Chord - A scalable peer-to-peer lookup protocol

submitted for the course

ADVANCED COMPUTER NETWORKS (CSN-503)

by

Shubham Raj(15114069)

Gulshan Raj (15114030)

Kamal Kant (15114034)

S John S (15114060)

Sudhanshu Sambharya(15114071)

Under the guidance of

Dr. P. Sateesh Kumar

Assistant Professor

Department of Computer Science and Engineering

Indian Institute of Technology Roorkee

Roorkee, Uttarakhand- 247667, INDIA

November 5, 2018

Introduction

A fundamental problem that confronts peer-to-peer applications is to efficiently locate the node that stores a particular data item. Chord is a protocol and algorithm for a peer-to-peer distributed hash table. A distributed hash table stores key-value pairs by assigning keys to different computers (known as "nodes"), a node stores the values for all the keys for which it is responsible. Chord specifies how keys are assigned to nodes, and how a node can discover the value for a given key by first locating the node responsible for that key.

It was introduced in 2001 by Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan, and was developed at MIT.

System Model

Chord has the following features -

1. **Load Balance**
   Chord spreads keys evenly over nodes as it uses a distributed hash function.

2. **Decentralization**
   It is fully distributed. No node is more important than the others.

3. **Scalability**
   The cost of a Chord lookup grows as the log of the number of nodes, so even very large systems are feasible.

4. Availability

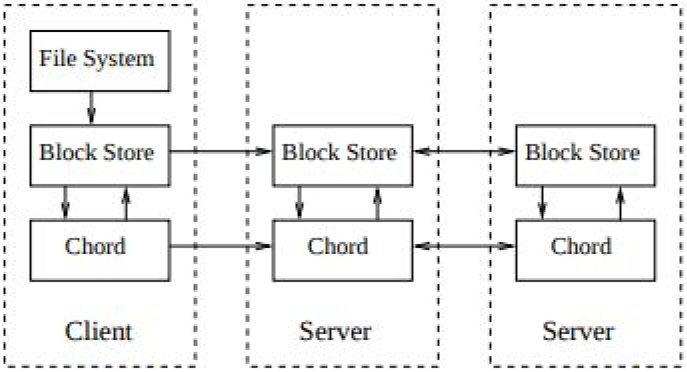
Chord automatically adjusts its internal tables to reflect newly joined nodes as well as node failures, ensuring that, barring major failures in the underlying network, the node responsible for a key can always be found. This is true even if the system is in a continuous state of change.

5. Flexible naming

Chord places no constraints on the structure of the keys it looks up: the Chord key-space is flat. This gives applications a large amount of flexibility in how they map their own names to Chord keys.

Architecture

The architecture of chord protocol performs two basic functions. First, the Chord library provides a lookup(key) function that yields the IP address of the node responsible for the key. Second, 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 software to, for example, move corresponding values to their new homes when a new node joins.



Basic structure of Chord based distributed storage system

Applications

1. Cooperative mirroring

It is useful in which multiple providers of content cooperate to store and serve each others' data.Spreading the total load evenly over all participants' hosts lowers the total cost of the system, since each participant need provide capacity only for the average load, not for that participant's peak load.

2 . Time-shared storage

In time-shared storage systems,if someone wishes their data to be always available, but their server is only occasionally available, they can offer to store others' data while they are connected, in return for having their data stored elsewhere when they are disconnected. The data's name can serve as a key to identify the (live) Chord node responsible for storing the data item at any given time.

3. Distributed indexes

To support Gnutella- or Napster-like keyword search. A key in this application could be derived from the desired keywords, while values could be lists of machines offering documents with those keywords.

4. Large-scale combinatorial search

In this case keys are candidate solutions to the problem (such as cryptographic keys). Chord maps these keys to the machines responsible for testing them as solutions.

