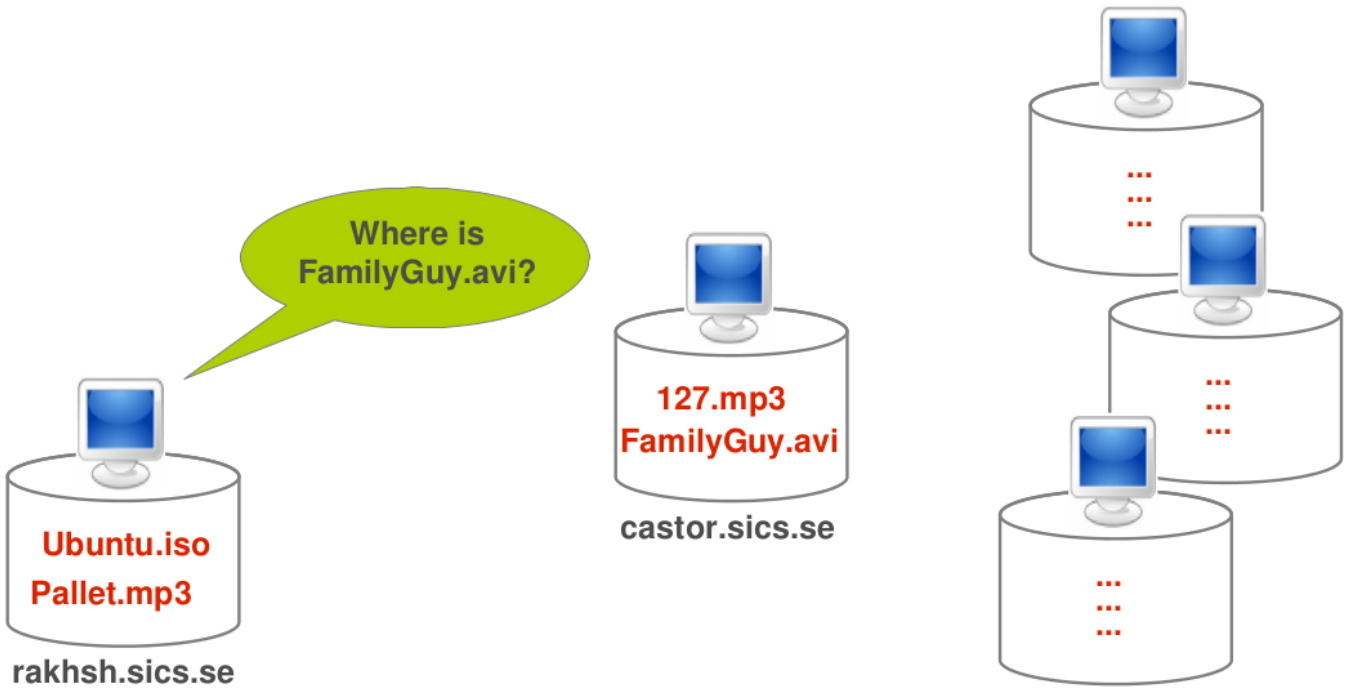
Distributed Hash Tables

Amir H. Payberah
amir@sics.se

Amirkabir University of Technology
(Tehran Polytechnic)

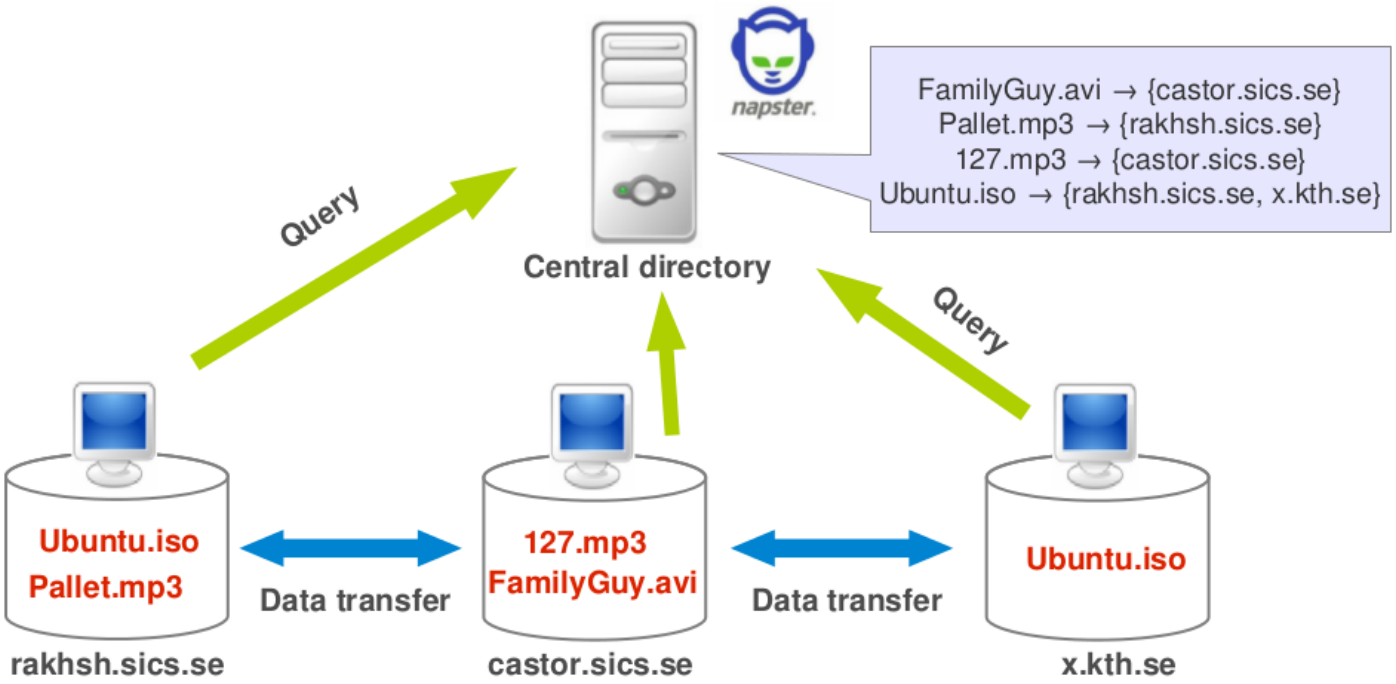
What is the Problem?

What is the Problem?



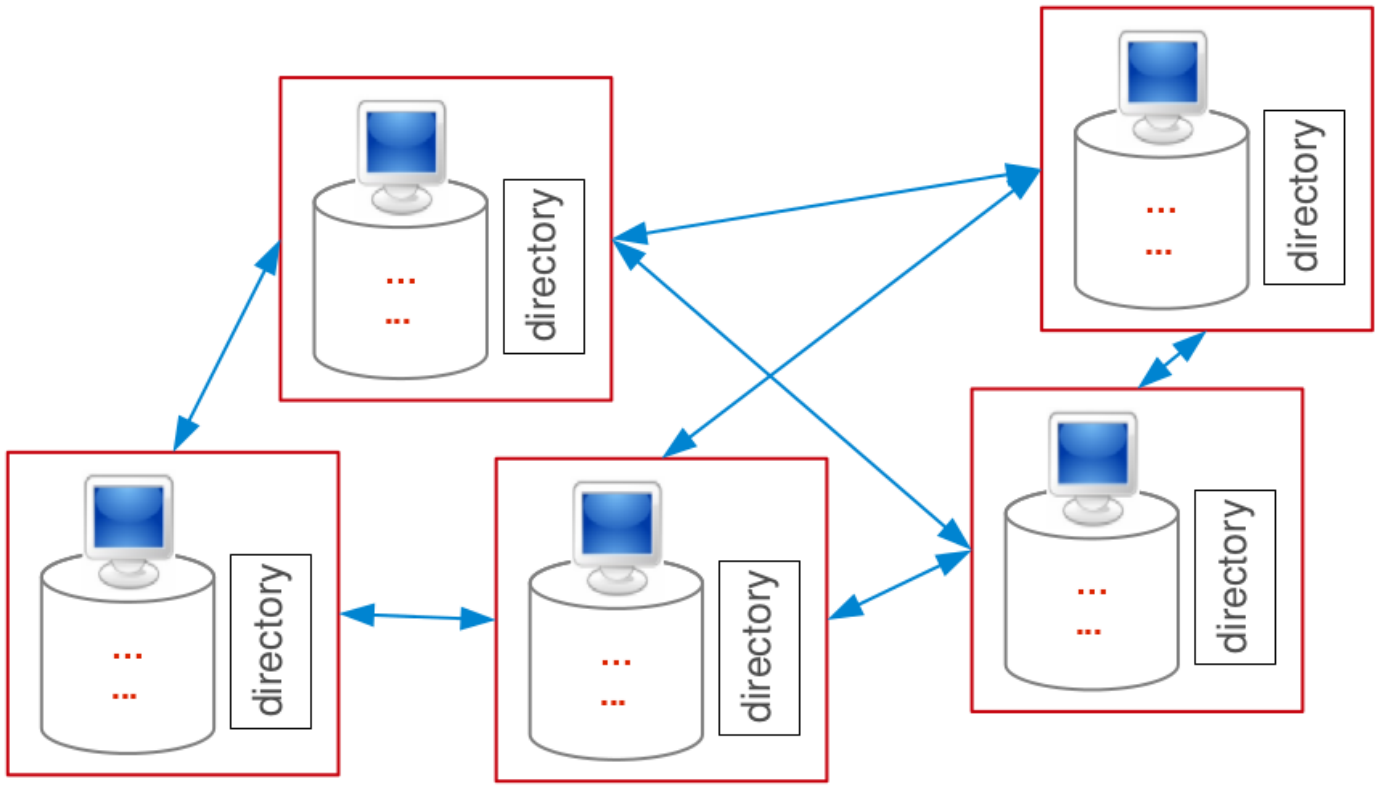
Possible Solutions (1/3)

- Central directory



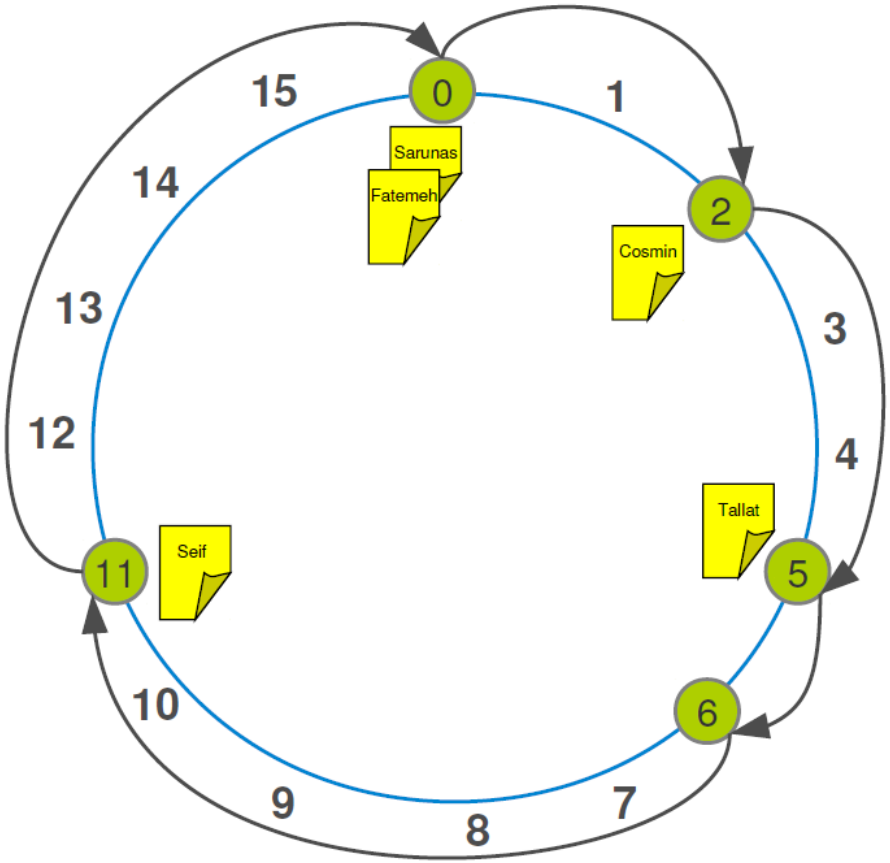
Possible Solutions (2/3)

- Flooding



Possible Solutions (3/3)

- Distributed Hash Table (DHT)



Distributed Hash Table (DHT)

Distributed Hash Table

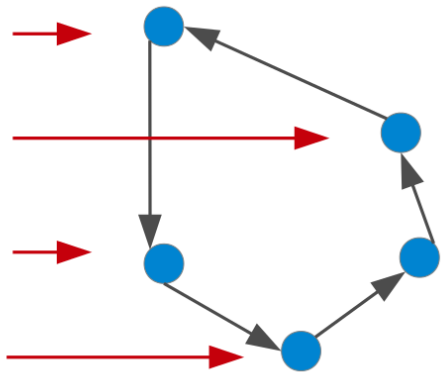
- An ordinary hash-table, which is ...

