Chord: Scalable P2P Lookup Service

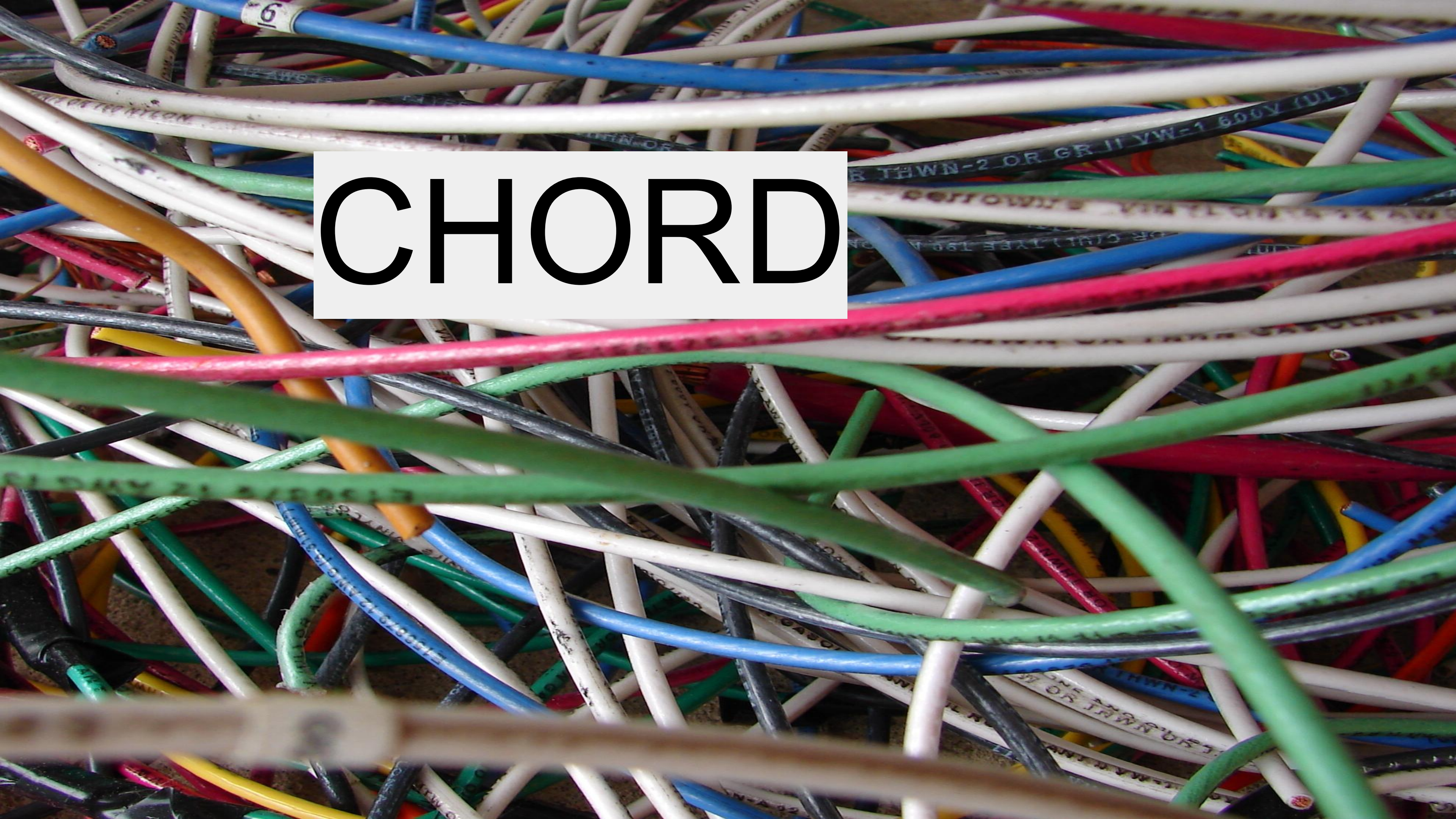
… for the internet (duh)

Hi

John-Alan Simmons
@iamjsimnz

- CTO at ConferenceCloud
- “Champion Gopher” - OSRC
- Ramblings of a crazy man - blog.jsimnz.io
- This talk a result of HERMES

CHORD



Distributed Systems

- mesos
- CoreOS
- Raft
- Etcd
- Riak
- wrong

P2P Systems

- Been around for a long time
- not as “new” or “sexy” as other types of distributed systems
- Leader less
- Eveyone is the same
- Karl Marx

but...but….

