INF3200: Mandatory Assignment 1

Simon Niedermayr Luca Manzi

August 8, 2022

Abstract

We implemented and evaluated a decentralized key value store as proposed by the Stoica *et al.* [1] in their paper named "Chord". Our implementation offers retrieving and storing functionality that is accessible via a HTTP API. We measures the retrieve and store throughput for varying number of nodes and found that latency increases close to proportional to the number of nodes. We argue that this scalability issue can be solved with the usage of finger tables [1] and and a better intermediate node communication based on the UDP protocol instead of our current TCP implementation.

1 Introduction

Chord, is a Distributed Hash Table and a mechanism for associating keys with values in a decentralized way. As for the others DHTs, the main advantage of Chord is its scalability, and security, giving the absence of a single access point. At the base level, it implements the only important operation needed to create a P2P network, given a key, it will determine the place where to store an associated value.

We can think about Chord as a whole system, where having any node that takes part of the network, we can get the same answer: even if a node doesn't have any information about the given key, it knows enough about the network to let the request be handled by others node in Chord Ring. We can analyze the information needed, and how can we achieve a reduction of the complexity of the lookup phase on N nodes from $O(N)$ to $O(log(N))$. This mechanism implemented in Chord gained importance, solving some problems regarding safety and scalability of other older peer-to-peer systems.

2 Background

An Hash Table, is a data structure that maps a key to a value. Given an array of slots, the hash table applies an Hash function to generate the index where to store the passed value. The range of the hash function is the whole array and when trying to find a certain value, in the lookup phase, the key gets hashed to obtain again the location of the value. Naturally there is no certainty that each value is mapped to a different point in the array, since, as mentioned before, the range is the array, but we have no information about the domain, which can also be infinite. So, theoretically can happen that more objects have the same hash: this occurrence is called Hash Collision.

A problem that can emerge is that if the number of the objects gets really large, there is no possibility for any single node to store all the information.

The solution can be splitting the large hash table in smaller ones in different machines.

A Distributed Hash Table than, has to provide a similar functionality of a standard HT: store and retrieve values in the slots, or 'buckets", that in this case are the nodes of the DHT. There is no direct indexing to the memory, but the hashing function range is a virtual space, and each node is responsible of handling data for a subset of it. To avoid hashing collision, we follow the suggestion of the paper [1], generating a big hash key on 64 bits using a deterministic hash function.

3 Related Work

We analyzed some other implementation of DHTs, more than the one discussed in class, to understand some differences that can make this better. The most notable was in Kademlia [2], that before sending a PUT request, checks and returns a subset of nodes that seemed the best option to store the data. Doing this, it reduces the internal communication, but the implementation has to provide a way of sharing the presence of data in the node, to make possible the initial lookup for the suitable nodes.

4 Project Description

Goal of the project is to implement a basic chord network that offers the basic functionality of a key value store — storing and retrieving values associated with unique keys. The network is static, meaning no nodes are leaving or joining it after its initial creation.

Each node in the network has the same functionality and looks like a enclosed system to the clients that are communicating with it. The clients interact with a node via a HTTP API, that offers the following functionality:

- `GET /storage/<key>`: Values can be retrieved from the chord network via a HTTP GET request by specifying the values key in the path. The response's HTTP body holds the value, or returns the HTTP status code `404`, if the key does not exist within the storage.

- `PUT /storage/<key>`: A HTTP PUT request stores a key value pair in the chord network. For this the key is provided in the path and the value is in the HTTP body.

- `GET /neighbors`: A list of a nodes neighbors can be retrieved this way. In our case this list only contains the nodes successor.

4.1 Architecture

The chord network is a decentralized system, meaning that each participant (node) is equally important. In our case this means that each node is a process of the same program, only distinguishable by their unique id. Those processes can run on one or preferably multiple machines.

The program has two main components:

- **HTTP API**: This is the API used for the clients to communicate with the network to store and retrieve data

- **chord API**: An API used for communication between the nodes to retrieve the location (owner node) of a key.

4.2 Design

Identifiers Chord arranges nodes and keys on a ring by hashing them and arranging the hashes in a modulo group. A nodes/keys hash within the modulo group is called its Identifier. For our implementation we use SHA256 as a hash function and a modulo group size of 2^{64} (size of unsigned integer).

A keys identifier is retrieved by hashing its string value. For the node we hash the nodes IP address and port for the chord API. This allows us to run multiple nodes on one system by either choosing different ports or network interfaces.

Communication Communication within the chord network is performed with TCP packages. There are two kind of messages:

- `Lookup(<key>)`: A request for finding the node responsible for a given `<key>`.