Description

1. Overview

Chord provides fast distributed computation of a hash function mapping keys to nodes responsible for them.Chord assigns keys to nodes with consistent hashing which has several desirable properties such as scalability and load balancing. When a node leaves or joins the network, only a fraction of the keys are moved to a different location,thus ensuring load balancing.Chord improves the scalability of consistent hashing by avoiding the requirement that every node know about every other node. A node only maintains information about a small number of its neighbours.This information is distributed and for a network with n nodes,each node maintains information about ~(log n) other nodes, and thus lookup requires ~ (log n) messages.

2. Consistent Hashing

In traditional hash table, a change in number of array slots causes nearly all keys to be remapped because the mapping between the keys and the slots is defined by a modular operation. In contrast to these hash tables, consistent hashing is special kind of hashing such that when a hash table is resized, only K/n keys are needed to be remapped on average. Here, K is the number of keys, and n is the number of slots.

Chord uses consistent hashing to assign m- bit identifiers to nodes and keys. It is integral to the robustness and performance of Chord because both keys and node are uniformly distributed in the same identifier space with a negative possibility of collision, thus allowing nodes to join and leave the network without disruption. A common way of load balancing n nache machines is to put object o in cache machine number hash(o) (mod n). But it does no work if a cache machine is added or removed changing the value f n and all objects are haashed to new location. Consistent mapping hashes the objects to the same machine, as far as possible. Thus if a cache machine is added, it takes its share from all the other machines and when it's removed, its share is redistributed between all remaining ones.

The technique is based on mapping all the objects to a point on the edge of a circle (or, mapping each object to a real angle). The system maps each available machine to a pseudo randomly distributed points on the edge of the same circle. To find where an object is placed, the system finds the location of its key on he edge of the circle, then walks around the circle until it finds an empty bucket. A bucket contains all the resources located between each one of its points and the previous points that belong to other buckets. Therefore in case of removal of a bucket, the points it maps to will be removed. Request for the resources that would have mapped to any of those points will be mapped to next highest points.
The portion of keys that a bucket is mapped with can be altered by changing the number if angles that bucket maps to.

3. Scalable Key location

The most important usage of Chord protocol is to query a key from a client,i.e. to find successor(k). A simple approach is to pass the query to a node's successor, if it cannot find the key locally. This will lead to a O(N) query time where N is the number of machines in the circle.

Finger Table

To avoid the linear search time complexity in above situation, Chord implements a faster search method by requiring each node to keep a finger table containing up to m entries, m is the number of bits in the hash key. The i^{th} entry of the node n will contain successor $((n+2^{i-1}) \bmod 2^{m})$. The first entry of finger table is the node's immediate successor. Every time a node wants to look up a key k, it will pass the query to the closest successor or predecessor of in its finger table, until a node finds out the key is stored in its immediate successor. With the help of finger table, the number of nodes that must be reached to find a key in a N-node network is O(log N).

4. Node Join functionality

In a dynamic network, nodes can join (and leave) at any time. The main challenge in implementing these operations is preserving the ability to locate every key in the network. To achieve this goal, Chord needs to preserve two invariants :--

- Each node's successor is correctly maintained.
- For every key k, node successor(k) is responsible for k.

