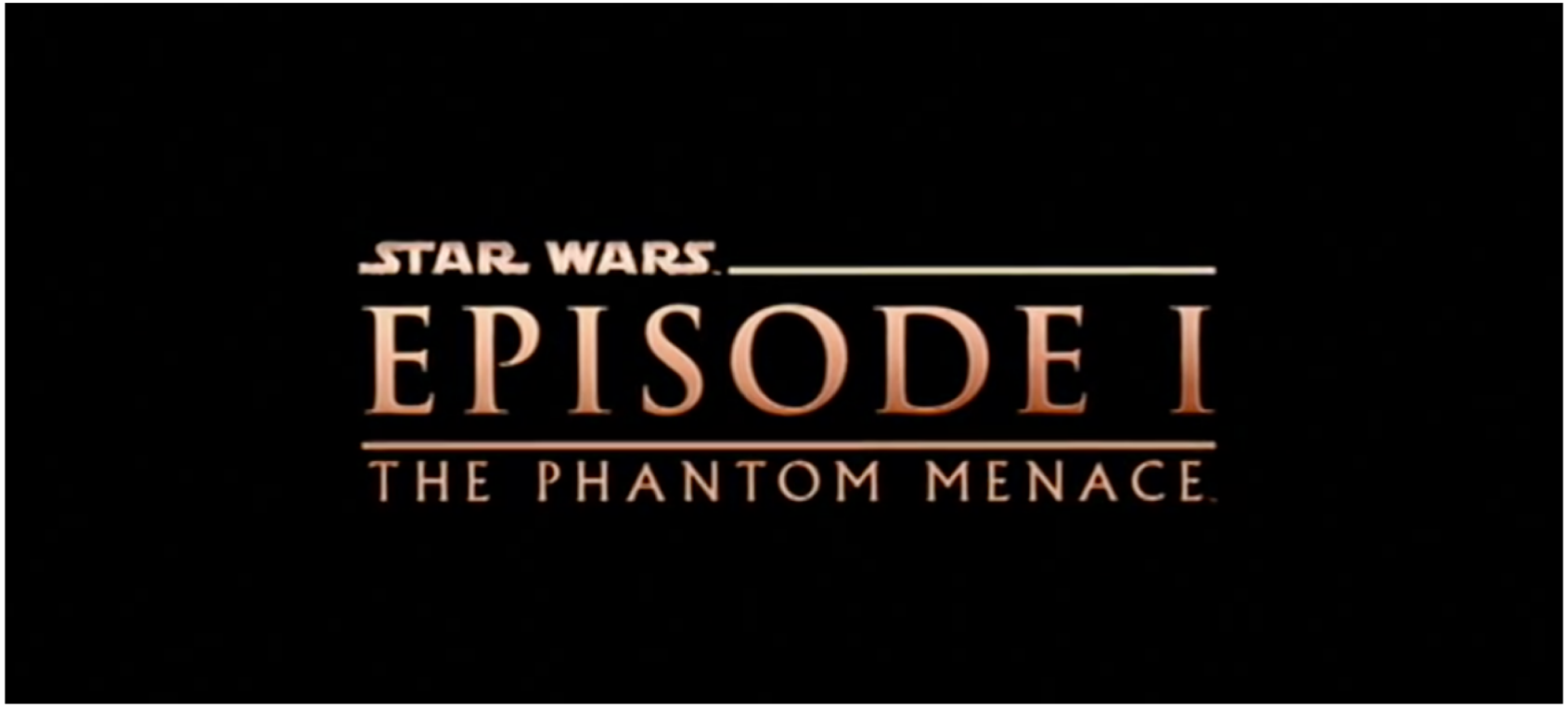
Lecture 9

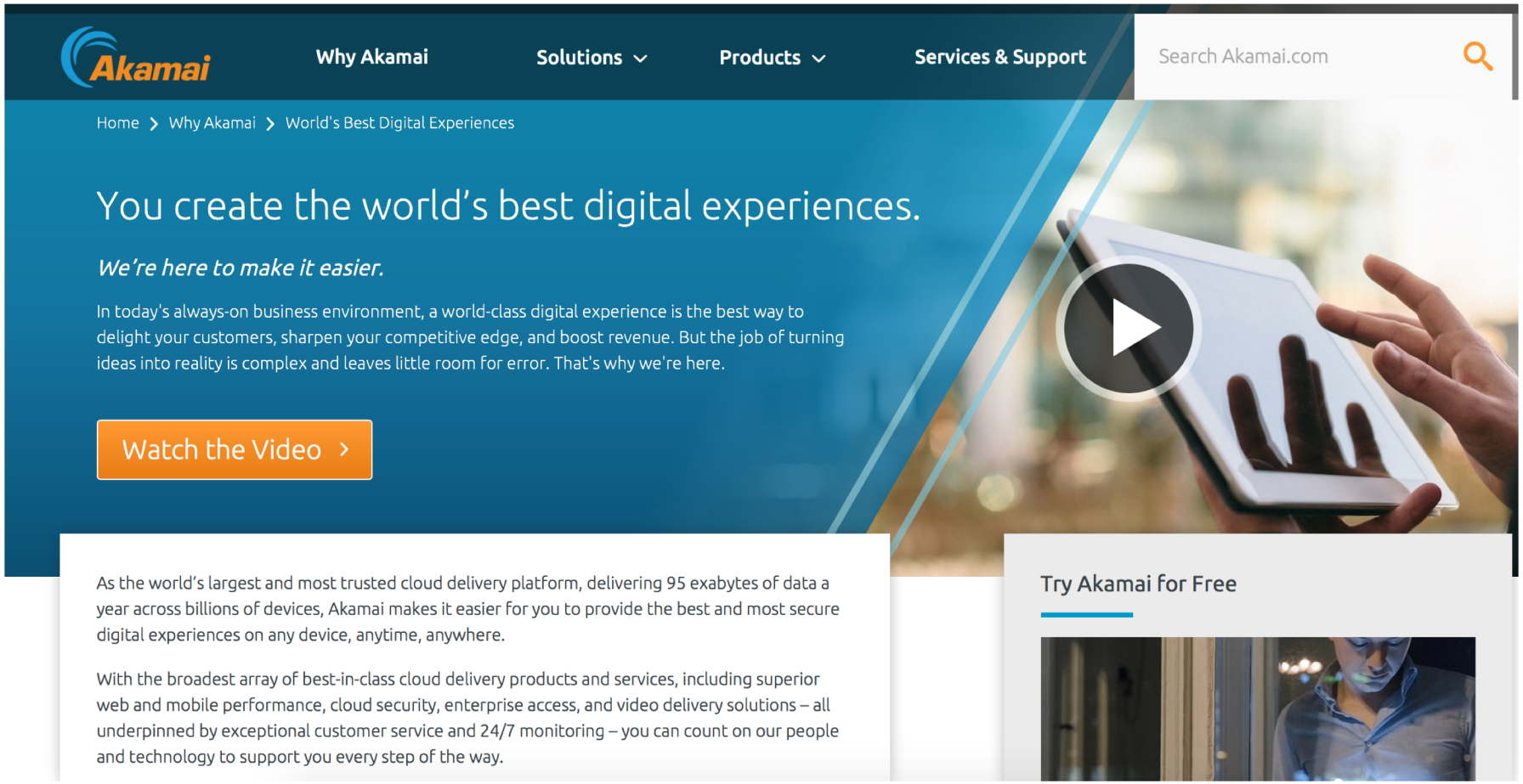
Consistent Hashing & Chord Algorithm

Wilson Rivera

Consistent Hashing!!



Consistent Caching: Akamai

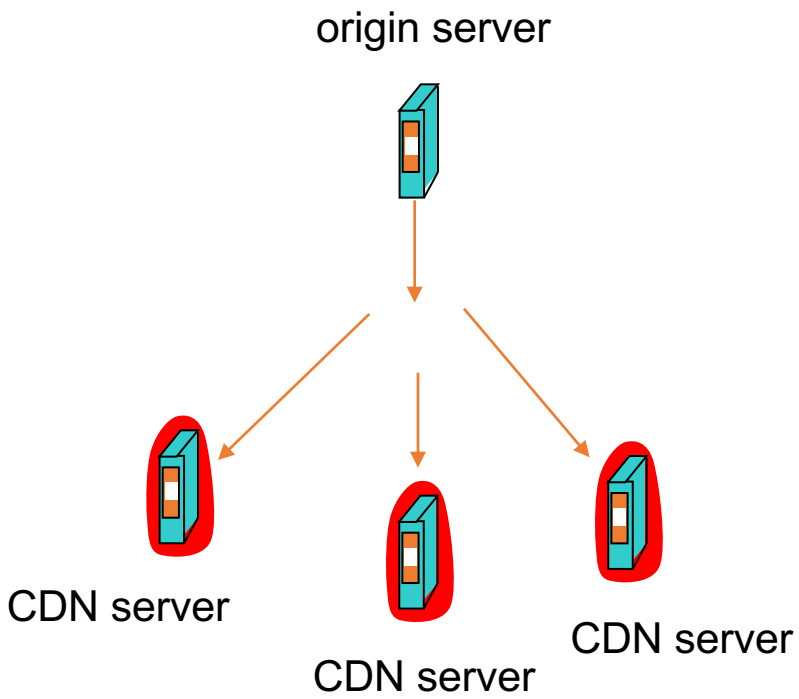


Consistent Caching

- Web caching
  - Faster response, less traffic
- Spread caching over multiple machines
- Consistent hashing
  - 1997: Research paper
  - 1998: Akamai is funded
  - 1999: start wars trailer (apple)

Scalable algorithms for discovery

- If many nodes are available to cache, which one should file be assigned to?
- If content is cached in some node, how can we discover where it is located, avoiding centralized directory or all-to-all communication?



