Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications

[Slides by Amir H. Payberah (amir@sics.se), Jim Dowling]

Consistent Hashing

- Imagine we want to store information about books on 4 nodes (servers).
  - Use the ISBN to identify each book.
- We could use one of the nodes as a central directory server
- But, with the hash of the ISBN, we don't need a central server:

```
switch (SHA-1(ISBN) mod 4) {
    case 0: // store on node1
    case 1: // store on node2
    case 2: // store on node3
    case 3: // store on node4
}
```

- Our store gets bigger......we need to add more 2 nodes.
  - **We now have to re-calculate where all the books are stored.**
- Do the books stay on the same nodes?
- The only books stored on the same node as before are those where

```
SHA-1(ISBN) mod 4 == SHA-1(ISBN) mod 6
```

Consistent Hashing

- Consistent hashing allows you to add more nodes and only a small minority of books will have to move to new nodes.
- **Key property**: *low cost hash-table expansion*. That is, a book's hash key is independent of the number of books and independent of the number of nodes.
  - If you add or remove nodes or books, a book's hash key remains the same.
- **Mechanism**: hash something constant at each node
  - E.g., a node's MAC address

See: Karger et. Al, "Consistent Hashing and Random Trees..."

Consistent Hashing

- Each node is responsible for all books with hash keys between its own hash key and the hash key of the next node (going upwards).

- Imagine we have books with SHA-1(ISBN) in a range 0..16
- For node1..node4, the nodes' hash keys are:
  - `{node1→0, node2→6, node3→11, node4→16}`

- So, a book with SHA-1(ISBN) → 1 would be stored at node1.
  - `(node1) 0 < 1 (book) < 6 (node2)`

- Now if we add new nodes positions 4 and 8, respectively:
  - Nodes have hash keys: `{0, `**`4`**`, 6, `**`8`**`, 11, 16}`

- Fewer books need to be moved
  - Books with hash keys (6..7) get moved from node2 to the first new node
  - Books with hash keys (8..10) get moved from node3 to the second new node

Recap

Distributed Hash Tables (DHT)

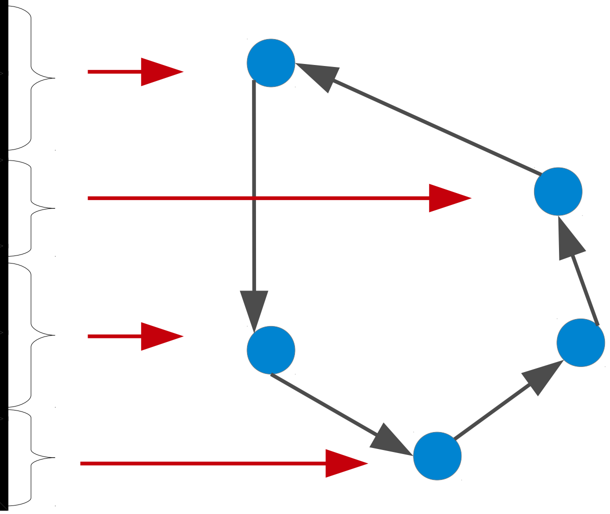
- An ordinary hash-table, which is ...

Key	Value
Fatemeh	Stockholm
Sarunas	Lausanne
Tallat	Islamabad
Cosmin	Bucharest
Seif	Stockholm
Amir	Tehran

Distributed Hash Tables (DHT)

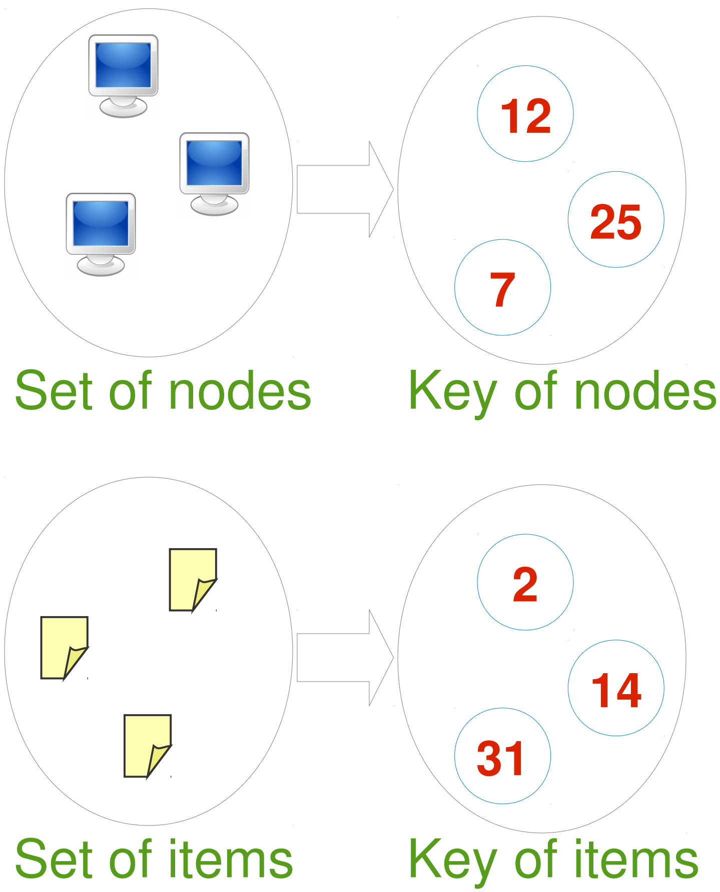
- An ordinary hash-table, which is distributed.

Key	Value
Fatemeh	Stockholm
Sarunas	Lausanne
Tallat	Islamabad
Cosmin	Bucharest
Seif	Stockholm
Amir	Tehran

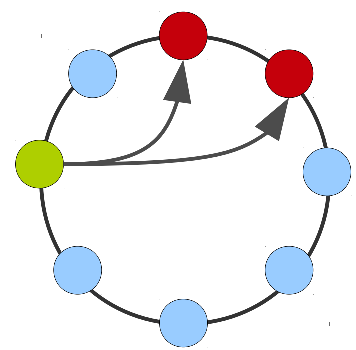


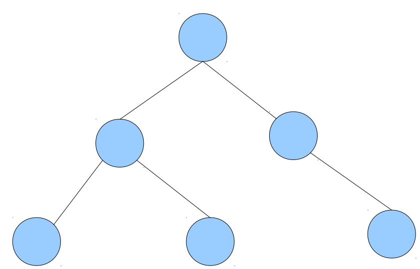
Distributed Hash Tables (DHT)

1. Decide on a common key space for nodes and values

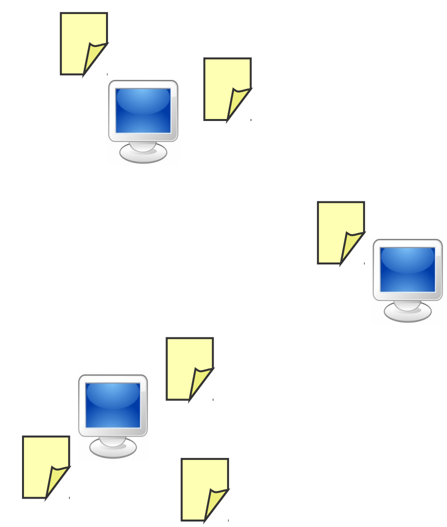


2. Connect nodes using a small, bounded number of links s.t. max hop count is minimized





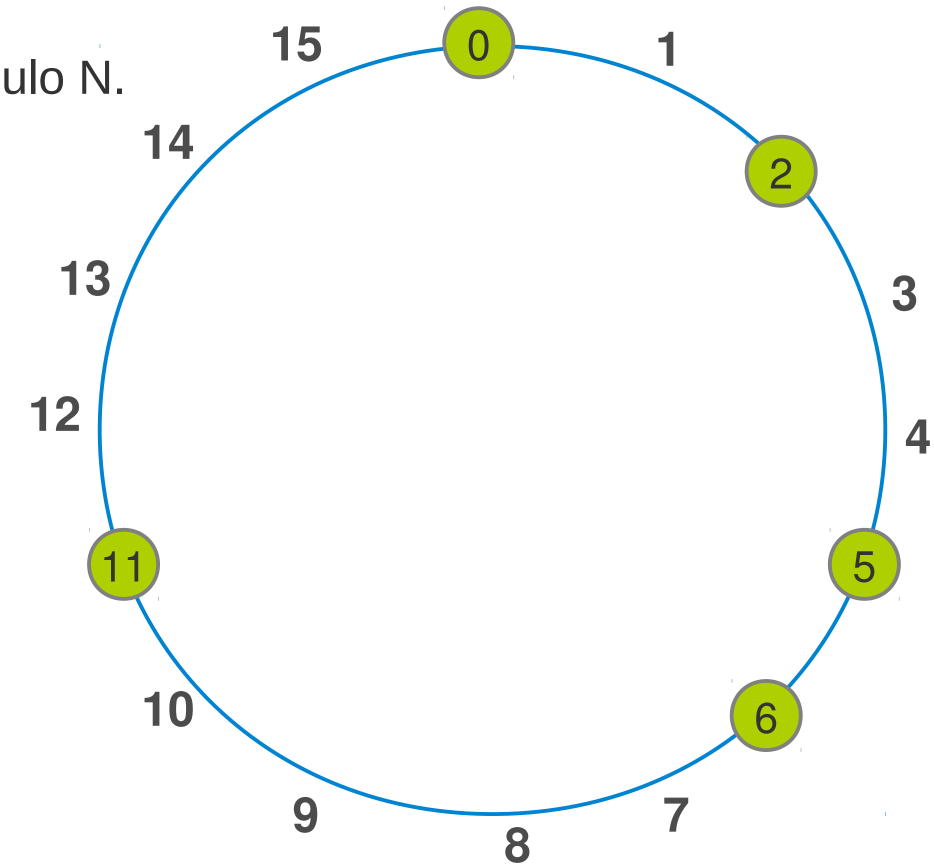
3. Define a strategy for assigning items to nodes



Chord an Example of DHT

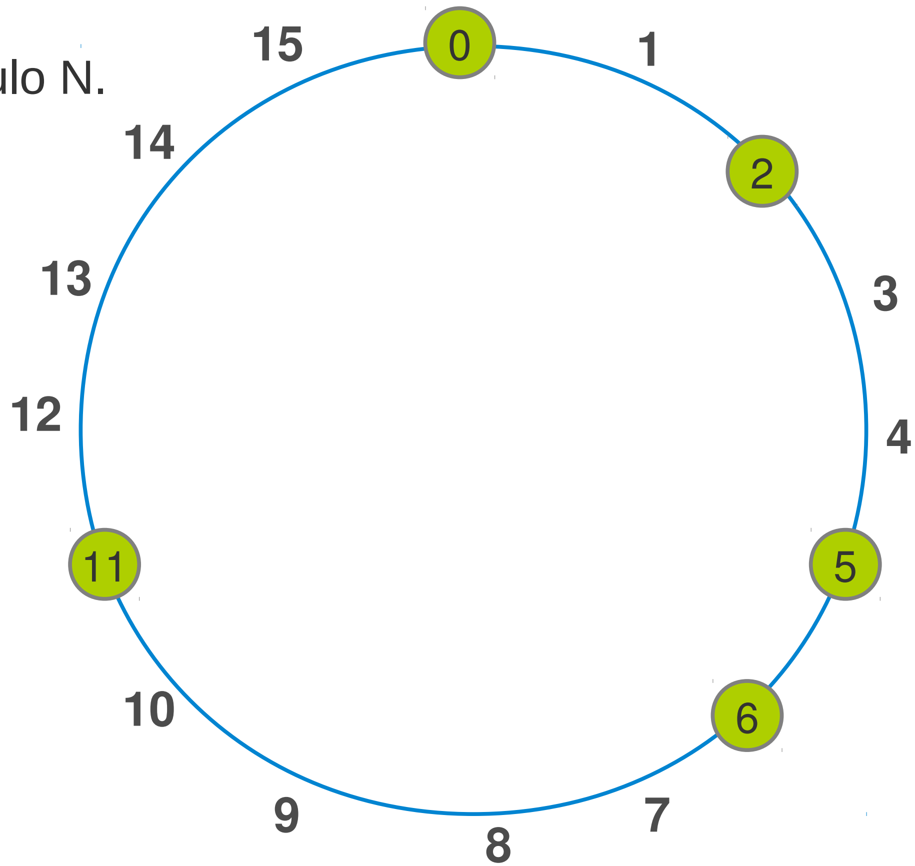
How to Construct a DHT (Chord)?

- Use a logical name space, called the identifier space, consisting of identifiers {0,1,2,..., N-1}
- Identifier space is a logical ring modulo N.



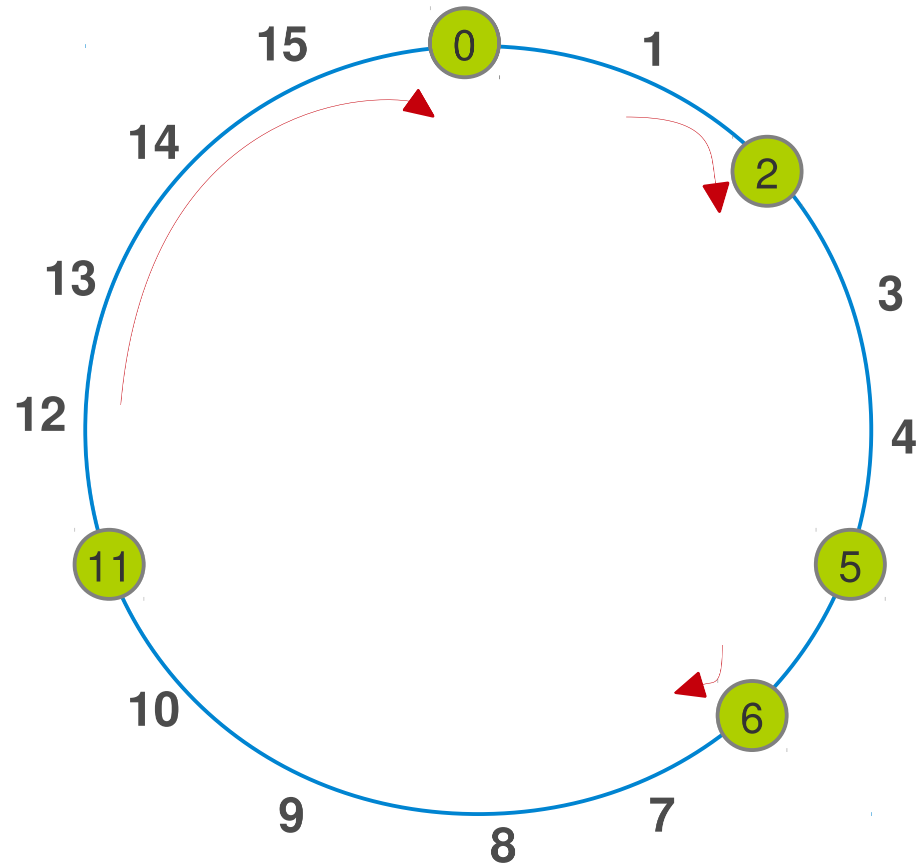
How to Construct a DHT (Chord)?

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



Successor ...

