Distributed Hash Table

DHT ?

This part introduce the **theory** and **implementation** of distributed Hash Table

A distributed hash table is a class of a decentralized distributed system that provides a lookup service similar to a hash table: (key, value)

--Wikipedia

Decentralized is a vital character of Distributed Hash Table

Why and How to

Why Decentralized

- ✗ Too much data & high expense
- ✗ Convenient
- ✗ Anonymous
- ✗ Security

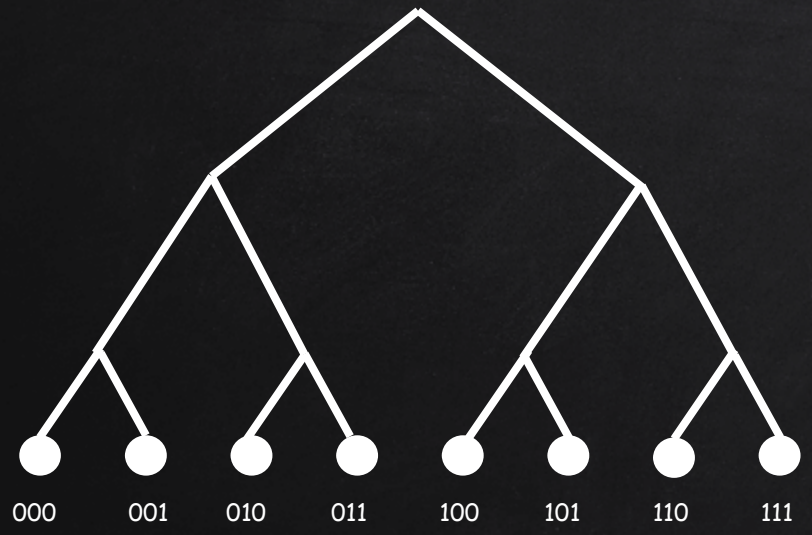
How to become Decentralized

- ✗ Smart approach to organize computers in network
- ✗ Efficient hash function
- ✗ Efficient strategy to look up values via key

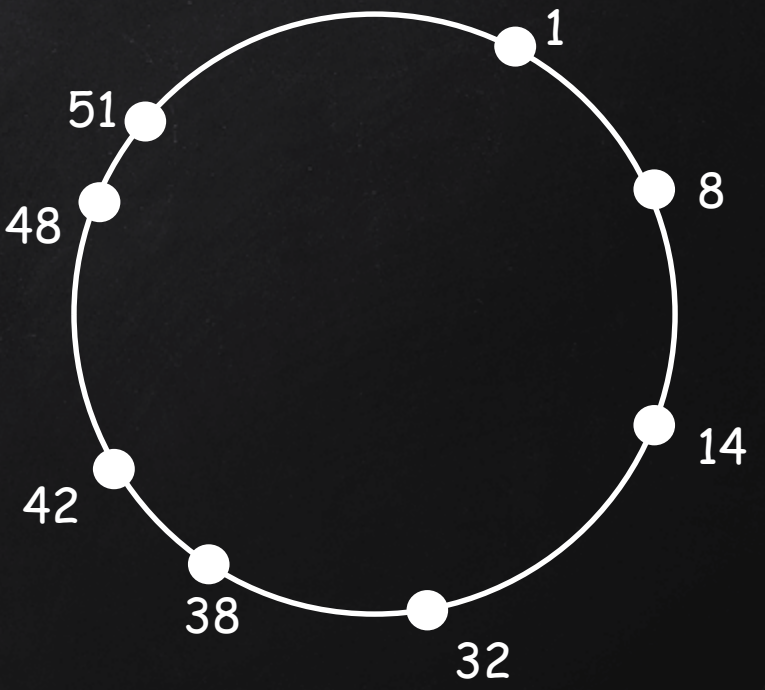
Implementation

Kademlia / Chord/ Pastry/ CAN/Koordes...

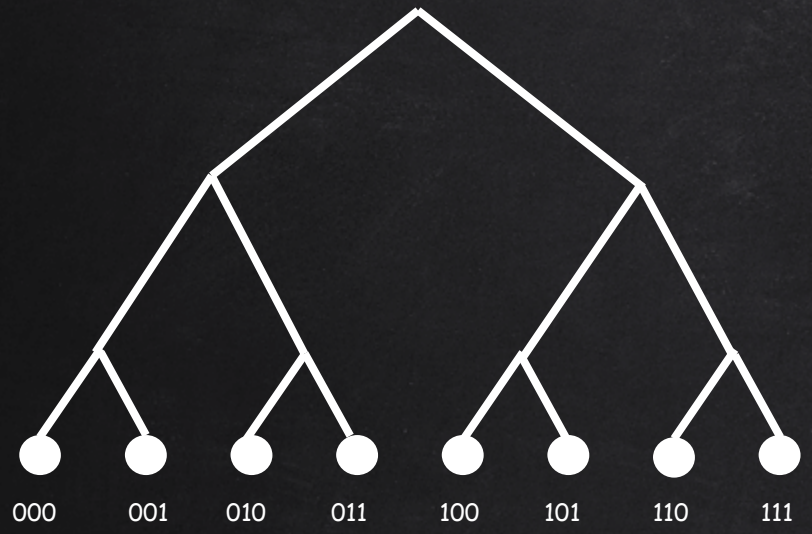
Kademlia - The most widely used



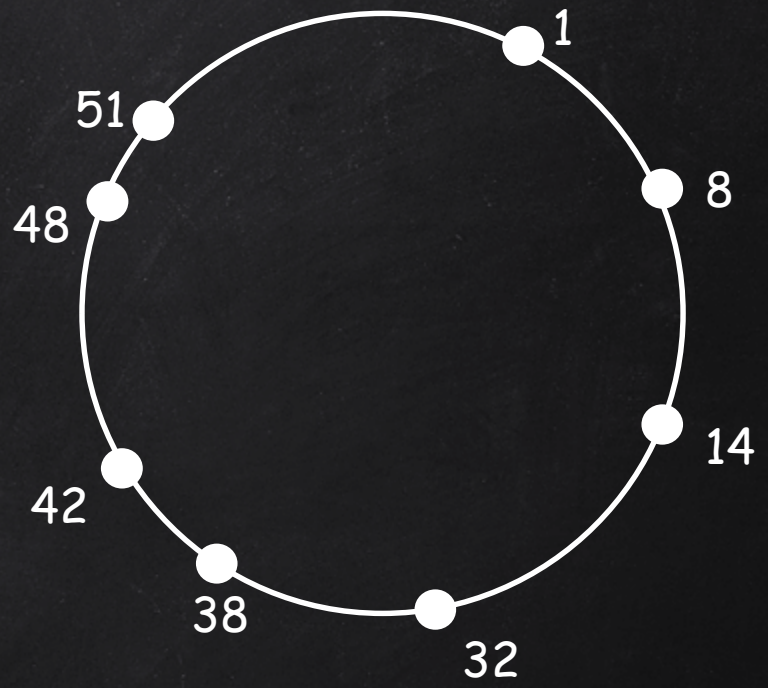
Chord - What we implemented in PPCA



Kademlia - The most widely used



Chord - What we implemented in PPCA



Key := "Fan Zhou"

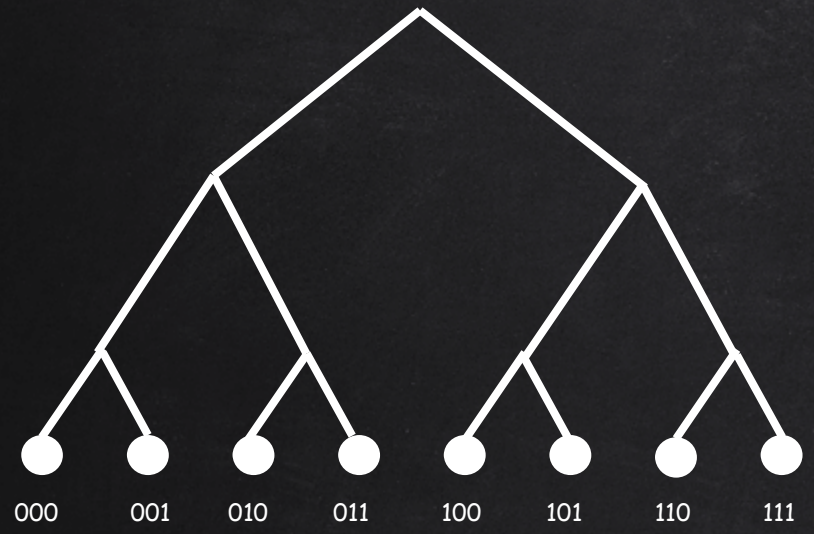
Node ID := "114.51.41.91:810"

SHA-1

Hashed Id (hex):

2486a1e52703c5b049d1
61c0cec2137a9ca97f73
965c4e5d7803f1f3f65c
bea68888d86e84ca5950

