Report 5: Chordy

Thomas Galliker

October 6, 2011

1 Introduction

In this assignment we were asked to implement a distributed hash table similar to the peer-to-peer protocol "chord" which was developed at MIT. Distributed hash tables (DHT) mainly provide three big advantages:
- Physical (node-wise) distribution of information without any central coordination.
- Fault tolerance: Nodes may come and go (crash).
- Scalable and efficient handling of stored information.

We simply started building a node module that was able to connect a precessing as well as a succussing node. Each node was addressed by an unique identifier. A sophisticated algorithm is responsible for bringing nodes into the right order and making data stores of nodes accessible to clients. Features like fault tolerance and replicated data stores were implemented in the last stage of this exercise.

2 Main problems and solutions

Most of the really dirty problems came up as I extended the node to support failure detection. Not mainly because of the failure detection function, but because the representation of successor and predecessor nodes has changed from {Key, Pid} to {Key, Ref, Pid}. (Ref is used to monitor neighboring nodes). Many function calls had to be adapted which war a terrible work since we don't know exactly where the old message format is used. (Thanks god for type proof programming languages!)

After introducing debug command I experienced problems with the data stores of my nodes. As soon as a new node is sneaked in between two neighboring nodes, both existing nodes split their data stores and hand over them to their new neighbor. In this process it is important, that 1. we split the data stores at the right point and 2. we give away the correct part.

But not splitting is an important process, also the merge of key elements was a crucial task. That sounds easy, but it has actually caused headache! Imagine a ring with two nodes: node 0 and node 6 (see Figure 1).

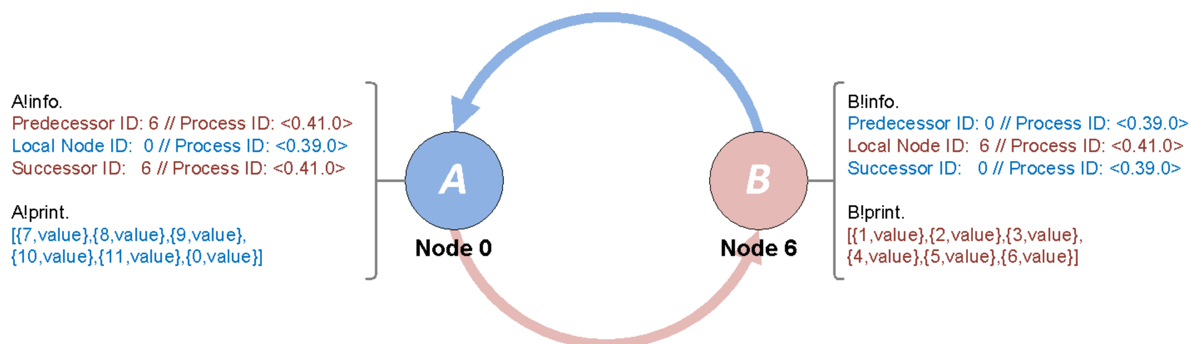

Figure 1: Setup of a ring network with two nodes.

Twelve messages (key 0..11) are being generated. Node 0 is responsible for messages 7,8,9,10,11,0 while node 6 takes care about messages 1,2,3,4,5,6. If we add two more nodes between nodes 0 and 6, we expect that both existing nodes split their store and hand over parts of the store the their new neighbors.

This works only fine if elements are NOT sorted during split and merge operations! We must preserve the exact order of keys, otherwise it might happen that keys get mixed up which ends up in a disaster... (Keys are then on nodes that will never be contacted for lookups, means, these keys use storage but are in fact lost).