- Still powers very cool and interesting technologies
- Bitcoin / alt coins / Bitcoin 2.0
- Torrents
- Ditributed File Systems
    - GlusterFS
- No SPF

Lets break it down

- Chord is a DHT (kinda) => **Distributed** Hash Table
- Hash Table Ex
    - Java -> Hash Maps / Hash Table
    - Python -> Dictionaries
    - Go -> Maps
    - Lookup Tables
- Key => Values

Examples of (D)HT

- DNS
- GlusterFS
- K/V stores
- FreeNet

Why should we care?

- Chord Is:
  - Simple Functionality
    - Maps Keys => Nodes
    - Routes Messages to Nodes
  - Simple Protocol
    - Provably correct
  - Great performance
  - Correct & Performant under high Churn
    - Degrades effeciently
  - Minimal information required by each node

KV(N) All the things!

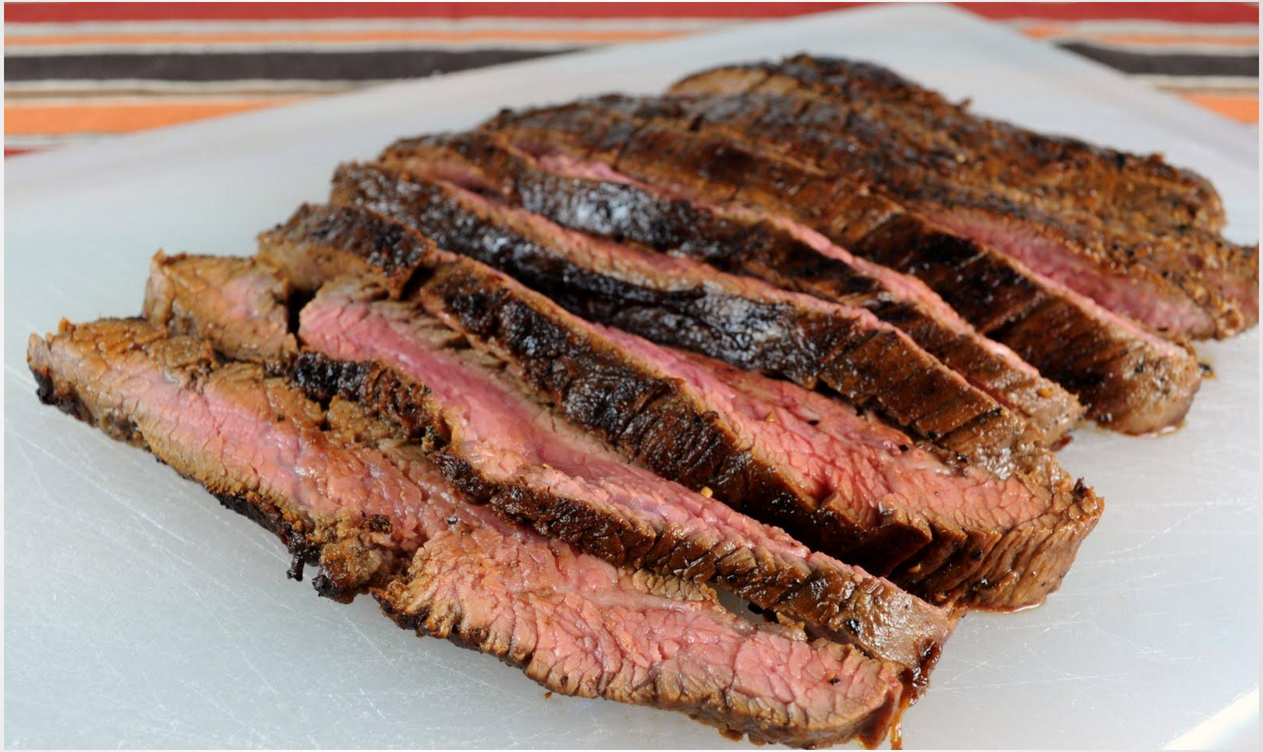
- HashMaps, Dictionaries, Maps, etc…
  - Given K, return V
- Chord
  - Given K
    - A) return node responsible for K in the DHT
    - B) Route a message to a Node
  - No Naming structure for Keys

Design Goals

- A p2p system that is:
  - Load Balanced
  - Decentralized
  - Scalable
  - Availability

Harness the Power

- Distributed Storage System
  - Keys: File name / data
  - Vals: node responsible for the file
- Large-Scale Combinatorial Search...yeah
  - Keys: Candidate Soln
  - vals: Machines responsible for testing said soln
- Distributed IMDB
  - Ex
    - Key: “Greatest movie of all time?”
    - Val: Node responsible for **The Matrix**...shhh...don’t argue



Bro, do you even Map?