Key	Value
Fatemeh	Stockholm
Ali	California
Tallat	Islamabad
Cosmin	Bucharest
Seif	Stockholm
Amir	Tehran

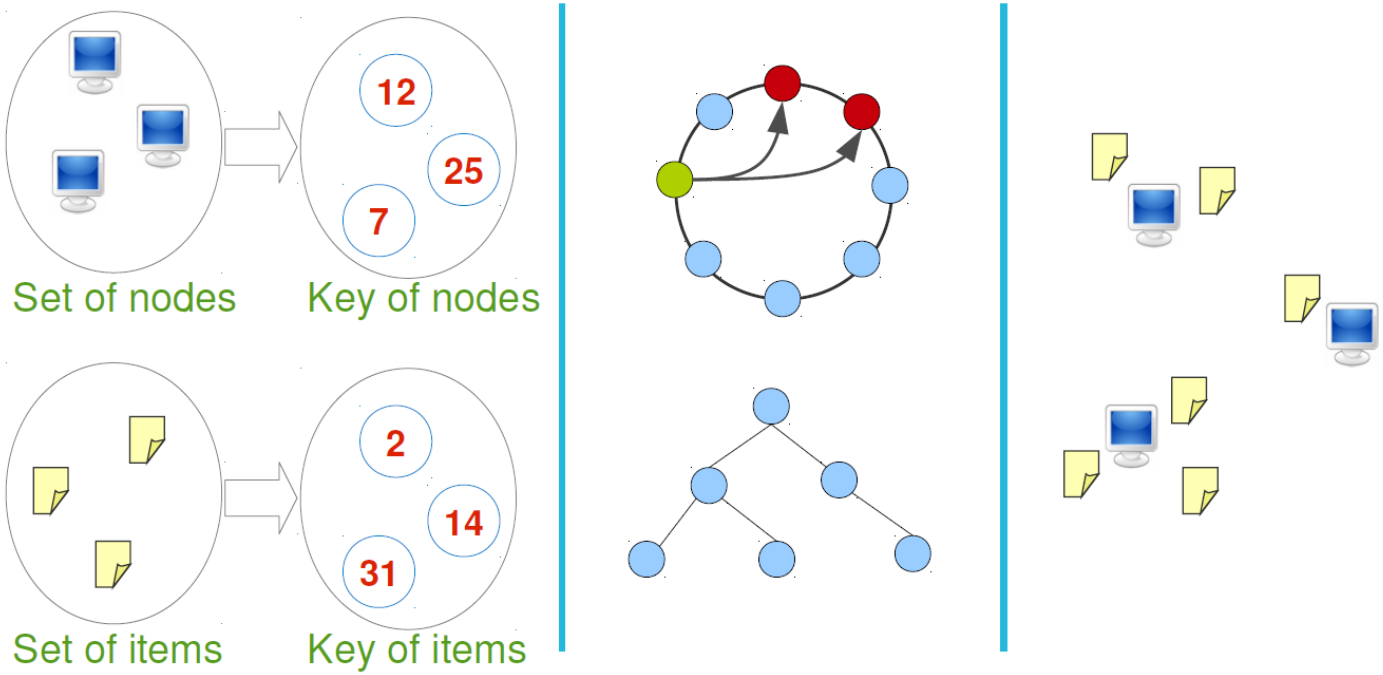
Distributed Hash Table

- An ordinary hash-table, which is distributed.

Key	Value
Fatemeh	Stockholm
Ali	California
Tallat	Islamabad
Cosmin	Bucharest
Seif	Stockholm
Amir	Tehran

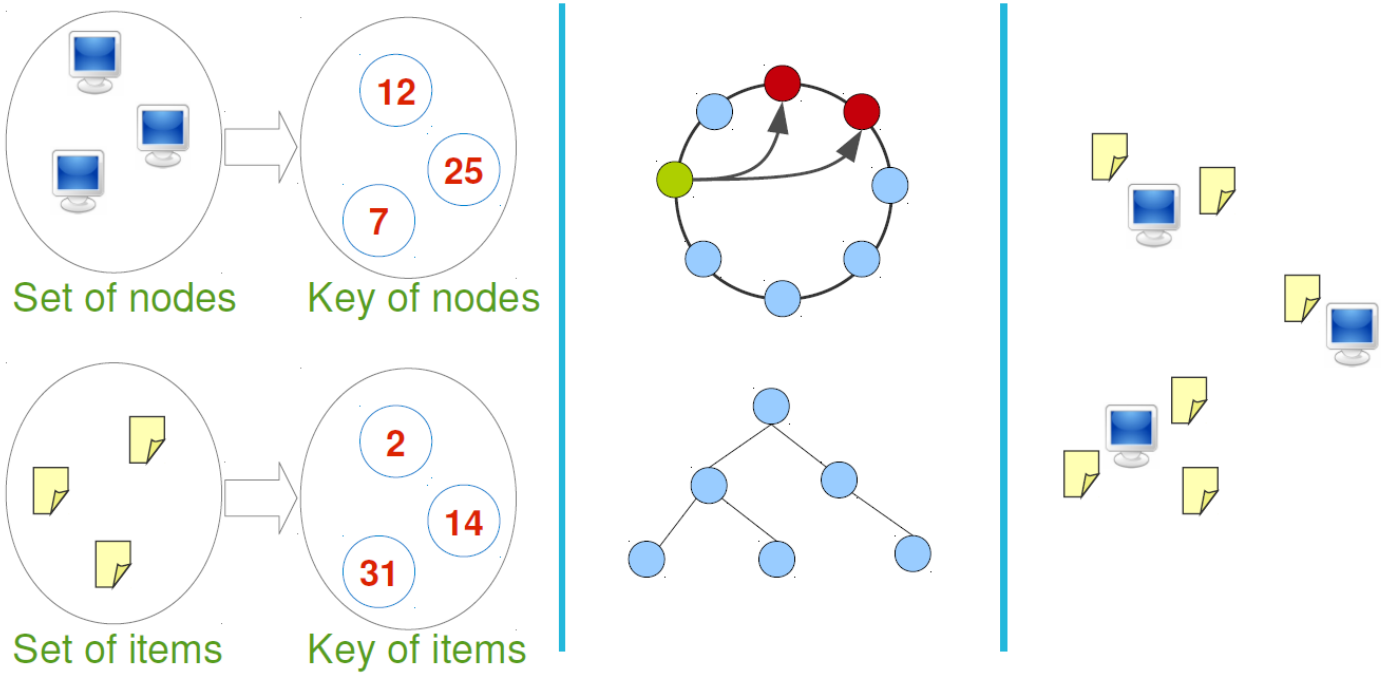


Steps to Build a DHT



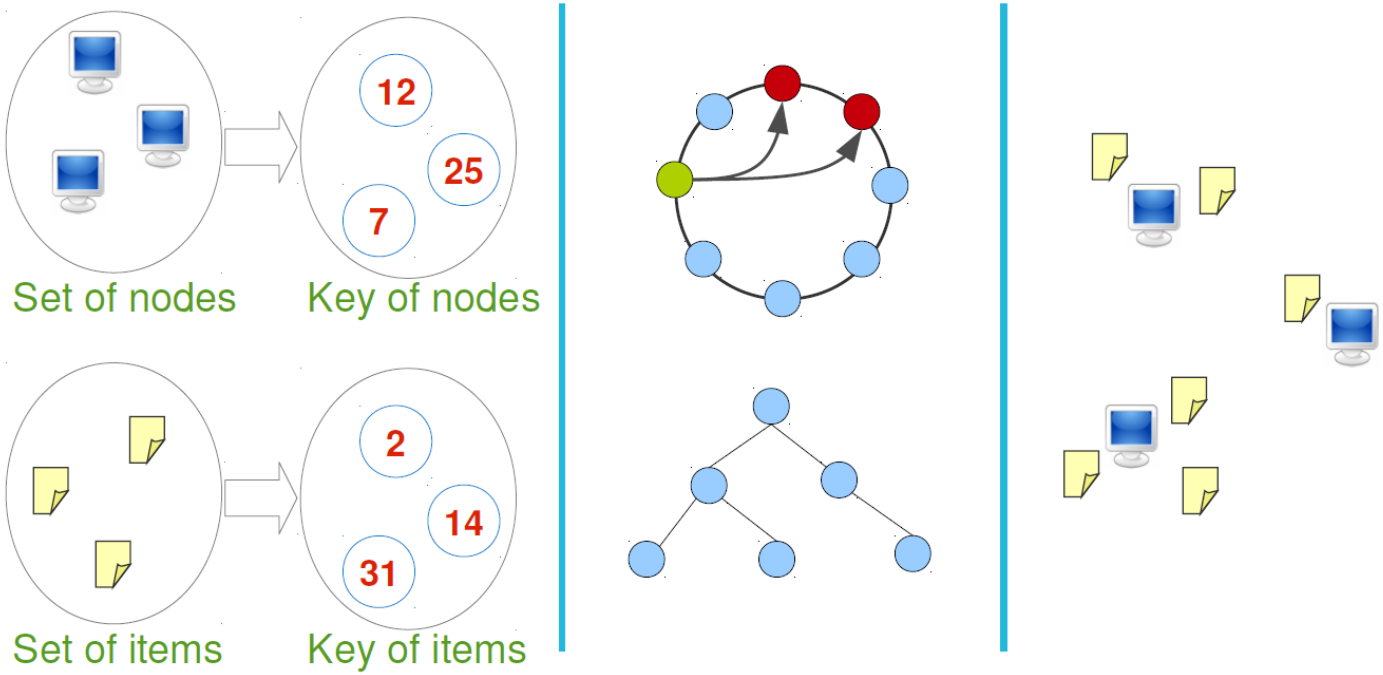
- Step 1: decide on common key space for nodes and values.

Steps to Build a DHT



- Step 1: decide on common key space for nodes and values.
- Step 2: connect the nodes smartly.

Steps to Build a DHT

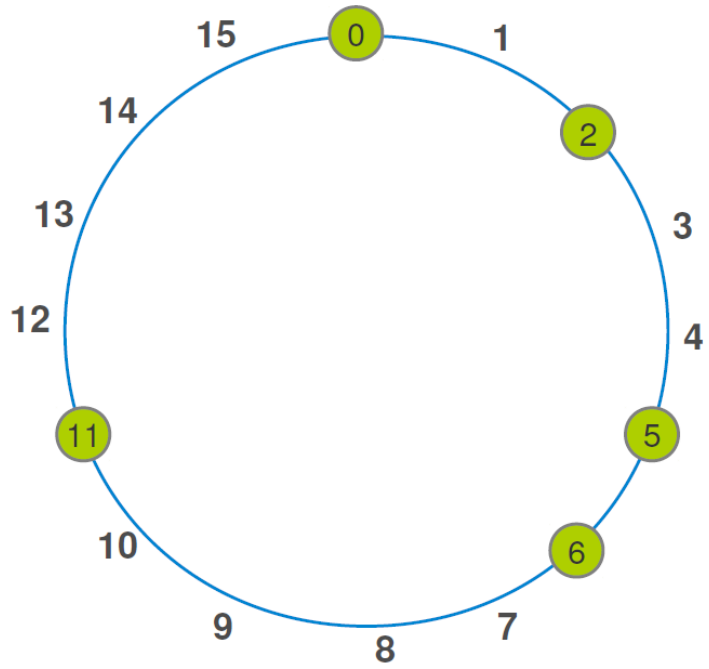


- Step 1: decide on common key space for nodes and values.
- Step 2: connect the nodes smartly.
- Step 3: make a strategy for assigning items to nodes.

Chord: an Example of a DHT

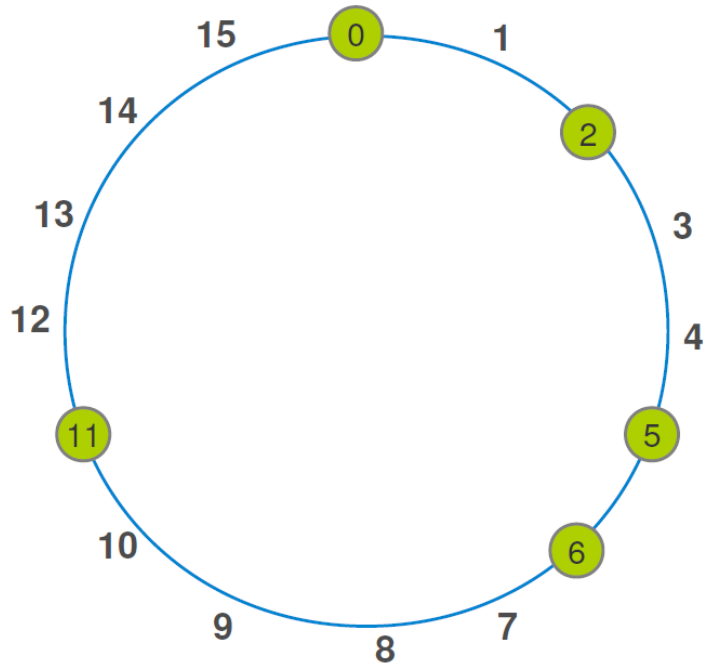
Construct Chord - Step 1

- Use a logical name space, called the id space, consisting of identifiers $\{0, 1, 2, \cdots, N-1\}$.