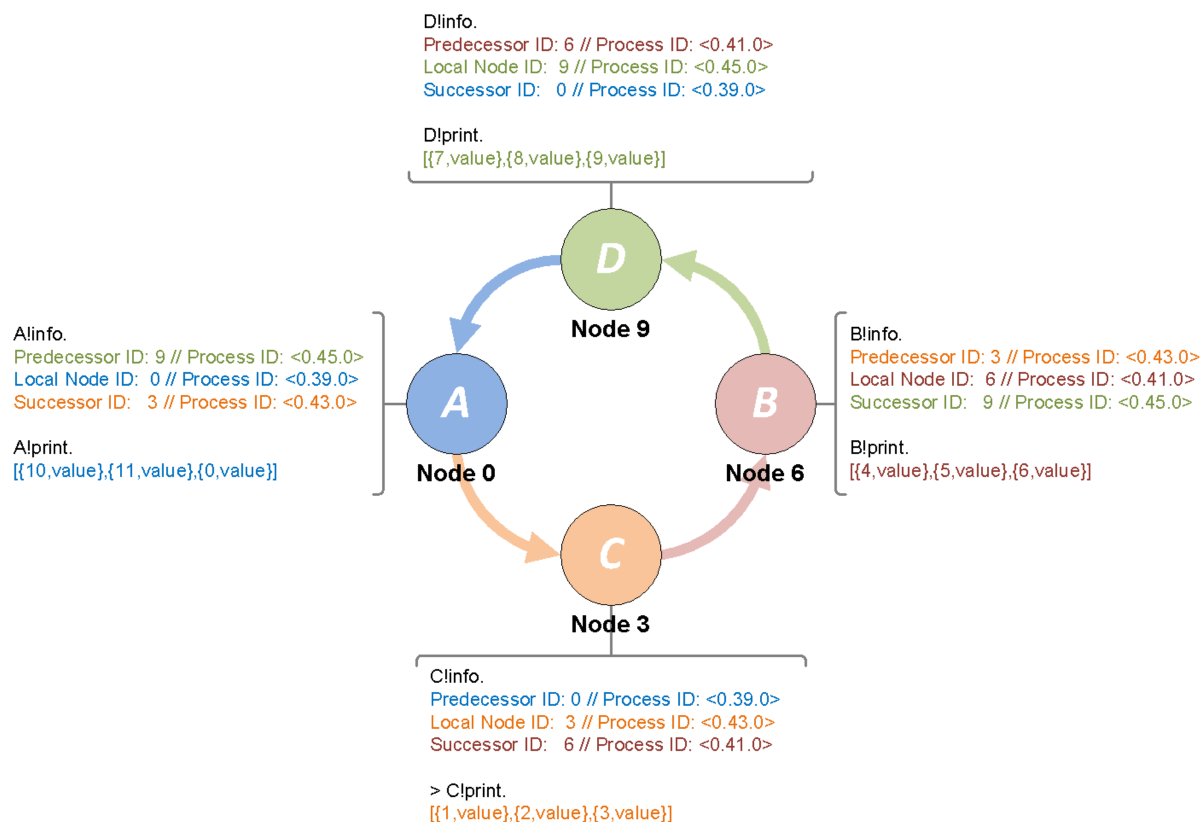

Figure 2: The ring was extended by node 3 (C) and 9 (D) what caused a redistribution of the keys.

3 Evaluation

The following tests are thought to check the correctness and - above all - the efficiency and scalability of our distributed hash table. I have set up a test script that allows me to determine the number of nodes in the ring as well as how much read and write operations on each node I want to perform.

I expected to get an exponentially decreasing amount of time for each node added to the ring. After some test runs with very strange results, means, very long read/write times for just a few requests. The reason was found in a faulty number assignment principle which was used in the test script. Each node just got an integer value as unique identifier. Starting from 1, increased by 1. What happened? If I have a ring with 10 nodes which is storing 1000 key-value pairs, what happens if these 10 nodes have ID's like 1,2,3,...10? - The distribution of such a ring is unusable! Only 2 nodes are actually storing 99% of the keys while the other 8 care about 1% of the data. Such a system doesn't scale well. To get significant test results, I organised nodes in a way that they are evenly distributed. To get a well-distributed ring, productive systems should implement algorithms to balance nodes from time to time.

One of the tests made is illustrated in Figure 1. A number of 8000 successive read and write commands had been issued to a different number of concurrently running nodes. The more nodes, the less time is taken to get a respond. It is necessary to mention, that total number of requests was distributed amongst all nodes.

