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

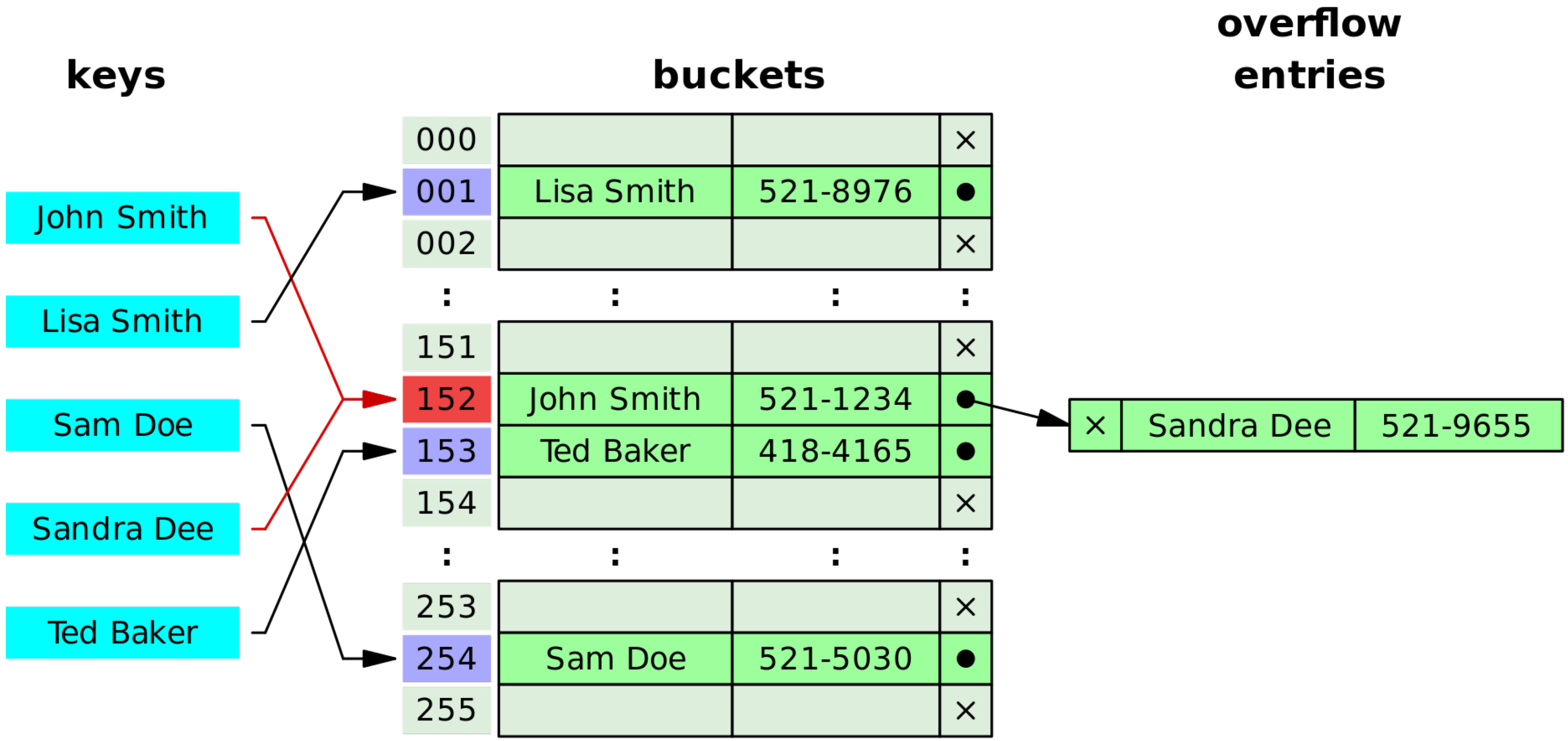
p2p lookup problem

- Traditional approach for this kind of problem is domain name system (DNS)
  - DNS maps a name to an IP address
  - DNS is tree based; with p2p, what happens if the root node (or node near the root) fails?
  - Also, the load is not shared, even with caching

p2p lookup problem

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

Review: Hash Tables



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)
  - Nodes have connectivity through an overlay network (essentially, each node connects to only a small number of nodes)
- Goals of DHT:
  - Load Balance, Decentralization, Scalability, Fault Tolerance (leaving and joining)

