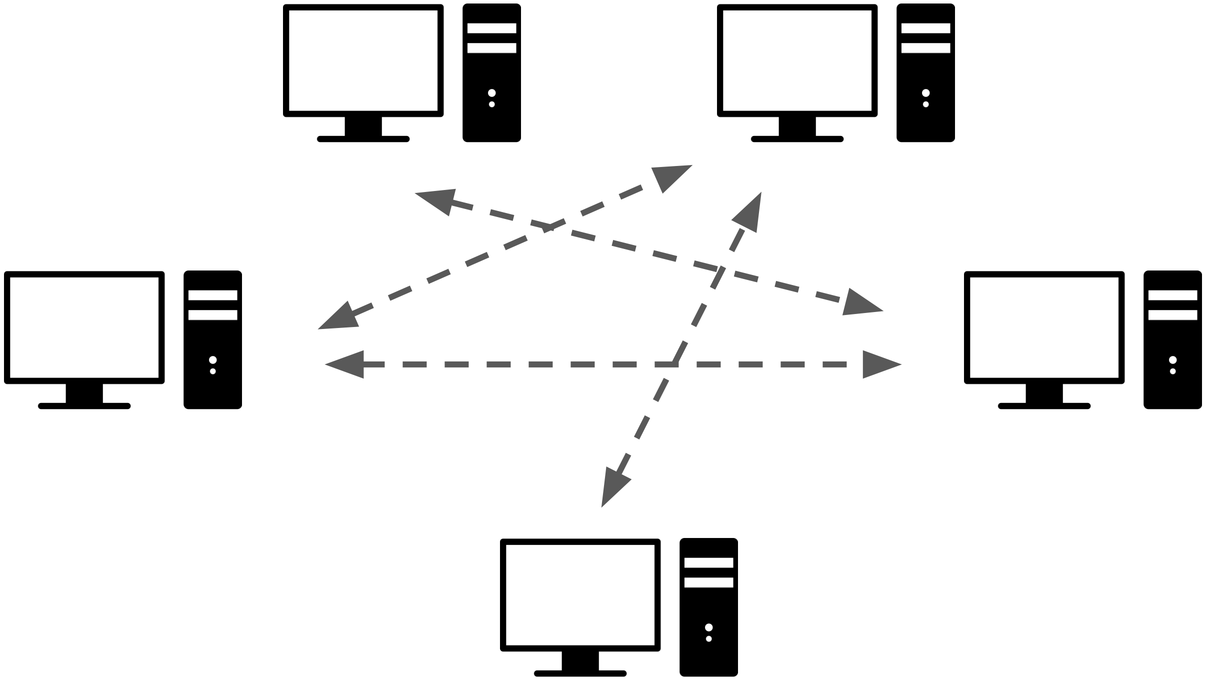
P2P: Distributed Hash Tables

Chord + Routing Geometries

Nirvan Tyagi

CS 6410 Fall16

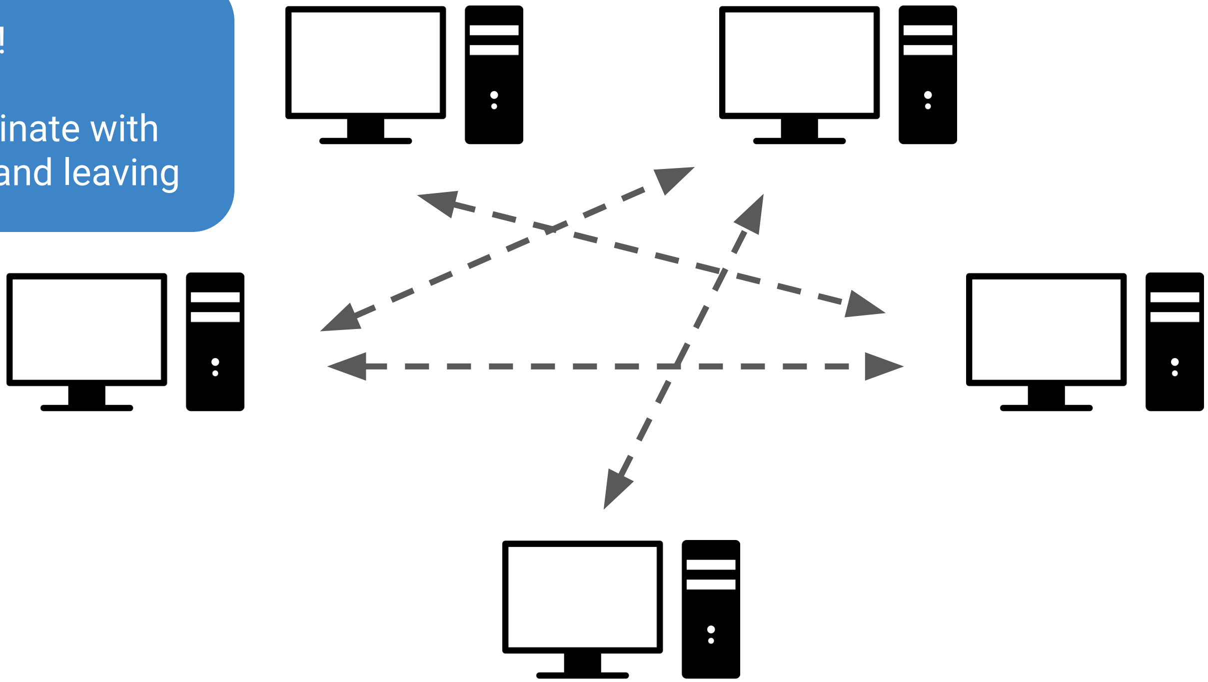
Peer-to-peer (P2P)



Peer-to-peer (P2P)

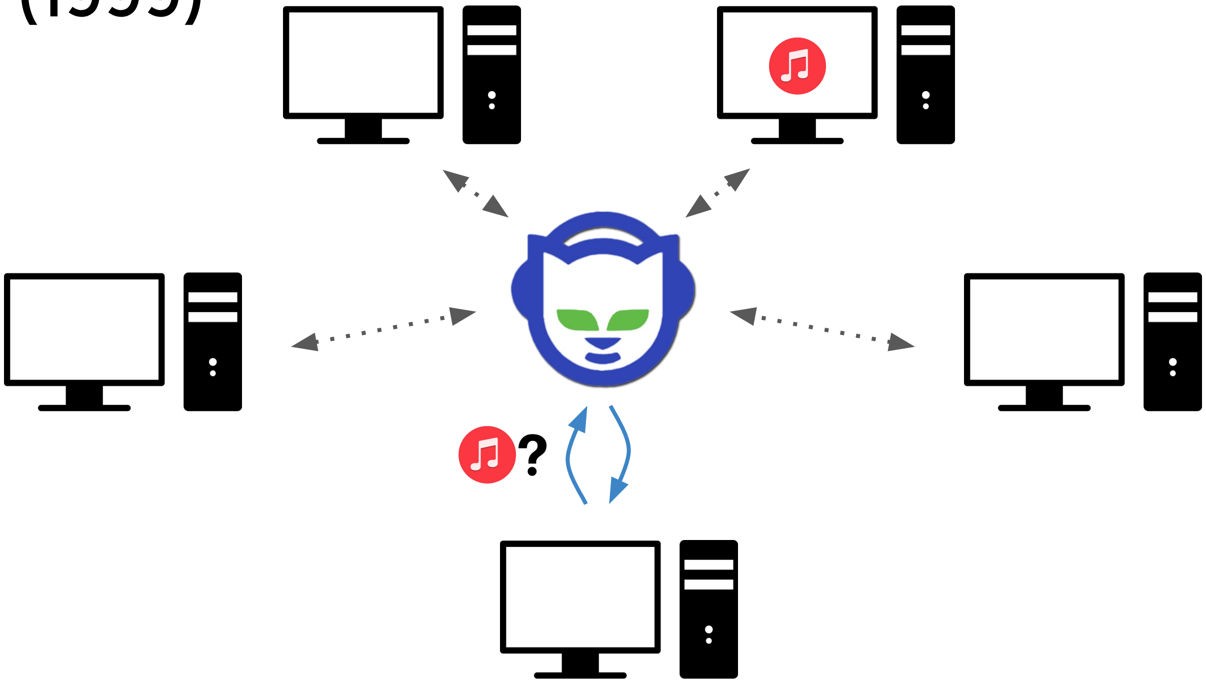
Decentralized!

Hard to coordinate with peers joining and leaving



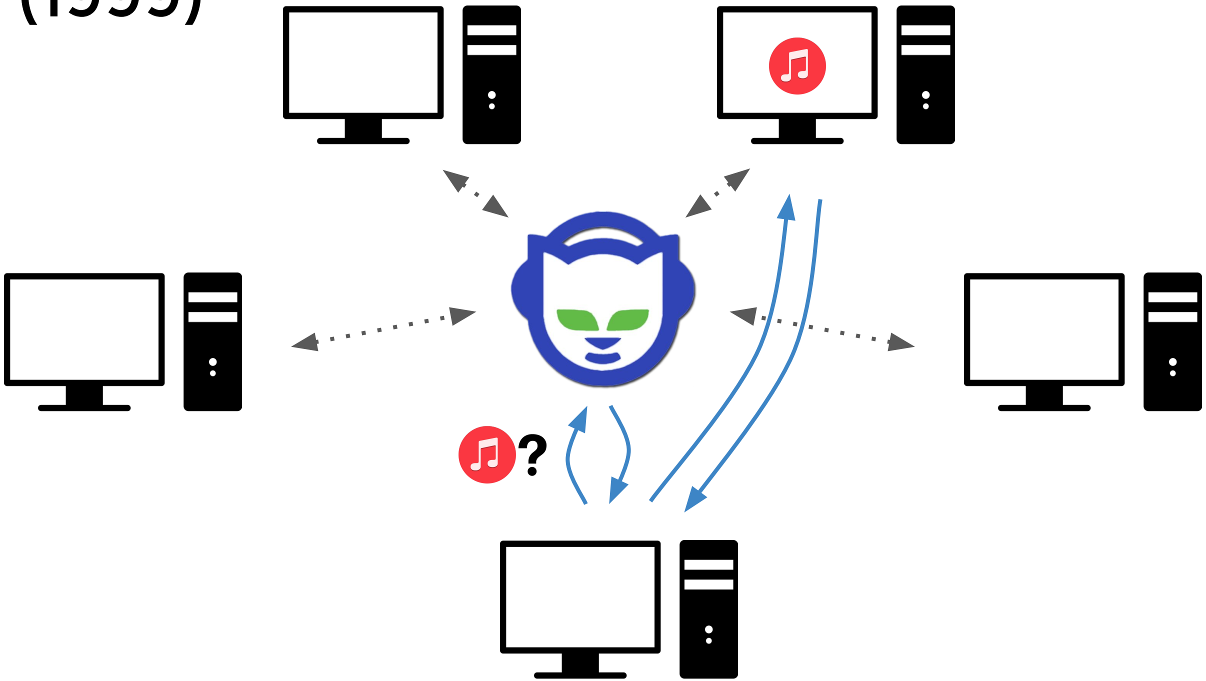
Peer-to-peer (P2P)

Napster (1999)



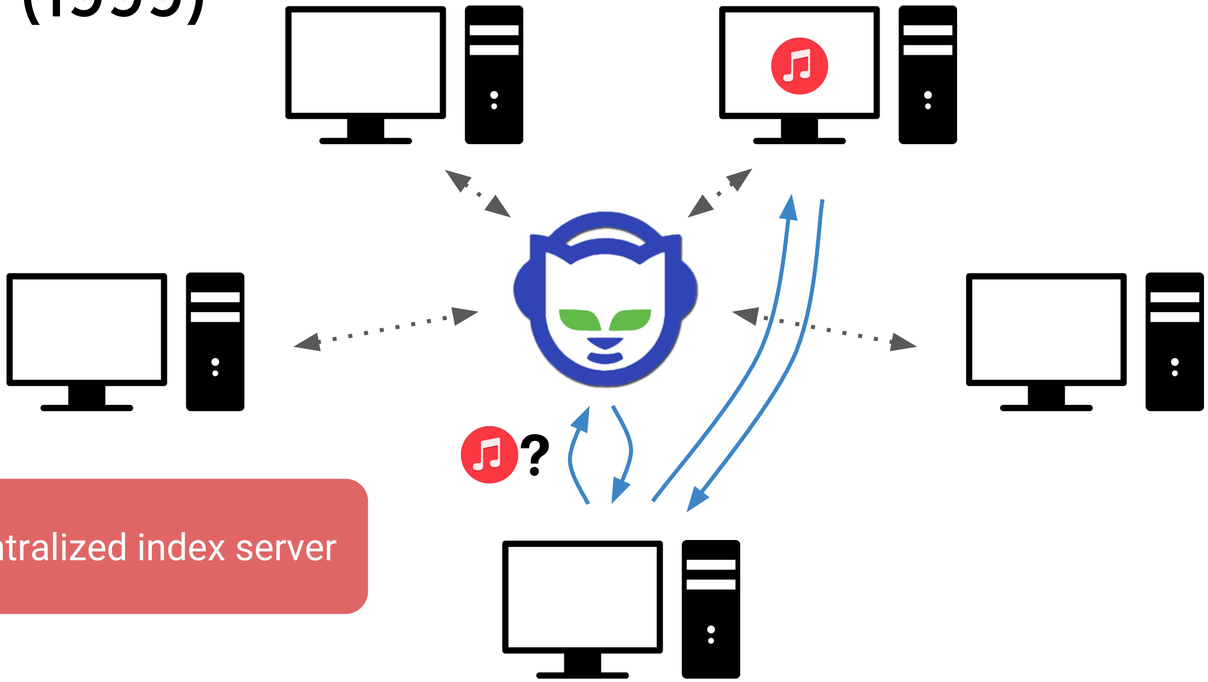
Peer-to-peer (P2P)

Napster (1999)



Peer-to-peer (P2P)

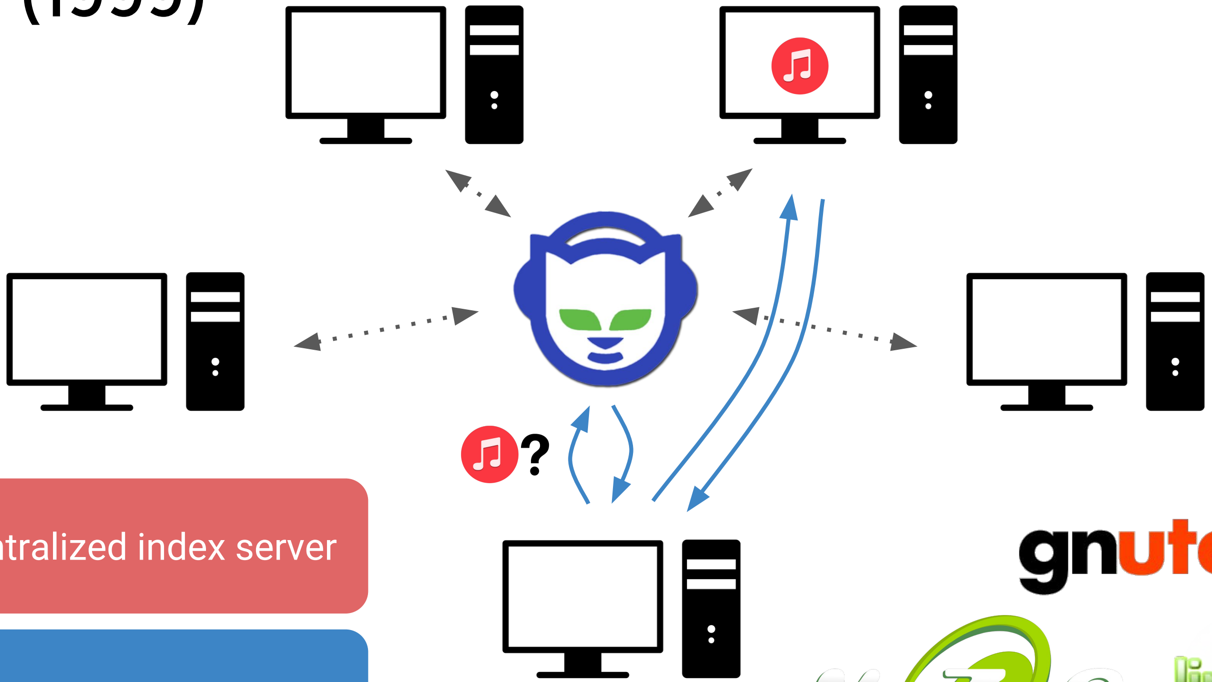
Napster (1999)



