Report 5: Chordy, A Distributed Hash Table

Luxiang Lin

October 9, 2024

1 Introduction

In this assignment, the focus is on constructing a simple distributed hash table based on the Chord algorithm. The two main tasks in the homework are to build a ring structure and implement the storage of key-value pairs.

2 Main problems and solutions

To meet the requirements of this assignment, which involves developing a distributed hash table system, I have followed the guidelines outlined in the assignment description and divided the implementation into two parts.

2.1 The Ring

In a ring structure, every node inside the ring should be aware of its predecessor and successor and keep track of them. This means that every time a new node is inserted, the nodes should update their predecessor and successor if needed. To achieve this, I implemented an important method called 'between(Key, From, To)' in the key module. This method can determine whether the input key is located inside the interval (From, To].

2.2 The Storage

After completing the ring structure, I moved on to implement the storage module. I used a list of tuples to keep the key-value pairs, and a node is given an empty store list once it is initialized. When adding a new node, the system must consider where this node should be placed and how the existing key-value pairs should be split (i.e., which part should belong to which node).

3 Evaluation

3.1 Adding Values

In the first test, I started a ring with 4 nodes, then added values and checked whether the values could be correctly looked up.

```
1> node2:start(5).
<0.87.0>
2> node2:start(10, <0.87.0>).
<0.90.0>
3> node2:start(15, <0.90.0>).
<0.93.0>
4> node2:start(20, <0.93.0>).
<0.96.0>
5> <0.87.0> ! probe.

Time =1, Ring: [5,10,15,20]
probe
6> <0.93.0> ! probe.

Time =102, Ring: [15,20,5,10]
probe
7> test:add(9, "hello", <0.96.0>).
ok
8> test:lookup(9, <0.87.0>).
{{9,"hello"},10}
9> test:add(4, "hello", <0.93.0>).
ok
10> test:lookup(4, <0.93.0>).
{{4,"hello"},5}
```

Figure 1

3.2 Increase the Load

The second test I conducted was to increase the workload and observe how it influences the time my procedure consumes.

First, I had one test process add 4000 values and perform lookups on one node.

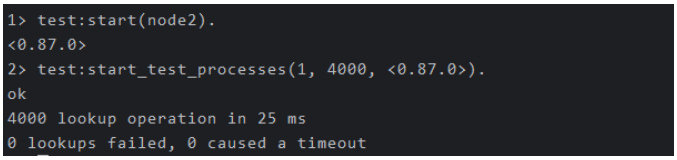

Figure 2

Then, I spawned 4 test processes and let each process add 1000 values and perform lookups on one node:

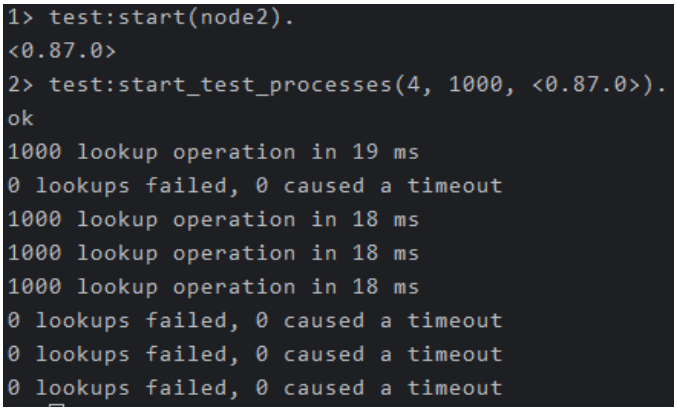

Figure 3

It is clear that it is more efficiency when client processes working concurently.

After this, I spawned 4 test processes and let each process add 1000 values and perform lookups on 4 nodes:

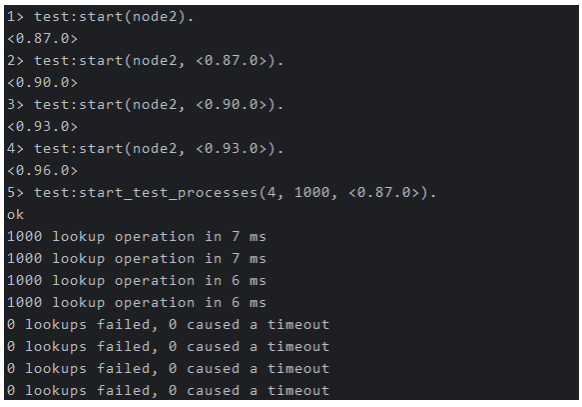

Figure 4

As shown in the graph, the time the procedure consumes decreased significantly compared to 18ms and 19ms when there was only one node in the ring.

Finally, I increased the total amount of request to 10000:

```
1> test:start(node2).
<0.87.0>
2> test:start(node2, <0.87.0>).
<0.90.0>
3> test:start(node2, <0.90.0>).
<0.93.0>
4> test:start(node2, <0.93.0>).
<0.96.0>
5> test:start_test_processes(4, 2500, <0.87.0>).
ok
2500 lookup operation in 40 ms
2500 lookup operation in 40 ms
0 lookups failed, 0 caused a timeout
2500 lookup operation in 40 ms
2500 lookup operation in 40 ms
0 lookups failed, 0 caused a timeout
0 lookups failed, 0 caused a timeout
0 lookups failed, 0 caused a timeout
```

Figure 5

As the workload increased dramatically, it took the system more time to handle the requests.

4 Conclusions

In this assignment, I deepened my understanding of the Chord scheme and learned how to apply its theoretical principles in practice. It was insightful to see how the concepts translate into real-world implementation, especially in managing node relationships and efficiently storing and retrieving data within a distributed hash table.