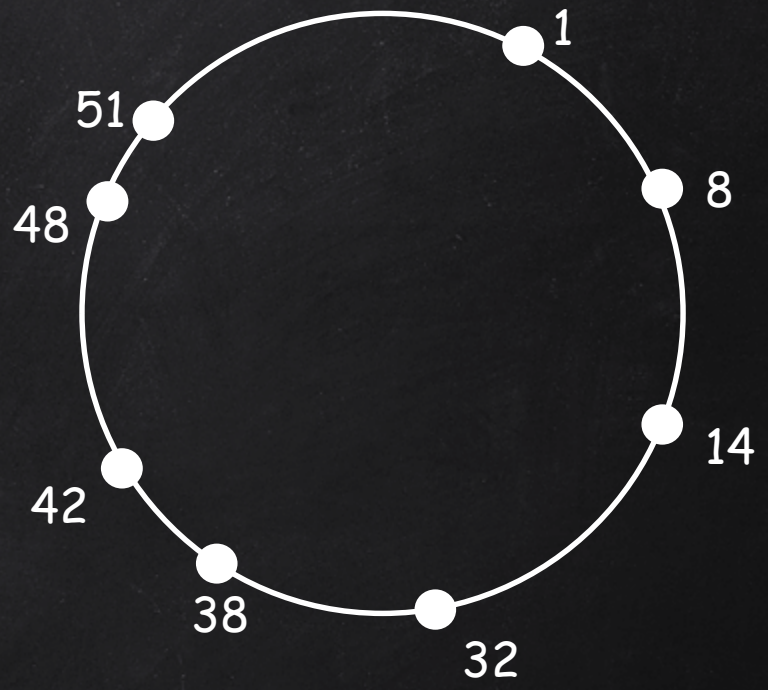
Kademlia – The most widely used



dis(a, b) :=

a xor b == b xor a

Chord – What we implemented in PPCA

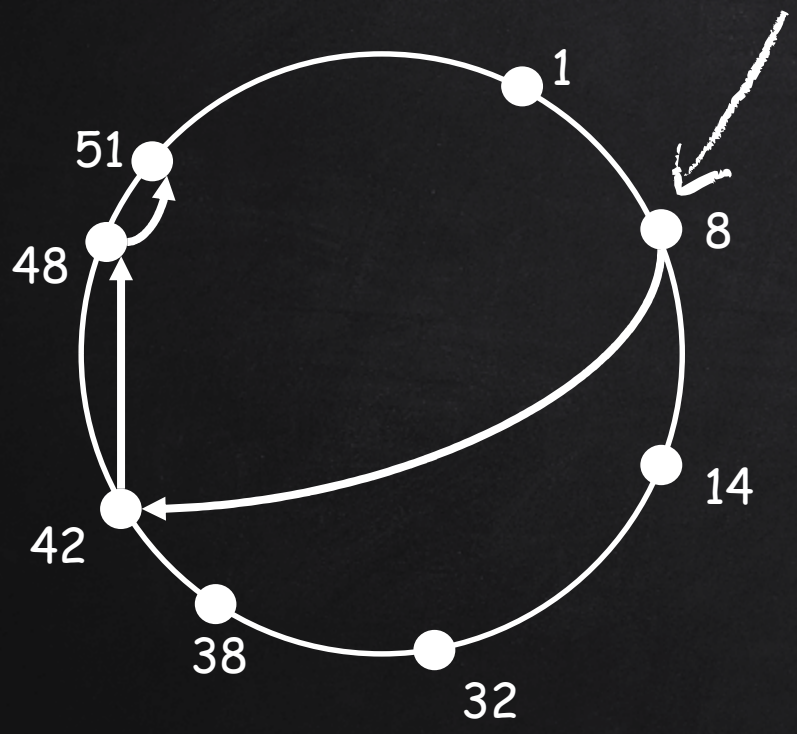


dis(a, b) :=

$(int(b) - int(a) + 2^{160}) \bmod 2^{160}$

The <KEY, VALUE> will be given to the closest node

Chord – Look Up

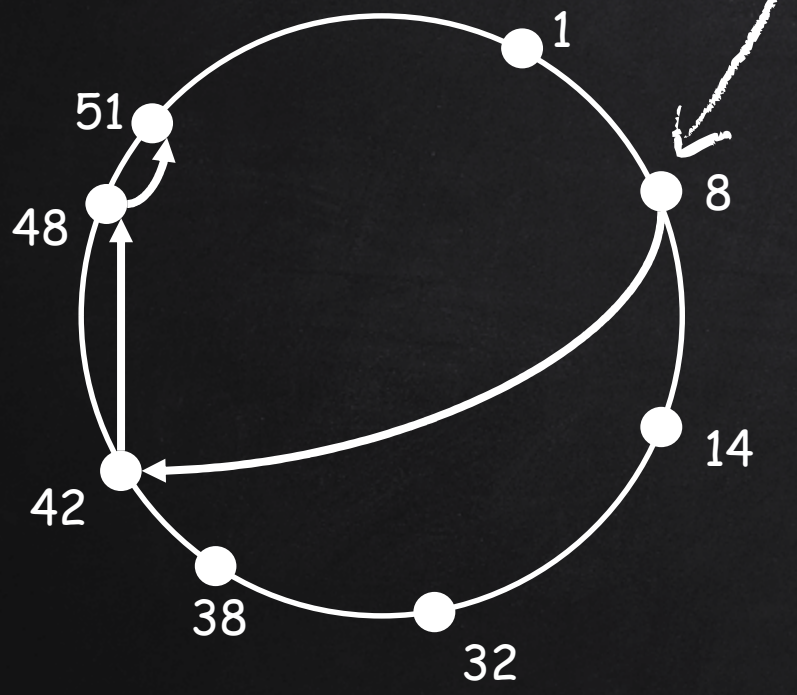


ID	Node
$8 + 2^0$	#14
$8 + 2^1$	#14
$8 + 2^2$	#14
$8 + 2^3$	#32
$8 + 2^4$	#32
$8 + 2^5$	#42

Steps of look up:

- ✗ 1. Node #8 want to look up who has key 49.
- ✗ 2. Node #8 check whether his successor 14 is the nearest node of 49, obviously not.
- ✗ 3. Node #8 iterate its finger table from bottom to top for the available closest preceding node of 49 he knows, then he find Node #42.
- ✗ 4. Node #8 ask node Node #42 to repeat these steps until the closest node of 49, Node #51 is found.

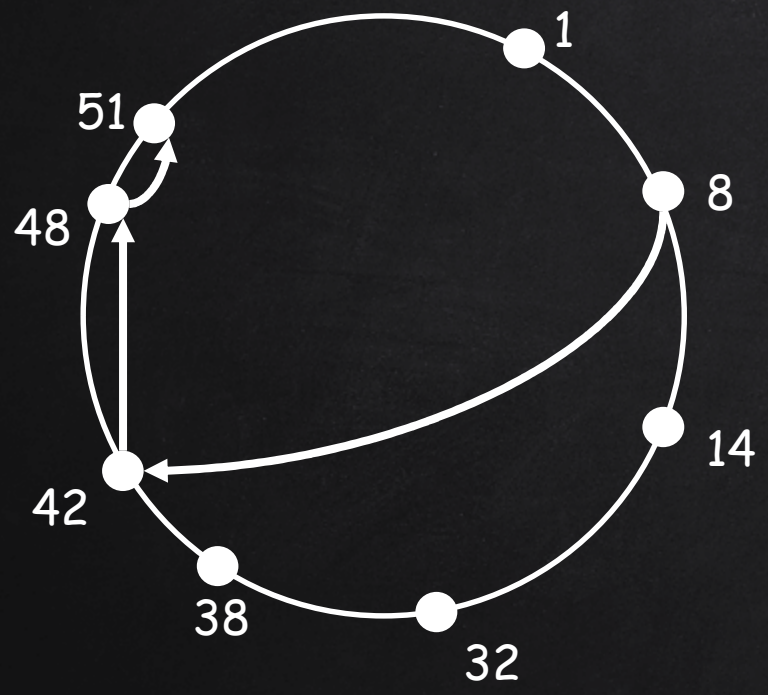
Chord – Look Up



ID	Node
$8 + 2^0$	#14
$8 + 2^1$	#14
$8 + 2^2$	#14
$8 + 2^3$	#32
$8 + 2^4$	#32
$8 + 2^5$	#42

O(logN)

