Report 5: Chordy

Chen Sihan

October 6, 2020

1 Introduction

The goal of this project is to implement a DHT (distributed hash table) following the Chord scheme.

2 Main problems and solutions

2.1 Build a basic ring in Erlang

Firstly we should build a ring of nodes. In the paper we firstly build the ring of nodes and stabilize it and then update the entries of the "finger tables" of the nodes. To build the ring of nodes, we have the init and connect functions. If we want to create a new node in the ring, we should specify the successor of this node.

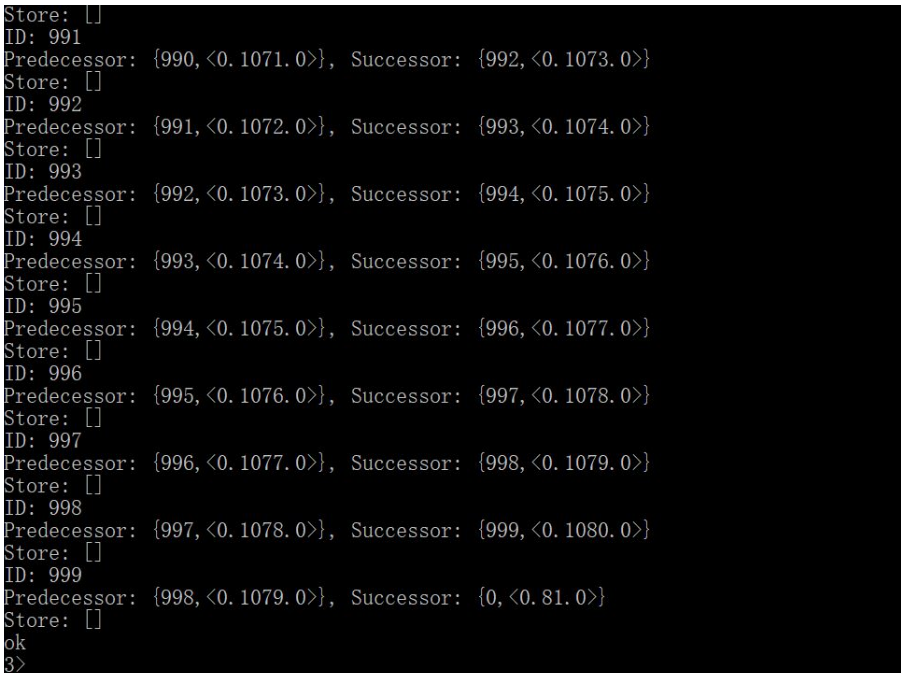
After that, we should stabilize the connections of this node to its predecessor and successor because only specifying its successor is not enough. In that case we have the stabilization process right after the connect process. This process is to establish a fully doubly linked list relationship between the added node and its predecessor and successor.

2.2 Add key-values and update the finger tables

After we stabilize all the connections between nodes, we try to add key-values and store them in them. In the program we have a Store list on each node to store all the k-v pairs. In the program, we store the all the keys (with values) between the interval closed at the right $(n.predecessor, n]$ in the Store list of n. As shown in Figure 2, we can see the node with ID 996 store all the keys between its predecessor and itself, namely in $(995, 996]$, the $[996, 13, 996, 716]$.

3 Bonus: Fault tolerance

Here the program only deals with the situation that one node in the ring is down. If a row of two nodes are down together then we are doomed.

