INF3200: Mandatory Assignment 2

Simon Niedermayr Luca Manzi

August 8, 2022

Abstract

We implemented and evaluated a decentralized key value store as proposed by the Stoica et al. in their paper named "Chord". Our implementation offers retrieving and storing functionality, as well as a dynamic join and leave operations. Then we can simulate a crash and recover of a node, that leaves the network without notifying. We measures the time that the nodes take to stabilize in this dynamic environment, either creating a network, or leaving it, and found that latency increases proportionally to the number of nodes. The lookup around the circle will be linear; we argue that this scalability issue can be solved with the usage of finger tables.

1 Introduction

Chord, is a Distributed Hash Table (DHT) 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 fault tolerance, 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.

This mechanism implemented in Chord gained importance, solving some problems regarding safety and scalability of other older peer-to-peer systems. [1]

2 Background

An Hash Table, is a data structure that maps a key to a value. 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.

It's crucial how to compose the network, how a node can enter in the right place and leave the network leaving the structure fully functional. Also it can happen that a node can't quit in a clean way, so the network must have the capacity of spotting these situations and fix them.

3 Project Description

The project enhances the functions of our last CHORD implementation, adding dynamic node join and leave functionality, as well as a form of fault tolerance. The improvements are described in the following.

3.1 Join Procedure

- POST /join?nprime=HOST:PORT: Join a network. The node that receives this message must join nprime's network, as shown below.

A node can enter in the ring at any given moment, it just need to have another node location for obtaining all the other addresses it needs: it

needs to follow the ring to look for a node that is its successor, based on the hash of their sockets: then it will take the other node as a successor (that is the only compulsory field), and the other node's predecessor as predecessor: putting itself in the middle of them. Other chord async operations will fix neighbors' parameters.

3.2 Leave Procedure

- POST /leave: Leave the network. The node must go back to its starting single-node state, and the rest of the network must adjust accordingly.

In this occurrence the node that leaves the network communicates and prepares the network to be connected in its absence:

- node's successor should swap its new predecessor to the is node's predecessor
- gives to it's predecessor it's proper successor, as new successor.

This directly stabilizes the network (closes gap created by leave), but it works only for small quantity and distributed leaves: too many concurrent leaves can lead to the network to subdivide into multiple rings or go into non recoverable state But for most of the cases sequentially issued leave requests are no problem : we always reach a stable network.

3.3 Fault tolerance

The API that we needed to implement for the fault tolerance tests consist of two calls:

- POST /sim-crash: Simulate a crash. The node must stop processing requests from other nodes, without notifying them first. The network must detect that the node is not responding, and rearrange itself as if the node has left.
- POST /sim-recover : The "crashed" node must respond only to this call to go back in a responsive state.

We decided to simulate a crash using a flag to go in crashed state, and than make all messages answered with an error 500 - INTERNAL SERVER ERROR (but /sim-recover)

To be ready to handle a crash, we need something more than a successor for every node. With just a successor a node is unable of communicating forward in the ring, because its messages will be lost in the unresponsive successor. So in the new implementation:

- Each node keeps track of its successors' successor (second successor)
- There is a periodic task (every 500ms) that checks if the successor is alive
  - **if alive** store second successor
  - **else** set successor to second successor and second successor to none

In this way the network can always handle one node crash, and also multiple nodes failing at once, but network can only recover if they are not neighbors nodes. This strategy can be enhanced using other strategies (discussed in Conclusion section).

4 Evaluation

4.1 Stabilization Test

To evaluate our stabilisation procedure we conducted numerous experiments. For this we created multiple nodes, tell all of them simultaneously to join the same (arbitrary) node and wait until the chord network stabilizes. The reported time is the duration between the first issued join request and the stabilization.

To measure when the network stabilizes we run a stabilization test task every second. This task uses the `/node-info` API not retrieve the successor of each node. A network is seen as stable when the size of the set of successors is equal to the number of nodes in the network. Node that this test does not cover the case, where the network is divided into multiple rings. If this case needs to be tested for, one could walk around the ring (using the retrieved successors) and check the length of the path against the number of nodes.

We created networks with different number of nodes and measured their respective stabilization times. The results of these experiments can be seen in Figure 2.

4.2 Leave Stabilization

We evaluate the stabilization time after nodes leave the network. For the experiments half of the total nodes leave the network in a random order. The results can be seen in Figure 3.

Since a leaving node closes the gap in the network by communicating the new successors and predecessor to its neighbours, no stabilization is needed in most of the times. This can also be seen in the plot, since the stabilization times are close to instant.

4.3 Node Failure Tolerance

To provide a form of failure recovery each node additionally stores its successors successor (second successor). This allows for a recovery, if the successor fails, by replacing the successor with its second successor. As stated earlier, this only guarantees a recovery if only one nodes fails at one point in time. Nevertheless, the network is able to recover if there are no neighboring failing nodes.