- Namesake
- distributed HASH table
- Can anyone Guess….

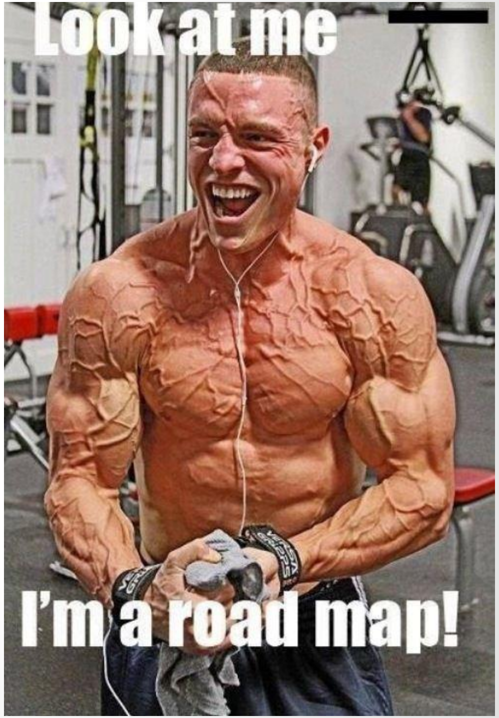
  …

  …...

  ...

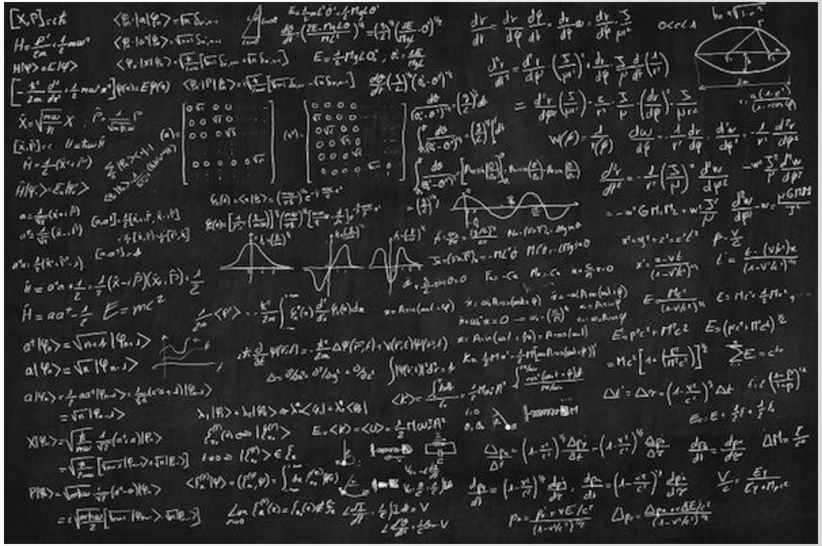
  (jeorpardy theme song)

