Chord: A scalable Peer-to-Peer Lookup Service for Internet Applications

Παρουσίαση: Καραγιάννης, Πλίτσης, Στεφανίδης, Τζιοβάρα, Τσώτσος

Authors: Stoica, Morris, Karger, Kaashoek, Balakrishman

SIGCOMM 2001

Topic of Presentation

- Peer to Peer applications need to locate efficiently the node that stores a particular data item
- Chord, a distributed lookup protocol, addresses this problem
  - Given a key, Chord maps the key onto a node

Introduction

- P2P systems and applications are distributed systems without any centralized control or hierarchical organization
- The software that runs at each node is equivalent in functionality
- The core operation in most P2P systems is the efficient location of data items
- Chord is a scalable protocol for lookup in a dynamic P2P systems with frequent node arrivals and departures

Introduction

- The Chord protocol supports just one operation: given a key, it maps the key onto a node
- Chord uses a variant of consistent hashing to assign keys to Chord nodes
- Consistent hashing tends to balance load
  - Each node receives roughly the same number of keys
  - Involves little movement of keys when nodes join and leave the system

Introduction

- Each Chord node needs "routing" information about only a few other nodes
  - The routing table is distributed
- In an N-node system, each node maintains information only about O(logN) other nodes, and resolves all lookups via O(logN) messages to other nodes
- Chord maintains its routing information as nodes join and leave the system; with high probability each such event results in no more than $O(\log^2 N)$ messages

Introduction

- Features that distinguish Chord from other P2P lookup protocols are:
  - Simplicity
  - Provable correctness
  - Provable performance
- A Chord node requires information about O(logN) other nodes for efficient routing
- Performance degrades gracefully when information is out of date
  - Nodes join and leave arbitrarily
- Only one piece information per node needs to be correct (slower lookup)

System Model

- Load balance: Chord acts as a distributed hash function, spreading keys evenly over the nodes
  - This provides a degree of natural load balance
- Decentralization: Chord is fully distributed: no node is more important than any other
  - This improves robustness
- Scalability: The cost of a Chord lookup grows as the log of the number of nodes
  - Very large systems are feasible

System Model

- Availability: Chord automatically adjusts its internal tables to reflect newly joined nodes as well as node failures
  - The node responsible for a key can always be found
  - This is true even if the system is in a continuous state of change
- Flexible naming: Chord places no constraints on the structure of the keys it looks up

System Model

- An application interacts with Chord in two main ways:
  - Chord provides a lookup(key) algorithm that yields the IP address of the node responsible for the key
  - The Chord software on each node notifies the application of changes in the set of keys that the node is responsible for
    - This allows the application to move corresponding values to their new homes when a new node joins

Examples of Applications

- Cooperative Mirroring:
  - In order to balance the load across all servers, data is replicated and cached
- Time-Shared Storage:
  - If a person wishes some data to be always available, but their machine is only occasionally available, they can offer to store others' data while they are up, in return for having their data stored elsewhere when they are down
- Distributed Indices:
  - Each machine maintains lists of machines that offer documents. Each document is specified by a keyword. The lookup is faster when using these lists
- Large-Scale Combinatorial Search:
  - In this case, keys are candidate solutions to the problem. Chord maps these keys to the machines responsible for testing them as solutions

The Base Chord Protocol

- The Chord protocol specifies:
  - how to find the locations of keys
  - how new nodes join the system
  - how to recover from the failure (or planned departure) of existing nodes

- Here we describe a simplified version of the protocol that does not handle concurrent joins or failures

The Base Chord Protocol

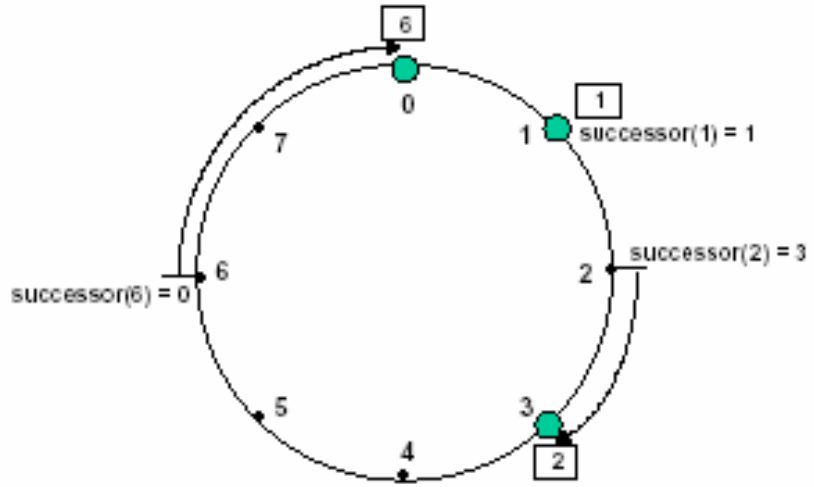
- Chord uses consistent hashing in order to map keys to nodes responsible for them
- With high probability the hash function balances load
- With high probability, when an N^{th} node joins (or leaves) the network, only an O(1/N) fraction of the keys are moved to a different location

