COP5615 Project 3 Chord Protocol

Teammember

Yi-Ming Chang (UFID 83816537)

What is Working?

All Functions Below:

Function	Paper Pseudocode Function
Chord Ring create	n.create()
Node Join	n.join(n’)
Node Stabilize periodically	n.stabilize()
Node Notify successor	n.notify(n’)
Each server update Finger table periodically	n.fix_fingers()
Check predecessor periodically	n.check_predecessor()

We use a 3 server with 8 identifier as an example (M=3, NumberOfNodes = 3)

```
[INFO][10/28/2021 6:24:01 PM][Thread 0001][remoting (akka://proj3Master)] Starting remoting
[INFO][10/28/2021 6:24:01 PM][Thread 0001][remoting (akka://proj3Master)] Remoting started; listening on address
[INFO][10/28/2021 6:24:01 PM][Thread 0001][remoting (akka://proj3Master)] Remoting now listens on addresses: [ak
input arguments:
[|"proj3.fsx"; "3"; "5"|]
system systemParams:
{ NumOfNodes = 3
  NumOfRequest = 5
  PowM = 3
  NumOfIdentifier = 8 }
```

Check server#1 (ChordID 1)

```
SererNum: 1, chordId 1, successor 6, fingerTable:
[|{ Idx = 0
    KeyId = 2
    Succesor = 1 }; { Idx = 1
                      KeyId = 3
                      Succesor = 1 }; { Idx = 2
                                        KeyId = 5
                                        Succesor = 1 }|]
```

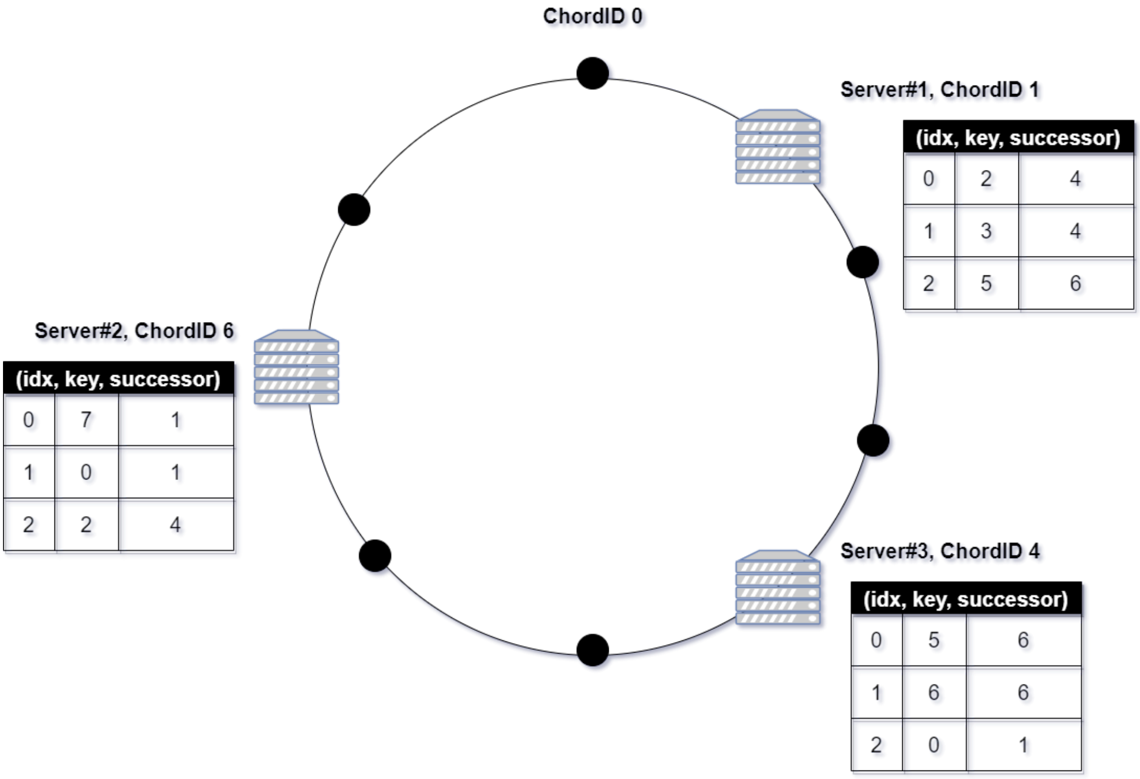
Next, when server#2 (ChordID 6) Joins, server#1 finger table updates

```
SererNum: 1, chordId 1, successor 6, fingerTable:
[|{ Idx = 0
    KeyId = 2
    Succesor = 6 }; { Idx = 1
                      KeyId = 3
                      Succesor = 6 }; { Idx = 2
                                        KeyId = 5
                                        Succesor = 6 }|]
```