  ….
- Hash Functions

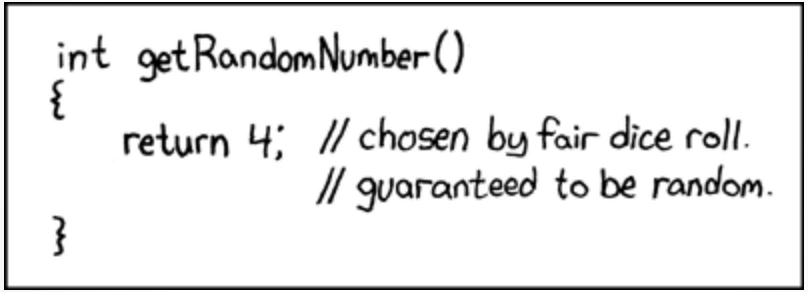


Hashing

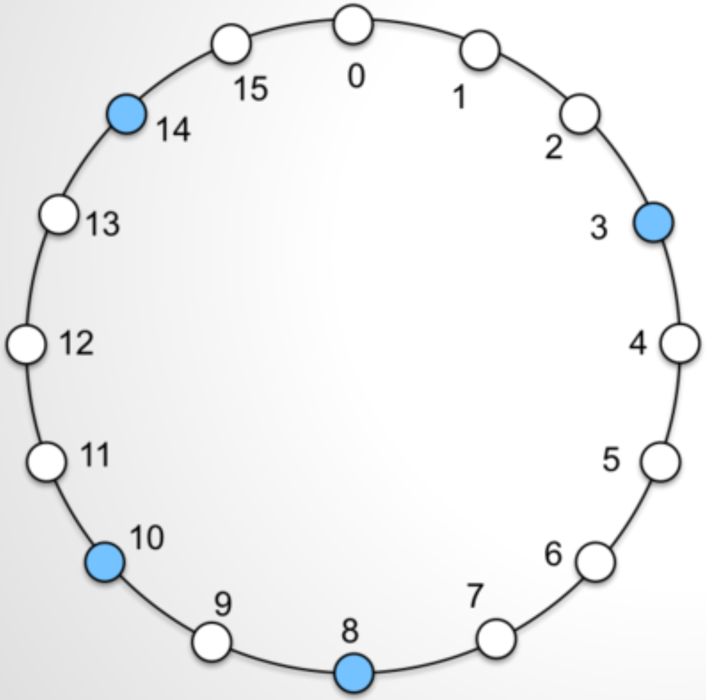
- Some Hash function *fn*
- *fn(Key) = Node Identifier*
- Example of hash functions
  - MD5 (<-- you're all using this right)
  - SHA
  - Keccack
- Size of Hash function?
  - size of SHA?
    - 3? .. number of characters?
    - Solution Space



Consistent Hashing



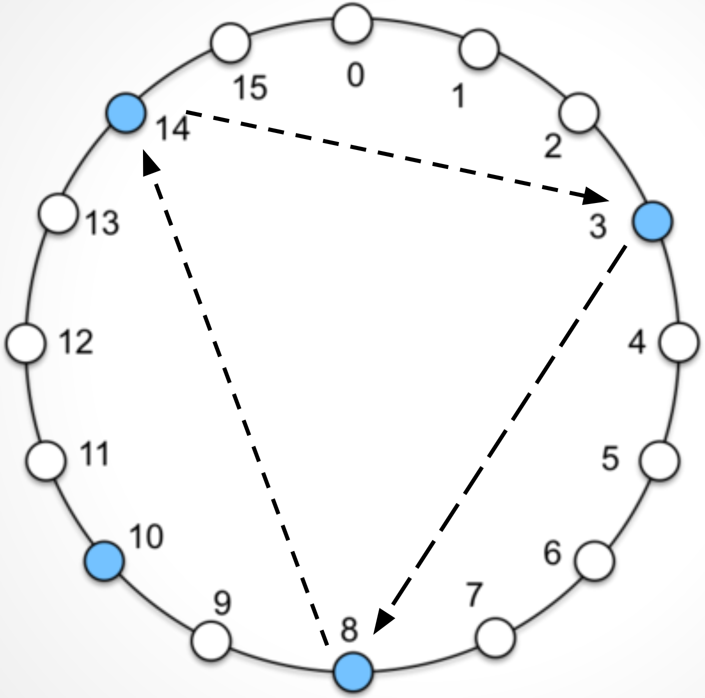
Hash Ring and Node IDs



- Successor of *K* is the first node that is equal to or succeeds *K*
- *successor(K)*
- Ex
  - *successor(2) = ?*
  - *successor(8) = ?*
  - *successor(15) = ?*