Consistent Hashing

- The consistent hash function assigns each node and key an *m*-bit *identifier* using a base hash function such as SHA-1
- A node's identifier is chosen by hashing the node's IP address
- A key's identifier is produced by hashing the key
- The identifier length *m* must be large enough to make the probability of two nodes or keys hashing to the same identifier negligible

Consistent Hashing

- Identifiers are ordered in an identifier circle modulo 2^m
- Key k is assigned to the first node whose identifier is equal to or follows (the identifier of) in the identifier space
- This node is called the successor node of key
- If identifiers are represented as a circle of numbers from 0 to 2^m - 1, then successor(k) is the first node clockwise from k



Consistent Hashing

- Consistent hashing is designed to let nodes enter and leave the network with minimal disruption
- When a node n joins the network, certain keys previously assigned to n's successor now become assigned to n
- When node n leaves the network, all of its assigned keys are reassigned to n's successor
- No other changes in assignment of keys to nodes need occur

Scalable Key Location

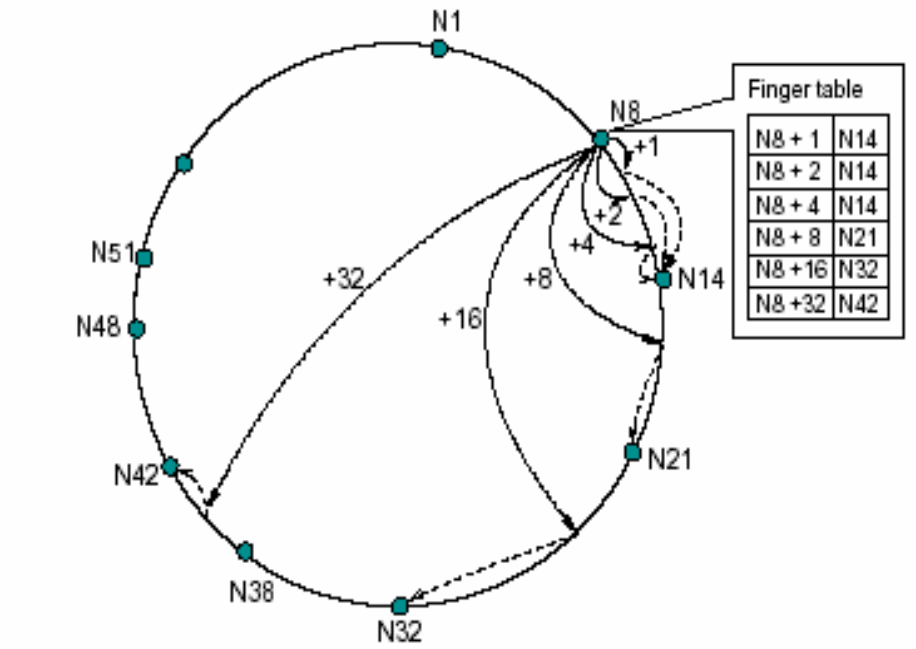
- Each node need only be aware of its successor node on the circle
- Queries for a given identifier can be passed around the circle via these successor pointers until they first encounter a node that succeeds the identifier
- Chord protocol maintains these successor pointers
- It may require traversing all N nodes to find the appropriate mapping
- Chord maintains additional routing information

Scalable Key Location

- m: number of bits in the key/node identifiers
- finger table: each node n maintains a routing table with at most m entries
- The i^{th} entry in the table at node n contains the identity of the first node, s, that succeeds n by at least 2^{i-1} on the identifier circle, i.e., s=successor(n + 2^{i-1}), where $1 \leq i \leq m$ and all arithmetic is modulo 2^m
- A finger table entry includes the Chord identifier and the IP address of the relevant node

Scalable Key Location

- Each node stores information about nodes closely following it on the identifier circle
- A node's finger table generally does not contain enough information to determine the successor of an arbitrary key



Scalable Key Location

- When a node n does not know the successor of a key k, it finds a node whose ID is closer than its own to k
- That node will know more about the identifier circle in the region of k than n does
- By repeating this process, n learns about nodes with IDs closer and closer to k

Search Process

- Lookup for id's successor
  - Find id's predecessor
  - Return predecessor's successor
- Find id's predecessor
  - If n is the predecessor
    - Return n
  - Else, return the closest preceding finger
- Find the closest preceding finger
  - Search finger table from last item to first, in order to find a node which is closer to id, among the nodes in the finger table