- `Result(<chord_addr>,<http_addr>)`: The response to a `Lookup` message containing the chord API address (IP and port) and HTTP API address of the node responsible for the key.

An example for the retrieval of a value from the chord network can be seen in Figure 1. The retrieval starts with a HTTP request to an arbitrary node within the network. If the node is responsible for the key it looks up the value in its own storage and responds to the HTTP request with the value (if it exists).

In the the case, that it is not responsible for the key, it asks its successor for the responsible node. The successor node then replies with his own address, if he is responsible, or again asks its own successor node for the responsible node. Once the initial node, that received the HTTP request, has gotten the address of the node, that is responsible for the id, it sends a HTTP request to this node to retrieve the value and then forward it to the client that performed the HTTP request. The process for storing a value is analog.

Network Creation The network is static and the number of nodes and their location on the chord ring do not change during the run time. Therefore all nodes and there IP addresses and ports need to be known for the network creation. For the creation each node's identifier is calculated and the nodes are arranged on the ring. Then the node processes are started and each node received its own address as well the the address of its predecessor and successor on the ring.

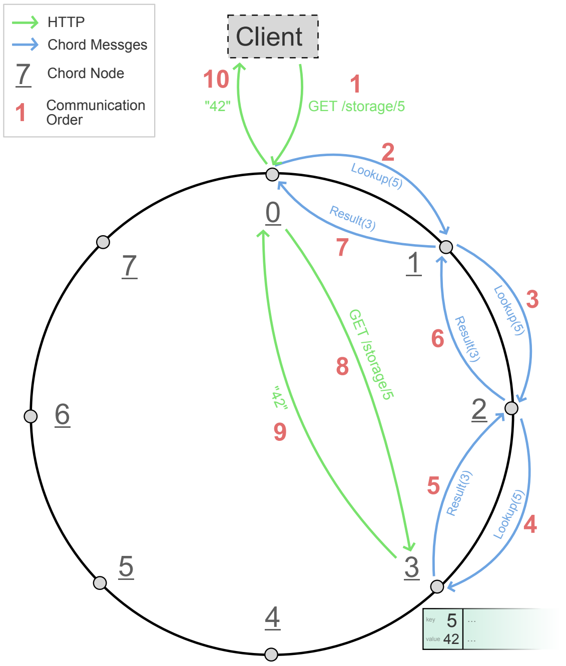

Figure 1: **Chord Communication.** This image shows an example for a value retrieval from the chord network. The red numbers indicate the order the messages are being sent in. The Client makes a HTTP GET request to retrieve the the value (here 42) for the key 5, that is stored in node 3.

4.3 Implementation

Our chord network is implemented in the programming language rust and uses tokio for TCP (HTTP) communication and concurrency handling. [3] The HTTP API is implemented with warp. [4] We use a in-memory Hash-map as the key-value storage on each node.

The network is created with a Python script that starts the nodes on their respective machines by running the compiled rust program and passing the required information (address, predecessor and successor) as a command-line argument to the program.

5 Evaluation

The cluster we are working with is composed by 118 computers with Intel(R) Xeon(R) CPU, in range @2.40-@3.60 GHZ. Therefore to avoid special cases, to evaluate the system with each test performed, random nodes were selected from those available, and we used networks of 1, 2, 4, 8, 16, 32 nodes on different machines.

After passing all the tests on the features, and not having encountered any case of error, we built a Python script, based on the client that we had for testing, removing checks on the correctness of the results. This Script ask to a node to store and retrieve a value (PUT,GET), and change node at each iteration, for a number of try, 300 in our case seemed a fair amount of tries, because we expected linear results between 1 and 10 seconds. Same thing was done for just the PUT request. The experiment was tried for 10 runs, measuring the time elapsed after completing all the requests, then converted the time for the operation in $(PUT + GET)/sec$ and $(GET)/s$, as specified by the assignment.

6 Results

Here the data we got from the experiments, in two formats. We omitted the 32 node case, because it makes the plot less clear.

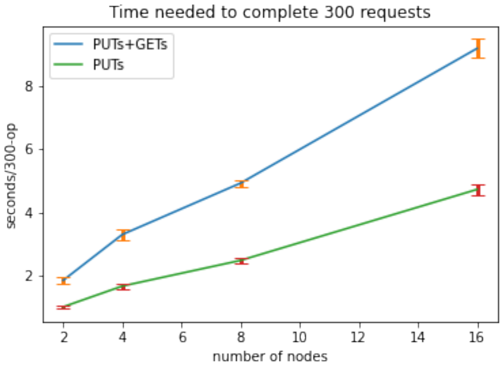

Figure 2: **Test raw data.** This image shows the data used to analyze the throughput of the network, in red the standard deviation error bars.