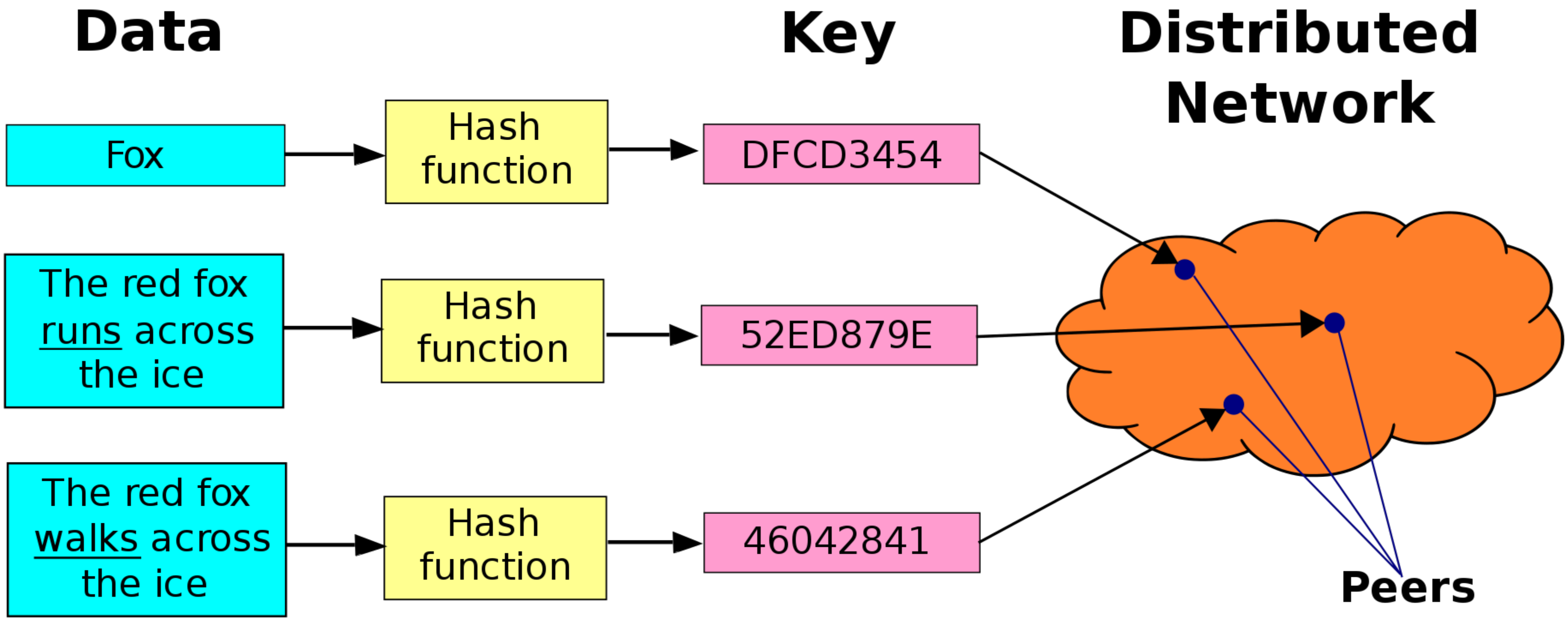
DHT Uses

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

DHT Operations

- Operations are as follows:
  - Given a key *k* and data *d*, we call *put(k, d)* to store that data on a node that is determined by *k*
    - Practically, we may hash the data *d* to produce key *k*
    - Hash function needs to spread keys out evenly between the nodes
  - To retrieve *d*, call *get(k)*, which automatically retrieves it from the proper node