Problem - Centralized index server

Peer-to-peer (P2P)

Napster (1999)



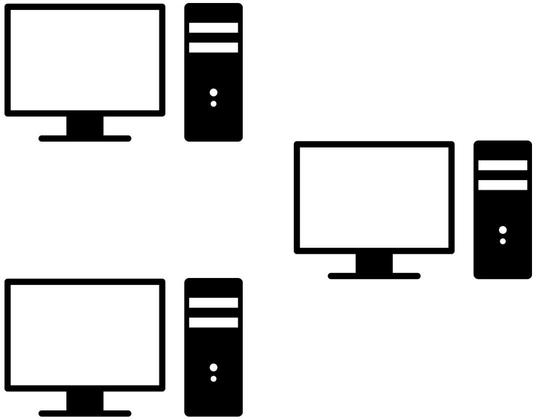
Problem - Centralized index server

Solution - Distributed hash table

Distributed Hash Tables (DHT)

Hash table

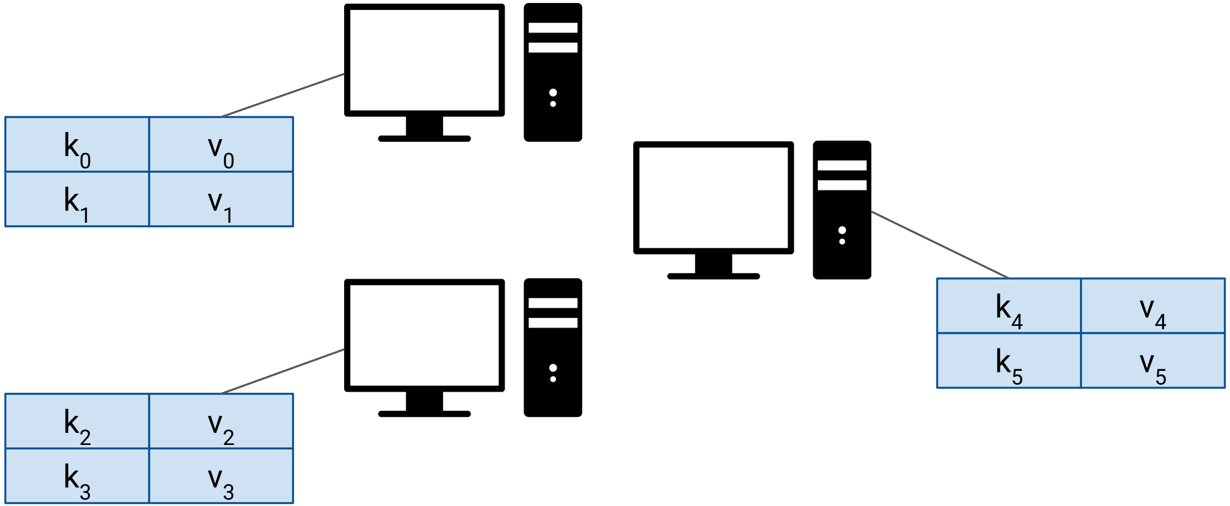
k_0	v_0
k_1	v_1
k_2	v_2
k_3	v_3
k_4	v_4
k_5	v_5



Distributed Hash Tables (DHT)

Hash table

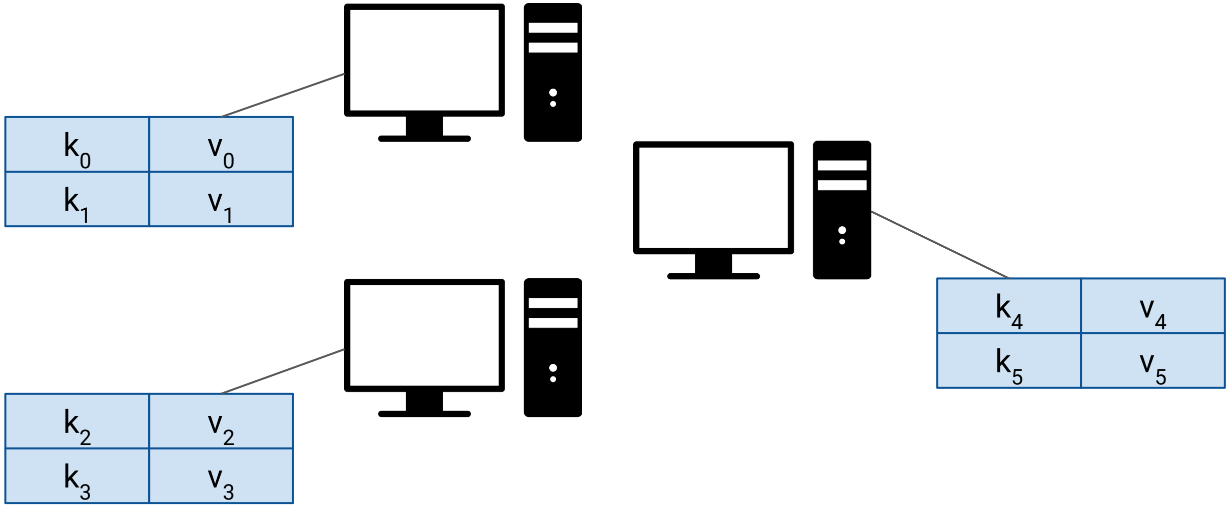
k_0	v_0
k_1	v_1
k_2	v_2
k_3	v_3
k_4	v_4
k_5	v_5



Distributed Hash Tables (DHT)

Desirable Properties

- Decentralization
- Load-balancing
- Scalability
- Availability



Outline

Chord

- Specific DHT protocol for P2P systems
- Simple, efficient

DHT Routing Geometry

- Effect of different DHT protocols on desirable system properties

Chord

A scalable P2P lookup service for internet applications

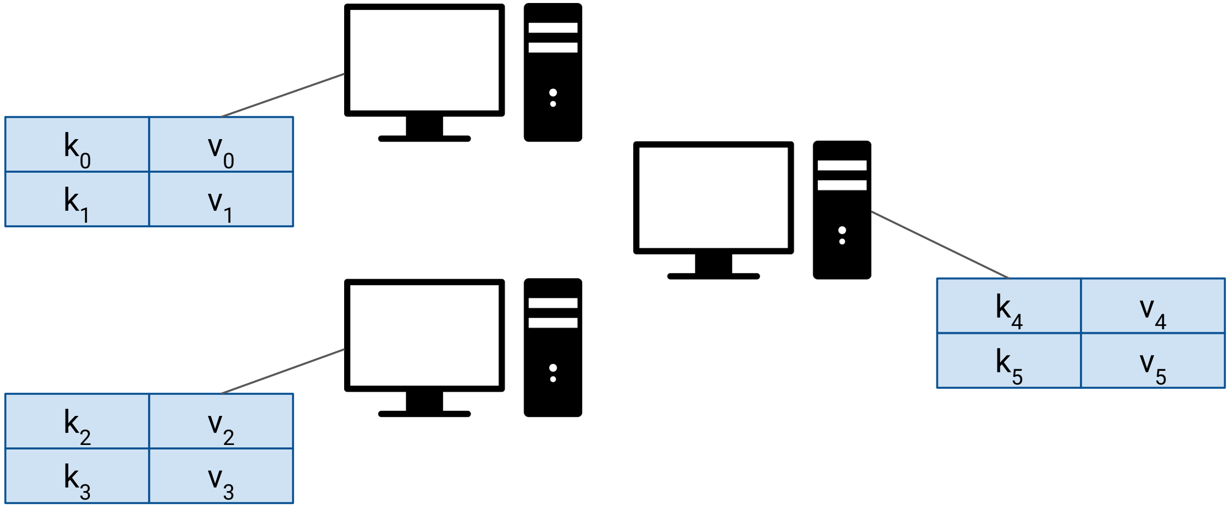
Ion Stoica, Robert Morris, David Karger
Frans Kaashoek, Hari Balakrishnan

Chord - Overview

How to assign keys to peers?

Hash table

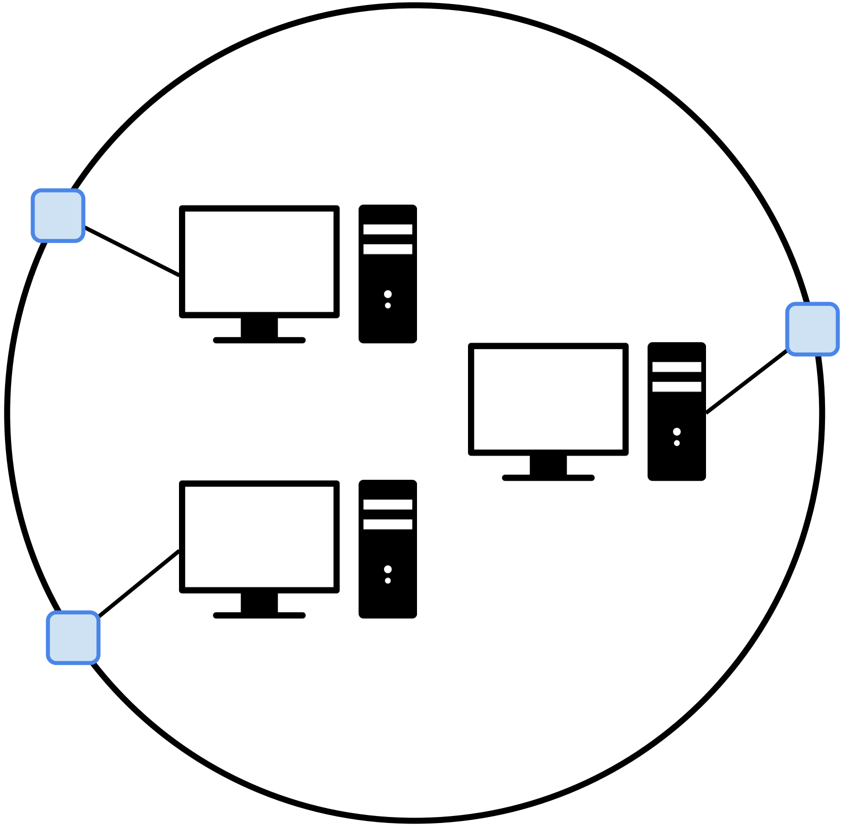
k_0	v_0
k_1	v_1
k_2	v_2
k_3	v_3
k_4	v_4
k_5	v_5



Chord - Overview

Hash table

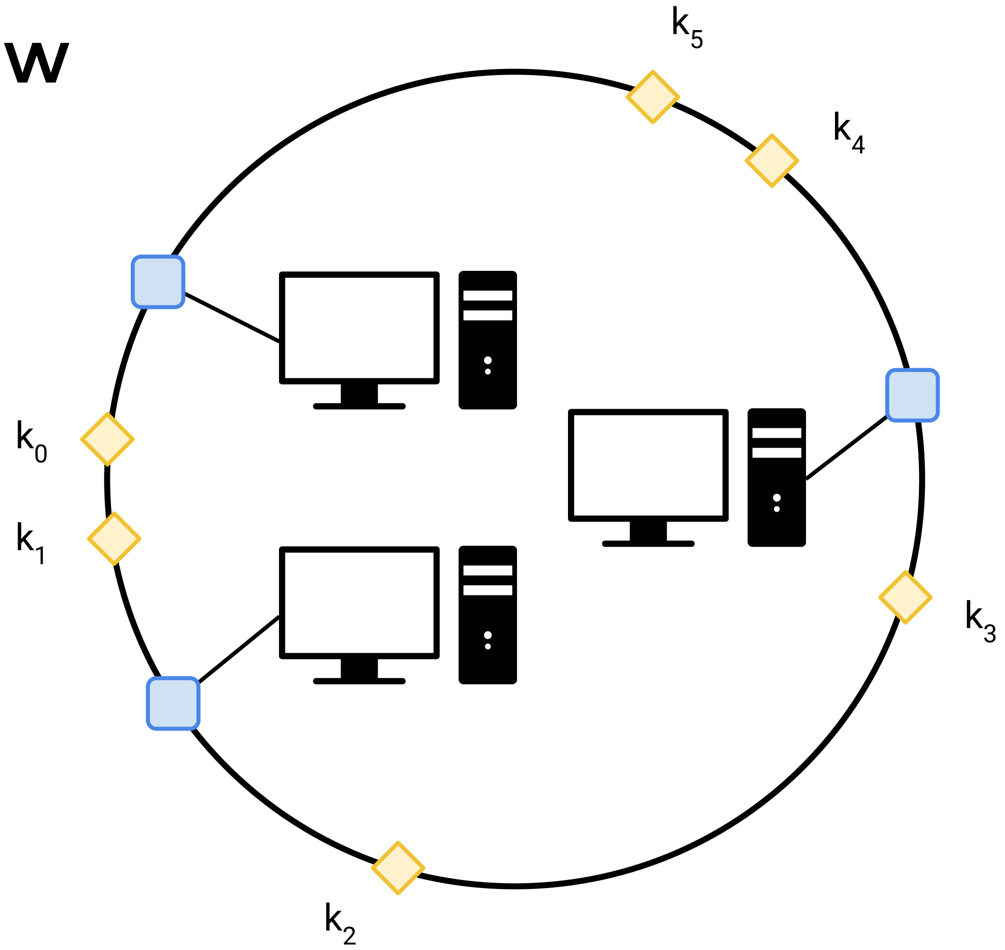
k_0	v_0
k_1	v_1
k_2	v_2
k_3	v_3
k_4	v_4
k_5	v_5



Chord - Overview

Hash table

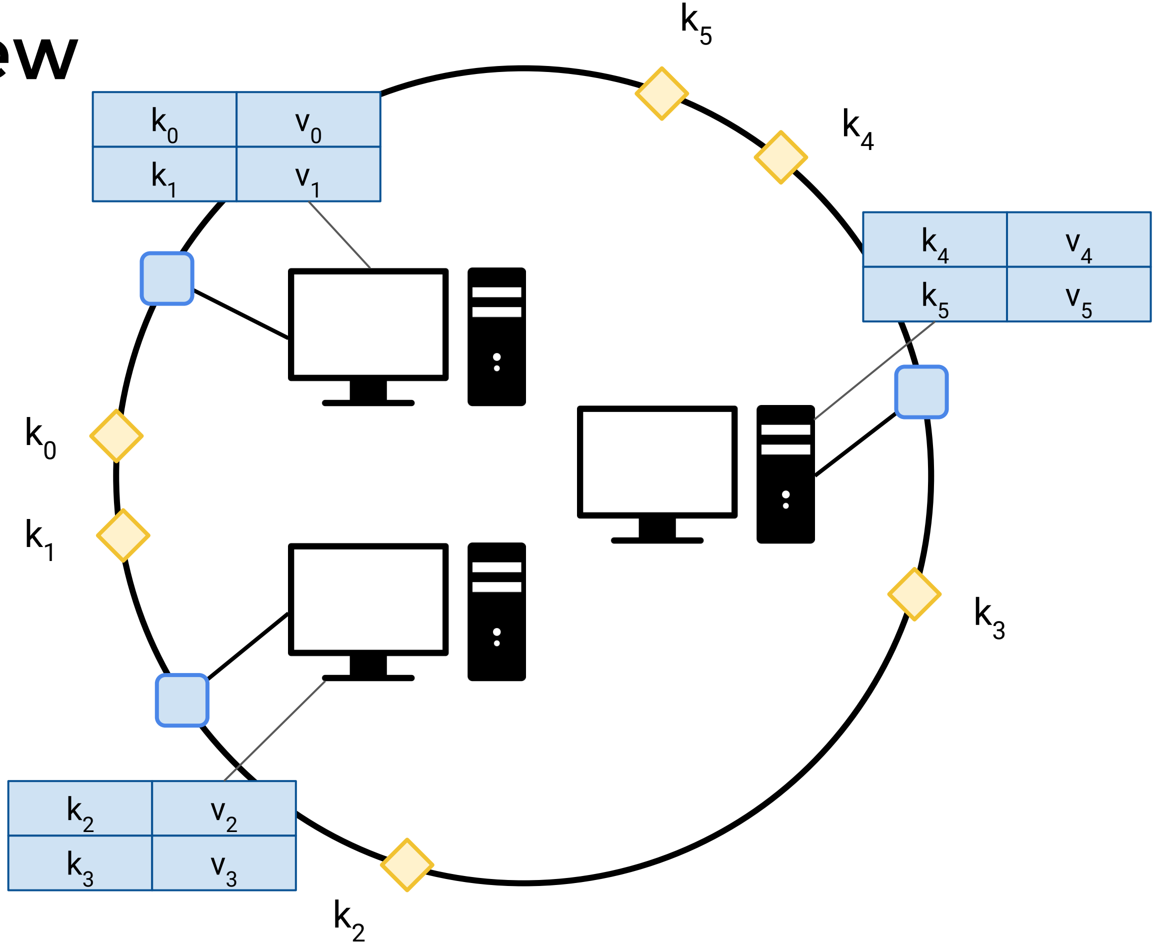
k_0	v_0
k_1	v_1
k_2	v_2
k_3	v_3
k_4	v_4
k_5	v_5