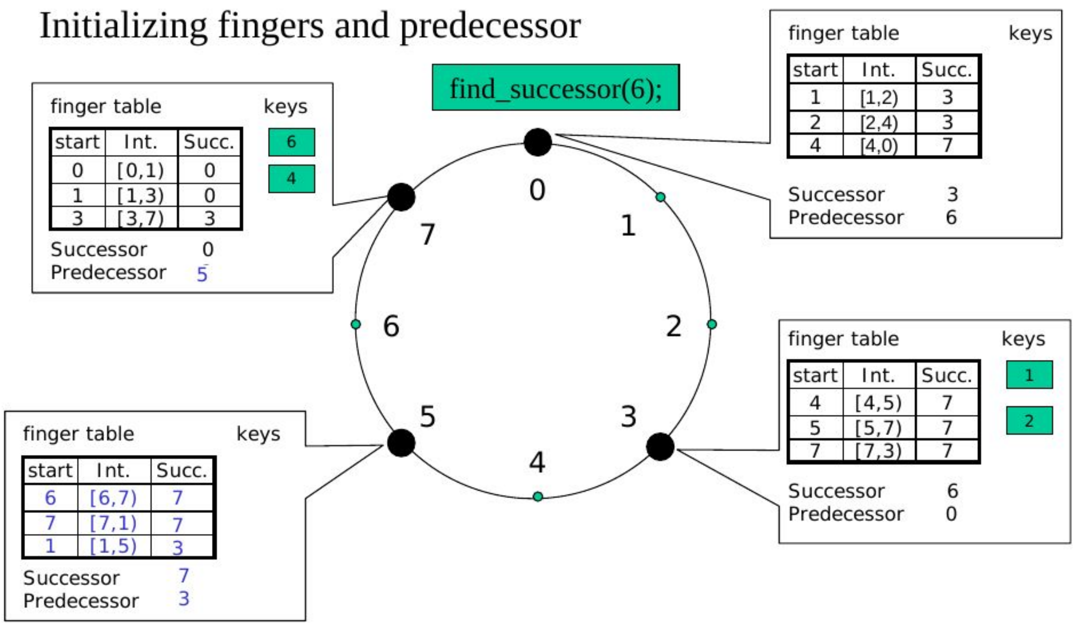
In order for lookups to be fast, it is also desirable for the finger tables to be correct.

With high probability, any node joining or leaving, an N-node Chord network will use O($\log^2$ N)
messages to re-establish the Chord routing invariants and finger tables.
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.

To preserve the invariants stated above, Chord must perform three tasks when a node joins the network:

1. Initialize the predecessor and fingers of node n.
2. Update the fingers and predecessors of existing nodes to reflect the addition of n .
3. Notify the higher layer software so that it can transfer state (e.g. values) associated with keys that node n is now responsible for.

Diagrammatic representation of the node join operation



Initializing fingers and predecessor

find_successor(6);