Picture of Distributed Hash Table



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

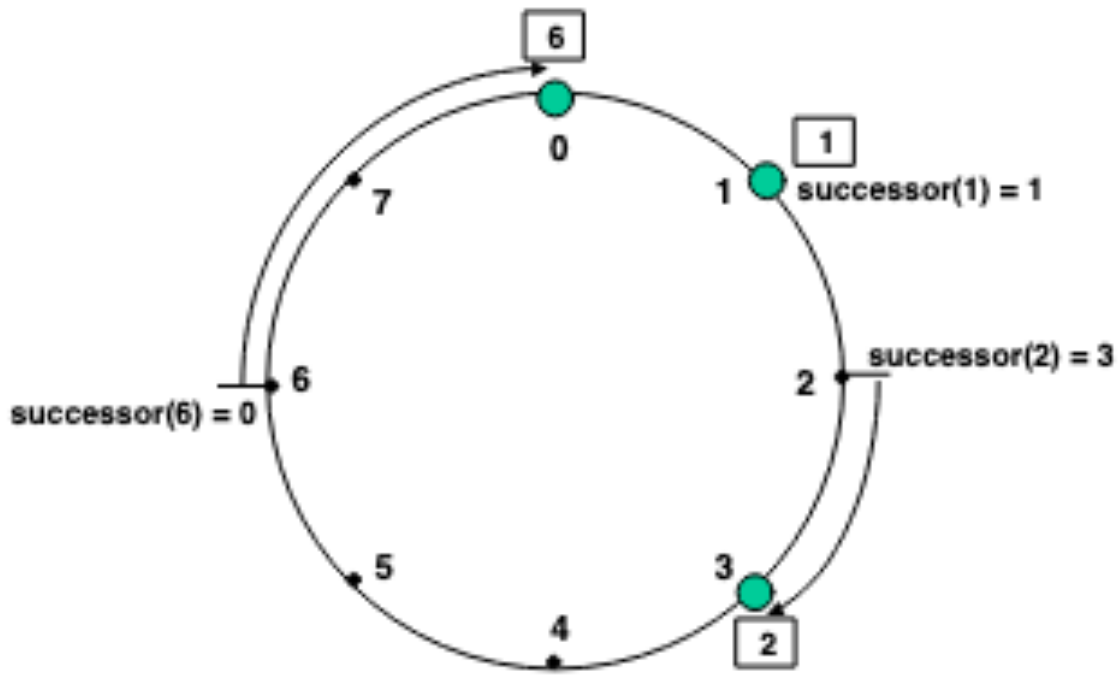
DHT Issues

- Balancing load: need to distribute the data evenly, or else may as well use DNS
- 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
- 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 a mapping from node id to IP address