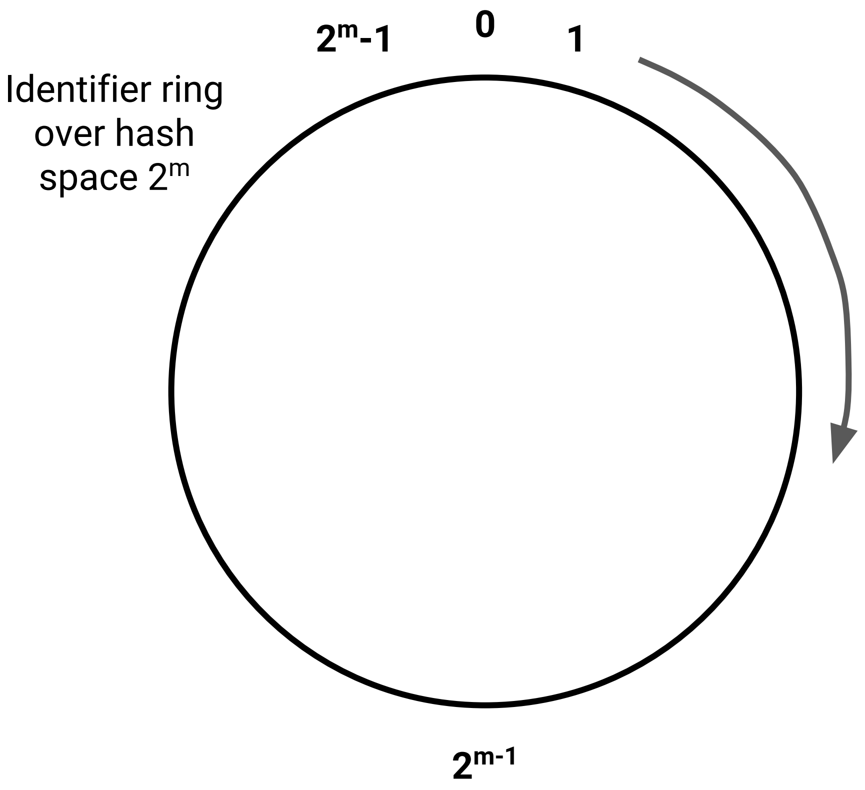
Chord - Overview

Hash table

k_0	v_0
k_1	v_1
k_2	v_2
k_3	v_3
k_4	v_4
k_5	v_5

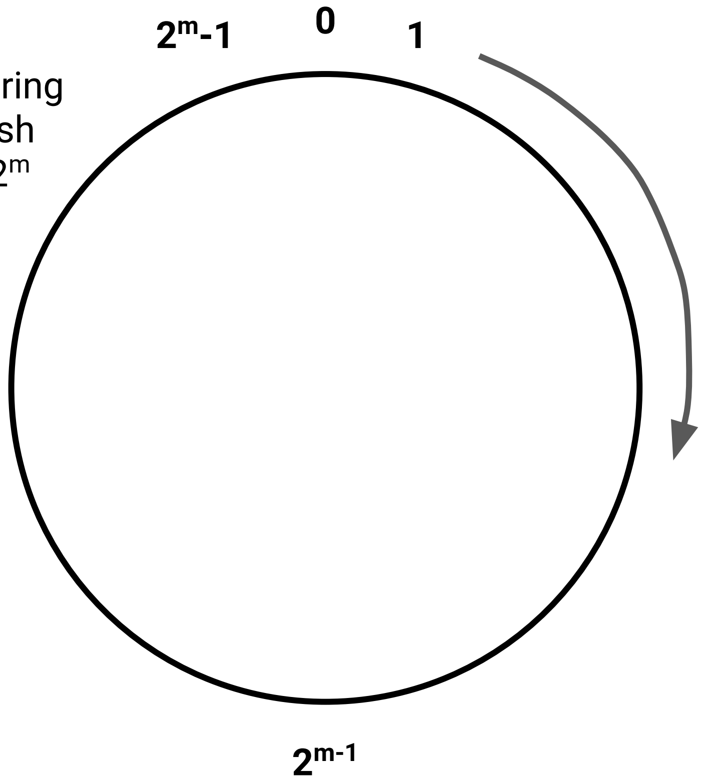


Chord - Overview



Chord - Overview

Identifier ring over hash space 2^m



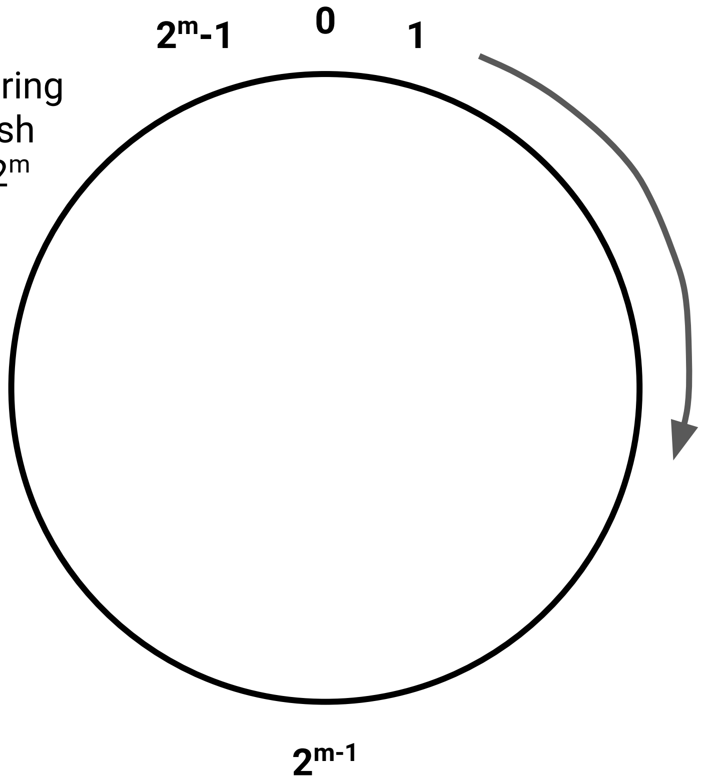
 = node

 = key

node id = hash(node)
key id = hash(key)

Chord - Overview

Identifier ring over hash space 2^m



 = node

 = key

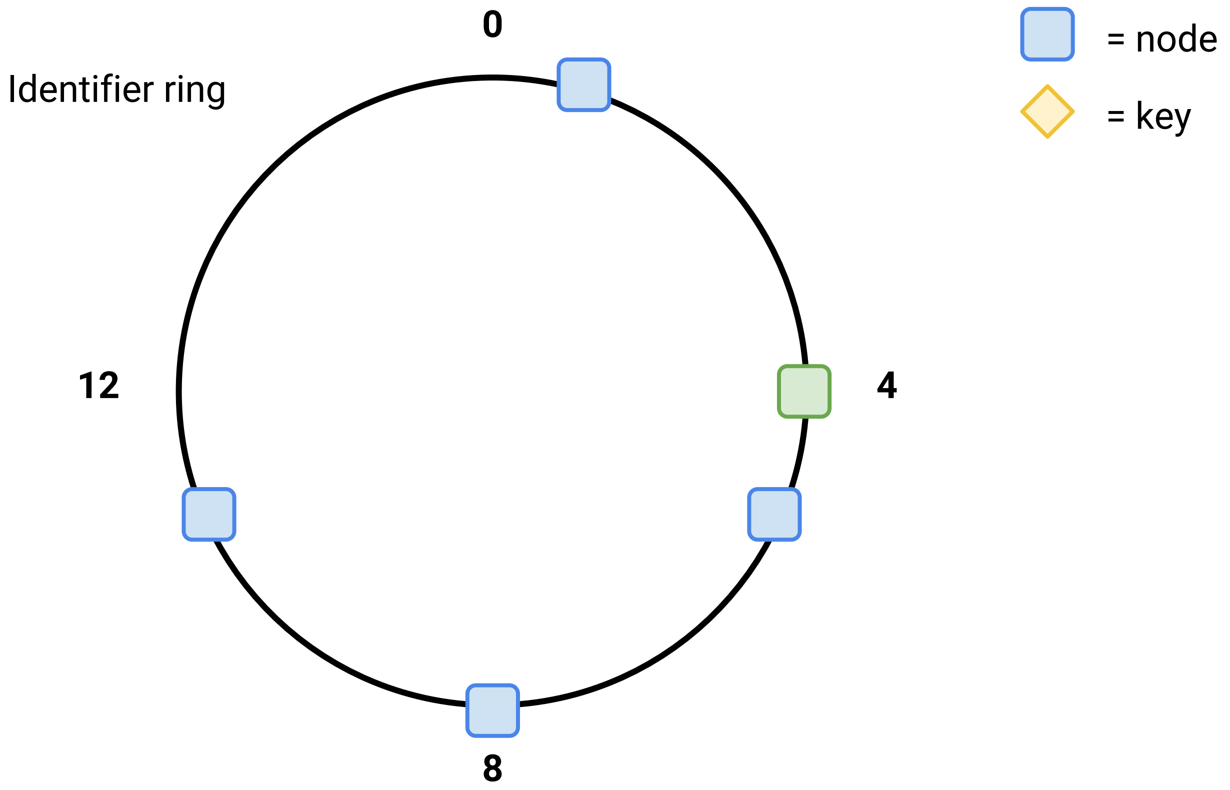
node id = hash(node)
key id = hash(key)

successor(id)

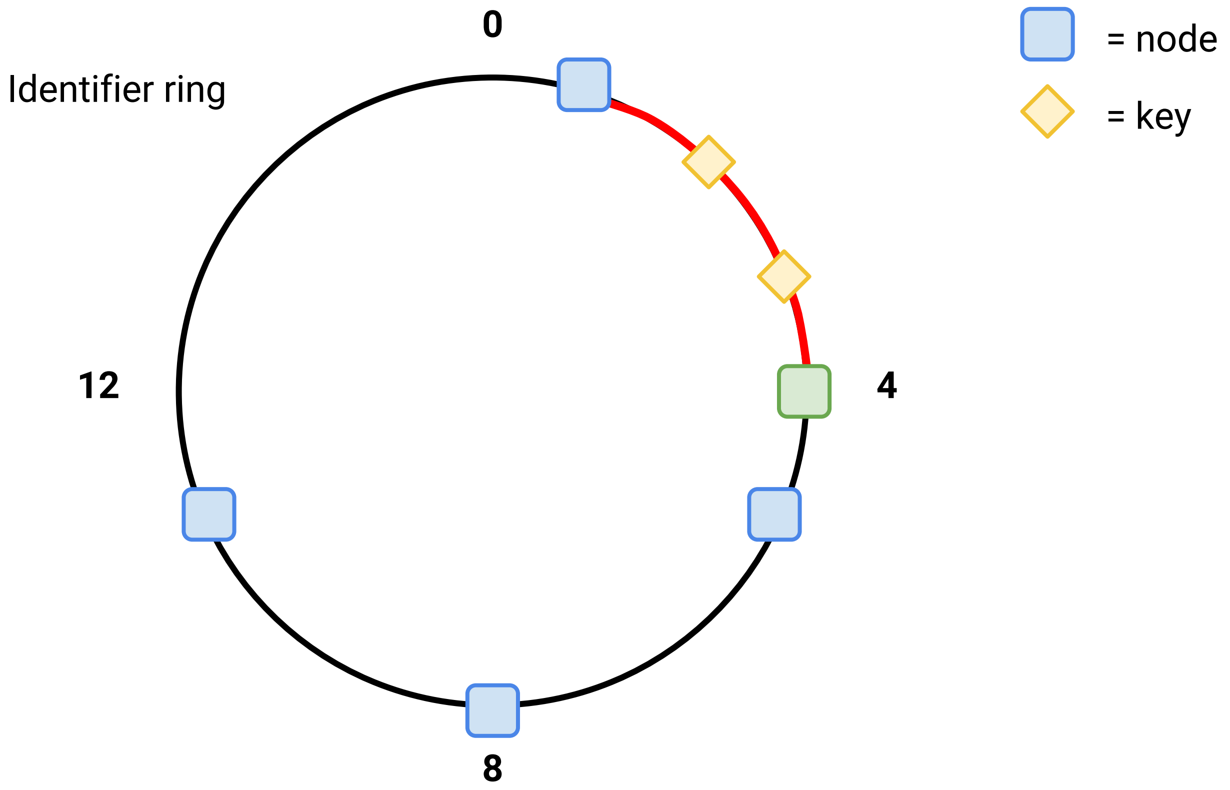
finger table for node at id *i*

finger	node id
1	succ(*i*)
2	succ(*i* + 2)
j	succ($i + 2^{j-1}$)

