Report : Chordy - a distributed hash table

Zhanbo Cui

October 13, 2021

1 Introduction

In this homework, I implement a distributed hash table following the Chord scheme. The structure of Chordy is a ring, every node(server) has a successor and predecessor in the Chordy, each node has the responsibility to store some Key-Value pairs which key should from the number of node's predecessor to node's number. Furthermore, I implement handling failures and replication on the original Chordy.

2 Main problems and solutions

2.1 Building a ring

The construction of ring base on the message exchange between nodes. Nodes decide their predecessor and successor according to the message from other nodes such as *notify* and *status*.

1. Start with a node

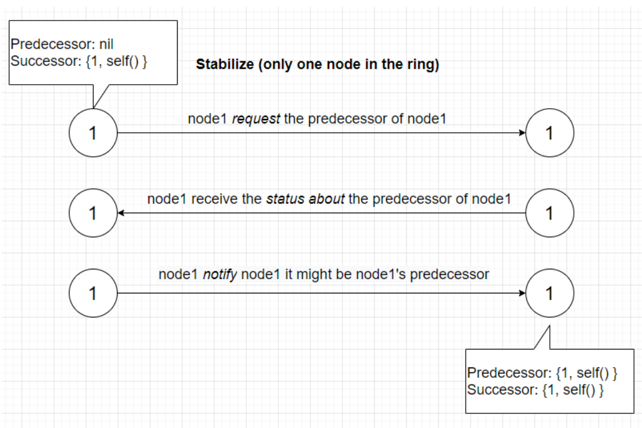


Figure 1: Initiation and Stabalize when only one node in the ring

2. New node join a ring

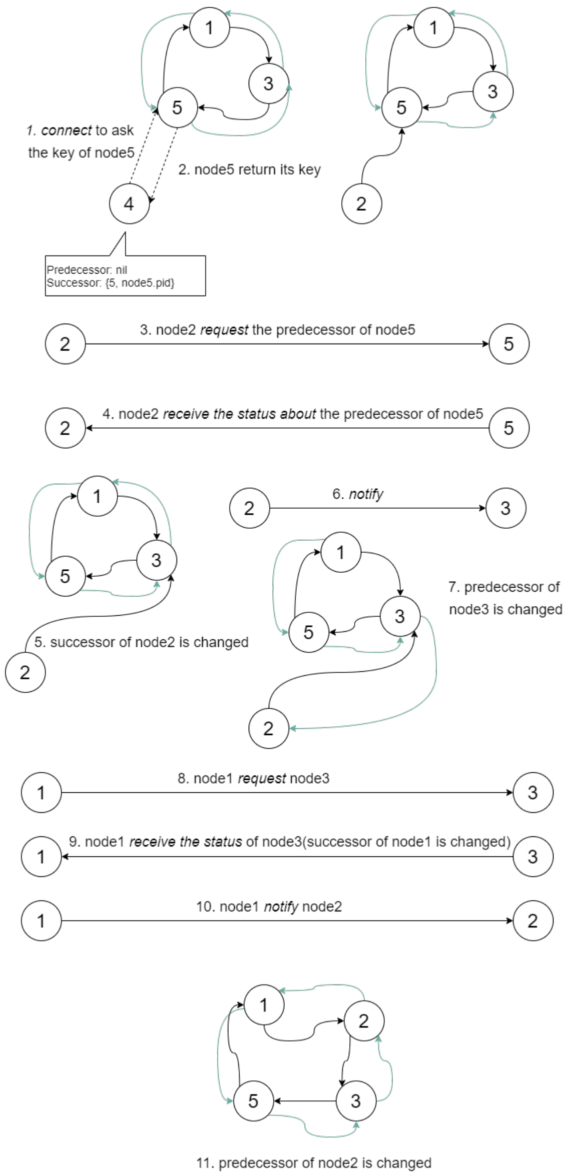


Figure 2: join a ring

When a new node want to join, it will first arbitrarily take the chosen node as its *successor*, And its correct *successor* will be obtained during the stabilize phase and wait for a potential *predecessor* to find it. The whole process is like a double linked lists being sorted. The most important point is that *stabilize* is done periodically

3 Store

Key-value pairs are stored as a list of tuples {Key, Value} in the nodes. Every node should be responsible for the Key-value pairs which key is belong to (Predecessor.Key , Node.Key]. The new Key-value pairs will pass along the ring until find the node it should stored in.

A new node should of course take over part of the responsibility and must then of course also take over already added elements. If receive *notify* from a new node and this proposal is approved, we should judge which key-value pairs to keep. The heart code as following:

```
handover(Id, Store, Replicate, Nkey, Npid)->
    %k-v in (Nkey, Id] is "keep"
    %k-v in (Pkey, Nkey] is "NotKeep"
    {KeepS, NotKeepS} = storage:split(Nkey, Id, Store),
    {KeepR, NotKeepR} = storage:split(Nkey, Id, Replicate),
    Npid ! {handover, {NotKeepS,NotKeepR}},
    {KeepS, KeepR}.
```

4 Handling failures

To handle failures we can use *monitor* to detect the predecessor and successor, meanwhile to keep tack of the successor of our successor. When predecessor and successor is terminated, we can receive the 'DAWN' message. According to the *Ref* in the 'DAWN' message, we can change our predecessor and successor as following code:

```
%match with Predecessor
down(Ref, {_, Ref, _}, Successor, Next) ->
    %Predecessor set to nil and wait
    %other to request and notify me!
    {nil, Successor, Next};
%match with Successor
down(Ref, Predecessor, {_, Ref, _}, {Nkey, Npid}) ->
    %Next become new Successor
    Nref = monitor(Npid),
    NewSuccessor = {Nkey,Nref,Npid},
    stabilize(NewSuccessor),
    {Predecessor, NewSuccessor, nil}.
```

5 Replication

Replication is a backup of *Store*, it should be kept by the successor of a node. So when a node want to store a key-value pair, it should also forward this Key-value pair to its successor as *Replication*.

