A simple distributed hash table with Chord overlay

Bastien Duboc
Reykjavik University
bastien20@ru.is

Jón Steinn Elíasson
Reykjavik University
jone20@ru.is

Ragnar Stefánsson
Reykjavik University
ragnars15@ru.is

January 31, 2021

Abstract

Key features of a distributed hash table include balancing, scaling and efficiently locating nodes. This paper is about our implementation of a simplified Chord overlay. We limit our system to adding new nodes and writing and explore how well it performs with respect to these features.

I. Introduction

The initial research for Distributed hash table (DHT) was motivated by rising popularity in systems using peer-to-peer (P2P). These systems provided a single interface for file sharing that utilized resources distributed over the internet made possible with increasing network bandwidth and disk space. All of these P2P systems used their own ways to locate the data provided by their peers but the benefits of using DHT was that it had a more reliable structure of key based routing. They were decentralized and gave efficient and correct results while still being reliable given the unreliability of nodes (servers).[1] There are multiple different DHT infrastructures in existence. One of these is Chord [2] that we base our project on.

The rest of this paper is structured as follows. Chapter II introduces key terminology and concepts. In chapter III we describe our implementation. In chapter IV we specify the experimental setup and finally interpret the results in Chapter V.

II. Background

DHT is decentralized lookup service with a similar interface to normal hash tables. It stores key-value pairs across multiple different nodes. Nodes can be added or removed from DHT with minimal redistribution of keys. As with a normal hash table, keys are mapped to any value and serve as a unique identifier for that value. Each node can either retrieve the value or forward the query to the appropriate node.[1]

Chord is an overlay network for DHT to efficiently handle routing between nodes. Each node's IP address is hashed (using the consistent hashing algorithm SHA-1) to some id and placed in an identifier circle modulo 2^m and maintains links to the previous node and the node responsible for the id with offsets $2^0, 2^1, \ldots, 2^{m-1}$ from itself where m must be chosen such that probability of collision is minuscule. Note that the offset is increasing exponentially so each node knows more about nodes closer than those further away. The latter links are called finger tables and it is what Chord uses to reach the desired destination efficiently. When asking for a value given a key, that key is also hashed and the next node id succeeding that value (wrapping around the circle) is the one responsible for it.[2]

Suppose there are two nodes with ids 551 and 813 and no other nodes have ids in $[551, 813]$. Keys can share ids with nodes and a key mapped to 551 belongs to the node with

the same id while any key mapped to $(551, 813]$ belongs to the node with id 813.

When nodes join and leave, Chord makes sure that all nodes in a Chord circle are connected with a successor and predecessors. When multiple nodes are joining concurrently it becomes very difficult to make sure that all finger tables are still correct. It uses a background task called Stabilization to periodically update finger tables to ensure correct look ups and notify nodes about joining and leaving nodes.[2]

III. Methods

Our simple DHT has a predefined number of extents (e.g. files) to which it can write and a predefined amount of initial nodes. Each key of the initial nodes is hashed with SHA-1 and mapped to an identifier circle and that will be the node's id. In our simple example we start with a fixed number of nodes so we can construct the Chord ring easily by sorting the nodes by their ids. With sorted nodes we can use a greater or equal variation of binary search (that wraps) to construct the finger tables and predecessor links for the nodes.

After creating the nodes we add the predefined extents in a similar manner. Each key is hashed and mapped to the identifier circle and added to its successor node using binary search. For each extent, N number of copies is created and distributed to the next N nodes on the Chord ring. This approach has the downside of putting the entire load of a failing node on its neighbor although nodes in our simplified version will neither fail or leave.

Only two operations are supported when the DHT has been initialized, writing to an extent and adding a new node. Given a requesting node and an extent to write to, we start from said node and look for the extent's successor using the finger tables to jump as far ahead as we can, not succeeding the id on the Chord ring. This will guarantee logarithmic jumps in order to find a successor. When found we update the extent and all of its copies.

Adding new nodes was by far the most challenging part of our implementation. We do keep track of sorted node ids for statistical and unit testing purposes but using them to find nodes would defeat the purpose of a distributed system. As before, we start by hashing the node's key but now we use the finger tables to find its successor. The new node is stored and we update the predecessor and successor links for the new node and the two adjacent nodes and construct a finger table for the new node (using the finger tables already in place). Instead of stabilizing the finger tables of the remaining nodes in a background task we do so right away and finally we move the extents that should now belong to the new node as well as moving the copies so the N successors system persists as shown in Figure 1.