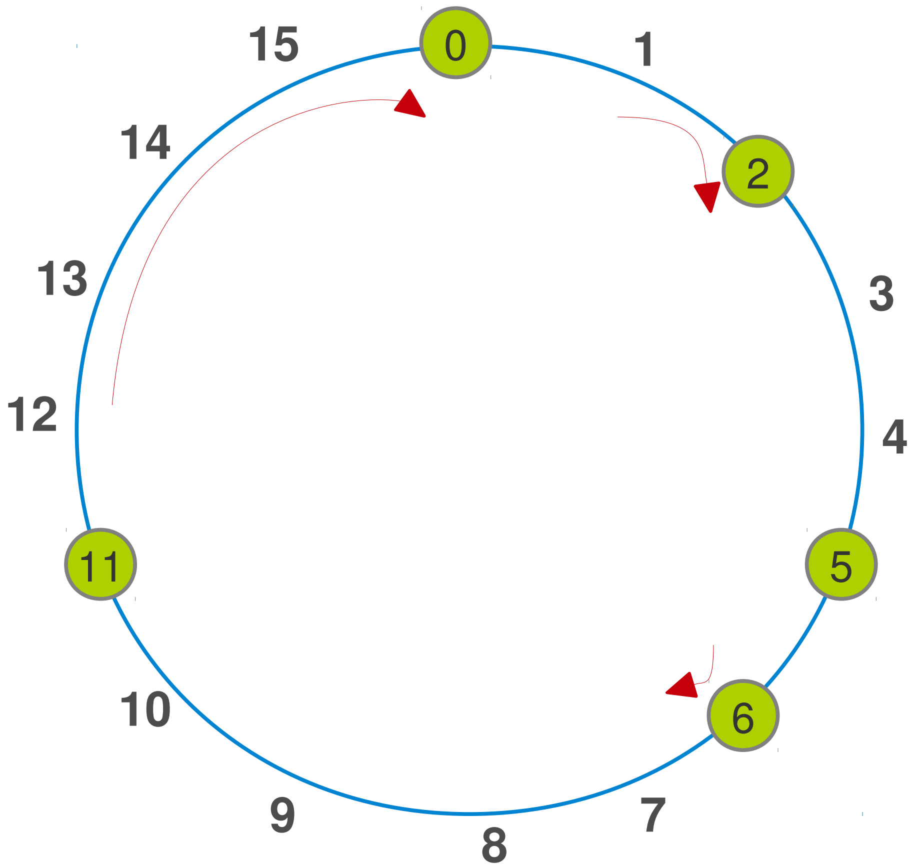
- The successor of an identifier is the first node met going in clockwise direction starting at the identifier.



Successor ...

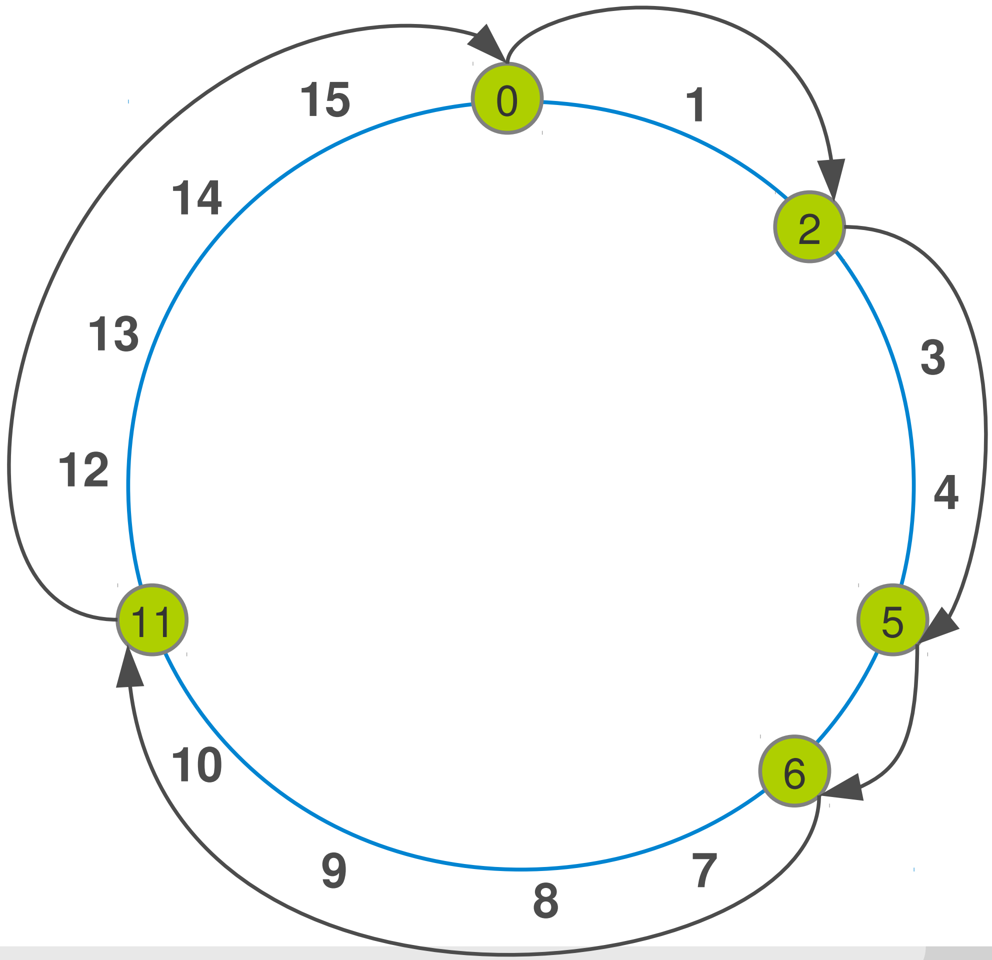
- The successor of an identifier is the first node met going in clockwise direction starting at the identifier.

- 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



Connect the Nodes

- 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
  - 5's successor is succ(6) = 6
  - 6's successor is succ(7) = 11
  - 11's successor is succ(12) = 0



3

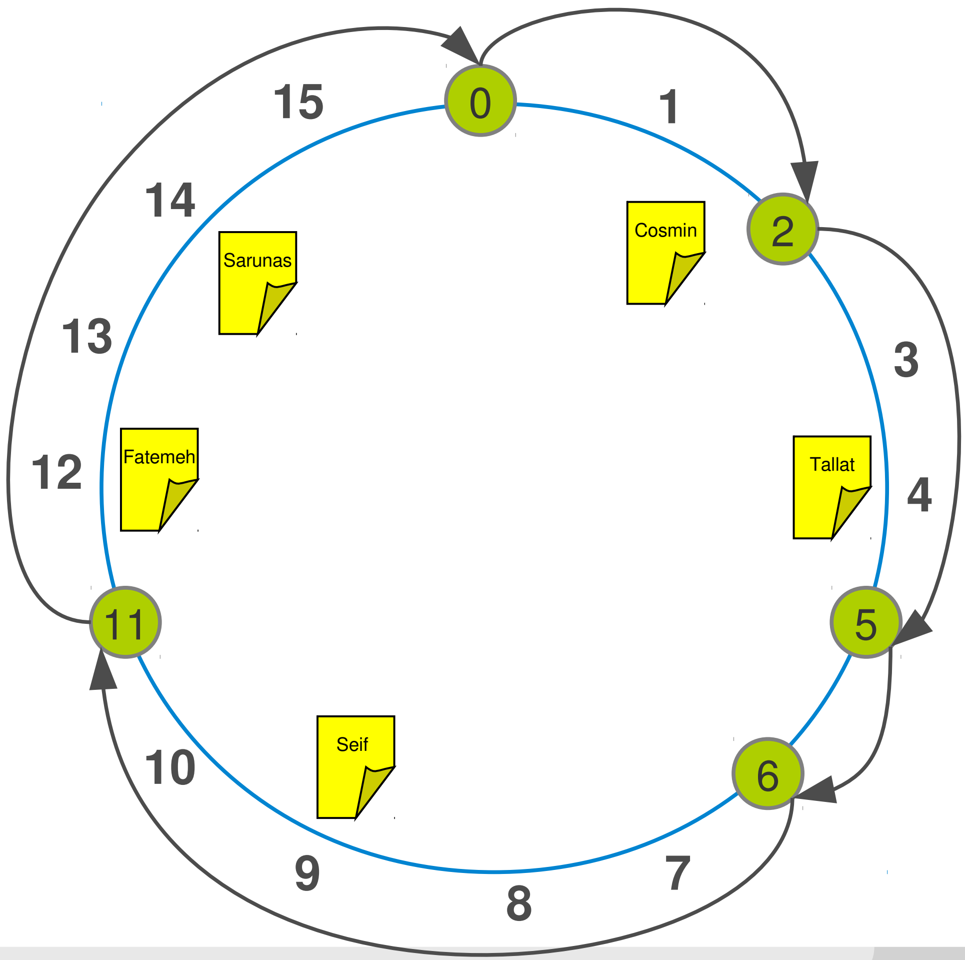
Where to Store Data?

- Use globally known hash function, H.
- Each item <key,value> gets identifier H(key) = k.
  - H("Fatemeh") = 12
  - H("Cosmin") = 2
  - H("Seif") = 9
  - H("Sarunas") = 14
  - H("Tallat") = 4

3

Where to Store Data?

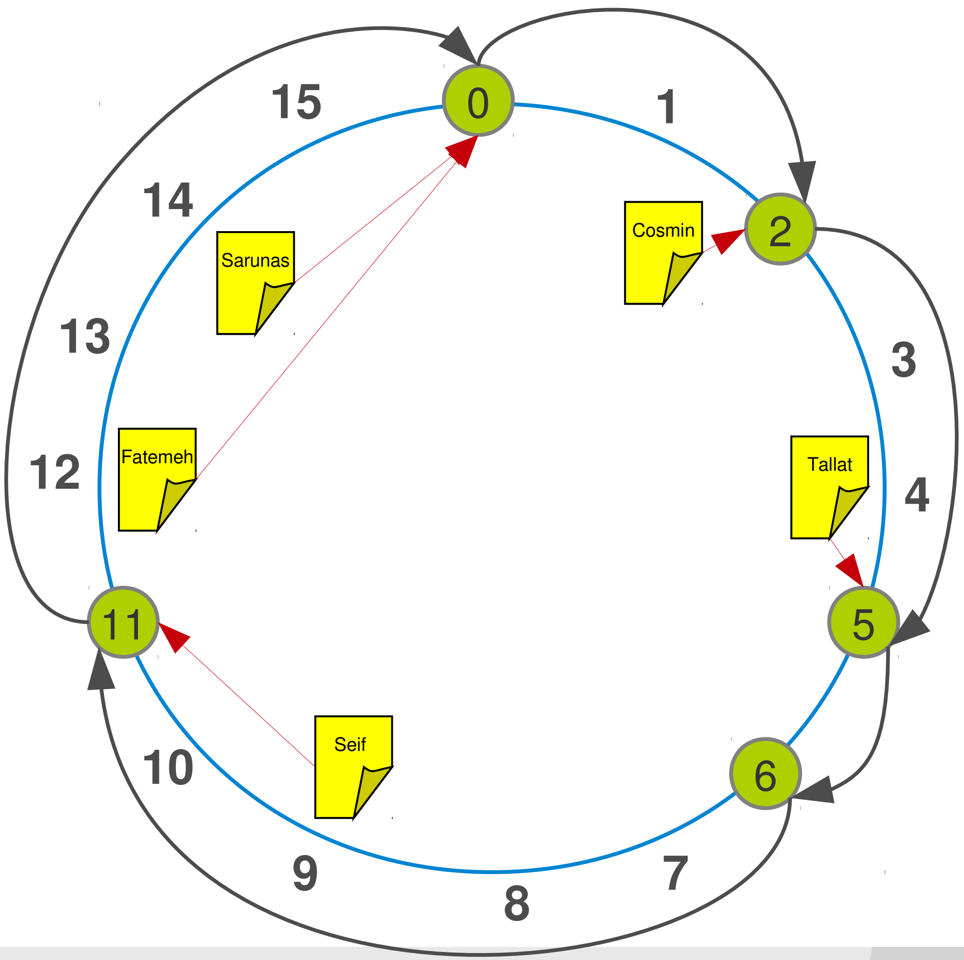
- Use globally known hash function, H.
- Each item <key,value> gets identifier H(key) = k.
  - H("Fatemeh") = 12
  - H("Cosmin") = 2
  - H("Seif") = 9
  - H("Sarunas") = 14
  - H("Tallat") = 4



❸

Where to Store Data?

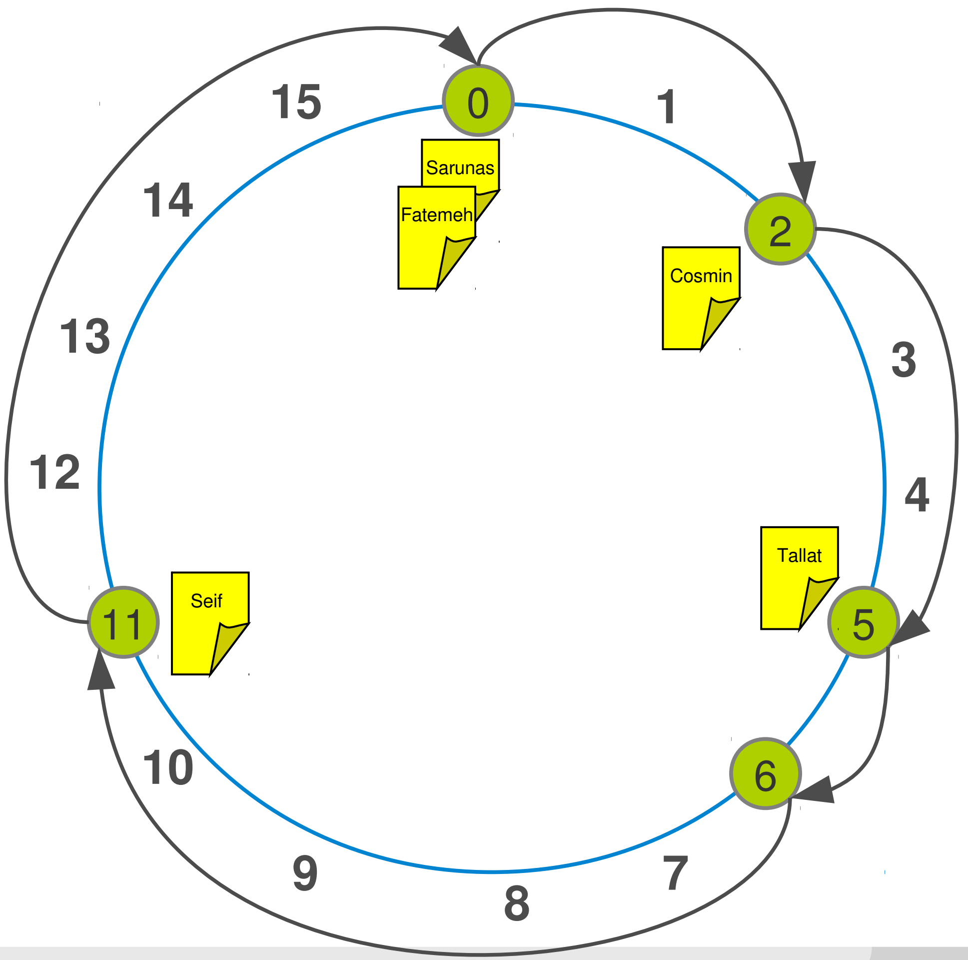
- Use globally known hash function, H.
- Each item <key,value> gets identifier H(key) = k.
  - H("Fatemeh") = 12
  - H("Cosmin") = 2
  - H("Seif") = 9
  - H("Sarunas") = 14
  - H("Tallat") = 4
- Store each item at its successor.



3

Where to Store Data?

