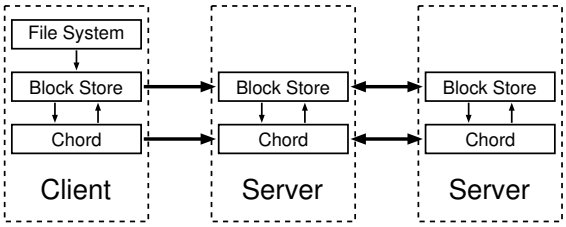
# Chord (peer-to-peer)

In computing, **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 will store 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.

Chord is one of the four original distributed hash table protocols, along with CAN, Tapestry, and Pastry. It was introduced in 2001 by Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan, and was developed at MIT.[1]

## 1 Overview




Nodes and keys are assigned an

$m$

-bit *identifier* using *consistent hashing*. The SHA-1 algorithm is the base hashing function for consistent hashing. Consistent hashing is integral to the robustness and performance of Chord because both keys and nodes (in fact, their IP addresses) are uniformly distributed in the same identifier space with a negligible possibility of collision. Thus, it also allows nodes to join and leave the network without disruption. In the protocol, the term *node* is used to refer to both a node itself and its identifier (ID) without ambiguity. So is the term *key*.

Using the Chord lookup protocol, nodes and keys are arranged in an identifier circle that has at most

$2^m$

nodes, ranging from

$0$

to

$2^m - 1$

. (

$m$

should be large enough to avoid collision.) Some of these nodes will map to machines or keys while others (most) will be empty.

Each node has a *successor* and a *predecessor*. The successor to a node is the next node in the identifier circle in a clockwise direction. The predecessor is counter-clockwise. If there is a node for each possible ID, the successor of node 0 is node 1, and the predecessor of node 0 is node

$2^m - 1$

; however, normally there are "holes" in the sequence. For example, the successor of node 153 may be node 167 (and nodes from 154 to 166 do not exist); in this case, the predecessor of node 167 will be node 153.

The concept of successor can be used for keys as well. The *successor node* of a key

$k$

is the first node whose ID equals to

$k$

or follows

$k$

in the identifier circle, denoted by

$successor(k)$

. Every key is assigned to (stored at) its successor node, so looking up a key

$k$

is to query

$successor(k)$

.

Since the successor (or predecessor) of a node may disappear from the network (because of failure or departure), each node records a whole segment of the circle adjacent to it, i.e., the

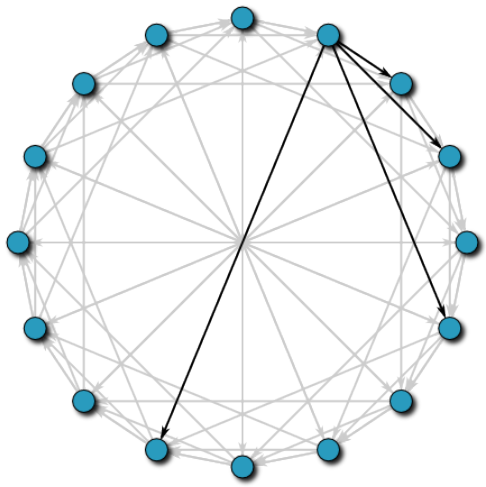
$r$

nodes preceding it and the

$r$

nodes following it. This list results in a high probability that a node is able to correctly locate its successor or predecessor, even if the network in question suffers from a high failure rate.

# 2 Protocol details



*A 16-node Chord network. The "fingers" for one of the nodes are highlighted.*

## 2.1 Basic query

The core usage of the Chord protocol is to query a key from a client (generally a node as well), i.e. to find

$successor(k)$

. The basic 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 ring.

## 2.2 Finger table

To avoid the linear search above, Chord implements a faster search method by requiring each node to keep a *finger table* containing up to

$m$

entries, recall that

$m$

is the number of bits in the hash key. The

$i^{th}$

entry of node

$n$

will contain

$successor((n + 2^{i-1}) \bmod 2^m)$

. The first entry of finger table is actually the node's immediate successor (and therefore an extra successor field is not needed). Every time a node wants to look up a key

$k$

, it will pass the query to the closest successor or predecessor (depending on the finger table) of

$k$

in its finger table (the "largest" one on the circle whose ID is smaller than

$k$

), until a node finds out the key is stored in its immediate successor.

With such a finger table, the number of nodes that must be contacted to find a successor in an *N*-node network is

$O(\log N)$

. (See proof below.)

## 2.3 Node join

Whenever a new node joins, three invariants should be maintained (the first two ensure correctness and the last one keeps querying fast):

1. Each node's successor points to its immediate successor correctly.
2. Each key is stored in $successor(k)$ .
3. Each node's finger table should be correct.