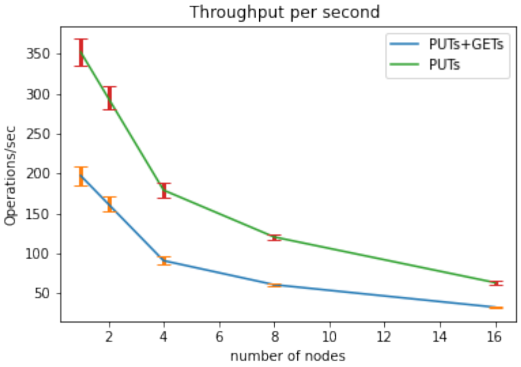

Figure 3: **Throughput per second.** This image shows the $(PUT)/s$ in Green and the $(GET)/s$ in Blue. Once again in red we have the error bars based on the standard deviation.

7 Discussion

As expected, the implementation of the Chord request without finger table, shows a linear time, with respect to the growing number of nodes. This is due to the fact that the nodes are forwarding requests that they aren't able to handle in a linear way along the Chord circular space. Finger tables can be useful, saving more that one successor, and are explained later in the next paragraph.

On the other hand, as $seconds/operations$ grows linearly, its inverse decreases as $1/x$: we ca see it clearly in the second plot, that represents the throughput per second. More forwarding operations are needed to store/retrieve a single value, therefore we need to wait more time for the response.

In the second graph we can state that with smaller number of nodes, variance seems to be bigger, but it can be related to the higher speed and so higher possibility of any random event to have an impact on the measurement.

8 Future Work

- An improvement we would like to try, and then test our intuition, would be using UDP for communication: messages can run faster in the network and it can be useful in a environment where a lot of messages are sent and connection are opened.

- Redundancy to prevent node failures would be another interesting point to discuss: copying the information of each node into the neighbor(s), to prevent the loss of information held, if this node leaves the network without communicating it for any reason.

- One of the best improvement that can be done to traverse the Chord network in a optimized way is storing a finger table instead of a just one successor node. This can be achieved storing a table containing $log(N)$ other entries. Each i^{th} entry is the address of the node immediately after (in other words responsible of handling an item at) a distance of 2^{i-1} In this way the redirect phase after a lookup, when a node receive a request and it is not responsible to handle it, should be executed comparing the address we are looking for, backwards from the last entry of the finger table. In this way we are "skipping" half of the distance of the circle at the first step, half of that on the second step, and so on. This key feature reduces drastically the complexity of the internal connections, allowing the network to scale well on big number of nodes.

- Another idea than can be explored is using redirects: as we can see in fig 1 when doing a request, the first node that forwards the request for a value opens a connection and stays in the retrieving process till the end, transferring the data to the client. Maybe this operation can be avoided, passing a redirect, to let the client communicate directly to the node that owns the data. This procedure can set the first entry node free, especially if handling large files. Naturally this redirection is not so faithful to the aim of this course: that we need to create a high level distributed system that acts as if it was one single entity.

- Fix the already implemented Creation / Stabilization process, in the next parameter.

9 Conclusion

Building this project was at the same time interesting and fun, especially using Rust as a language, using modern libraries and async functions.

Perhaps initially we got carried away, directly implementing the stabilization and join functions for the new nodes: everything seemed to work in the beginning, the randomly generated network was stabilizing every time, assuming a perfect circular shape, with every node taking a successor and a predecessor, but leaving sometimes some doubts in the responsibility-checks and not responding every time with 100% of success rate. Just for this, and just for now, we had to figure out another temporary solution, assigning nodes successor and predecessor from our Python initializer script and working with a stable, fixed network from the creation. We will take our time debugging and correcting those blind spots. Basic functionalities are working as expected, but there is a lot of space of improvements.

References

[1] Ion Stoica et al. "Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications". In: *IEEE/ACM Trans. Netw.* 11.1 (Feb. 2003), pp. 17–32. ISSN: 1063-6692. DOI: 10.1109/TNET.2002.808407. URL: https://doi.org/10.1109/TNET.2002.808407.

[2] Petar Maymounkov and David Mazieres. "Kademlia: A Peer-to-peer Information System Based on the XOR Metric". In: 1 (Feb. 2002), pp. 1–5. DOI: 45748-8_5,2002. URL: http://kademlia.scs.cs.nyu.edu.

[3] *Rust official documantation: Tokio, a runtime for writing reliable network applications without compromising speed.* https://docs.rs/tokio/1.12.0/tokio/. Sept. 2021.

[4] *Rust official documentation: Warp, a super-easy, composable, web server framework for warp speeds.* https://docs.rs/warp/0.3.1/warp/. Sept. 2021.