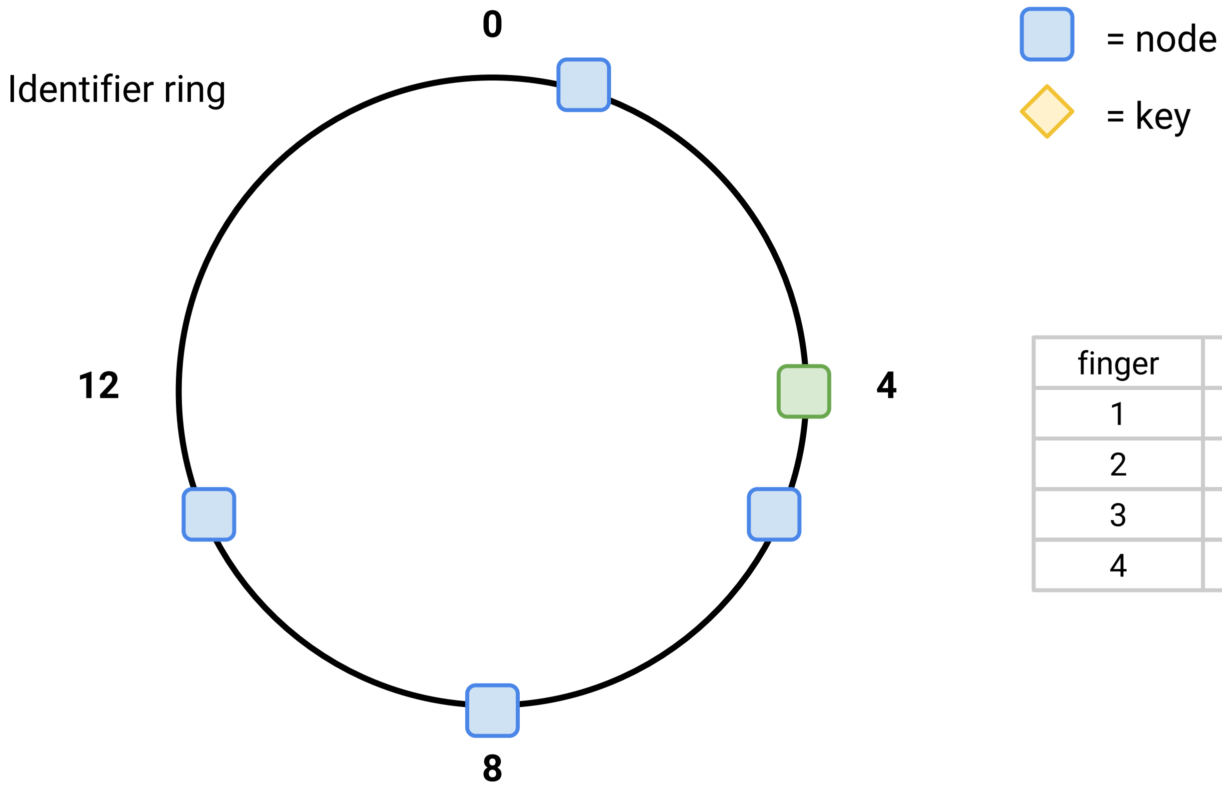
Chord - Overview



Chord - Overview

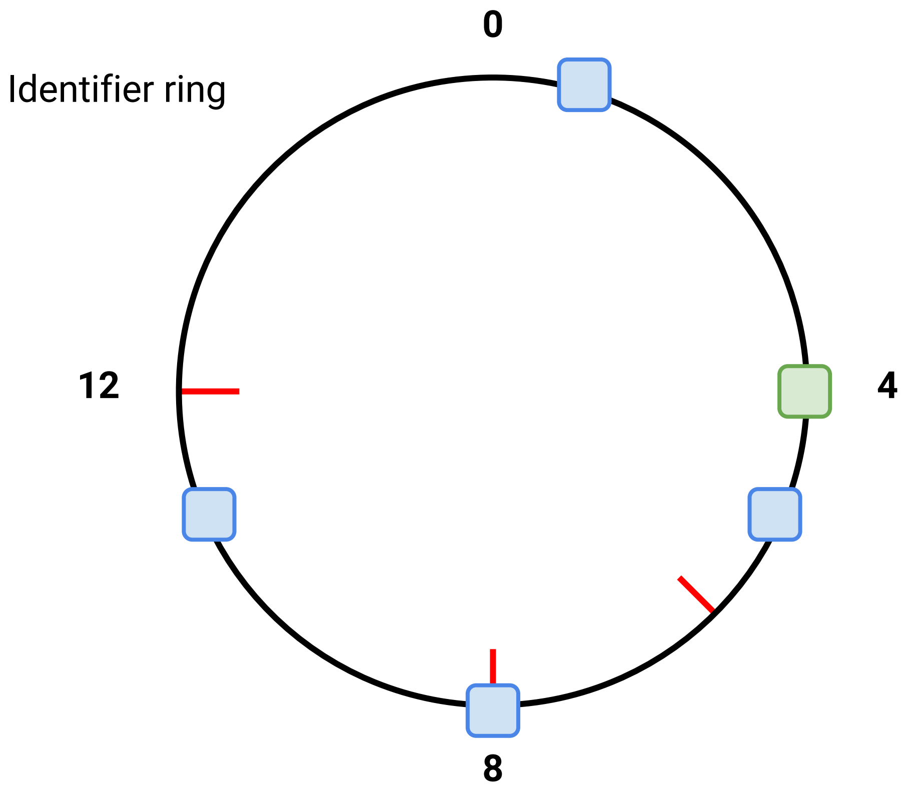


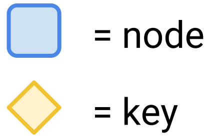
Chord - Overview



finger	node id
1	succ(i)
2	succ(i + 2)
3	succ($i + 2^2$)
4	succ($i + 2^3$)

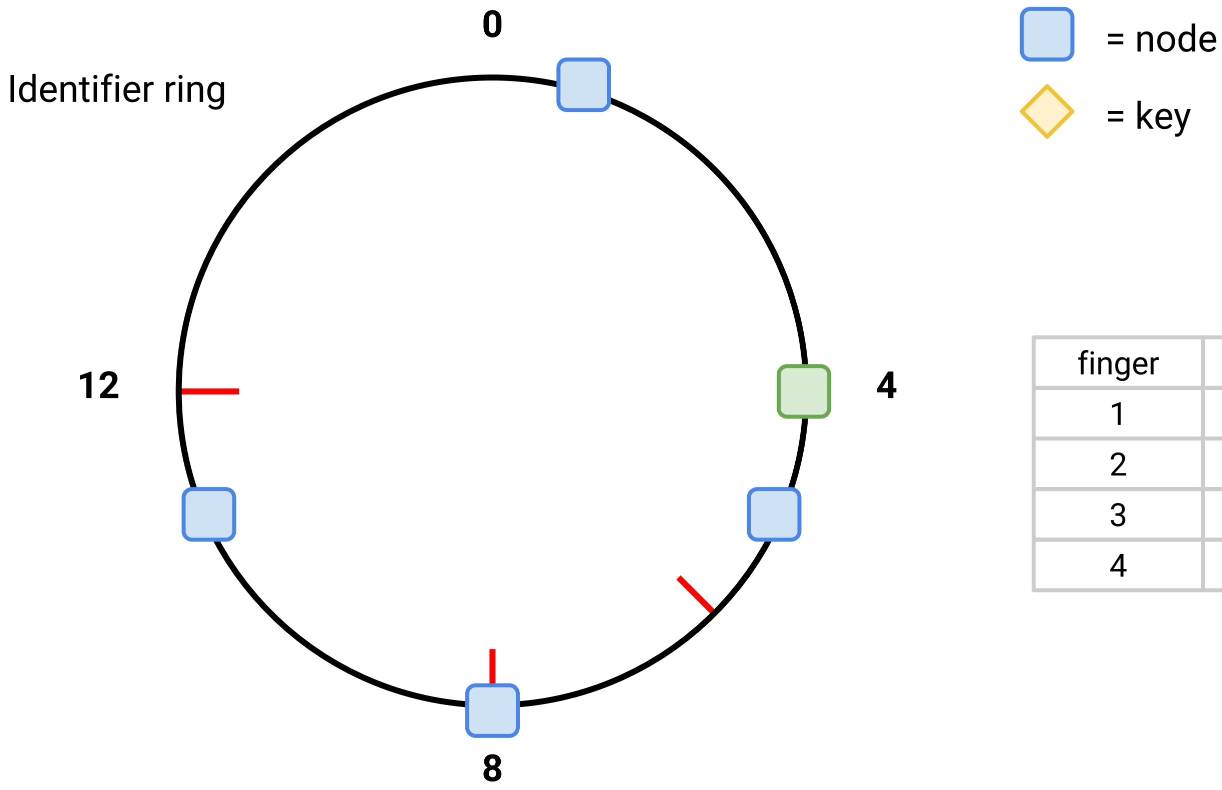
Chord - Overview





finger	node id
1	succ(4)
2	succ(4 + 2)
3	succ(4 + 2^2)
4	succ(4 + 2^3)

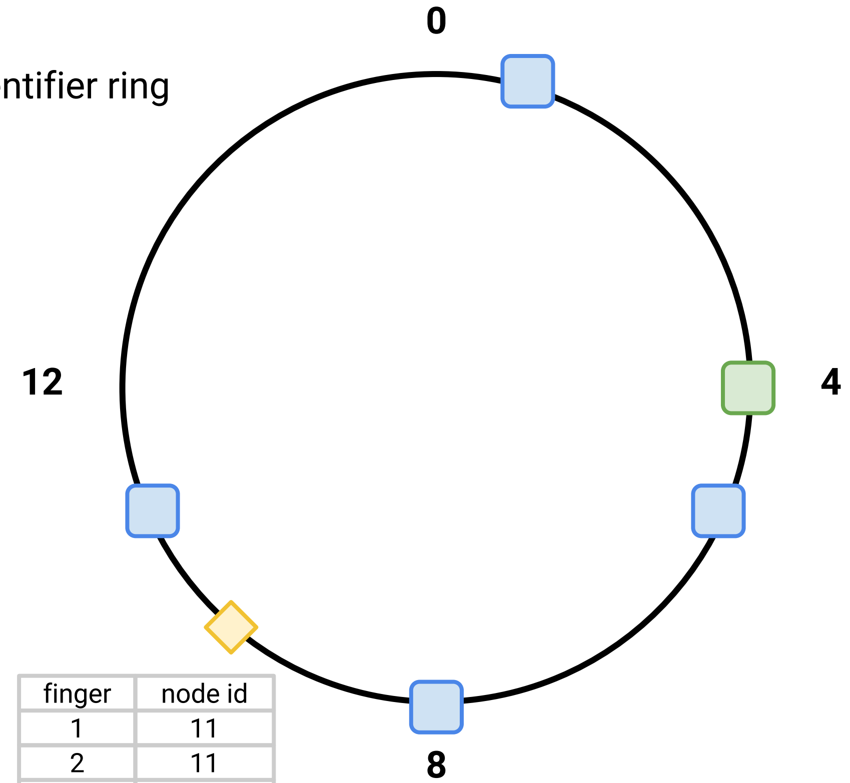
Chord - Overview



finger	node id
1	5
2	8
3	8
4	1

Chord - Lookup

Identifier ring



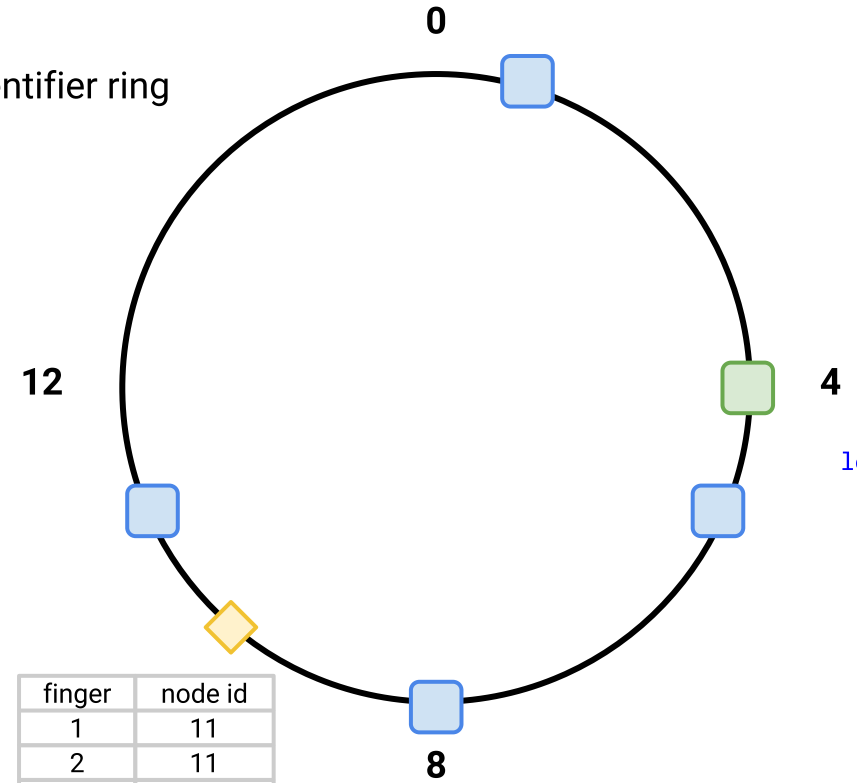
finger	node id
1	5
2	8
3	8
4	1

finger	node id
1	11
2	11
3	1
4	1

```
find_successor(id):
  p = find_predecessor(id)
  return p.successor

find_predecessor(id):
  n = self
  while id not between (n,
n.successor]:
    n = n.closest_preceding_finger(id)
  return n
```

Chord - Lookup

Identifier ring




```
find_successor(id):
  p = find_predecessor(id)
  return p.successor

find_predecessor(id):
  n = self
  while id not between (n,
n.successor]:
    n = n.closest_preceding_finger(id)
  return n
```

finger	node id
1	5
2	8
3	8
4	1

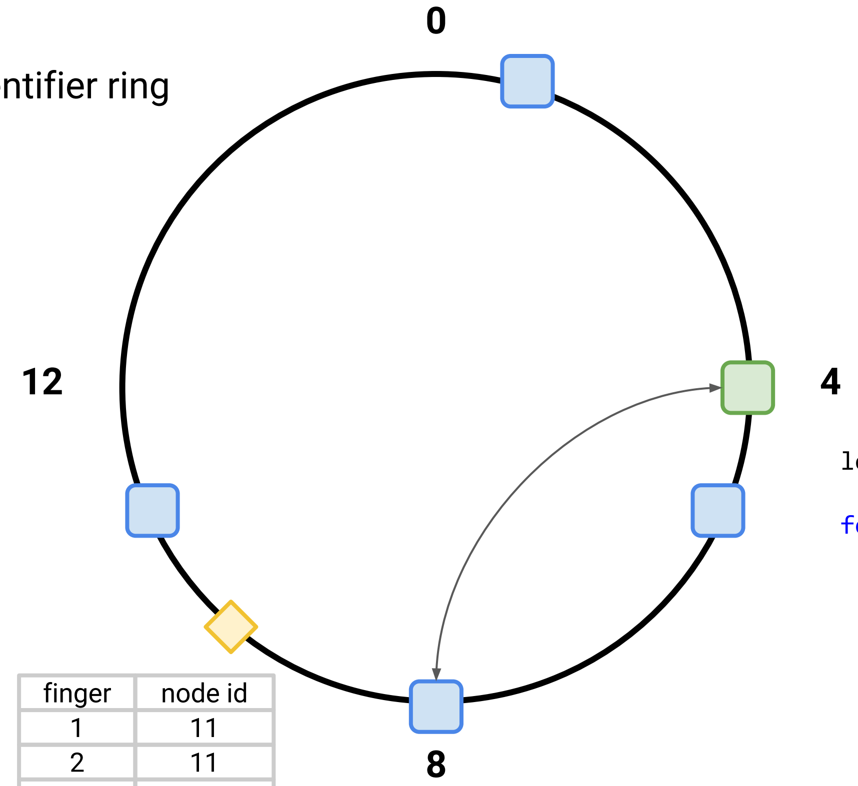
`lookup(10)`

finger	node id
1	11
2	11
3	1
4	1

Chord - Lookup

```
find_successor(id):
  p = find_predecessor(id)
  return p.successor

find_predecessor(id):
  n = self
  while id not between (n,
n.successor]:
    n = n.closest_preceding_finger(id)
  return n
```

Identifier ring



finger	node id
1	5
2	8
3	8
4	1

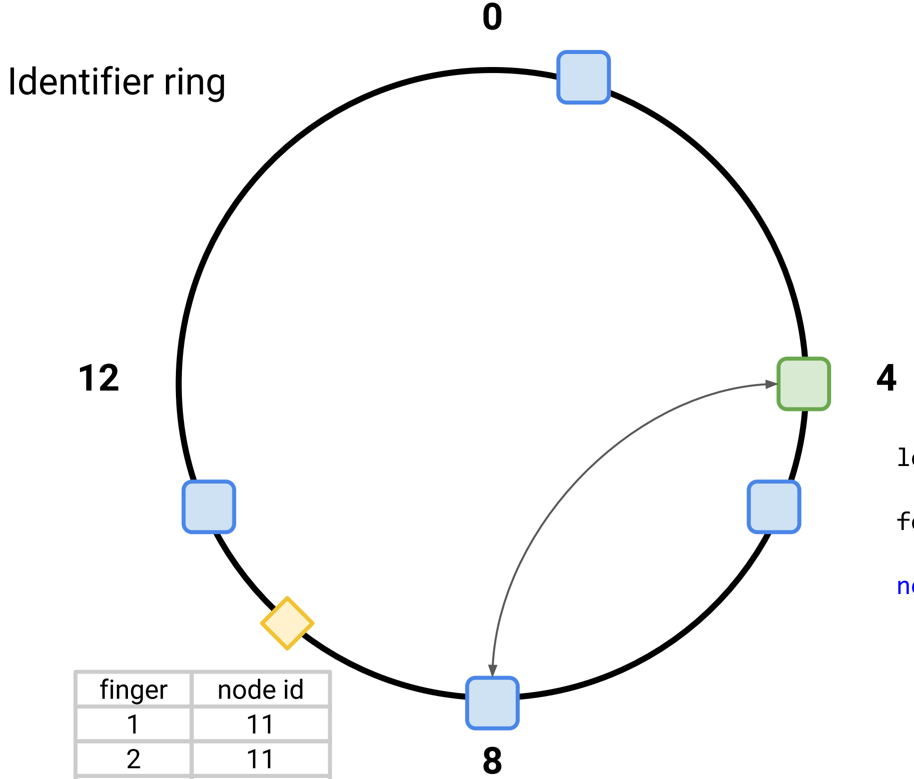
`lookup(10)`

`follow finger 3 to node id 8`

finger	node id
1	11
2	11
3	1
4	1

Chord - Lookup

```
find_successor(id):
  p = find_predecessor(id)
  return p.successor

find_predecessor(id):
  n = self
  while id not between (n,
n.successor]:
    n = n.closest_preceding_finger(id)
  return n
```