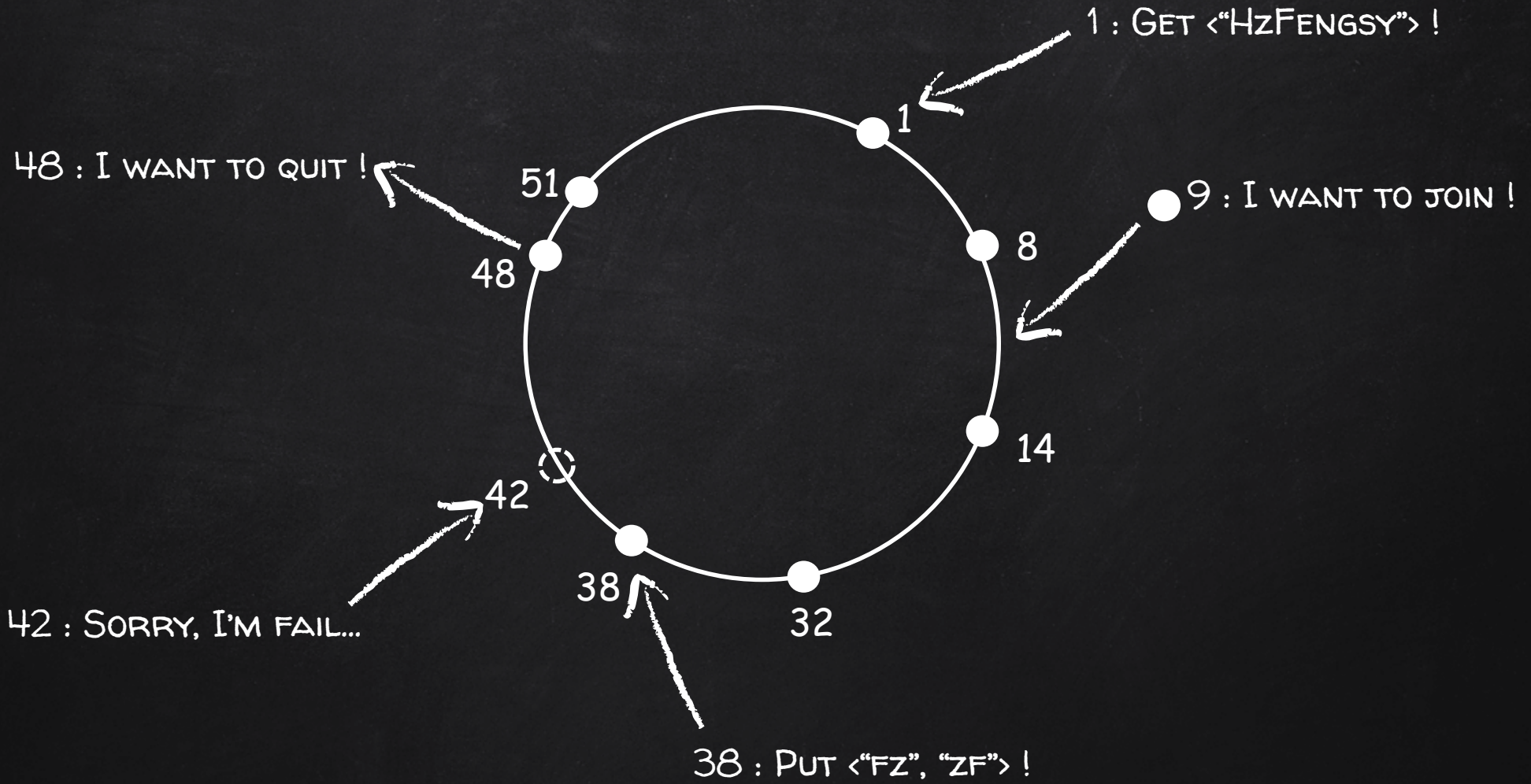
Chord – Look Up



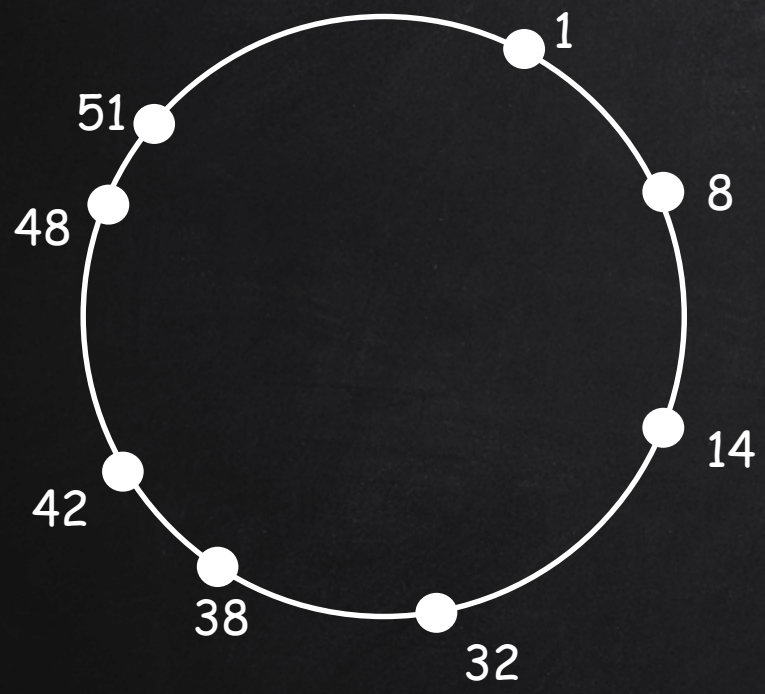
Available operation for any computer in the network:

- ✗ 1. Put <Key, Value>
- ✗ 2. Look up <Key> for <Value>
- ✗ 3. Remove <Key, Value> from network
- ✗ 4. Modify <Key, Value> to <Key, New Value>

Chord – Robustness



Chord – Robustness



Sacrifice TIME

For

STABILITY and CORRECTNESS

Chord – Robustness

Strategies to maintaining the network:

- ✗ 1.Maintain a successor list via copying successor's SUCCESSOR LIST, removing the last entry of it and adding the successor as the first entry of the list.
- ✗ 2. Check whether successor changed and notify the new successor about himself if changed periodically.
- ✗ 3.Tell the successor about himself periodically so that the successor can update his predecessor
- ✗ 4.Fix one finger table entry per fix interval
- ✗ 5. Notify the predecessor and successor when a node want to quit
- ✗ 6.Check whether predecessor is failed periodically so that it can be updated to new available predecessor

Chord – Robustness

Strategies to maintaining the network:

- ✗ 1.Maintain a successor list via copying successor's SUCCESSOR LIST, removing the last entry of it and adding the successor as the first entry of the list.
- ✗ 2. Check whether successor changed and notify the new successor about himself if changed periodically.
- ✗ 3.Tell the successor about himself periodically so that the successor can update his predecessor
- ✗ 4.Fix one finger table entry per fix interval
- ✗ 5. Notify the predecessor and successor when a node want to quit
- ✗ 6.Check whether predecessor is failed periodically so that it can be updated to new available predecessor

For 1,2 and 3 we use stabilize() and notify()

```
n.stabilize()
        x := n.succ.pre
        if x between n and n.succ
                n.succ = x
        n.succList = n.succ.succList
        n.succList.pop_back()
        n.succList.push_front(n.succ)
        n.succ.notify(n)

n.notify(Node n')
        if n.pre == nil or n' is between n.pre and n
                n.pre = n'
```

Chord – Robustness

Strategies to maintaining the network:

- ✗ 1.Maintain a successor list via copying successor's SUCCESSOR LIST, removing the last entry of it and adding the successor as the first entry of the list.
- ✗ 2. Check whether successor changed and notify the new successor about himself if changed periodically.
- ✗ 3.Tell the successor about himself periodically so that the successor can update his predecessor
- ✗ 4.Fix one finger table entry per fix interval
- ✗ 5. Notify the predecessor and successor when a node want to quit
- ✗ 6.Check whether predecessor is failed periodically so that it can be updated to new available predecessor

For 4 we use function fix_finger()

```
n.fix_finger()
        n.finger[next]  = n.find_succ(n.id + 2^next)
        next = next + 1
        if next > 159
                next = 0
```

//n.finger is a array with length of 160, and here we use C-Style indexing

//find_succ() is a function that find which node the specific key belonged to.

Chord – Robustness

Strategies to maintaining the network:

- ✗ 1.Maintain a successor list via copying successor's SUCCESSOR LIST, removing the last entry of it and adding the successor as the first entry of the list.
- ✗ 2. Check whether successor changed and notify the new successor about himself if changed periodically.
- ✗ 3.Tell the successor about himself periodically so that the successor can update his predecessor
- ✗ 4.Fix one finger table entry per fix interval
- ✗ 5. Notify the predecessor and successor when a node want to quit
- ✗ 6.Check whether predecessor is failed periodically so that it can be updated to new available predecessor

For 5 we use quit()
n.quit()

```
n.pre.succList.remove(n)
last := len(n.succList)
n.pre.succList.append(n.succList[len - 1])
n.succ.pre = n.pre
```

// data also should be transferred here, this will be discuss in next part

Chord – Robustness

Strategies to maintaining the network:

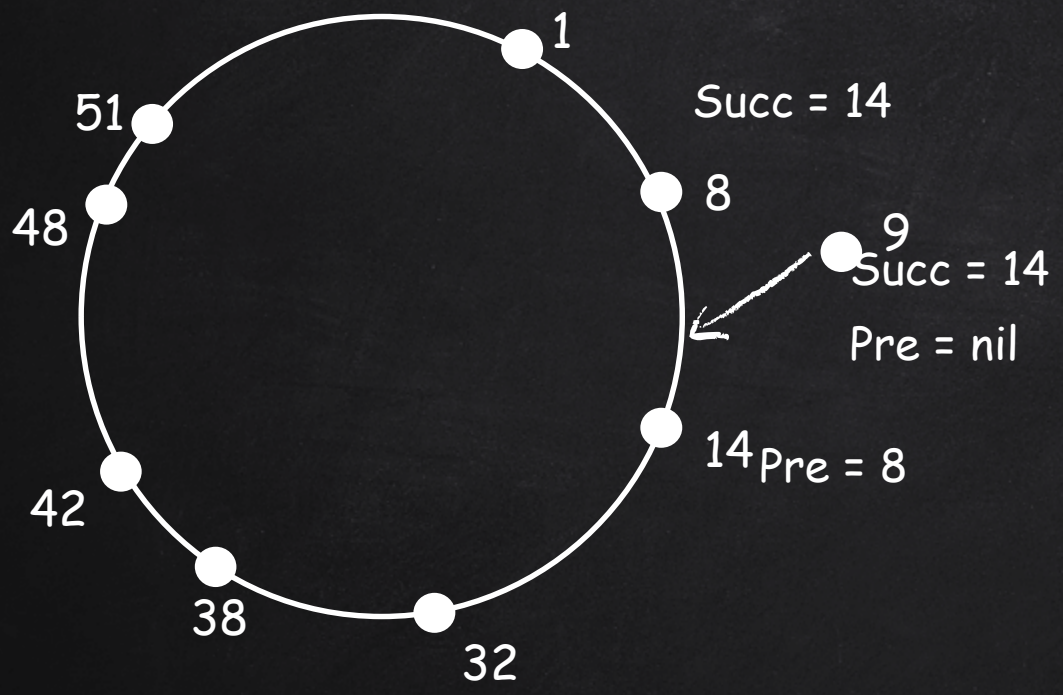
- ✗ 1.Maintain a successor list via copying successor's SUCCESSOR LIST, removing the last entry of it and adding the successor as the first entry of the list.
- ✗ 2. Check whether successor changed and notify the new successor about himself if changed periodically.
- ✗ 3.Tell the successor about himself periodically so that the successor can update his predecessor
- ✗ 4.Fix one finger table entry per fix interval
- ✗ 5. Notify the predecessor and successor when a node want to quit
- ✗ 6.Check whether predecessor is failed periodically so that it can be updated to new available predecessor