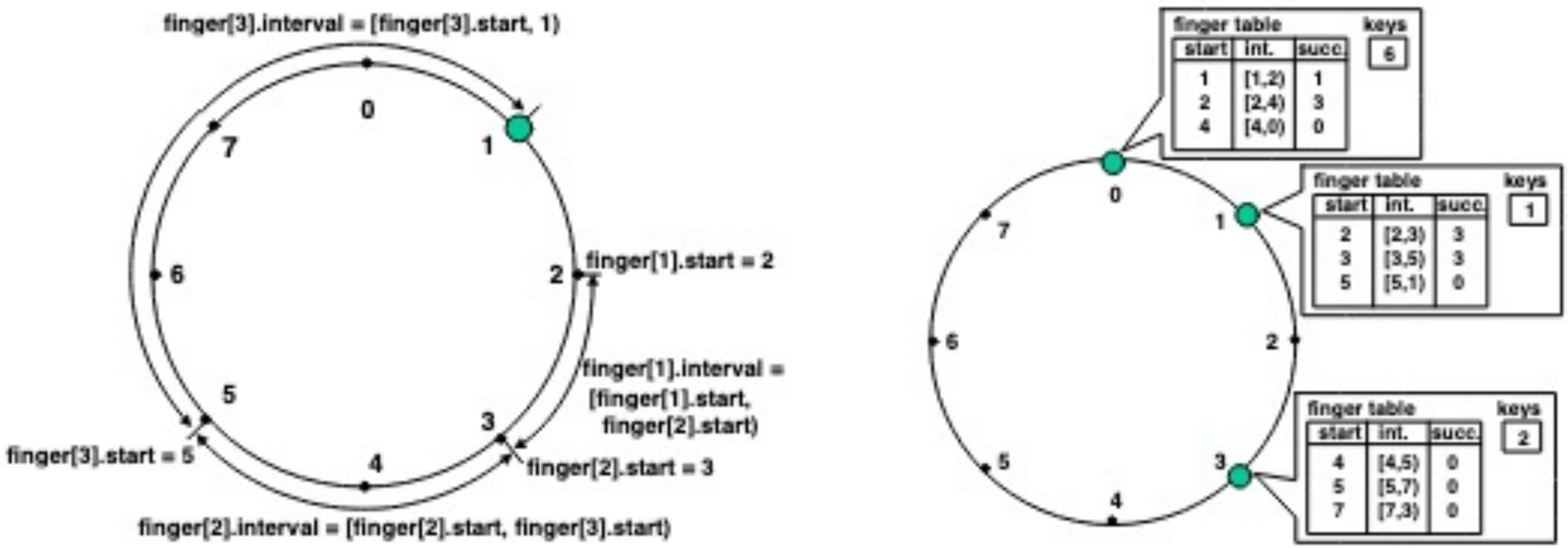
The function *successor* produces the node on which a key is stored --defined as the smallest node id greater than or equal to the key

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

- 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



(a) (b)

(a) Shows the finger intervals for node 1
(b) Shows the finger tables and key locations for nodes 0, 1, 3 and keys 1, 2, and 6

Finding a node for a key in Chord

- To find where to place key *k*, starting at node *n*:

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

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
  - Assuming a hash function that distributes evenly
- 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