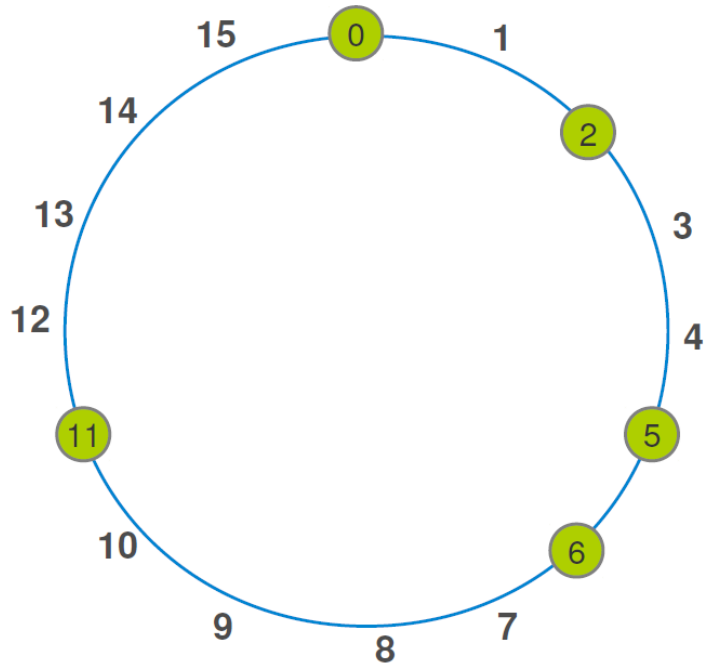
Construct Chord - Step 1

- Use a logical name space, called the id space, consisting of identifiers $\{0, 1, 2, \cdots, N-1\}$.
- Id space is a logical ring modulo N.



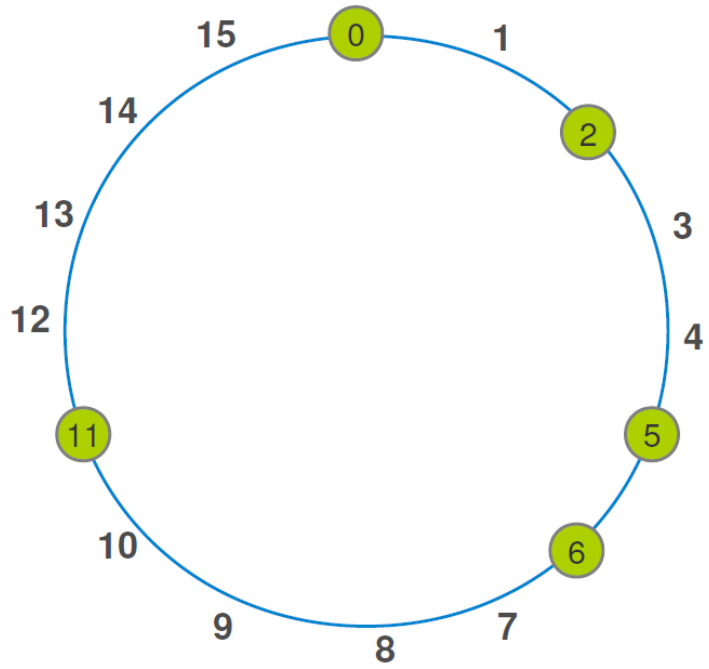
Construct Chord - Step 1

- Use a logical name space, called the id space, consisting of identifiers $\{0, 1, 2, \cdots, N-1\}$.
- Id space is a logical ring modulo N.
- Every node picks a random id though Hash H.



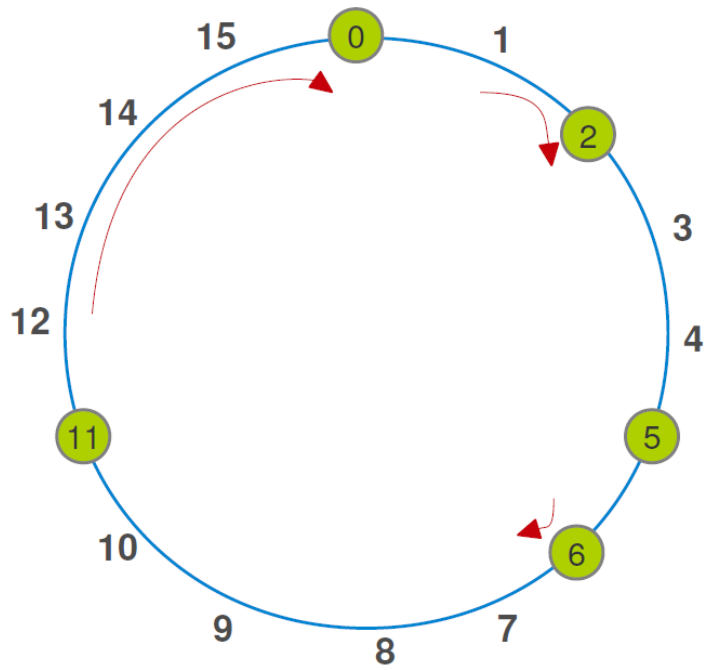
Construct Chord - Step 1

- Use a logical name space, called the id space, consisting of identifiers $\{0, 1, 2, \cdots, N-1\}$.
- Id space is a logical ring modulo N.
- Every node picks a random id though Hash H.
- Example:
  - Space $N = 16\{0, \cdots, 15\}$
  - Five nodes a, b, c, d, e.
  - $H(a) = 6$
  - $H(b) = 5$
  - $H(c) = 0$
  - $H(d) = 11$
  - $H(e) = 2$



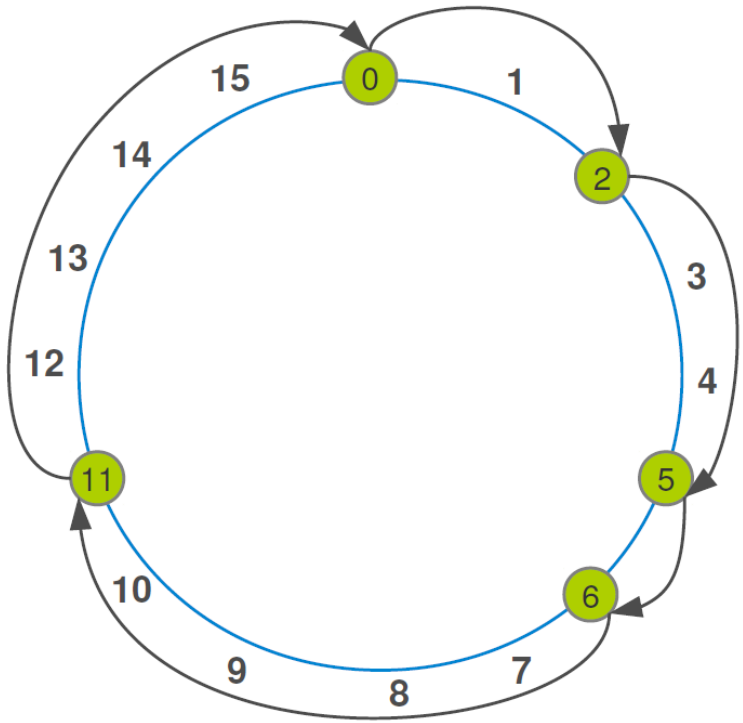
Construct Chord - Step 2 (1/2)

- The successor of an id is the first node met going in clockwise direction starting at the id.
- $succ(x)$: is the first node on the ring with id greater than or equal x.
  - $succ(12) = 0$
  - $succ(1) = 2$
  - $succ(6) = 6$



Construct Chord - Step 2 (2/2)

- Each node points to its successor.
- The successor of a node n is $succ(n+1)$.
  - 0's successor is $succ(1) = 2$.
  - 2's successor is $succ(3) = 5$.
  - 11's successor is $succ(12) = 0$.

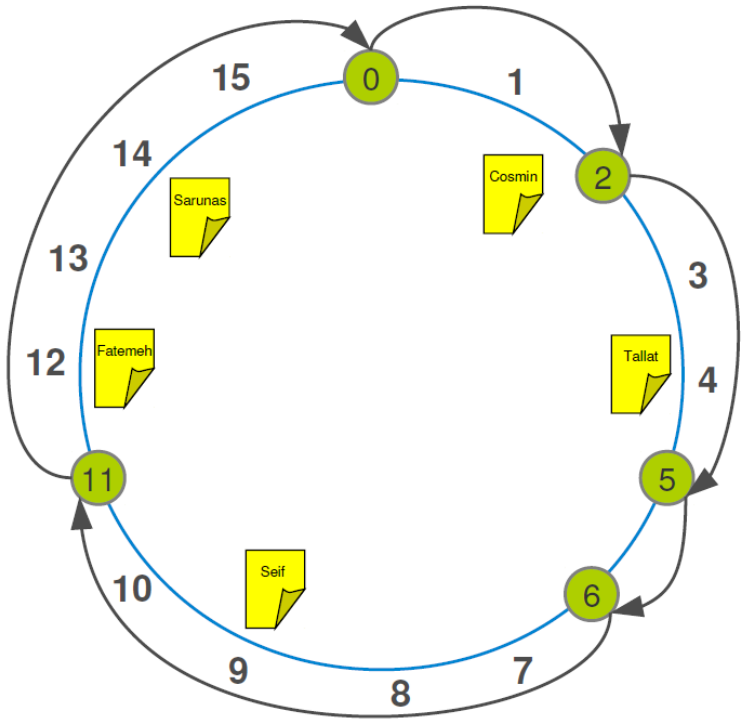


Construct Chord - Step 3

- Where to store data?

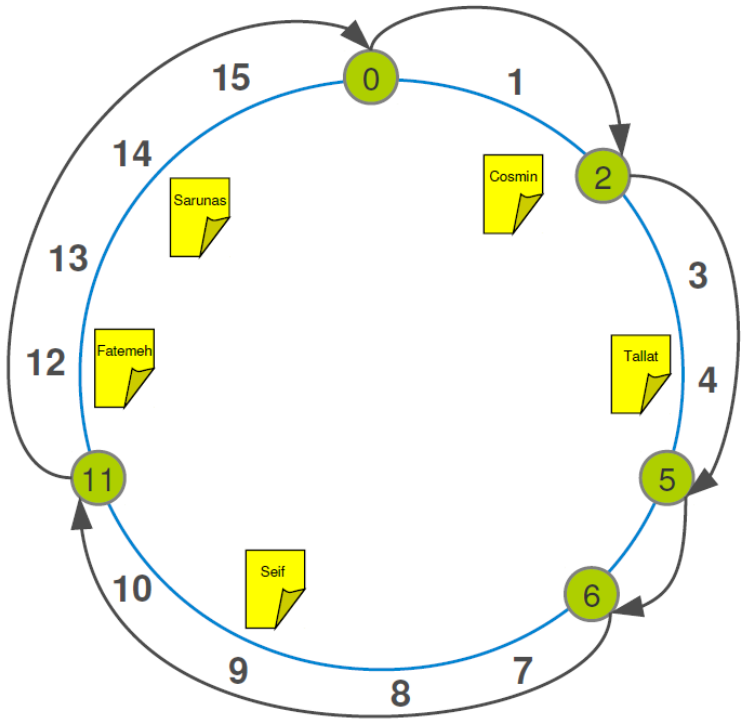
Construct Chord - Step 3

- Where to store data?
- Use globally known hash function H.



Construct Chord - Step 3

- Where to store data?
- Use globally known hash function H.
- Each item $\langle key, value \rangle$ gets identifier $H(key) = k$.
  - Space $N = 16\{0, \cdots, 15\}$
  - Five nodes a, b, c, d, e.
  - $H(Fatemeh) = 12$
  - $H(Cosmin) = 2$
  - $H(Seif) = 9$
  - $H(Sarunas) = 14$
  - $H(Tallat) = 4$

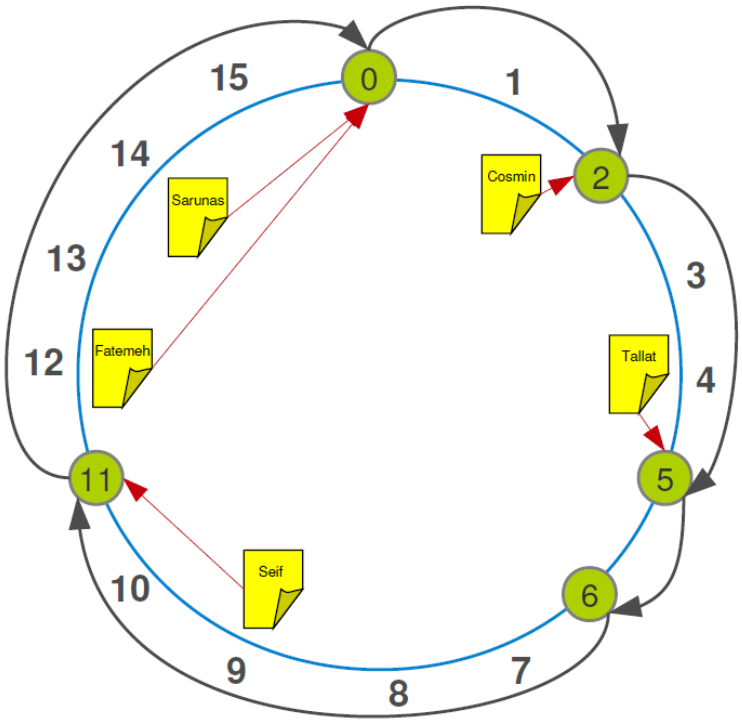


Construct Chord - Step 3

- Where to store data?
- Use globally known hash function H.
- Each item $\langle key, value \rangle$ gets identifier $H(key) = k$.
  - Space $N = 16\{0, \cdots, 15\}$
  - Five nodes a, b, c, d, e.
  - $H(Fatemeh) = 12$
  - $H(Cosmin) = 2$
  - $H(Seif) = 9$
  - $H(Sarunas) = 14$
  - $H(Tallat) = 4$
- Store each item at its successor.