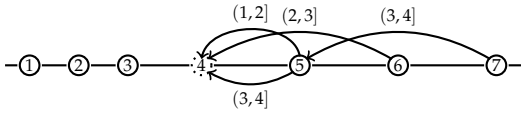

Figure 1: *Movement of extents and copies ($N = 2$) when a node is added (4) with copies above and original extents below. Intervals are node label range of extent successors.*

IV. Results

We used randomly generated non-private IP addresses and movie titles with `.avi` endings as keys for nodes and extents respectively. This was done to make the experiment more applicable to real scenarios. Java's standard library was used for SHA-1 hashing. A consistent hashing algorithm, to balance the load of nodes. Our identifier circle was $[0, 2^{40})$ which was large enough to avoid collisions.

We initialized the DHT with 10 nodes and 10^4 extents. In each iteration we picked a random node and made it request 10^6 writes to random extents. We then added 5 more nodes to the DHT and performed the writes again until a node count of 30 was reached. The parameters used are summarized in Table 1. Every write was tracked, both for nodes and extents. After each iteration we reset the write counters so we could compare if the distribu-

S	Number of nodes initially	10
E	Number of extents	10^4
N	Copies of each extent	3
W	Writes each iteration	10^6
I	Nodes added in each iteration	5
S_m	Maximum number of nodes	30

Table 1: *The parameters used for our experiment.*

tion of writes for nodes was analogous to the distribution of extents assigned to nodes. That way we avoid biasing older nodes. The distributions for 10 and 30 nodes can be seen in Figure 2 and Figure 3 respectively. The standard deviation of these distributions for all number of nodes can be seen in Table 2.

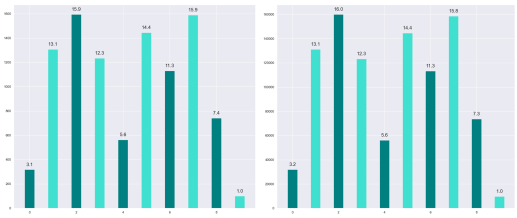

Figure 2: *The distribution of extents over 10 nodes on the left and distribution of writes over the same nodes on the right.*

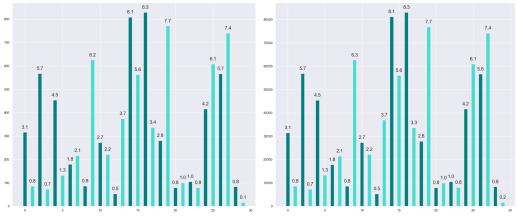

Figure 3: *The distribution of extents over 30 nodes (20 added to the original 10) on the left and distribution of writes over the same nodes on the right.*

S	10	15	20	25	30
μ_{extents}	537.61	382.75	353.02	314.04	258.53
σ_{writes}	53848.17	38224.04	35254.81	31456.25	25866.75

Table 2: *Standard deviation of extents and writes over nodes.*

We also tracked jumps when finding successor nodes but only did so with writes. Their distribution can be seen in Figure 4 for 10 and 30 nodes.

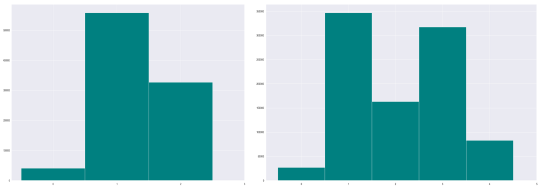

Figure 4: *The distribution of number of jumps for 10 (left) and 30 (right) nodes.*

V. Conclusions

The nodes were not as evenly distributed as we had hoped but in such a large space with such few nodes, its hard to guarantee even load. As we can see in Table 2 the standard deviation for the distributions is decreasing as new nodes are added and therefore the load is balancing.

As was expected, the jumps required were not many. With $\log_2(30) \approx 5$ and few nodes, each containing 40 entries in the finger table we are bound to find one soon.

During this experiment we gained a better understanding of the underlying infrastructure and the challenges they aim to solve.

References

[1] Eng Keong Lua et al. "A Survey and Comparison of Peer-to-Peer Overlay Network Schemes". In: *IEEE COMMUNICATIONS SURVEYS AND TUTORIALS 7* (2005), pp. 72–93.

[2] Ion Stoica et al. "Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications". In: *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. SIGCOMM '01. San Diego, California, USA: Association for Computing Machinery, 2001, pp. 149–160. ISBN: 1581134118. DOI: 10.1145/383059.383071. URL: https://doi.org/10.1145/383059.383071.