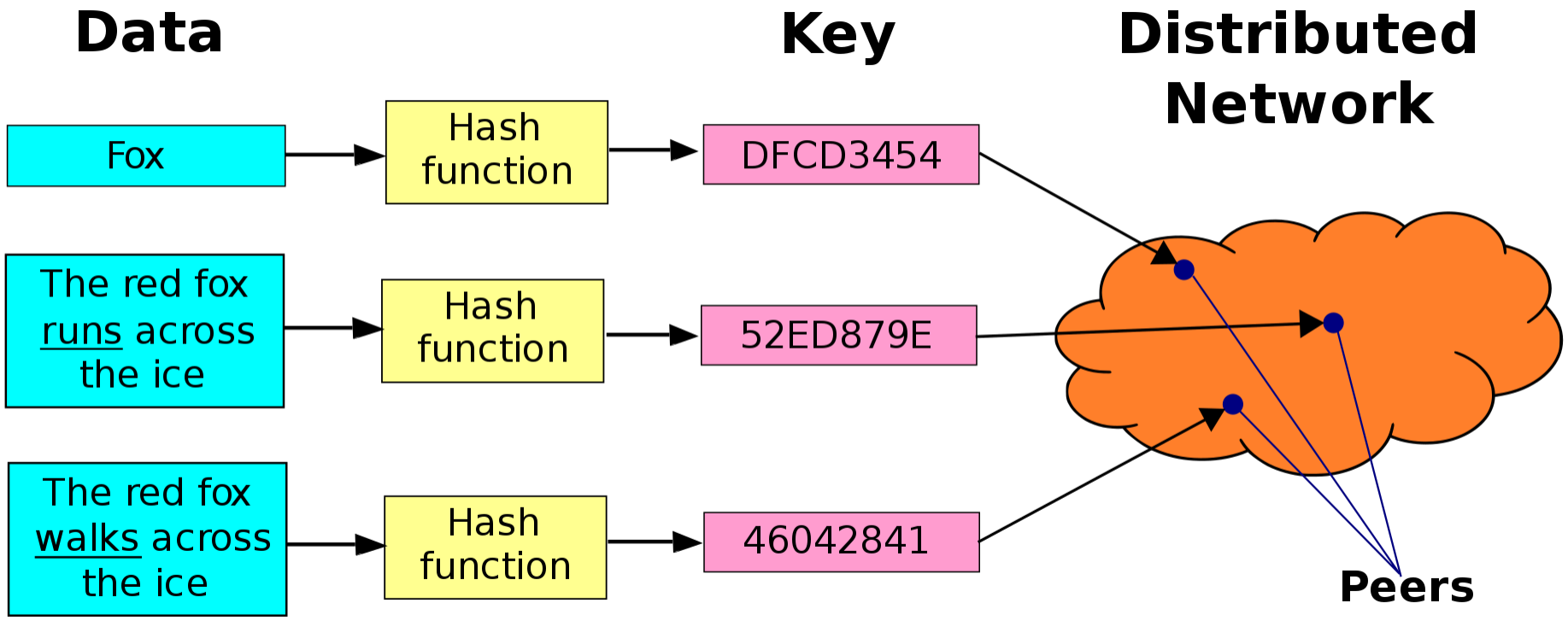
DHT Uses

- Some applications use/used DHTs as their fundamental underlying structure
  - Distributed file systems
  - Content distribution networks
  - Cooperative caching
  - Fast, in-memory distributed databases

DHT Operations

- There are only two operations in a DHT:
  - *put(k, d)*: Given a key *k* and data *d*, stores *d* on a node that is determined by *k*
    - Where did the key come from?
      - key *k* may be obtained by hashing all or part of *d*
    - Hash function needs to spread keys out evenly between the nodes
  - *get(k)*: Retrieves *d* from the proper node

Picture of Distributed Hash Table



By Jnlin - Own work, Public Domain
https://commons.wikimedia.org/w/index.php?curid=1585652