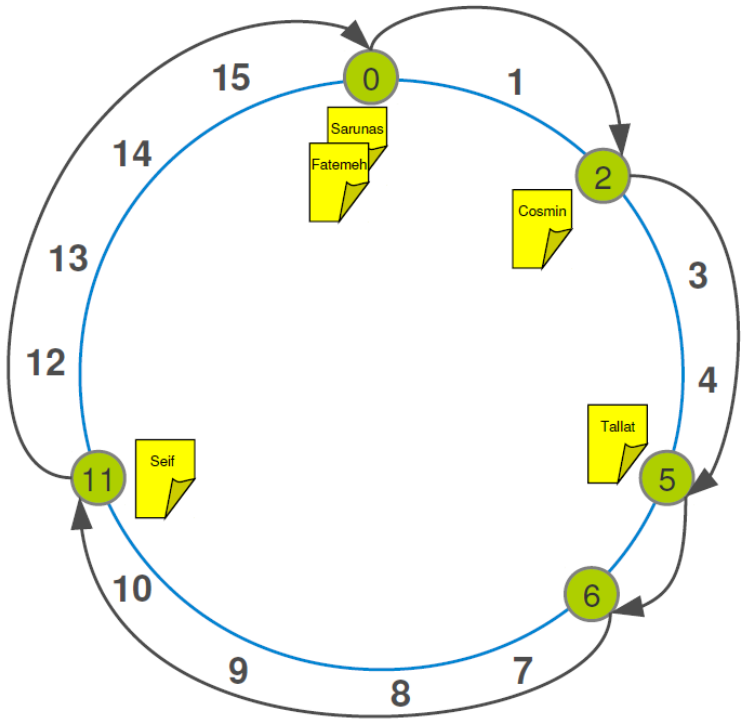
Construct Chord - Step 3

- Where to store data?
- Use globally known hash function H.
- Each item $\langle key, value \rangle$ gets identifier $H(key) = k$.
  - Space $N = 16\{0, \cdots, 15\}$
  - Five nodes a, b, c, d, e.
  - $H(Fatemeh) = 12$
  - $H(Cosmin) = 2$
  - $H(Seif) = 9$
  - $H(Sarunas) = 14$
  - $H(Tallat) = 4$
- Store each item at its successor.



Construct Chord - Step 3

- Where to store data?
- Use globally known hash function H.
- Each item $\langle key, value \rangle$ gets identifier $H(key) = k$.
  - Space $N = 16\{0, \cdots, 15\}$
  - Five nodes a, b, c, d, e.
  - $H(Fatemeh) = 12$
  - $H(Cosmin) = 2$
  - $H(Seif) = 9$
  - $H(Sarunas) = 14$
  - $H(Tallat) = 4$
- Store each item at its successor.



How to Lookup?

Lookup (1/2)

- To lookup a key k:

Lookup (1/2)

- To lookup a key k:
  - Calculate $H(k)$.

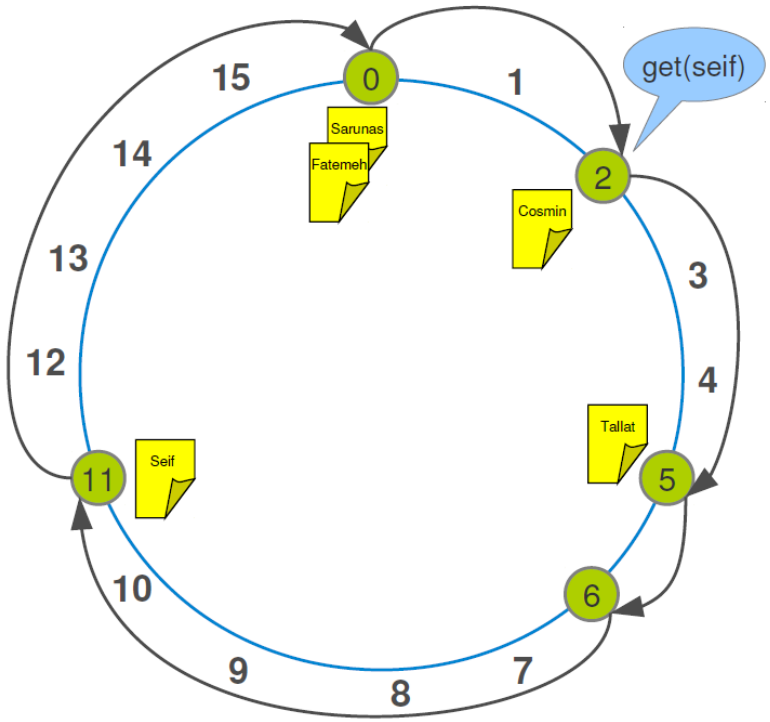
Lookup (1/2)

- To lookup a key k:
  - Calculate $H(k)$.
  - Follow succ pointers until item k is found.

Lookup (1/2)

- To lookup a key k:
  - Calculate $H(k)$.
  - Follow succ pointers until item k is found.
- Example:
  - Lookup Seif at node 2.
  - $H(Seif) = 9$
  - Traverse nodes: 2, 5, 6, 11
  - Return Stockholm to initiator

Key	Value
Seif	Stockholm



Lookup (2/2)

Algorithm 1 Ask node n to find the successor of id

```
1: procedure n.findSuccessor(id)
2: if pred ≠ ∅ and id ∈ (pred, n] then
3:     return n
4: else if id ∈ (n, succ] then
5:     return succ
6: else// forward the query around the circle
7:     return succ.findSuccessor(id)
8: end if
9: end procedure
```

- $(a, b]$ the segment of the ring moving clockwise from but not including a until and including b.
- n.`foo(.)` denotes an RPC of `foo(.)` to node n.
- n.`bar` denotes and RPC to fetch the value of the variable `bar` in node n.

Put and Get

Algorithm 2 Store $value$ with key id in the DHT

1: **procedure** n.put($id, value$)
2: n = findSuccessor(id)
3: s.store($id, value$)
4: **end procedure**

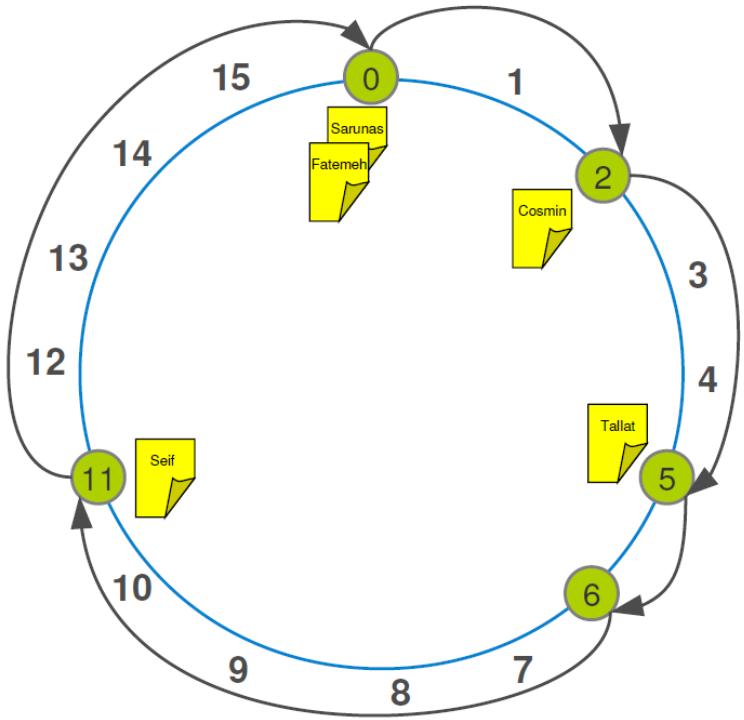
Algorithm 3 Retrieve the value of the key id from the DHT

1: **procedure** n.get(id)
2: n = findSuccessor(id)
3: return s.retrieve(id)
4: **end procedure**

Any Improvement?

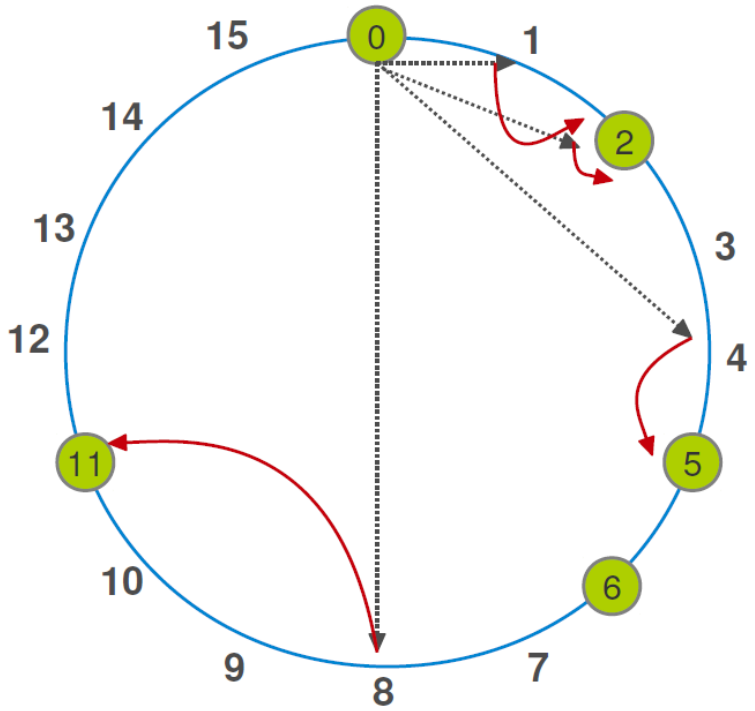
Improvement

- Speeding up lookups.
- If only the successor pointers are used:
  - Worst case lookup time is N, for N nodes.



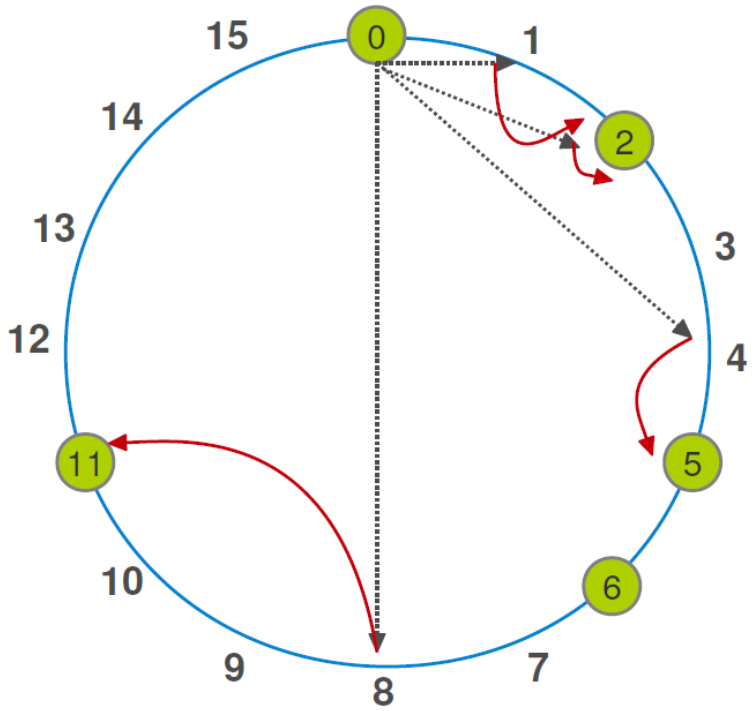
Speeding up Lookups (1/2)

- Finger/routing table:
  - Point to $succ(n+1)$
  - Point to $succ(n+2)$
  - Point to $succ(n+4)$
  - ...
  - Point to $succ(n+2^{M-1})$ ($N = 2^M$, N: the id space size)



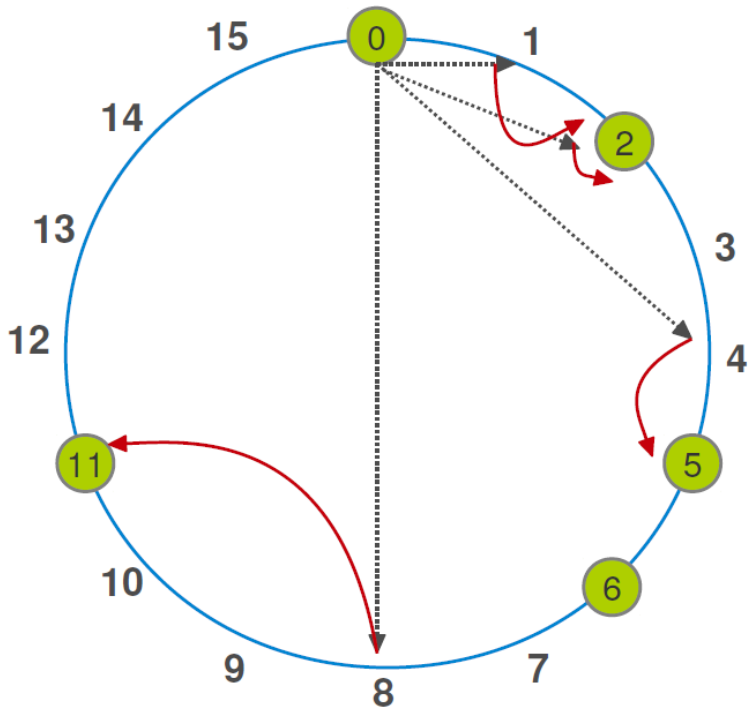
Speeding up Lookups (1/2)

- Finger/routing table:
  - Point to $succ(n+1)$
  - Point to $succ(n+2)$
  - Point to $succ(n+4)$
  - ...
  - Point to $succ(n+2^{M-1})$ ($N = 2^M$, N: the id space size)
- Distance always halved to the destination.