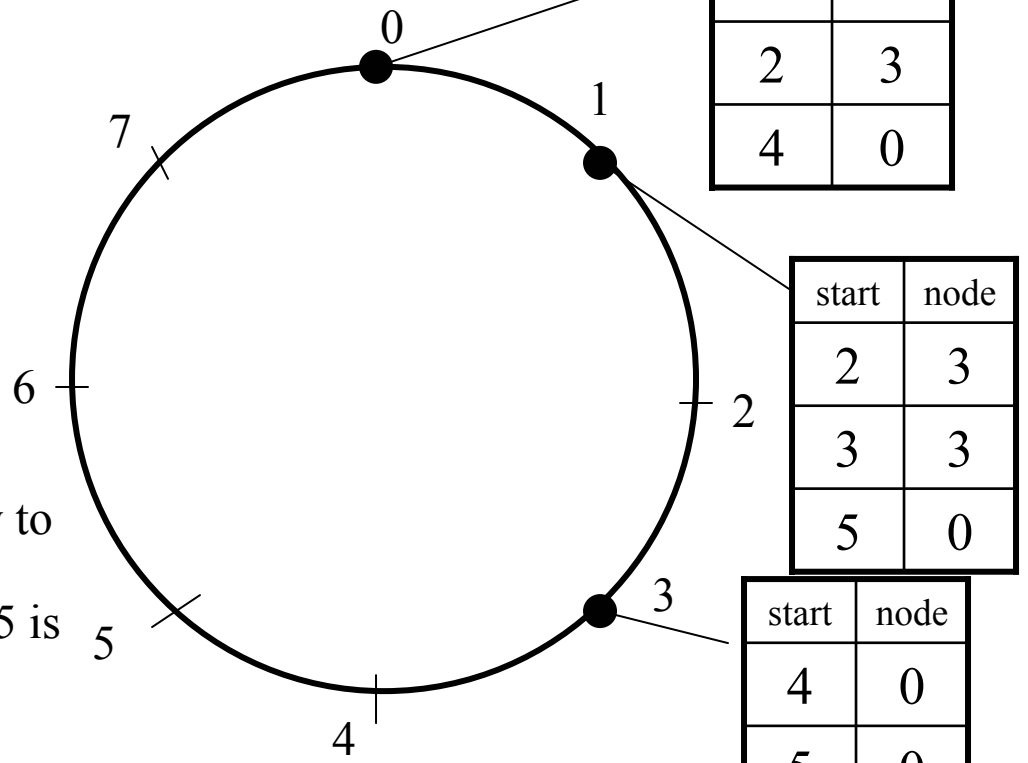
Search Process

```
n.find_successor(id)                    //ask node n to find id's successor
    n' = find_predecessor(id);
    return n'.successor;
n.find_predecessor(id)                  // ask node n to find id's predecessor
    n' = n;
    while(id ∉ (n', n'.successor])
        n' = n'.closest_preceding_finger(id);
    return n';
n.closest_preceding_finger(id)          //return closest finger preceding id
for i=m downto 1
    if(finger[i].node ∈(n, id))
        return finger[i].node;
return n;
```

Lookup Examples

1) Node 0 searches for id 2
- Node 0 examines its finger table
- Id 2 has as a successor node 3

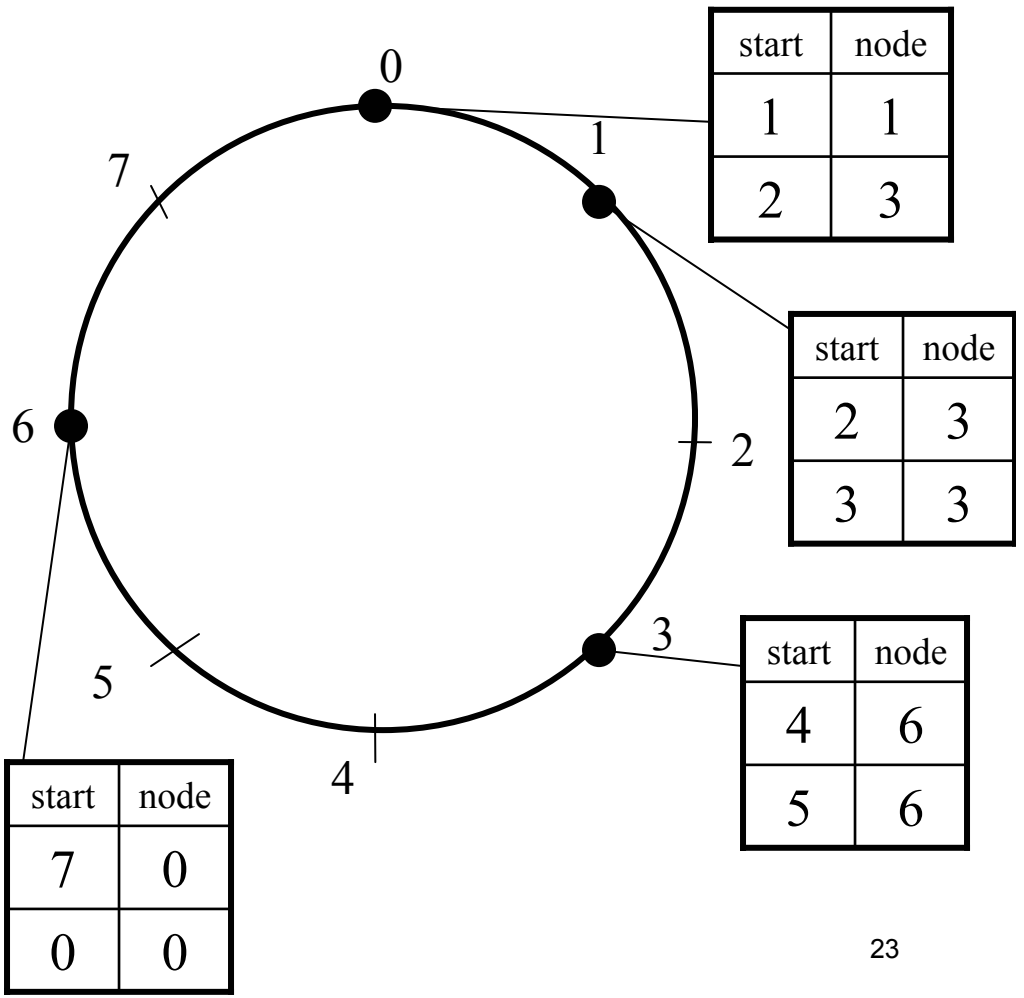
2) Node 0 searches for id 5
- Firstly, searches for 5's predecessor
- Node 0 is not the 5's predecessor
- Searches for the closest preceding finger
- Scans finger table from the last entry to the first
- The first node that is between 0 and 5 is returned (node 3)
- Node 3 is checked and it is 5's predecessor
- Node 3's successor is returned (this is 5's successor)