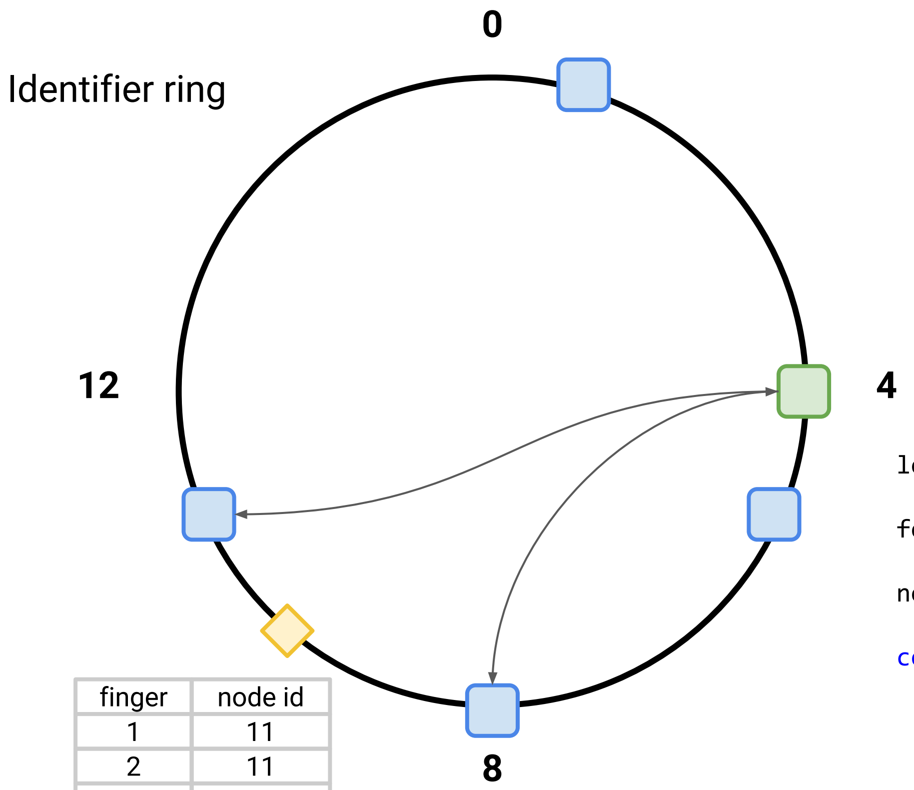


finger	node id
1	5
2	8
3	8
4	1

```
lookup(10)

follow finger 3 to node id 8

node id 8 identifies as predecessor of id 10
```

finger	node id
1	11
2	11
3	1
4	1

Chord - Lookup



finger	node id
1	11
2	11
3	1
4	1

```
find_successor(id):
  p = find_predecessor(id)
  return p.successor

find_predecessor(id):
  n = self
  while id not between (n,
n.successor]:
    n = n.closest_preceding_finger(id)
  return n
```

finger	node id
1	5
2	8
3	8
4	1

`lookup(10)`

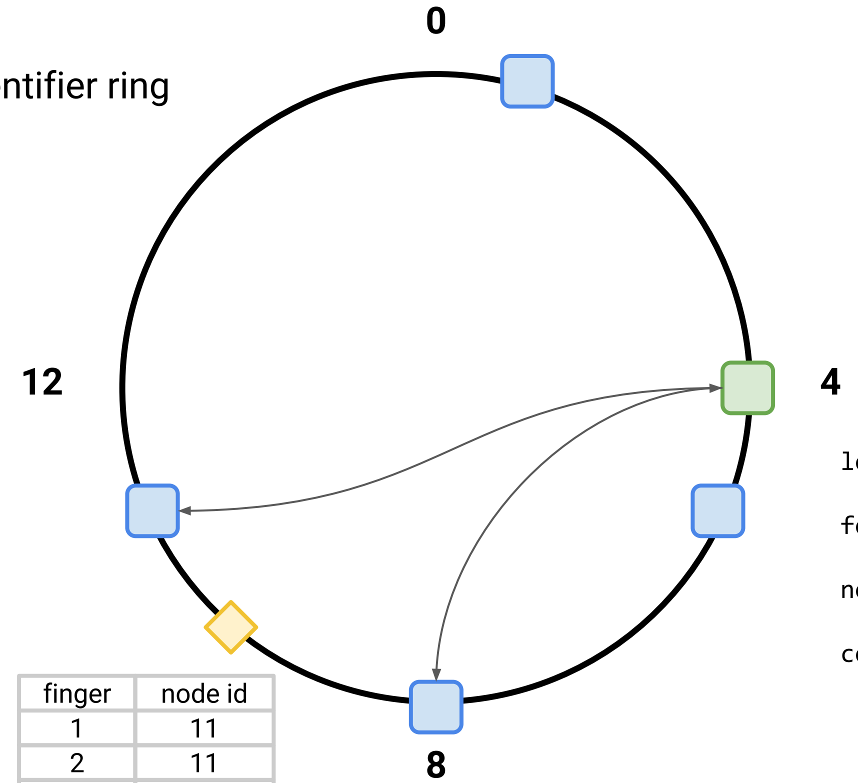
`follow finger 3 to node id 8`

`node id 8 identifies as predecessor of id 10`

`complete lookup at successor of node id 8`

Chord - Lookup

Identifier ring



finger	node id
1	11
2	11
3	1
4	1

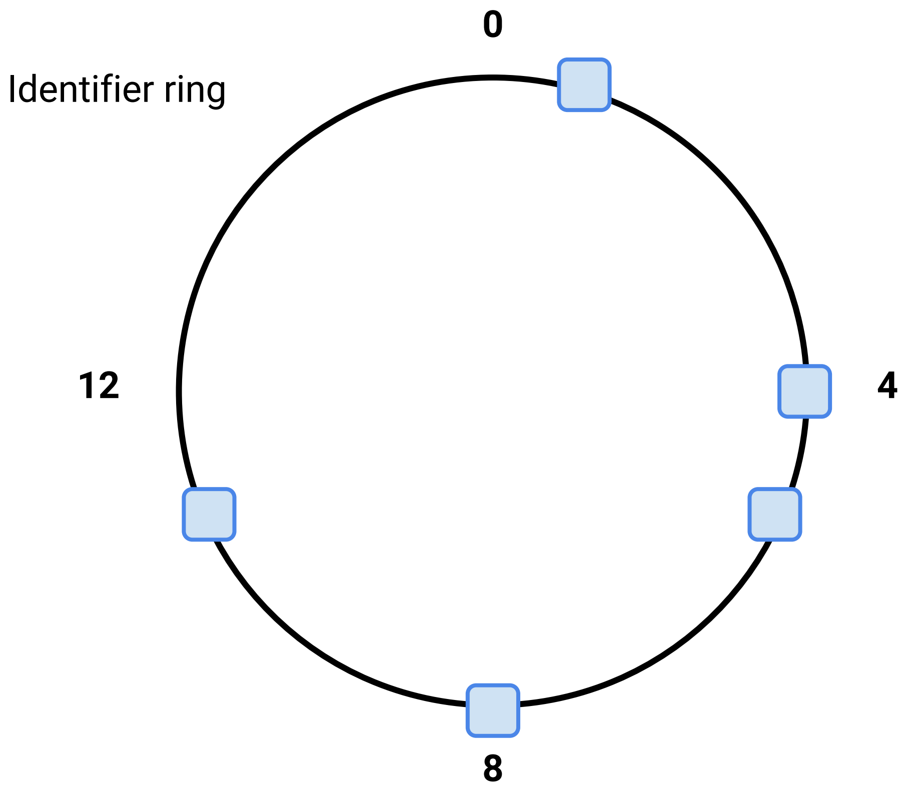
```
find_successor(id):
  p = find_predecessor(id)
  return p.successor

find_predecessor(id):
  n = self
  while id not between (n,
n.successor]:
    n = n.closest_preceding_finger(id)
  return n
```

finger	node id
1	5
2	8
3	8
4	1

Hops?
Each finger lookup halves distance to key
O(log N)

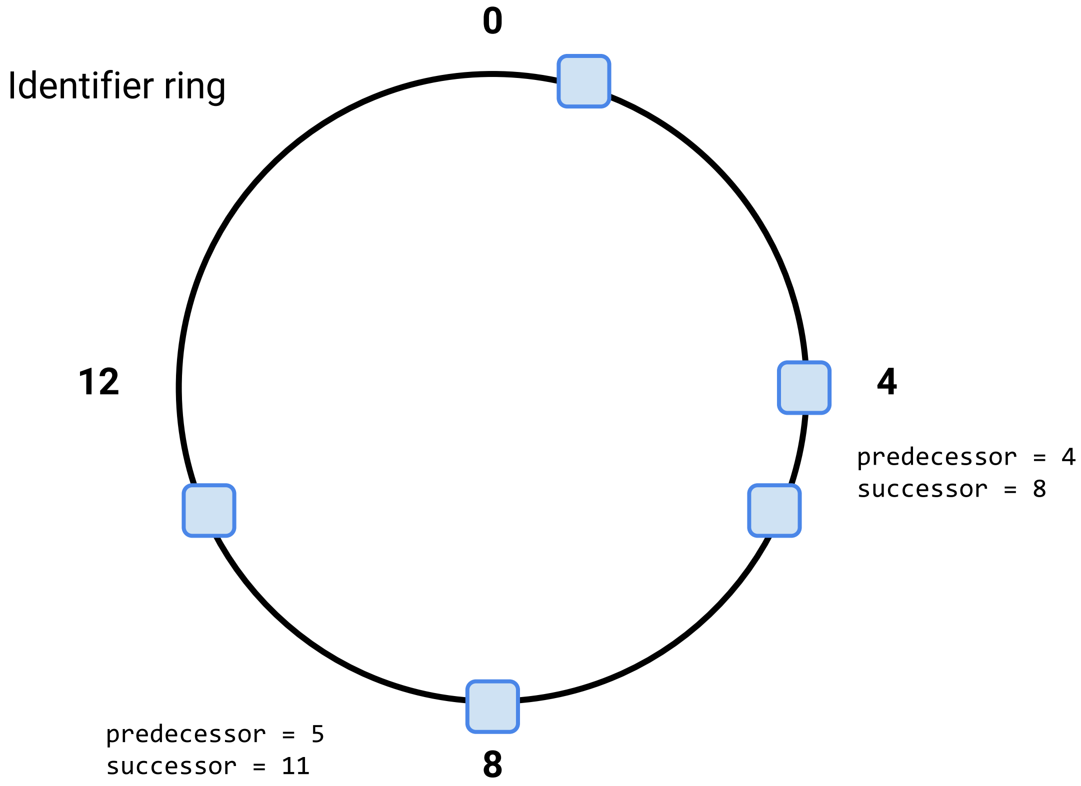
```
lookup(10)

follow finger 3 to node id 8

node id 8 identifies as predecessor of id 10

complete lookup at successor of node id 8
```

Chord - Joins + Stabilization



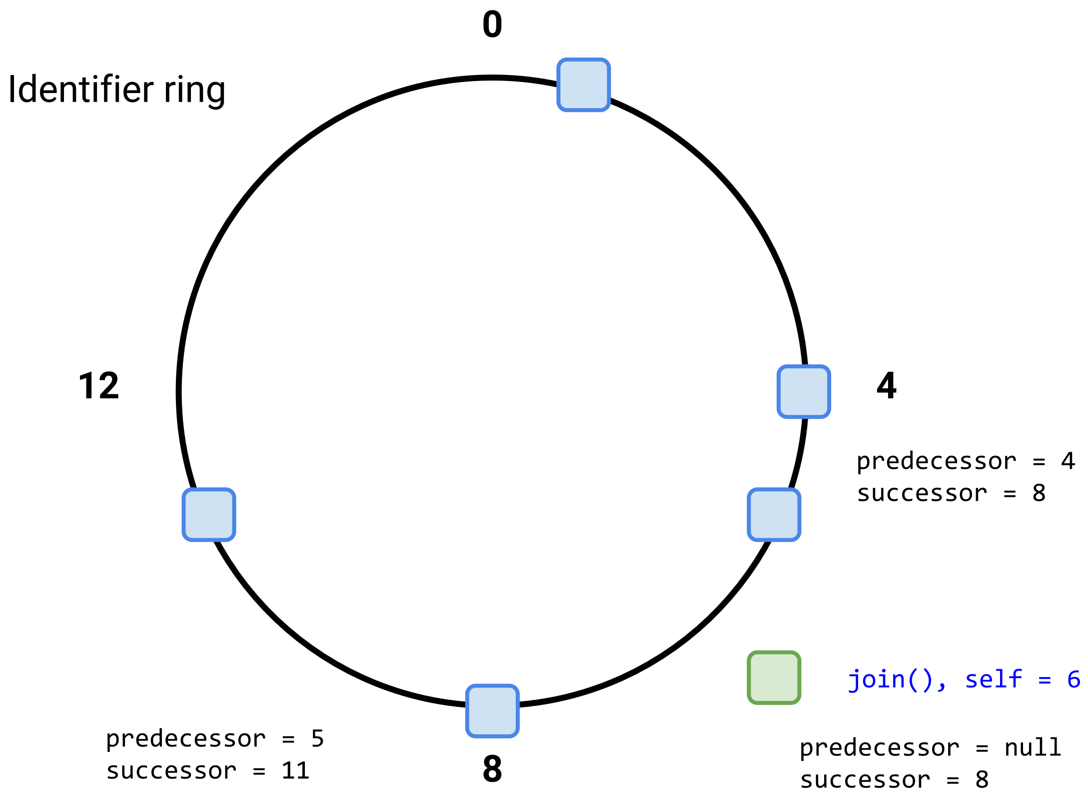

```
join():
  self.predecessor = null
  self.successor = find_successor(self)

stabilize():
  p = self.successor.predecessor
  if p between (self, self.successor):
    self.successor = p
  self.successor.notify(self)

notify(n):
  if self.predecessor == null ||
     n between (self.predecessor, self):
    self.predecessor = n
```

Chord - Joins + Stabilization



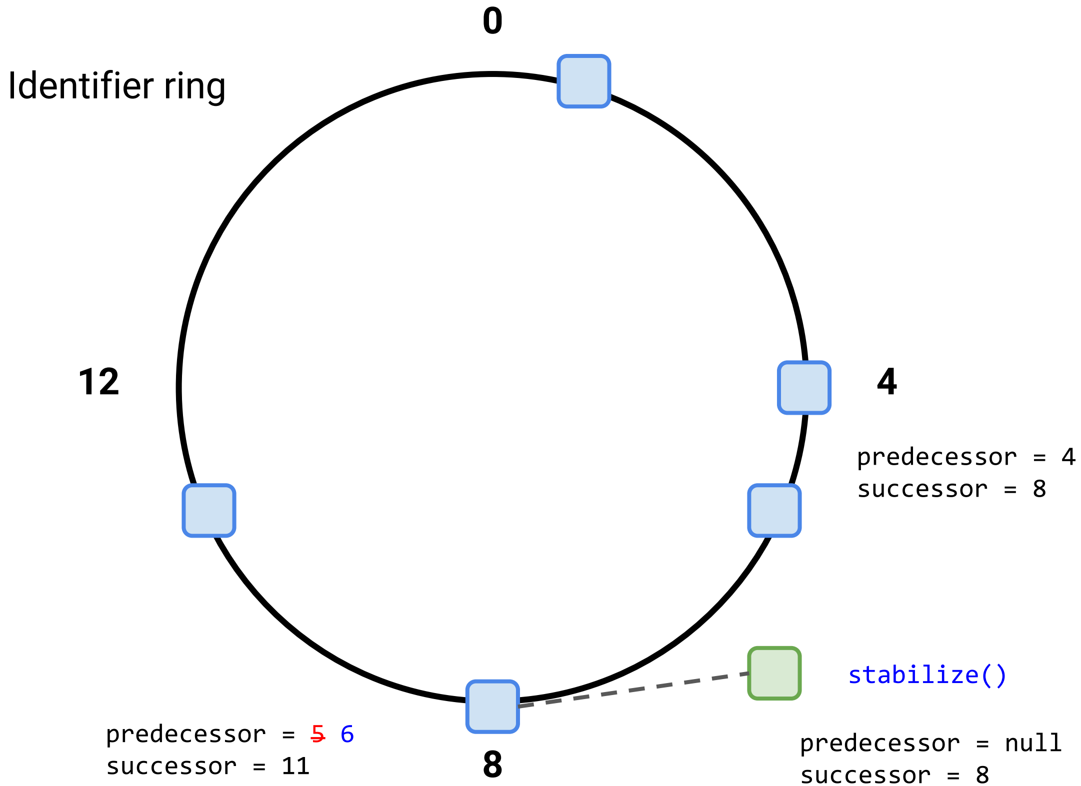

```
join():
  self.predecessor = null
  self.successor = find_successor(self)

stabilize():
  p = self.successor.predecessor
  if p between (self, self.successor):
    self.successor = p
  self.successor.notify(self)

notify(n):
  if self.predecessor == null ||
     n between (self.predecessor, self):
    self.predecessor = n
```