Speeding up Lookups (2/2)

- Every node n knows $succ(n + 2^{i-1})$ for $i = 1, \cdots, M$.
- Size of routing tables is logarithmic:
  - Routing table size: M, where $N = 2^M$
  - Routing entries $= log_2(N)$.
  - Example: $Log_2(1000000) \approx 20$



Lookup Improvement (1/3)

Algorithm 4 Ask node n to find the successor of id

```
1: procedure n.findSuccessor(id)
2: if pred ≠ ∅ and id ∈ (pred, n] then
3:     return n
4: else if id ∈ (n, succ] then
5:     return succ
6: else// forward the query around the circle
7:     return succ.findSuccessor(id)
8: end if
9: end procedure
```

Lookup Improvement (2/3)

Algorithm 5 Ask node n to find the successor of id

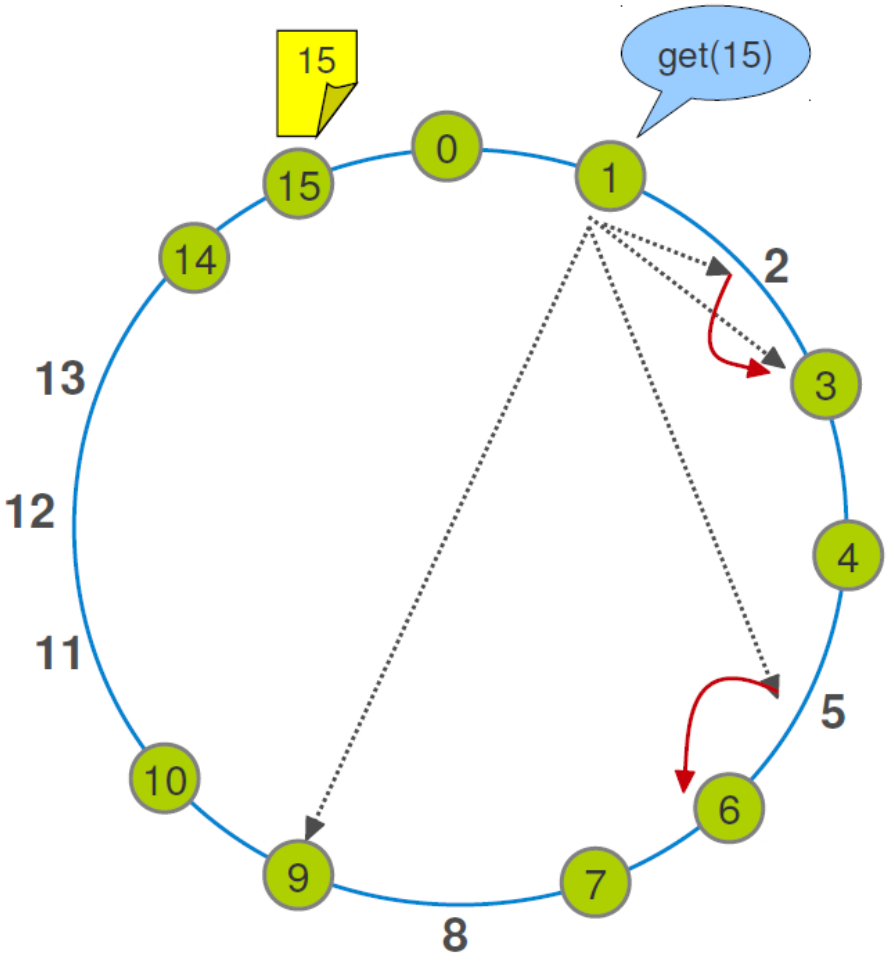
```
1: procedure n.findSuccessor(id)
2: if pred ≠ ∅ and id ∈ (pred, n] then
3:     return n
4: else if id ∈ (n, succ] then
5:     return succ
6: else// forward the query around the circle
7:     p ← closestPrecedingNode(id)
8:     return p.findSuccessor(id)
9: end if
10: end procedure
```

Lookup Improvement (3/3)

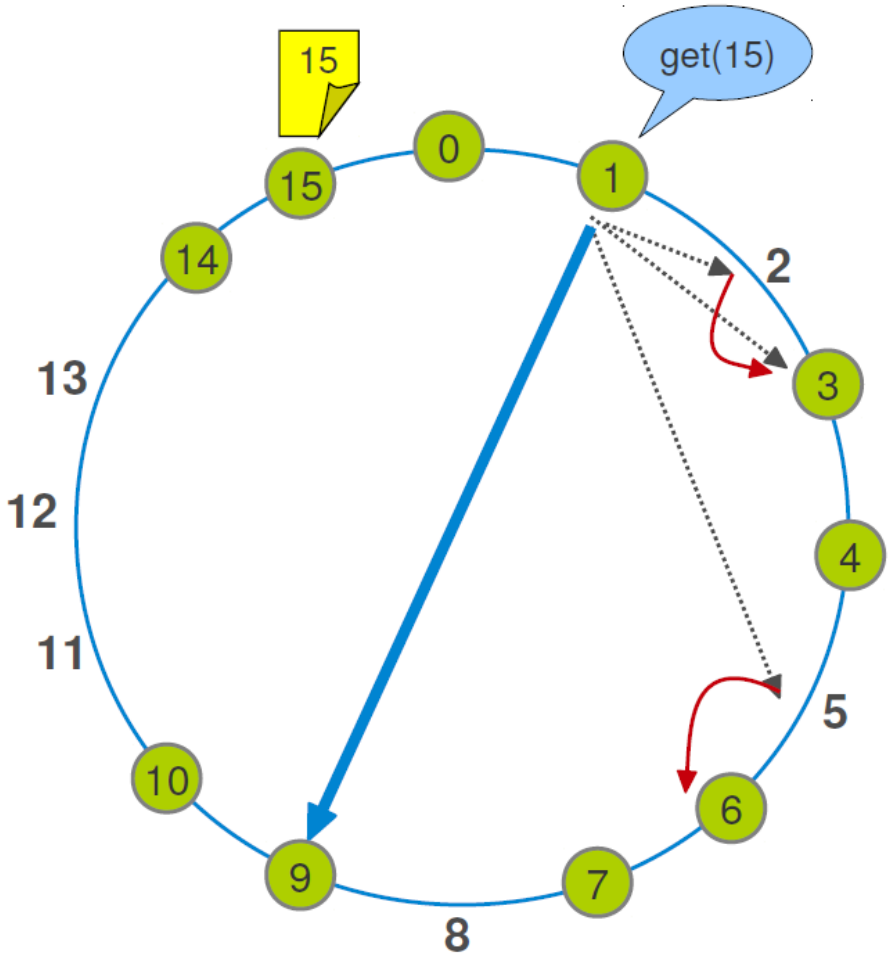
Algorithm 6 Search locally for the highest predecessor of id

```
1: procedure closestPrecedingNode(id)
2: for i = m downto 1 do
3:     if finget[i] ∈ (n, id) then
4:         return finger[i]
5:     end if
6: end for
7: end procedure
```

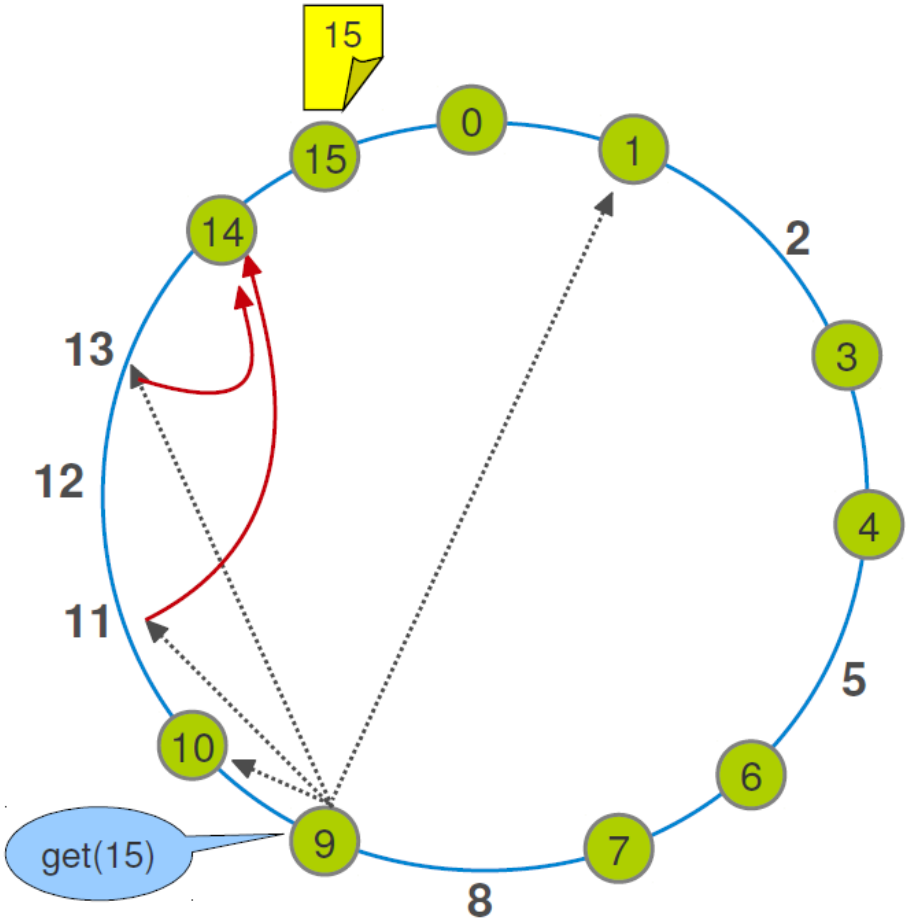
Lookups (1/7)



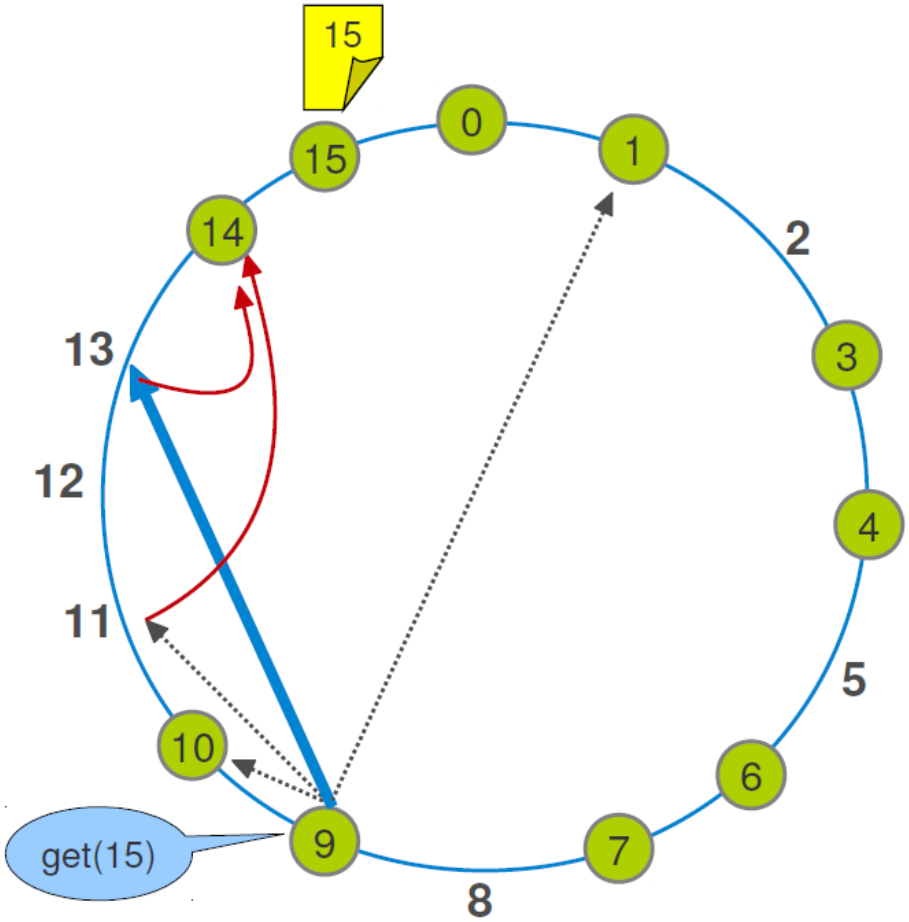
Lookups (2/7)



