HW5: Chordy - A Distributed Hash Table

Jian Wen

October 9, 2023

1 Introduction

In this assignment, the main task is to build a distributed hash table with a ring structure. The nodes that connected to a ring all possess a range of storage key-value pairs. This range is given by the key between the node itself and its predecessor. To make this hash table work, we are required to implement the method for join, query and so on. The data storage can be updated through nodes as new node joins and get assigned a range from its successor. As for bonus tasks, we need to handle the possible node failure and storage replication.

2 Main problems and solutions

2.1 Building a chord structure

The construction of a ring includes two steps: starting the first node, and get the rest nodes joining one by one.

Start the first node: In our start() and init(), the predecessor is set to nil, and the successor is set to itself. After requesting status and receiving notify from itself during the stabilize procedure, the predecessor is updated to itself. By now, we have a ring including only one node.

Join new nodes: When a new node joins the ring, it will first set the chosen node as its (temperal) successor search for its predecessor and successor by sending request and gets status update. The new node will notify its successor and then the successor can update the predecessor; since we have a scheduler for auto update, the predecessor of new node can also update its successor. Up to now, the new node has joint our ring successfully and the status on all relevant nodes can be updated.

2.2 Key-value pair storage

In the module node2, we are required to add a pair storage for each node. Those nodes are assigned a key and responsible for the pairs with key between the range (Predecessor.Key, itself.Key]. For each node, if the request is related to the pair in its responsibility, it will process it; otherwise, it will just pass this to its successor.

```
add(Key, Value, Qref, Client, Id, {Pkey, _}, {_, Spid}, Store)->
  case key:between(Key, Pkey, Id) of
    true ->
      Client ! {Qref, ok},
      storage:add(Key, Value, Store);
    false ->
      Spid ! {add, Key, Value, Qref, Client},
      Store
  end.

lookup(Key, Qref, Client, Id, {Pkey, _}, {_, Spid}, Store)->
  case key:between(Key, Pkey, Id) of
    true->
      Result = storage:lookup(Key, Store),
      Client ! {Qref, Result};
    false ->
      Spid ! {lookup, Key, Qref, Client}
  end.
```

3 Bonus tasks: Handling failures & Replication

3.1 Failure Handling

Similar to what we did to the leader in the HW4 assignment, on each node, we setup monitors to track the DOWN message from its predecessor and successor. If we detect the predecessor has crushed, we only need to set the predecessor as nil and wait for the next request and status update; If we detect the successor has crushed, we will set our successor as the successor of our previous successor, and trigger stabilize. Therefore, the successor of successor should be recorded as part of node's parameters.

```
down(Ref, {_, Ref, _}, Successor, Next) ->
  {nil, Successor, Next};
down(Ref, Predecessor, {_, Ref, _}, {Nkey, Npid}) ->
  Nref = monitor(Npid),
  NewSuccessor = {Nkey, Nref, Npid},
  stabilize(NewSuccessor),
  {Predecessor, NewSuccessor, nil}.
```

3.2 Store Replication

In case of unexpected node failures, it is necessary to keep a replication of a node's pair storage. Since each node is responsible for the pairs between itself and its predecessor, we should have the replication stored at each node's successor - in this way, if its successor crushed, we can simply take over its storage by merge the replica with local store.

