Chordy - a distributed hash table

Mattias Cederlund, mcede@kth.se

October 15, 2014

1 Introduction

The task was to implement a distributed hash table following the Chord scheme. In this implementation the nodes are in a ring structure and we are able to store and lookup key-value pairs. Failure handling is also included, but limited to one successor and in order to keep the data when nodes die it is replicated once.

The objective of the assignment was to gain a better understanding in distributed hash tables, how they work and how they could be implemented.

2 Building a ring

Nodes in this implementation should always (when stable) be structured as a ring with the help of two pointers, the successor and predecessor pointer. To maintain the ring structure a procedure stabilize(Pred, Id, Successor) was implemented. To stabilize a node will send request, self() to its successor and will get status, Pred in return, where Pred is the predescessor of its succesor. Depending on the value of Pred we should do the following:

- If it is nil the node should suggest adding itself as predescessor by sending notify, Id, self() to its successor.
- If it's pointing to itself we should do the same as if it were nil.
- If it's pointing back to us we should do nothing.

- If it's poiting to another node we need to check if we should position ourself in between the nodes or not.
  - If our Id is in between the predescessor of our successor and our successor we adopt that node as our successor and send request, self() to it. Causing it to reply with its predecessor and initialize the stabilization again.
  - Otherwise we should be in between the nodes by suggesting itself as predecessor by sending notify, Id, self() to the successor.

When a node receives a suggestion about a predecessor it will do some research before adopting it:

- If the node doesn't have any predecessor it adopts it right away.
- If the node have a predecessor it checks if the suggested predecessor is in between the current predecessor and itself. If it is the node will adopt the suggested predecessor and otherwise not.

We do not need to inform the new node about our decision since it will eventually issue a new stabilization and find out anyway.

When testing the initial implementation of the ring of nodes (on the same machine) everything worked out and I managed to send a probe around the ring with it passing every node.

3 Adding a store

Adding a store was the simplest part of this assignment. To add and lookup a value we check if the key is in between our predecessors Id and the current nodes Id. If it is, the key-value pair should be stored in the node, otherwise the request should be forwarded to its successor. When a new node joins the ring we should hand over the values with keys lower than the new predecessors key.

I continued by doing some performance tests, but I only had access to one physical machine so all tests are performed in a single Erlang node. Adding and looking up 4000 values in a ring with a single node took about 1,6 seconds. With 2 nodes and the keys evenly distributed (nodes with Id 1 and 500,000,000) it was slightly quicker, 1,4 seconds. With 4 nodes it took about 1,3 seconds. My conclusion is that searching of the values in a long list is what takes a lot of time here, while forwarding messages are quick. In fact, this is even more visible when running the test with 10000 values. Then the test will use about 9 seconds with one node and only 3-4 seconds with 4 nodes. Although I guess this would be different with nodes on different physical machines, where network traffic is limiting the transfer rate of messages between nodes.

4 Handliing failures and replication

To handle failure (nodes going down) we keep a pointer to the successor of our successor (next successor). When a status message is sent the successor is also included and the stabilize function is now updating the next successor pointer when adopting a new successor. The next successor should then be set to the one we previously had as successor before adopting the new one.

To detect nodes going down we use the built in monitor functions (with wrappers). Every time we adopt a new successor or predecessor we should now monitor it and if we already were monitoring one when adopting the new one we should demonitor the old one first. When we receive a 'DOWN' message from the monitors we should find out if it's coming from our sucessor or predecessor. If its coming from the predecessor we do nothing but removing the reference to it since someone will eventually suggest a new one when running the stabilize function. If it's coming from the successor we should use our next successor as successor instead.

Although, we still have the problem of false detection and if a node is temporary down it might have been removed from the ring by the other nodes. If the node itself still has OK references to its successor and predecessor it will eventually run stabilize and be included in the ring again. It's worse if it thinks that those nodes are down at the same time as the next successor. Then it won't be able to reconnect to the ring by itself.

To replicate data in the ring every node has a replica storage in their successor. Every time we add a key-value pair it is now also forwarded to the replica, which is replying the client if it succeded to make sure we always have a replica of the added data. If the confirmation message got lost and the client resends the data nothing would happen since the storage is implemented to just replace the old data with new data (now without any change).

If we detect that our predecessor dies we merge our storage with the replica and if we detect that the successor dies we send a replica of our data to the new successor for it to be replicated there. If a new node joins the network it will be handed over a part of the replica together with the storage it is responsible for. It will also need to hand over a replica of it's new data to its successor when receiving a handover.

5 Conclusions

I think this assignment was by far the toughest one since there was a lot of the implementation missing (unlike the previous assignment, groupy) and a lot of the implementation was open for my own intepretation of the problem. Nevertheless I really think it was a good way of improving my understanding in how a Chord (ish) distributed hash table works and some of the problems we need to tackle to make it fault-tolerant.