Partitioning Problem

- Consider problem of data partition:
  - Given data X, choose one of k servers to use

- Suppose we use modulo hashing
  - Number servers 1..k
  - Place X on server *i = hash (X) mod k*
    - Problem? What happens if a server fails or joins (k → k±1)?
    - Answer: All entries get remapped to new nodes!

Chord Algorithm – Consistent Hashing

- Supports just one operation: given a key, it maps the key onto a node.

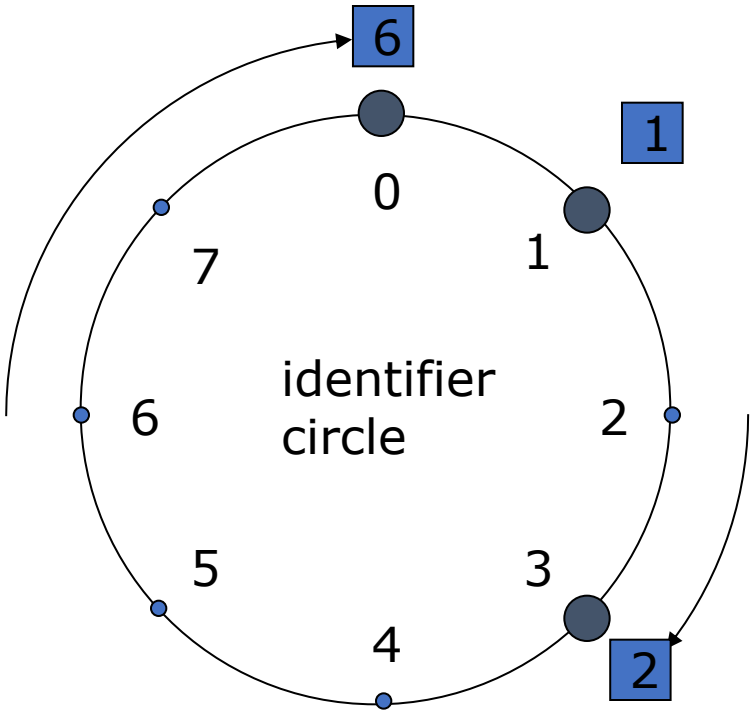
- Consistent Hashing
  - Assign each node and key an m-bit identifier using a base hash function such as SHA-1
  - Identifiers are ordered in an identifier circle modulo 2^m (Chord ring)
  - Key k is assigned to the first node whose identifier is equal to or follows k.

I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In Proceedings of the 2001 ACM SIGCOMM Conference, pages 149–160, 2001.

Chord Algorithm – Consistent Hashing

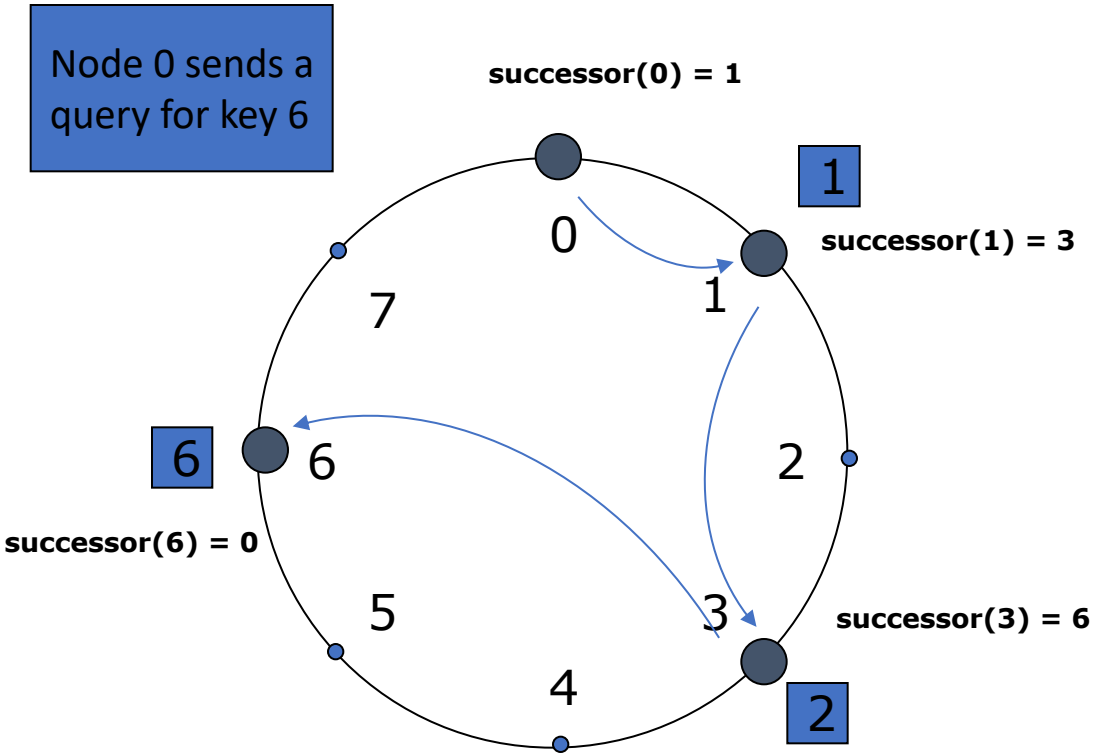
identifier space : m=3

node	key
0	1
1	2
3	6



Chord Algorithm – Simple Key Lookup

- Simple Key Lookup
  - Queries are passed around the circle via successor pointers
  - Requires traversing all Nodes to find the appropriate mapping



