# Extensible Chord Distributed Hash Table

Evelina Dumitrescu
Politehnica University of Bucharest
Bucharest, Romania
Email: evelina.a.dumitrescu@gmail.com
Alexandru Tudorica
Politehnica University of Bucharest
Bucharest, Romania
Email: tudalex@gmail.com

***Abstract*—The major problem in integrating a P2P system in existing software is often one related to the existing architecture and design of the system. This paper proposes a way of implementing a Distributed Hash Table based on the Chord protocol that is highly configurable in order to fit into multiple design criteria.**

**Keywords:** peer to peer, distributed hash table, chord, integration, architecture

## I. Introduction

A peer-to-peer system is a distributed application architecture that divides the workloads across a number of peers. Each peer has the same privileges and decision power as the rest. Hence a peer-to-peer system doesn't have a centralized point of control nor does it have a hierarchical organization. Peer-to-peer systems have many uses: authentication, anonymity, data distribution, data storage, search. But all of this systems rely on the base property of efficiently and reliably locating data in the network despite structural network changes (nodes leaving, network partitioning, etc.).

A Distributed Hash Table works like a normal Hash Table, given a key it returns the information stored for that key. Therefore a P2P system is a great fit for a Distributed Hash Table, thus being the simplest system that can be built upon a P2P architecture and used as a building block for more complex systems.

The Chord protocol proposed by Stoica et al.[1] provides a DHT implementation that allows the creation of a key space of $2^m$ with up to the same number of nodes. At the root of the key lookup lies a consistent hashing algorithm that is able to take any data and map it to the key space. The proposed algorithm is SHA1. Each node has an id mapped in the key space, this node stores all the keys that are between it's predecessor's id and his. The identifiers are represented as a circle of number, such that the successor for each key is found by going in a clockwise direction, so every key has a successor node.

Faster lookups are provided in $O(log^2 N)$ using a special routing table named *finger table*. As opposed to other DHT protocols the *finger table* needs to store only $O(logN)$ entries where N is the number of nodes. The routing table provides a best effort routing, since nodes can drop from the network without prior warning. Also it's costly since nodes joining and leaving require changes to be made in the finger tables of approximately all the nodes in the network. So the Chord architecture, as defined in the original proposal is efficient for networks which are mostly stable, but overtime there have been adaptations to the Chord protocol in order to overcome this as can bee seen in the next section.

The purpose of this paper is to create a DHT based upon the Chord concept that offers a ready to use and simple experience to it's users, fault tolerant and fast. It uses optimizations based on the F-Chord system for fast uniform routing and optimizations explained in Hybrid Chord protocol in order to adapt to fast changing networks like the ones present in a P2P opportunistic network.

The rest of the paper is structured as follows. Section II presents the existing work related to the design of DHTs based on Chord. Section III presents the architecture of our Chord DHT version and the specific differences from the original one. Section IV presents the implementation decisions and the frameworks that were used. Section V presents a small overview of the possibilities of further extending this framework for different scenarios.

## II. Related work

In the last decade, a lot of research effort focused on the area of peer to peer systems. Several architectures have been proposed that can be classified depending on how data is indexed, level of decentralization and by the structure they show [2]. Chord uses distributed indexes for information localization and has a centralized, structured architecture. Chord was introduced in [1] and allows efficient looking up in a distributed hash table (DHT). By using logarithmic size routing tables in each node, Chord allows to find in logarithmic n number of routing hops the node of a P2P system that is responsible for a given key. Adding or removing a node is accomplished at a cost of O(log 2 N ) messages.

The Chord architecture has proven to be efficient for a wide variety of applications, such as content delivery networks, mobile ad hoc networks, file sharing or wireless sensor networks. This has lead to several extensions and improvements of Chord.

In some circumstances, nodes stay in the Chord ring only for a short time, leading to heavy traffic load. Such situations are mobile scenarios, where storage capacity, transmission data

rate and battery power are limited resources, so the heavy traffic load generated by the shifting of object references can lead to severe problems.

Therefore, the Hybrid Chord Protocol (HCP)[3] was designed to solve the problem of frequent joins and leaves of nodes. HCP uses two types of nodes static nodes and temporary nodes, addresses the grouping of shared objects in context spaces. It introduces info profiles to label all provided content in the network. Every shared object is described by an info profile which contains several keywords. By means of these keywords the shared object is assigned to multiple context spaces, each containing all available information for a given keyword. Similary, Mobile Chord[4] proposes optimizations such as aggressive table update overlay table broadcasting, greedy forwarding and passive bootstrapping.