Chord - Joins + Stabilization



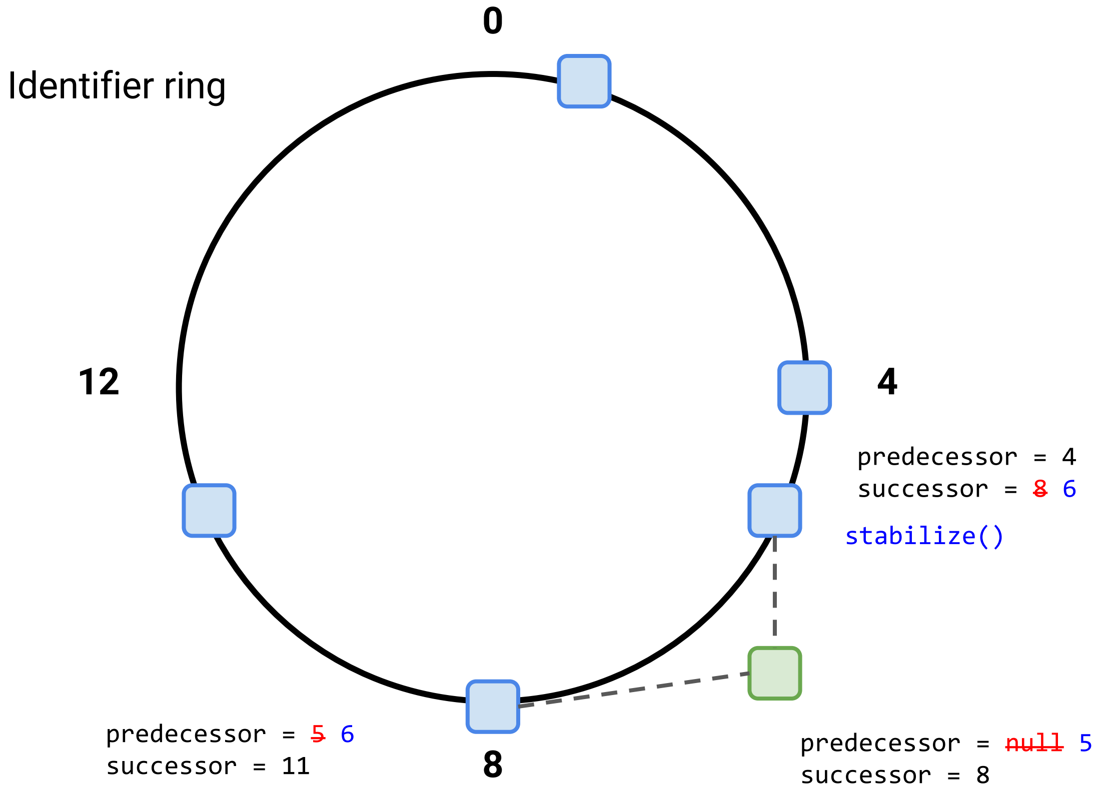

```
join():
  self.predecessor = null
  self.successor = find_successor(self)

stabilize():
  p = self.successor.predecessor
  if p between (self, self.successor):
    self.successor = p
  self.successor.notify(self)

notify(n):
  if self.predecessor == null ||
     n between (self.predecessor, self):
    self.predecessor = n
```

Chord - Joins + Stabilization



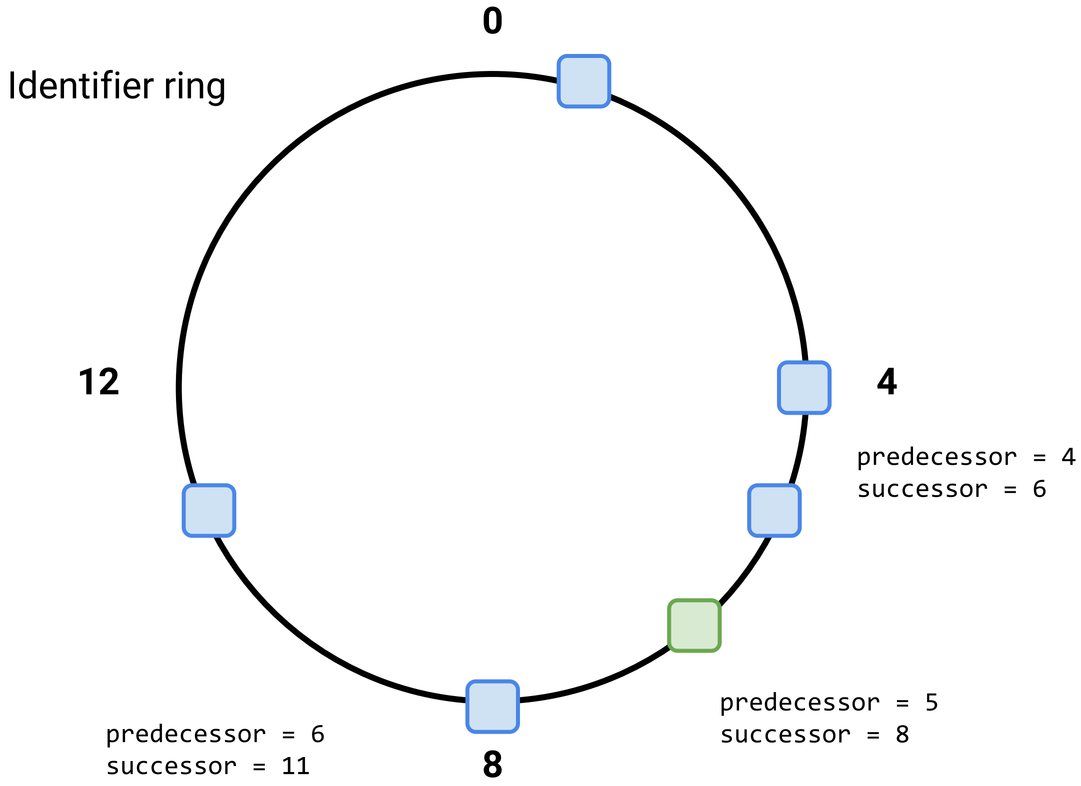

```
join():
  self.predecessor = null
  self.successor = find_successor(self)

stabilize():
  p = self.successor.predecessor
  if p between (self, self.successor):
    self.successor = p
  self.successor.notify(self)

notify(n):
  if self.predecessor == null ||
     n between (self.predecessor, self):
    self.predecessor = n
```

Chord - Joins + Stabilization




```
join():
  self.predecessor = null
  self.successor = find_successor(self)

stabilize():
  p = self.successor.predecessor
  if p between (self, self.successor):
    self.successor = p
  self.successor.notify(self)

notify(n):
  if self.predecessor == null ||
     n between (self.predecessor, self):
    self.predecessor = n
```

Chord - Joins + Stabilization

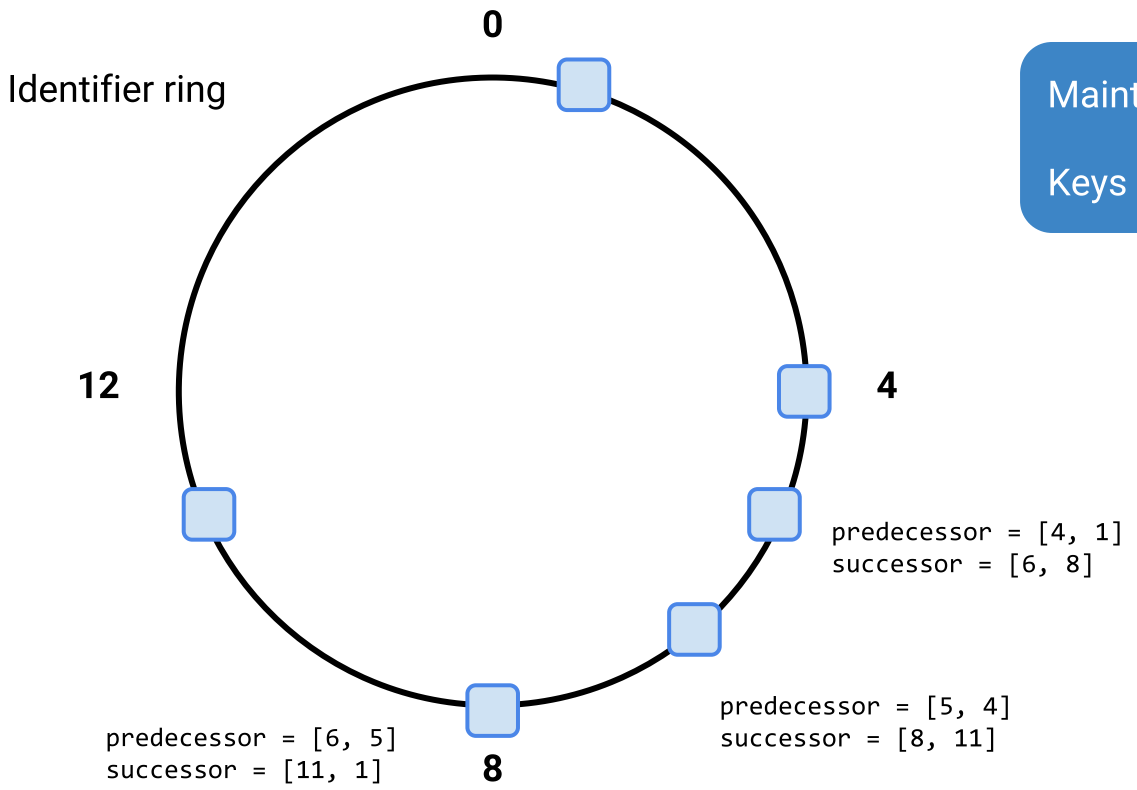



```
join():
  self.predecessor = null
  self.successor = find_successor(self)

stabilize():
  p = self.successor.predecessor
  if p between (self, self.successor):
    self.successor = p
  self.successor.notify(self)

notify(n):
  if self.predecessor == null ||
     n between (self.predecessor, self):
    self.predecessor = n
```

Outcomes of incomplete stabilization:

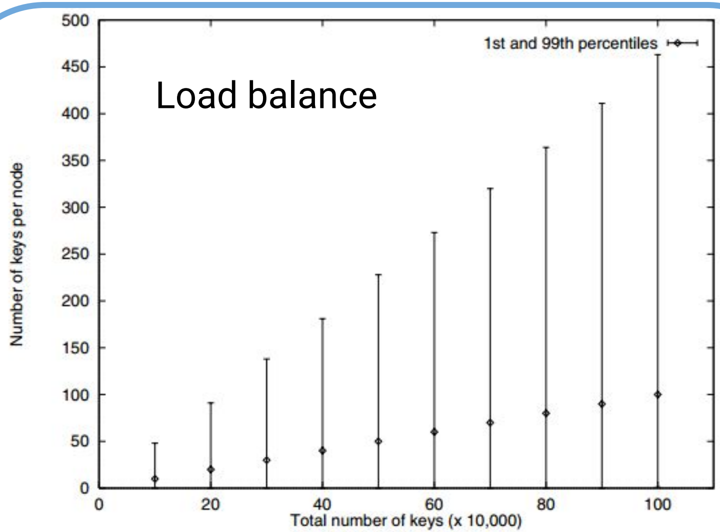
1. Lookup unaffected
2. Fingers out-dated, successors correct -> lookup slow but correct
3. Successors in lookup region still stabilizing -> lookup fails

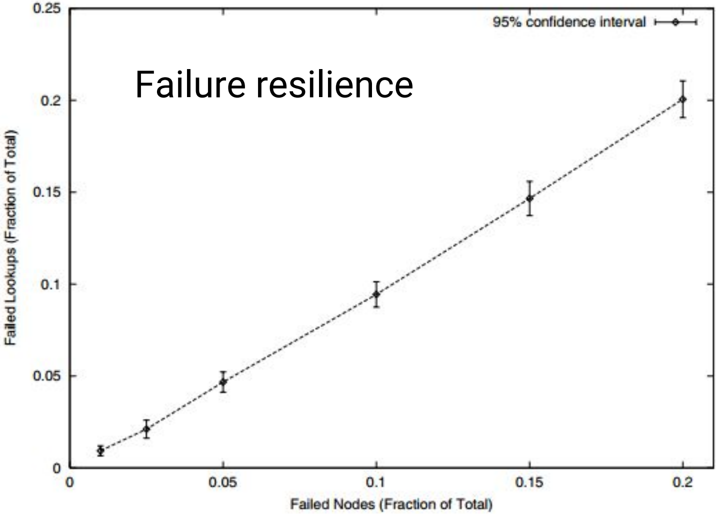
Chord - Failure + Replication

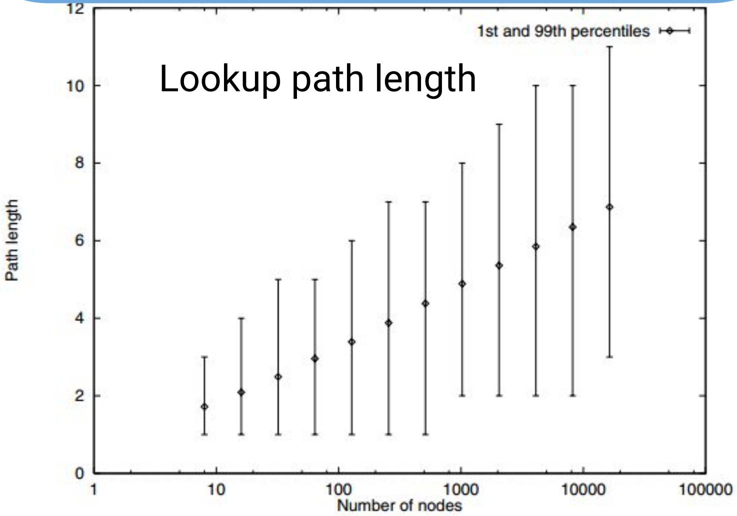


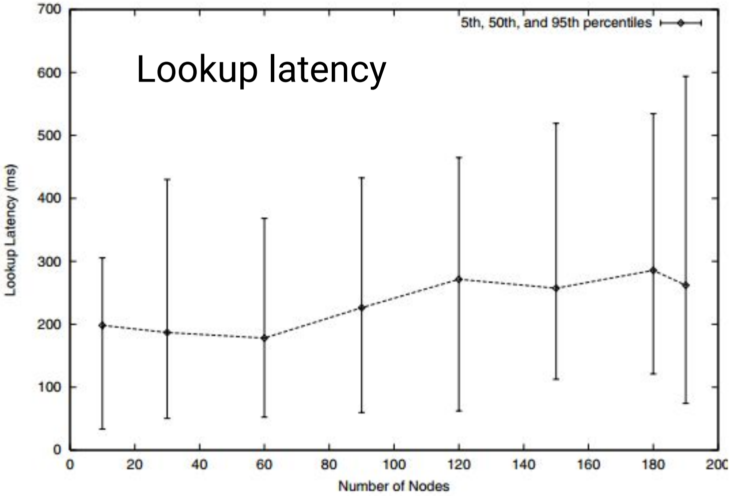
Maintain list of k successors

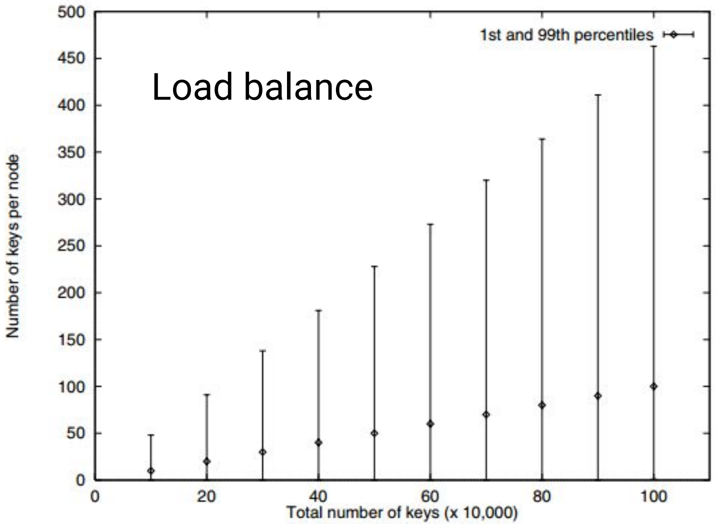
Keys replicated on all k successors

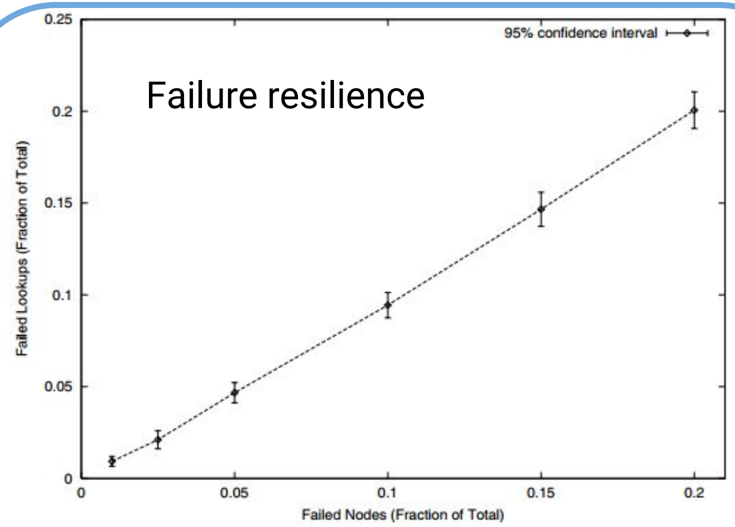

Load balance
1st and 99th percentiles
Number of keys per node
500
450
400
350
300
250
200
150
100
50
0
0
20
40
60
80
100
Total number of keys (x 10,000)