Lastly, when server #3 (ChordID 4) Joins, server #1 finger table updates
And this include the function of
Node_Join, Fix_fingler, Node_notify and Stabilize

```
SererNum: 1, chordId 1, successor 4, fingerTable:
[|{ Idx = 0
    KeyId = 2
    Succesor = 4 }; { Idx = 1
                      KeyId = 3
                      Succesor = 4 }; { Idx = 2
                                        KeyId = 5
                                        Succesor = 6 }|]
```

The result of each server node will be like the figure show as below:



Requesting Key Value:

This part is to simulate to get the key-value (correct data) on a specific server request. We can imagine that one client links to the server and sends a request for the content.

```
START from (ServerNum: 3, chordID 4), GET key 6 stored on Server with chordID: 6
START from (ServerNum: 3, chordID 4), GET key 5 stored on Server with chordID: 6
START from (ServerNum: 3, chordID 4), GET key 7 stored on Server with chordID: 1
START from (ServerNum: 3, chordID 4), GET key 3 stored on Server with chordID: 4
START from (ServerNum: 3, chordID 4), GET key 3 stored on Server with chordID: 4
```

BONUS

We have made the **n.check_predecessor** to update the status of the predecessor if failure. The design is to send a message to an actor with the sleep time duration and can temporarily or permanently make the actor dead. (Which we build a failure node set and set a failure condition, for instance, when the requesting time reached 30% input, the node in the set will shut down temporarily)

The figure below shows that ServerNum 1 with chordID 1 is dead, for some condition. And the ServerNum3 with c0hordID 4 will detect the change and after the ServerNum 1 joins back it will update again as its predecessor.

```
$ dotnet fsi proj3.fsx 3 10
Real: 00:00:00.000, CPU: 00:00:00.000, GC gen0: 0, gen1: 0, gen2: 0
Real: 00:00:00.000, CPU: 00:00:00.000, GC gen0: 0, gen1: 0, gen2: 0
Real: 00:00:00.000, CPU: 00:00:00.000, GC gen0: 0, gen1: 0, gen2: 0
[INFO][10/30/2021 4:41:40 AM][Thread 0001][remoting (akka://proj3Master)] Starting remoting
[INFO][10/30/2021 4:41:40 AM][Thread 0001][remoting (akka://proj3Master)] Remoting started; listening on addresses : [akka.tcp://proj3Mast
[INFO][10/30/2021 4:41:40 AM][Thread 0001][remoting (akka://proj3Master)] Remoting now listens on addresses: [akka.tcp://proj3Master@loca
input arguments:
[|"proj3.fsx"; "3"; "10"|]
system systemParams:
{ NumOfNodes = 3
  NumOfRequest = 10
  PowM = 3
  NumOfIdentifier = 8 }
SERVER [ServerNum: 1, chordID 1] Receive PING FROM 4, isAlive true
SERVER [ServerNum: 3, chordID 4] predecessor 1 check resp true
SERVER [ServerNum: 1, chordID 1] Receive PING FROM 4, isAlive true
SERVER [ServerNum: 3, chordID 4] predecessor 1 check resp true
SERVER Dead [ServerNum: 1, chordID 1]
SERVER [ServerNum: 3, chordID 4] predecessor 1 is unreachable, predecessor = NULL
SERVER [ServerNum: 1, chordID 1] Receive PING FROM 4, isAlive true
[INFO][10/30/2021 4:42:05 AM][Thread 0011][akka://proj3Master/deadLetters] Message [Boolean] from [akka://proj3Master/temp/m] to [akka://p
not delivered. [1] dead letters encountered. If this is not an expected behavior then [akka://proj3Master/deadLetters] may have terminatec
can be turned off or adjusted with configuration settings 'akka.log-dead-letters' and 'akka.log-dead-letters-during-shutdown'.
SERVER [ServerNum: 1, chordID 1] Receive PING FROM 4, isAlive true
SERVER [ServerNum: 3, chordID 4] predecessor 1 check resp true
SERVER [ServerNum: 1, chordID 1] Receive PING FROM 4, isAlive true
SERVER [ServerNum: 3, chordID 4] predecessor 1 check resp true

Real: 00:00:30.511, CPU: 00:01:25.359, GC gen0: 4928, gen1: 4, gen2: 0
```

What is the largest network you managed to deal with

500 Node Server, 16384 identifier (2^14, m=14)
Each server sends 10 requests for data.
Down below is the last attendance (ChordID: 14983) requesting for keys.
The console will show

