Chordy: a distributed hash table

Anestos Nikolaos - Ektoras

October 16, 2014

1 Introduction

In this assignment we implement a distributed hash table following the Chord scheme.

Chord is a distributed hash table protocol where a set of nodes store key-value pairs for which they are responsible. There are several aspects of this protocol that we didn't cover in the assignment, such as that nodes store direct links (fingers) to many other nodes in the network to achieve optimized routing; but even so, we now have better understanding of how it works.

2 Main problems and solutions

The assignment is divided in three main issues:

1. Building a logical ring of nodes
2. Adding storage
3. Handle node failures

All of these had different challenges, but the description was very thorough and made my life easier.

2.1 Who is your predecessor?

In the first approach of the problem, we build the module **node1**, where we can start up a node and give him an Id, and then adding new nodes to the system which point to any existing node in the scheme. We also have a module **key** which has methods to provide a random number (we use it for node Id) and check if a node should be placed or not in a position in the ring.

Each node that joins the network must ask the node he is pointing to who is his predecessor. If the reply suggests that the new node is a better predecessor, the ring changes and the new node joins the network.

There are two ways to understand where the node should be placed is to compare the Id of the node we are querying and his predecessor.

1. If the Id of the node querying is higher than the predecessors but lower than the node he is asking, he squeezes himself in that spot
2. If the Id of the node querying is higher than the predecessors and higher than the node he is asking, he squeezes himself in that spot

That question was answered by ***key:between(Key,From,To)*** function which was the trickier in the first section.

```
between(Key,From,To)->
    if From > T o ->
            Key > From orelse Key =< To;
        From < To ->
            Key > From andalso Key =< To;
        true ->
            true
    end.
```

2.2 Am I responsible for this?

After successfully building a ring of nodes, we need to make them store key-value pairs. We introduce the storage to the system using **node2** (an updated version of node1) and **storage** modules.

Each node stores key-value pairs when the key is between his Id and his predecessors Id. We use add message to add a new key-value pair to the storage and lookup message to get the value for a given key. Whenever a node receives a message to add a key-value pair to his storage, he checks if he should be responsible for that and if not he forwards the message to his successor.

When a new node enters the ring though, we have to make sure that there aren't any stored values in the storage of a node which we wont run lookup successfully. The hardest part of this section, was **storage:split(Key,Store)** function. This is my solution:

```
split(Key,Store)->
    Sorted = lists:keysort(1, Store),
    split(Sorted,Key,[]).

split(Store,Key,Give)->
    case Store of
        []->
            {[],[]};
        [{SomeKey,SomeValue}|T]->
```

```
            case Key<SomeKey of
                true->
                    {Store,Give};
                false->
                    split(T,Key,[{SomeKey,SomeValue}]++Give)
            end
    end.
```

2.3 What if someone breaks the ring?

If we leave our system be vulnerable to one node's life, we have accomplished nothing. In **node3** module, we make the nodes aware not only for their successor, but also their successor's successor (Next). Also, each node monitors his successor and his predecessor. In case the predecessor dies, we just put our predecessor to nil. If our successor dies, we need to add the "Next" as our successor and get his successor as our new Next.

3 Evaluation

3.1 node1

I wrote a small test to quickly startup nodes. Each node added to the network at first points to the node that started first. After a while (depending on the time I set the nodes to run stabilize), the ring is fully formed. Using the probe message, I printed out on the shell every hop the message takes and at the end, when the probe did a full ring, I printed out the time it took and how many hops it did.

In some experiments where I started 100 nodes, I saw that if I tried the probe in the first few seconds of life of the network, not every node was connected. After a few seconds, when I run the probe test again, I saw that all 100 nodes where used to pass the message to their successor until the full ring was formed. Is there a better way to initialize the network?

3.2 node2

I used the test from the first part to start up the nodes and then added values to the storage. Even if I sent the add message to a node that was not responsible for it, he forwarded it to the next and it eventually reached the node that was supposed to store it. I used fixed numbers (hundreds) on the node Ids to make the testing easier. I started up nodes 100 to 2000 and added values with keys 400, 440, 450, 460, 470, 480, 500. The lookup method for key 400 showed me that the node with Id 400 had this value stored while every other value was stored in the node with Id 500.

After that I added a new node to the system, with Id 450. When the node took position between nodes with Id 400 and 500, it took over the responsibility for some of later node's storage. Checking again with the lookup message showed me that node with Id 450 now has values with keys 440 and 450 in his storage, and node 500 has the rest.

3.3 node3

Since i didn't have the time to work with replication, i used node1 to add the failure handling instead of node2 (just to have an easier file to work with). Using the same logic, i started several nodes waited for the ring to be stable, added a few more and then i started stoping them one by one, either by closing the shell window or with the stop message. I saw what was expected, that the ring was repaired and messages were still being passed through it.

There are of course many more issues that we should address in order to make the system more reliable or to improve performance.

4 Conclusions

Working is this assignment was unexpectedly fun. Even if what we did was a very simple version of Chord protocol, it made me think about what is need to be done in large scale real life applications, where performance and fault tolerance is crucial.