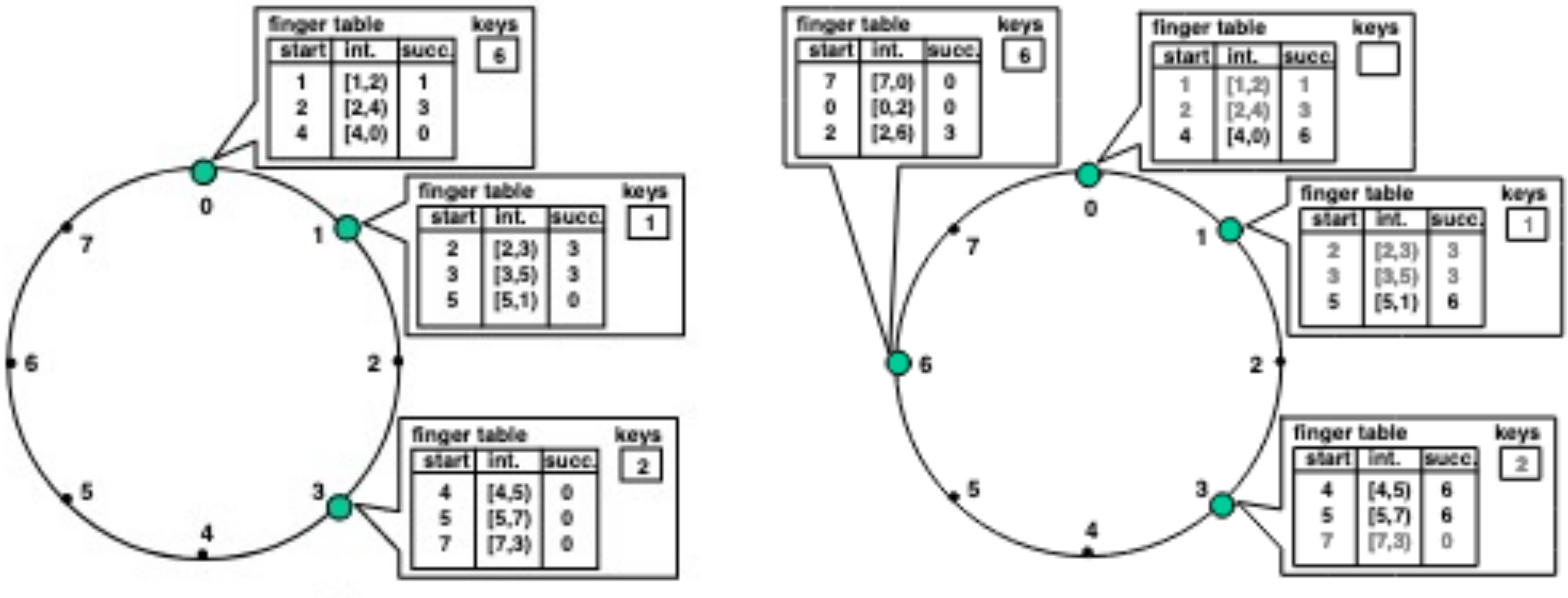
Adding a node in Chord

- To add a new node *i* (assumes *i* is unique and known)
  - 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 in Chord