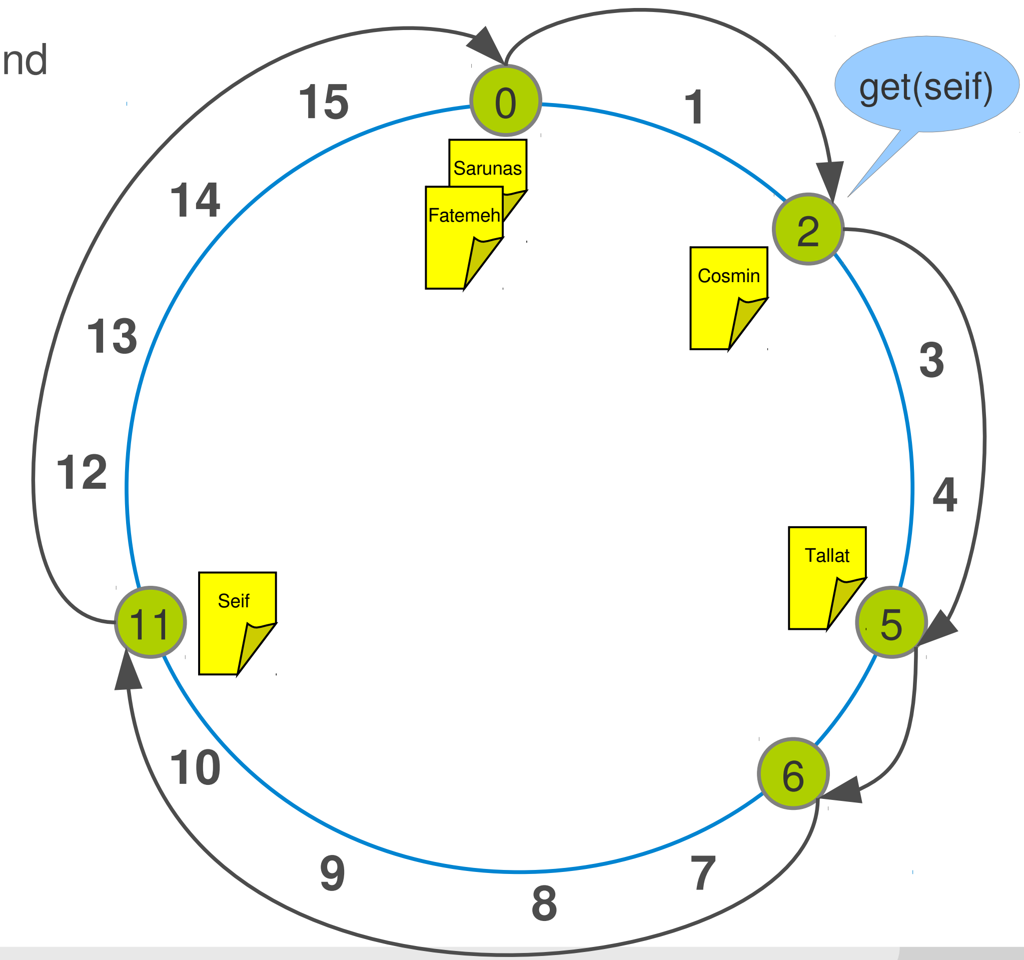
- Use globally known hash function, H.
- Each item <key,value> gets identifier H(key) = k.
  - H("Fatemeh") = 12
  - H("Cosmin") = 2
  - H("Seif") = 9
  - H("Sarunas") = 14
  - H("Tallat") = 4
- Store each item at its successor.



Lookup?

Lookup?

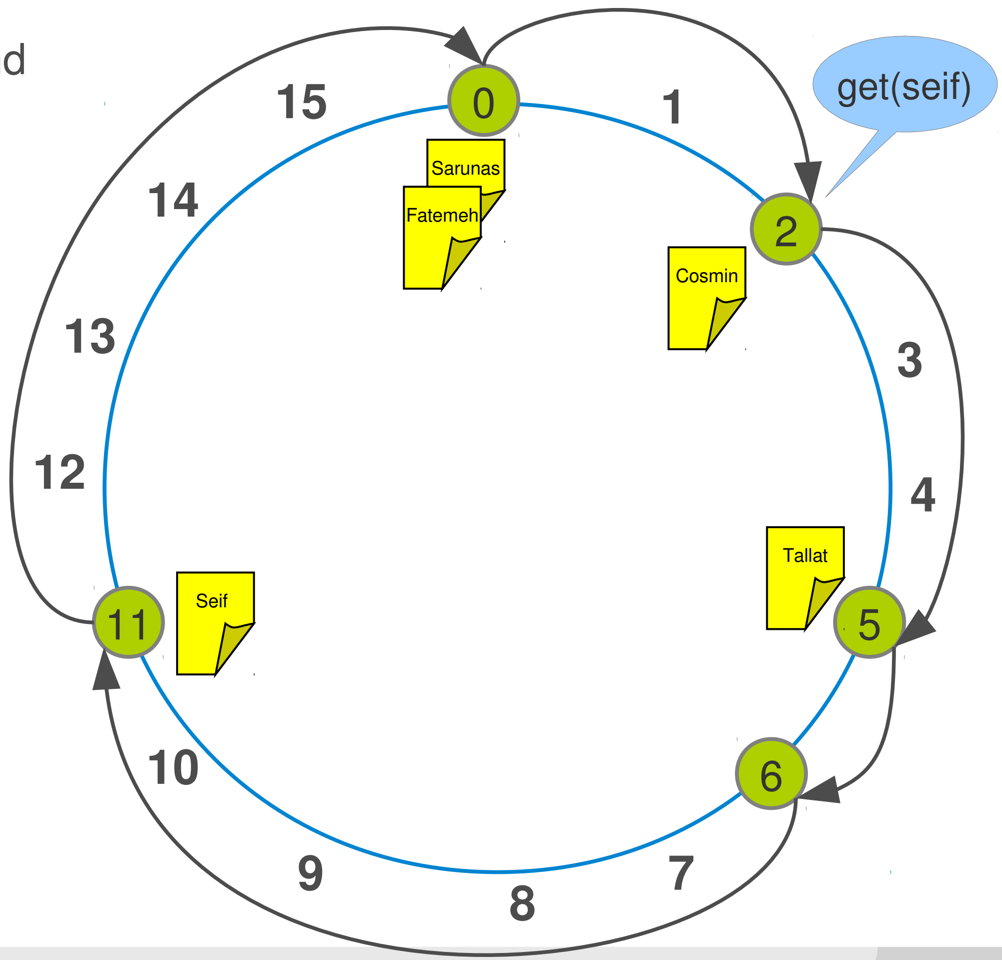
- To lookup a key k
  - Calculate H(k)
  - Follow succ pointers until item k is found



Lookup?

- 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 (BINGO)
  - Return "Stockholm" to initiator

Key	Value
Seif	Stockholm



Lookup?

```
// ask node n to find the successor of id
procedure n.findSuccessor(id) {
   if (predecessor ≠ nil and id ∈ (predecessor, n]) then return n
   else if (id ∈ (n, successor]) then
      return successor
   else // forward the query around the circle
      return successor.findSuccessor(id)
}
```

- (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

```
procedure n.put(id, value) {
  s = findSuccessor(id)
  s.store(id, value)
}
```

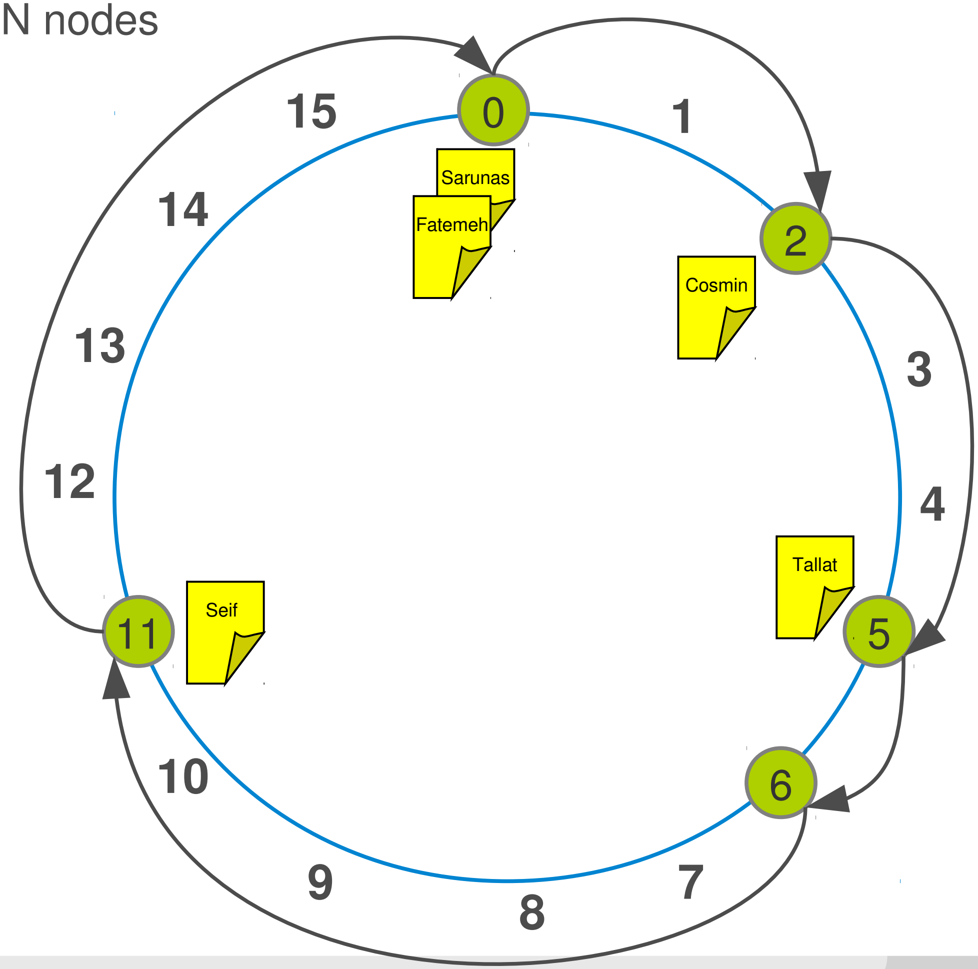
```
procedure n.get(id) {
  s = findSuccessor(id)
  return s.retrieve(id)
}
```

- PUT and GET are nothing but lookups!!

How can we improve this?

Cost of Lookup Operations

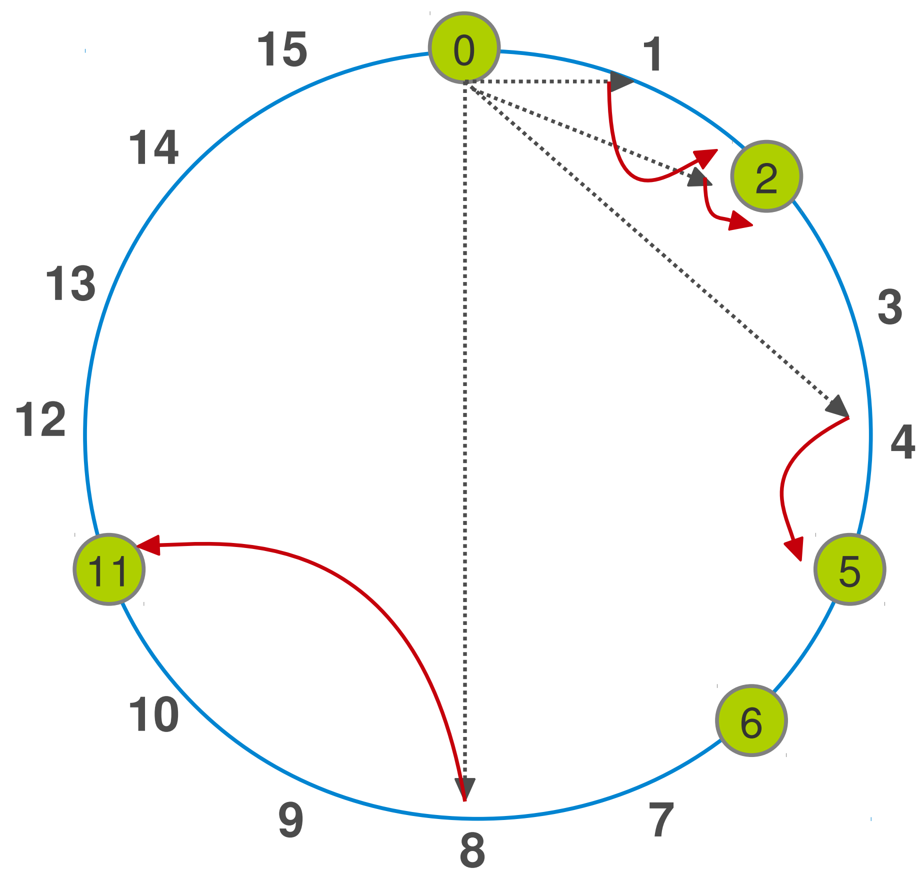
- If only the pointer to succ(n+1) is used
  - Worst case lookup time is O(N), for N nodes



Speeding up Lookups

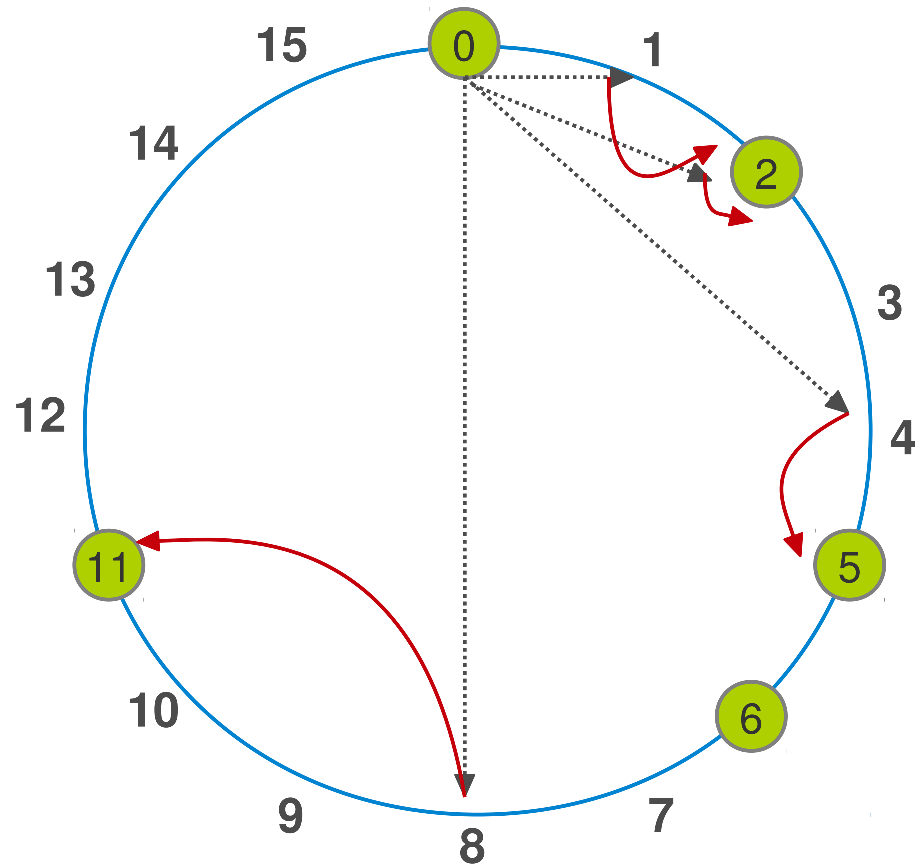
- Finger/routing table:
  - Point to succ(n+1)
  - Point to succ(n+2)
  - Point to succ(n+4)
  - Point to succ(n+8)
  - …
  - Point to succ(n+2^{M-1})

- Distance always halved to the destination.