finger table keys

start	Int.	Succ.
0	[0,1)	0
1	[1,3)	0
3	[3,7)	3

Successor 0
Predecessor 5

keys: 6, 4

finger table keys

start	Int.	Succ.
1	[1,2)	3
2	[2,4)	3
4	[4,0)	7

Successor 3
Predecessor 6

finger table keys

start	Int.	Succ.
4	[4,5)	7
5	[5,7)	7
7	[7,3)	7

Successor 6
Predecessor 0

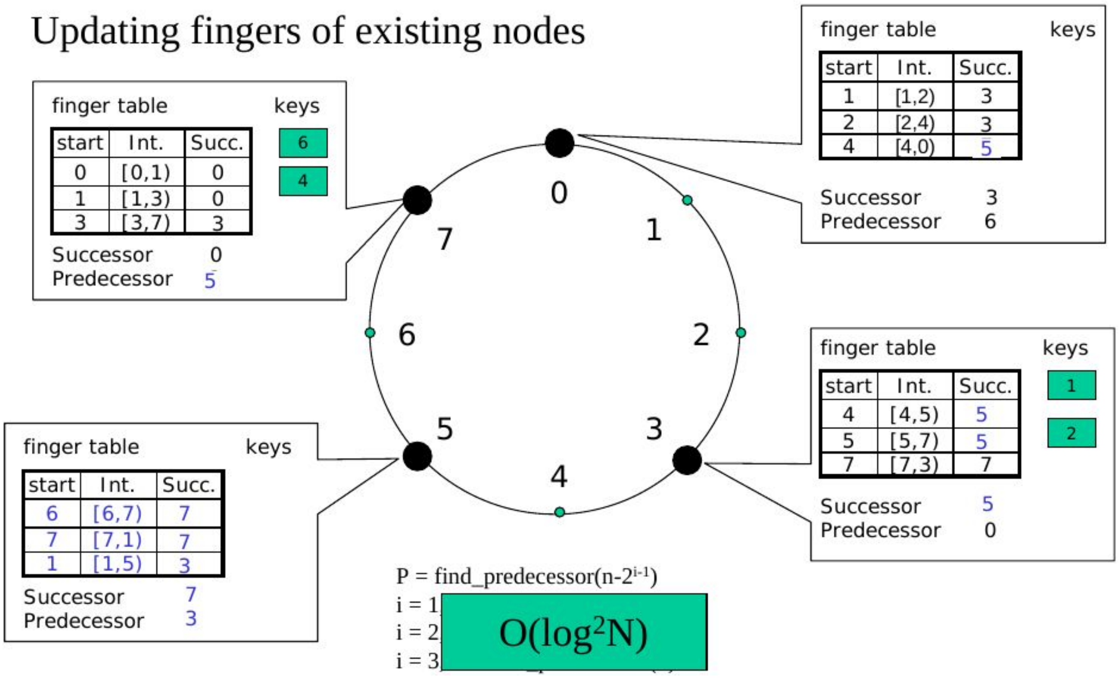
keys: 1, 2

finger table keys

start	Int.	Succ.
6	[6,7)	7
7	[7,1)	7
1	[1,5)	3

Successor 7
Predecessor 3

Updating fingers of existing nodes



finger table keys

start	Int.	Succ.
1	[1,2)	3
2	[2,4)	3
4	[4,0)	5

Successor 3
Predecessor 6

finger table keys 🟩6 🟩4

start	Int.	Succ.
0	[0,1)	0
1	[1,3)	0
3	[3,7)	3

Successor 0
Predecessor 5

finger table keys 🟩1 🟩2

start	Int.	Succ.
4	[4,5)	5
5	[5,7)	5
7	[7,3)	7

Successor 5
Predecessor 0

finger table keys

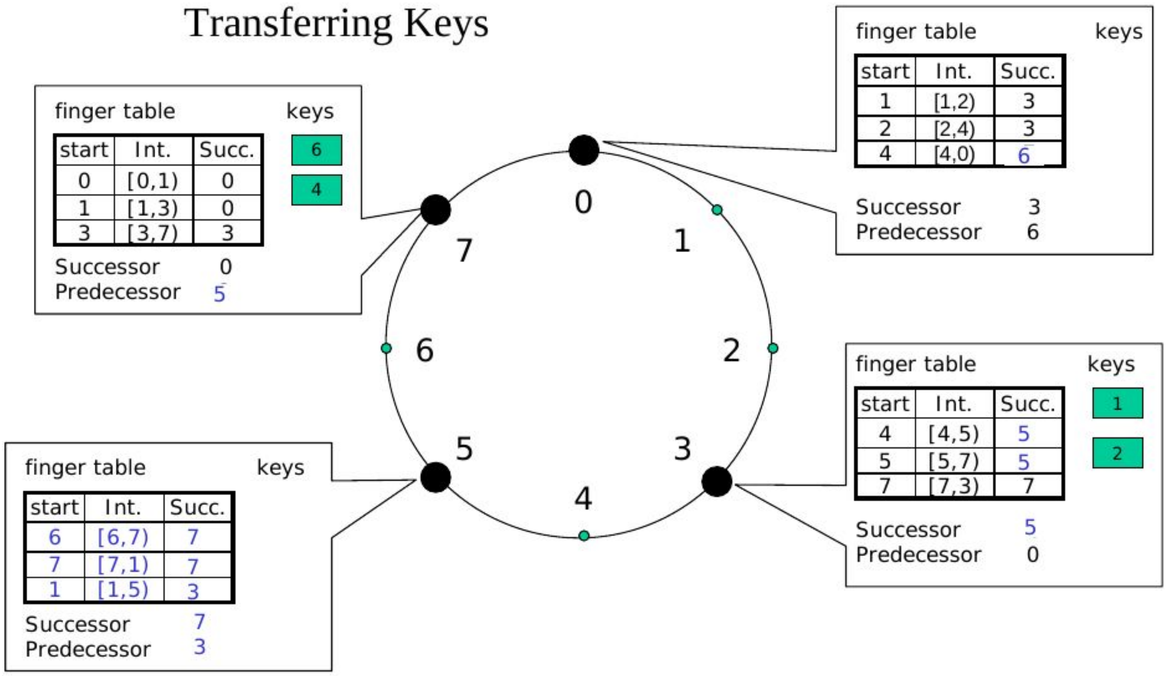
start	Int.	Succ.
6	[6,7)	7
7	[7,1)	7
1	[1,5)	3