For 6 we use check_predecessor()

```
n.check_predecessor()
        if n.pre is failed
            n.pre = nil
```

//we can use function like ping to check whether the predecessor is failed

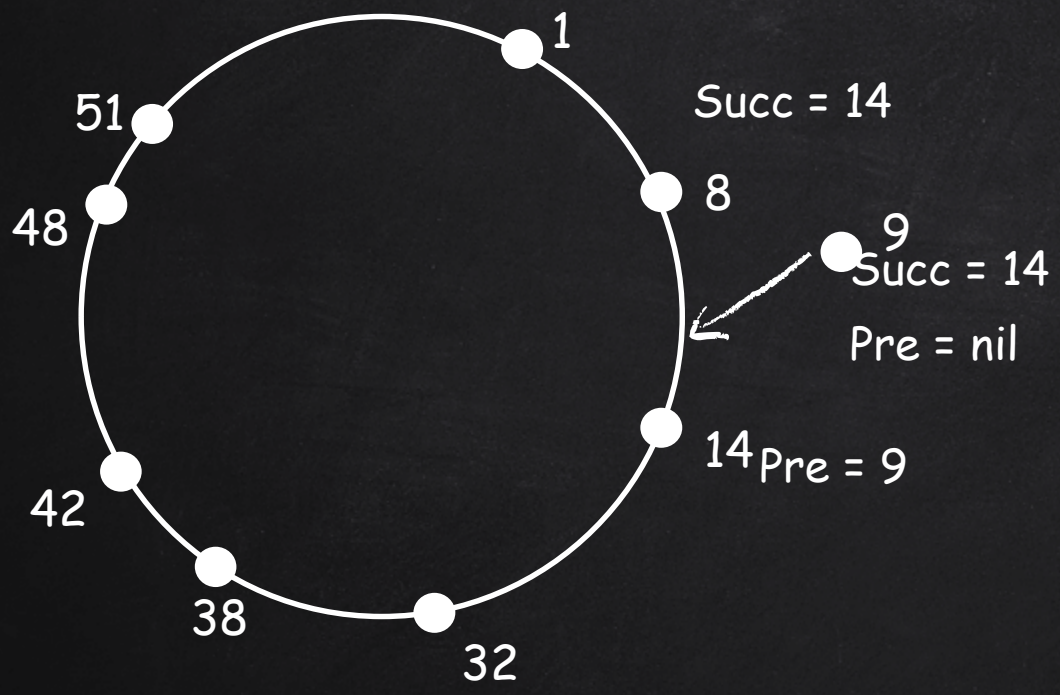
Chord – Robustness – Simulation



Condition: a node with id 9 join the network via node #51.

✗ node #9 call node #51's find_succ() and it get #14 as successor

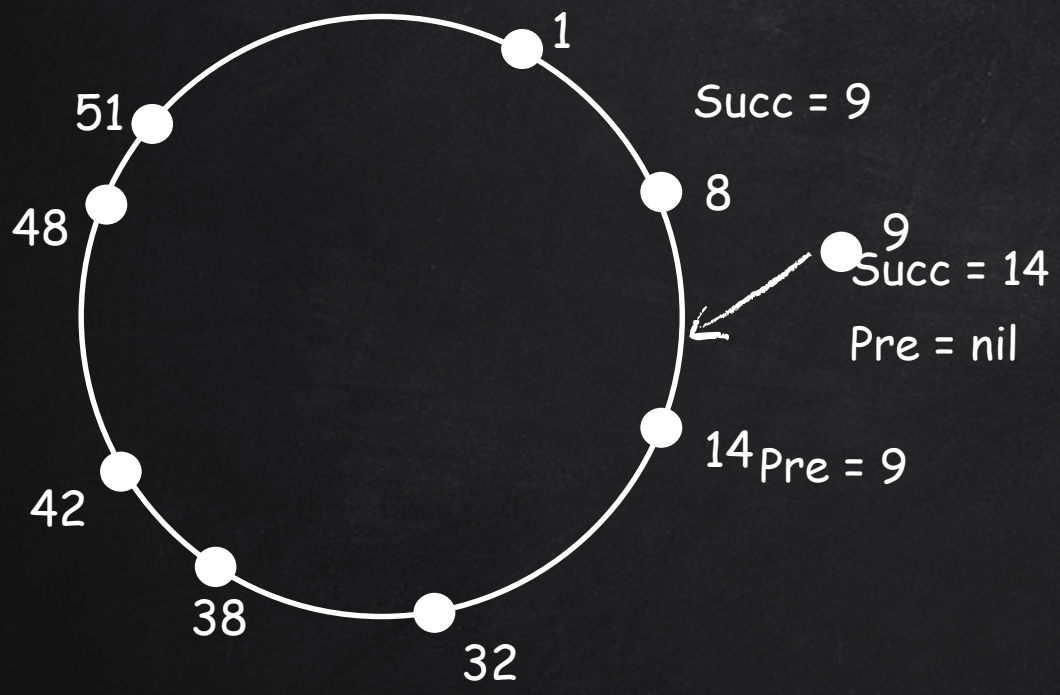
Chord – Robustness – Simulation



Condition: a node with id 9 join the network via node #51.

- ✗ node #9 call node #51's find_succ() and it get #14 as successor
- ✗ node #9 runs notify() in stabilize() and node #14 update his predecessor to 9

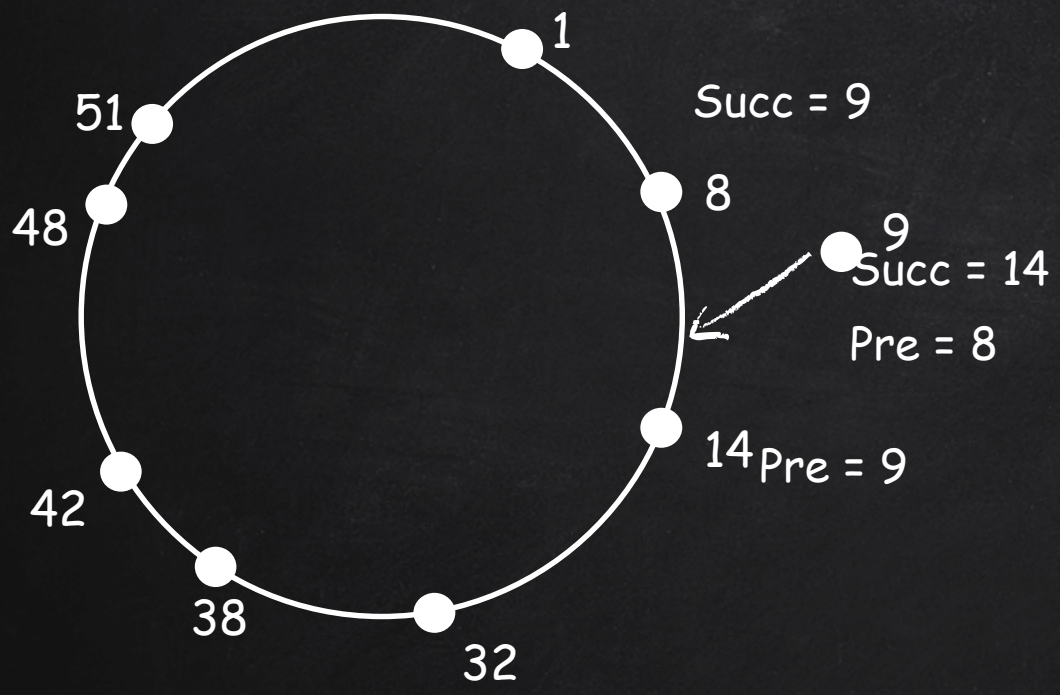
Chord – Robustness – Simulation



Condition: a node with id 9 join the network via node #51.