Speeding up Lookups

- Size of routing tables is logarithmic.:
  - Routing table size: M, where N = 2^M.
- Every node n knows successor(n + 2^(i-1)) for i = 1... M
- Routing entries = $\log_2(N)$
  - $\log_2(N)$ hops from any node to any other node
- Example: $\text{Log}_2(1000000) \approx 20$



DHT Lookup

```
// ask node n to find the successor of id
procedure n.findSuccessor(id) {
  if (predecessor ≠ nil and id ∈ (predecessor, n]) then return n
  else if (id ∈ (n, successor]) then
    return successor
  else // forward the query around the circle
    return successor.findSuccessor(id)
}
```

DHT Lookup

```
// ask node n to find the successor of id
procedure n.findSuccessor(id) {
    if (predecessor ≠ nil and id ∈ (predecessor, n]) then return n
    else if (id ∈ (n, successor]) then
        return successor
    else // forward the query around the circle
        return successor.findSuccessor(id)
}
```

closestPrecedingNode(id)

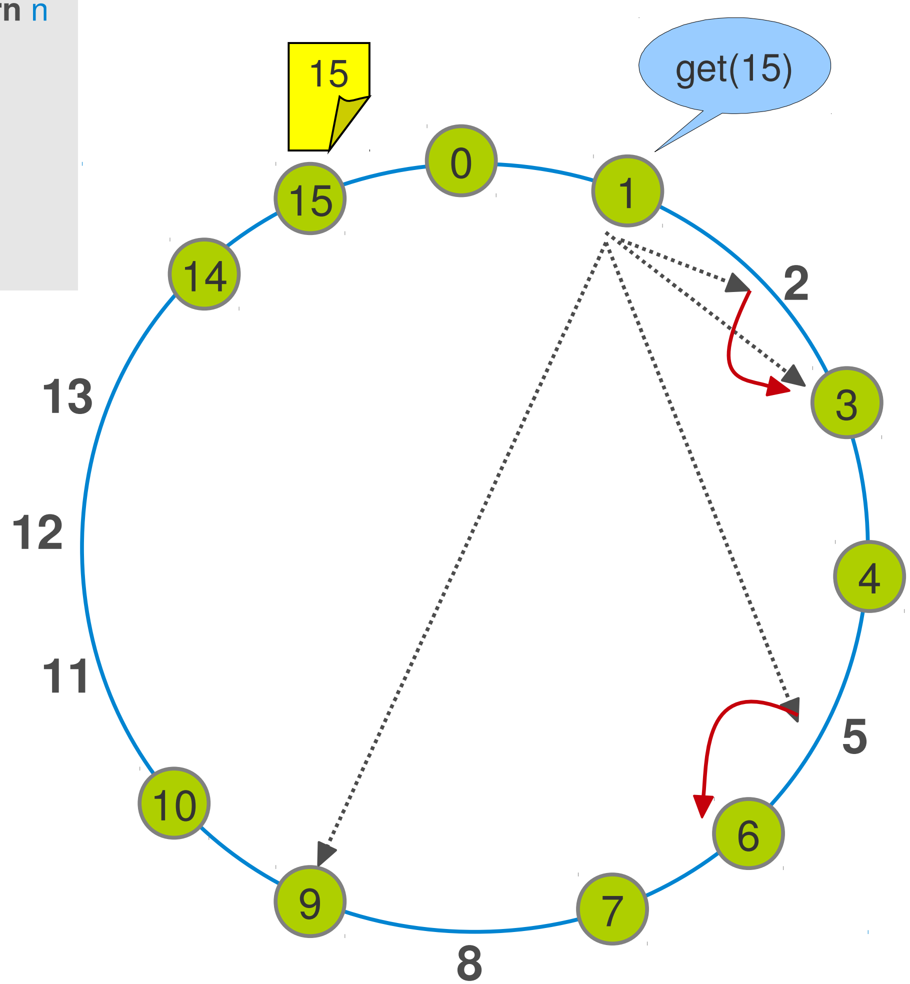
DHT Lookup

```
// ask node n to find the successor of id
procedure n.findSuccessor(id) {
   if (predecessor ≠ nil and id ∈ (predecessor, n]) then return n
   else if (id ∈ (n, successor]) then
      return successor
   else { // forward the query around the circle
      m := closestPrecedingNode(id)
      return m.findSuccessor(id)
   }
}
```

```
// search locally for the highest predecessor of id
procedure closestPrecedingNode(id) {
   for i = m downto 1 do {
      if (finger[i] ∈ (n, id)) then
         return finger[i]
   }
   return n
}
```

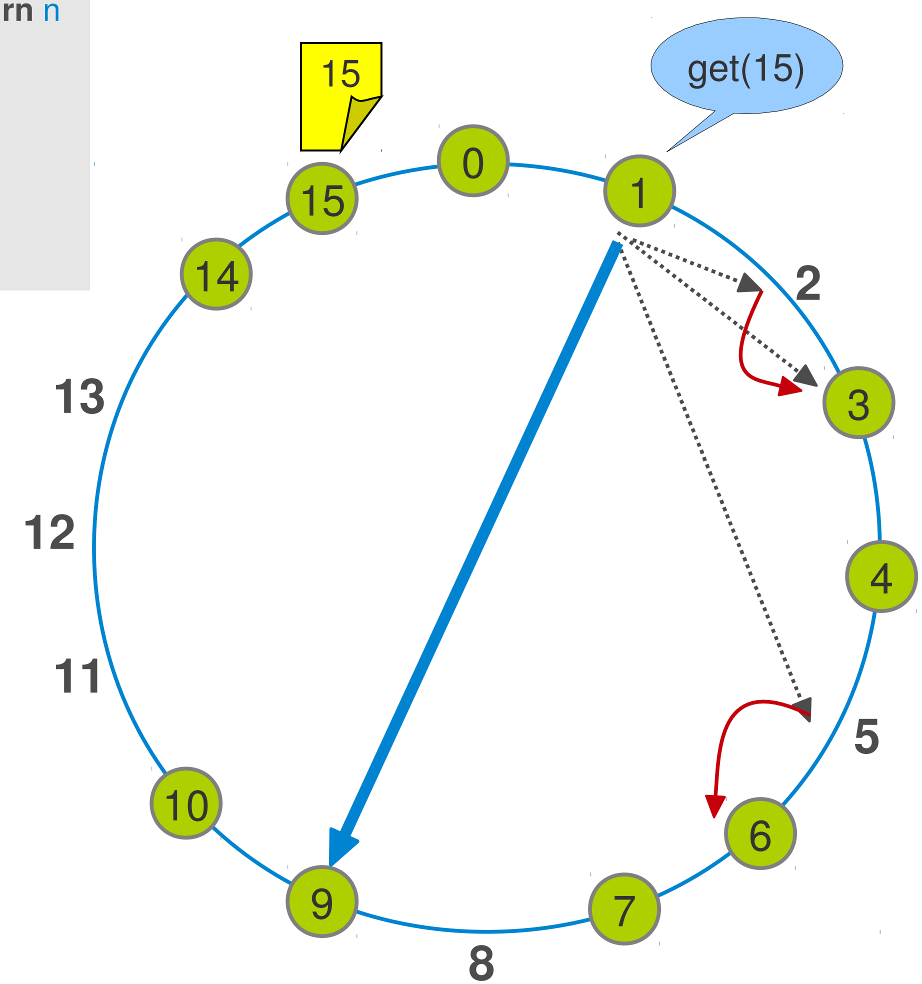
Chord – Lookup (1/4)

```
procedure n.findSuccessor(id) {
  if (predecessor ≠ nil and id ∈ (predecessor, n]) then return n
  else if (id ∈ (n, successor]) then
     return successor
  else { // forward the query around the circle
     m := closestPrecedingNode(id)
     return m.findSuccessor(id)
  }
}
```



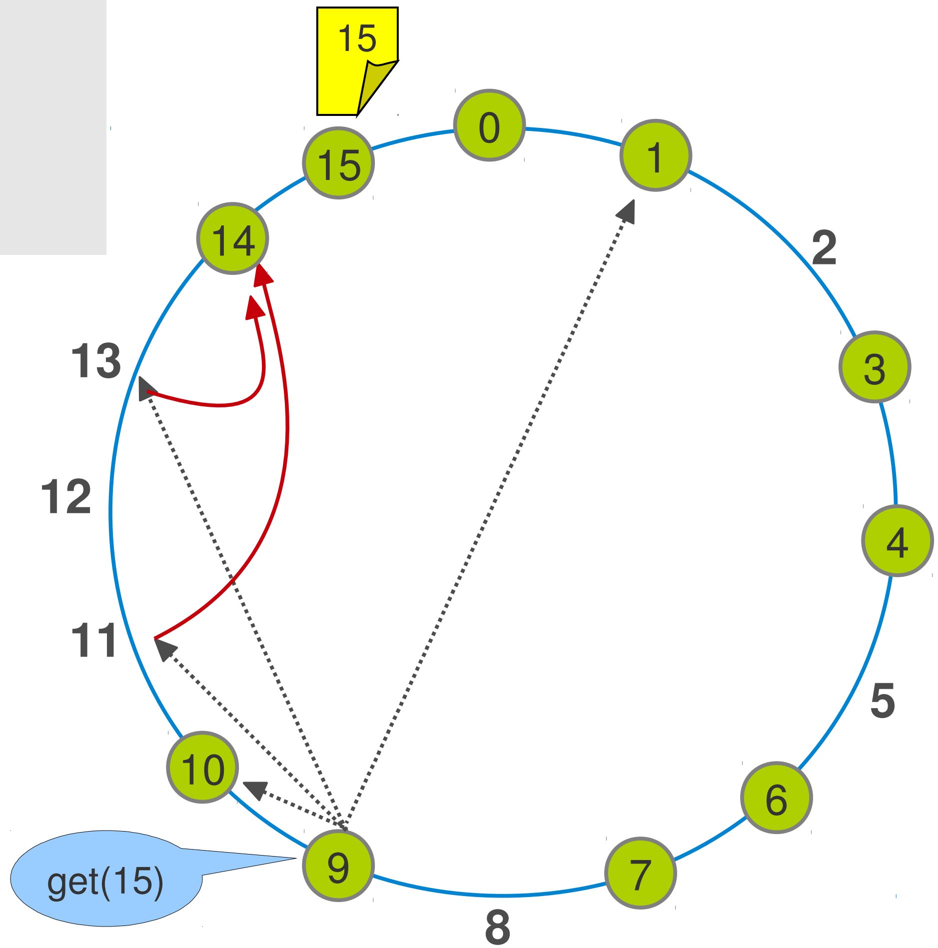
Chord – Lookup (1/4)

```
procedure n.findSuccessor(id) {
  if (predecessor ≠ nil and id ∈ (predecessor, n]) then return n
  else if (id ∈ (n, successor]) then
     return successor
  else { // forward the query around the circle
     m := closestPrecedingNode(id)
     return m.findSuccessor(id)
  }
}
```



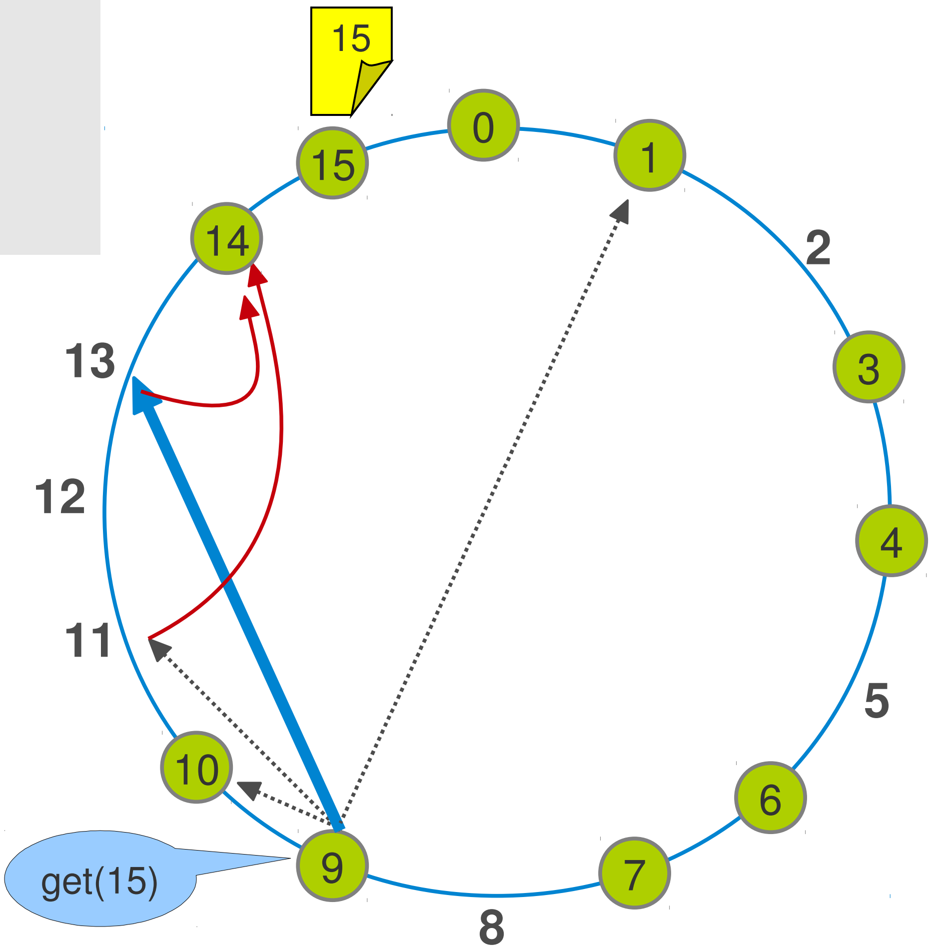
Chord – Lookup (2/4)

```
procedure n.findSuccessor(id) {
  if (predecessor ≠ nil and id ∈ (predecessor, n]) then return n
  else if (id ∈ (n, successor]) then
     return successor
  else { // forward the query around the circle
     m := closestPrecedingNode(id)
     return m.findSuccessor(id)
  }
}
```



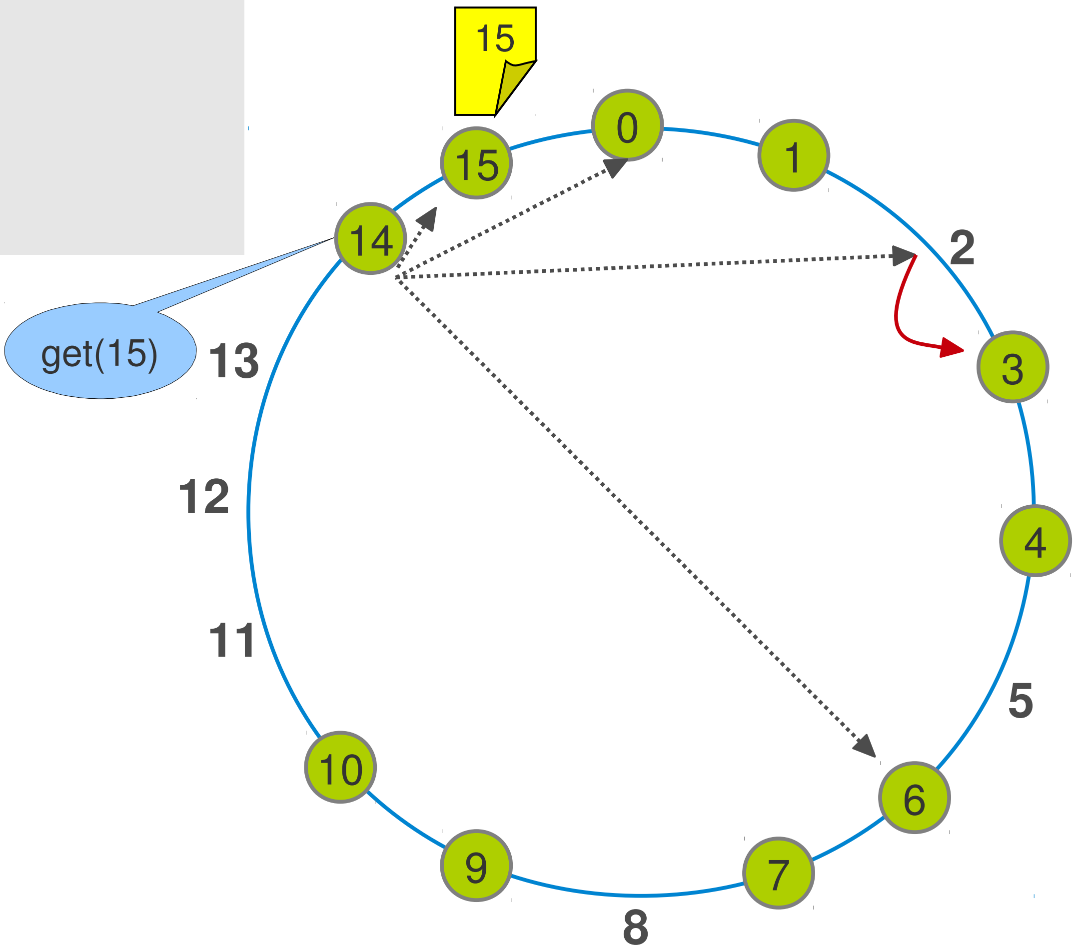
Chord – Lookup (2/4)

```
procedure n.findSuccessor(id) {
  if (predecessor ≠ nil and id ∈ (predecessor, n]) then return n
  else if (id ∈ (n, successor]) then
     return successor
  else { // forward the query around the circle
     m := closestPrecedingNode(id)
     return m.findSuccessor(id)
  }
}
```