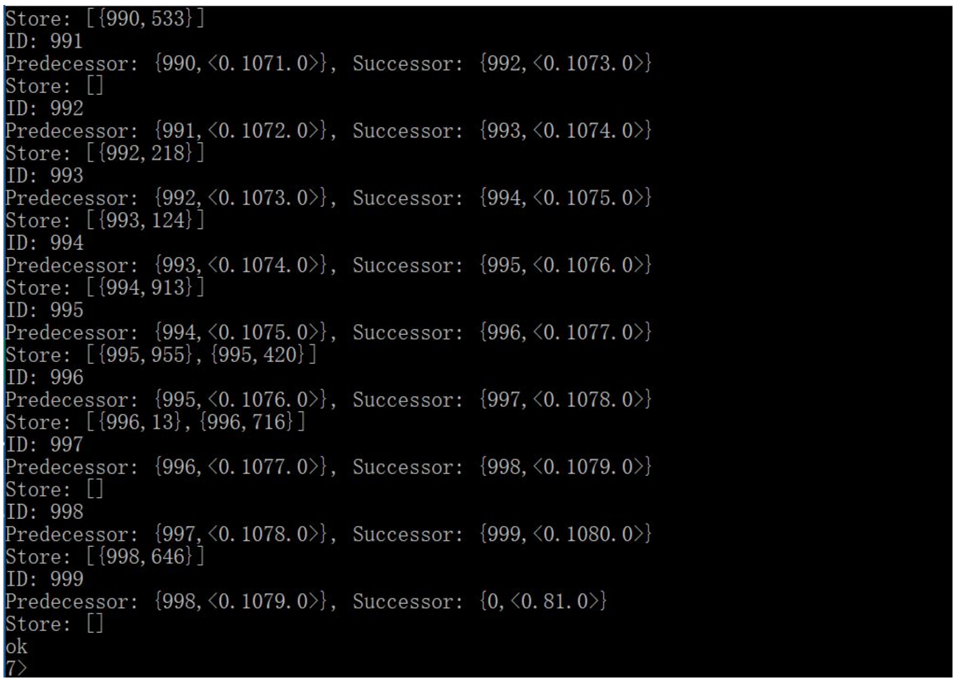


```
Store: []
ID: 991
Predecessor: {990,<0.1071.0>}, Successor: {992,<0.1073.0>}
Store: []
ID: 992
Predecessor: {991,<0.1072.0>}, Successor: {993,<0.1074.0>}
Store: []
ID: 993
Predecessor: {992,<0.1073.0>}, Successor: {994,<0.1075.0>}
Store: []
ID: 994
Predecessor: {993,<0.1074.0>}, Successor: {995,<0.1076.0>}
Store: []
ID: 995
Predecessor: {994,<0.1075.0>}, Successor: {996,<0.1077.0>}
Store: []
ID: 996
Predecessor: {995,<0.1076.0>}, Successor: {997,<0.1078.0>}
Store: []
ID: 997
Predecessor: {996,<0.1077.0>}, Successor: {998,<0.1079.0>}
Store: []
ID: 998
Predecessor: {997,<0.1078.0>}, Successor: {999,<0.1080.0>}
Store: []
ID: 999
Predecessor: {998,<0.1079.0>}, Successor: {0,<0.81.0>}
Store: []
ok
3>
```

Figure 1: Build 1000 nodes as a ring (doubly linked list)

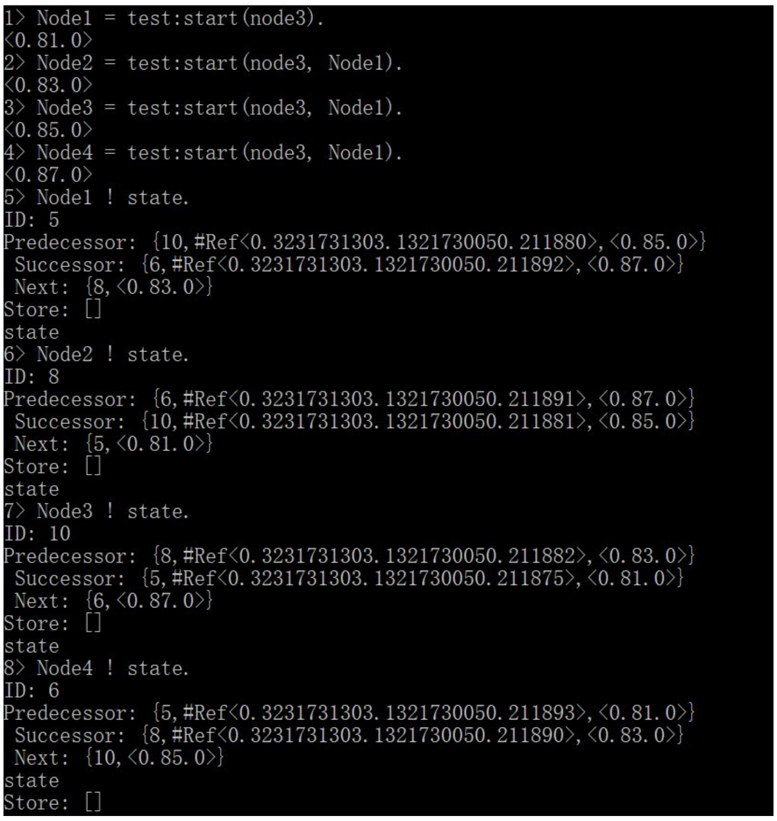
The solution here is to find a Next node. As shown in Figure 3, the ring is built with 4 nodes with the structure $Node1 => Node4 => Node2 => Node3 => Node1...$, and there is a next node to show which the successor of the successor of the current node is. Each node will monitor its predecessor and successor. If the node dies, its successor will demonitor the node and sets the successor's predecessor to be nil and wait for someone to present themselves as a proper predecessor. Meanwhile, the predecessor of the node will demonitor the node, and set a new successor (namely the Next node).

In Figure 4, we can see if we stop the Node2 and the ring will becomes $Node1 => Node4 => Node3 => Node1...$, and the Node3, the original Next node of Node4 is currently the successor of it.