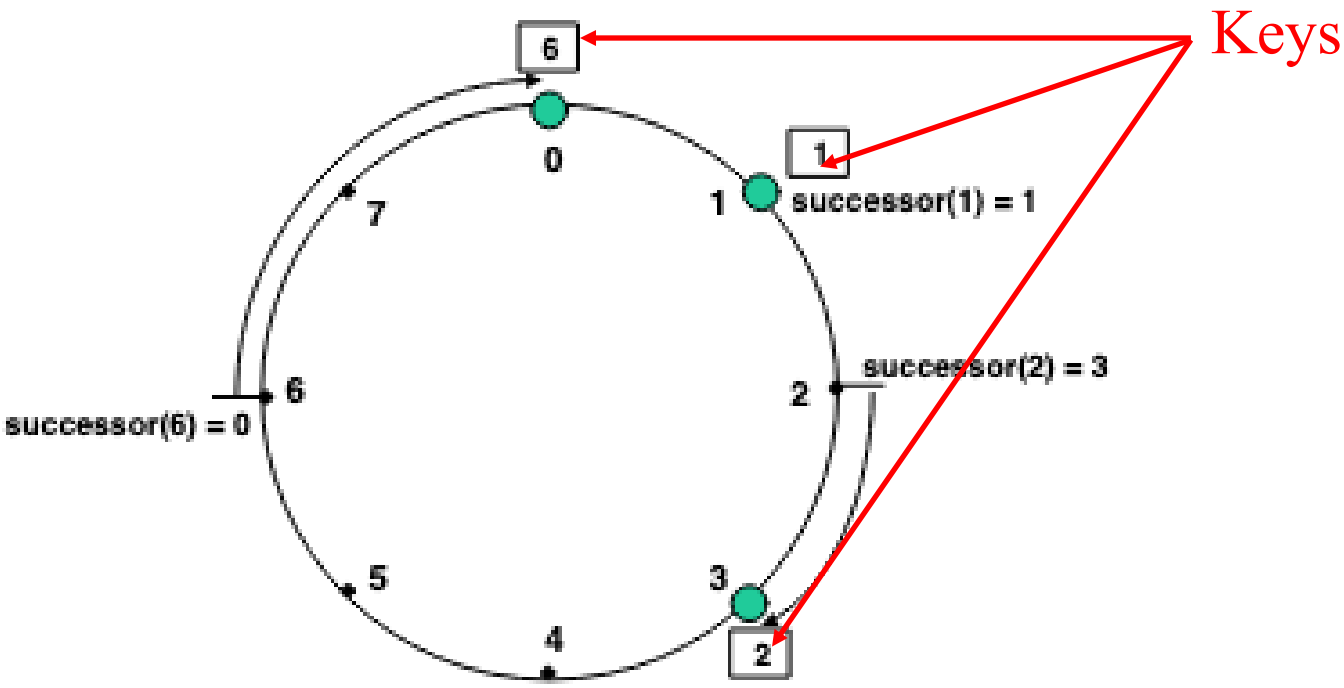
DHT Issues

- Load balance: need to distribute the data evenly
- Forwarding algorithm: how does a node move a request closer to the node that has the data
  - Nodes only know about a small percentage of the nodes in the system
- Fault tolerance
  - Handling joins, departures, and failures

Example DHT: Chord

(Stoica et al, SIGCOMM 2001; picture from paper)