Successor 7
Predecessor 3

P = find_predecessor($n-2^{i-1}$)
i = 1
i = 2
i = 3

O($\log^2 N$)

Transferring Keys



finger table keys

start	Int.	Succ.
1	[1,2)	3
2	[2,4)	3
4	[4,0)	6

Successor 3
Predecessor 6

finger table keys 🟩6 🟩4

start	Int.	Succ.
0	[0,1)	0
1	[1,3)	0
3	[3,7)	3

Successor 0
Predecessor 5

finger table keys 🟩1 🟩2

start	Int.	Succ.
4	[4,5)	5
5	[5,7)	5
7	[7,3)	7

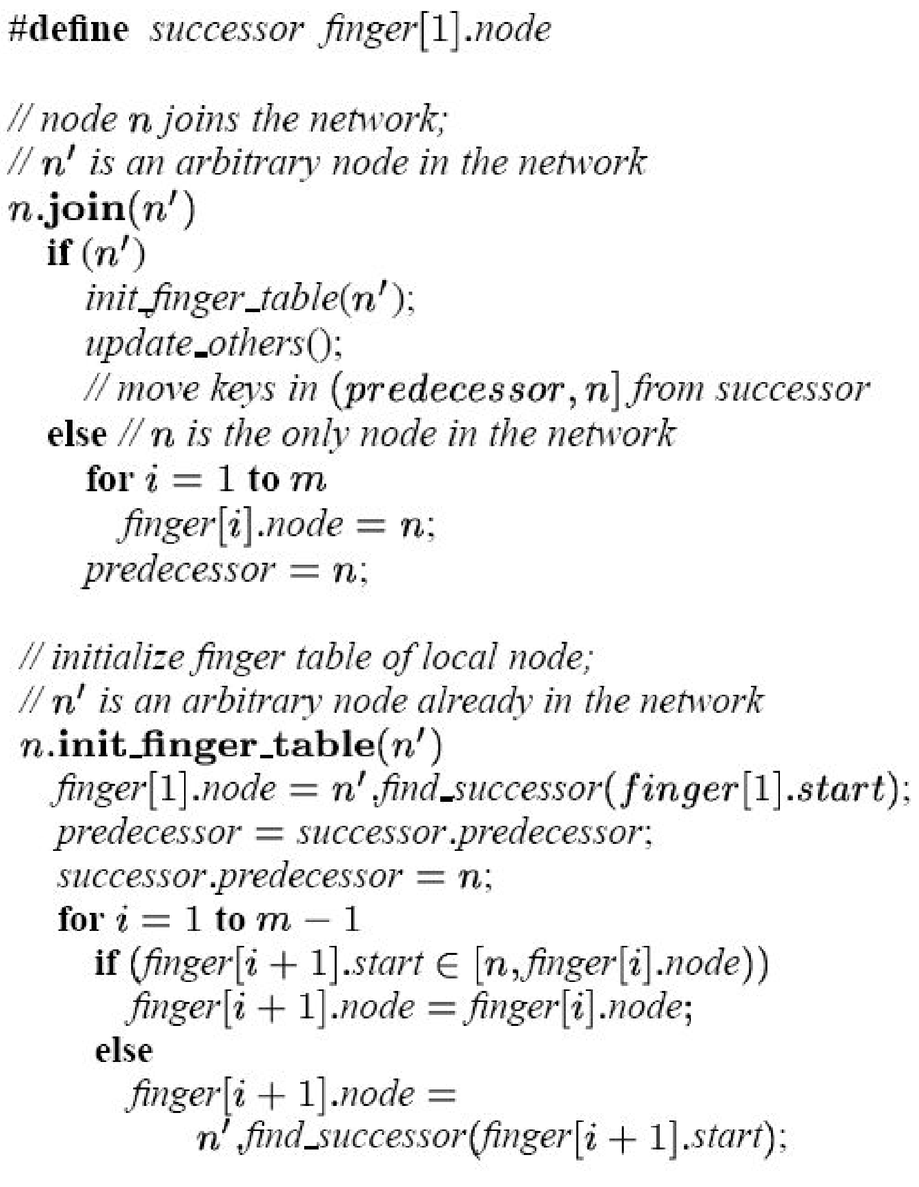
Successor 5
Predecessor 0

finger table keys

start	Int.	Succ.
6	[6,7)	7
7	[7,1)	7
1	[1,5)	3

Successor 7
Predecessor 3

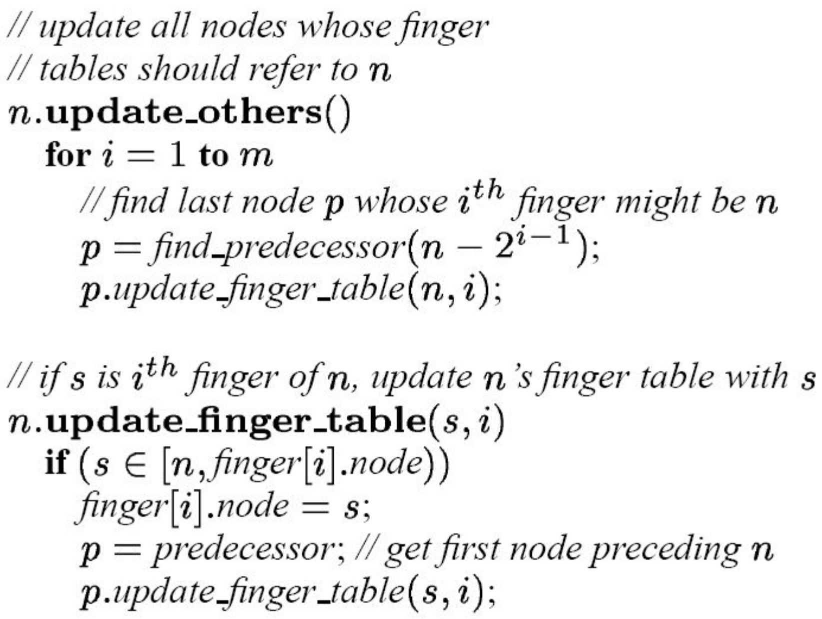
Pseudocode for the node join operation



```
#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 $(\mathbf{predecessor}, \mathbf{n}]$ 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($\boldsymbol{finger}[1].\boldsymbol{start}$);
  predecessor = successor.predecessor;
  successor.predecessor = $n$;
  for $i = 1$ to $m - 1$
    if (finger[$i + 1$].start $\in$ [$n$, 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 ($s \in [n$, finger[$i$].node))
    finger[$i$].node = $s$;
    $p$ = predecessor; // get first node preceding $n$
    $p$.update_finger_table($s, i$);
```

5. Stabilization

Stabilization helps maintain the correctness of the algorithm without compromising with the algorithm's performance. A basic stabilization protocol is used to keep nodes' successor pointers up to date, which is sufficient to guarantee correctness of lookups. Those successor pointers are then used to verify and correct finger table entries, which allows these lookups to be fast as well as correct.