Great, now what

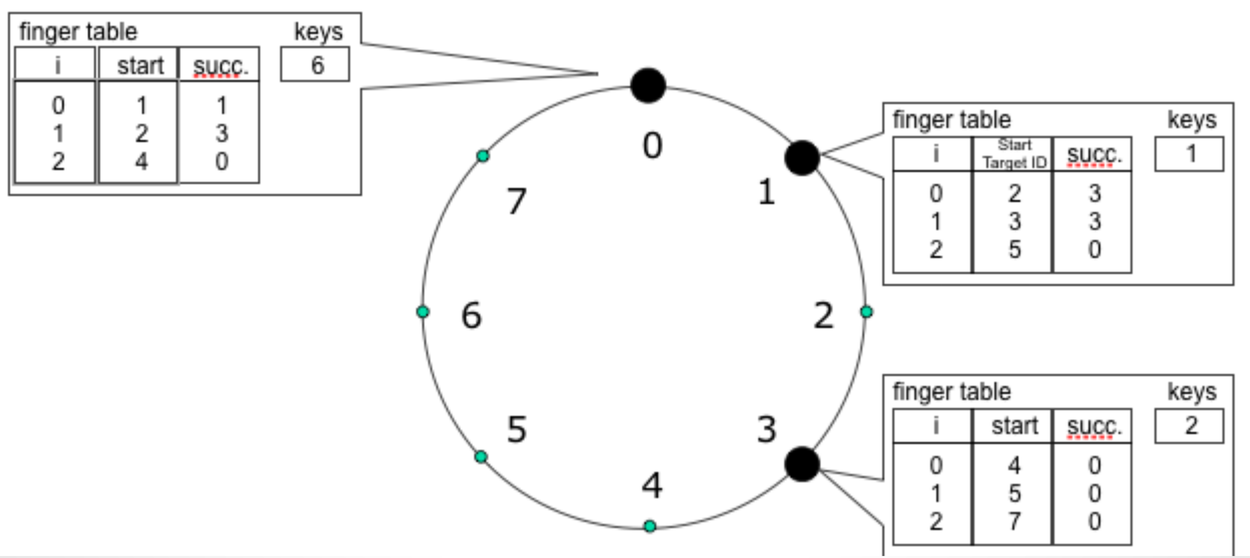
- Chord: Hash(Key) -> ID of *successor*(Key)
- Simplistic Protocol
  - Each node *n* maintains a pointer to its *successor, n.successor*
  - Continuously pass some message from *successor* to *successor* untill we reach *successor(K)*
    - We now have the node responsible for **The Matrix**
  - Guaranteed correctness
  - Inefficient O(N)


0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

That's all Folks!

Finger Tables

- If we maintain a some information about a small subset of nodes…
  - Increased performance
  - Provably correct
- Specifically maintain info on *m = size of Hash fn*, nodes
- For some node n, the i^{th} node n maintains…
  - first node that succeeds n by 2^i-1 (mod 2^m)
  - *successor*$(n + 2^{i+1})$
  - called the i^{th} finer of n => n.finger[i].node
  - Note: 1^{st} finger of *n* == *n.successor*



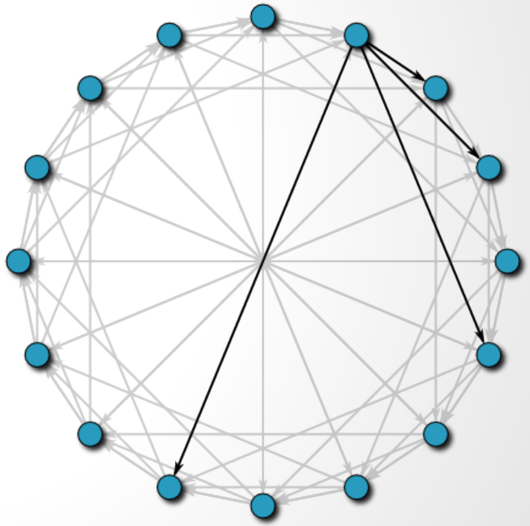
finger table
keys
i
start
succ.
0 1 1
1 2 3
2 4 0
6
0
1
2
3
4
5
6
7
finger table
keys
i
Start Target ID
succ.
0 2 3
1 3 3
2 5 0
1
finger table
keys
i
start
succ.
0 4 0
1 5 0
2 7 0
2