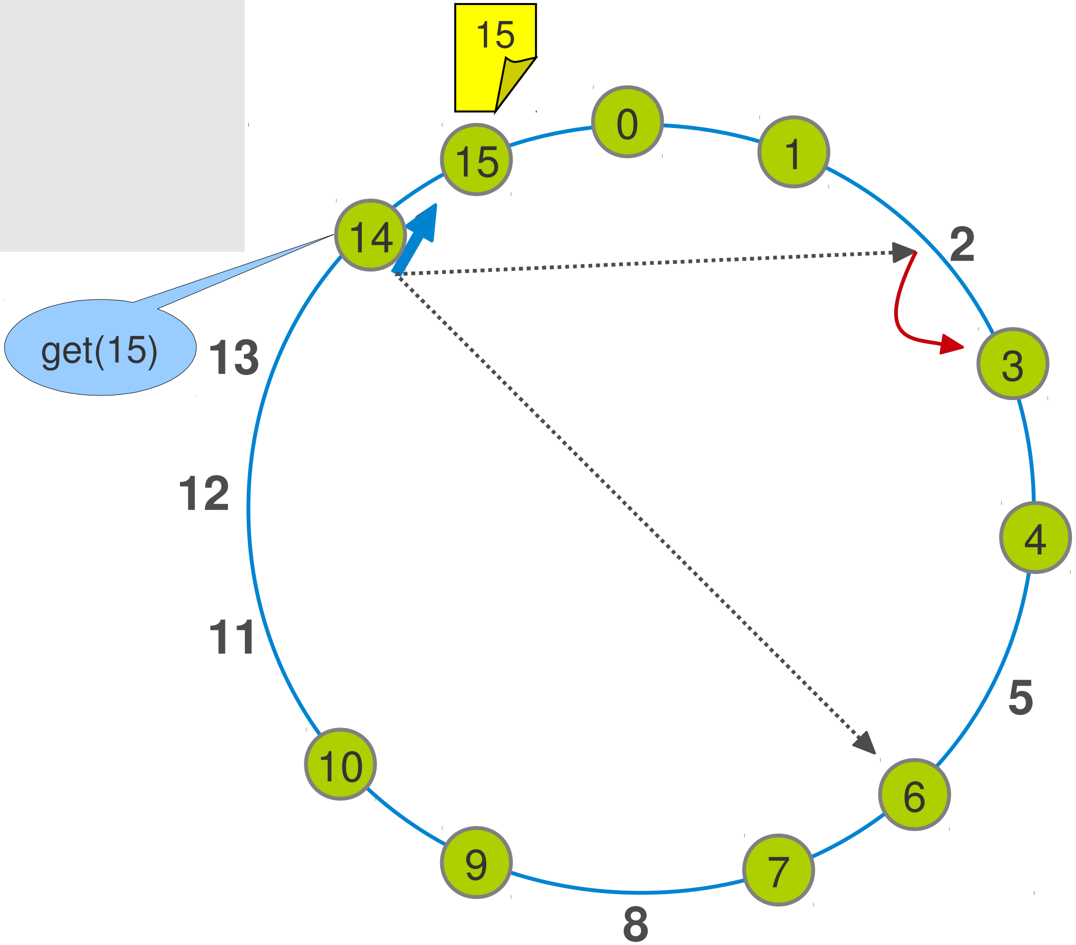
Chord – Lookup (3/4)

```
procedure n.findSuccessor(id) {
  if (predecessor ≠ nil and id ∈ (predecessor, n]) then return n
  else if (id ∈ (n, successor]) then
     return successor
  else { // forward the query around the circle
     m := closestPrecedingNode(id)
     return m.findSuccessor(id)
  }
}
```



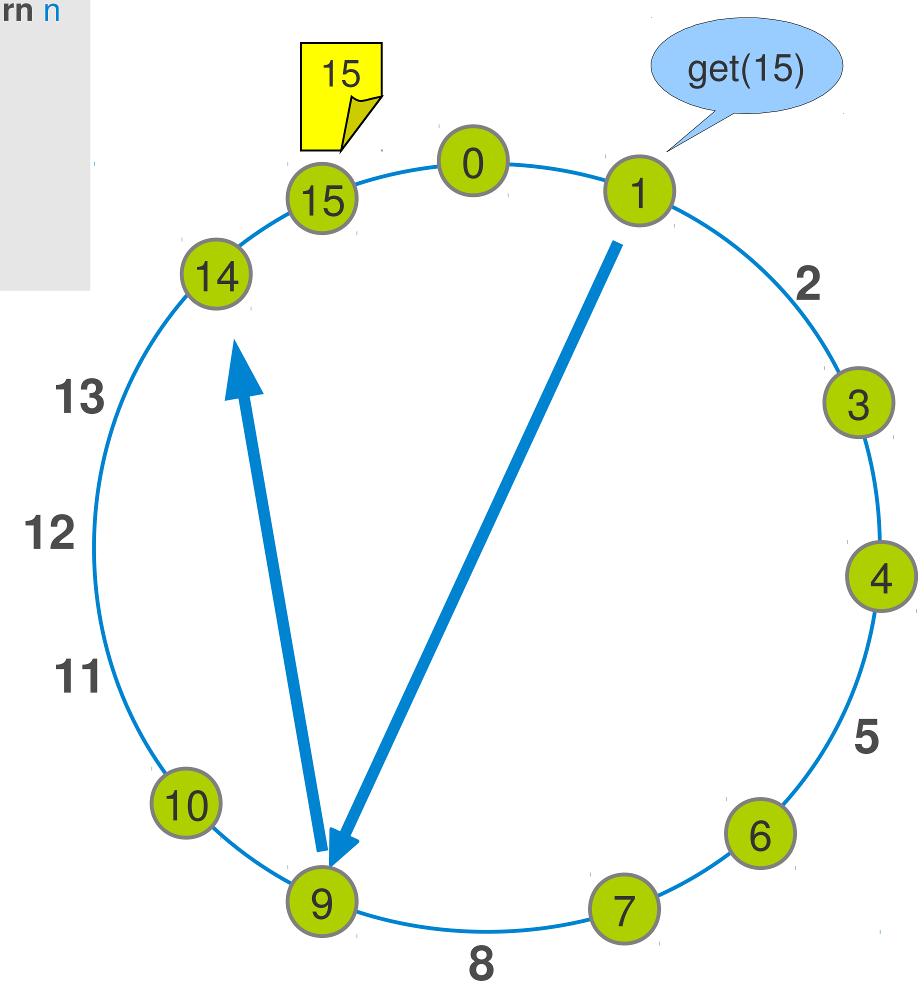
Chord – Lookup (3/4)

```
procedure n.findSuccessor(id) {
  if (predecessor ≠ nil and id ∈ (predecessor, n]) then return n
  else if (id ∈ (n, successor]) then
     return successor
  else { // forward the query around the circle
     m := closestPrecedingNode(id)
     return m.findSuccessor(id)
  }
}
```



Chord – Lookup (4/4)

```
procedure n.findSuccessor(id) {
  if (predecessor ≠ nil and id ∈ (predecessor, n]) then return n
  else if (id ∈ (n, successor]) then
     return successor
  else { // forward the query around the circle
     m := closestPrecedingNode(id)
     return m.findSuccessor(id)
  }
}
```



Discussion

- We are basically done.
- But …
- What about joins and failures/leaves?
  - Nodes come and go as they wish.
- What about data?
  - Should I lose my doc because some kid decided to shut down his machine and he happened to store my file?
- So actually we just started ...

Handling Dynamism?
Ring Maintenance?

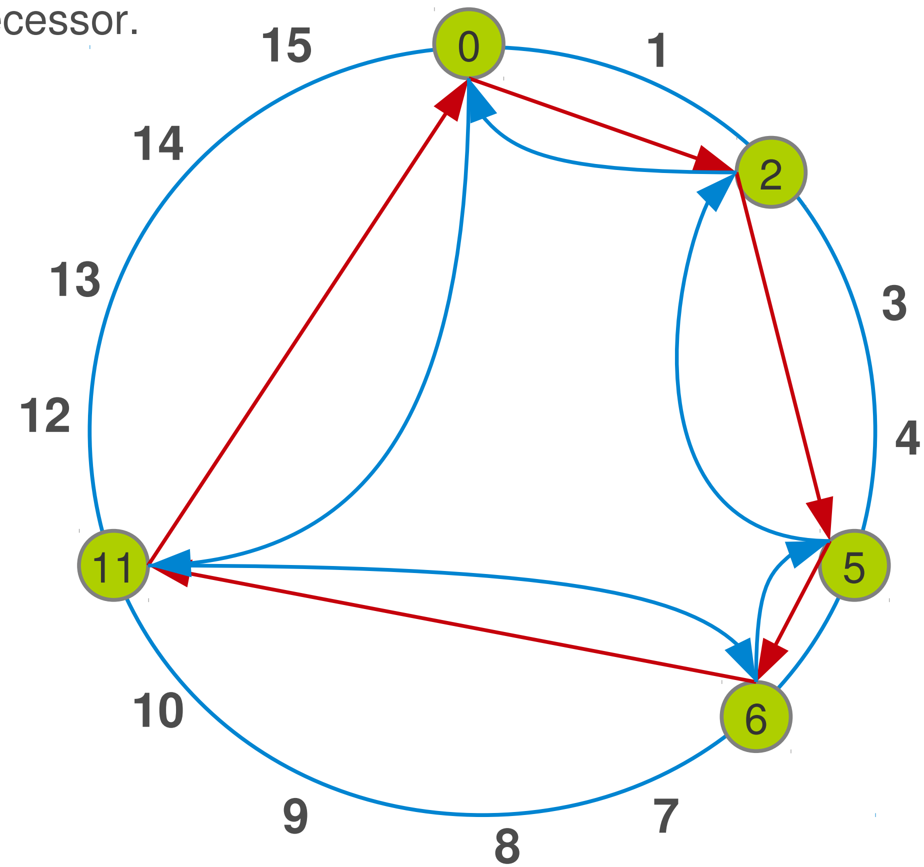
Handling Dynamism - Ring Maintenance

- Everything depends on successor pointers.

- In Chord, in addition to the successor pointer, every node has a predecessor pointer as well for ring maintenance.
  - Predecessor of node n is the first node met in anti-clockwise direction starting at n-1.

Handling Dynamism - Ring Maintenance

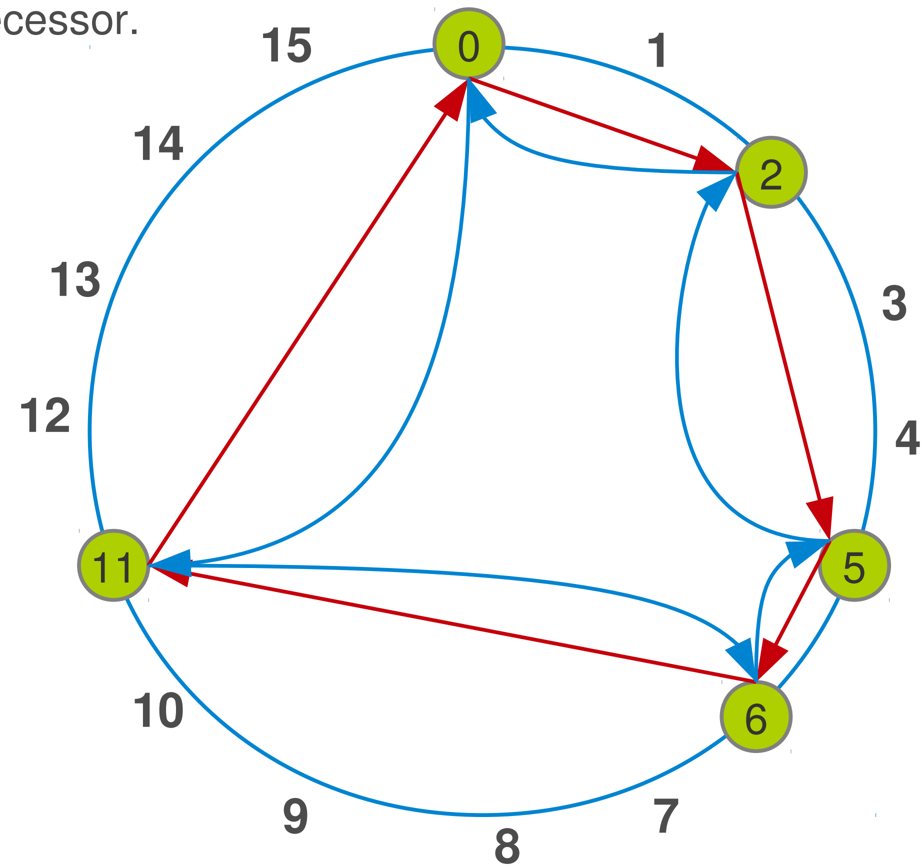
- Periodic stabilization is used to make pointers eventually correct.
  - Try pointing succ to closest alive successor.
  - Try pointing pred to closest alive predecessor.