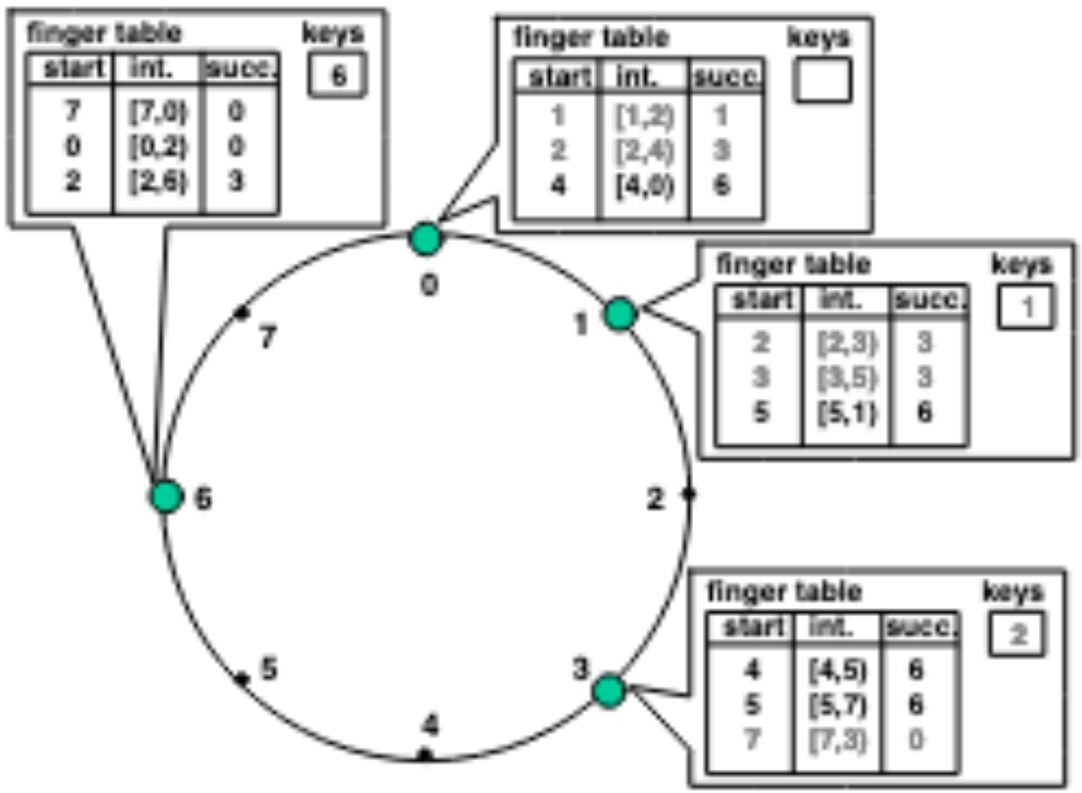
Taken from Stoica et al, SIGCOMM 2001



Adding node 6. Note that key 6 moved from node 0 to node 6

Removing in Chord

Picture taken from Stoica et al, SIGCOMM 2001

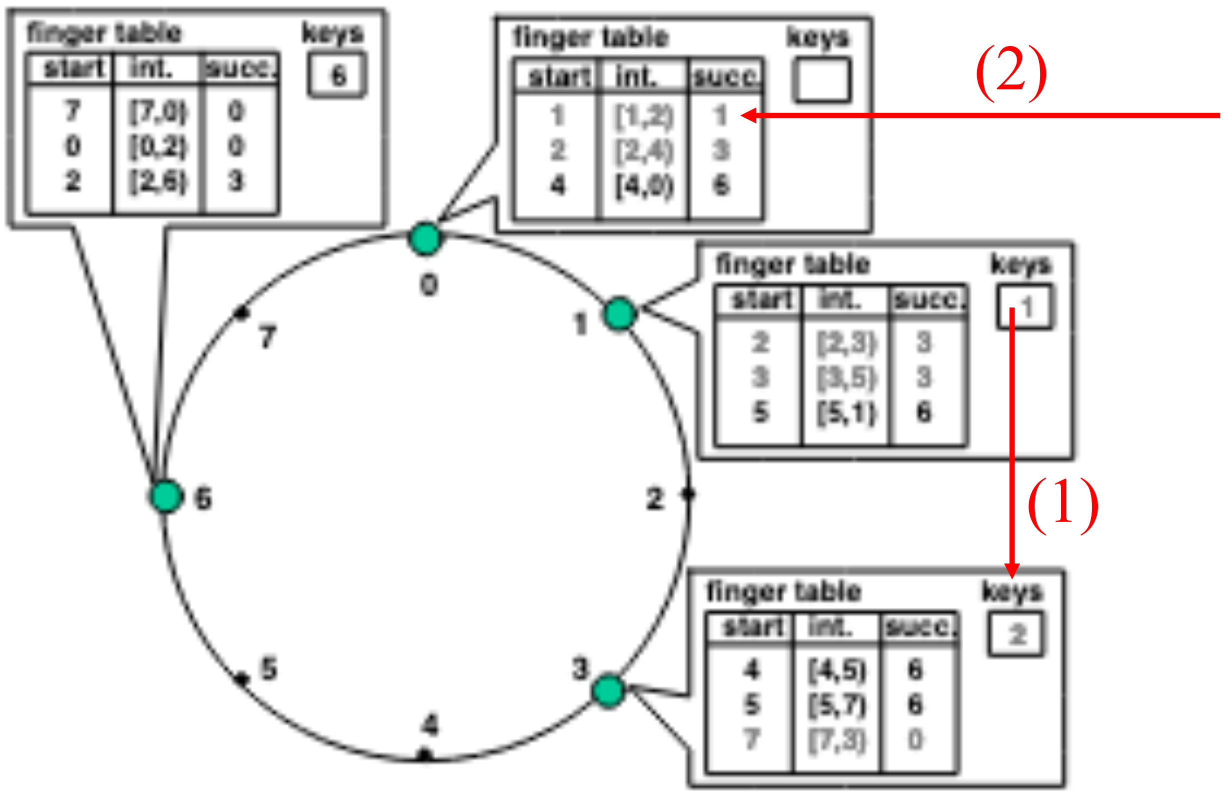


What has to happen to remove node 1?

Removing in Chord

Picture taken from Stoica et al, SIGCOMM 2001

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



(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)