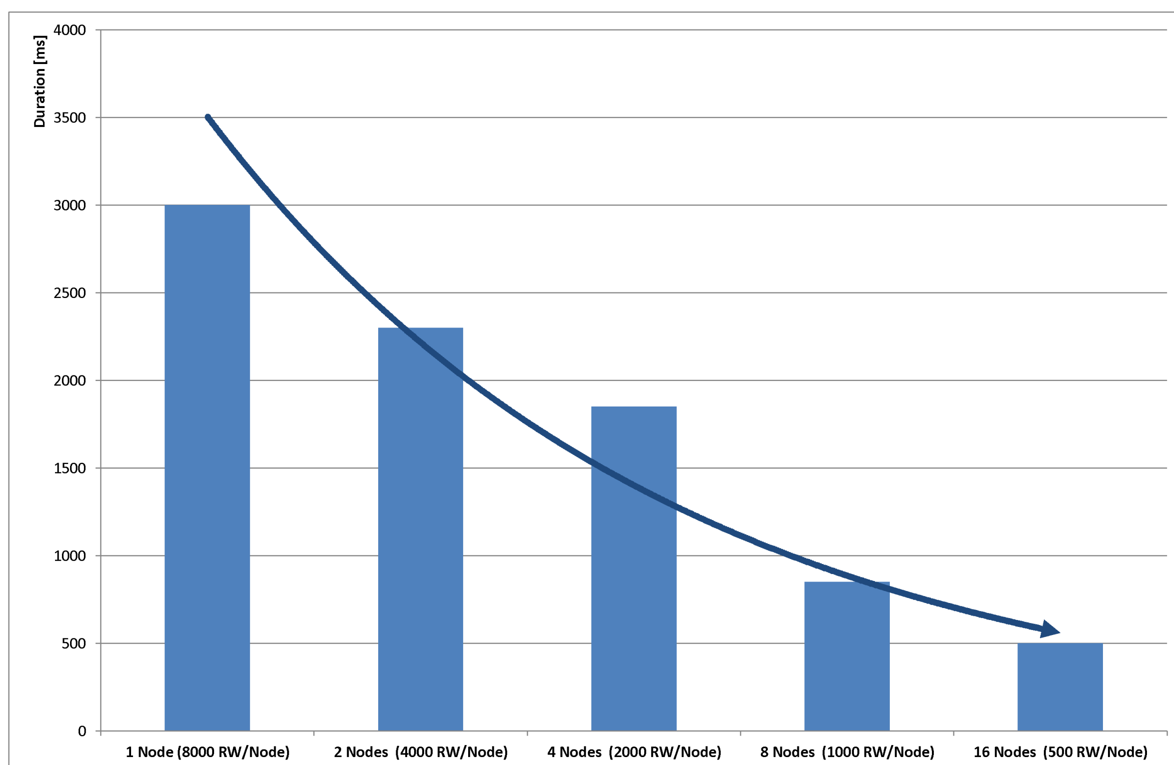

Figure 3: Performance tests with evenly distributed nodes to demonstrate the scalability of this distributed ring network.

4 Optimisation

We could introduce an automatic balancing algorithm based on statistical information exchange between neighboring nodes. Neighbors could tell each other how much data (key-values pairs) they maintain in their stores. If the difference is significant, they could hand over parts of the store. This is not as easy as it sounds because in our solution the node identifier depends on the range of key-value pairs that they are responsible for.

Further improvements could be made in the routing algorithm. Efficient routing decisions are crucial in large ring-like overlay networks since they help reducing unnecessary traffic. We could introduce a "finger table" as it is used in chord protocol. This could reduce the longest way to find a key in the ring from $O(\mathrm{N})$ down to $O(\log(\mathrm{N}))$.

5 Conclusions

This was finally the exercise that cost me the most effort compared to all the other assignments of course ID2201! Nevertheless, it was very exiting to see how ring-based distributed systems work. Despite of all advantages that Erlang brings along, if one has to make fundamental changes in the data structure e.g. of message formats, it can become a nightmare! Thanks god for custom data types and strict type checking as it comes with other programming languages.

I want to take this opportunity to thank our teacher, Johan Montelius, and his assistants for their dedication and the support throughout the term.

6 Links

http://pdos.csail.mit.edu/chord
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.125.7646rep=rep1type=pdf
http://en.wikipedia.org/wiki/Chord_(peer-to-peer)
http://de.wikipedia.org/wiki/Chord
http://en.wikipedia.org/wiki/Distributed_hash_table
http://www.thomasgalliker.ch