Report Homework 5: Groupy

Riccardo Fragale

October 8, 2025

1 Introduction

This assignment's goal was the implementation of a Distributed Hash table based on *Chord*, a lookup protocol designed to efficiently identify the node storing a particular data item in a distributed storage system. *Chord* addresses many problems those systems face such as load balancing, decentralization, scalability, availability, and flexible naming. It is a very well known standard and we were asked to correctly implement a slight variation of it, focusing in particular on error handling and reliability of the system for the bonus points.

2 Implementation

2.1 Ring structure

The whole network of nodes consists of a ring structure. To correctly identify nodes and keys we were using simple numerical IDs instead of hashing functions in order to ease our work. With respect to the original *Chord paper* we implemented a simple routing table that does not allow for $O(logN)$ lookup times. Each node only keeps track of its successor and predecessor in the ring instead of building the so called *finger table*. The ring structure is periodically stabilized through a `stabilize/3` function, which gradually fills the ring when new nodes join by having each node ask its successor for their predecessor and adjusting the ring accordingly. As the structure will have to wrap around at some point, the `between/3` function was implemented in a way to handle `From` being larger than `To` by checking if the `Id` is either larger than `From` or smaller than `To`.

2.2 Storage

Each node in the ring should be able to store key-value pairs. I implemented it using maps library. These pairs can be added and looked up by sending corresponding messages to the nodes. The messages are routed through the ring structure until they reach the node responsible for the given key; the client is notified through a message including the reference it provided on the message. When new nodes are joining the ring, a handover message containing the keys split based on the two nodes keys, is sent to the new node.

2.3 Bonus task 1: Failure handling

I need to specify tat we were asked for a failure handler that doesn't take into consideration that two neighbouring nodes may fail at the same time. The following solution is not working in that edge case, To detect failures, Erlang failure detectors were used to have each node monitoring both its predecessor and its successor. Nodes keep a record of their successors' successor, during each stabilization round. This is done to maintain the structure active in case their original successor fail. Should the predecessor fail, nodes set it to *nil* and wait for the predecessors predecessor to notify them.

2.4 Bonus task 2: Replication

Since data is only stored in memory, when a node fails we completely lose the data stored there. As a general solution for DHTs, nodes keep a replica of their predecessors data to avoid this very bad situation happen. In my solution the client is notified of a successful operation when both the main storage and the replica have been updated. Both the handover and failure handlers were updated to correctly split and merge the replica storage depending on the state of the ring. When a predecessor dies, the node merges the replica storage with its main storage and propagates its new state to its successor. When a new node joins, it receives the relevant keys for both its main storage and replica storage from its predecessor. We might reason on how many numbers of replica we need(maybe 3), but as for now, my solution is ok to guarantee a huge reduction of data loss events.

3 Problems and solutions

A first issue was to correctly apply the replication mechanism. While it seems simple to correctly create a copy of the nodes it is not easy to understand the procedure when a new node joins the network or a node fails. In those cases I had to consider how the keys were to be split and how the whole replication mechanism should have reacted. It took me a while to correctly build the whole procedure which is not that hard in the end, even though it slows a bit the system with respect to the implementation without copies. I also found hard to keep a complete view of the system. Since many features were to be implemented, it was hard to correctly debug the system and to understand where errors were. I needed to introduce a lot of debug logs and to read a lot of times the functions I was writing to avoid mismatched values and syntax errors.

4 Evaluation

I tried some tests and discovered that the routing table lookup times are quite fast with respect to what I expected (in the order of milliseconds), and performance degrades when huge quantity of data is added to the DHT. Each client (called machine in the table below) is actually adding 1000 keys to the ring. There is also an outlier case with 8 nodes in the ring and 2 machines pinging the system where the lookup time is quite high. As a consequence, probably the efficiency is reduced when there is a very big number of nodes in the ring as the procedures are generally longer.

Nodes ↓ / Machines →	**1**	**2**	**4**	**8**
1	2.84	2.95	6.49	–
2	1.49	3.76	6.81	7.72
4	1.72	4.99	7.01	–
8	2.35	10.26	–	–

Average lookup time (in milliseconds) measured for different numbers of nodes (clients) and machines.

In each case I used a single entrypoint (actually the first node) and it is a bottleneck. I also did a test with nodes connected on different machines (actually everyone on my machine but on different terminals) and the general performances were worse as there is a network delay due to communication between different Erlang nodes. On the other hand this is an important aspect as in real life DHTs we might prefer to go for a closer node with respect to a more logical routing structure as the network delay and node distancing has a big impact. As far as the first bonus part is concerned I developed a test called **run3()**. We start the ring, insert 1000 keys, then probe and check lookups. Then nodes N4 and N5 are killed, the test waits for the ring to recover, probe and run the final lookup check to confirm data availability. As it is already said in the description of this homework we still encounter some data loss when nodes are failing.

Regarding the second bonus part I implemented a test function in which a ring is created; keys are saved in two waves and then some nodes are killed and shut down. The logs are showing that the keys are correctly handed over when the system detects a failed node. All lookups are still working after the nodes are shut down so I can assume that my solution is working.

5 Conclusions

The advantages of this distributed storage come both in performance, through load balancing, and fault tolerance. Moreover, this system is certainly more scalable than a traditional database management system. Fault tolerance should be however prioritized, especially in productive applications, as data loss is not acceptable for industry solutions.

Future improvements definitely includes a new type of rouitng table with a solution as the *finger table* to improve the performance and to entirely implement the Chordy protocol logic.