Handling Dynamism - Ring Maintenance

- Periodic stabilization is used to make pointers eventually correct.
  - Try pointing succ to closest alive successor.
  - Try pointing pred to closest alive predecessor.

```
// Periodically at n:
v := succ.pred
if (v ≠ nil and v ∈ (n,succ]) then
    set succ := v
send a notify(n) to succ
```

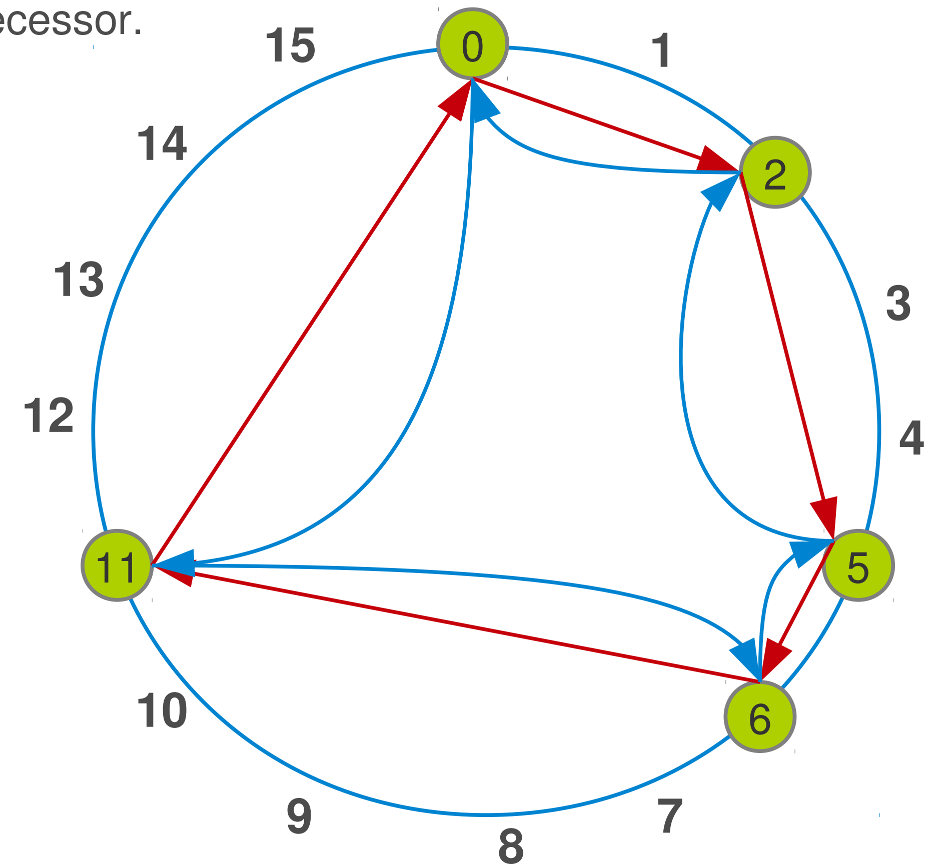


Handling Dynamism - Ring Maintenance

- Periodic stabilization is used to make pointers eventually correct.
  - Try pointing succ to closest alive successor.
  - Try pointing pred to closest alive predecessor.

```
// Periodically at n:
v := succ.pred
if (v ≠ nil and v ∈ (n,succ]) then
    set succ := v
send a notify(n) to succ
```

```
// When receiving notify(p) at n:
if (pred = nil or p ∈ (pred, n]) then
    set pred := p
```



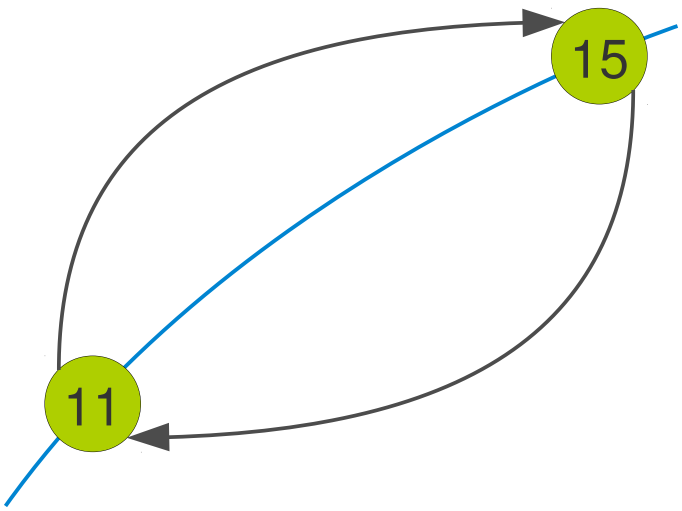
Handling Join?

Chord – Handling Join (1/5)

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

```
// Periodically at n:
v := succ.pred
if (v ≠ nil and v ∈ (n,succ]) then
    set succ := v
send a notify(n) to succ
```

```
// When receiving notify(p) at n:
if (pred = nil or p ∈ (pred, n]) then
    set pred := p
```

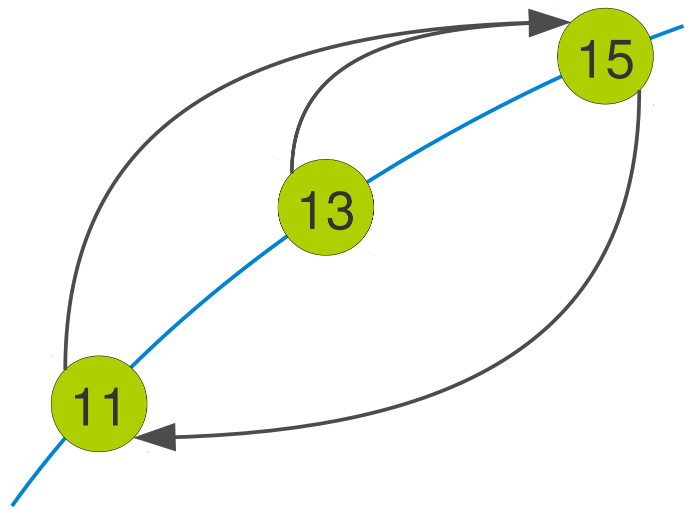


Chord – Handling Join (2/5)

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

```
// Periodically at n:
v := succ.pred
if (v ≠ nil and v ∈ (n,succ]) then
    set succ := v
send a notify(n) to succ
```

```
// When receiving notify(p) at n:
if (pred = nil or p ∈ (pred, n]) then
    set pred := p
```

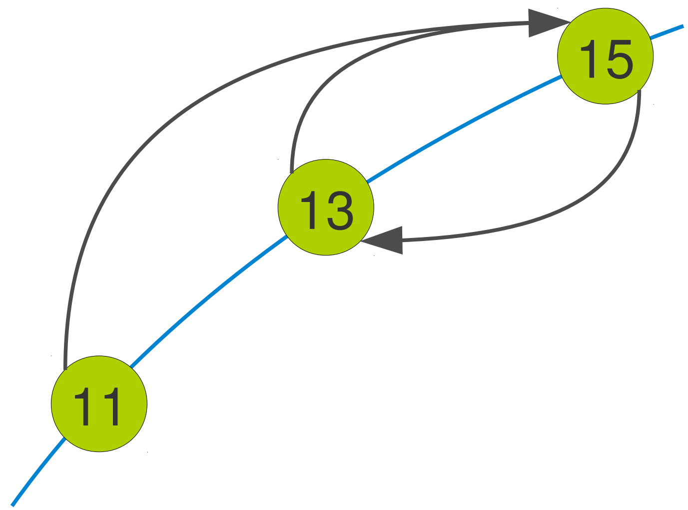


Chord – Handling Join (3/5)

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

```
// Periodically at n:
v := succ.pred
if (v ≠ nil and v ∈ (n,succ]) then
    set succ := v
send a notify(n) to succ
```

```
// When receiving notify(p) at n:
if (pred = nil or p ∈ (pred, n]) then
    set pred := p
```

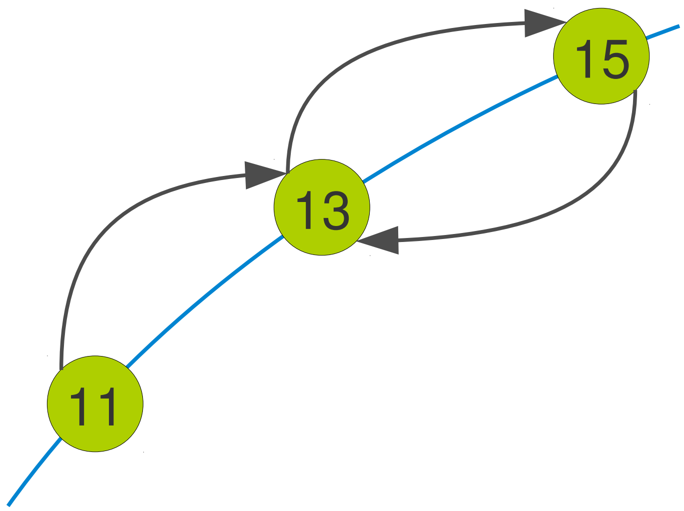


Chord – Handling Join (4/5)

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

```
// Periodically at n:
v := succ.pred
if (v ≠ nil and v ∈ (n,succ]) then
    set succ := v
send a notify(n) to succ
```

```
// When receiving notify(p) at n:
if (pred = nil or p ∈ (pred, n]) then
    set pred := p
```

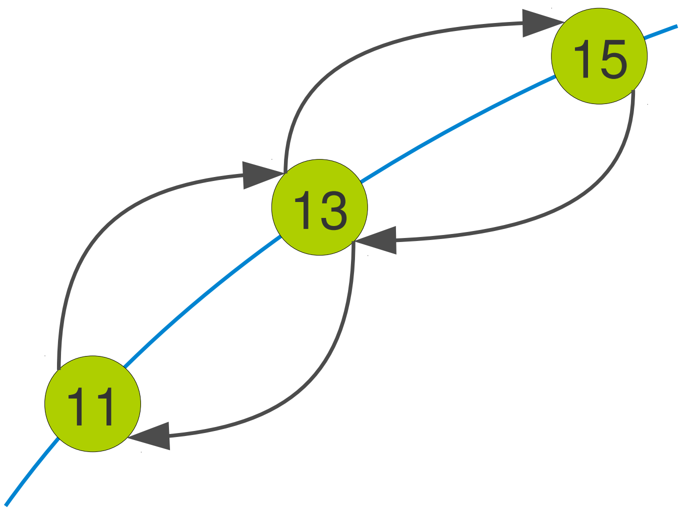


Chord – Handling Join (5/5)

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

```
// Periodically at n:
v := succ.pred
if (v ≠ nil and v ∈ (n,succ]) then
    set succ := v
send a notify(n) to succ
```

```
// When receiving notify(p) at n:
if (pred = nil or p ∈ (pred, n]) then
    set pred := p
```



Fixing Fingers

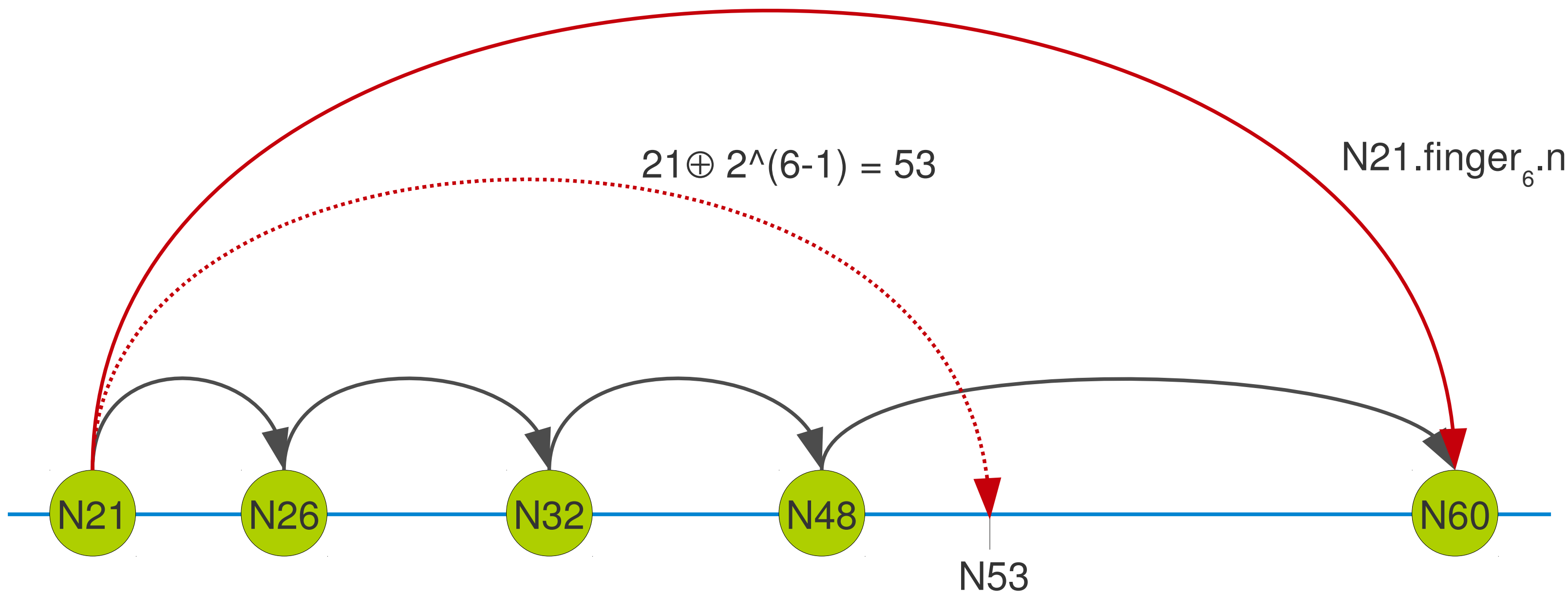
Chord – Fixing Fingers

- Periodically refresh finger table entries, and store the index of the next finger to fix.
- Local variable next initially is 0.

```
// When receiving notify(p) at n:
procedure n.fixFingers() {
   next := next+1
   if (next > m) then
      next := 1
   finger[next] := findSuccessor(n ⊕ 2^(next - 1))
}
```

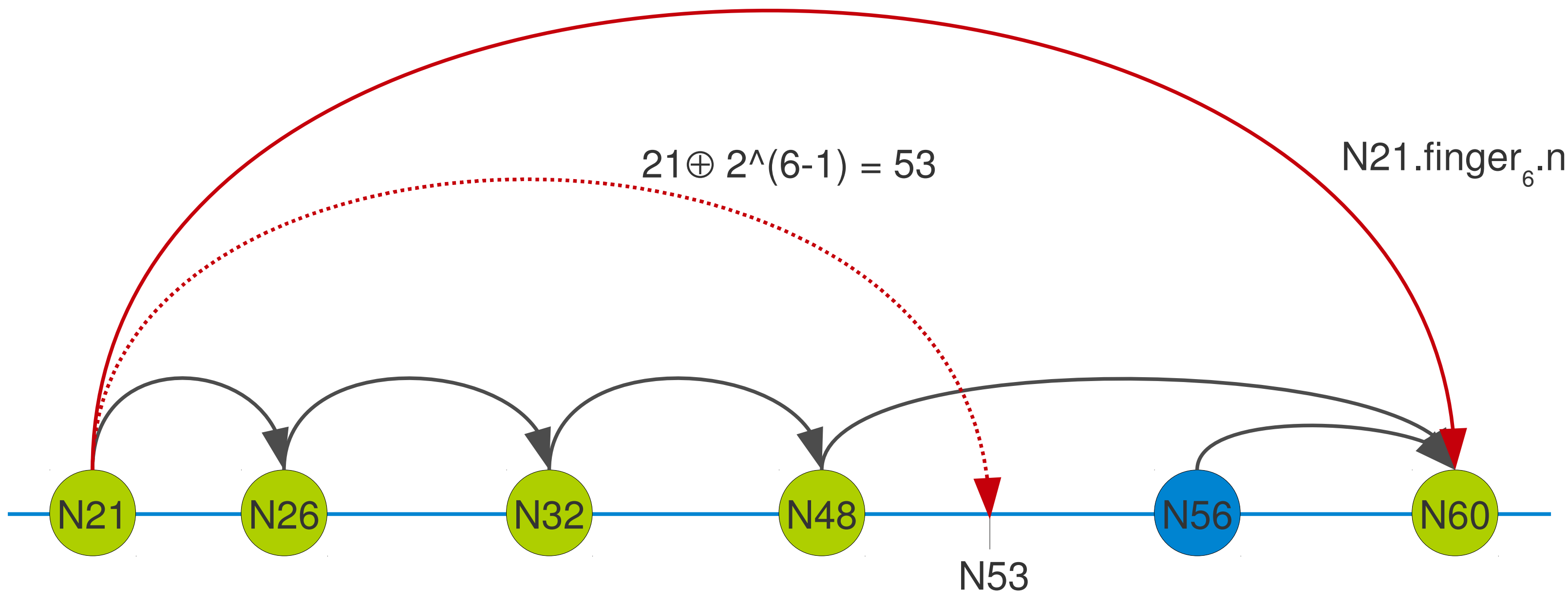
Chord – Fixing Fingers (1/4)

- Current situation: succ(N48) is N60.
- Succ(21⊕ 2^(6-1)) = Succ(53) = N60.