Chord Algorithm – Scalable Key Location

- Finger Table
  - Each node n maintains a routing table with up to m entries
  - The i^{th} entry in the table at node n contains the identifier of the first node s that succeeds n by at least 2^{i-1} on the identifier circle.
    - ($s = successor(n+2^{i-1})$).
  - s is called the i^{th} finger of node n.

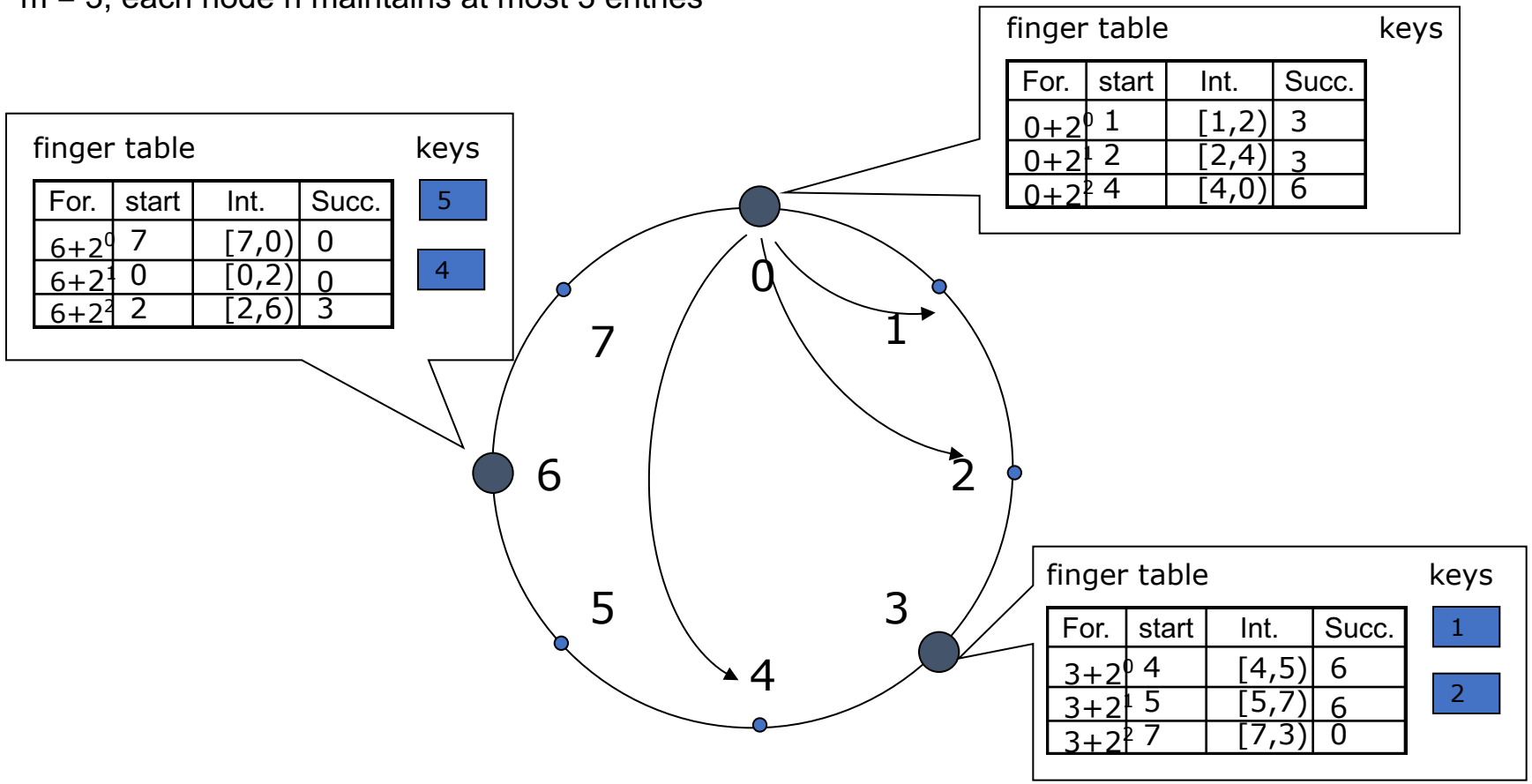
▷ Definition of variables for node n

Notation	Definition
$finger[k].start$	$(n + 2^{k-1}) \bmod 2^m, 1 \leq k \leq m$
$.interval$	$[finger[k].start, finger[k+1].start)$
$.node$	first node $\geq n.finger[k].start$
successor	the next node on the identifier circle; $finger[1].node$
predecessor	the previous node on the identifier circle