F-Chord [5] proposes an improved uniform routing algorithm on Chord, based on the Fibonacci number system. It improves on the maximum/average number of hops for lookups and the routing table size per node. Self-Chord[6] proposes self-structured P2P systems, in which the association of keys with hosts is not predetermined, but adapted to the modification of the environment. Keys are given a semantic meaning, which enables the execution of class queries. The load of the maintenance is reduced because, as new peers join the ring, mobile agents will spontaneously reorganize the keys in logarithmic time. EpiChord[7] is able to adapt to a wide range of lookup workloads, achieving O(1)-hop lookup performance under lookup-intensive workloads and at least O(log n) hop lookup performance under churn-intensive workloads. The reactive routing state maintenance strategy allows one to maintain large amount of routing state with only a modest amount of bandwidth, while parallel queries help to reduce lookup latency and to avoid costly lookup timeouts. The nodes fill their caches from observing network traffic and the cache entries are flushed after a fixed period of time.

## III. Implementation

This paper aims as a deployment scenario, a Chord based DHT that is suitable to be used as a cache layer for web oriented applications like a CDN where the are more reads than writes and also provide reliable data storage.

### A. Node joining

Node joining happens similar to how they are described in the Chord Protocol. We analyzed the need of modifying the node joining procedure in order to allow for faster bootstrapping of new nodes but didn't find it necessary for our use case, as the server topology would not change that fast.

Each node keeps a record of it's successor and predecessor in order to facilitate new nodes joining. When a node joins it uses another node to lookup it's key in the network and then find it's successor and predecessor and inform them of his joining. He then receives his documents from it's successor.

### B. Fast lookup

The lookup algorithm uses a finger table as described in the paper. The finger table needs to be updated for each node that joins or fails. In order to construct it's finger table each node needs to send out at most $O(log^2 N)$ messages. Each node constructs it's finger table of size M. For our implementation we chose M to be 160 in order to cater our hashes sizes of 128 and 160 bits.

When a lookup is being processed the type of the hash that is looked up must be specified, this is somewhat similar to keeping two Chord Networks, one with the SHA1 hash and one with the MD5 hash.

### C. Throughput

Improved read throughput is achieved by querying both nodes at the same time. This eliminates the bottleneck that can be introduced by having a slow node and improves long tail latency.

Write throughput on the other hand suffers by the fact that both nodes must be updated before a write can be confirmed.

### D. Caching the data with Memcached

Users consume an order of magnitude more content than they create. This behavior results in a workload dominated by fetching data and suggests that caching can have significant advantages. Second, read operations fetch data from a variety of sources such as databases, distributed filesystems installations and backend services. This heterogeneity requires a flexible caching strategy able to store data from disparate sources.

Memcached[8] is a general-purpose memory caching system that provides a low latency access to a shared storage pool. The content is identified by key-value pairs. Dynamic database-driven websites cand be made faster by caching data and objects in RAM to reduce the number of times an external data source must be read.

Memcached's APIs provide a very large hash table distributed across multiple machines. When the table is full, subsequent inserts cause older data to be purged in least recently used (LRU) order.

The size of this hash table is often very large. It is limited by available memory across a large number of servers in a data centre. Memcached can be equally valuable for situations where either the number of requests for content is high, or the cost of generating a particular piece of content is high.

### E. Data replication and consistent hashing

The Chord Protocol leaves multiple implementation details unspecified mostly related to the ability to recover from node failures. In our implementation, all objects stored on the Chord ring have a replication factor of 2. We use two different hash functions to map keys onto the ring(MD5 and SHA1). Firstly the algorithm places the value on the server indicated by the SHA1 hash then on the one determined using the MD5 hash. If the servers indicated by the two hashes are the same (collision case), the value will be placed on the respective server and on the next one in the order indicated by the MD5 hash function. This situation is dealt with both put() and get() primitives. The major disadvantage is that it requires twice the storage space as a normal DHT would require, but it's almost impossible

to offer high-availability without having data duplication. If a memcached node runs out of memory, then the less recently used items are replaced by new ones.

Algorithms for decentralized data distribution work best in a system that is fully built before it first used; adding or removing components results in either extensive reorganization of data or load imbalance in the system. Therefore, we use a consistent hashing[9] technique such that when a hash table is resized, only K/N keys need to be rearranged on average, where K is the number of hash keys, and N is the number of slots.

### *F. Heartbeats for timeout-free failure detection*

For failure detection, we introduce at every node heartbeats[10], failure detection mechanisms that can be implemented without timeouts to solve the problem of quiescent reliable communication in systems with process crashes and lossy links. Every node from the Chord ring monitors the health state from the predecessor and successor nodes by sending regular ping messages. If one node fails to respond after a fixed number of ping messages, the node is considered down. As mentioned in the previous section, we use two different hash functions to map keys on the Chord ring (MD5 and SHA1) and no collisions are allowed(an object must have two replicas stored on different nodes). If a node gets down, the adjacent nodes contact the other nodes from the Chord ring that owe the replica of the objects stored on the failed one.