Lookup Examples

3) Node 1 searches for id 7

- Search 7's predecessor
- Node 1 is not 7's predecessor
- Search for the closest preceding finger
- Scan finger table from last entry to first
- The first node that is between 1 and 7 is returned (node 3)
- Node 3 is checked and it is not 7's predecessor
- Search for the closest preceding finger from node 3
- Scan 3's finger table from last entry to first
- The first node that is between 3 and 7 is returned (node 6)
- Node 6 is checked and it is 7's predecessor
- Node 6's successor is returned (this is 7's successor)



Node Joins

- In a dynamic network, nodes can join and leave at any time
- The main goal is to have the ability to locate every key in the network at any time
- Chord preserves two invariants
  - Each node's successor is correctly maintained
  - For every key k, node successor(k) is responsible for k
- In order for lookups to be fast, it is also desirable for the finger tables to be correct

Node Joins

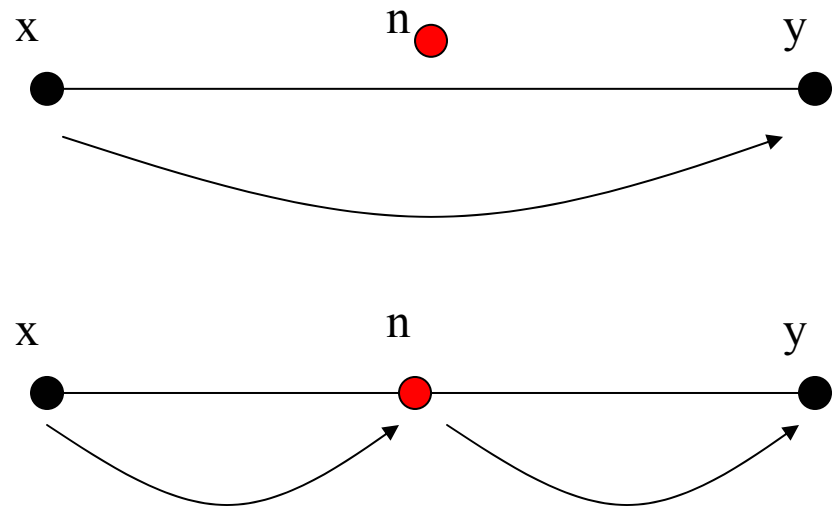
- To simplify the join and leave mechanisms, each node in Chord maintains a predecessor pointer
- A node's predecessor pointer contains the Chord identifier and IP address of the immediate predecessor of that node, and can be used to walk counterclockwise around the identifier circle

Node Joins

- Chord must perform three tasks when a node joins the network:
  - Initialize the predecessor and fingers of node n
  - Update the fingers and predecessors of existing nodes to reflect the addition of n
  - Notify the higher layer software so that it can transfer state (e.g. values) associated with keys that node n is now responsible for

Initializing Fingers and Predecessor

- In order to find the predecessor of n:
  - Find the predecessor of n's successor
- Now, the predecessor of n's successor is n and the old predecessor of n's successor is n's predecessor