Chord Algorithm – Scalable Key Location

Finger table

m = 3, each node n maintains at most 3 entries



Chord Algorithm – Scalable Key Location

```
// ask node n to find id's successor
n.find_successor(id)
   n' = find_predecessor(id);
   return n'.successor;

// ask node n to find id's predecessor
n.find_predecessor(id)
   n' = n;
   while (id ∉ (n', n'.successor])
      n' = n'.closest_preceding_finger(id);
   return n';

// return closest finger preceding id
n.closest_preceding_finger(id)
   for i = m downto 1
      if (finger[i].node ∈ (n, id))
         return finger[i].node;
   return n;
```

Find id's successor by finding the immediate predecessor of the id

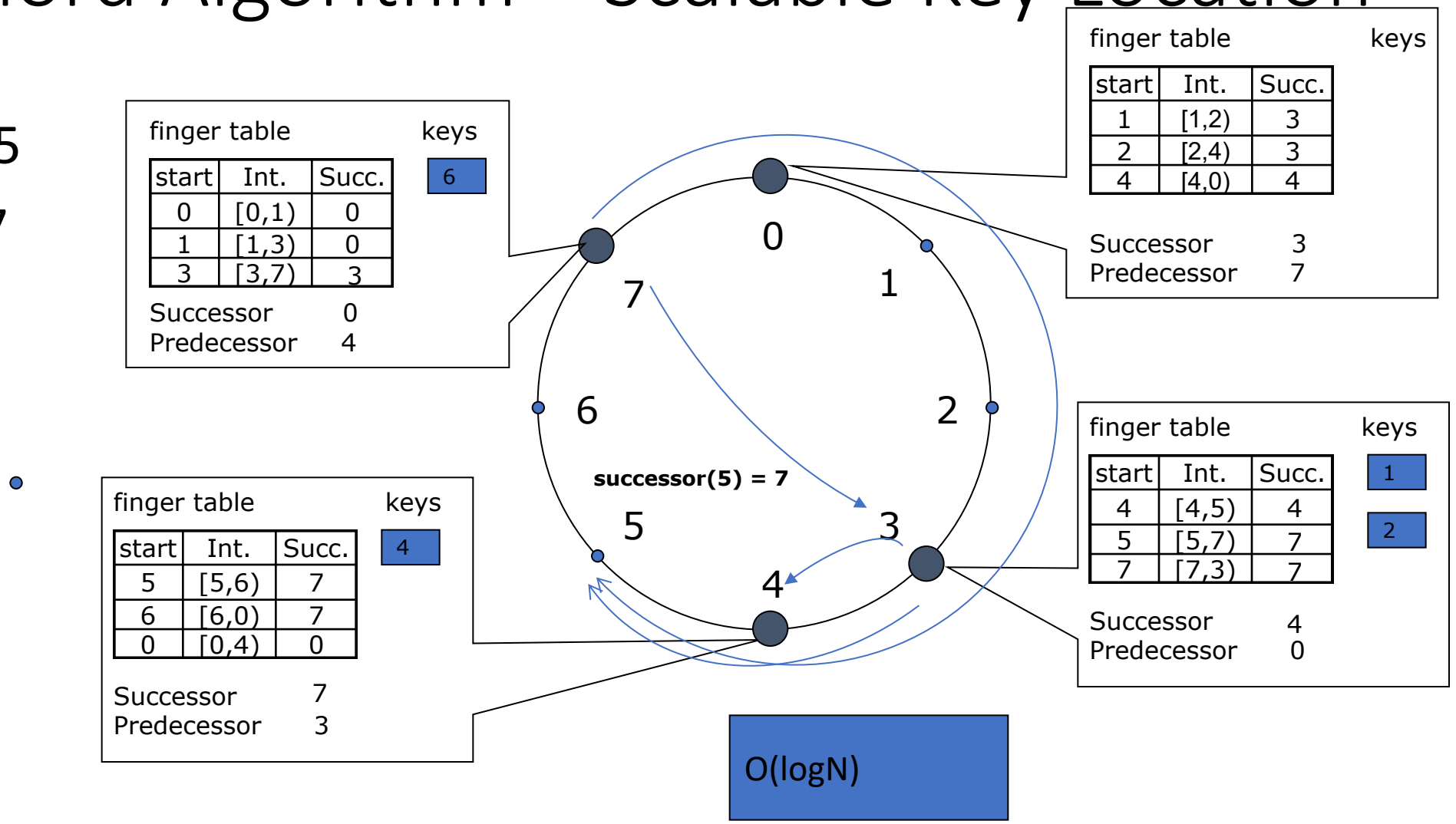
Walk clockwise to find the node which precedes id and whose successor succeeds id

Start with the m^{th} finger of node n
See if it comes between node n and the id, if not, check the $m-1^{th}$ finger until we find one wich does.
This is the closest node preceding id among all the fingers of n