Finger Table Properties

- Maintains information of few nodes, *m*

- More local nodes the further ones

- Most likely won’t have some arbitrary key in the table



Routing with Finger Tables

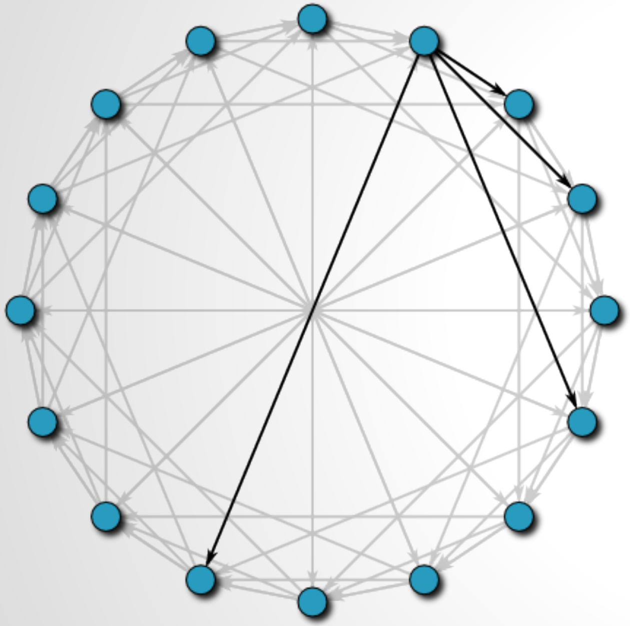
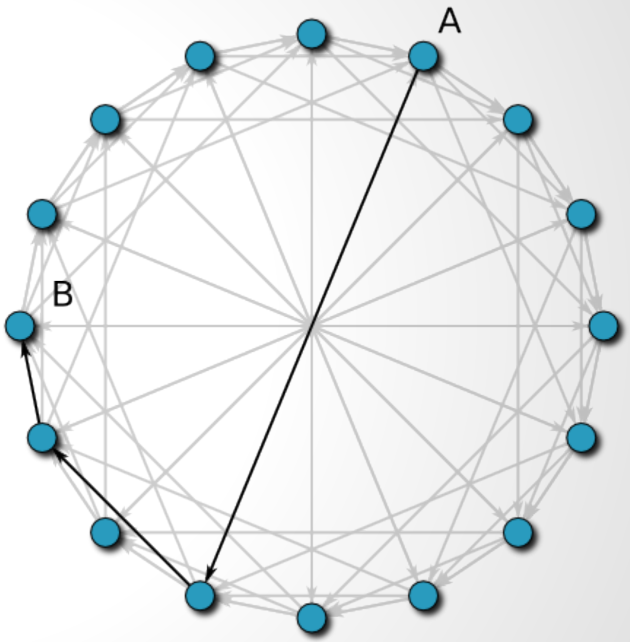
- Process of routing some key *K*
- Case I:
  - *K* is in our finger table, life is easy, and we deliver the message
- Case II:
  - *K* isn't in our finger table
    - Search our finger table for the largest key that preceeds *K*
    - Forward to that node, and repeat untill finished

Lets Get pseudo

```
n.find_successor(id):
    n' = find_predecessor(id)
    return n'.successor

n.find_predecessor(id):
    n' = n
    while (id !between (n', n'.successor)
        n' = n'.closest_preceeding)finger(id)
    return n'
```

```
n.closest_preceeding_finger(id):
    for i = m downto 1
        if (n.finger[i].node between (n, id))
            return n.finger[i].node
    return n
```



A
B

<u>Theorem</u>: With high probabilty (hardness assumption), the number of nodes that need to be contacted to reach some key K is O(LogN)

<u>Proof:</u>

n wants to find successor(K) for some key k, needs to find successor(K).predecessor = p

Looks at finger table for largest predecessor to K

Finds p in the i^{th} finger, ie p between (n, n.finger[i].start)

∃ some node f in the finger table responsible for that interval => f is at least 2^{i-1} from n

Distance between f and p, is <u>at most</u> 2^{i-1} since they are both in the same finger interval

⇔ distance from f to p is at most ½ the distance from n to p

*And if the distance decreases ½ each step, down to a distance of 1, then the number of nodes required to contact is **O(LogN)***

QED

What do we have so far

1. A method for mapping keys to nodes => Hash function

2. A <u>simple</u> method for routing requests around the network, which is provably correct

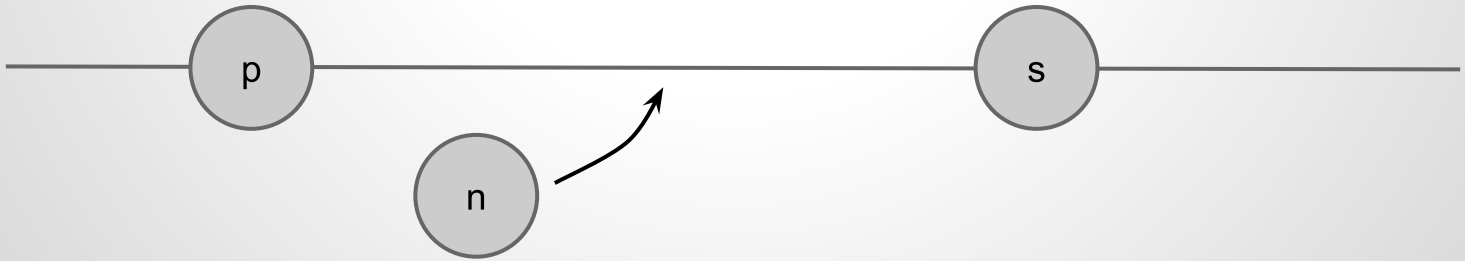
3. An <u>efficient</u> method for routing requests around the network, as long as me maintain information on *m* many nodes using a Finger Table, that is proved to operate on *O(LogN)*