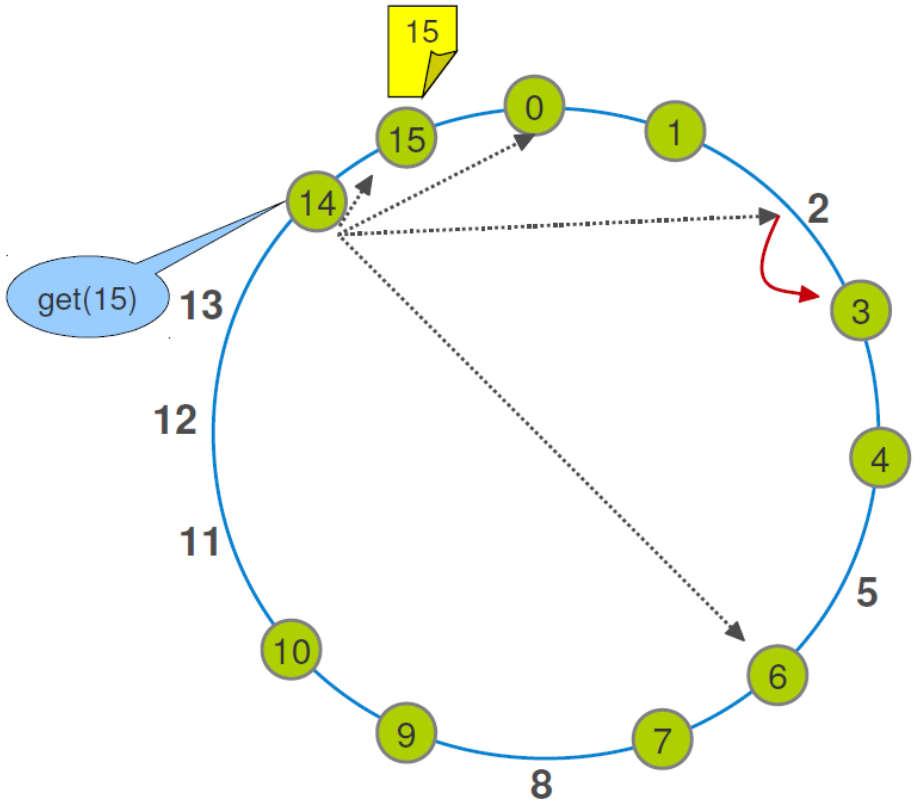
Lookups (3/7)



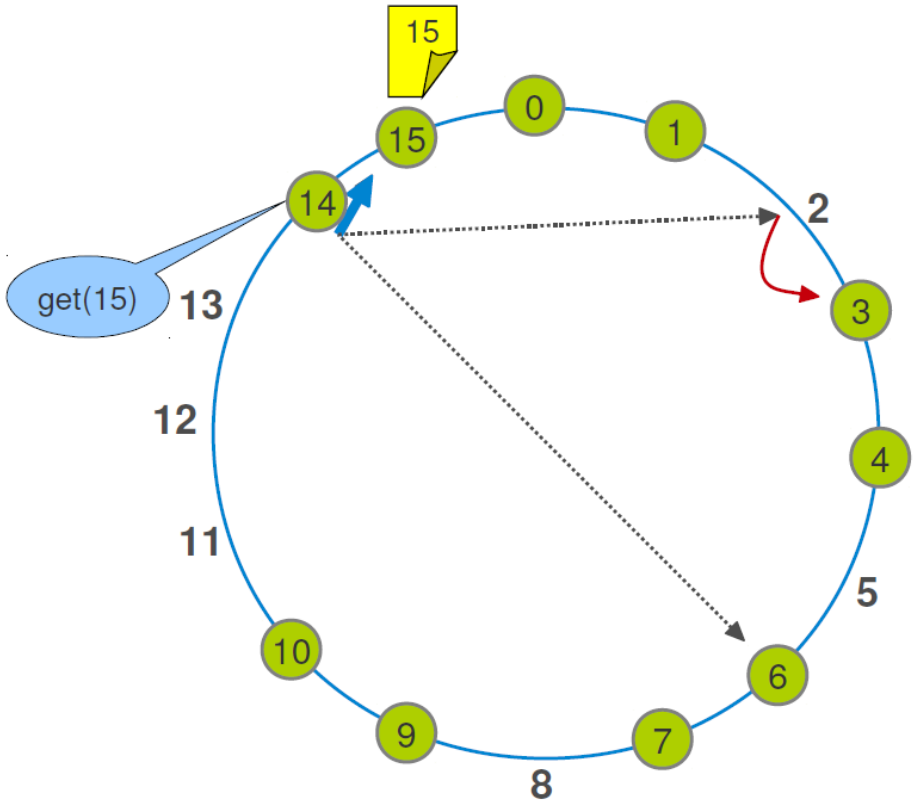
Lookups (4/7)



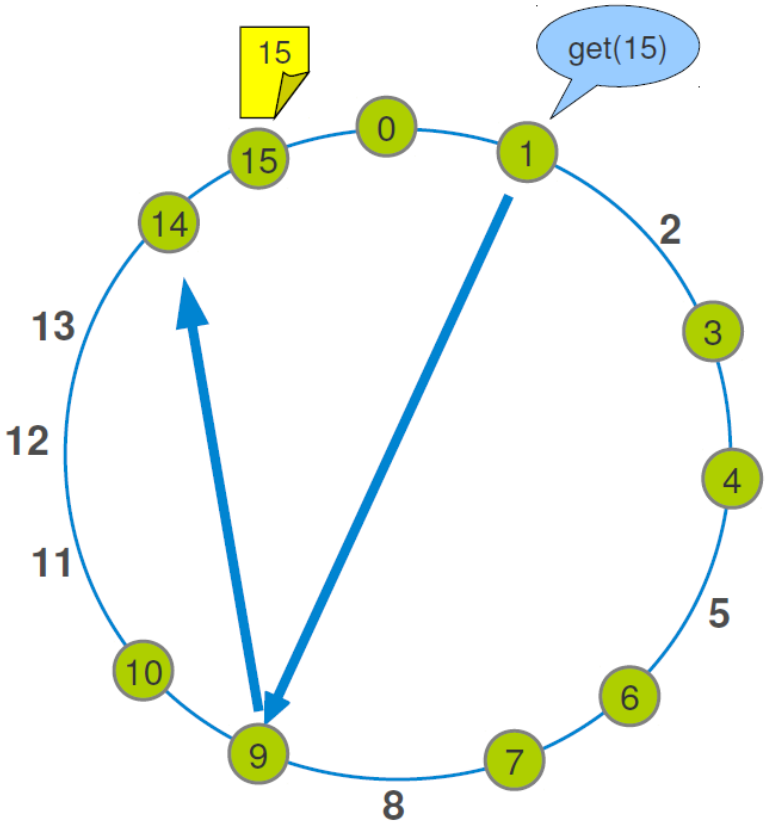
Lookups (5/7)



Lookups (6/7)



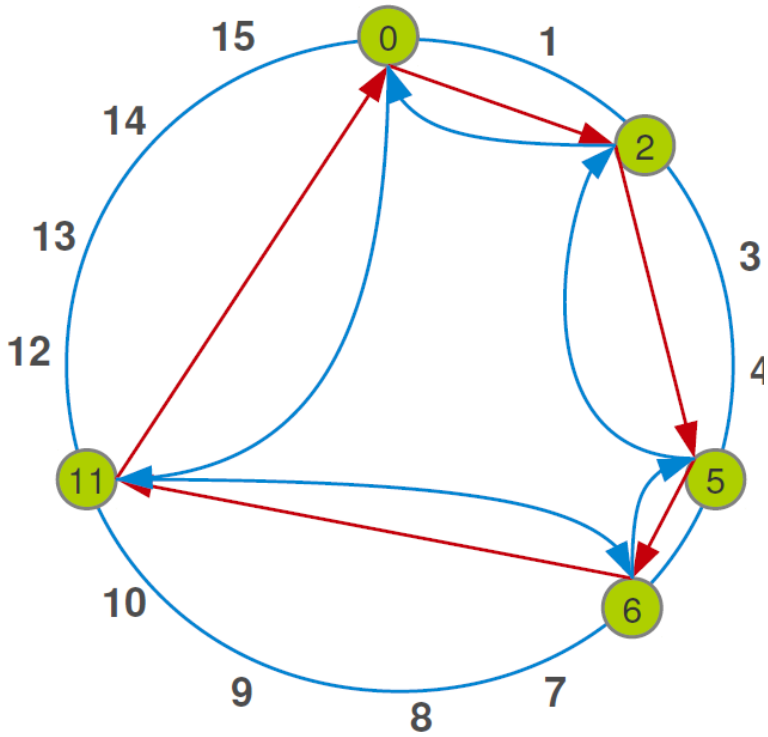
Lookups (7/7)



How to Maintain the Ring?

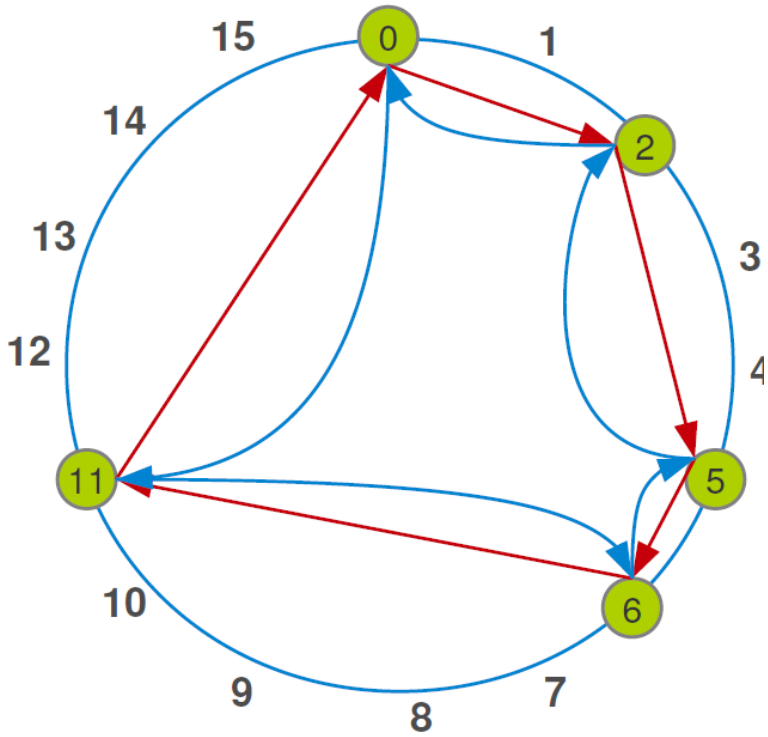
Periodic Stabilization (1/2)

- In Chord, in addition to the successor pointer, every node has a predecessor pointer.
  - Predecessor of node n is the first node met in anti-clockwise direction starting at n.



Periodic Stabilization (1/2)

- In Chord, in addition to the successor pointer, every node has a predecessor pointer.
  - Predecessor of node n is the first node met in anti-clockwise direction starting at n.
- Periodic stabilization is used to make pointers eventually correct.
  - Pointing $succ$ to closest alive successor.
  - Pointing $pred$ to closest alive predecessor.



Periodic Stabilization (2/2)

Algorithm 7 Periodically at n

1: **procedure** n.stabilize()
2: $v \leftarrow succ.pred$
3: **if** $v \neq \emptyset$ **and** $v \in (n, succ]$ **then**
4: $succ \leftarrow v$
5: **end if**
6: **send** notify(n) to $succ$
7: **end procedure**

Algorithm 8 Upon receipt a $notify(p)$ at node m

1: **on receive** $\langle \text{NOTIFY} \mid p \rangle$ **from** n **do**
2: **if** $pred = \emptyset$ **or** $p \in (pred, m]$ **then**
3: $pred \leftarrow p$
4: **end if**
5: **end event**

Handling Join

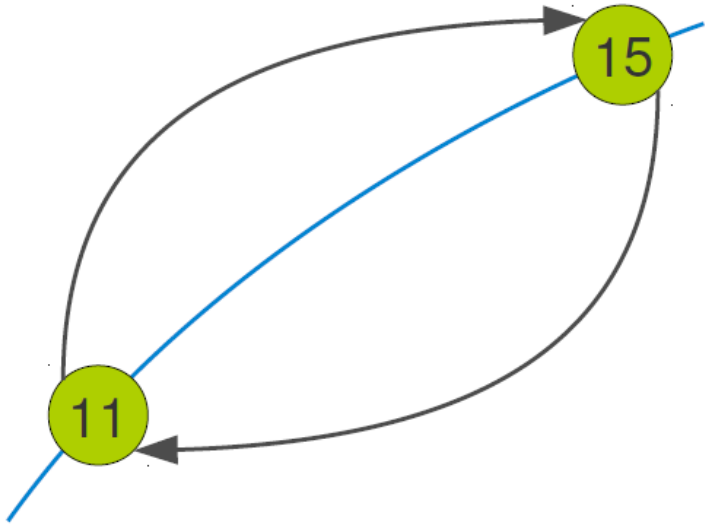
Handling Join

- When n joins:
  - Find n's successor with $lookup(n)$.
  - Set $succ$ to n's successor.
  - Stabilization fixes the rest.