Chord Algorithm – Scalable Key Location

id=5

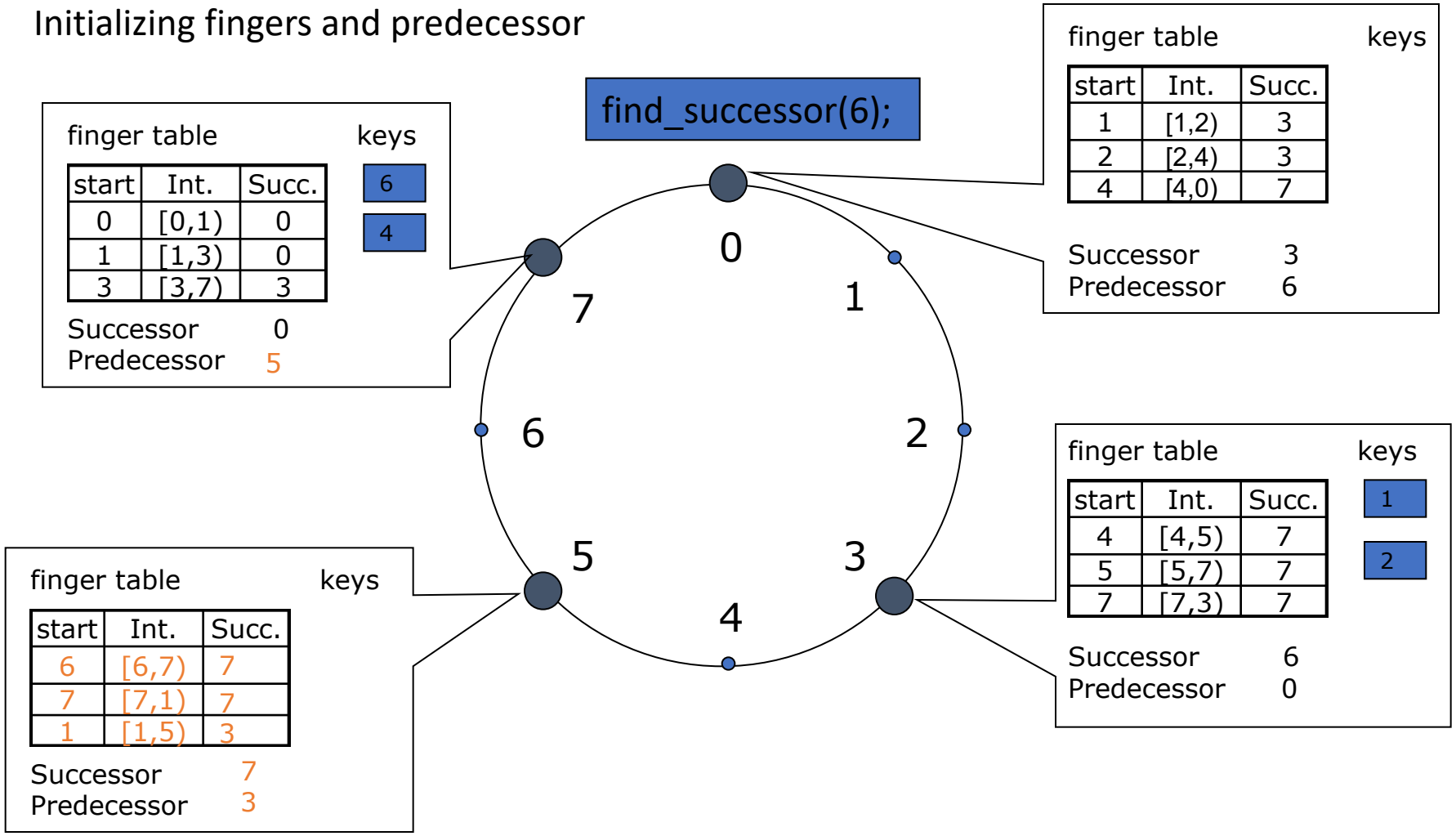
n=7



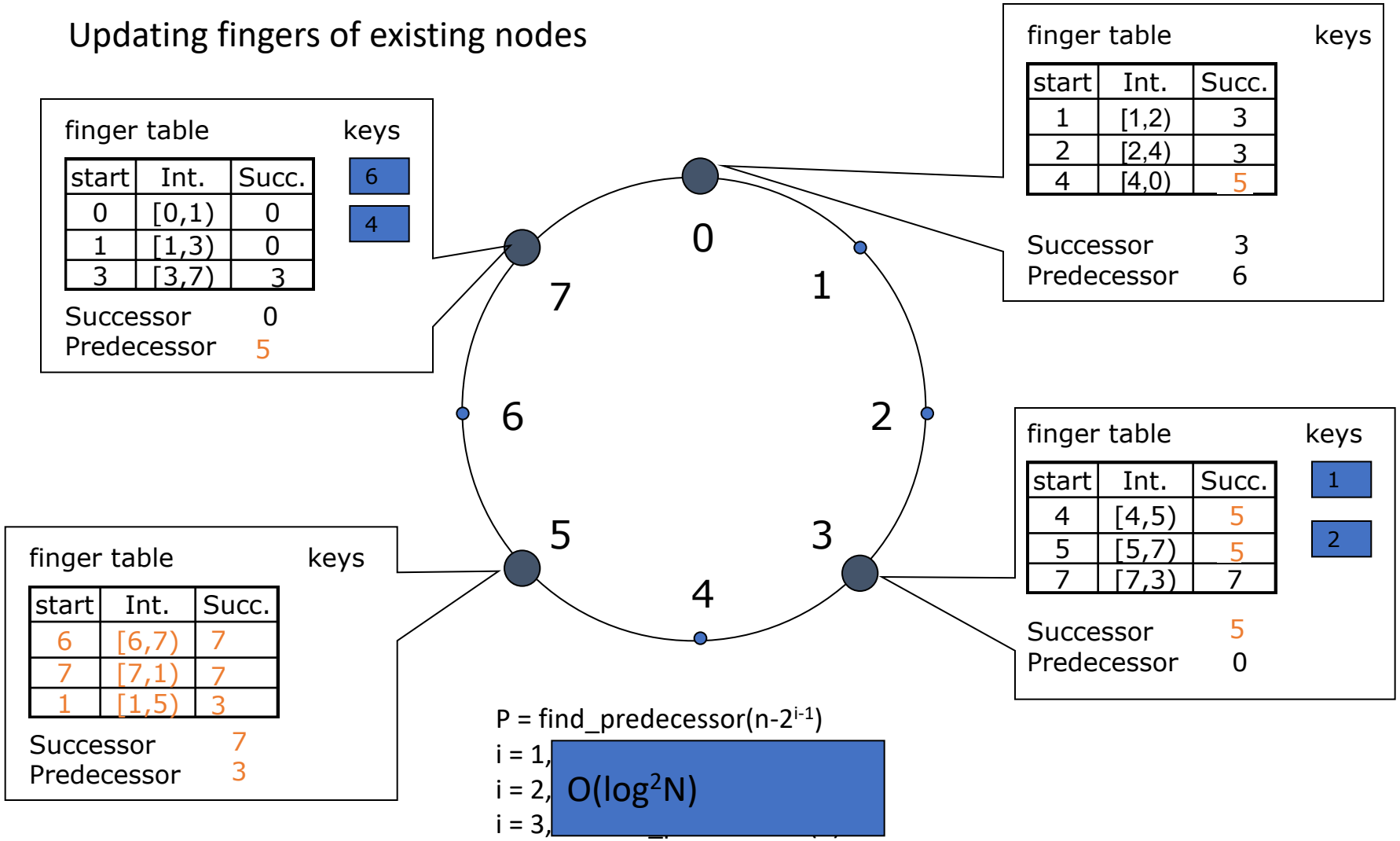
Chord Algorithm - Node joins

- Invariants to be preserve
  - Each node's successor is correctly maintained
  - For every key k, node successor(k) is responsible for k
- It is desirable for the finger tables to be correct
- Tasks to be performed by Chord
  - Initialize the predecessor and fingers of node n
  - Update the fingers and predecessor of existing nodes to reflect the addition of n
  - Notify the higher layer software so that it can transfer state associated with keys that node n is now responsible for

