# COP5615 – Fall 2018

# Project 3

## Group Members

**Prafful Mehrotra UFID: 1099-5311**

**Siddhant Mohan Mittal UFID: 6061-8545**

## Questions asked in Problem Statement

**What is working?**

- We have implemented all the desired modules as mentioned in Sec. 4 of the Chord research paper.
- Implemented the chord ring (using join functionality) and have performed lookups for the random strings decoded to keys identifiers in intervals of 1 millisecond.
- For the bonus part, we have implemented a failure model to demonstrate simultaneous node failures and lookups.

**What is the largest network you managed to deal with?**

- numNodes = 10_000
- numRequest = any reasonable value

## Project Directory Structure

```
├── chord
│   ├── _build/
│   ├── config/
│   ├── lib
│   │   └── chord
│   │       ├── application.ex
│   │       ├── driver.ex
│   │       ├── node.ex
│   │       ├── node_supervisor.ex
│   │       └── stabilize.ex
│   ├── main.exs
│   ├── mix.exs
│   └── test/
├── chord-bonus
│   ├── _build/
│   ├── config/
│   ├── lib
│   │   └── chord
│   │       ├── application.ex
│   │       ├── driver.ex
│   │       ├── node.ex
│   │       ├── node_supervisor.ex
│   │       └── stabilize.ex
│   ├── main.exs
│   ├── mix.exs
│   └── test/
├── images
│   └── Average Hops vs Numnodes.png
├── Proj3.pdf
├── Project 3 Bonus Report.docx
├── Project 3 Bonus Report.pdf
└── README.md
```

## Defining Application architecture

- There is a main supervisor which supervises 2 modules: Driver and Stabilize and a subsupervisor: Node_Supervisor. The strategy used is one_for_all because we want the application (all modules) to restart if any of the modules fail to function.
- Node_Supervisor supervises numNodes number of Node modules. Strategy used here is one_for_one because we want only the node terminated unintentionally to respawn.

## Instructions for running the code

After unziping the file

```
$ cd chord
$ mix run --no-halt main.exs 100 5
```

For bonus part, after unziping the file

```
$ cd chord-bonus
$ mix run --no-halt main.exs 100 5 20
```

## Input Format:

```
/chord$ mix run --no-halt main.exs arg1 arg2
```

```
/chord-bonus$ mix run --no-halt main.exs arg1 arg2 arg3
```

| Argument | Value | Possible Values |
|---|---|---|
| arg1 | numNodes | any positive integer |
| arg2 | numRequests | any positive integer |
| arg3 | percentage-failure | any value between 0 and 100 |

## Output Format:

Example output for chord:

```
["avg number of hops = ", 3.642]
["log2(100) = ", 6.643856189774724]
["log10(100) = ", 2.0]
```

Average number of hops.

Log of numNodes base 2

Log of numNodes base 10

Example output for chord-bonus:

```
["avg number of hops = ", 3.912]
["log2(100) = ", 6.643856189774724]
["log10(100) = ", 2.0]
After Failure
["avg number of hops = ", 3.254]
["log2(80) = ", 6.321928094887363]
["log10(80) = ", 1.9030899869919435]
```

Average number of hops

Log of numNodes base 2

Log of numNodes base 10

After Failure

Average number of hops after deletion

Log of {numNodes after deletion} base 2

Log of (numNodes after deletion} base 10

# Implementation:

- We are using a supervisor which supervise a node supervisor and 3 children genServers; driver, stabilize and fix_finger.
- Node supervisor, supervise genServers of individual nodes.
- Individual nodes maintain predecessor and finger table(1st entry is successor) as its state
- We have implemented the one-by-one node joining strategy, in which any node can join the chord ring at any time and leave at any time.
- Details of individual modules can be found in next section
- Stablize and fix_finger processes run concurrently to lookups. They keep running at a periodic interval in the background
- Whenever a node N1 joins, it asks a node N2 for its place in on the ring. N2 runs find_successor module to find the successor S1 of N1 in the chord ring. N1 calls notify module to let its successor S1 know about its presence and this is how the predecessor and successor states of node N1 and S1 gets updated.
- The concurrent modules of stabilize and fix_finger runs periodically on all nodes. They update the predecessor and finger stable states of all the nodes respectively.
- Lookup of any key on any node uses the finger table to optimize the run time complexity of the protocol. They use the module find*successor and find_closest* preceding modules to perform lookups

# Modules:

## join(key)

- Whenever a node joins the chord ring this function is called
- If length of chord is 0 then it calls the create() function to create a chord ring of single node having predecessor nil and successor itself.