Building and Maintaining

- Original statements: Chord works great under high churn - nodes joining and leaving - while maintaining *efficient* and *correct* lookups
- Need to devise a method to allow nodes to *join* the network, and allow messages to be routed with the above requirements
- Need one extra piece of information
    - predecessor pointers

Join (Simple Model)

- *n* wants to join the network, knows of some *n' already in the network*
- Insert n into the network:
    - *n* asks *n'* for the *successor(n) = s*
    - *n.successor = s, n.predecessor = s.predecessor = p*
    - *notify(n.successor)*



Init Fingers

- Now that *n* has *joined* the network, he can begin participating and routing messages
  - Since the only requirement for correctness, is maintaining successors
- Needs to initialize its finger table to be effecient and complete

Initialize Fingers

- *for i = 1 .. m - 1*
  if n.finger[i+1] ∈ [n, finger[i].node)
  finger[i+1].node = finger[i].node
  else
  finger[i+1].node = n'.find_successor (finger[i+1].start)

Fix Network Fingers

- *for i = 1 .. m*
  p = find_successor(n - 2^{i-1}*)*
  p.update_finger_table(n, i)

Update Finger Table(s, i)

- *if s ∈ [n, finger[i].node)*
  finger[i].node = s
  p = predecessor
  p.update_finger_table(s, i)

Problems

- This method requires *$O(Log^2N)$* messages
- Complicated
- Fix:
  - Have node n get successor(n) finger table, since it will be ~similar~, and use that as a base table
  - Results in *$O(LogN)$*
- This method doesn’t scale well
- Purpose, was to fimilarize you with the process of initializing finger tables

