Chordy: a distributed hash table

Aristotelis-Antonios Kotsias

October 8, 2020

1 Introduction

This assignment covers the implementation of Chordy, a distributed hash table based on Chord which is a distributed lookup protocol. Chord protocol maps a key to a node. Data storage is build upon the Chord protocol where each key is associated with a value and it is stored in the respective node. Chordy is able to adapt to the changes meaning that nodes can come and go while data is not lost and is always accessible.

The goal of the assignment is to create a network of nodes in the form of a ring where each node has a unique key identifier. Moreover, each node will be able to store $key - value$ data where the key is a unique identifier while the value can be anything.

2 Main problems and solutions

The first problem that I faced was the difficulty of understanding the protocol and how the system should work. It is complicated system where different parameters; such as current node, predecessor node and successor node, need to be taken into consideration. However, studying the Chord paper together with online tutorials I was able to a gain a theoretical background. The assignment was separated into two tasks; creating the ring of nodes and building a storage in each node for saving and looking up key-value data.

2.1 Ring

At first the ring has to be created. Each node of the ring is associated with a unique key (aka identifier) which is used to distinguish it from other nodes and also avoid duplicates. The uniqueness of the key is achieved by generating a random number from 1 to 1.000.000.000 (30-bits). For the simplicity of the assignment, no hash function such as SHA-1 is used as it should have normally been done. Before a new node is added to the ring, first it will contact a node that already exists in the ring. Secondly, the new node's key is compared with the keys of the contacted node and its predecessor in order to determine its location in the ring.

The correct order between the nodes is achieved by stabilizing them periodically. This is done by a node sending a {request,self()} message to its successor and then expecting a {status, Pred} in return. When it knows the predecessor of its successor it can check if the ring is stable or if the successor needs to be notified about its existence through a {notify, {Id, self()}} message.

2.2 Store

After the creation of the ring it is time to add a local store to each node. This store is used to save {key,value} data to each node and also retrieve this data when it is needed. The store is in the form of a list of tuples, i.e. [{Key1, Value1}, {Key2, Value2}, {Key3, Value3}, ...]. The functionality of the store is easy. In order to add new {Key, Value} we must first determine if our node is the node that should take care of the key. A node will take care of all keys from (but not including) the identifier of its predecessor to (and including) the identifier of itself. If we are not responsible we simply send a add message to our successor. The lookup procedure is similar, where a node needs to do the same test in order to determine if its the one responsible for the key. If not, it simply forwards the request to its successor.

Last but not least, something that has to been taken into consideration is that when a new node is added to the ring then some its successor's stored {Key, Values} data must be transferred to the new node's store.

3 Evaluation

For the evaluation, the given test module was used to evaluate the functionality of the ring and the store.

At the beginning, I tested that the ring is created. In order to do that, a probe message was passed around the nodes of the ring. This is done by spawning a couple of nodes and then sending a probe message from the first node and forward it until it reaches back the first node. This means that the message did a full circle by starting and ending at the same node. At the end, the path that the message followed together with the time it took it, are printed on the output. The path, confirms that the the ring is "closed" and also that the nodes have been placed in the right order.

Next, I checked that a new node can be added on the ring. This was done by notifying a node in the ring about the existence of a new node. Then, the new node would be placed on the right spot inside the ring. This was confirmed by sending a new probe message and checking that the new node has been added into the path.

Moreover, the store was tested by flooding the ring with a number of {Key, Value} data and then checking where this data had been stored. It was noticed that the higher the number of {Key, Value} data the higher the failed lookup procedures were. This is normal, since it was harder to find the pair in the first node.

Finally, I tested that when a new node is added to the ring then the store of the new node's successor would split. This was done by printing out the store of the successor before and after the addition of a new node.

4 Conclusions

In conclusion, this was the most interesting and hardest assignment I had faced so far. Building the ring and testing it was demanding as small details had to be taken into consideration. At first, working with a ring structure is not trivial, especially when comparing the keys but after a while it gets natural.