## find_successor(key)

- Key can be a node identifier or a key identifier
- If the value of key lies between the current node or its successor, it returns the successor of the node
- Otherwise it calls the find*closest* preceding module

## find*closest* preceding (key)

- This module goes over the finger table to find the closest node preceding the key and returns its value

## stabilize()

- This module runs over every node periodically and updates its predecessors if they got changes due to the presence of new nodes
- This module runs in parallel to the join and lookup processes.

## fix_finger()

- Just like stabilize() this process also runs in concurrency and is performed periodically over every node
- It updates the m size finger table of very node. That is for any ith entry of m size finger table we call the find_successor(n + 2^(i-1)) on the node n itself

## notify()

- Whenever a new node is joined in the system it calls notify module to tell its successor of its presence and asks the successor to update its predecessor

# Stimulation setup:

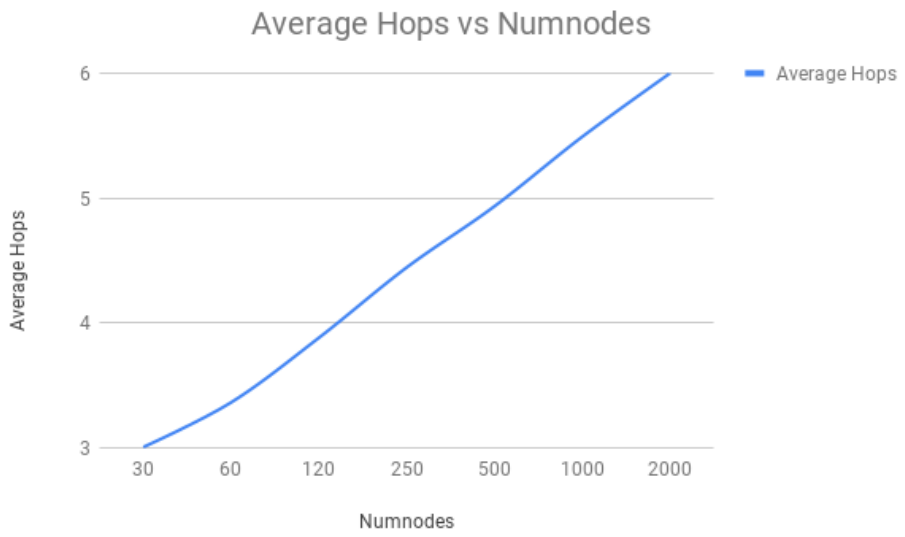
- Since we are using a function programming language Elixir for chord protocol stimulation, we have implemented chord protocol in a recursive style.
- In the recursive style, each intermediate node forwards a request to the next node until it reaches the successor
- Since in our implementation a chord is generated dynamically by join and departure of nodes and stabilize and fix_finger updating and making the chord ring stable, lookup can be performed concurrently. Though it might give a slow key lookup but it always will be correct
- Due to the fact that our main objective is to find the average hop time that is we are interested in time complexity of the chord protocol, we wait for the chord ring to get stable before starting the lookup.
- The power of stabilize and fix finger process will be highlighted in the bonus part where we handle the node failure and departure situation
- Chord's performance depends in part on the number of nodes that must be visited to resolve a query. Which is O(logN) that chord protocol states with very high probability
- To understand Chord's routing performance in practice, we simulated a network with N = 2^k nodes. We varied k from 5 to 11 and conducted a separate experiment for each value. Each node in an experiment picked a random set of keys to query from the system, and we measured each query's path length. This is done numRequests times and the average of all such nodes is calculate to find average hop of a experiment

## Hallmarks of our implementation:

- No node is special and its departure wont disrupt the chord ring.
- No node contains information about all the other nodes of the ring. Every node contains some keys and a finger table of size m
- Only driver module initiates join and lookup, then nodes communicate between themselves to perform specified task
- Stabilize and fix_finger are continuously running in the background. Hence the chord ring is mostly in stable state

## Observation and result

- Below graphs shows the average time for different numbers of numNodes.




- We observed from our result that average hop time is approximately equal to $(1/2)\log_2(10)$.
- The value of the constant term (1/2) can be understood as follows. Consider a node making a query for a randomly chosen key. Represent the distance in identifier space between node and key in binary. The most significant (say ith) bit of this distance can be corrected to 0 by following the node's ith finger. If the next significant bit of the distance is 1, it too needs to be corrected by following a finger, but if it is 0, then no i − 1st finger is followed—instead, we move on the the i−2nd bit. In general, the number of fingers we need to follow will be the number of ones in the binary representation of the distance from node to query. Since the node identifiers are randomly distributed, we expect half of the bits to be ones. After the logN most-significant bits have been fixed, in expectation there is only one node remaining between the current position and the key. Thus the average path length will be about $(1/2)\log_2(10)$.