- ✗ node #9 call node #51's find_succ() and it get #14 as successor
- ✗ node #9 runs notify() in stabilize() and node #14 update his predecessor to 9
- ✗ node #8 runs stabilize() and update his succ to 9

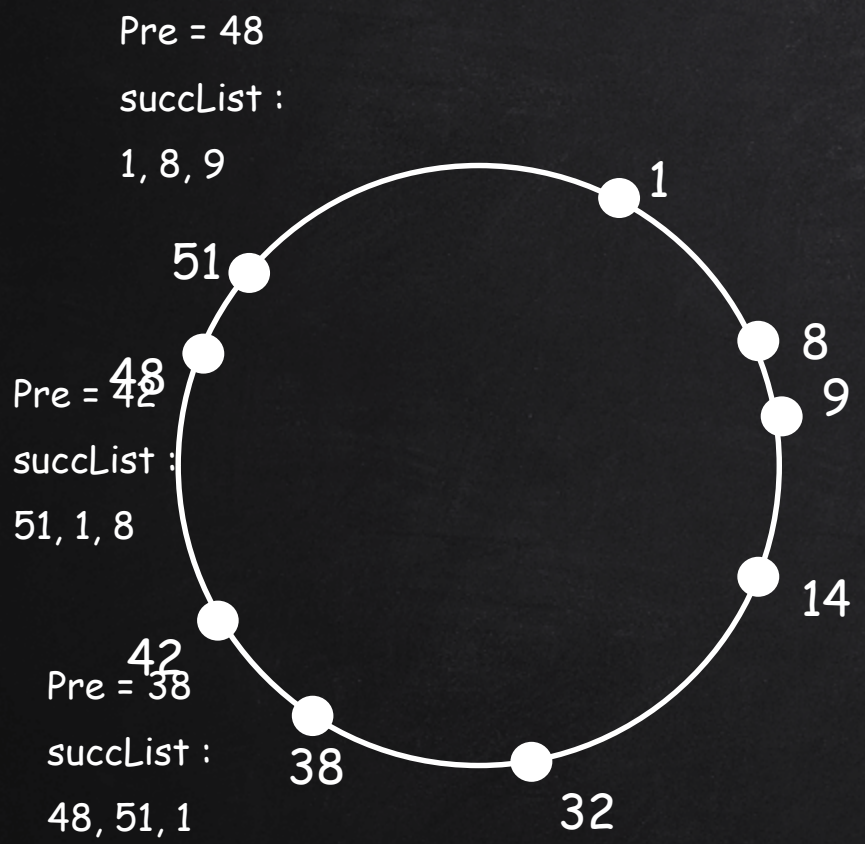
Chord – Robustness – Simulation



Condition: a node with id 9 join the network via node #51.

- ✗ node #9 call node #51's find_succ() and it get #14 as successor
- ✗ node #9 runs notify() in stabilize() and node #14 update his predecessor to 9
- ✗ node #8 runs stabilize() and update his succ to 9
- ✗ node#8 runs notify() in stabilize() and node #9 update his pre to 8
- ✗ As the time passed, finger table of each node will be fixed gradually by fix_finger() function, then the network become stable again

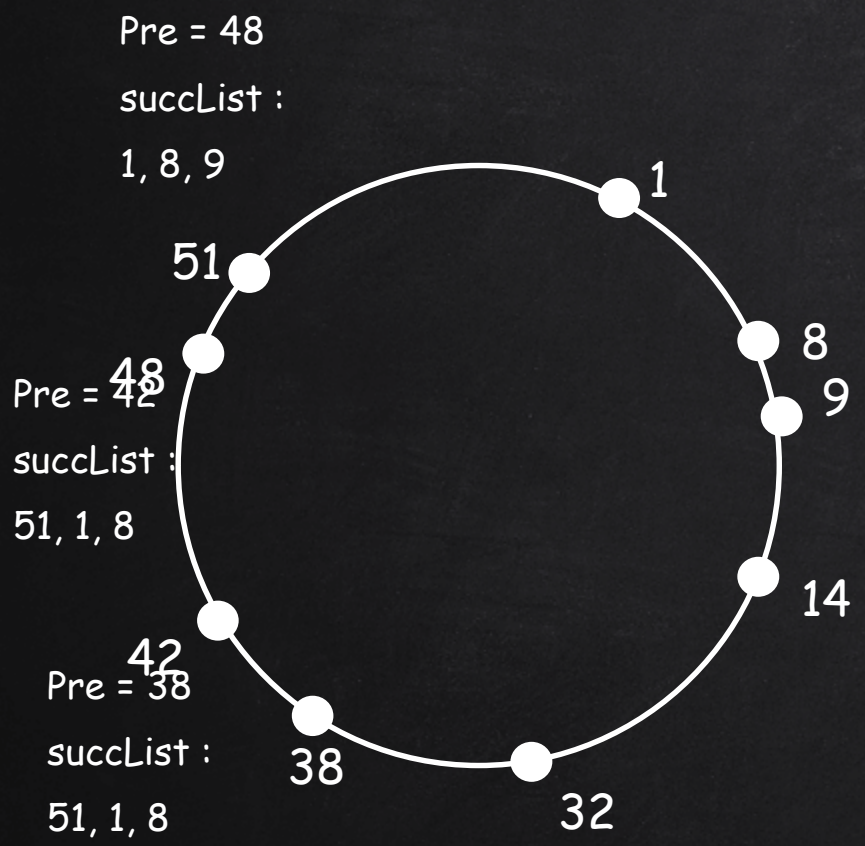
Chord – Robustness – Simulation



Condition: node #48 quit

✗ node #48 runs function quit()

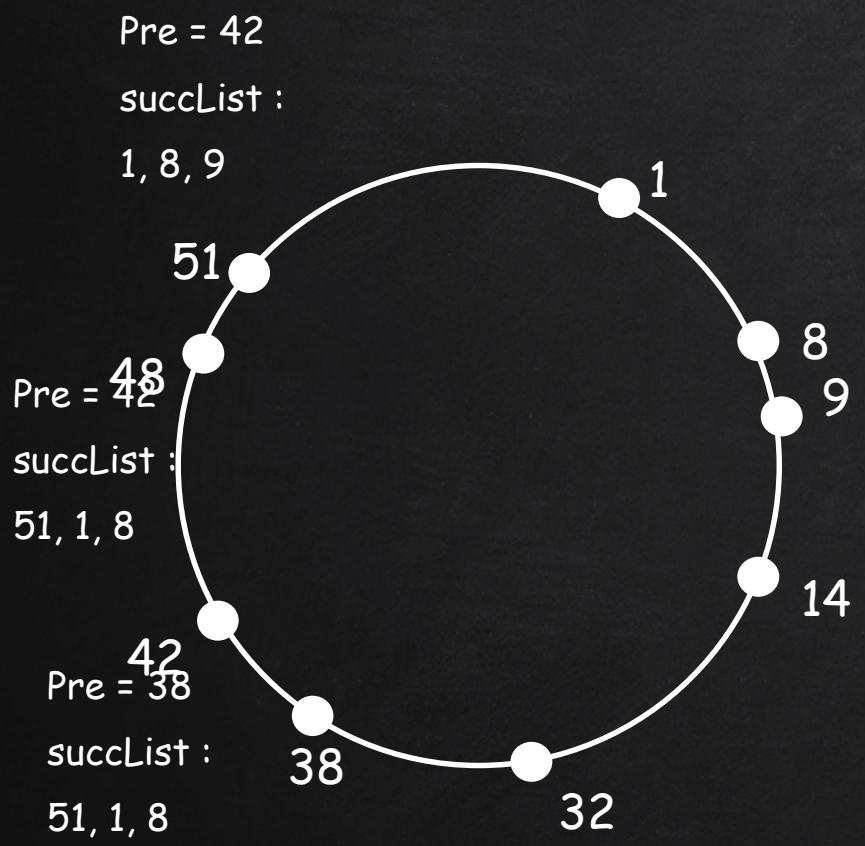
Chord – Robustness – Simulation



Condition: node #48 quit

- ✗ node #48 runs function quit()
- ✗ node #42 is called to remove 48 from his succList and add 8 to the end of his succList

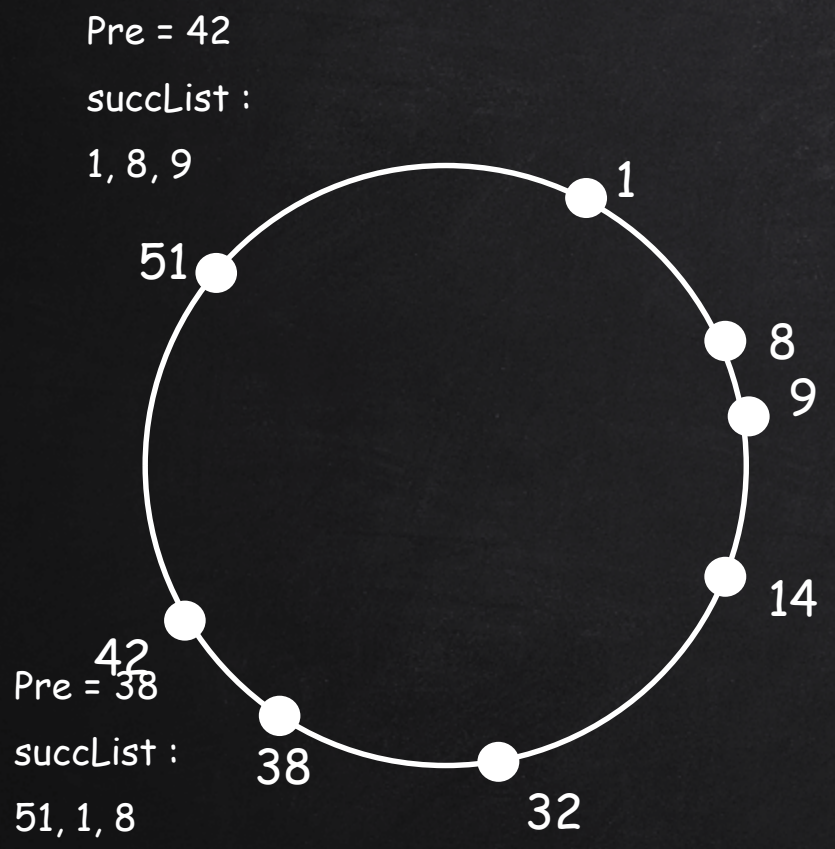
Chord – Robustness – Simulation



Condition: node #48 quit

- ✗ node #48 runs function quit()
- ✗ node #42 is called to remove 48 from his succList and add 8 to the end of his succList
- ✗ node #51 is called to update his predecessor to 42