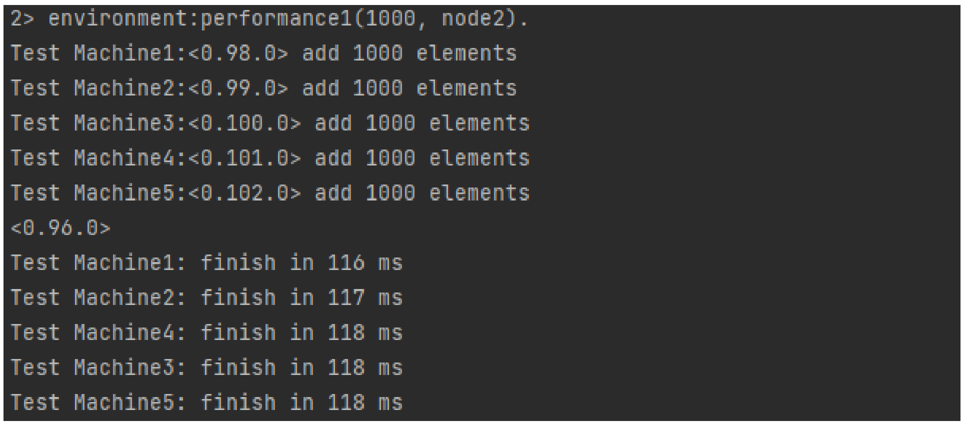
```
{add, Key, Value, Qref, Client} ->
      Added = add(Key, Value, Qref, Client, Id, Predecessor, Successor, Store),
      node(Id, Predecessor, Successor, Added, Next, Replica);
{replicate, Key, Value, Qref, Client} ->
      NewReplica = add_replica(Key, Value, Qref, Client, Id, Replica),
      node(Id, Predecessor, Successor, Store, Next, NewReplica);
...
add(Key, Value, Qref, Client, Id, {Pkey,_,_}, {_,_,Spid}, Store)->
      ...
      Spid ! {replicate, Key, Value, Qref, Client},
      ...
add_replica(Key, Value, Qref, Client, Id, Replica)->
  Added = storage:add(Key, Value, Replica),
  Client ! {Qref, Id},
  Added.
...
recoverStore(Store, Replica)->
  storage:merge(Replica, Store).
```

3.3 Evaluation

To make the test a bit simpler, I have setup a environment module to create several processes and build a chord hash table.