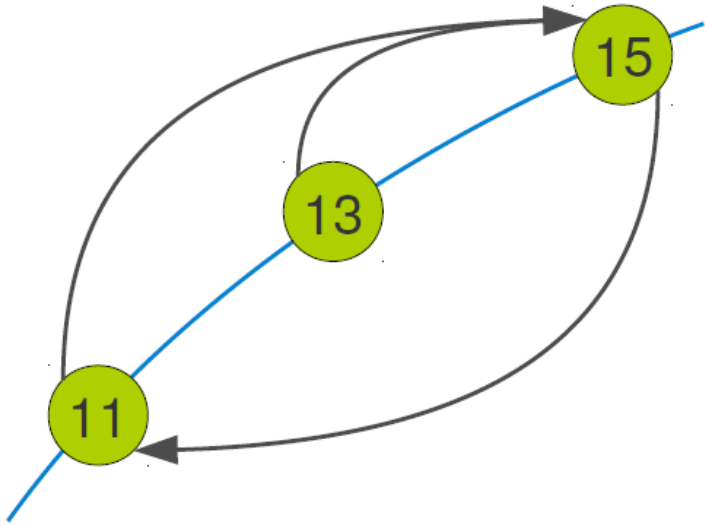
Algorithm 9 Join a Chord ring containing node m

```
1: procedure n.join(m)
2: pred ← ∅
3: succ ← m.findSuccessor(n)
4: end procedure
```

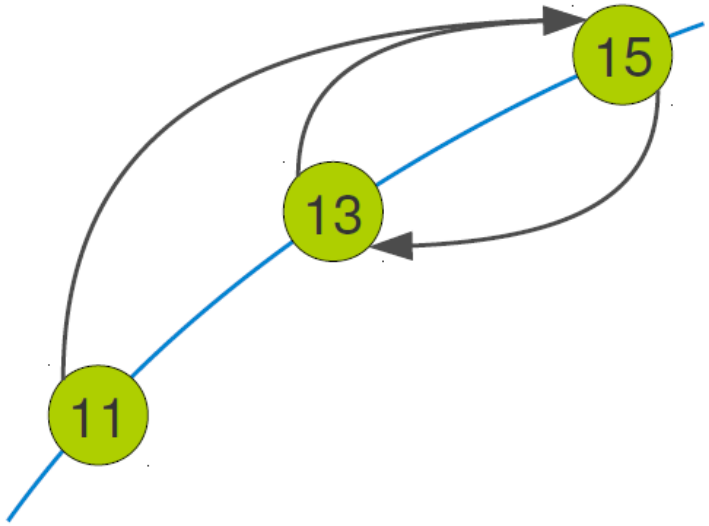
Join (1/5)



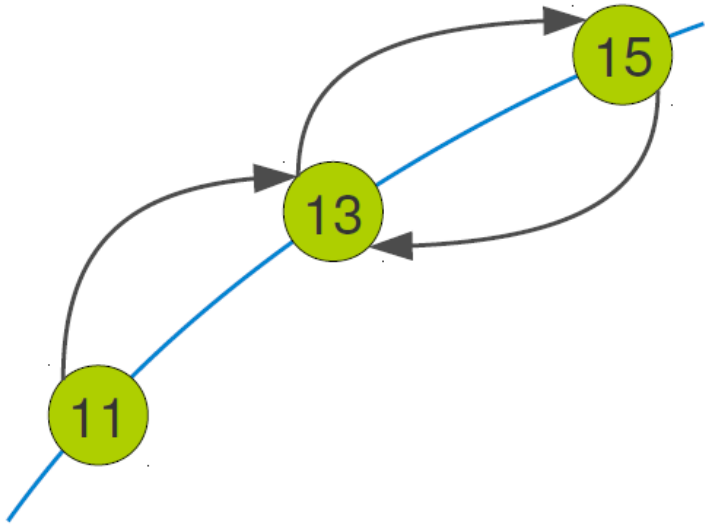
Join (2/5)



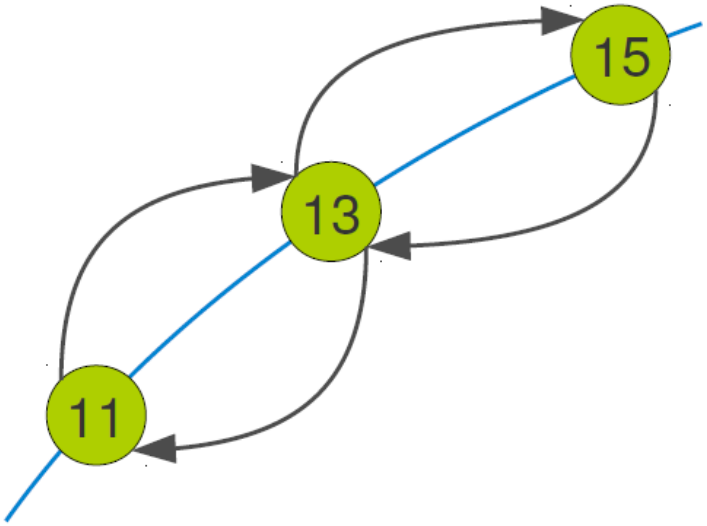
Join (3/5)



Join (4/5)



Join (5/5)



Fix Fingers (1/4)

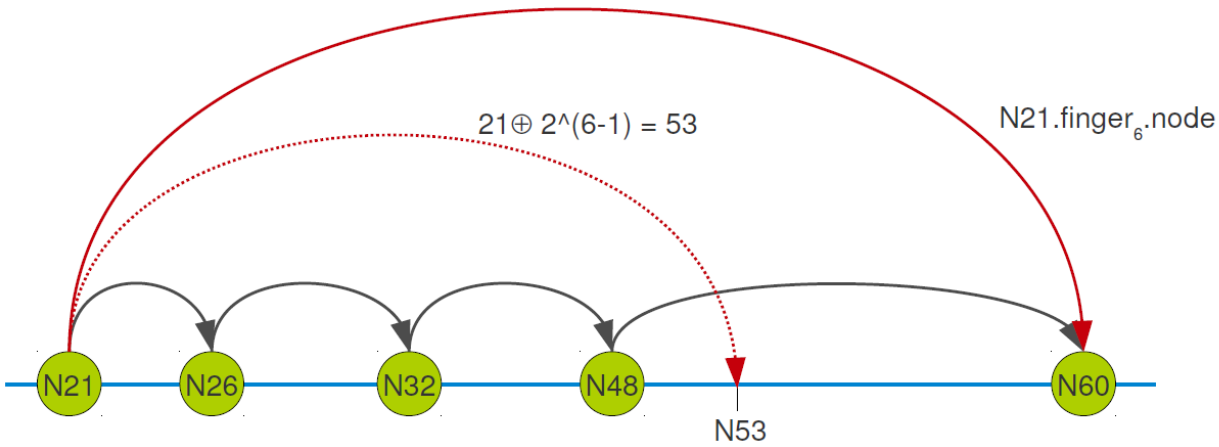
- Periodically refresh finger table entries, and store the index of the next finger to fix.

Algorithm 10 When receiving $notify(p)$ at n

```
1: procedure n.fixFingers()
2: next ← next + 1
3: if next > m then
4:     next ← 1
5: end if
6: finger[next] ← findSuccessor(n ⊕ 2^(next−1))
7: end procedure
```

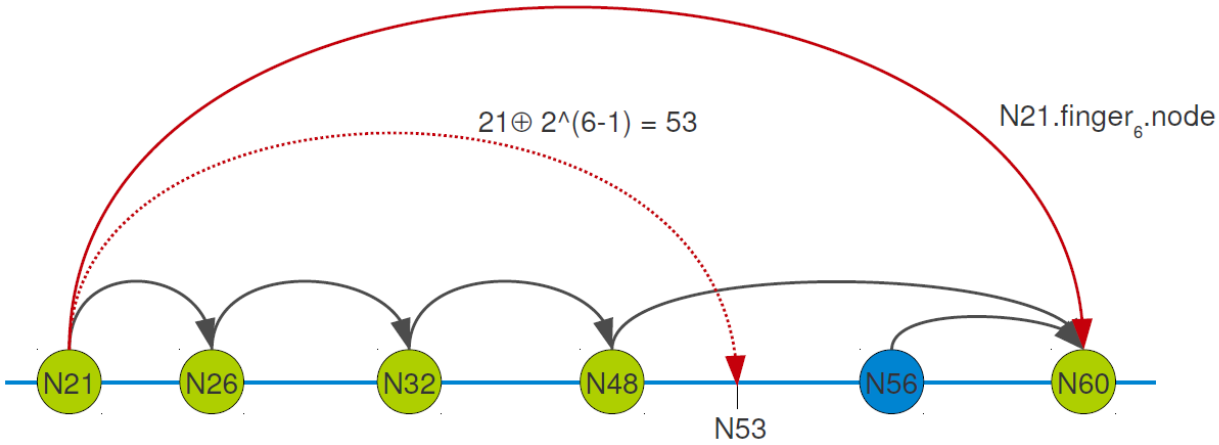
Fix Fingers (2/4)

- Current situation: $succ(N48) = N60$
- $succ(21 \oplus 2^5) = succ(53) = N60$



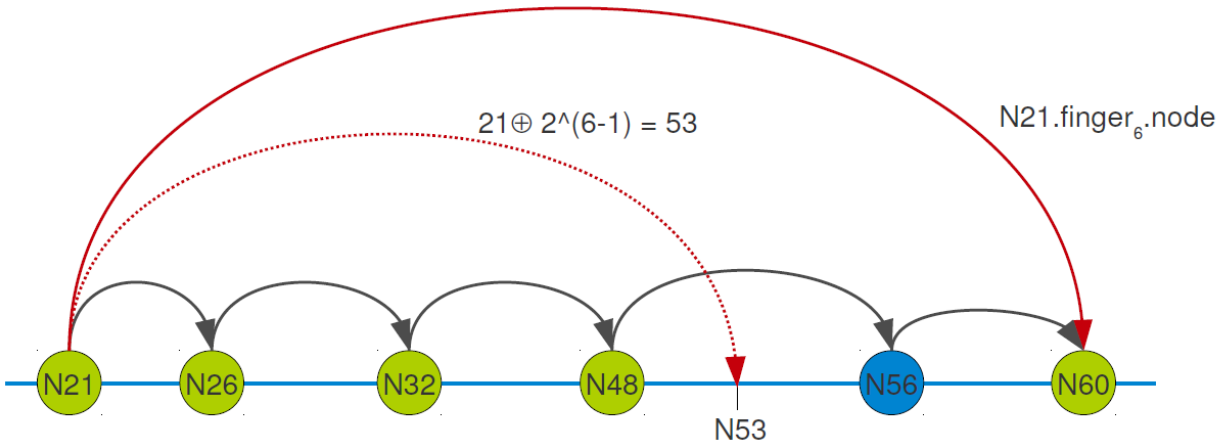
Fix Fingers (3/4)

- $succ(21 \oplus 2^5) = succ(53) = ?$
- New node $N56$ joins and stabilizes successor pointer.
- Finger 6 of node $N21$ is wrong now.
- $N21$ eventually try to fix finger 6 by looking up 53 which stops at N48.



Fix Fingers (4/4)

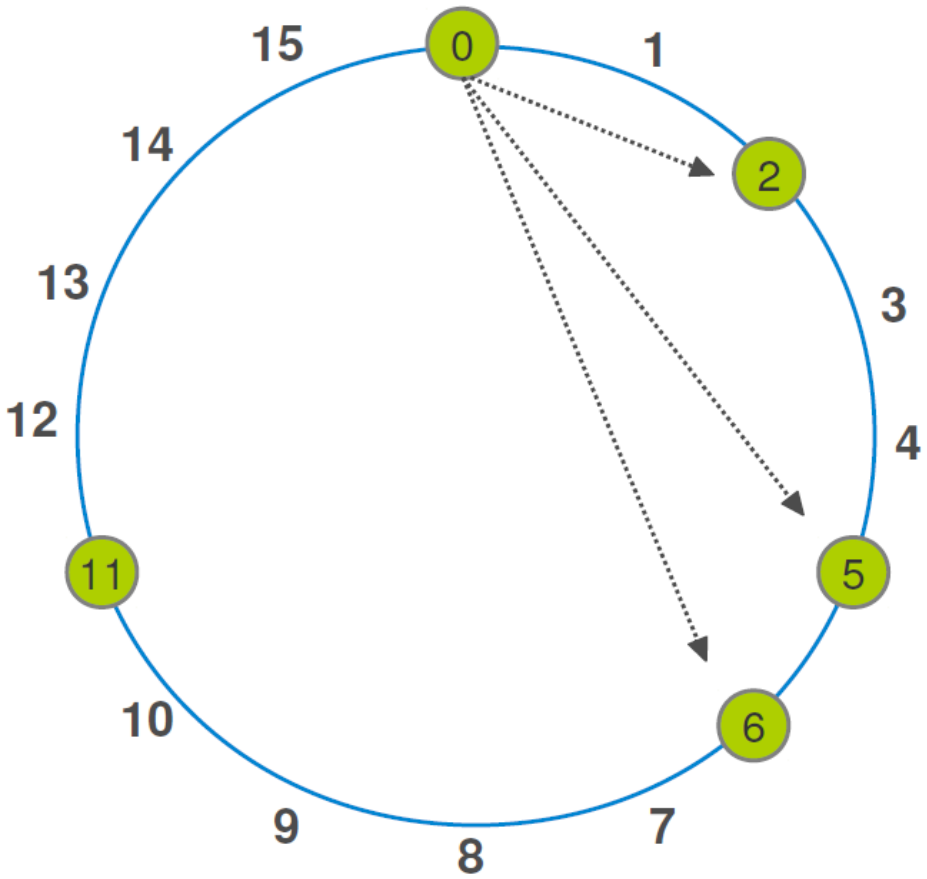
- $succ(21 \oplus 2^5) = succ(53) = N56$
- $N48$ will eventually stabilize its successor.
- This means the ring is correct now.