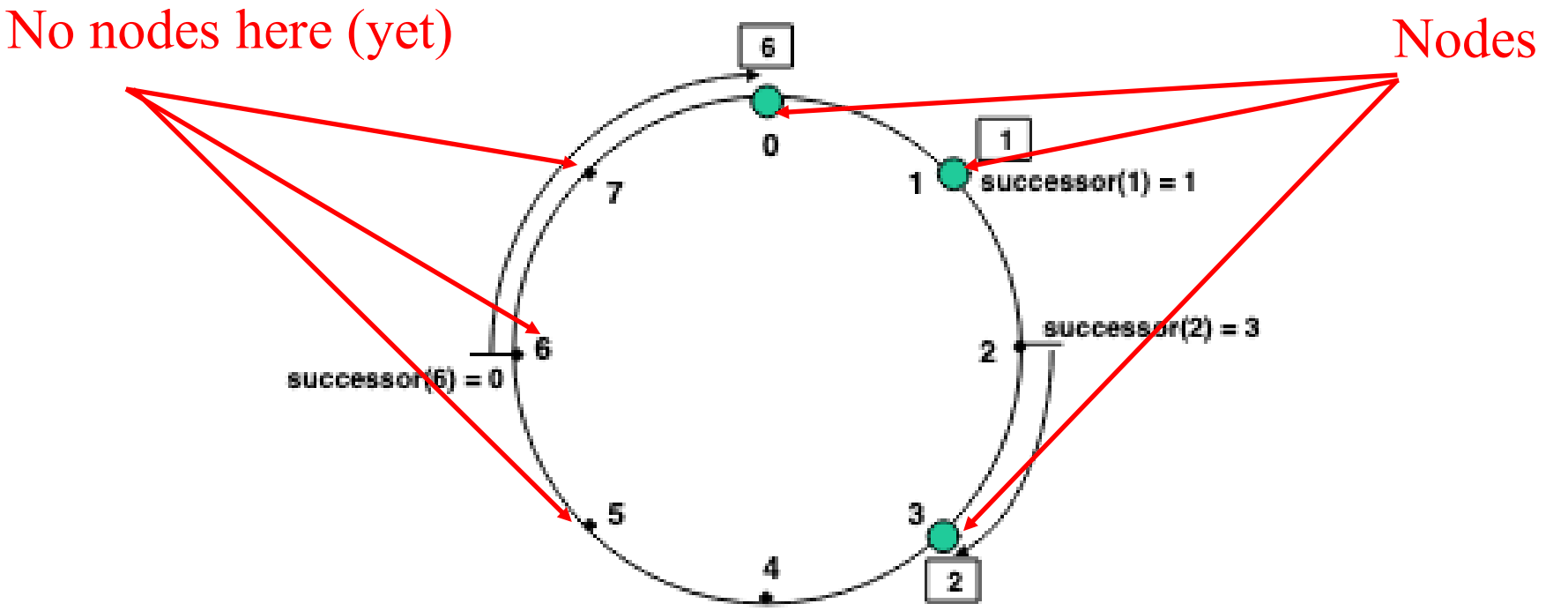
- Nodes form a logical circle in order of node id
  - Assume given a mapping from node id to IP address



Example DHT: Chord

(Stoica et al, SIGCOMM 2001; picture from paper)

- Nodes form a logical circle in order of node id
  - Assume given a mapping from node id to IP address



Function *successor(key)* produces the node on which a key is stored
--defined as the smallest node id greater than or equal to the key

Example DHT: Chord
(Stoica et al, SIGCOMM 2001)

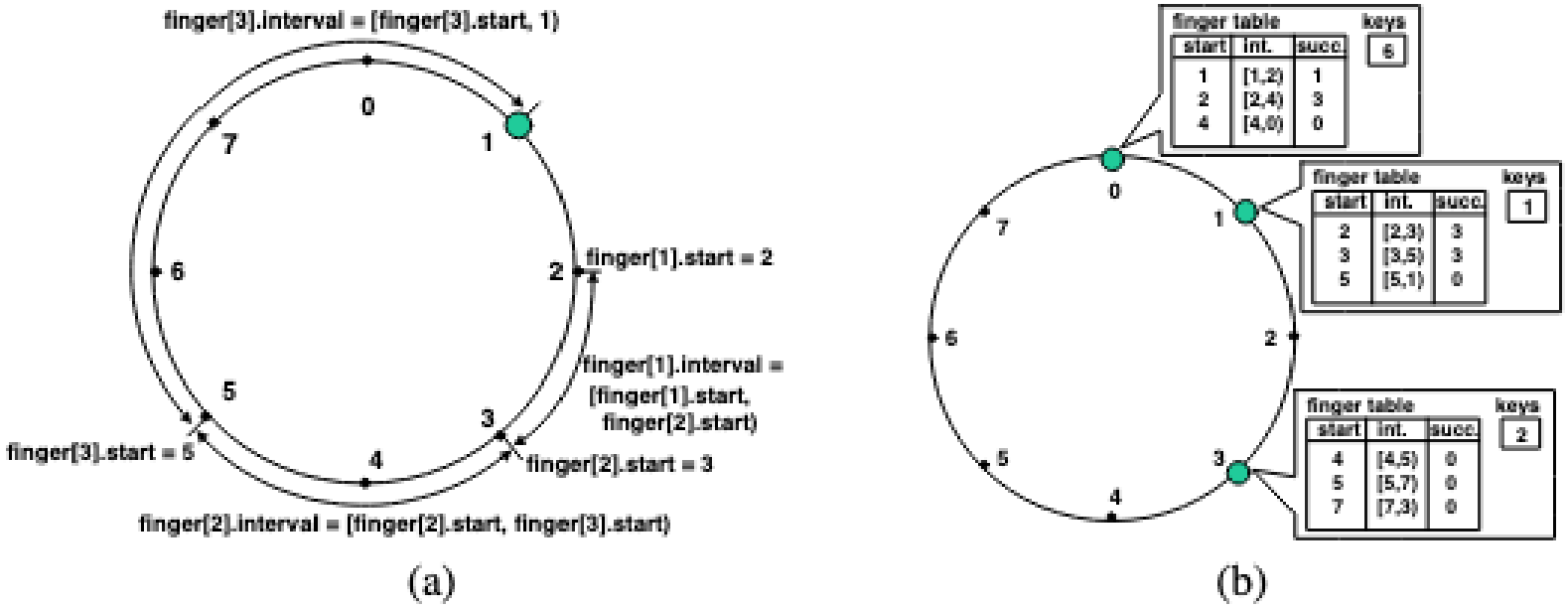
- Naïve forwarding would simply pass requests around the ring
  - Stop when the key is stored on a node
  - This is not scalable---finding a key is linear time in the number of nodes