Chord – Robustness – Simulation



Condition: node #48 quit

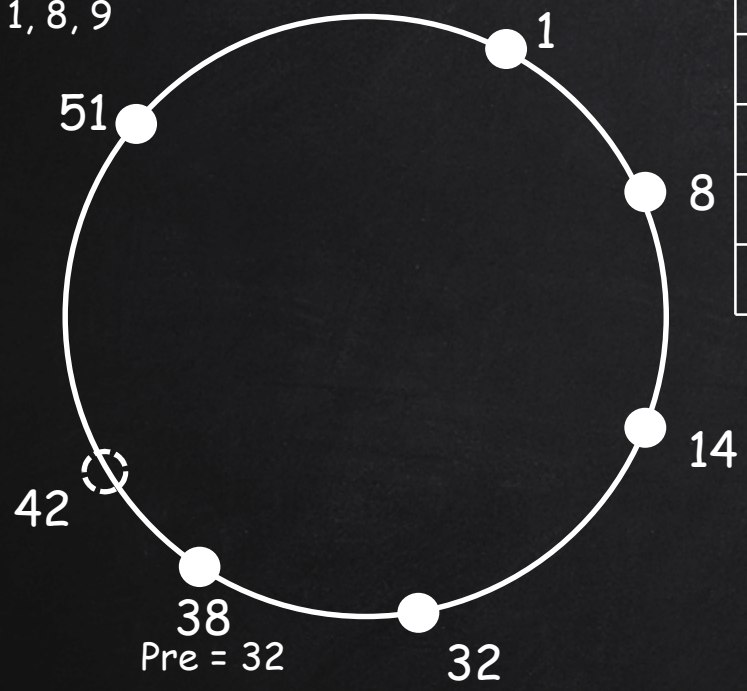
- ✗ node #48 runs function quit()
- ✗ node #42 is called to remove 48 from his succList and add 8 to the end of his succList
- ✗ node #51 is called to update his predecessor to 42
- ✗ then node #48 quit successfully
- ✗ As the time passed, finger table of each node will be fixed gradually by fix_finger() function, then the network become stable again

Chord – Robustness – Simulation

Pre = nil

succList :

1, 8, 9



Pre = 32

succList :

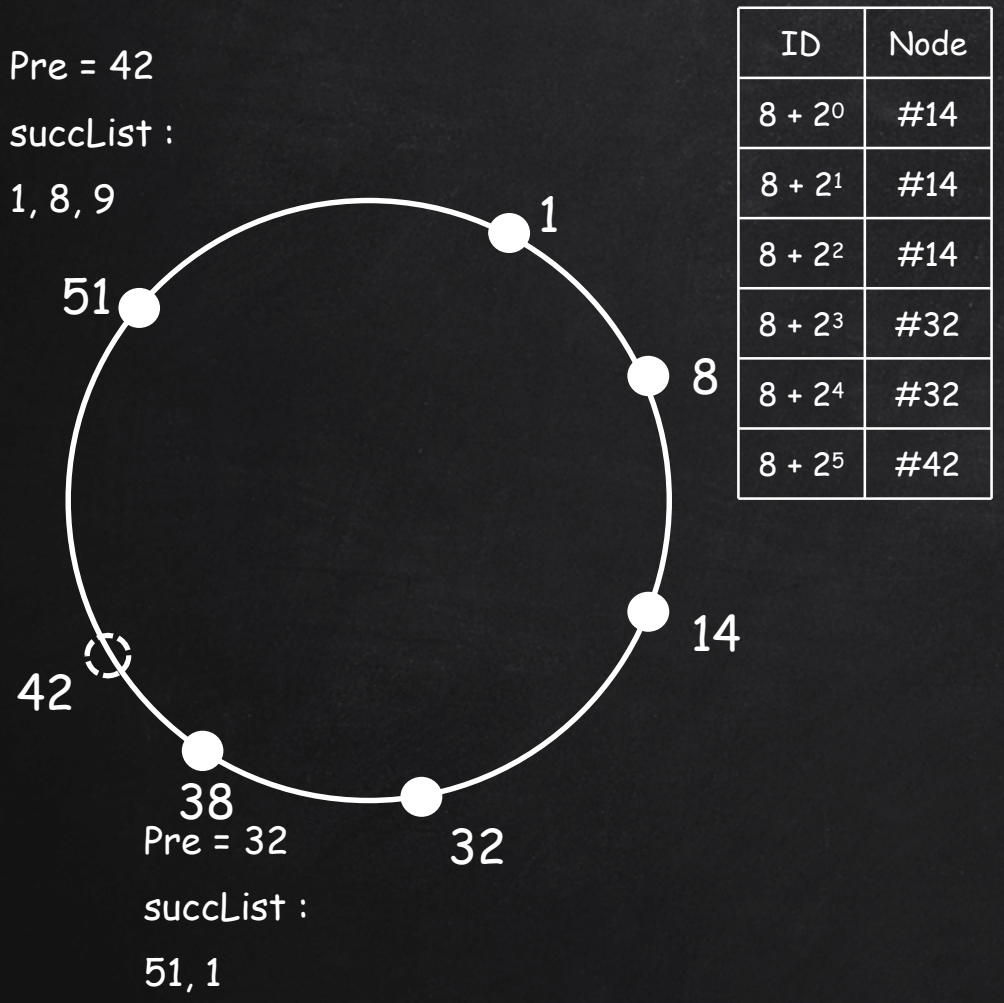
42, 51, 1

ID	Node
$8 + 2^0$	#14
$8 + 2^1$	#14
$8 + 2^2$	#14
$8 + 2^3$	#32
$8 + 2^4$	#32
$8 + 2^5$	#42

Condition: node #42 failed

✗ node #51 runs check_predecessor and find his predecessor failed, so he set to nil

Chord – Robustness – Simulation

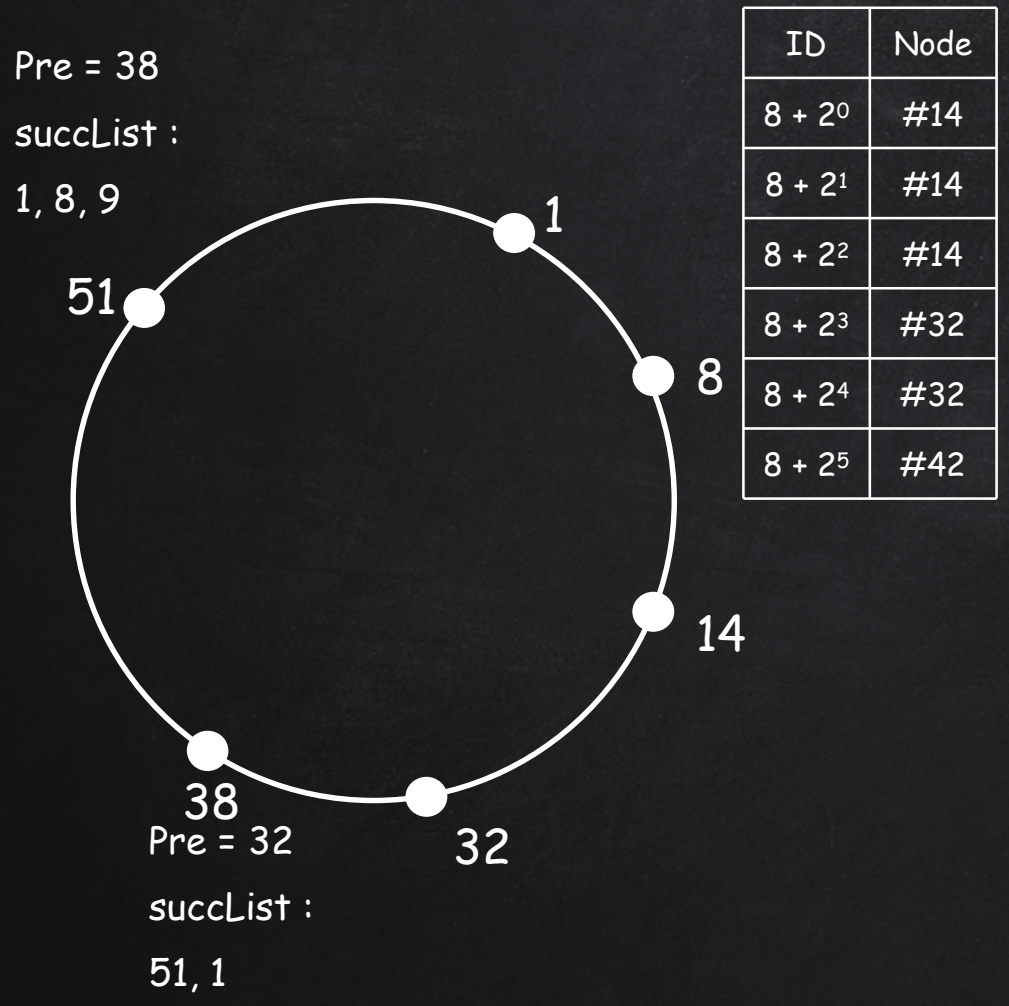


ID	Node
$8 + 2^0$	#14
$8 + 2^1$	#14
$8 + 2^2$	#14
$8 + 2^3$	#32
$8 + 2^4$	#32
$8 + 2^5$	#42

Condition: node #42 failed

- ✗ node #51 runs check_predecessor and find his predecessor failed, so he set to nil
- ✗ node #38 runs stabilize() and find his successor failed, so he remove 42 from succList and choose 51 as new successor.