First, *y* is the successor of *x*
After *n*'s insertion, the successor of *x* is *n* and the successor of *n* is *y*

Initializing Fingers and Predecessor

- The naively way to make the finger table is to find the successor of each entry
- Better solution: if two sequential entries have the same successor there is no need to find the successor of the second one
  - This reduces the number of remote calls
- Practical optimization: ask an immediate neighbor for a copy of its complete finger table and its predecessor

Updating Fingers of existing Nodes

- Node n will need to be entered into the finger tables of some existing nodes
- Node n will become the i^{th} finger of node p if and only if
  - p precedes n by at least 2^{i-1}
  - The i^{th} finger on node p succeeds n

Updating Fingers of Existing Nodes

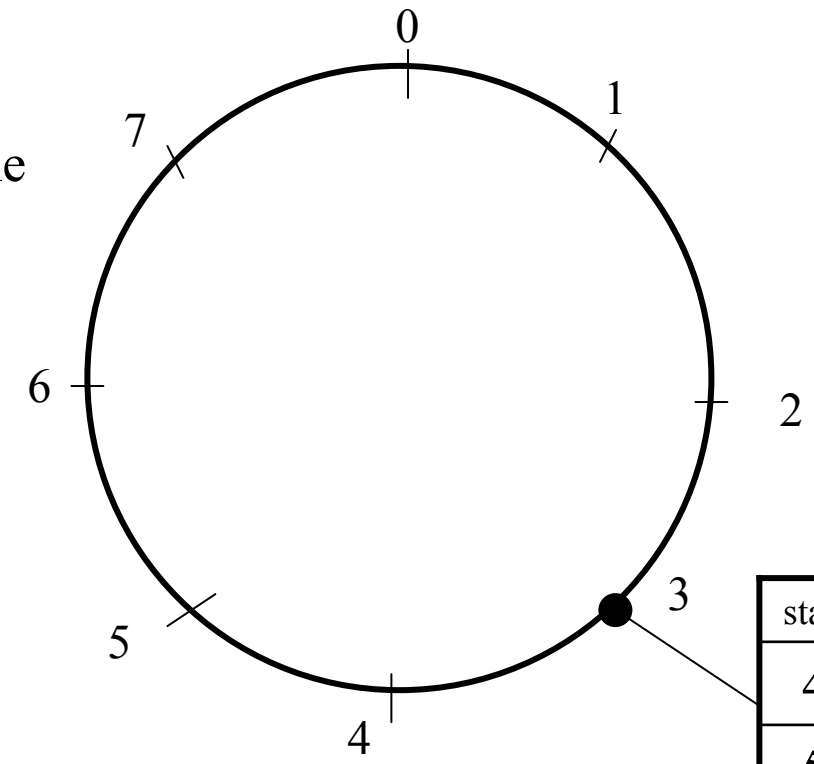
- Find the predecessor p of n
- Check if the i^{th} finger of p needs update
- The same check occurs to the predecessor of p for i^{th} finger
  - Recursively, do the same thing for all others
- The above steps are repeated for every i

Transferring Keys

- When a node n joins the network we have to move responsibility for all the keys for which node n is now the successor
- Exactly what this entails depends on the higher-layer software using Chord
- This would involve moving the data associated with each key to the new node

