ID 2201 Report

Chordy: A Distributed Hash Table

Yilai Chen

Introduction

This report presents an implementation of a distributed hash table (DHT) based on the Chord scheme. A hash table is a data structure that implements an associative array abstract data type that can map keys to values. A DHT is simply a hash table that has been distributed across multiple nodes in a network. A node will store the values of all the keys it is responsible for.

In Chord, nodes are arranged in a ring structure where each node has a successor (i.e., the next node in the ring) and a predecessor (i.e., the previous node). Nodes also have a unique ID. Chord specifies how keys are assigned to nodes and how a node can retrieve the value of a given key by locating the node responsible for that key. The main topic of this research is to gain insight into the problems caused by DHT and how to solve them.

Establishment of structure and function

In the initial implementation of DHT, the focus is on getting a valid ring structure up and running. Node IDs are generated by **key.erl** through a random function, which randomizes values between 1 and 1 000 000 000 as IDs. The key module also contains a function between(Key, From, To) that checks whether a key is between two other keys in the ring.
However, the main work is implemented in the **node.erl**, among which the function **stabilize(Pred, Id, Successor)** is the most important. It ensures that the complete ring structure is maintained when a new node enters the ring. A node sends a request message to its successor node to obtain the predecessor node of its successor node. Based on the value of Pred, the node can propose to become the new predecessor node of the successor node. The successor node will receive the proposal notification message from the node, call the notification function and decide whether the node should become the new predecessor node. The procedure **schedule_stabilize/1** is called when a node is created or every 100 milliseconds.

Subsequently a **storage.erl** was added to the DHT. The storage is implemented as a list of tuples **{Key, Value}** and we can then use the key function from the list module to search for entries. A node will be responsible for all keys from (but not including) the Id of its predecessor up to (including) its own Id. If a node is not responsible, it will simply send an add message to its successor. In addition to this, when a new node is added to the ring, it should take over part of the responsibility of storing the key-value pairs from its successor. This is handled by the function **handover(Id, Store, Nkey, Npid)** . However, to add a new key-value pair to the DHT, an add function was implemented which checks if the node is responsible for a given key or if it should forward it to its successor and if so, simply calls the add function from the storage module to put the data in its list of tuples. A similar process is performed for the lookup function to retrieve the data associated with a particular key.

In order to always maintain a fully connected ring structure, the ring must be able to self-heal when a node crashes. This is usually done with a built-in **monitoring function**. If a node's current successor crashes, it must be able to update the successor to the next node, which is the successor of its successor. Introduce a new pointer to the successor of the successor and name it **Next**. The node and stabilize functions are extended with an extra parameter Next. If a node's predecessor dies, there is no way to find the predecessor of a node's predecessor, but if a node's predecessor is set to nil, sooner or later some node will present itself as a possible predecessor. If the successor dies, the next node is adopted as the successor, and the new successor is then monitored and the stabilization procedure is run.

Test

```
5> P1 = test:start(node).
Id: 443584618
<0.107.0>
6> P2 = test:start(node,P1).
Id: 723040206
<0.110.0>
7> P3 = test:start(node,P2).
Id: 945816365
<0.113.0>
8> P4 = test:start(node,P3).
Id: 501490715
<0.116.0>
9> P5 = test:start(node,P1).
Id: 311326755
<0.119.0>
```

First, we add five nodes in sequence to form a ring.

```
10> test:add(500,"hello",P1).
Storing key 500 at node with Id: 311326755
ok
11> test:add(500_000_000,"world",P2).
Storing key 500000000 at node with Id: 501490715
ok
12> test:add(300_000_000,"bingo",P2).
Storing key 300000000 at node with Id: 311326755
ok
13> test:lookup(500,P4).
Found key 500 at node with Id: 311326755
{500,"hello"}
```