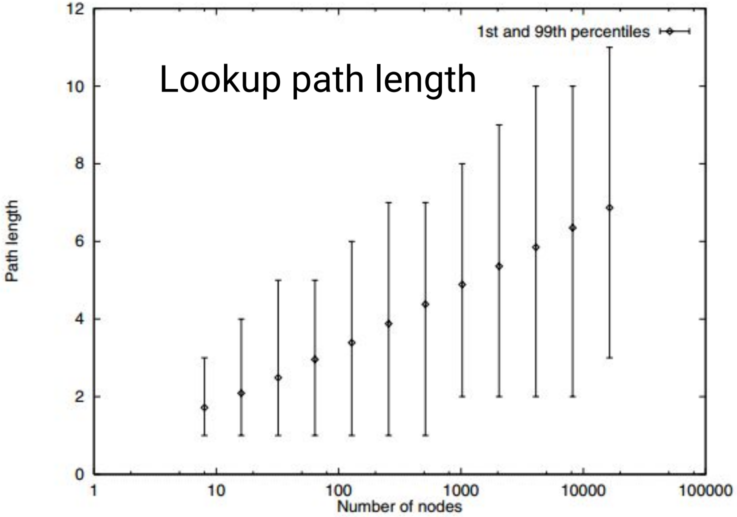

Failure resilience
95% confidence interval
Failed Lookups (Fraction of Total)
0.25
0.2
0.15
0.1
0.05
0
0
0.05
0.1
0.15
0.2
Failed Nodes (Fraction of Total)

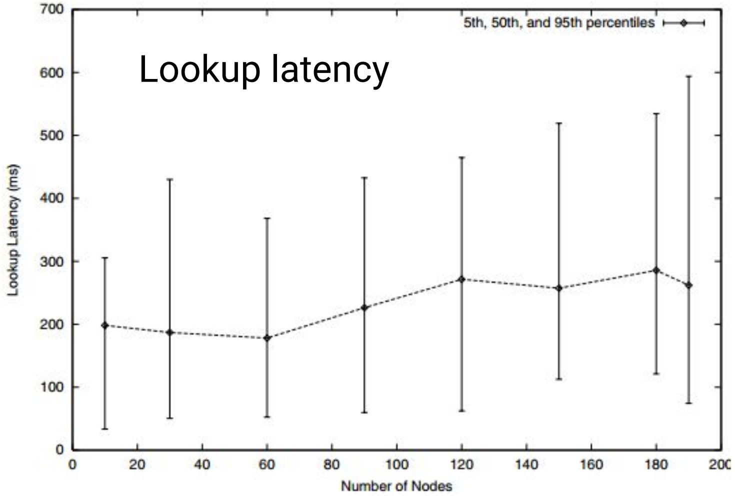

Lookup path length
1st and 99th percentiles
Path length
12
10
8
6
4
2
0
1
10
100
1000
10000
100000
Number of nodes

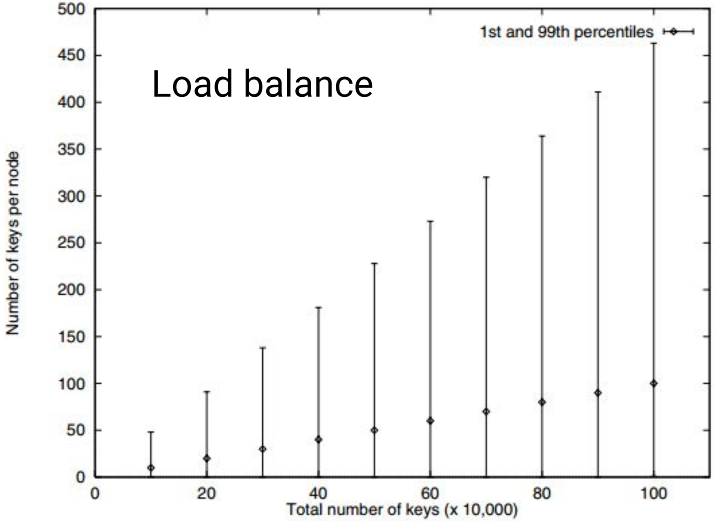

Lookup latency
5th, 50th, and 95th percentiles
Lookup Latency (ms)
700
600
500
400
300
200
100
0
0
20
40
60
80
100
120
140
160
180
200
Number of Nodes

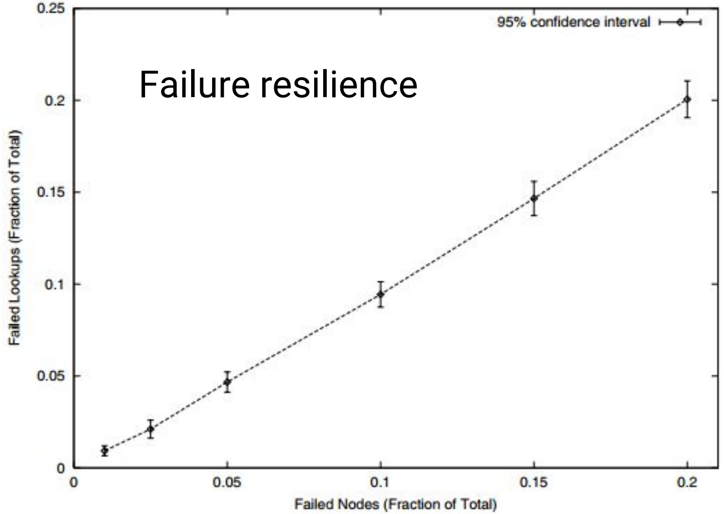

Load balance
1st and 99th percentiles
Number of keys per node
Total number of keys (x 10,000)

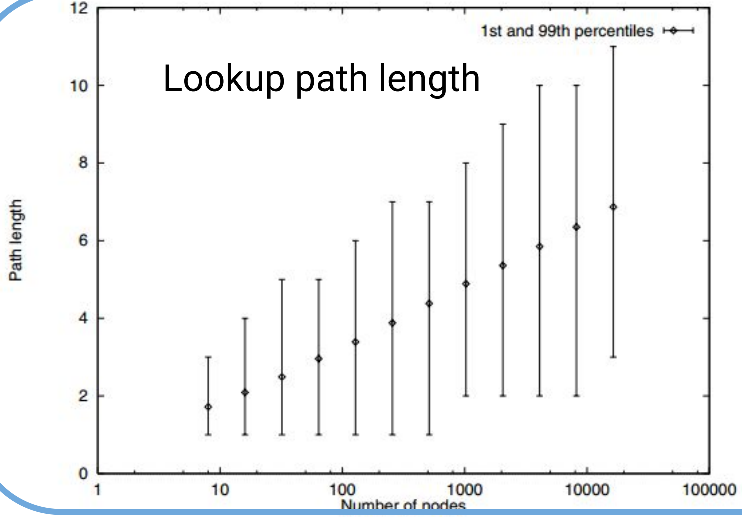

Failure resilience
95% confidence interval
Failed Lookups (Fraction of Total)
Failed Nodes (Fraction of Total)

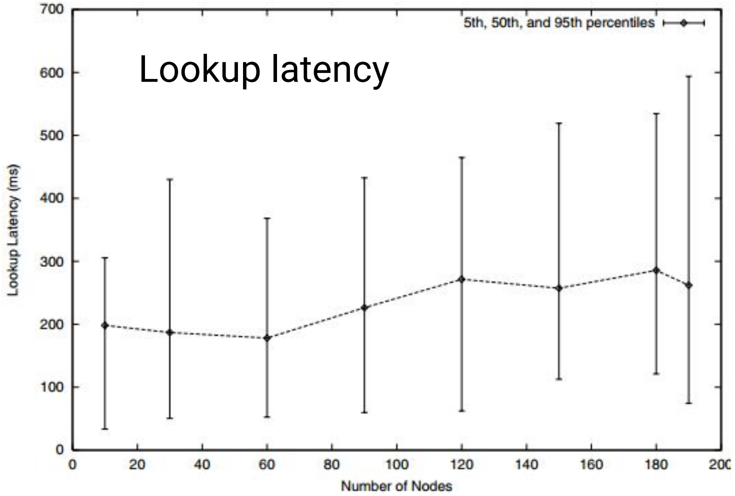

Lookup path length
1st and 99th percentiles
Path length
Number of nodes

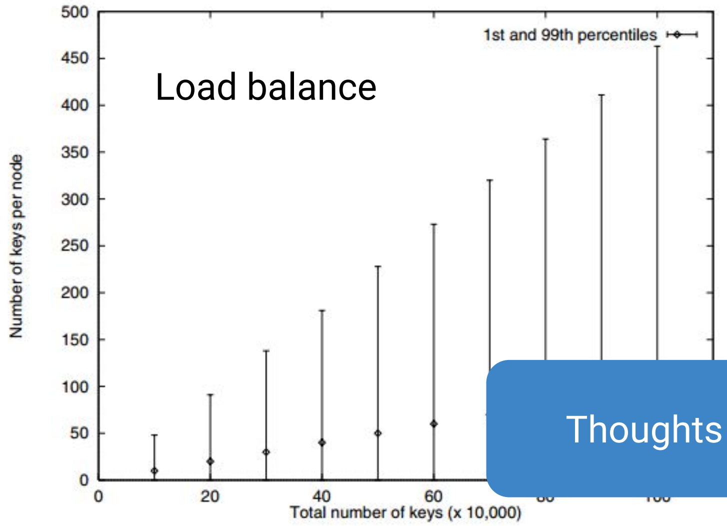

Lookup latency
5th, 50th, and 95th percentiles
Lookup Latency (ms)
Number of Nodes

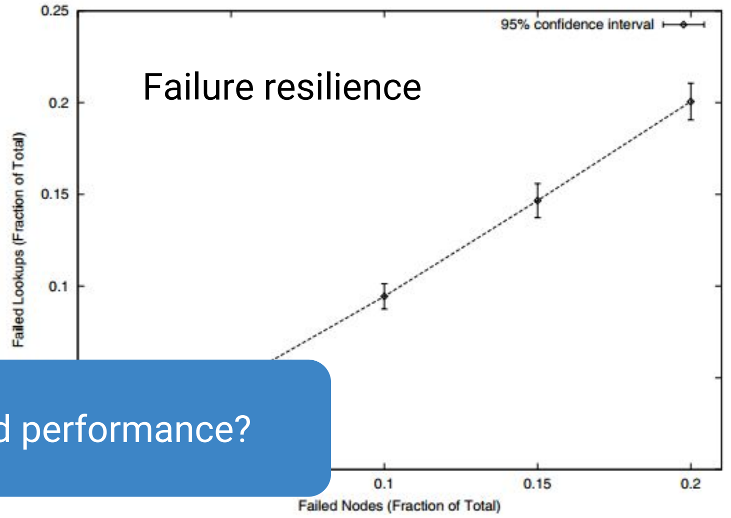

Load balance
1st and 99th percentiles
Number of keys per node
Total number of keys (x 10,000)


Failure resilience
95% confidence interval
Failed Lookups (Fraction of Total)
Failed Nodes (Fraction of Total)

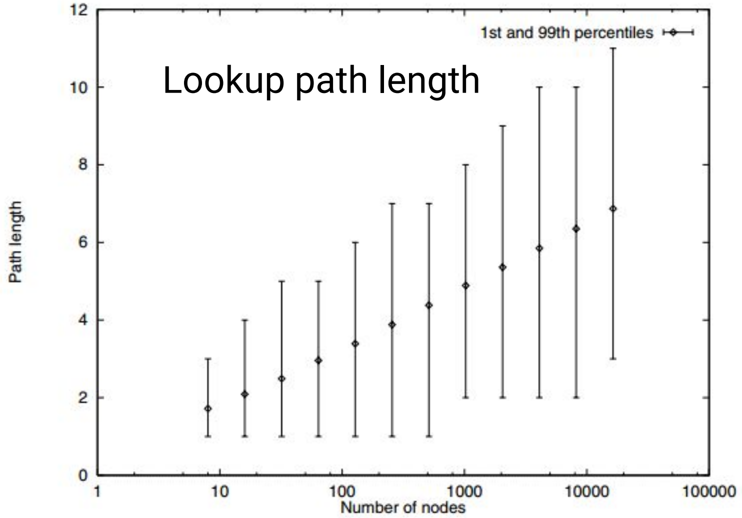

Lookup path length
1st and 99th percentiles
Path length
Number of nodes

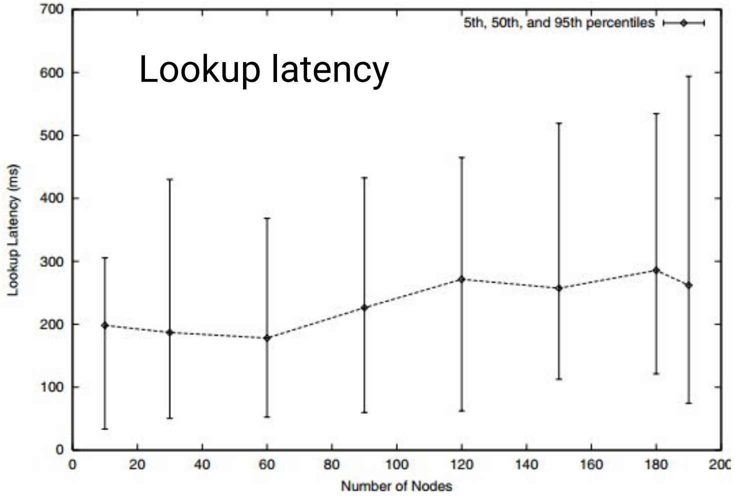

Lookup latency
5th, 50th, and 95th percentiles
Lookup Latency (ms)
Number of Nodes

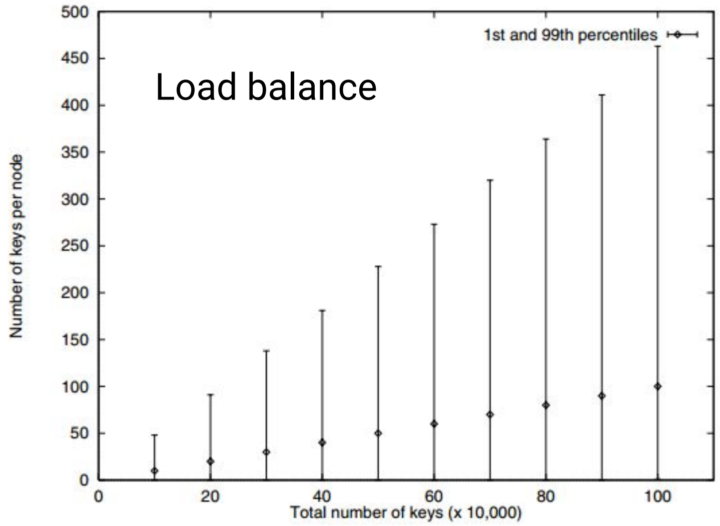


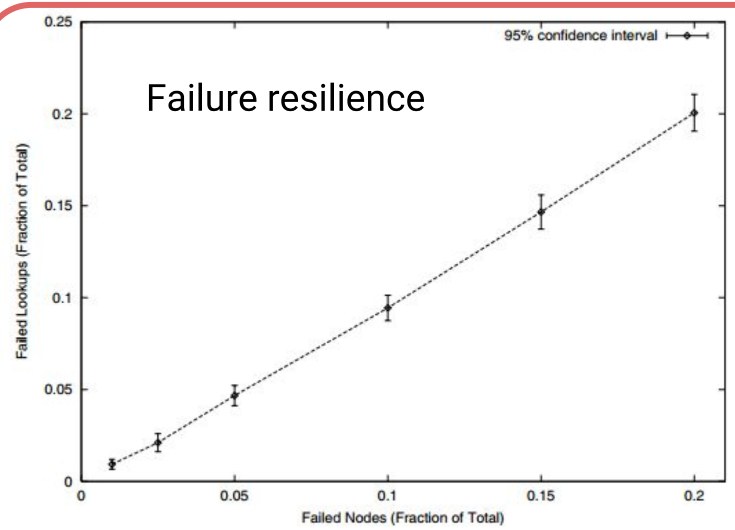


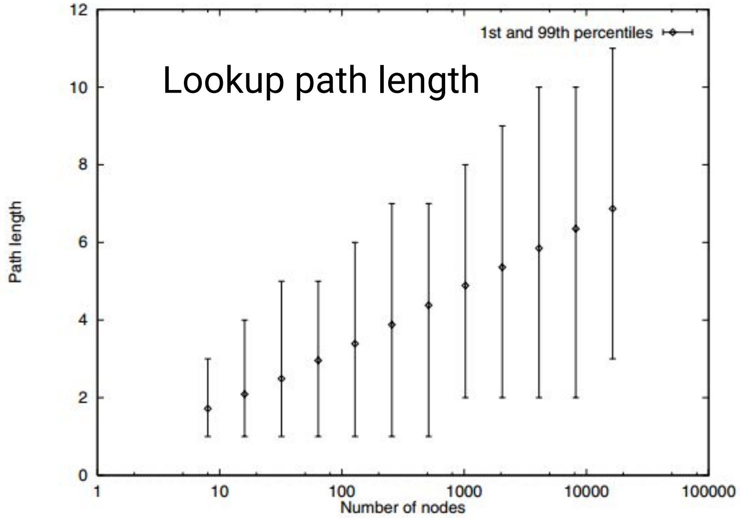
Thoughts on Chord performance?