1. Request Key A from Server X
2. Start from server X, Get key A stored on Server Y (chordID)
3. Finger Table of the server

```
$ dotnet fsi proj3.fsx 500 10
Real: 00:00:00.000, CPU: 00:00:00.000, GC gen0: 0, gen1: 0, gen2: 0
Real: 00:00:00.000, CPU: 00:00:00.000, GC gen0: 0, gen1: 0, gen2: 0
Real: 00:00:00.000, CPU: 00:00:00.000, GC gen0: 0, gen1: 0, gen2: 0
[INFO][10/29/2021 4:06:27 AM][Thread 0001][remoting (akka://proj3Master)] Starting remoting
[INFO][10/29/2021 4:06:27 AM][Thread 0001][remoting (akka://proj3Master)] Remoting started; listening on addresses : [akka.tcp://proj3Master@localhost:5567]
[INFO][10/29/2021 4:06:27 AM][Thread 0001][remoting (akka://proj3Master)] Remoting now listens on addresses: [akka.tcp://proj3Master@localhost:5567]
input arguments:
[|"proj3.fsx"; "500"; "10"|]
system systemParams:
{ NumOfNodes = 500
  NumOfRequest = 10
  PowM = 14
  NumOfIdentifier = 16384 }
Same ChordID Exception! "Actor name "9790" is not unique!"
Same ChordID Exception! "Actor name "1899" is not unique!"
Same ChordID Exception! "Actor name "11386" is not unique!"
Same ChordID Exception! "Actor name "15860" is not unique!"
Same ChordID Exception! "Actor name "13115" is not unique!"
Request key 4565 from (ServerNum: 500, chordID 14983)
Request key 386 from (ServerNum: 500, chordID 14983)
Request key 10543 from (ServerNum: 500, chordID 14983)
START from (ServerNum: 500, chordID 14983), GET key 4565 stored on Server with chordID: 7526
Request key 1039 from (ServerNum: 500, chordID 14983)
START from (ServerNum: 500, chordID 14983), GET key 386 stored on Server with chordID: 7526
Request key 1968 from (ServerNum: 500, chordID 14983)
START from (ServerNum: 500, chordID 14983), GET key 10543 stored on Server with chordID: 11386
Request key 3343 from (ServerNum: 500, chordID 14983)
START from (ServerNum: 500, chordID 14983), GET key 1039 stored on Server with chordID: 7526
Request key 9277 from (ServerNum: 500, chordID 14983)
START from (ServerNum: 500, chordID 14983), GET key 1968 stored on Server with chordID: 7526
Request key 741 from (ServerNum: 500, chordID 14983)
START from (ServerNum: 500, chordID 14983), GET key 3343 stored on Server with chordID: 7526
Request key 9237 from (ServerNum: 500, chordID 14983)
START from (ServerNum: 500, chordID 14983), GET key 9277 stored on Server with chordID: 9495
Request key 14517 from (ServerNum: 500, chordID 14983)
START from (ServerNum: 500, chordID 14983), GET key 741 stored on Server with chordID: 7526
START from (ServerNum: 500, chordID 14983), GET key 9237 stored on Server with chordID: 9271
START from (ServerNum: 500, chordID 14983), GET key 14517 stored on Server with chordID: 14536
END SERVER (ServerNum: 500, chordID 14983), Finger Table:
[|{ Idx = 0
    KeyId = 14984
    Succesor = 7526 }; { Idx = 1
                         KeyId = 14985
                         Succesor = 7526 }; { Idx = 2
                                              KeyId = 14987
                                              Succesor = 7526 };
  { Idx = 3
    KeyId = 14991
    Succesor = 7526 }; { Idx = 4
                         KeyId = 14999
                         Succesor = 7526 }; { Idx = 5
                                              KeyId = 15015
                                              Succesor = 7526 };
  { Idx = 6
    KeyId = 15047
    Succesor = 7526 }; { Idx = 7
                         KeyId = 15111
                         Succesor = 7526 }; { Idx = 8
                                              KeyId = 15239
                                              Succesor = 7526 };
  { Idx = 9
    KeyId = 15495
    Succesor = 7526 }; { Idx = 10
                         KeyId = 16007
                         Succesor = 7526 }; { Idx = 11
                                              KeyId = 647
                                              Succesor = 7526 };
  { Idx = 12
    KeyId = 2695
    Succesor = 7526 }; { Idx = 13
                         KeyId = 6791
```

Appendix

Project Requirements

Input: The input provided (as command line to yourproject3.scala) will be of the form:

project3 numNodes numRequests

Where numNodes N is the number of peers to be created in the peer-to-peer system and numRequests K is the number of requests each peer has to make. When all peers perform that many requests, the program can exit. Each peer should send a request/second.

Scenario

We consider this peer-to-peer system as a distributed cache system. First, we are going to assign M key values to the system to store these data. Then each node will be seen as a server that will arbitrarily request for K key-value. To match the request we will need a hashtable to **lookup** the object location. In the real world, this structure can be implemented as an API server + Redis cache to form a distributed cache system. Or else, we can see the value as any type of data, for instance as a piece of the file, then this will be a distributed file system.

Consistent Hashing

The first challenge is to assign the key values to the nodes. Intuitively, we can simply use the key as a hash-function input and module N (the number of servers). However, if one of the servers crashes, the event will trigger a severe rehashing, since we module N in the arrangement.

So, We need a distribution scheme that does not depend directly on the number of servers, so that, when adding or removing servers, the number of keys that need to be relocated is minimized - Consistent Hashing

Consistent hashing evenly distributes K objects across N bins as K/N for each. Thus, when N changes not all objects need to be moved.