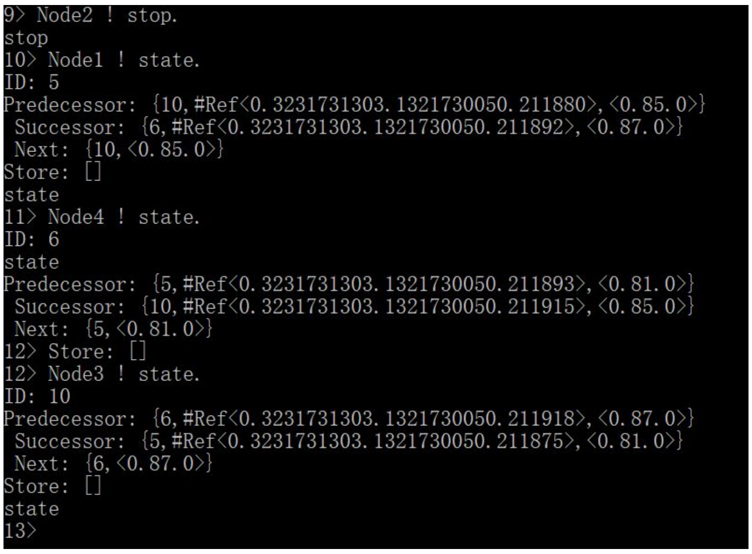
```
Store: [{990,533}]
ID: 991
Predecessor: {990,<0.1071.0>}, Successor: {992,<0.1073.0>}
Store: []
ID: 992
Predecessor: {991,<0.1072.0>}, Successor: {993,<0.1074.0>}
Store: [{992,218}]
ID: 993
Predecessor: {992,<0.1073.0>}, Successor: {994,<0.1075.0>}
Store: [{993,124}]
ID: 994
Predecessor: {993,<0.1074.0>}, Successor: {995,<0.1076.0>}
Store: [{994,913}]
ID: 995
Predecessor: {994,<0.1075.0>}, Successor: {996,<0.1077.0>}
Store: [{995,955},{995,420}]
ID: 996
Predecessor: {995,<0.1076.0>}, Successor: {997,<0.1078.0>}
Store: [{996,13},{996,716}]
ID: 997
Predecessor: {996,<0.1077.0>}, Successor: {998,<0.1079.0>}
Store: []
ID: 998
Predecessor: {997,<0.1078.0>}, Successor: {999,<0.1080.0>}
Store: [{998,646}]
ID: 999
Predecessor: {998,<0.1079.0>}, Successor: {0,<0.81.0>}
Store: []
ok
7>
```

Figure 2: Key value pairs in finger tables of 1000 nodes



```
1> Node1 = test:start(node3).
<0.81.0>
2> Node2 = test:start(node3, Node1).
<0.83.0>
3> Node3 = test:start(node3, Node1).
<0.85.0>
4> Node4 = test:start(node3, Node1).
<0.87.0>
5> Node1 ! state.
ID: 5
Predecessor: {10,#Ref<0.3231731303.1321730050.211880>,<0.85.0>}
 Successor: {6,#Ref<0.3231731303.1321730050.211892>,<0.87.0>}
 Next: {8,<0.83.0>}
Store: []
state
6> Node2 ! state.
ID: 8
Predecessor: {6,#Ref<0.3231731303.1321730050.211891>,<0.87.0>}
 Successor: {10,#Ref<0.3231731303.1321730050.211881>,<0.85.0>}
 Next: {5,<0.81.0>}
Store: []
state
7> Node3 ! state.
ID: 10
Predecessor: {8,#Ref<0.3231731303.1321730050.211882>,<0.83.0>}
 Successor: {5,#Ref<0.3231731303.1321730050.211875>,<0.81.0>}
 Next: {6,<0.87.0>}
Store: []
state
8> Node4 ! state.
ID: 6
Predecessor: {5,#Ref<0.3231731303.1321730050.211893>,<0.81.0>}
 Successor: {8,#Ref<0.3231731303.1321730050.211890>,<0.83.0>}
 Next: {10,<0.85.0>}
state
Store: []
```

Figure 3: start a ring



```
9> Node2 ! stop.
stop
10> Node1 ! state.
ID: 5
Predecessor: {10,#Ref<0.3231731303.1321730050.211880>,<0.85.0>}
 Successor: {6,#Ref<0.3231731303.1321730050.211892>,<0.87.0>}
 Next: {10,<0.85.0>}
Store: []
state
11> Node4 ! state.
ID: 6
state
Predecessor: {5,#Ref<0.3231731303.1321730050.211893>,<0.81.0>}
 Successor: {10,#Ref<0.3231731303.1321730050.211915>,<0.85.0>}
 Next: {5,<0.81.0>}
12> Store: []
12> Node3 ! state.
ID: 10
Predecessor: {6,#Ref<0.3231731303.1321730050.211918>,<0.87.0>}
 Successor: {5,#Ref<0.3231731303.1321730050.211875>,<0.81.0>}
 Next: {6,<0.87.0>}
Store: []
state
13>
```

Figure 4: stop a node