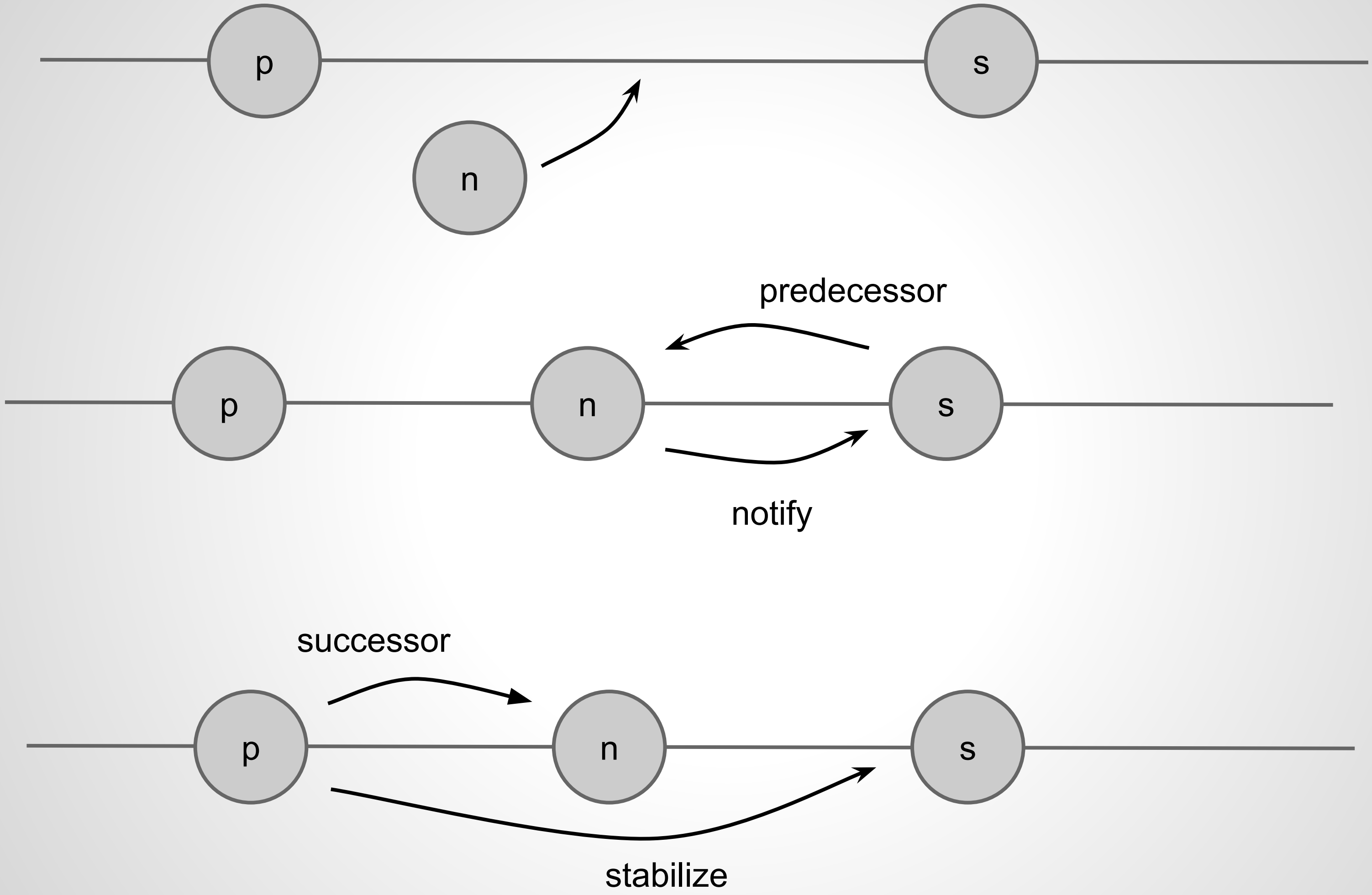
Better Method

- Concurrent operations
- Original method aggressively maintains finger table
- Need to separate correctness from performance
  - Correctness: only need successor pointers

Better Join

- Using some n’ already in the network

  n.successor = n’.find_successor(n)
- *Stabilize*
  - *Instead of initializing predecessor immediately, the stabilize routine is called periodically*
  - *Asks successor who its predecessor is*
  - *If its us, we’re good*
  - *if not, then we have a new successor, and we notify him of us*
- Fix Fingers
  - Instead of actively initializing our entire finger table, periodically set some random finger in the finger table
  - i = random number $\in$ *(1, m]*
  - *finger[i].node = find_successor(finger[i].start)*



p
s
n
predecessor
p
n
s
notify
successor
p
n
s
stabilize

What did we just get?

- Correct
  - After a node joins the network, even before the fingers are set, we have correctness because, *join* and *stabilize* maintain the correct predecessor/successor pointers
- Efficient
  - Even before the fingers are set, and if (from the previous example), someone was looking for successor(s), the current state of the network finger tables would route to p, then p would route one extra hop to n, then finally to s.
  - Still *O(LogN)*


Failure

Failures

- Need to handle failures, while maintaining correctness, and performance
- Basic Idea
  - If we maintain a little more info on nodes, we might get there
  - Maintain a successor-list of the *r* closest successors
  - If a node notices his successor has failed, then he can just route to the next successor from the successor list
  - => Correct
    - Since we maintained the correct successors

Failure AND Performance? Not Possible

- When nodes fail, over time stabilize/fix_fingers recognizes a failed node and ensures all successor/predecessor and fingers are up to date
- However before a node is aware of a failure, and tries to route to that node, what happens?
    - The route will fail
    - Try again with the next preceding node in the finger table
        - or successor list if index is low

- *<u>Theorem</u>: With a successor list of size r = O(LogN), in network that is intialy stable, then every node fails with a probability of ½, find_successor(K) will return the <u>correct</u> living node responsible for K*

- *"Proof" Intuition*
    - *The probability of <u>all</u> nodes in the successor list failing is 2^{-r} = 1/N, therefore with high probability n still maintains a successor from the successor list to route to.*

Did we achieve our Goal?

- Simple protocol/functionality?
- Provably correct?
  - Under high churn?
- Provably performant
  - Under high churn?

What did we get along the way?

- Load Balance
- Decentralized
- Scalability
- Availability

That's all Folks!