Chord – Robustness – Simulation

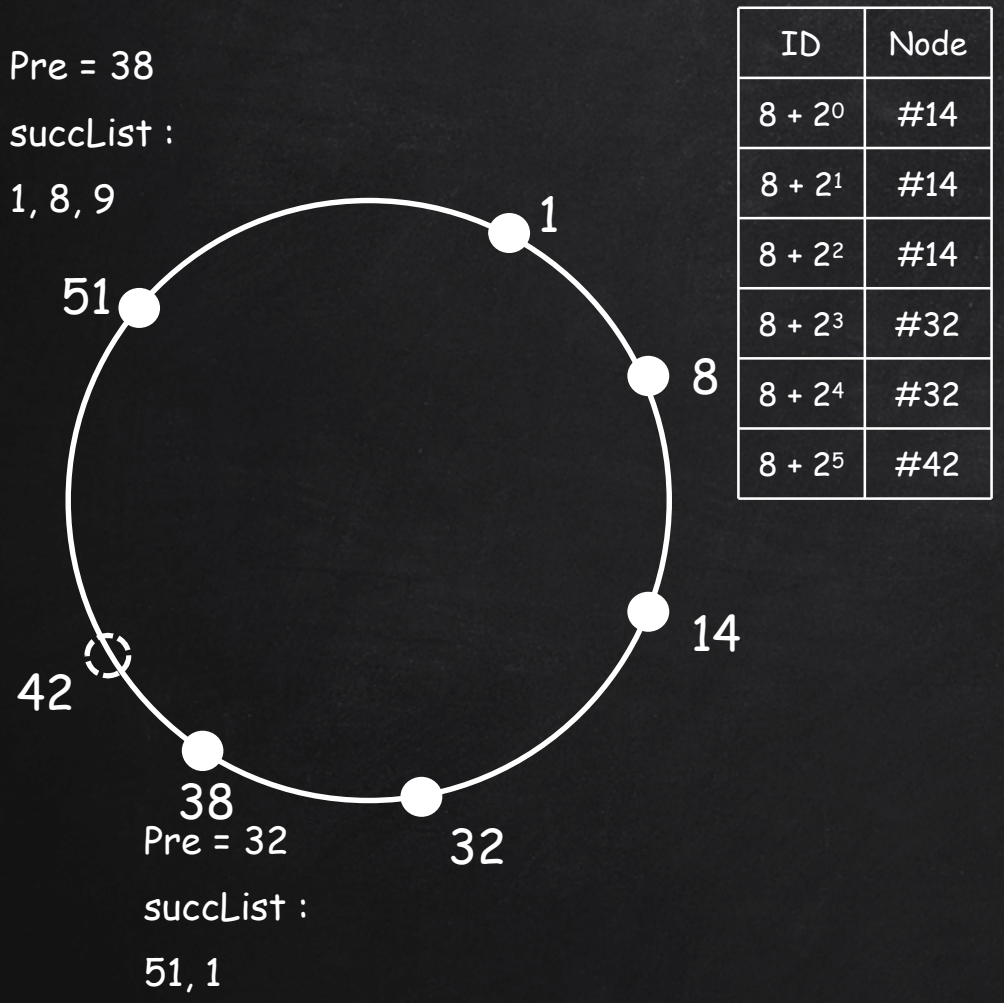


ID	Node
$8 + 2^0$	#14
$8 + 2^1$	#14
$8 + 2^2$	#14
$8 + 2^3$	#32
$8 + 2^4$	#32
$8 + 2^5$	#42

Condition: node #42 failed

- ✗ node #51 runs check_predecessor and find his predecessor failed, so he set to nil
- ✗ node #38 runs stabilize() and find his successor failed, so he remove 42 from succList and choose 51 as new successor.
- ✗ then node #38 notify his new successor for his existence, and node #51's predecessor is nil, so node #51 set his predecessor to 38
- ✗ then node impact of the failure of node #42 is now eliminated

Chord – Robustness – Simulation

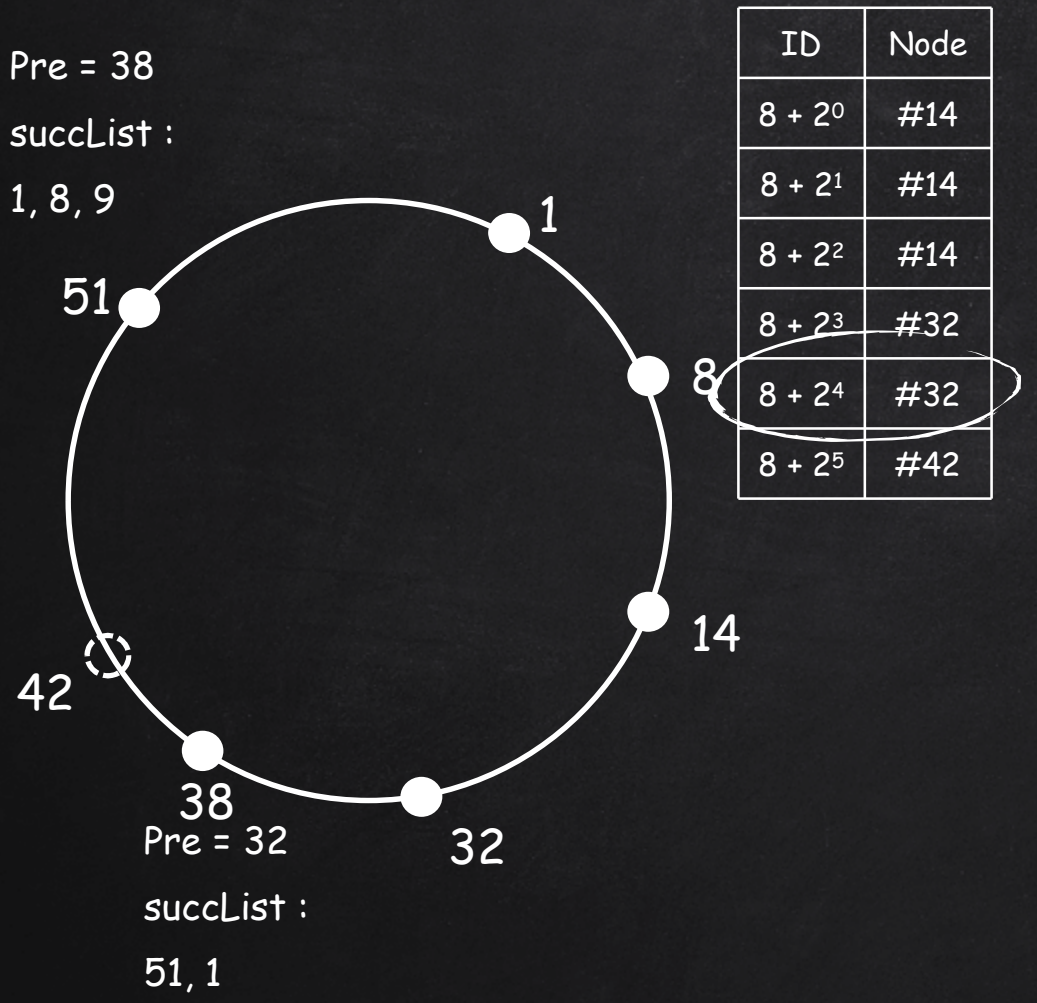


ID	Node
$8 + 2^0$	#14
$8 + 2^1$	#14
$8 + 2^2$	#14
$8 + 2^3$	#32
$8 + 2^4$	#32
$8 + 2^5$	#42

Condition: node #8 fix finger table

✗ take the previous condition, and node #8 want to fix the 5^{th} entry of his finger table