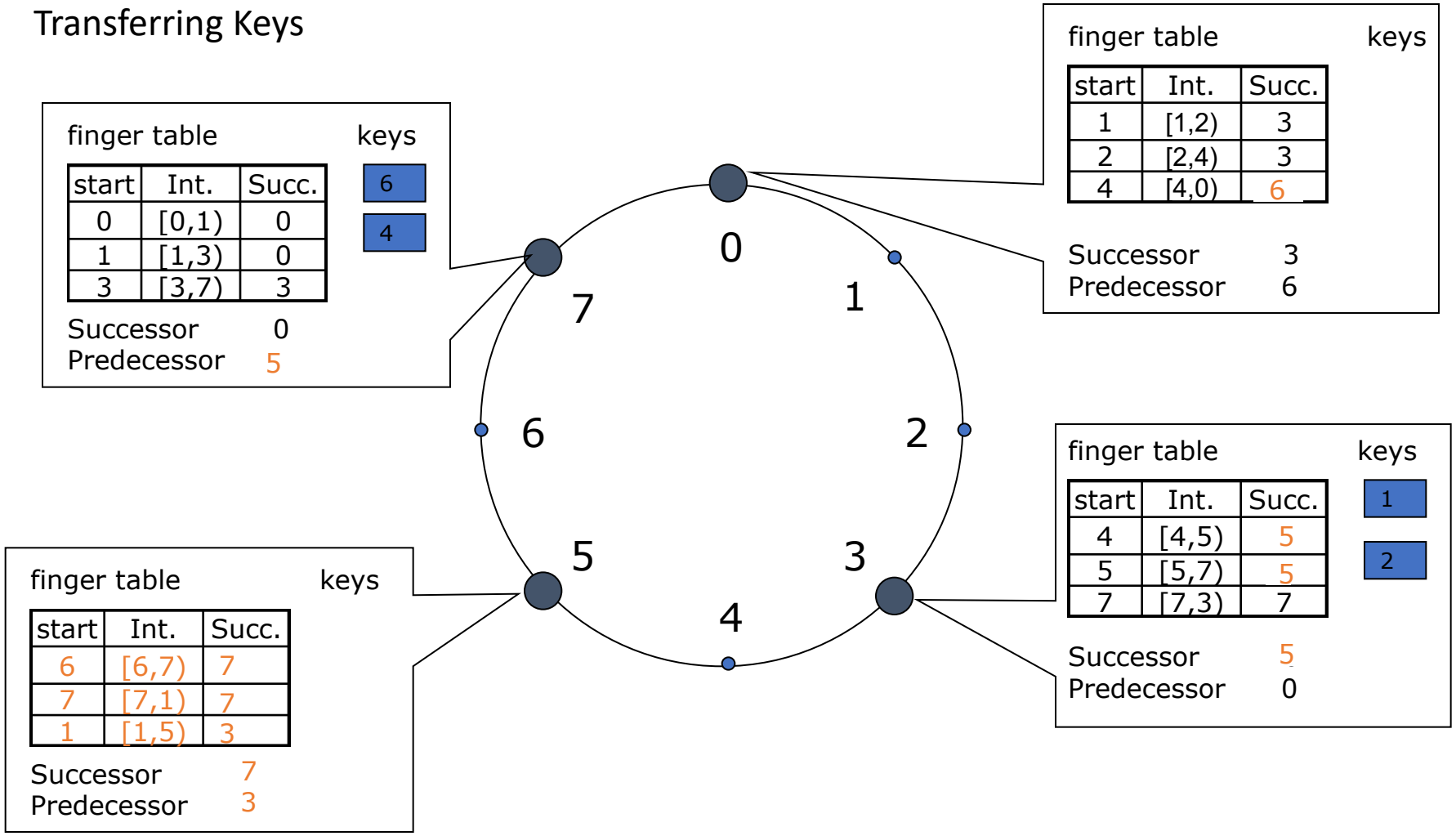
Chord Algorithm - Node joins



Chord Algorithm - Node joins



Chord Algorithm - Node joins



Chord Algorithm – node joins

```
#define successor finger[1].node

// node n joins the network;
// n' is an arbitrary node in the network
n.join(n')
  if (n')
    init_finger_table(n');
    update_others();
    // move keys in (predecessor, n] from successor
  else // n is the only node in the network
    for i = 1 to m
      finger[i].node = n;
    predecessor = n;

// initialize finger table of local node;
// n' is an arbitrary node already in the network
n.init_finger_table(n')
  finger[1].node = n'.find_successor(finger[1].start);
  predecessor = successor.predecessor;
  successor.predecessor = n;
  for i = 1 to m - 1
    if (finger[i + 1].start ∈ [n,finger[i].node))
      finger[i + 1].node = finger[i].node;
    else
      finger[i + 1].node =
          n'.find_successor(finger[i + 1].start);
```

```
// update all nodes whose finger
// tables should refer to n
n.update_others()
  for i = 1 to m
    // find last node p whose i^th finger might be n
    p = find_predecessor(n - 2^(i-1));
    p.update_finger_table(n, i);

// if s is i^th finger of n, update n's finger table with s
n.update_finger_table(s, i)
  if (s ∈ [n,finger[i].node))
    finger[i].node = s;
    p = predecessor; // get first node preceding n
    p.update_finger_table(s, i);
```

Pseudocode for the node join operation

Chord Algorithm - Stabilization

- Stabilization
  - Correctness and performance
  - Keep node‘s successor pointers up to date
  - Use successor pointers to verify correct finger table entries

Chord Algorithm - Stabilization

```
n.join(n')
    predecessor = nil;
    successor = n'.find_successor(n);

// periodically verify n's immediate successor,
// and tell 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';

// periodically refresh finger table entries.
n.fix_fingers()
    i = random index > 1 into finger[];
    finger[i].node = find_successor(finger[i].start);
```

Join does not make the rest of the network aware of n

Every node runs stabilize periodically, to verify the successor

Node n asks its successor for the successor's predecessor x. See if x should be n's successor instead. (happens if x recently joined the system)

Notify n's successor of n's exist. Successor changes its predecessor to n if it knows no closer predecessor than n.

Use successor pointers to update finger tables.

Chord Algorithm – Node Failure

- Node Failure
  - Successor-list
  - If successor fails, replaces it with the first live entry in the list
  - Later run *stabilize* to correct finger table and successor-list

Summary

- The routing table at each node contains $O(log(n))$ entries
- Inserting and deleting nodes requires $O(log(n)^2)$ messages
- The hash for the node and key is generated by using the SHA-1 algorithm.