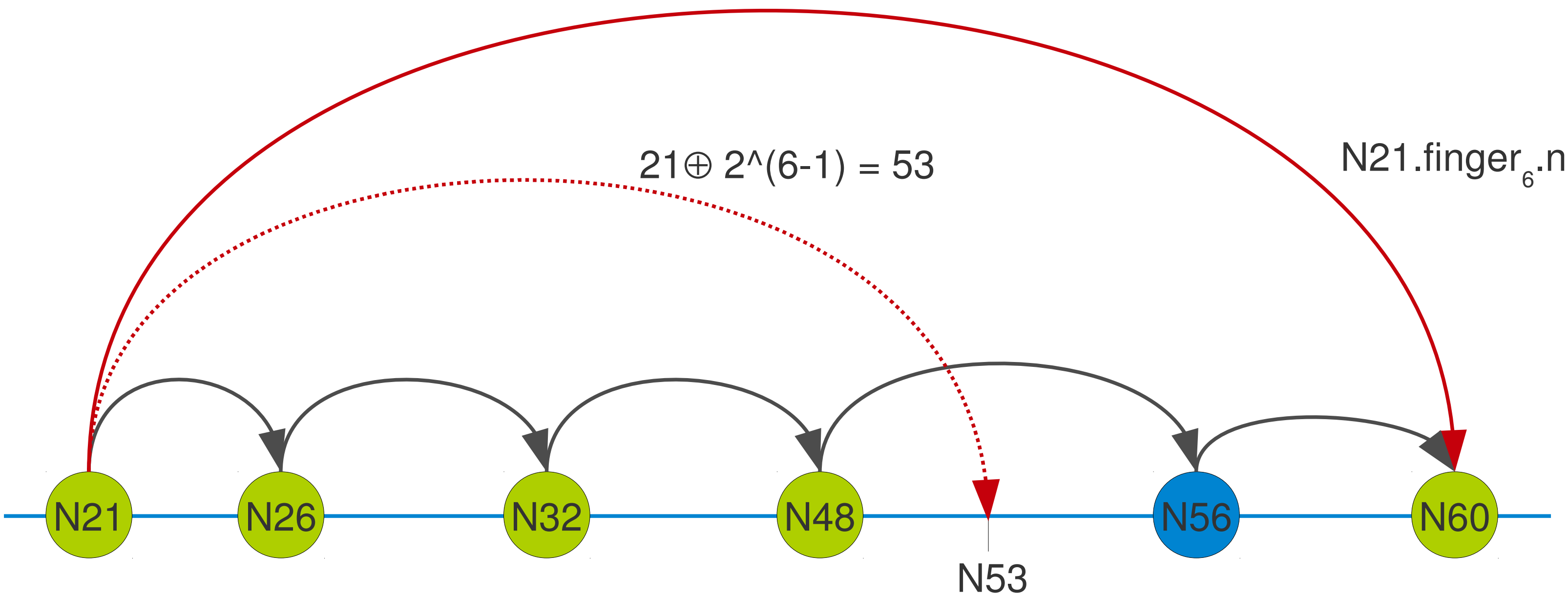
Chord – Fixing Fingers (2/4)

- Succ(21⊕ 2^(6-1)) = 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, however and nothing changes.



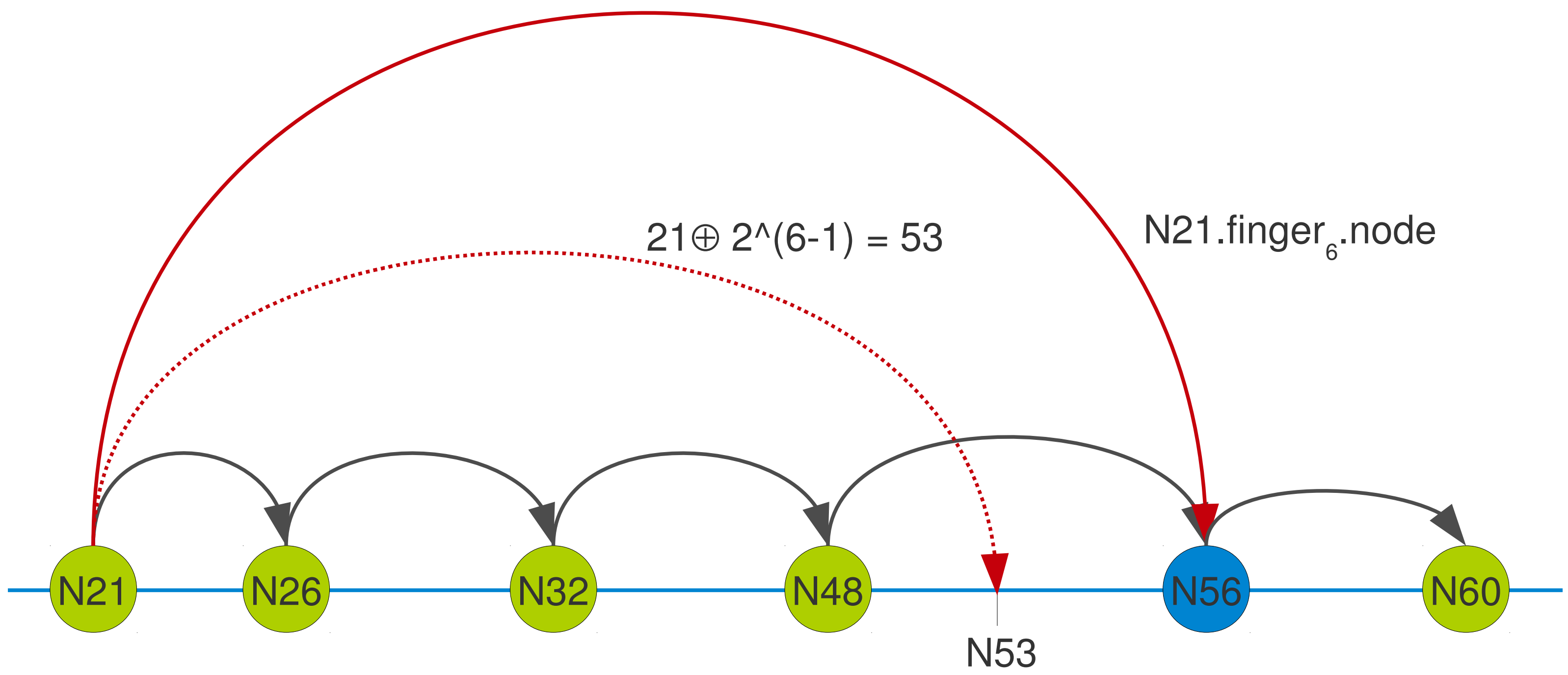
Chord – Fixing Fingers (3/4)

- Succ(21⊕ 2^(6-1)) = Succ(53) = ?
- N48 will eventually stabilize its successor.
- This means the ring is correct now.



Chord – Fixing Fingers (4/4)

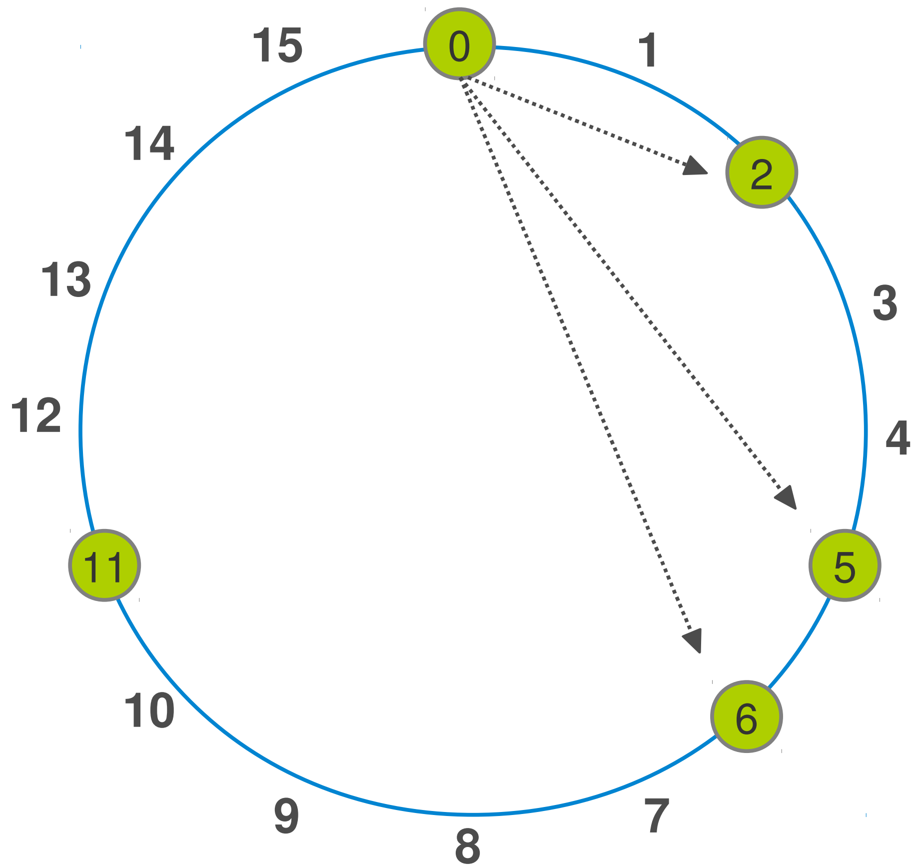
- Succ(21⊕ 2^(6-1)) = Succ(53) = N56
- When N21 tries to fix Finger 6 again, this time the response from N48 will be correct and N21 corrects the finger.



Handling Failure?

Successor List

- 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)+1)
- How big should r be?
  - log(N)



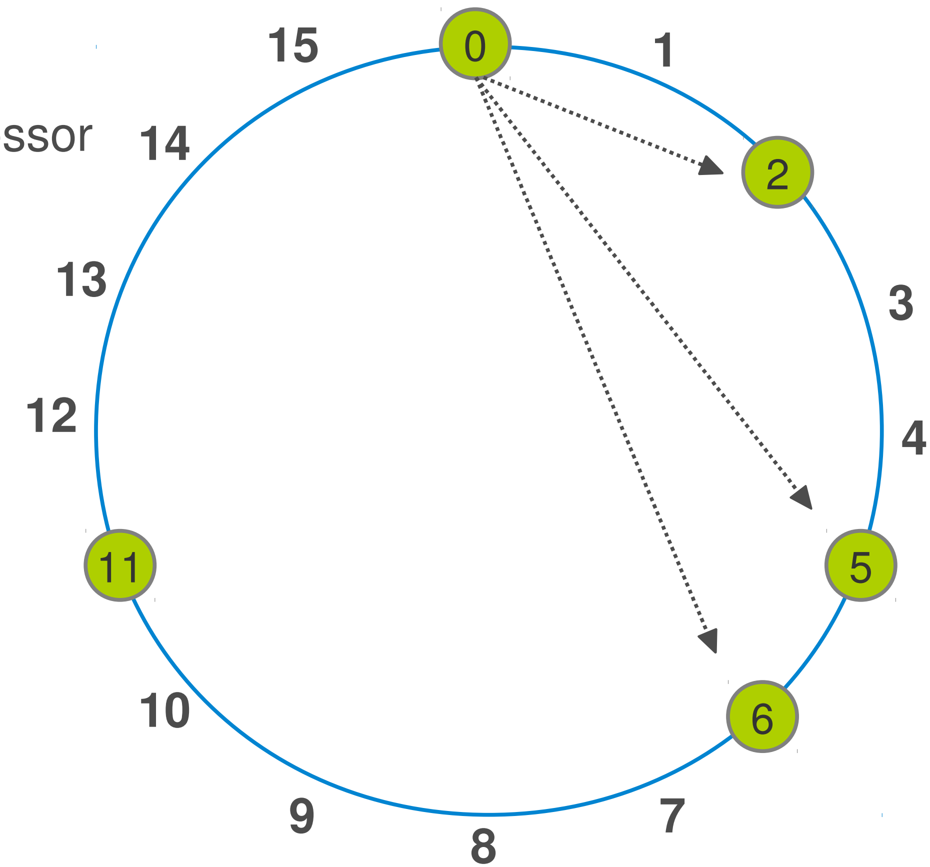
Successor List ...

```
// join a Chord ring containing node m
procedure n.join(m) {
  pred := nil
  Succ := m.findSuccessor(n)
  updateSuccesorList(succ.successorList)
}
```

```
// Periodically at n
procedure n.stabilize() {
  succ := find first alive node in successor list
  v := succ.pred
  if (v ≠ nil and v ∈ (n,succ]) then
     set succ := v
  send a notify(n) to succ
  updateSuccessorList(succ.successorList)
}
```

Dealing with Failures

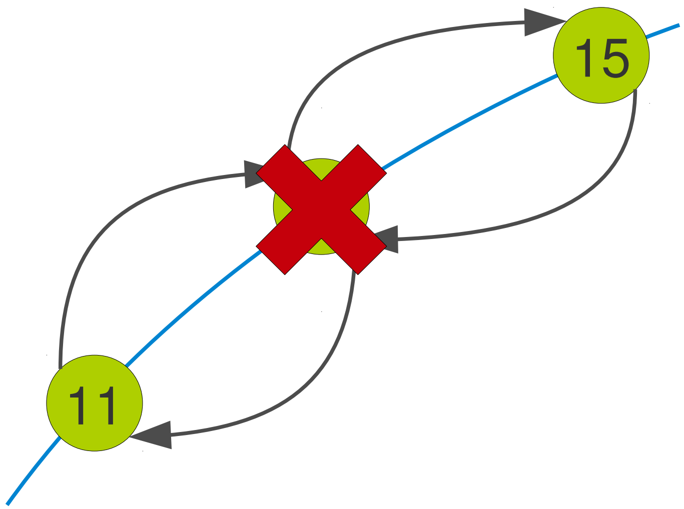
- Periodic stabilization
- If successor fails
  - Replace with closest alive successor
- If predecessor fails
  - Set pred to nil



Chord – Handling Failure (1/5)

- When n leaves Just disappear (like failure).
- When pred detected failed Set pred to nil.
- When succ detected failed Set succ to closest alive in successor list.

```
// Periodically at n:
v := succ.pred
if (v ≠ nil and v ∈ (n,succ]) then
   set succ := v
send a notify(n) to succ
```



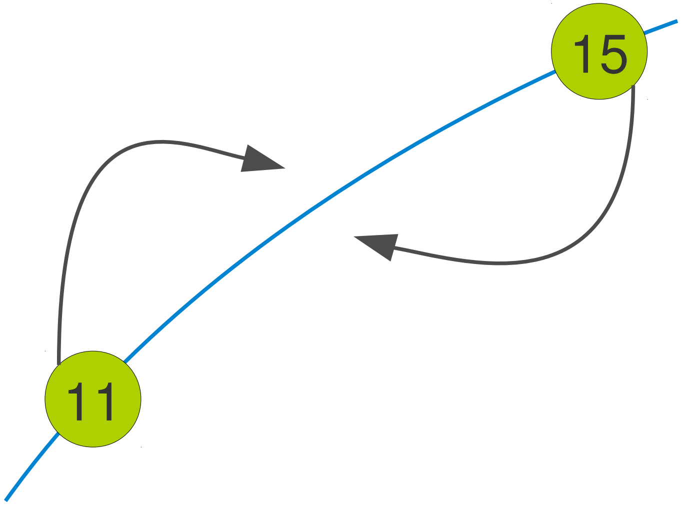

```
// When receiving notify(p) at n:
if (pred = nil or p ∈ (pred, n]) then
   set pred := p
```

```
procedure n.checkPredecessor() {
  if predecessor has failed then
     predecessor := nil
}
```

Chord – Handling Failure (2/5)