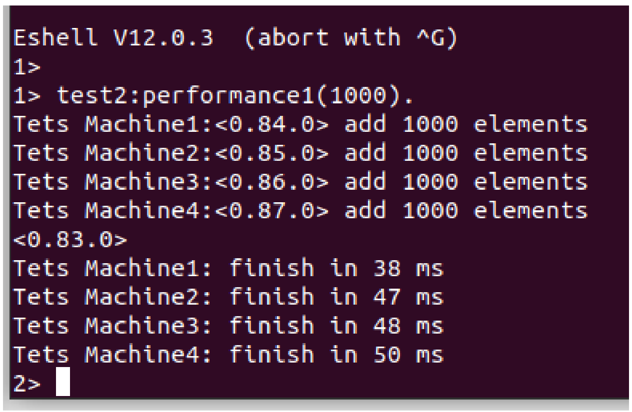
```
addS(...)->
    ...
    %add new k-v into store
    Added = storage:add(Key, Value, Store),
    %send replicate to our successor
    Spid ! {replicate, Key, Value, Qref, Client},
    ...
node(...)->
    ...
    %receive replicate from our predecessor
    {replicate, Key, Value, Qref, Client}->

    NewReplica = addR(Key, Value, Qref, Client, Id, Replica),
    node(Id, Predecessor, Successor, Store, Next, NewReplica);
    ...
```

As same as *Store*, a new node of course also take over part of already added *Replication*.

6 Evaluation

1. One node in the ring and let the four test machines add 1000 elements to the ring and then do a lookup of the elements.



```
Eshell V12.0.3  (abort with ^G)
1>
1> test2:performance1(1000).
Tets Machine1:<0.84.0> add 1000 elements
Tets Machine2:<0.85.0> add 1000 elements
Tets Machine3:<0.86.0> add 1000 elements
Tets Machine4:<0.87.0> add 1000 elements
<0.83.0>
Tets Machine1: finish in 38 ms
Tets Machine2: finish in 47 ms
Tets Machine3: finish in 48 ms
Tets Machine4: finish in 50 ms
2>
```

Figure 3: one node for 1000 elements from four test machines

2. One node to handle 4000 elements

```
2> test2:performance2(4000).
Tets Machine4000:<0.90.0> add 4000 elements
<0.89.0>
Tets Machine4000: finish in 45 ms
3>
```

Figure 4:

The time spent is almost the same, I think the limiting factor is the time cost of a node to handle each request.

3. Now let's do some experiments with the node4 module

```
3> N1 = node4:start(1).
<0.92.0>
4> N2 = node4:start(3,N1).
<0.94.0>
5> N3 = node4:start(5,N1).
<0.96.0>
6> N4 = node4:start(7,N1).
<0.98.0>
7> N1 ! probe.
End: 0 ms
probe
Ring structure: [1,3,5,7]
 Ring length 4
8>
```

Figure 5: Create a ring

```
8>
8> test:addR(4,zhanbo,N1).
Store at 5
Replicate at 7
ok
9>
```

Figure 6: Add key-value pairs and check replication

```
9> N3 ! status.
 Predecessor: {3,#Ref<0.3790484805.693633025.216956>,<0.94.0>}

                         Successor: {7,#Ref<0.3790484805.693633025.216966>,<0.98.0>}

                         Next: {1,<0.92.0>}

                         Store: [{4,zhanbo}]

                         Replica: []
status
```

Figure 7: Check node3(key: 5)

```
10> N4 ! Status.
* 1:6: variable 'Status' is unbound
11> N4 ! status.
 Predecessor: {5,#Ref<0.3790484805.693633025.216967>,<0.96.0>}

                         Successor: {1,#Ref<0.3790484805.693633025.216959>,<0.92.0>}

                         Next: {3,<0.94.0>}

                         Store: []

                         Replica: [{4,zhanbo}]
status
```

Figure 8: Check node4(key: 7)

```
12> exit(N3,"see u").
true
13> N1 ! probe.
End: 0 ms
probe
Ring structure: [1,3,7]
 Ring length 3
14>
```

Figure 9: Kill node3(key: 5)

```
15> test:lookup(4,N1).
{4,zhanbo}
16> N4 ! status.
 Predecessor: {3,#Ref<0.3790484805.693633025.216996>,<0.94.0>}

                         Successor: {1,#Ref<0.3790484805.693633025.216959>,<0.92.0>}

                         Next: {3,<0.94.0>}

                         Store: [{4,zhanbo}]

                         Replica: []
status
```

Figure 10: We can also lookup stored by node3

7 Conclusion

I have implemented a distributed hash table and implemented with handling failures and replication. Chord is a classical algorithms in P2P network.

This assignment also made me realize that there are a lot of details we need to work on to design a "perfect" distributed system!