To satisfy these invariants, a *predecessor* field is maintained for each node. As the successor is the first entry of the finger table, we do not need to maintain this field separately any more. The following tasks should be done for a newly joined node

$n$

:

1. Initialize node $n$ (the predecessor and the finger table).
2. Notify other nodes to update their predecessors and finger tables.
3. The new node takes over its responsible keys from its successor.

The predecessor of

$n$

can be easily obtained from the predecessor of

$successor(n)$

(in the previous circle). As for its finger table, there are various initialization methods. The simplest one is to execute find successor queries for all

$m$

entries, resulting in

$O(M \log N)$

initialization time. A better method is to check whether

$i^{th}$

entry in the finger table is still correct for the

$(i+1)^{th}$

entry. This will lead to

$O(\log^2 N)$

. The best method is to initialize the finger table from its immediate neighbours and make some updates, which is

$O(\log N)$

.

### 2.4 Stabilization

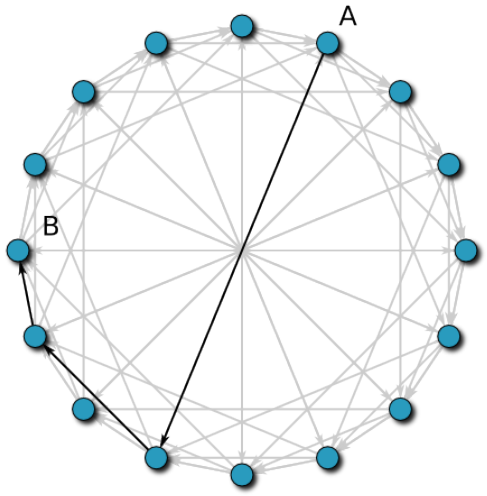
To ensure correct lookups, all successor pointers must be up to date. => stabilization protocol running periodically in the background. Updates finger tables and successor pointers. Stabilization protocol: Stabilize(): n asks its successor for its predecessor p and decides whether p should be n‘s successor instead (this is the case if p recently joined the system). Notify(): notifies n‘s successor of its existence, so it can change its predecessor to n Fix_fingers(): updates finger tables

### 2.5 Failures and replication

## 3 Potential uses

- Cooperative Mirroring: A load balancing mechanism by a local network hosting information available to computers outside of the local network. This scheme could allow developers to balance the load between many computers instead of a central server to ensure availability of their product.
- Time-shared storage: In a network, once a computer joins the network its available data is distributed throughout the network for retrieval when that computer disconnects from the network. As well as other computers' data is sent to the computer in question for offline retrieval when they are no longer connected to the network. Mainly for nodes without the ability to connect full-time to the network.
- Distributed Indices: Retrieval of files over the network within a searchable database. e.g. P2P file transfer clients.
- Large scale combinatorial searches: Keys being candidate solutions to a problem and each key mapping to the node, or computer, that is responsible for evaluating them as a solution or not. e.g. Code Breaking

## 4 Proof sketches




*The routing path between nodes A and B. Each hop cuts the remaining distance in half (or better).*

**With high probability, Chord contacts**

$O(\log N)$

**nodes to find a successor in an**

$N$

**-node network.**

Suppose node

$n$

wishes to find the successor of key

$k$

. Let

$p$

be the predecessor of

$k$

. We wish to find an upper bound for the number of steps it takes for a message to be routed from

$n$

to

$p$

. Node

$n$

will examine its finger table and route the request to the closest predecessor of

$k$

that it has. Call this node

$f$

. If

$f$

is the

$i^{th}$

entry in

$n$

's finger table, then both

$f$

and

$p$

are at distances between

$2^{i-1}$

and

$2^i$

from

$n$

along the identifier circle. Hence, the distance between

$f$

and

$p$

along this circle is at most

$2^{i-1}$

. Thus the distance from

$f$

to

$p$

is less than the distance from

$n$

to

$f$

: the new distance to

$p$

is at most half the initial distance.

This process of halving the remaining distance repeats itself, so after

$t$

steps, the distance remaining to

$p$

is at most

$2^m/2^t$

; in particular, after

$\log N$

steps, the remaining distance is at most

$2^m/N$

. Because nodes are distributed uniformly at random along the identifier circle, the expected number of nodes falling within an interval of this length is 1, and with high probability, there are fewer than

$\log N$

such nodes. Because the message always advances by at least one node, it takes at most

$\log N$

steps for a message to traverse this remaining distance. The total expected routing time is thus

$O(\log N)$

.

If Chord keeps track of

$r = O(\log N)$

predecessors/successors, then with high probability, if each node has probability of 1/4 of failing, find_successor (see below) and find_predecessor (see below) will return the correct nodes

Simply, the probability that all

$r$

nodes fail is

$\left(\frac{1}{4}\right)^r = O\left(\frac{1}{N}\right)$

, which is a low probability; so with high probability at least one of them is alive and the node will have the correct pointer.