- When n leaves Just disappear (like failure).
- When pred detected failed Set pred to nil.
- When succ detected failed Set succ to closest alive in successor list.

```
// Periodically at n:
v := succ.pred
if (v ≠ nil and v ∈ (n,succ]) then
    set succ := v
send a notify(n) to succ
```

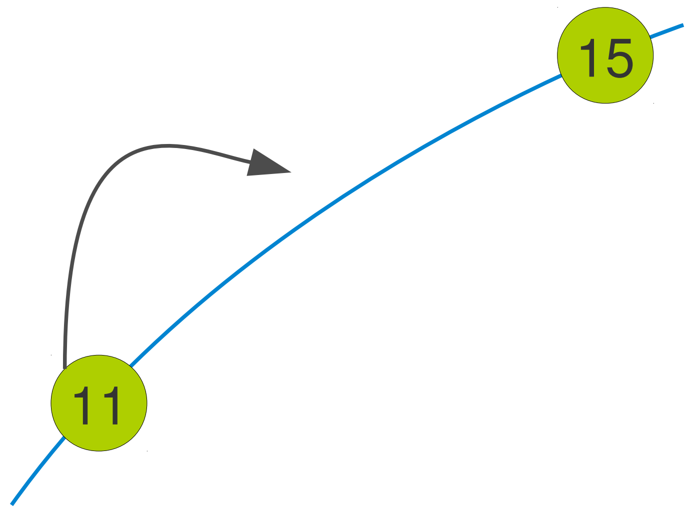



```
// When receiving notify(p) at n:
if (pred = nil or p ∈ (pred, n]) then
    set pred := p
```

```
procedure n.checkPredecessor() {
  if predecessor has failed then
      predecessor := nil
}
```

Chord – Handling Failure (3/5)

- When n leaves Just disappear (like failure).
- When pred detected failed Set pred to nil.
- When succ detected failed Set succ to closest alive in successor list.




```
// Periodically at n:
v := succ.pred
if (v ≠ nil and v ∈ (n,succ]) then
    set succ := v
send a notify(n) to succ
```

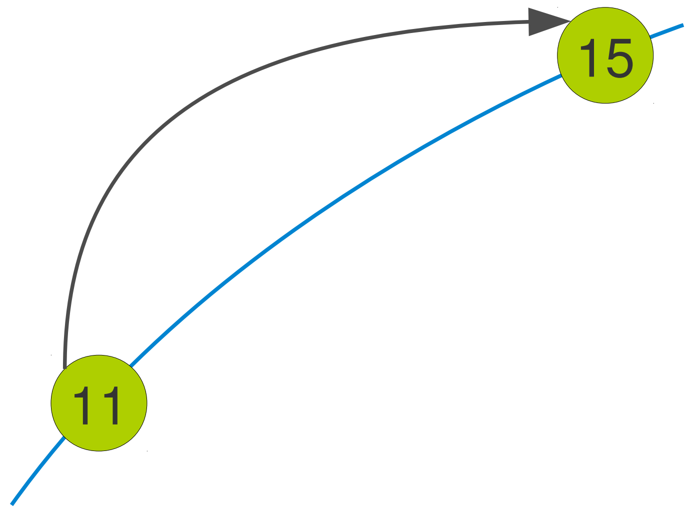
```
// When receiving notify(p) at n:
if (pred = nil or p ∈ (pred, n]) then
    set pred := p
```

```
procedure n.checkPredecessor() {
  if predecessor has failed then
      predecessor := nil
}
```

Chord – Handling Failure (4/5)

- When n leaves Just disappear (like failure).
- When pred detected failed Set pred to nil.
- When succ detected failed Set succ to closest alive in successor list.

```
// Periodically at n:
v := succ.pred
if (v ≠ nil and v ∈ (n,succ]) then
    set succ := v
send a notify(n) to succ
```



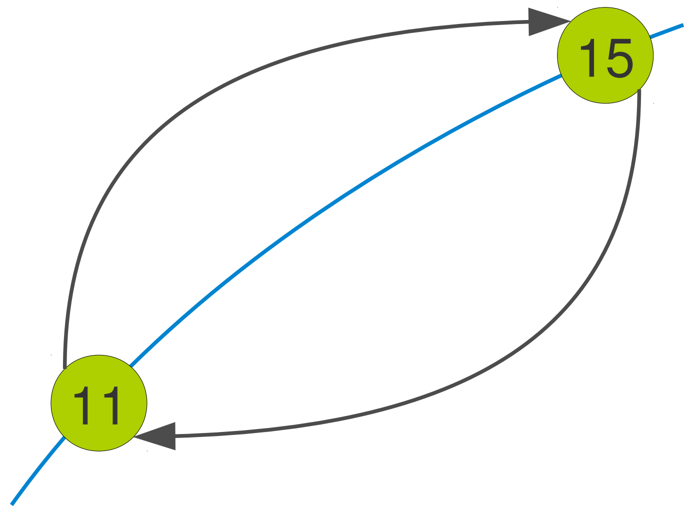

```
// When receiving notify(p) at n:
if (pred = nil or p ∈ (pred, n]) then
    set pred := p
```

```
procedure n.checkPredecessor() {
  if predecessor has failed then
      predecessor := nil
}
```

Chord – Handling Failure (5/5)

- When n leaves Just disappear (like failure).
- When pred detected failed Set pred to nil.
- When succ detected failed Set succ to closest alive in successor list.

```
// Periodically at n:
v := succ.pred
if (v ≠ nil and v ∈ (n,succ]) then
    set succ := v
send a notify(n) to succ
```




```
// When receiving notify(p) at n:
if (pred = nil or p ∈ (pred, n]) then
    set pred := p
```

```
procedure n.checkPredecessor() {
  if predecessor has failed then
      predecessor := nil
}
```

Variations of Chord

Variations of Chord

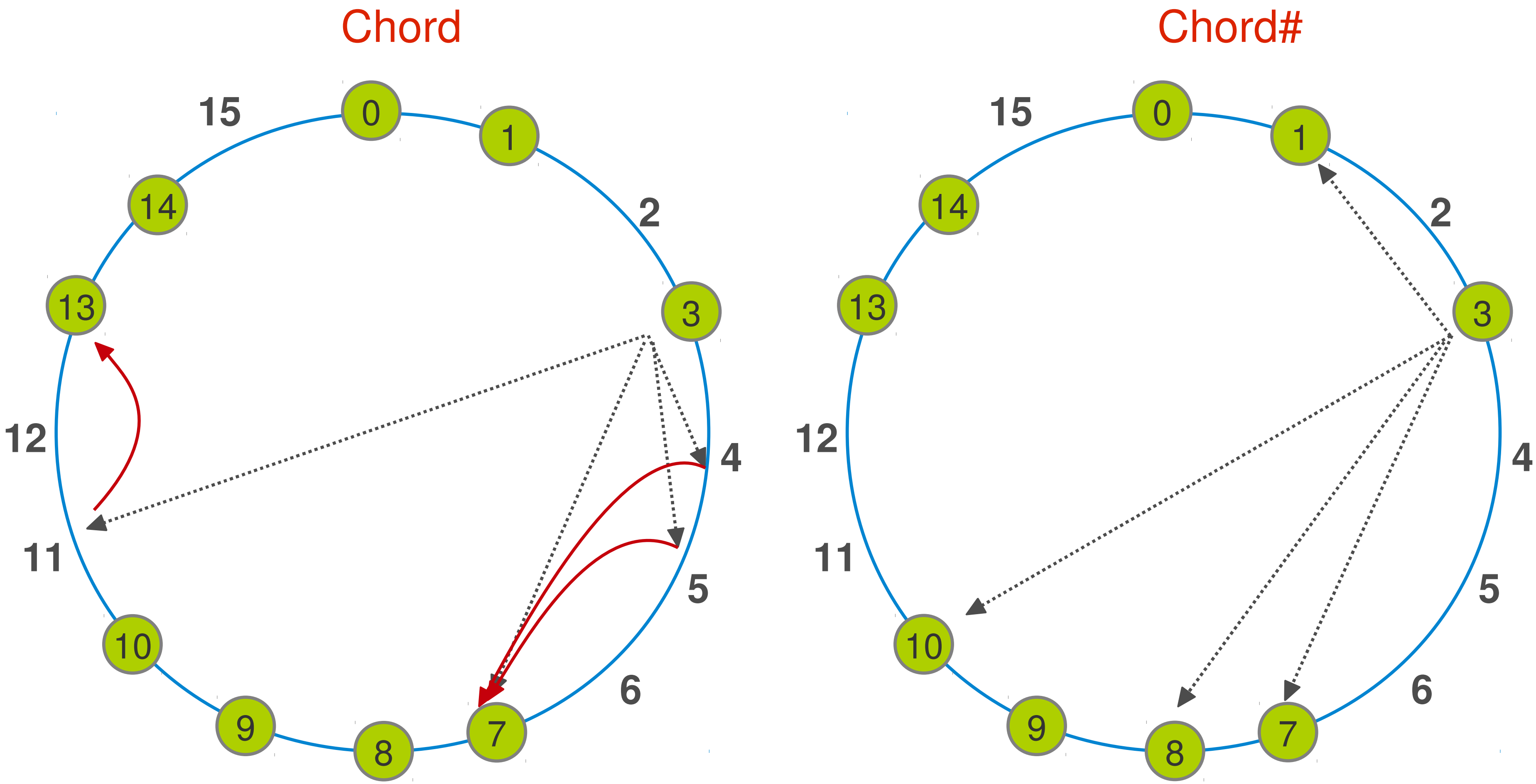
- Chord#
- DKS

Chord#

- The routing table has exponentially increasing pointers on the ring (node space) and NOT the identifier space.

$$pointer_i = \begin{cases} successor & : i = 0 \\ pointer_{i-1} \, . \, pointer_{i-1} & : i \neq 0 \end{cases}$$

Chord vs. Chord#



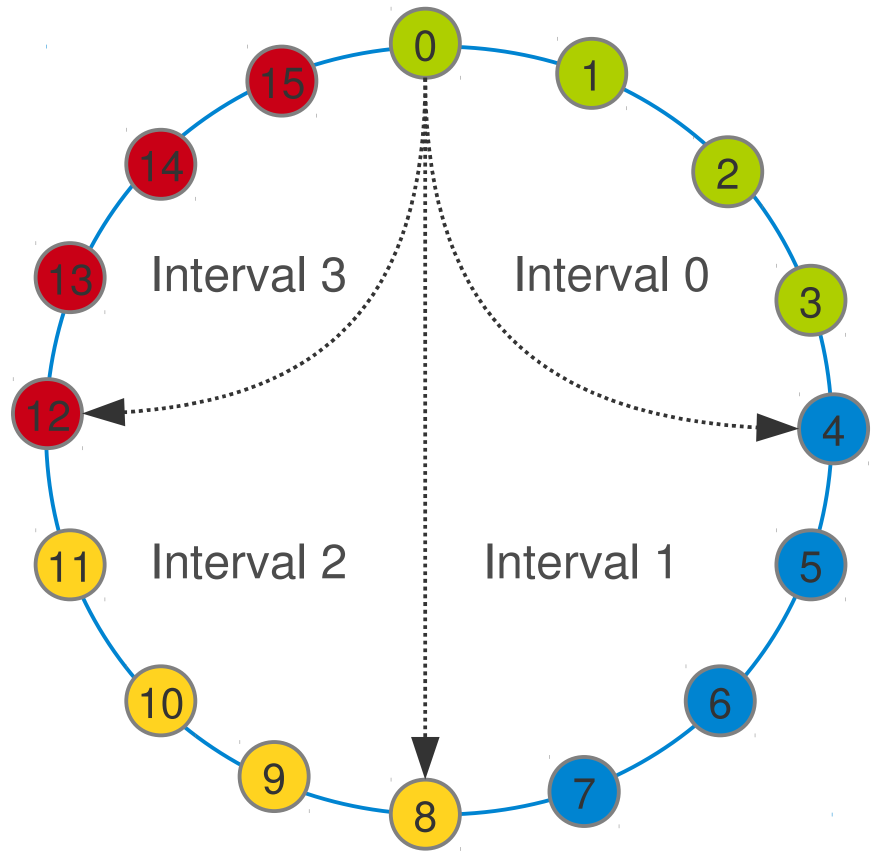
DKS

- Generalization of Chord to provide arbitrary arity
- Provide $\log_k(n)$ hops per lookup
  - k being a configurable parameter
  - n being the number of nodes
- Instead of only $\log_2(n)$

DKS – Lookup

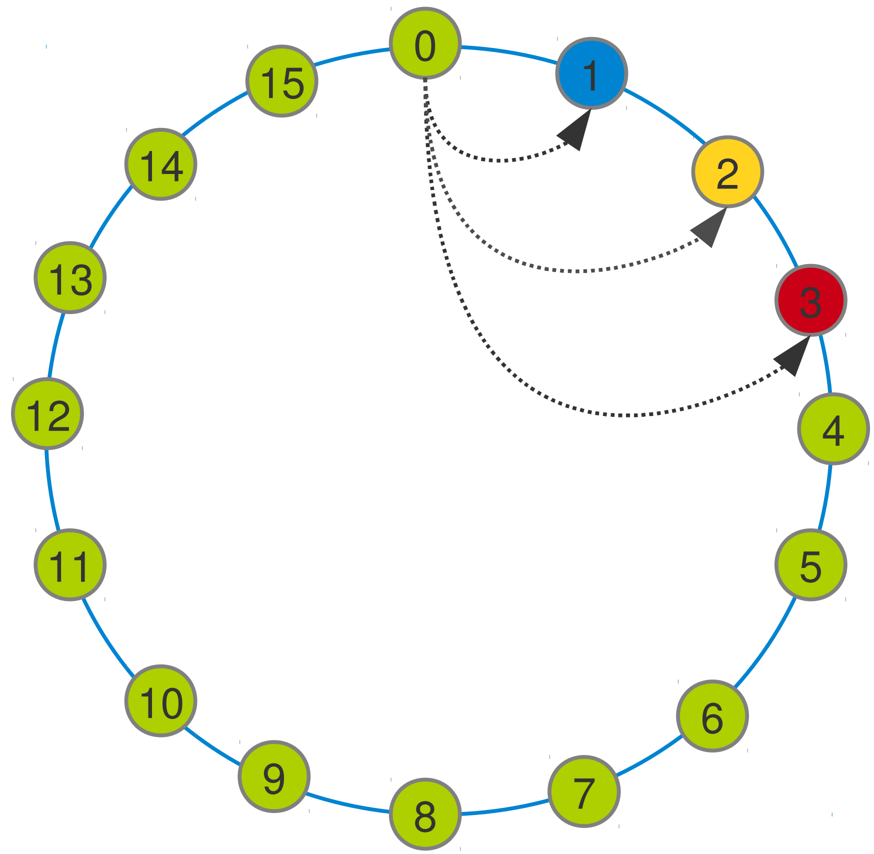
- Achieving $\log_k(n)$ lookup
- Each node contains $\log_k(N)=L$ levels, $N=k^L$
- Each level contains k intervals,
- Example, k=4, N=16 (4^2), node 0

Node 0	I0	I1	I2	I3
Level 1	0 ... 3	4 ... 7	8 ... 11	12 ... 15



DKS – Lookup

- Achieving $\log_k(n)$ lookup
- Each node contains $\log_k(N)=L$ levels, $N=k^L$
- Each level contains k intervals,
- Example, k=4, N=16 (4^2), node 0



Node 0	I0	I1	I2	I3
Level 1	0 ... 3	4 ... 7	8 ... 11	12 ... 15
Level 2	0	1	2	3

Summary

Summary

- Pointer of the nodes:
  - Successor: first clockwise node
  - Predecessor: first anti-clockwise node
  - Finger list: successor(n + 2^(i-1)) for i = 1... M (N = 2^M).
- Handling dynamism
  - Periodic stabilization
- Handling failure
  - Successor list
  - Periodic stabilization