Example DHT: Chord
(Stoica et al, SIGCOMM 2001)

- In Chord, each node has its own "skiplist", which contains the node half-way around the circle, quarter-way around the circle, etc.
  - Skiplist is called a *finger table* in Chord
- On a *put* or *get*, forward to node in skiplist that has the highest id not exceeding *k*
  - Results in O(log *N*) search time if there are *N* nodes
  - But what if there is a failure in a skiplist node?
    - Discussed later

Pictures of Chord, cont.

Taken from Stoica et al, SIGCOMM 2001



Left: finger intervals for node 1

Right: finger tables/key locations for nodes 0, 1, 3; and keys 1, 2, 6

The "succ." field indicates the node to forward to for a given interval

-- if this field is larger than the key, that is the node storing the key

-- not the same as the successor function applied to keys

Finding a node for a key in Chord

- To find a node for key *k*, starting at node *n*:

```
m = n
while (1) {
    i = interval in m's finger table that contains k
    if i.succ >= k then
        return i.succ
    else
        m = i.succ
}
```

How Chord achieves its properties

- Load balance: achieved by hashing the node's IP address to get the node id, and by hashing the data to get the key
  - Assumes a hash function with no collisions and an even distribution
- Decentralization: no node is more important than any other
- Scalability: skiplist makes lookups O(log *N*)
- Fault tolerance: next slide

Fault Tolerance in Chord

- If a node fails, searches cannot just fail while system is stabilizing
  - Finger table may produce a node that isn't working
  - To combat this, each node keeps a few nearby successors (in addition to the nodes in the finger table)
  - Effective as long as at least one of these successors is alive

Adding a node in Chord

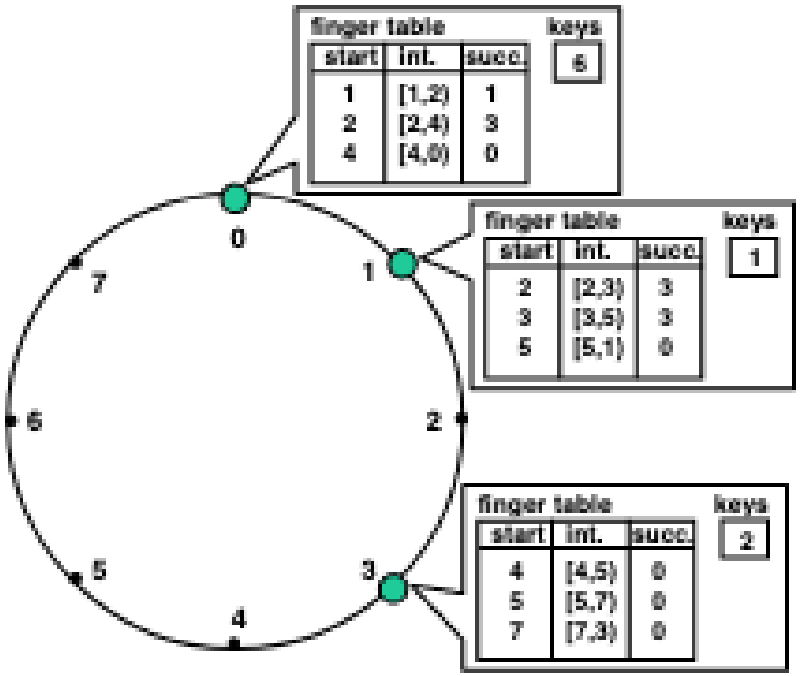
- To add a new node *i* (assumes *i* is unique)
  - Initialize predecessor, successors, and skiplist for node *i*
  - Update skiplist, successors, and predecessor of existing nodes
  - Move appropriate keys/data to this node