Handling Failure

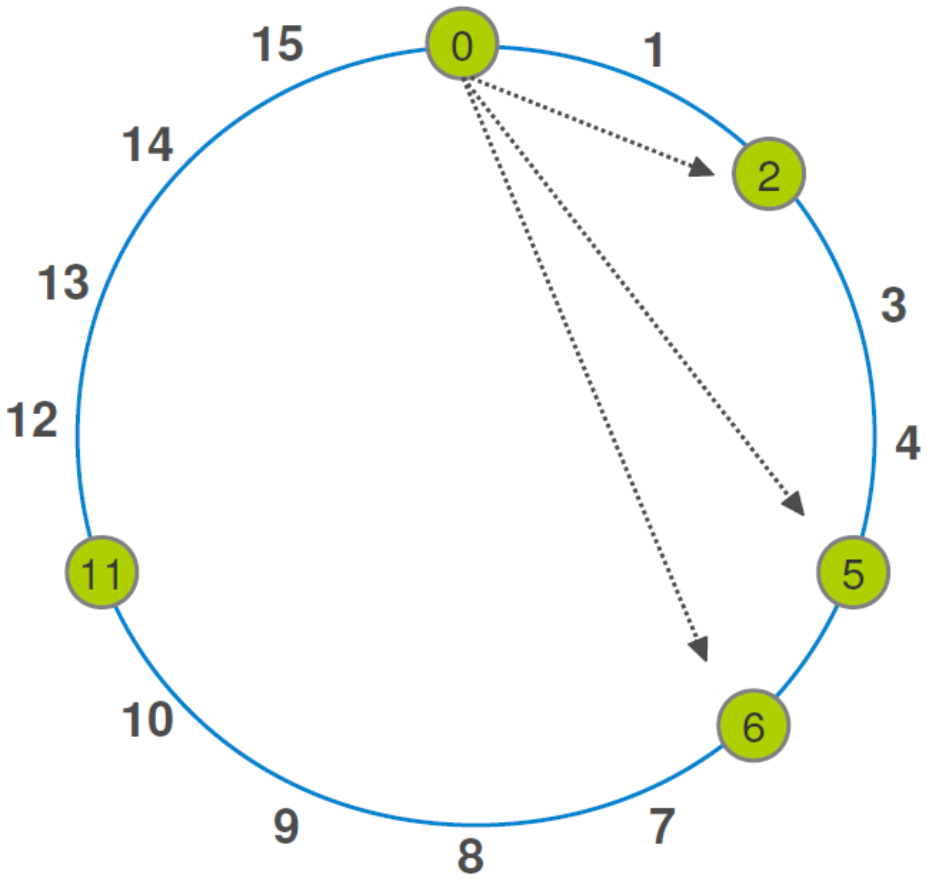
Successor List (1/2)

- A node has a successors list of size r containing the immediate r successors.
  - $succ(n+1)$
  - $succ(succ(n+1)+1)$
  - $succ(succ(succ(n+1)+1)$



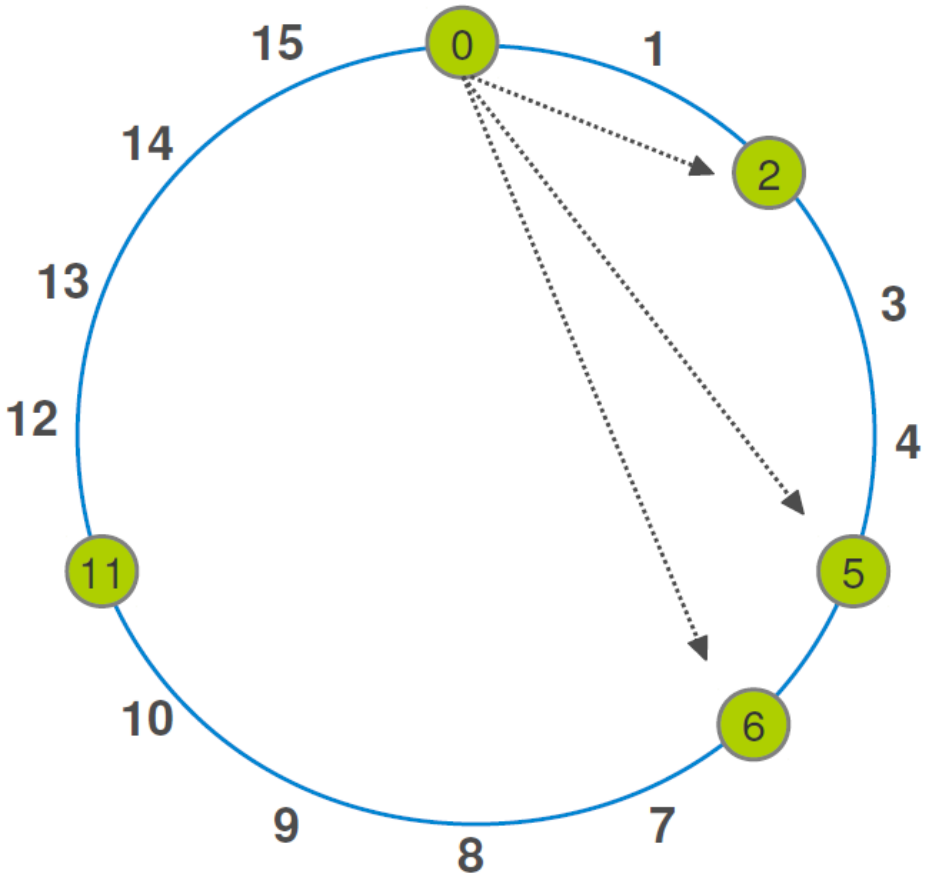
Successor List (1/2)

- A node has a successors list of size r containing the immediate r successors.
  - $succ(n+1)$
  - $succ(succ(n+1)+1)$
  - $succ(succ(succ(n+1)+1)$
- How big should r be?



Successor List (1/2)

- A node has a successors list of size r containing the immediate r successors.
  - $succ(n+1)$
  - $succ(succ(n+1)+1)$
  - $succ(succ(succ(n+1)+1)$
- How big should r be? $log_2(N)$



Successor List (2/2)

Algorithm 11 Join a Chord ring containing node m

1: **procedure** n.join(m)
2: $pred \leftarrow \emptyset$
3: $succ \leftarrow m$.findSuccessor(n)
4: updateSuccesorList($succ$.successorList)
5: **end procedure**

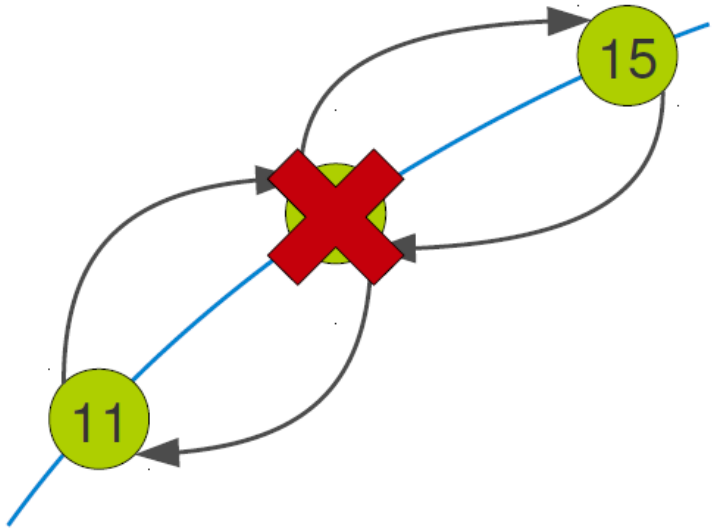
Algorithm 12 Periodically at n

1: **procedure** n.stabilize()
2: $succ \leftarrow$ find first alive node in successor list
3: $v \leftarrow succ.pred$
4: **if** $v \neq \emptyset$ **and** $v \in (n, succ]$ **then**
5: $succ \leftarrow v$
6: **end if**
7: **send** notify(n) to $succ$ updateSuccessorList($succ$.successorList)
8: **end procedure**

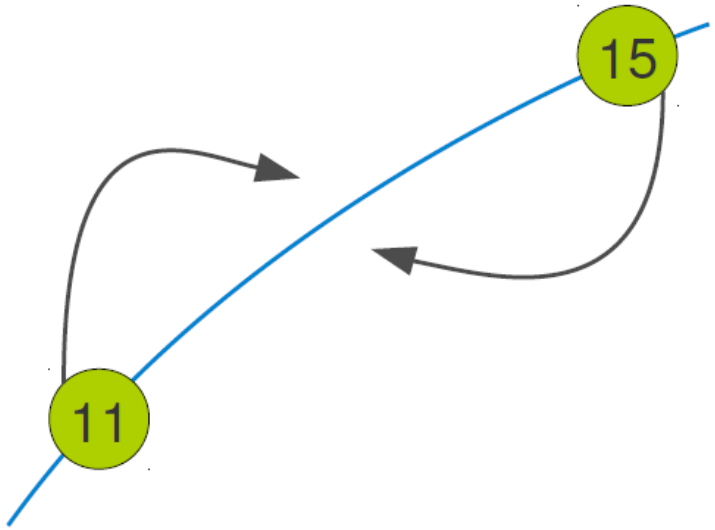
Dealing with Failure

- Periodic stabilization
- If successor fails: replace with closest alive successor
- If predecessor fails: set predecessor to nil

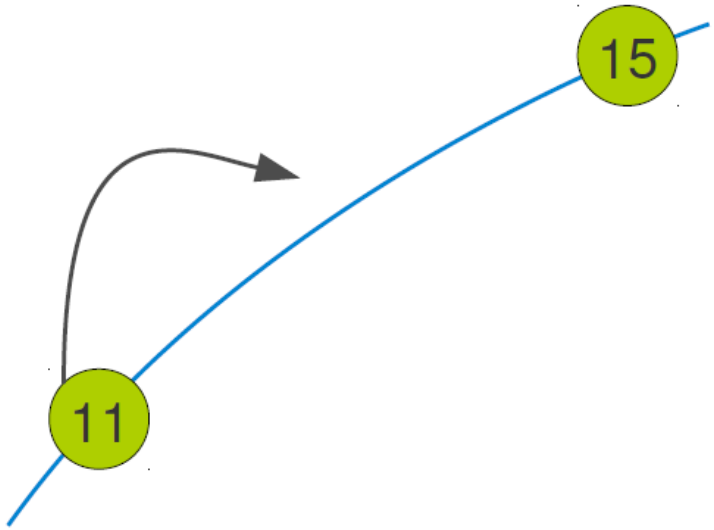
Failure (1/5)



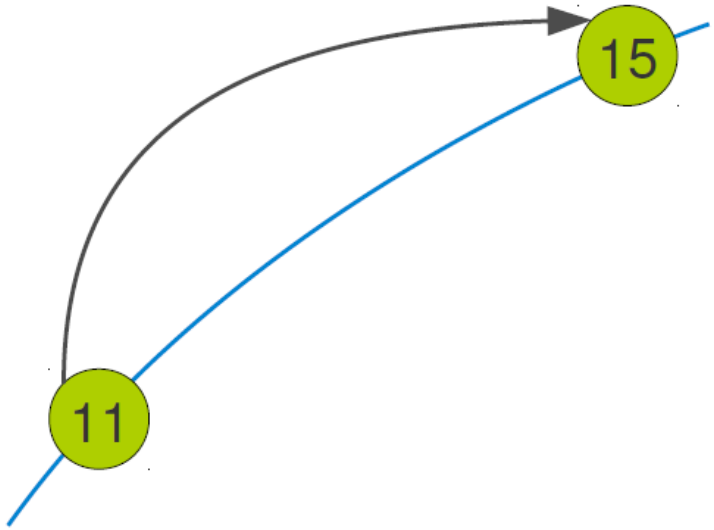
Failure (2/5)



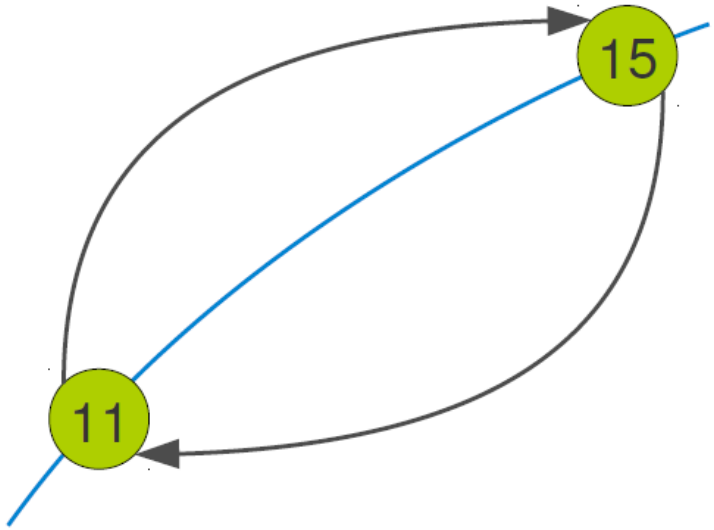
Failure (3/5)



Failure (4/5)



Failure (5/5)



Summary

Summary

- DHTs: distributed $\langle key, value \rangle$
- Lookup service
- Put and Get
- Finger list: improve the lookup
- Periodically stabilization
- Successor list

References:

- Ion Stoica et al., Chord: A scalable peer-to-peer lookup service for internet applications, SIGCOMM, 2001.

Questions?