Chordy - a distributed hash table

Chrysoula Dikonimaki

October 6, 2021

1 Introduction

The goal of this exercise was to implement Chordy, a distributed hash table. The nodes are placed in a ring. Each node has a unique key and a store with key-value pairs for which it is responsible. The store contains all the keys in the range (Pkey, Id], where Pkey is its Predecessor's key.

1.1 Debugging message

I added a new message called 'info' for debugging purposes. When a node receives a message like this it prints the most important information: the predecessor, the successor, the store etc.

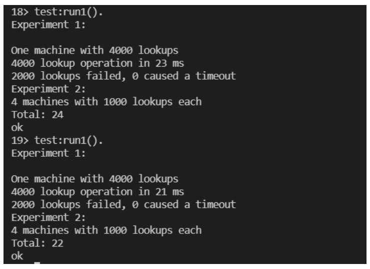
2 Performance

2.1 Experiment 1

The function called run1() creates 1000 messages that will be added to the ring and 1000 messages that won't added. The rings contains 4 nodes. I looked up for 4000 messages asking the first node added to the ring. Then I looked up for the same 4000 messages but asking 4 different nodes giving to each one of them 1000 messages. I ran the experiment twice and the results are shown in Figure 1. In general, the time will depend on how the keys are distributed.

2.2 Experiment 2

The results of the second experiments are shown in Table 1. I ran the function run2 setting the parameters shown in the table. It is obvious, that when we increase the number of Nodes the time increases.



```
18> test:run1().
Experiment 1:

One machine with 4000 lookups
4000 lookup operation in 23 ms
2000 lookups failed, 0 caused a timeout
Experiment 2:
4 machines with 1000 lookups each
Total: 24
ok
19> test:run1().
Experiment 1:

One machine with 4000 lookups
4000 lookup operation in 21 ms
2000 lookups failed, 0 caused a timeout
Experiment 2:
4 machines with 1000 lookups each
Total: 22
ok
```

Figure 1: Running the first experiment twice

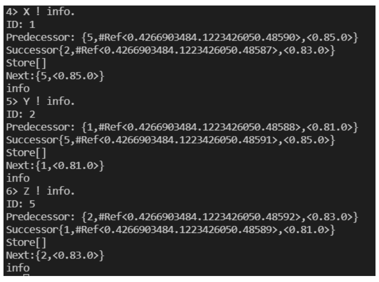
Number of Nodes	Time (in ms)
100	3
101	3
1000	16
10000	594
100000	919

Table 1: Experiment 2: number of pairs = 100

3 Handling Failures

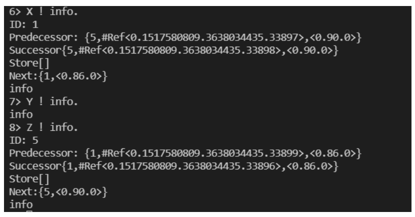
We should detect and handle failures by monitoring each node's successor and predecessor. When the predecessor dies it will be handled automatically but when the successor dies our new successor is the previous successor's successor. That's why we have to save the next node. If we wanted to handle many failures at the same time, we could have a list with the next nodes, but it would be inefficient.

I created 3 nodes (X, Y, Z) and then I stopped Y. Figure 2 shows the information for each node before Y stops and figure 3 shows the same information for X and Y after Y died.



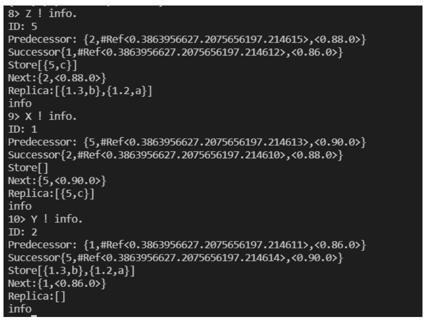
```
4> X ! info.
ID: 1
Predecessor: {5,#Ref<0.4266903484.1223426050.48590>,<0.85.0>}
Successor{2,#Ref<0.4266903484.1223426050.48587>,<0.83.0>}
Store[]
Next:{5,<0.85.0>}
info
5> Y ! info.
ID: 2
Predecessor: {1,#Ref<0.4266903484.1223426050.48588>,<0.81.0>}
Successor{5,#Ref<0.4266903484.1223426050.48591>,<0.85.0>}
Store[]
Next:{1,<0.81.0>}
info
6> Z ! info.
ID: 5
Predecessor: {2,#Ref<0.4266903484.1223426050.48592>,<0.83.0>}
Successor{1,#Ref<0.4266903484.1223426050.48589>,<0.81.0>}
Store[]
Next:{2,<0.83.0>}
info
```