Removing a new node is symmetric

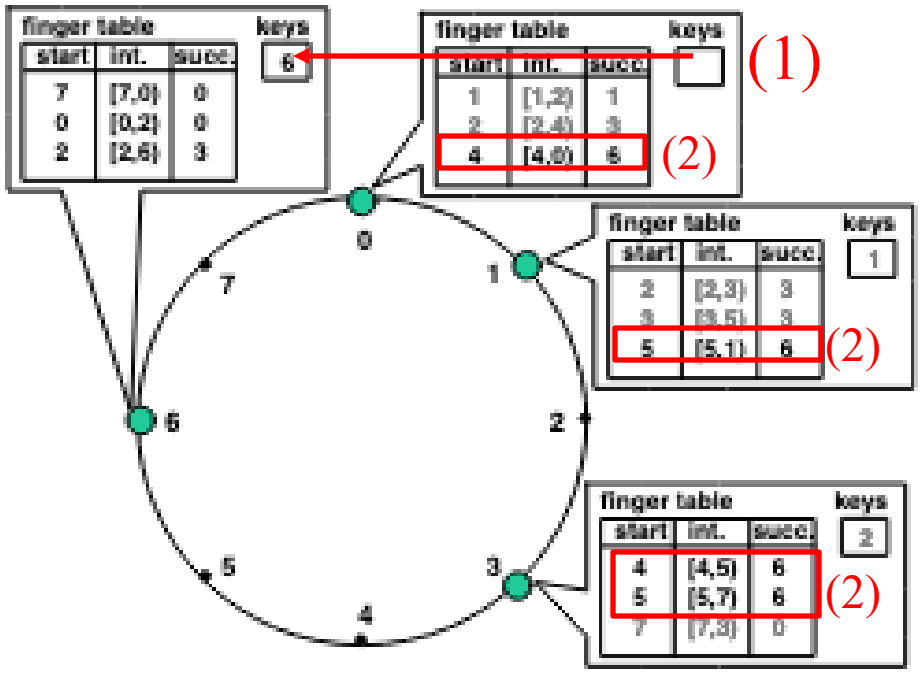
Adding a node in Chord

Taken from Stoica et al, SIGCOMM 2001

Before adding node 6



After adding node 6

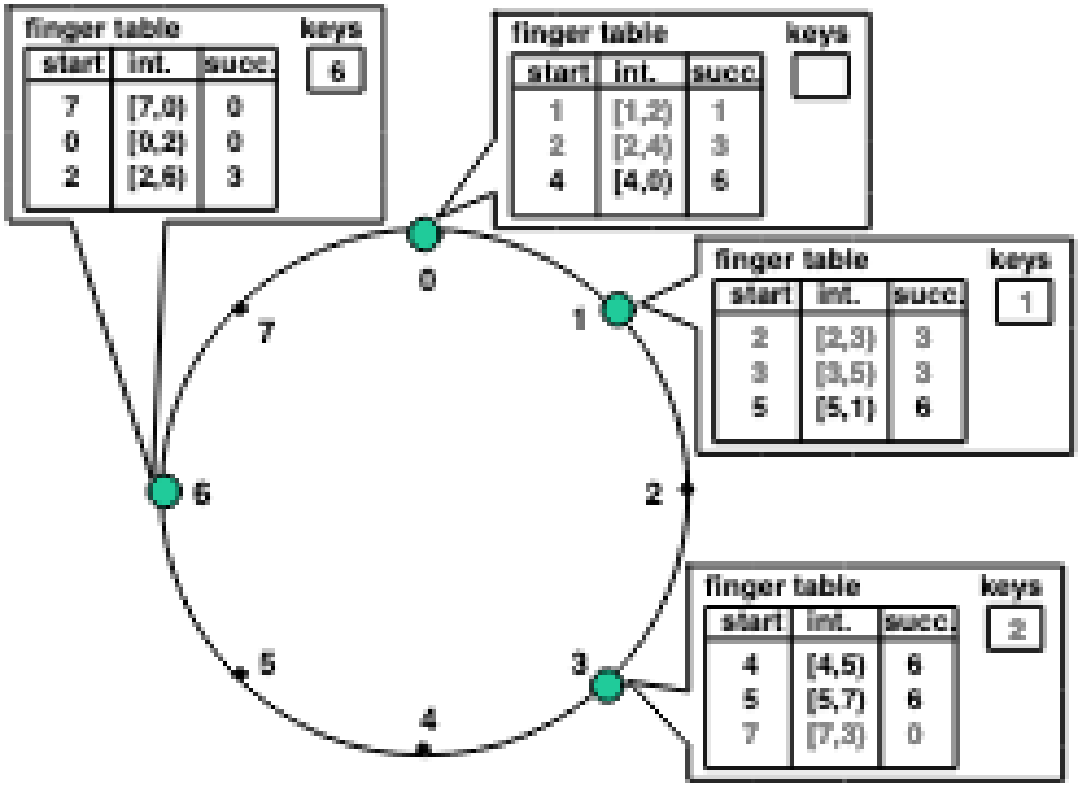