## IV. Simulation environment

For reproducing the networking experiments, we choose to use the Mininet[11] virtual network emulator. Mininet allows creating realistic topologies consisting in a collection of end-hosts, switches and links, running a real kernel and application code on a single machine.

Mininet uses lightweight virtualization features, including process groups for network namespaces, Linux traffic control for limiting the data rate of each link(tc) , virtual Ethernet interfaces(created and installed with ip link add/set ) and Linux bridge or Open Switch to switch packets across interfaces. We can consider different networking conditions in terms of link delay,jitter, packet loss and even handover scenarios. The following experimental evaluations are conducted: read/write throughput for objects of different sizes, object distribution across nodes (number of objects stored as a function of node rank), object availability as a function of the number of crashed. servers

## V. Communication

For communication we decided to use XML-RPC in order to make calls between the nodes. The XML-RPC framework comes by default with the Python programming language.

| Hash | Avg | Peak |
|---|---|---|
| MD5 | 121ms | 203ms |
| SHA1 | 123ms | 173ms |
| MD5 & SHA1 | 118ms | 140ms |

TABLE I
Response time

| Hash | Avg |
|---|---|
| MD5 | 8 |
| SHA1 | 8 |
| MD5 & SHA1 | 15 |

TABLE II
Number of messages

## VI. Performance evaluation

For testing the performance of the lookup we used our double lookup and with a single hash activated. For the testing of the query time we set up a network of size $2^8$ with 4 nodes simulated using Mininet.

As it can bee seen form Table I the MD5 and SHA1 functions have similar query time, while when used together they reduce the peak response time by reducing the long tail latency. Due to the way chord queries work there is a two times increase in the amount of messages sent between the nodes. For the MD5 & SHA1 case there are basically two queries that need to be sent over the chord network as it can bee seen in Table II.

The memory of the nodes increased by a factor of 2 since the information is kept under two hashes.

## VII. Conclusion and future work

While we can see that the performance of the chord network has increased, it also comes at the cost of needing to keep two copies of each key. The random distribution of data in the network helps to lower the query time, but increases the replication time of the lost keys when a node goes down, as the nodes that detect the failure are not always the ones that also have the keys.

As future research we want to look into replicating the keys on adjacent nodes in order to rapidly recover a lost node and possibly storing them on disk in order to reduce memory pressure on the nodes.

Another improvement could be the transport system, by keeping alive TCP connections between the nodes and their finger nodes we can reduce overhead. Also we can replace the XML-RPC system with one with less overhead, like one based on Protobuffers.

## References

[1] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applica-

tions. *ACM SIGCOMM Computer Communication Review*, 31(4):149–160, 2001.

[2] G Camarillo. Rfc 5694, november. *Peer-to-Peer (P2P) Architecture: Definition, Taxonomies, Examples, and Applicability*, 2009.

[3] Stefan Zöls, Rüdiger Schollmeier, Wolfgang Kellerer, and Anthony Tarlano. The hybrid chord protocol: a peer-to-peer lookup service for context-aware mobile applications. In *Networking-ICN 2005*, pages 781–792. Springer, 2005.

[4] Che-Liang Liu, Chih-Yu Wang, and Hung-Yu Wei. Mobile chord: enhancing p2p application performance over vehicular ad hoc network. In *GLOBECOM Workshops, 2008 IEEE*, pages 1–8. IEEE, 2008.

[5] Gennaro Cordasco, Luisa Gargano, Mikael Hammar, Alberto Negro, and Vittorio Scarano. F-chord: Improved uniform routing on chord. In *Structural Information and Communication Complexity*, pages 89–98. Springer, 2004.

[6] Agostino Forestiero, Emilio Leonardi, Carlo Mastroianni, and Michela Meo. Self-chord: a bio-inspired p2p framework for self-organizing distributed systems. *Networking, IEEE/ACM Transactions on*, 18(5):1651–1664, 2010.

[7] Ben Leong, Barbara Liskov, and Erik D Demaine. Epichord: parallelizing the chord lookup algorithm with reactive routing state management. *Computer Communications*, 29(9):1243–1259, 2006.

[8] Brad Fitzpatrick. Distributed caching with memcached. *Linux journal*, 2004(124):5, 2004.

[9] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 654–663. ACM, 1997.

[10] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. Heartbeat: A timeout-free failure detector for quiescent reliable communication. In *Distributed Algorithms*, pages 126–140. Springer, 1997.

[11] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, page 19. ACM, 2010.