Chornamo!

Connie Chen
cwc009@cs.ucsd.edu

Russell Reas
rreas@cs.ucsd.edu

Rakesh Varna
rvarna@cs.ucsd.edu

Department of Computer Science and Engineering
University of California, San Diego

ABSTRACT

In this report, we implement and evaluate the performance of a Distributed Hash Table (DHT) based on Chord [?]. We experiment with synchronous and asynchronous replication strategies based on sharing data with one or more successors. All code is available on GitHub [?] and written in Python using the Thrift RPC [?] framework.

General Terms

Reliability, Experimentation, Distribution

Keywords

Distributed Hash Tables, Distributed Key-Value Stores, Consistent Hashing, Chord, Replication, Thrift RPC, Asynchronous Replication

1. INTRODUCTION

Distributed Hash Tables (DHTs) serve as decentralized key-value stores that can scale well and are fault-tolerant. Each participating node is responsible for storing a set of keys and their corresponding values. DHTs are known for their fast, $O(\log N)$ key lookups [?].

The remainder of the report is structured as follows: Section 2 briefly describes Chord, a peer to peer lookup protocol that our implementation is based on. Section 3 describes the different lookup and replication strategies that we have implemented. We discuss the design of our experiments in Section 4. In Section 5, we analyze and discuss the results of our experiments. We conclude in section 6.

2. CHORD

Chord is a peer-to-peer lookup protocol that maps a given key onto a node. It uses consistent hashing [?] to assign keys to the nodes [?]. Consistent hashing tends to balance load with each node receiving approximately the same fraction of keys. In order to scale to large number of nodes, a given node should store minimal amount of information about the other nodes in the system. In an N-node system using Chord, any given node maintains information about only $O(\log n)$ other nodes and the lookup involves only $O(\log N)$ messages. Note that this is only for faster lookups. For correctness, it is sufficient to maintain a pointer to the next node along the circle. As nodes join and leave the system, the 'routing' information stored at any given might become stale. This results in degraded performance, yet does not affect correctness. Chord only requires that a given node maintain the correct information about its successor, the next node on the circle (clock-wise), considering the nodes being arranged in a circle, as per consistent hashing. Replication and storing values associated with the keys is handled in the higher level application.

2.1 Efficient lookups

The nodes in Chord maintain a 'finger-table', with the i^{th} entry in the table pointing to the successor of the key $(n + 2^{i-1}) \mod 2^m$, $1 \leq i \leq m$, where m is the number of bits in the output of the hash function used. Given that node is likely to be within a distance of $2^m/N^2$ of any other node[1], it is sufficient to maintain just $O(\log N)$ entries. Otherwise, a node can maintain m entries. Given a key, a node searches along its finger table to find the farthest node that lies before the key. With each such lookup, the distance is guaranteed to reduce by half, assuming the entries are stable and correct. When the finger tables are incorrect, in the worst case, the performance degrades to O(n) such lookups.

2.2 Ensuring correctness

Node joins can render successor pointers and finger table entries temporarily incorrect. Periodically, Chord runs a *stabilize* method, that allows a node to learn about newly joined nodes and if the node has a new successor. Similarly, a *fix_finger* method corrects the entries in the finger table by checking the new successor (if any) for the corresponding key. Node failures are much harder to handle and the successor pointer cannot be corrected without any information about the nodes beyond the failed successor. To increase the robustness

[1]The reader is referred to [?] for the theorem and correponding proof.

against failures, Chord maintains a *successor list* whose length can be tuned to meet fault tolerant requirements. If the length of the list is r, assuming that the nodes fail independently with probability p, the probability that all nodes in the successor fail simultaneously is p^r which can be made as small as required by increasing the value of r.

3. DESIGN

Our implementation closely follows the Chord protocol with a few differences including the addition of replication to successors.

3.1 Replication strategy

In our implementation, we follow the replication strategy detailed in [?] by replicating the key and the corresponding values at a given node in each of the successors in the successor list. When a successor fails and we obtain a new successor (at the end of the list), the keys are replicated to the new n^{th} successor.

We implement and experiment with both synchronous and asynchronous replication. In synchronous replication, when the key is put into a node, it is replicated to the successors before returning back to the client. In asynchronous replication, we store the pending replications in a queue and copy them to the successors in a separate thread.

These implementations represent a trade off between latency and strong consistency. In the former, adding a key to the DHT might take longer, but once a key and its value have been inserted, they can always be retrieved in the presence of failures independent of when the failures happen. With asynchronous replication, if the node containing the key-value pair fails before the replication thread copies the pair to any successors the pair is effectively lost.