(1) Move key 6 to its new home, which is node 6

(2) Adjust finger tables as necessary (requires traversing the ring)

Removing a node in Chord

Picture taken from Stoica et al, SIGCOMM 2001

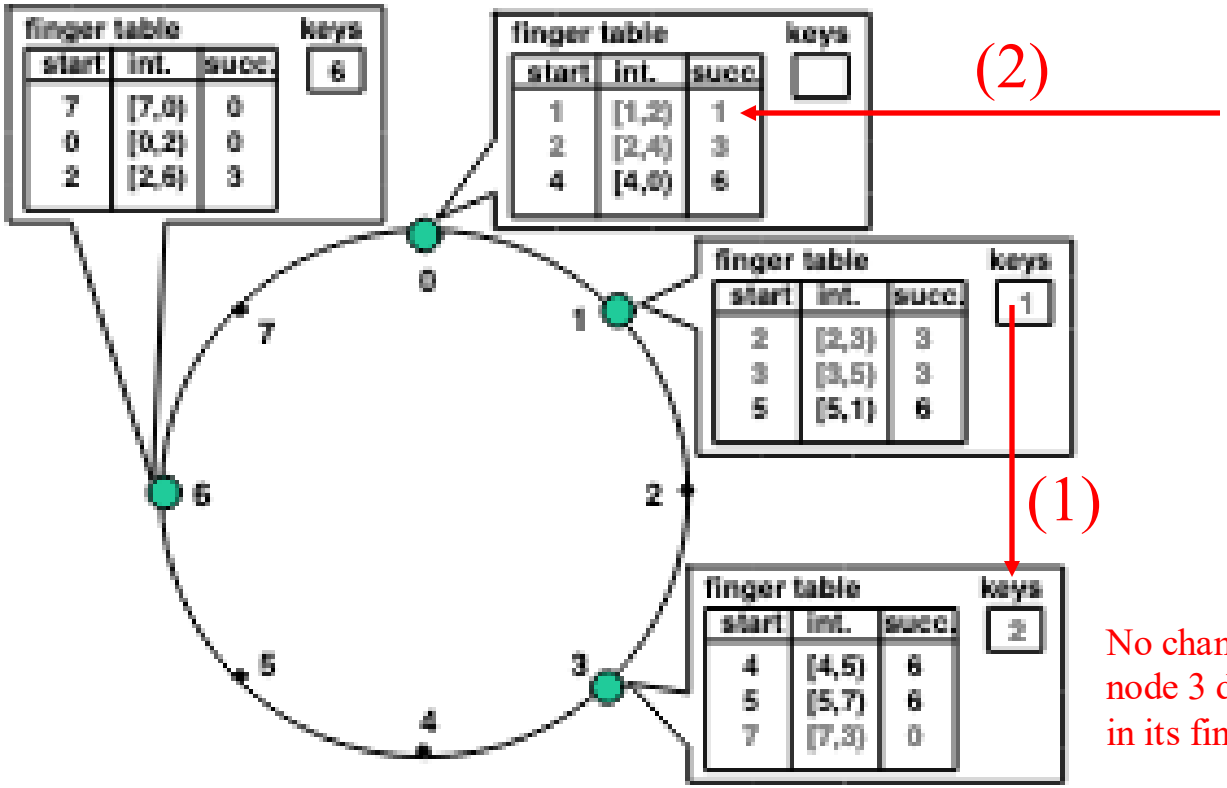


What must happen to remove node 1?