Node Join Example

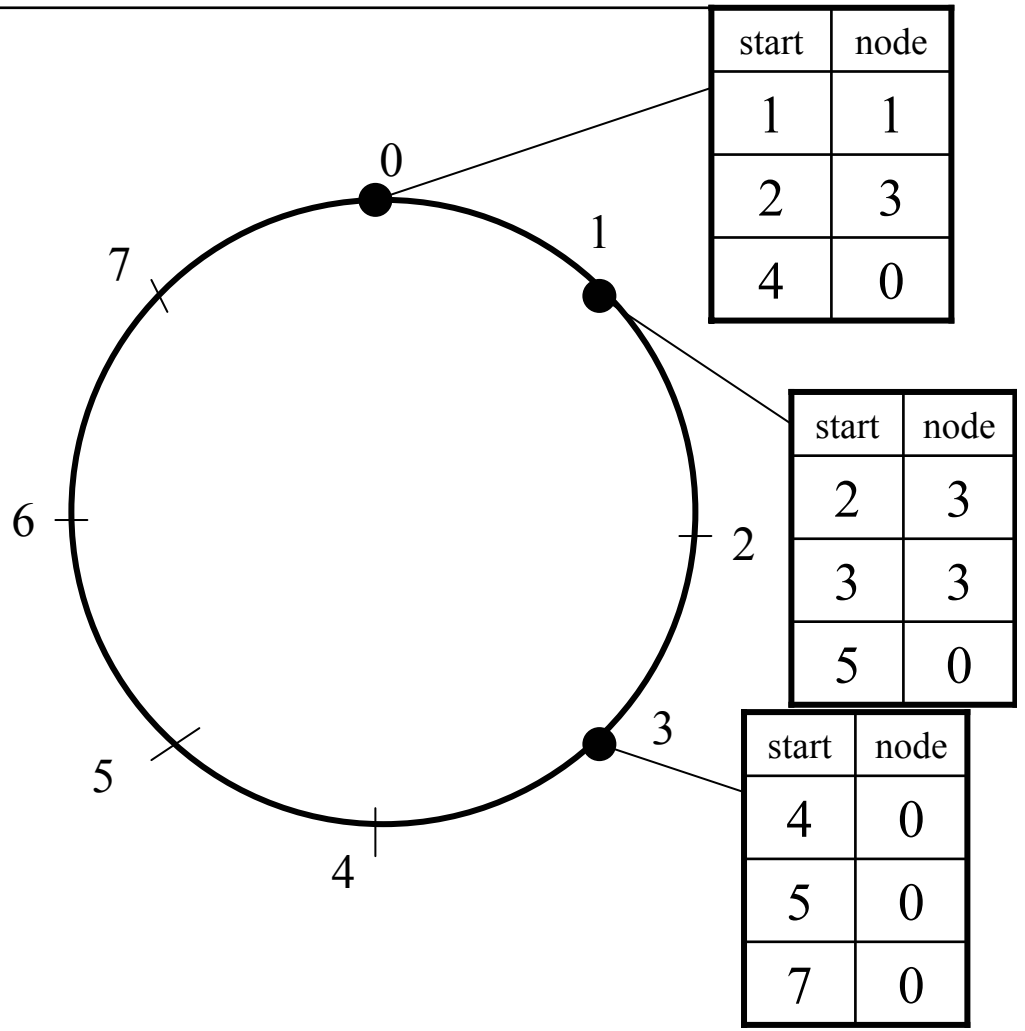
- 3 is the only node in the network
- 3's predecessor is 3
- 3's successor is 3
- The successor for all the keys in the finger table is 3



start	node
4	3
5	3
7	3

Node Join Example

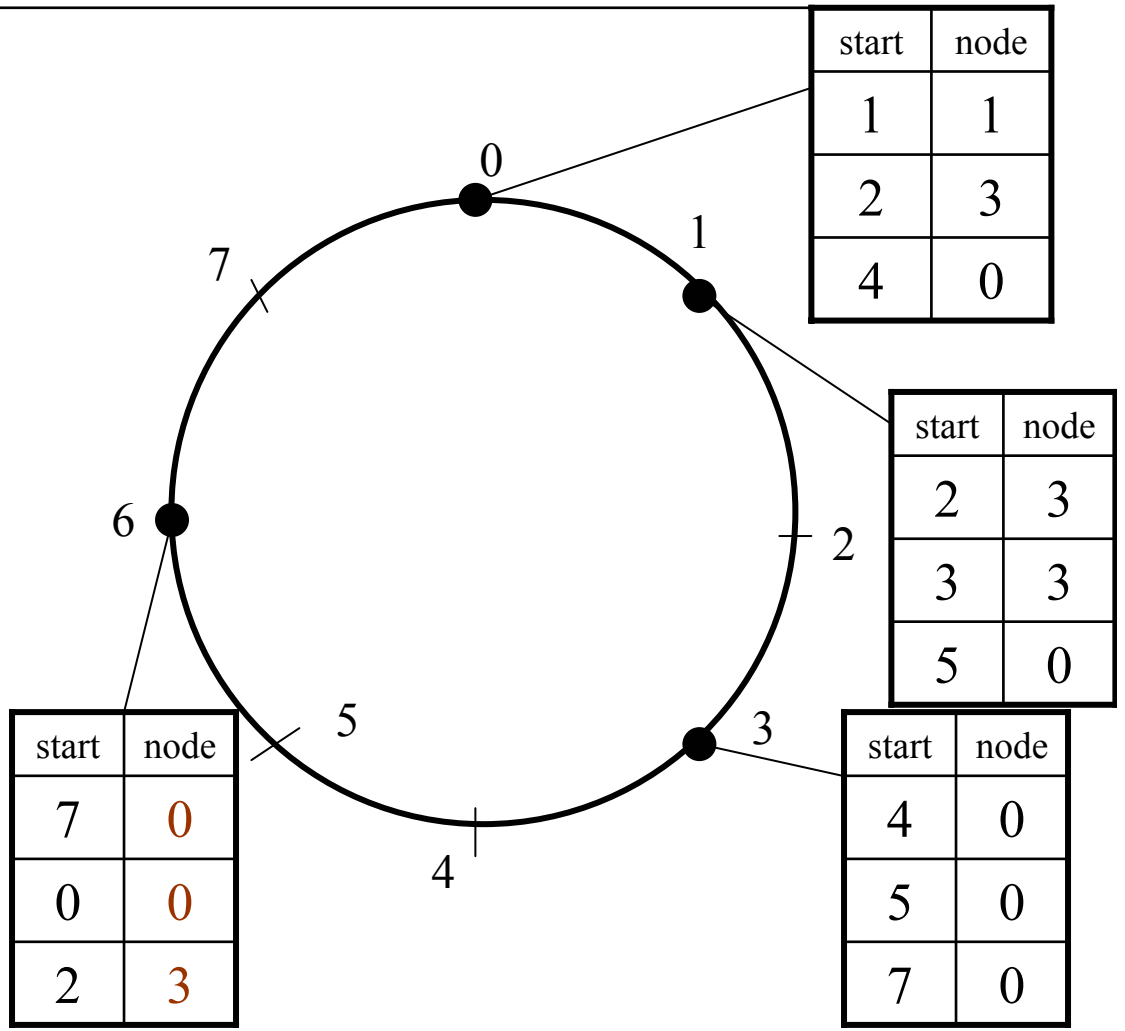
- Node 6 joins the network
- Node 0 helps 6's insertion by providing information
- Firstly, node 6 must find its successor which is 0
- 6's predecessor becomes 0's predecessor which is 3
- 0's predecessor becomes 6
- The next step is to initialize 6's finger table