3.2 Forced stabilization

In Chord, node failures might result in an incorrect successor pointer and therefore, the key lookup might fail temporarily. The client is required to retry after a pause, within which, the *stabilize* method is expected to run and fix the pointer. In our implementation, whenever a node realizes that its immediate successor has failed, it fixes the successor pointer using the successor list. The method for this forced stabilization is described in Algorithm 1.

It is possible that node maps a key to another node using the finger table and the latter has failed. In this case, the node proceeds with the next best guess from the finger table. At some point, we will reach the predecessor of the failed node which forces the stabilization and continues with the lookup. In the next round of fix-finger-table, all the pointers will be corrected. As a result of these choices, a client will get the value for a given key if it exists within any one of the nodes without requiring a retry. The latency might be slightly higher for that particular query.

Algorithm 1 Handling successor failure

```
for node in successor_list: do
    if node is alive then
        Store the old successor list.
        Make node the new successor.
        Obtain the new successor list from node.
        Insert successor the first entry in successor list
        Delete the last entry of successor list.
        for each successor not in old list: do
            copy the keys and values to the successor
        end for
        return
    end if
end for# All successors in the list have failed. We
do not handle this case.
exit()
```

4. IMPLEMENTATION

We have implemented multiple variants of Chord to experiment with and analyse their performance. The key value store is implemented in Python. We use Apache Thrift [?] for making RPCs between the nodes. We use MD5 hashing to convert the given key into a 128 bit identifier on the circle. The identifier for a node in Chornamo is the MD5 hash of hostname and port number of the node in the form "hostname:port_number". Each of the variants is described below.

Basic Chord.

This version just maintains the successor pointer for lookups and successor list for robustness. This results in an $O(n)$ lookup complexity for mapping a key to a node. Other variants of Chord are built over this basic version. The stabilize method is handled by a separate thread which runs every 2 seconds.

Chord with finger tables.

We extend the basic version of Chord by adding finger tables for faster lookups. In our implementation we have 128 entries in the finger table (equal to number of bits in our hashing function), with i^{th} entry representing the successor of $(n + 2^{i-1}) \mod 2^{128}$, $1 \leq i \leq 128$. The *fix_finger_table* and *stabilize* methods are run by the same thread (every 2 seconds).

Synchronous replication.

In this implementation, we replicate the keys synchronously, as and when they are added into the successors in the successor list. We maintain a maximum of 5 successors.

Asynchronous replication.

In this implementation, a separate thread handles the responsibility of copying the added keys onto the successors.

5. EXPERIMENTS

5.1 Key distribution

In this experiment, we check how well the keys are distributed among the nodes. Ideally, consistent hashing would divide the keys equally, but in practice the load varies across the nodes. We run the experiment with XX nodes (all hosted on the same server) and add YY keys and values. The results are presented in Figure **??**. The distribution of keys is uneven, with the maximum of XX keys being stored at a single node and minimum of YY keys. This is probably due to the uneven spacing of node identifiers along the circle.

5.2 Latency

In these experiments, we analyse the effect of replication and concurrent queries and to the DHT and usage of finger tables on the latency. We discuss each of them below.

5.2.1 Basic Chord vs. Chord with finger tables

Basic Chord has an $O(N)$ lookup complexiy while the implementation with finger tables is theoretically logarithmic. We aim to find how well each of them performs in practice. We run a succession of trials (on localhost), with the number of servers increasing in each successive trial, to see how that impacts latency. In each case, we add XX number of keys and values. The results are displayed in Figure **??**. It is clear that finger tables definitely help improve the lookup latency. In fact, the timing match the theoretical expectations.

Over the network.

We run the same set of experiments, but now over the network. Each node is now hosted on a separate server (when the number of nodes exceeds the number of servers at our disposal, we run with a maximum of two nodes at each server). Figure **??** presents the results. As expected, the network does not have any other effect on the results presented above, other than adding latency to each of them. For the rest of the experiments, we use implementations that incorporate finger tables.

5.3 Concurrent lookups

In this experiment, we analyze the effect of concurrent lookups on latency. We run the experiment with XX servers and YY clients, with each of them adding and retrieving ZZ keys from the nodes. The results are presented in Figure **??**.

5.4 Synchronous replication

5.5 Asynchronous replication

5.6 Node failures

6. LIMITATIONS

Python threads... bl

7. CONCLUSION AND FUTURE WORK

We have implemented a scalable and fault tolerant distributed hash table based on Chord [**?**]. We experimented with different replication and lookup strategies and compared their performance.

Beehive, Dynamo??