

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

Peer-to-peer systems and applications are distributed systems without any centralized control or hierarchical organization, where the software running at each node is equivalent in functionality. Possible fields of applications for peer-to-peer systems yield a long list: redundant storage, permanence, selection of nearby servers, anonymity, search, authentication, and hierarchical naming. Despite this rich set of features, the core operation in most peer-to-peer systems is efficient location of data items. The contribution of Chord is a scalable protocol for lookup in a dynamic peer-to-peer system with frequent node arrivals and departures.

In general Chord achieves the following properties:

- With **consistent hashing** Chord achieves load balancing. Each node is responsible for $O(\frac{K}{N})$ keys (K is the number of keys; N is the number of nodes).
- With the extra routing information kept in **finger tables** in each node Chord achieves $O(\log N)$ complexity for lookup cost.
- Chord is very efficient in the entrance or the departure of nodes in the system regarding the total messages that need to be exchanged between the nodes. The message complexity is bounded by $O(\log^2 N)$ messages.
- Chord is fully decentralized.

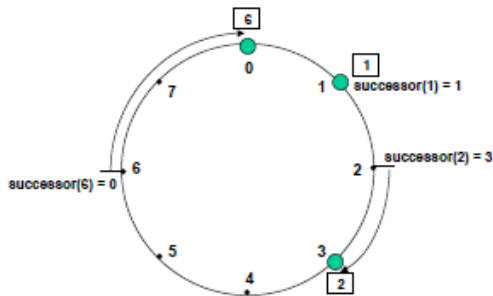
Chord operates as a Distributed Hash Table (DHT). The **Chord protocol** supports just one operation: given a key, it maps the key onto a node. Depending on the application using Chord, that node might be responsible for storing a value associated with the key. Chord uses a variant of **consistent hashing** to assign keys to Chord nodes. Consistent hashing tends to balance load, since each node receives roughly the same number of keys.

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, while a key identifier is produced by hashing the key. The identifier's length must be large enough to make the probability of two nodes or keys hashing to the same identifier negligible.

Consistent hashing assigns keys to nodes as follows. Identifiers of nodes are ordered in an identifier circle $\text{SHA-1}(\text{Node's IP address}) \bmod 2^m$. Key k is assigned to the first node whose identifier is equal to or follows (the identifier of) k in the identifier space. This node is called the *successor node* of key k, denoted by **successor (k)**. 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. It is theoretically proved that with consistent hashing, with N nodes and K keys every node is responsible for $O(\frac{K}{N})$.

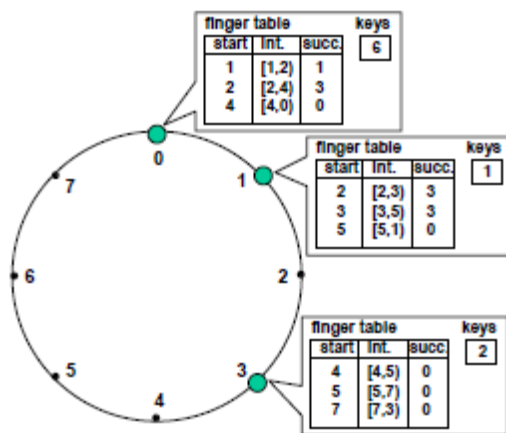
Consistent hashing is designed to let nodes enter and leave the network with minimal disruption. To maintain the consistent hashing mapping when a node n joins the

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. For example, as shown in figure if a node were to join with identifier 7, it would capture the key with identifier 6 from the node with identifier 0. The departure or arrival of a node demands $O(\log^2 N)$ messages.



A very small amount of routing information suffices to implement consistent hashing in a distributed environment. 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; this is the node the query maps to. A portion of the Chord protocol maintains these successor pointers, thus ensuring that all lookups are resolved correctly. However, this resolution scheme is inefficient: it may require traversing all N nodes to find the appropriate mapping. To accelerate this process, Chord maintains additional routing information. This additional information is not essential for correctness, which is achieved as long as the successor information is maintained correctly. The routing table kept in each node is called **finger table** and is basically a use of the skip list data structure in a peer to peer system.

The finger table is a matrix that keeps in the i th entry the address of the node that is at least 2^{i-1} nodes away from the current node. For example let assume a configuration with nodes 0, 1 and 3 as shown in the following figure.



The first node will have at the first entry of its finger table the node that succeeds at least $2^{i-1} = 2^0 = 1$ in the modulo 2^m circle. In this case it will be node 1. At its second entry it will contain the node that succeeds at least $2^{i-1} = 2^{2-1} = 2$. In this

case it is node 3. Finally in its third entry it will contain the address of the node that succeeds at least 4 that is node 0.

This scheme has two important characteristics. First, each node stores information about only a small number of other nodes, and knows more about nodes closely following it on the identifier circle than about nodes farther away. Second, a node's finger table generally does not contain enough information to determine the successor of an arbitrary key.