Removing a node in Chord

Picture taken from Stoica et al, SIGCOMM 2001

No change here,
but could have been
if was at node 5



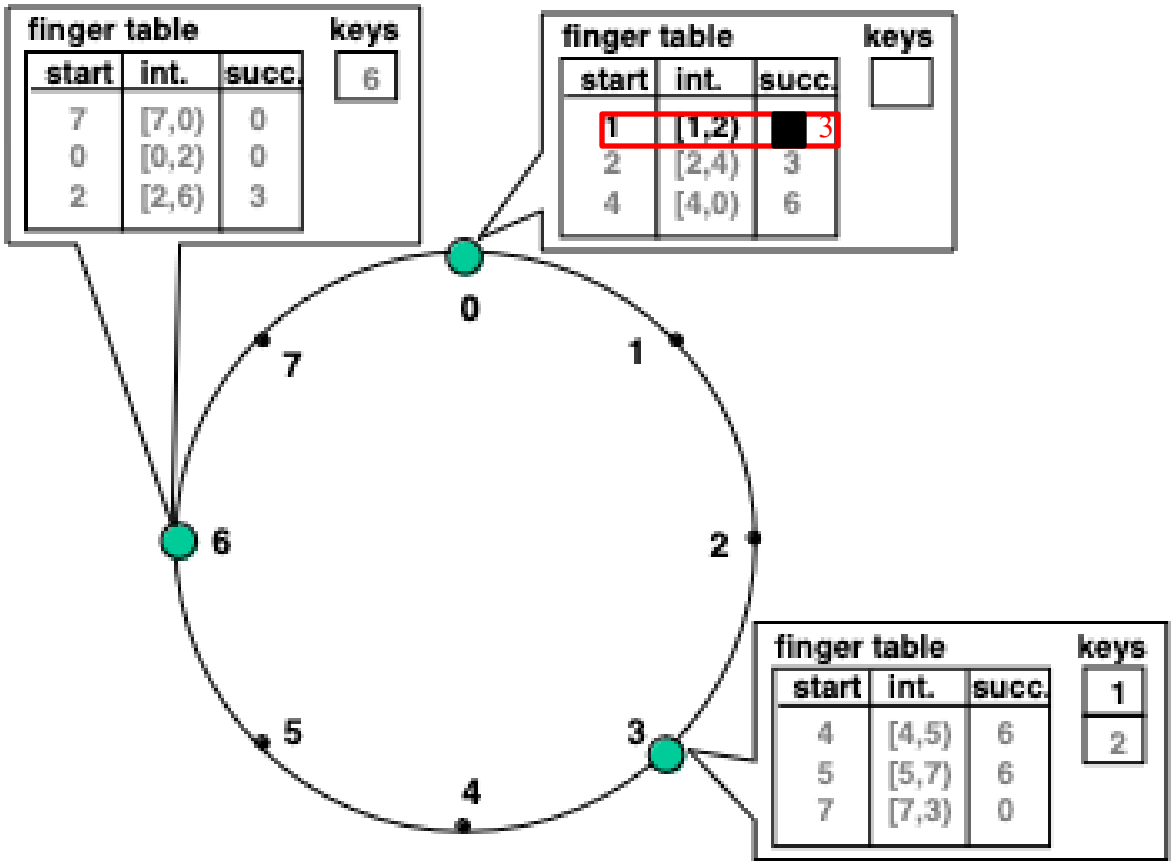
No change here;
node 3 doesn't have 1
in its finger table

(1) Move key 1 to its new home, which must be node 1's successor

(2) Adjust finger tables as necessary (requires traversing the ring)

Removing a node in Chord

Picture taken from Stoica et al, SIGCOMM 2001



Final state after Remove (node 1 is gone)