Figure 2:



```
6> X ! info.
ID: 1
Predecessor: {5,#Ref<0.1517580809.3638034435.33897>,<0.90.0>}
Successor{5,#Ref<0.1517580809.3638034435.33898>,<0.90.0>}
Store[]
Next:{1,<0.86.0>}
info
7> Y ! info.
info
8> Z ! info.
ID: 5
Predecessor: {1,#Ref<0.1517580809.3638034435.33899>,<0.86.0>}
Successor{1,#Ref<0.1517580809.3638034435.33896>,<0.86.0>}
Store[]
Next:{5,<0.90.0>}
info
```

Figure 3:



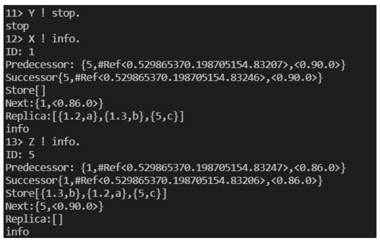
```
8> Z ! info.
ID: 5
Predecessor: {2,#Ref<0.3863956627.2075656197.214615>,<0.88.0>}
Successor{1,#Ref<0.3863956627.2075656197.214612>,<0.86.0>}
Store[{5,c}]
Next:{2,<0.88.0>}
Replica:[{1.3,b},{1.2,a}]
info
9> X ! info.
ID: 1
Predecessor: {5,#Ref<0.3863956627.2075656197.214613>,<0.90.0>}
Successor{2,#Ref<0.3863956627.2075656197.214610>,<0.88.0>}
Store[]
Next:{5,<0.90.0>}
Replica:[{5,c}]
info
10> Y ! info.
ID: 2
Predecessor: {1,#Ref<0.3863956627.2075656197.214611>,<0.86.0>}
Successor{5,#Ref<0.3863956627.2075656197.214614>,<0.90.0>}
Store[{1.3,b},{1.2,a}]
Next:{1,<0.86.0>}
Replica:[]
info
```

Figure 4:

4 Replication

Saving the replicas at the successor is a good idea because if a node crash its pairs will be at the right place and we won't do something special to get them to its successor. However, there is always a chance to crash both the node and its successor at the same time and data won't be found anywhere.

Figure 4 shows the information about 3 nodes (X, Y and Z). Y has the values a and b while Z has the value c. In Figure 5 we can see the information of X and Z after stopping the node Y. We can see that everything works correctly.



```
11> Y ! stop.
stop
12> X ! info.
ID: 1
Predecessor: {5,#Ref<0.529865370.198705154.83207>,<0.90.0>}
Successor{5,#Ref<0.529865370.198705154.83246>,<0.90.0>}
Store[]
Next:{1,<0.86.0>}
Replica:[{1.2,a},{1.3,b},{5,c}]
info
13> Z ! info.
ID: 5
Predecessor: {1,#Ref<0.529865370.198705154.83247>,<0.86.0>}
Successor{1,#Ref<0.529865370.198705154.83206>,<0.86.0>}
Store[{1.3,b},{1.2,a},{5,c}]
Next:{5,<0.90.0>}
Replica:[]
info
```

Figure 5:

5 Conclusions

Building a hash table is not that hard, but we have to think a lot about the possible failures and ways to handle them. The advantage of having a distributed store is the fault tolerance that it provides because even if the node fails its values will be saved somewhere and they won't get lost. However, storing data remotely has the overhead of transferring all these data over the network and maintain redundant copies. Things can get really hard if we want to modify the values because we will need to keep the copies updated as well.