Node Join Example

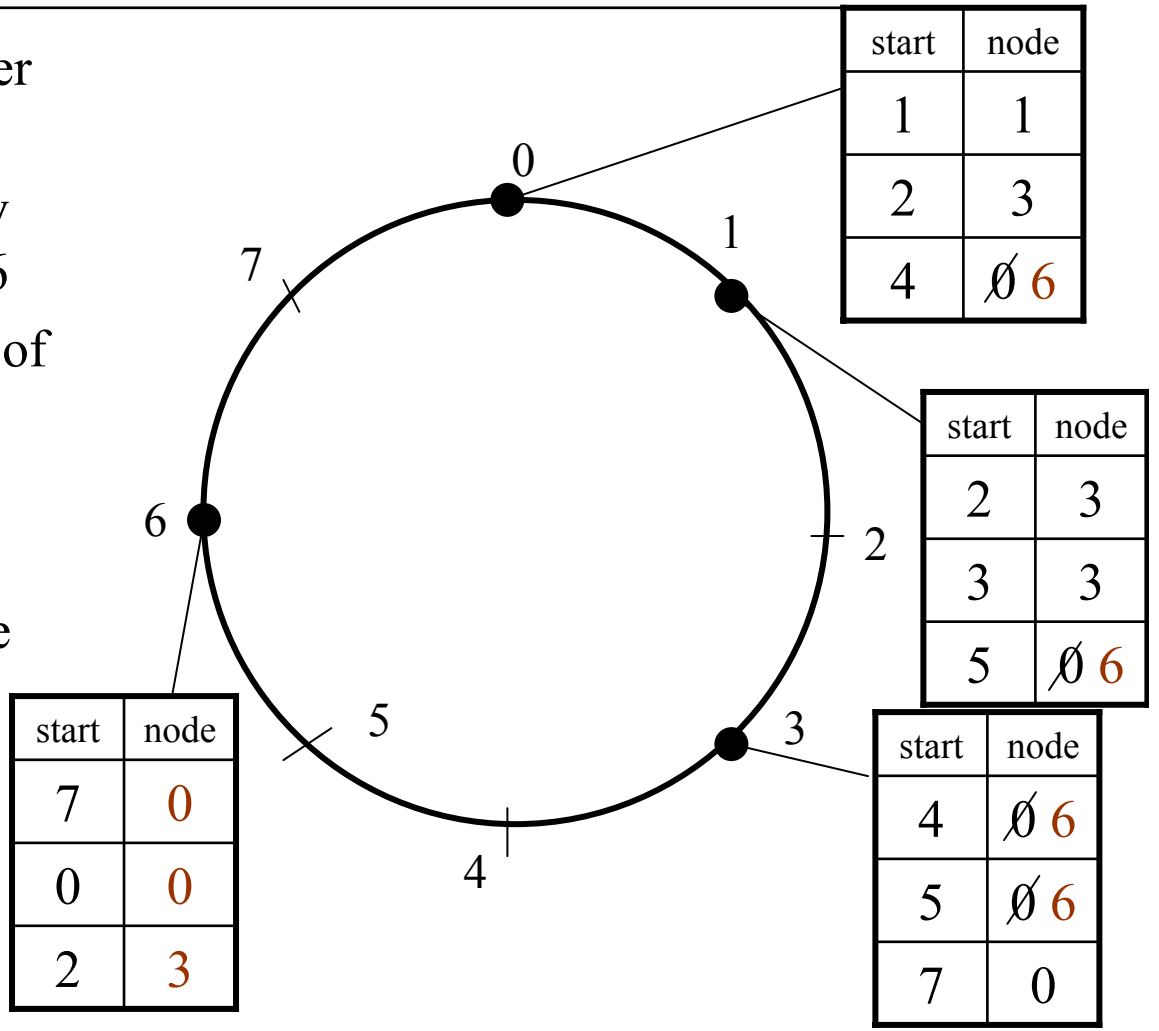
- The start column of 6's table is calculated using the formula $n+2^{i-1}$
- We calculate the successor of each finger
- The successor of id 7 and 0 is node 0
- The successor of id 2 is node 3