# 5 Pseudocode

**Definitions for pseudocode finger[k]** first node that succeeds $(n + 2^{k-1}) \bmod 2^m, 1 \leq k \leq m$

**successor** the next node from the node in question on the identifier ring

**predecessor** the previous node from the node in question on the identifier ring

The pseudocode to find the *successor* node of an id is given below:

// ask node n to find the successor of id n.find_successor(id) //Yes, that should be a closing square bracket to match the opening parenthesis.

//It is a half closed interval. if (id ∈ (n, successor] ) return successor; else // forward the query around the circle n0 = closest_preceding_node(id); return n0.find_successor(id); // search the local table for the highest predecessor of id n.closest_preceding_node(id) for i = m downto 1 if (finger[i]∈(n,id)) return finger[i]; return n;

The pseudocode to stabilize the chord ring/circle after node joins and departures is as follows:

// create a new Chord ring. n.create() predecessor = nil; successor = n; // join a Chord ring containing node n'. n.join(n') predecessor = nil; successor = n'.find_successor(n); // called periodically. n asks the successor // about its predecessor, verifies if n's immediate // successor is consistent, and tells 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'; // called periodically. refreshes finger table entries. // next stores the index of the finger to fix n.fix_fingers() next = next + 1; if (next > m) next = 1; finger[next] = find_successor(n+ $2^{next-1}$ ); // called periodically. checks whether predecessor has failed. n.check_predecessor() if (predecessor has failed) predecessor = nil;

## 6 See also

- Kademlia
- Koorde
- OverSim - the overlay simulation framework
- SimGrid - a toolkit for the simulation of distributed applications -

## 7 References

[1] Stoica, I.; Morris, R.; Karger, D.; Kaashoek, M. F.; Balakrishnan, H. (2001). "Chord: A scalable peer-to-peer lookup service for internet applications" (PDF). *ACM SIGCOMM Computer Communication Review*. **31** (4): 149. doi:10.1145/964723.383071.

## 8 External links

- The Chord Project (redirect from: http://pdos.lcs.mit.edu/chord/)
- Open Chord - An Open Source Java Implementation
- Chordless - Another Open Source Java Implementation
- jDHTUQ- An open source java implementation. API to generalize the implementation of peer-to-peer DHT systems. Contains GUI in mode data structure

# 9 Text and image sources, contributors, and licenses

## 9.1 Text

- **Chord (peer-to-peer)** *Source:* https://en.wikipedia.org/wiki/Chord_(peer-to-peer)?oldid=779919798 *Contributors:* AxelBoldt, M~enwiki, Ahoerstemeier, LittleDan, Nikai, Darkov, Dcoetzee, TittoAssini, DavidCary, Neilc, Quadell, MementoVivere, Kate, MeltBanana, Surachit, BrokenSegue, Unquietwiki, Beniz~enwiki, Mdd, Mgrinich, CyberSkull, Nkour, Mindmatrix, GregorB, FlaBot, AL SAM, MithrandirMage, Bgwhite, YurikBot, Sparky132, Nethgirb, Cedar101, That Guy, From That Show!, KnightRider~enwiki, SmackBot, Karmastan, Gilliam, TripleF, OrphanBot, Lobner, Mnulph, At2000, Dalstadt, M.B~enwiki, Wafulz, Toufeeq, Vectro, Cydebot, Blaisorblade, Epbr123, AntiVandalBot, BigChicken, Frank1634, Duncan.Hull, Bpringlemeir, Kbrose, Mcaramel, Sun Creator, Miami33139, Addbot, Yobot, Chauttm, Uhdeagle, Miym, I dream of horses, Arndt92, Raptium, Stjones86, Dewritech, Tetrashima, Chai Wei Jie, ClueBot NG, Rjloura, Sweet tea van, BG19bot, ChrisGualtieri, Dexbot, Shurakai, Sidsaurb, Hexcles and Anonymous: 79

## 9.2 Images

- **File:Chord_network.png** *Source:* https://upload.wikimedia.org/wikipedia/commons/2/20/Chord_network.png *License:* CC BY-SA 3.0 *Contributors:* Own work *Original artist:* Seth Terashima (Tetra7 (<a href='//commons.wikimedia.org/wiki/User_talk:Tetra7' title='User talk:Tetra7'>talk</a>))
- **File:Chord_project.svg** *Source:* https://upload.wikimedia.org/wikipedia/commons/1/16/Chord_project.svg *License:* CC BY-SA 3.0 *Contributors:* Own work based on: en:Image:Chord project.png by Surachit. *Original artist:* MithrandirMage
- **File:Chord_route.png** *Source:* https://upload.wikimedia.org/wikipedia/commons/e/e1/Chord_route.png *License:* CC BY-SA 3.0 *Contributors:* Own work *Original artist:* Seth Terashima (Tetra7 (<a href='//commons.wikimedia.org/wiki/User_talk:Tetra7' title='User talk:Tetra7'>talk</a>))

## 9.3 Content license

- Creative Commons Attribution-Share Alike 3.0