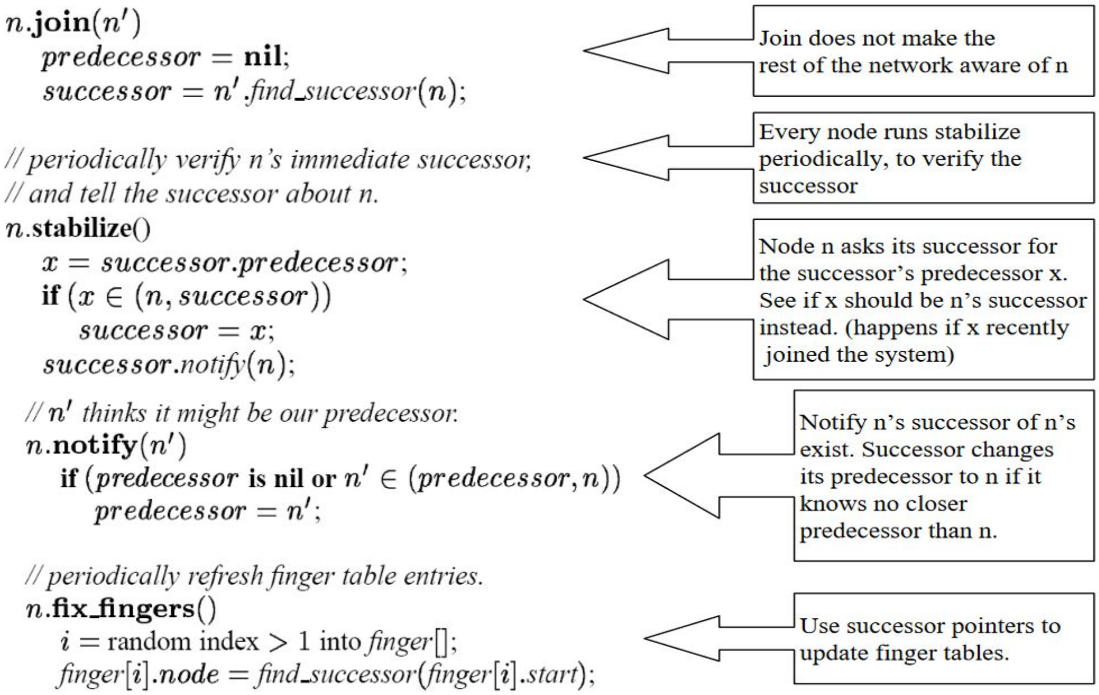
The used stabilization scheme adds nodes to a Chord ring in a way that preserves reachability of existing nodes, even in the face of concurrent joins and lost and reordered messages.

The stabilization algorithm handles concurrent joins. Every node runs stabilize periodically (this is how newly joined nodes are noticed by the network). When a node 'n' runs stabilize, it asks n's successor for the successor's predecessor p, and decides whether p should be n 's successor instead. This would be the case if node p recently joined the system. Stabilize also notifies node n's successor of n's existence, giving the successor the chance to change its predecessor to n. The successor does this only if it knows of no closer predecessor than n.

As soon as the successor pointers are correct, calls to find_predecessor (and thus find_successor function) function will work.

Also it turns out that joins don't substantially damage the performance of fingers. New joins influence the lookup only by getting in between the old predecessor and successor of a target query. Unless a large number of nodes joins the system, the number of nodes between two old nodes is likely to be very small, so the impact on lookup is negligible.

Pseudo-code for Stabilization:



```
$n$.join($n'$)
  $predecessor = \mathbf{nil}$;
  $successor = n'.find\_successor(n)$;

// periodically verify n's immediate successor,
// and tell the successor about n.
$n$.stabilize()
  $x = successor.predecessor$;
  if ($x \in (n, successor)$)
    $successor = x$;
  $successor.notify(n)$;

// $n'$ thinks it might be our predecessor.
$n$.notify($n'$)
  if ($predecessor$ is nil or $n' \in (predecessor, n)$)
    $predecessor = n'$;

// 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.

Summary

- ▷ Characteristics of Chord
  - Load balance

    distributed hash table
  - Decentralization

    fully distributed
  - Scalability

    cost of lookup grows logarithmic

Routing Hops	O(logN)
Arrival	O(log^{2}N)
Departure	O(log^{2}N)

  - Availability

    automatically adjusts internal tables
  - Flexible naming

    no constrains on the structure of the keys

Attractive features of Chord include its simplicity, provable correctness, and provable performance even in the face of concurrent node arrivals and departures. It continues to function correctly, albeit at degraded performance, when a node's information is only partially correct.

Chord scales well with the number of nodes, recovers from large numbers of simultaneous node failures and joins, and answers most lookups correctly even during recovery.