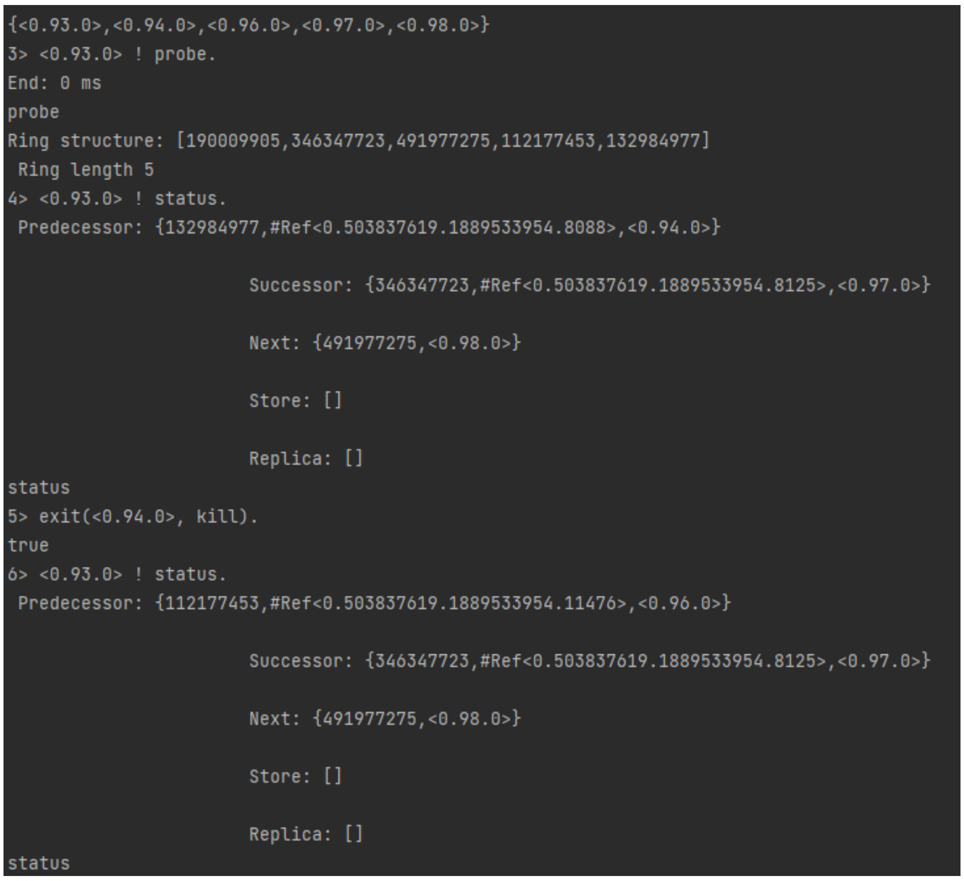
4 Conclusion

In conclusion, this assignment has provided me with hands-on experience in building a distributed hash table with a ring structure, that is chord. Actually, the chord structure is something that I have already theoretically learnt during a internetworking course. However, this assignment really give me a new understanding of such a structure. I successfully implemented key functionalities such as join, query, and handled node failures and storage replication. This exercise has deepened my understanding of distributed systems and prepared myself for future challenges in this field.



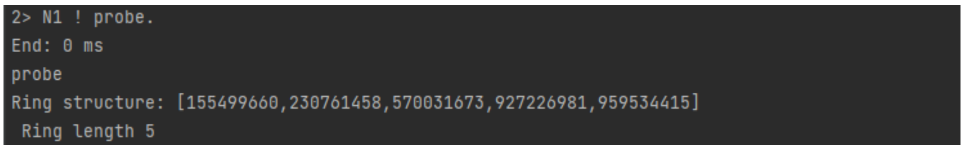
```
2> environment:performance1(1000, node2).
Test Machine1:<0.98.0> add 1000 elements
Test Machine2:<0.99.0> add 1000 elements
Test Machine3:<0.100.0> add 1000 elements
Test Machine4:<0.101.0> add 1000 elements
Test Machine5:<0.102.0> add 1000 elements
<0.96.0>
Test Machine1: finish in 116 ms
Test Machine2: finish in 117 ms
Test Machine4: finish in 118 ms
Test Machine3: finish in 118 ms
Test Machine5: finish in 118 ms
```

Figure 1: Testing pair storage in chord ring.



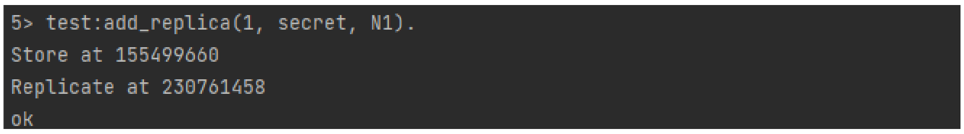
```
{<0.93.0>,<0.94.0>,<0.96.0>,<0.97.0>,<0.98.0>}
3> <0.93.0> ! probe.
End: 0 ms
probe
Ring structure: [190009905,346347723,491977275,112177453,132984977]
 Ring length 5
4> <0.93.0> ! status.
 Predecessor: {132984977,#Ref<0.503837619.1889533954.8088>,<0.94.0>}

                      Successor: {346347723,#Ref<0.503837619.1889533954.8125>,<0.97.0>}

                      Next: {491977275,<0.98.0>}

                      Store: []

                      Replica: []
status
5> exit(<0.94.0>, kill).
true
6> <0.93.0> ! status.
 Predecessor: {112177453,#Ref<0.503837619.1889533954.11476>,<0.96.0>}

                      Successor: {346347723,#Ref<0.503837619.1889533954.8125>,<0.97.0>}

                      Next: {491977275,<0.98.0>}

                      Store: []

                      Replica: []
status
```

Figure 2: Testing failure handling in node3.



```
2> N1 ! probe.
End: 0 ms
probe
Ring structure: [155499660,230761458,570031673,927226981,959534415]
 Ring length 5
```

Figure 3: Testing chord ring in node4.



```
5> test:add_replica(1, secret, N1).
Store at 155499660
Replicate at 230761458
ok
```

Figure 4: Testing replication in node4.