To evaluate the failure tolerance we calculated the probabilities of neighboring failing nodes. The results can be seen in Figure 1. We retrieved those numbers by generating random sets of failing nodes and check if there are neighboring nodes in the set.

It can be seen, that small networks are especially prone to fail, even for only a small number of failures. As the number of nodes grows, the probability of a recovery gets larger, even for large number of simultaneous node failures.

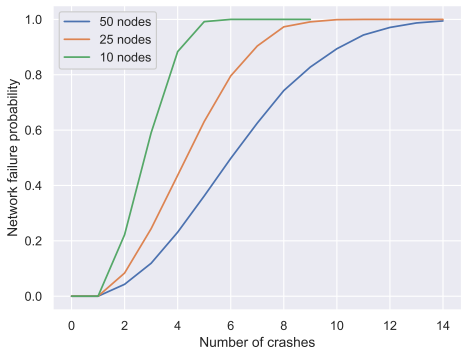

Figure 1: **Network failure probability**. This graph shows the probability of our chord network crashing when a certain number of nodes crash simultaneously. The network is not able to recover (and crashes) when two successive nodes crash. The data was retrieved running 10000 tests for each data-point.

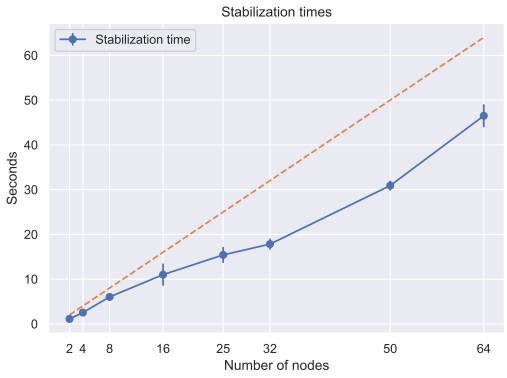

Figure 2: **Network stabilization time**. This graph shows the time required for the network to stabilize after creation. All nodes are created at once and join the same node at the same time. It can be seen that the stabilization time seems to be somewhat linear (orange line) with respect to the number of nodes. Each test was conducted 3 times and the graph shows the mean (line) and variance (error bar) of the measurements.

5 Discussion

In the Leave procedure one can expect some sort of regularity and linearity, because it's a is a neat and regular operation where a node communicates with their neighbours to make them collaborate before quitting the network. The plot shows what we expect, a linearity on the number of nodes. We can also see an increasing increment of the standard deviation, and it's not unexpected, because we are increasing the total number of the single operations.

From the crash test we got exactly the expected result: a node understand that it's successor is not responsive, cause in the sim-crash state the other node returns an error 500 - INTERNAL SERVER

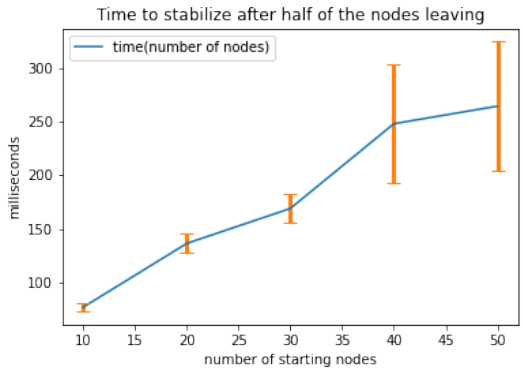

Figure 3: **Stabilization time after leave.** This image shows the time that our network needs to get stable after losing 1/2 of nodes. In red we have the error bars based on the standard deviation. Our leave procedure does not need stabilization in most of the cases. The stabilization time only increases for a larger number of nodes, because the number of issued HTTP requests for the stabilization test increases.

ERROR, so it immediately swaps his second successor with it's first one, obtaining a N-1 ring. The only irreversible issue happens if 2 nodes in a row are crashing simultaneously, this is more likely to happen, of course, increasing the number of node to be sim-crashed. In this occurrence, the chord ring becomes pretty much a linked list, with one of the end incapable of reaching the other side. Later in the conclusion section, we discuss how can we make the network more robust against this occurrence.

6 Conclusion

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. In this implementation, even if we manage to recover from a failure, we lose the information on the node in state of sim-crash.

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 two successors 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.

By the way our aim in this case was to make a network robust, and not fast and scalable onto big numbers of nodes. So we decided to implement just a list of next successors, with 2 successors, to have the possibility to recover in case of failure of a node. Note that if we have 2 failure of 2 successors in a row, the network doesn't have the possibility to recover, and instead of a ring, it becomes a linked list. We could have applied a fixed number of nodes as list of successor, to make it a little bit more robust for every successor more, against this type of occurrence. That wouldn't have helped us with speed or scalability, since everything remains with a linear complexity.

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.