- **Note:** in case two sequential ids have the same successor, there is no need to calculate the second one



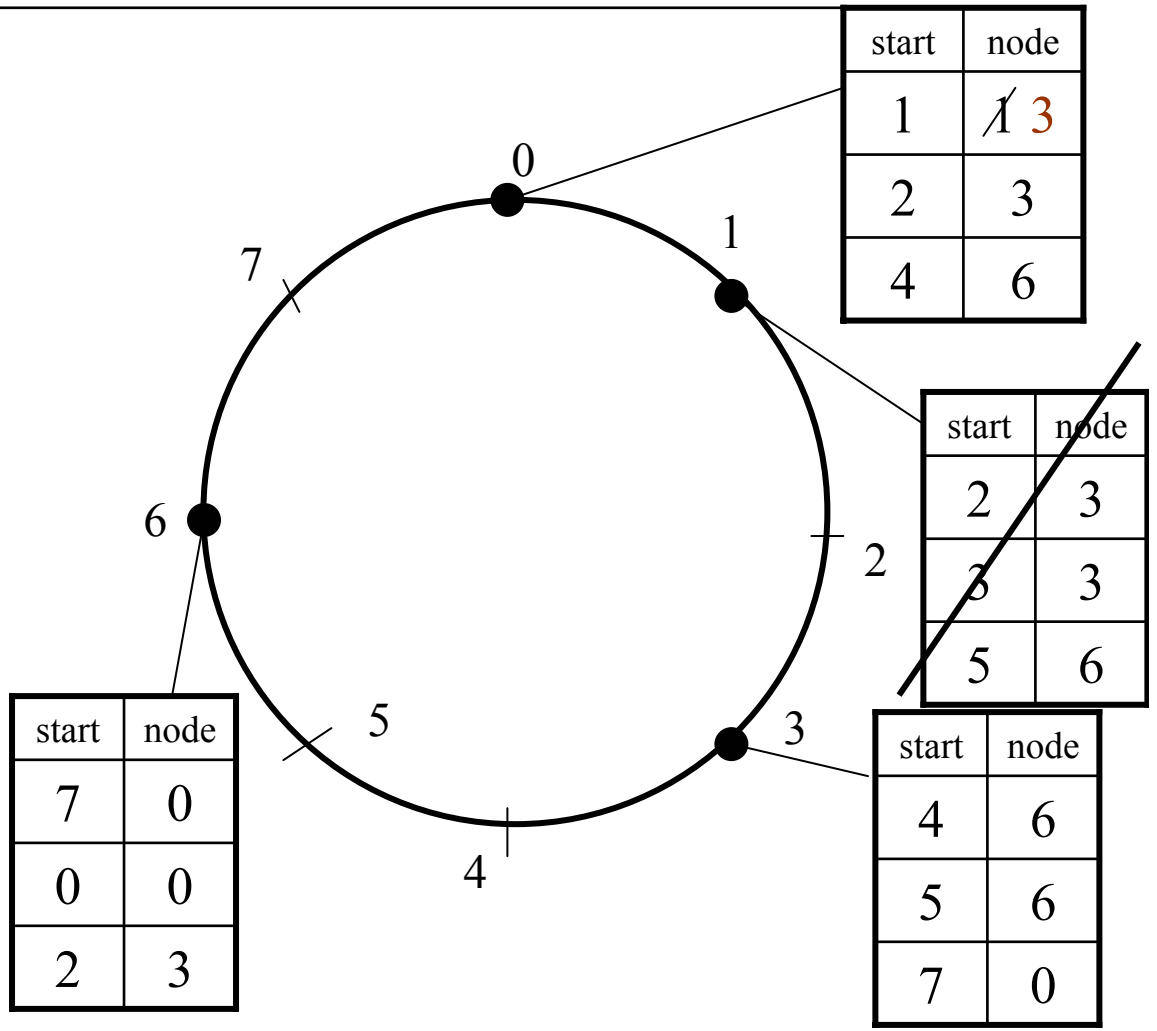
Node Join Example

- Here, the finger tables of other nodes are updated
- It checks the first finger entry of node 3. The new value is 6
- Then it checks the first entry of 3's predecessor, i.e., node 1
- This entry is not updated
- The same process is repeated for the 2nd and 3rd entry of the finger table



Node Leave Example

- Node 1 leaves the network
- All the keys that node 1 was responsible for, are now assigned to 1's successor, which is 3
- All the entries at the finger tables of the other nodes that have node 1 as a successor must be updated
- The new value is node 3 (1's successor)



The End