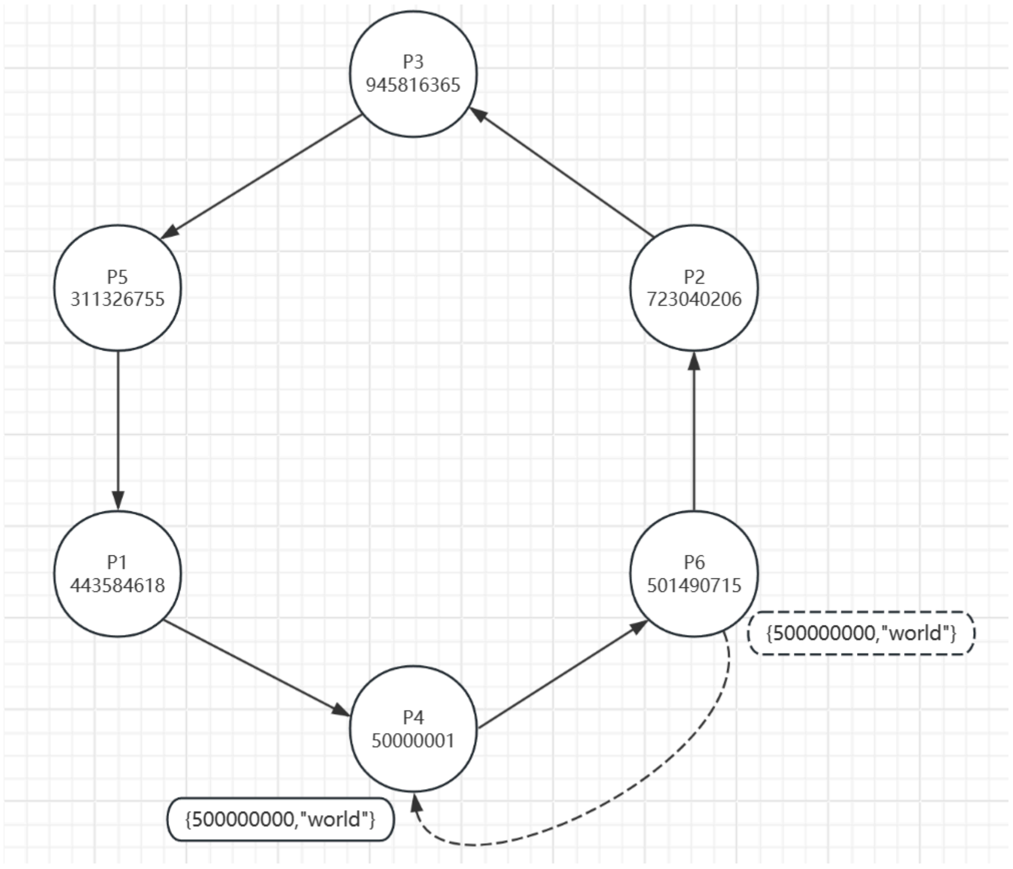
Then test the add and lookup functions

```
14> P6 = test:start(node3,P5, 500_000_001, special).
Id: 500000001
<0.126.0>
18> test:lookup(500_000_000,P4).
Found key 500000000 at node with Id: 500000001
{500000000,"world"}
```

We then use a special node start method to get a node with an ID of 500000001, and re-lookup the key-value pair with a key value of 500000000 to test the handover function.

```
19> P2 ! probe.
Probe time: 205 micro seconds
 Nodes: [501490715,500000001,443584618,311326755,945816365,723040206]probe
20> test:kill(P3).
Successor of #Ref<0.1260050464.373030913.61704> died
true
21> P2 ! probe.
Probe time: 103 micro seconds
 Nodes: [501490715,500000001,443584618,311326755,723040206]prob
```

Finally, we test probe and kill.



Conclusions

First, it took me a while to understand the concept of the chord scheme. Then it was time to implement the key module with the between function, which required careful thinking to ensure there were no errors. And it took me a while to understand why the nodes were reversed in the probe list. Also, it took a while to set up and test the failure handling, add some test methods to prove full functionality, and figure out what was going on.