Chord – Robustness – Simulation

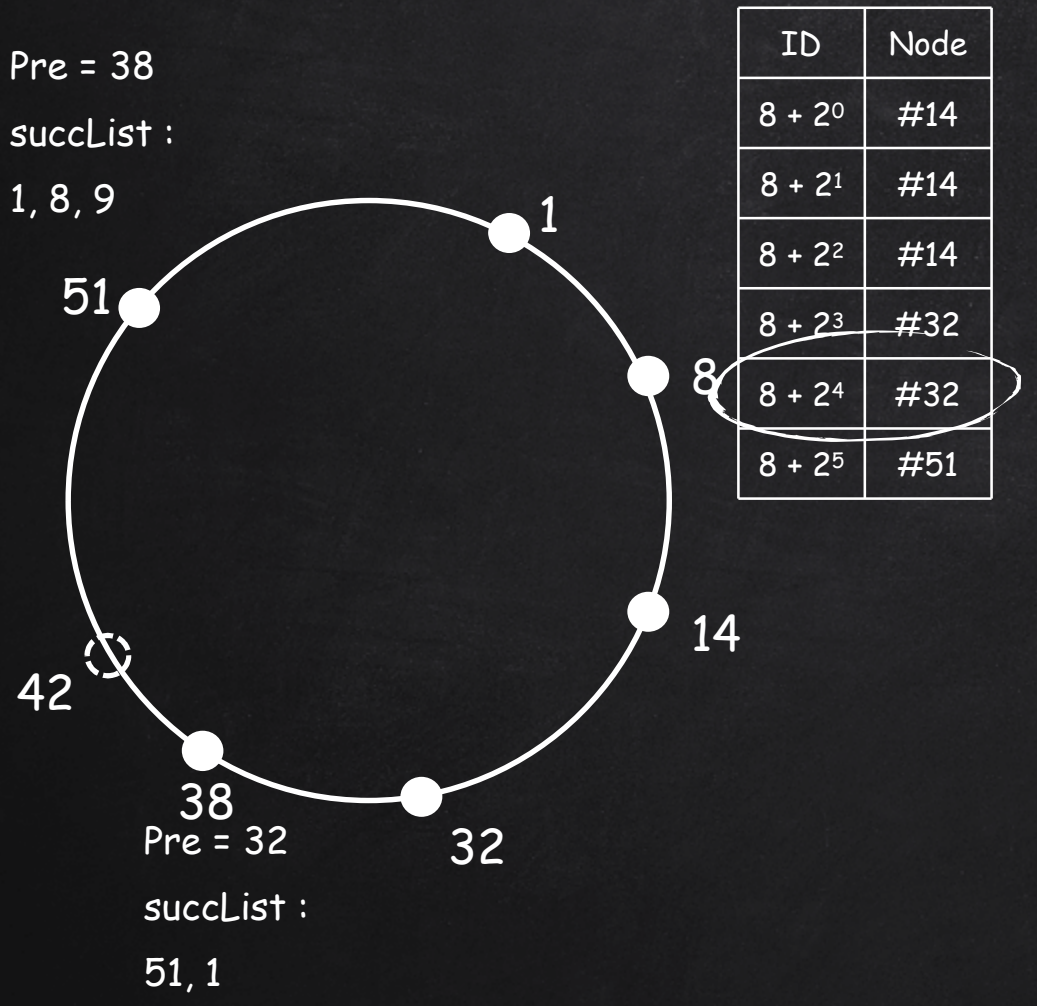


ID	Node
$8 + 2^0$	#14
$8 + 2^1$	#14
$8 + 2^2$	#14
$8 + 2^3$	#32
$8 + 2^4$	#32
$8 + 2^5$	#42

Condition: node #8 fix finger table

- ✗ take the previous condition, and node #8 want to fix the 5^{th} entry of his finger table
- ✗ he find in his for first valid predecessor of id 40 = $8 + 2^5$, and he get 32

Chord – Robustness – Simulation



ID	Node
$8 + 2^0$	#14
$8 + 2^1$	#14
$8 + 2^2$	#14
$8 + 2^3$	#32
$8 + 2^4$	#32
$8 + 2^5$	#51

Condition: node #8 fix finger table

- ✗ take the previous condition, and node #8 want to fix the 5^{th} entry of his finger table
- ✗ he find in his for first valid predecessor of id 40 = $8 + 2^5$, and he get 32
- ✗ he asks node #32 to find the successor of 40, then it become a normal look up. And finally the node #8 get 51 and update the 5^{th} entry of his finger table to 51

Chord – DATA

Strategies to maintaining data in Chord

- ✗ 1. When a node joined, some keys has assigned to his successor should be moved to him
- ✗ 2. When a node quit, all his data should be move to his successor
- ✗ 3. (Personal) When predecessor changed, some key assigned to him should be moved to the predecessor

For 1, we use a modified join() function

```
n.join(n')
        n.pre = nil
        n.succ = n'.find_predecessor()
        for k, v in n.succ.data
                if k is not between n and n.succ
                        n.data.append(k, v)
                        n.succ.data.remove(k)
```

Chord – DATA

Strategies to maintaining data in Chord

- ✗ 1. When a node joined, some keys has assigned to his successor should be moved to him
- ✗ 2. When a node quit, all his data should be move to his successor
- ✗ 3. (Personal) When predecessor changed, some key assigned to him should be moved to the predecessor

For 2, we use a modified quit() function

```
n.quit()
        n.pre.succList.remove(n)
        last := len(n.succList)
        n.pre.succList.append(n.succList[len - 1])
        n.succ.pre = n.pre
        for k, v in n.data
                n.succ.data.append(k ,v)
```

Chord – DATA

Strategies to maintaining data in Chord

- ✗ 1. When a node joined, some keys has assigned to his successor should be moved to him
- ✗ 2. When a node quit, all his data should be move to his successor
- ✗ 3. (Personal) When predecessor changed, some key assigned to him should be moved to the predecessor

For 3, we use a modified notify() function

```
n.notify(n')
        if n.pre == nil or n' is between n.pre and n
                n.pre = n'
                for k, v in n.data
                        if k not between n.pre and n
                                n.pre.data.append(k, v)
                                n.data.remove(k)
```

Chord – DATA

what about some node fails?

Chord – DATA

What about some node fails?

Applications on DHT

This part introduce some approaches that make DHT safer and my application on DHT

Make DHT safer

Create local buffer

Dump data in the memory to hard disk periodically in case the node fails

Multi key to a value

My application based on DHT

magnet:?xt=urn:btih:
e04f646aa669375c7a06b6569b70657a9bd82a0c

My application based on DHT

Stupid Chord Peer to peer File Sharing System "SCPFSS"

My application based on DHT

"SCPFSS"

A BitTorrent like system

Structue of SCPFSS

Basement: DHT

Put file hash as key and a LIST of the address of node who share this file as value

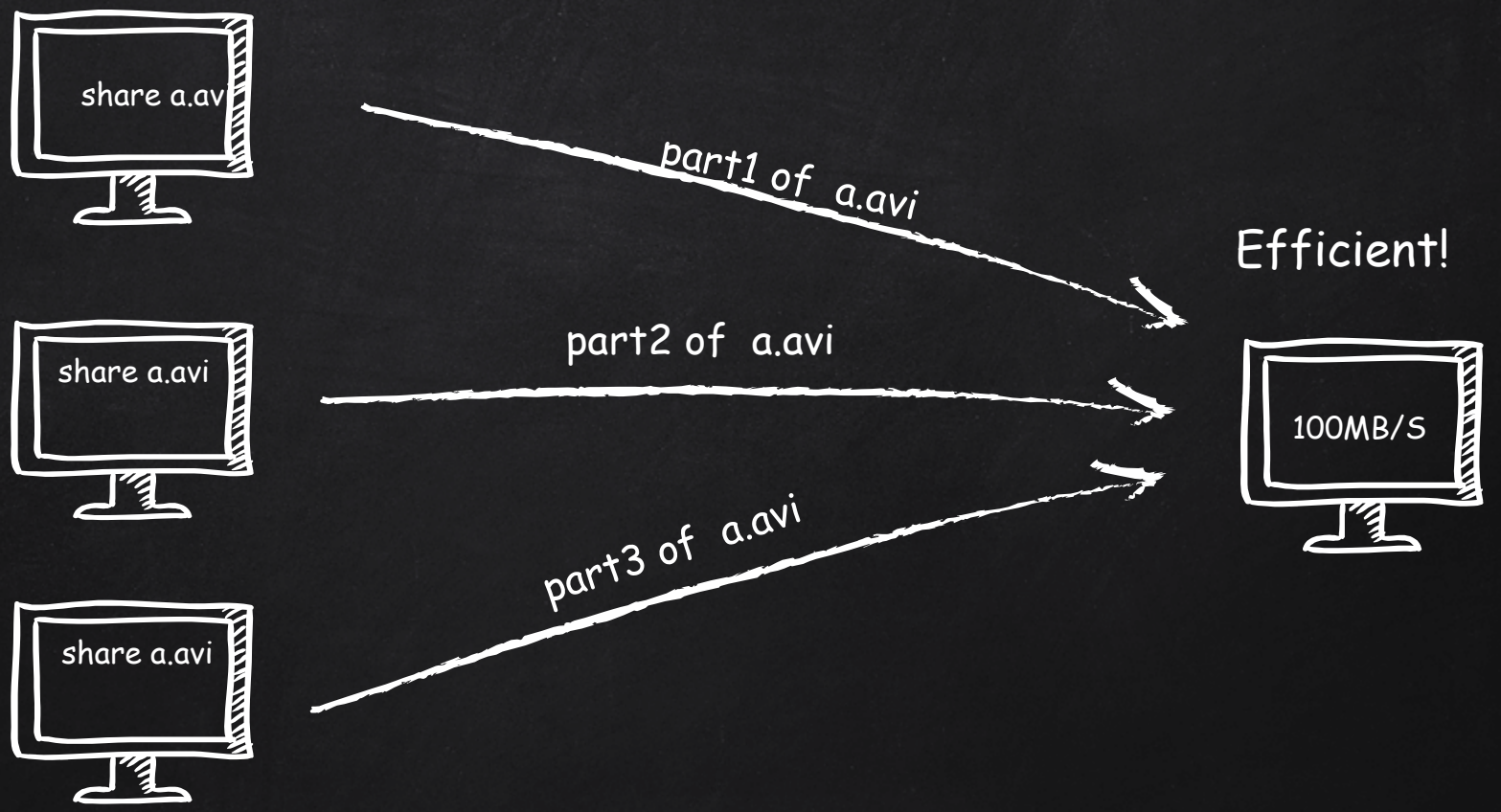
Middle: File Server RPC

Providing secive of looking up file and send file information

Upper: TCP File Server

Send (part of) files to other peers

Character of SCPFSS-Multi Threading



THANKS!

Any questions?