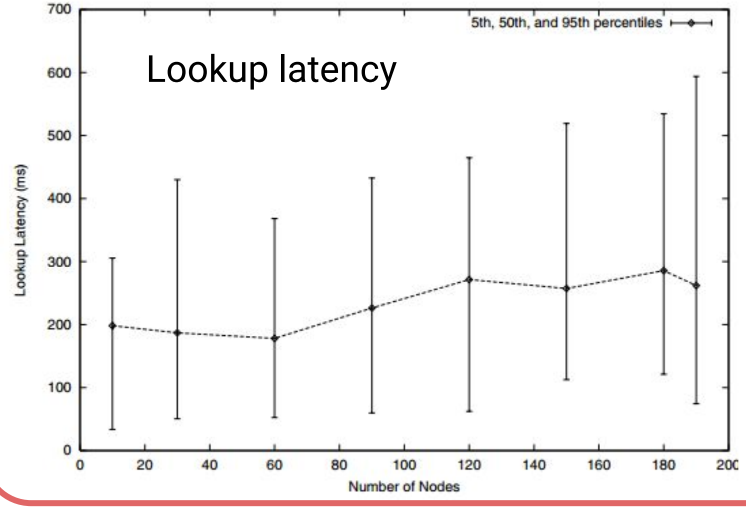




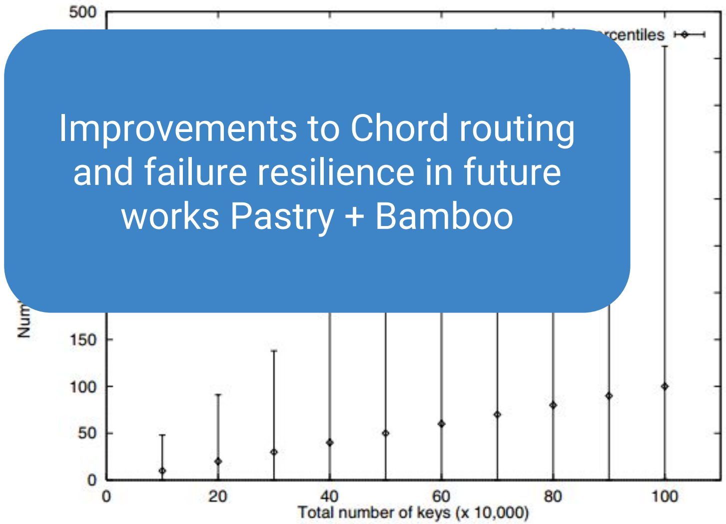

Load balance
1st and 99th percentiles
Number of keys per node
Total number of keys (x 10,000)

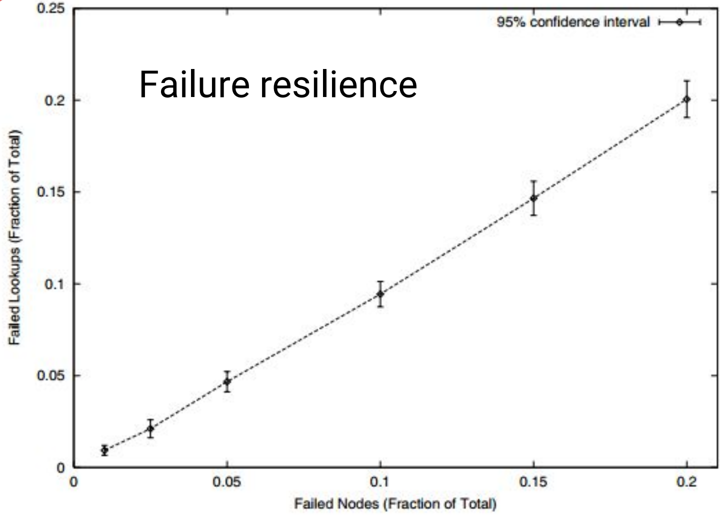

Failure resilience
95% confidence interval
Failed Lookups (Fraction of Total)
Failed Nodes (Fraction of Total)

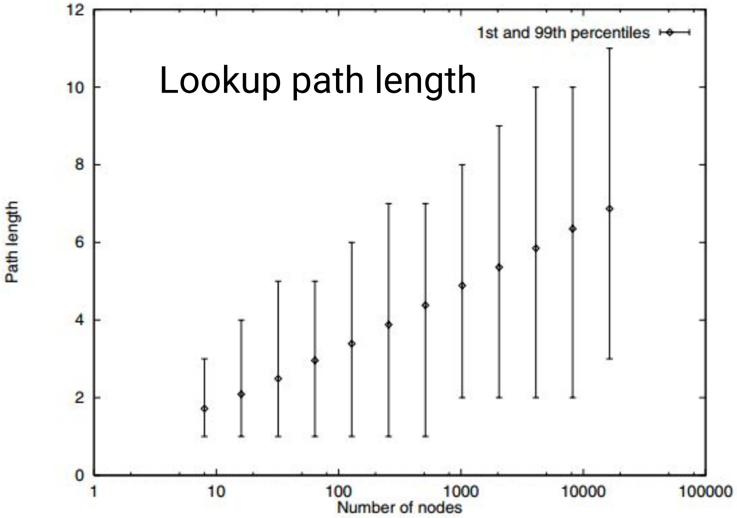

Lookup path length
1st and 99th percentiles
Path length
Number of nodes

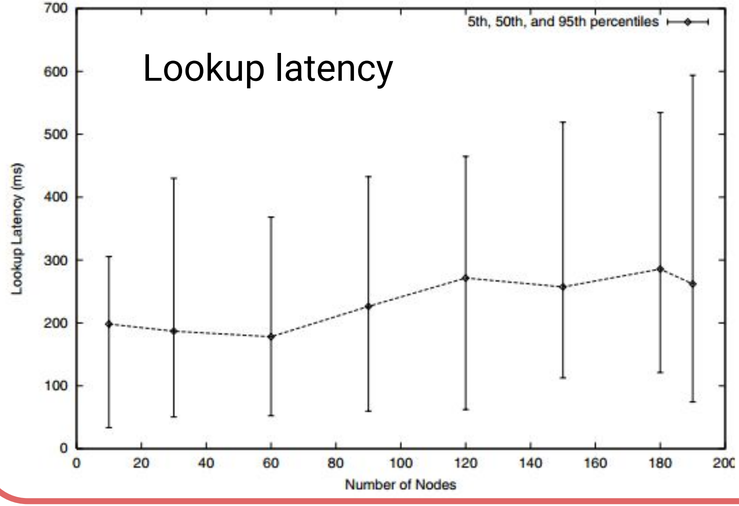

Lookup latency
5th, 50th, and 95th percentiles
Lookup Latency (ms)
Number of Nodes

Improvements to Chord routing and failure resilience in future works Pastry + Bamboo









The Impact of DHT Routing Geometry on Resilience and Proximity

K. Gummadi, R. Gummadi, S. Gribble
S. Ratnasamy, S. Shenker, I. Stoica

DHT Routing Geometries

Ring (Chord)

Tree (Tapestry, PRR)

Hypercube (CAN)

Butterfly (Viceroy)

XOR (Kademlia)

Hybrid (Pastry, Bamboo)

Proximity + Resilience

Proximity - Pick routes through “physically nearby” peers, reducing latency

Resilience - Continue to route requests despite network churn and failure

Proximity + Resilience

Proximity - Pick routes through “physically nearby” peers, reducing latency

Resilience - Continue to route requests despite network churn and failure

Flexibility

Neighbor selection - options in selecting which peers to keep in routing table

Route selection - options in selecting where to route to given a destination

Proximity + Resilience

Proximity - Pick routes through “physically nearby” peers, reducing latency

Resilience - Continue to route requests despite network churn and failure

Flexibility

Neighbor selection - options in selecting which peers to keep in routing table

Route selection - options in selecting where to route to given a destination

Flexibility in neighbor selection -> good proximity

Flexibility in route selection -> good resilience

DHT Routing Geometries

Ring (Chord)

Tree (Tapestry, PRR)

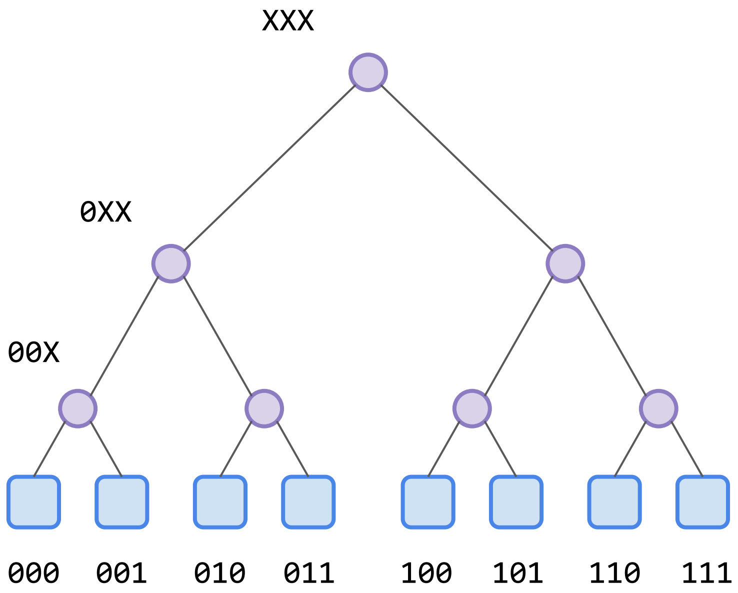
Hypercube (CAN)

Butterfly (Viceroy)

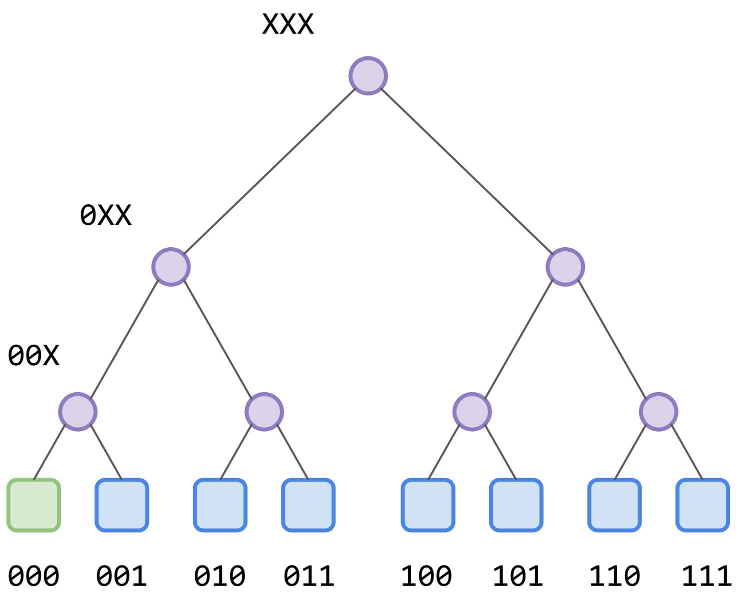
XOR (Kademlia)

Hybrid (Pastry, Bamboo)

DHT Routing Geometries - Tree

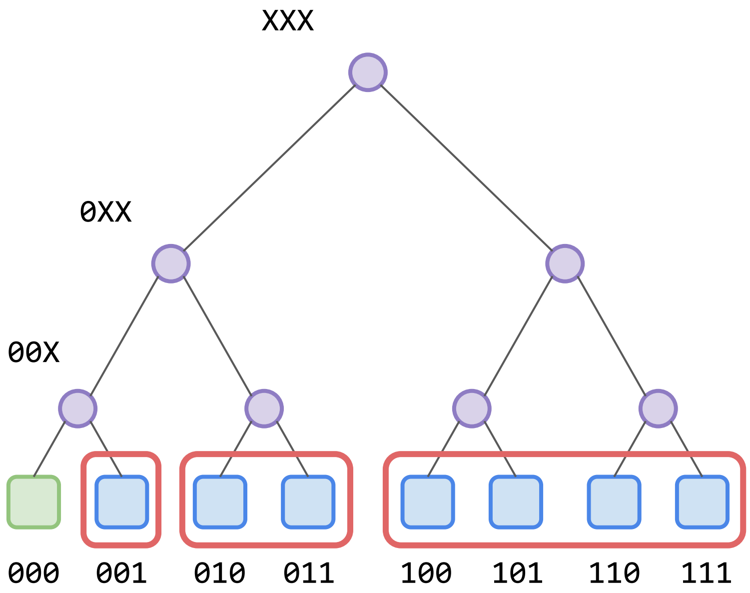


DHT Routing Geometries - Tree



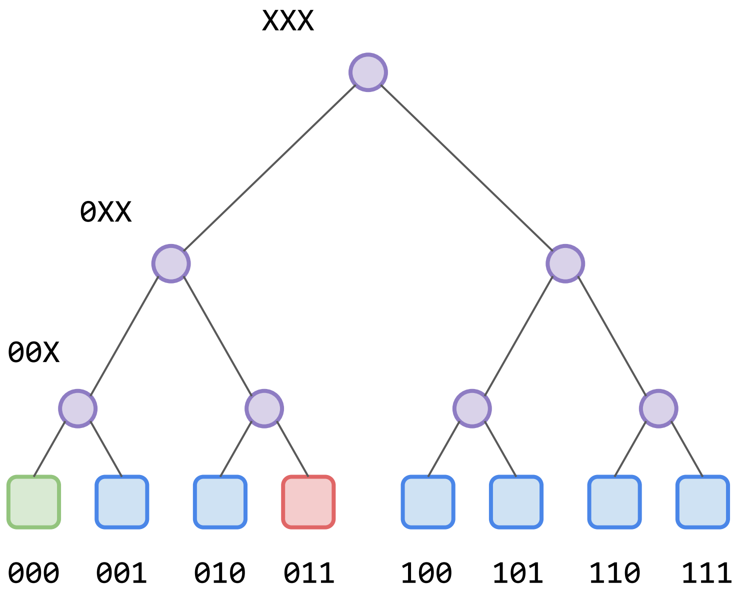
Neighbor selection - one neighbor for each prefix in opposite subtree

DHT Routing Geometries - Tree



Neighbor selection - one neighbor for each prefix in opposite subtree

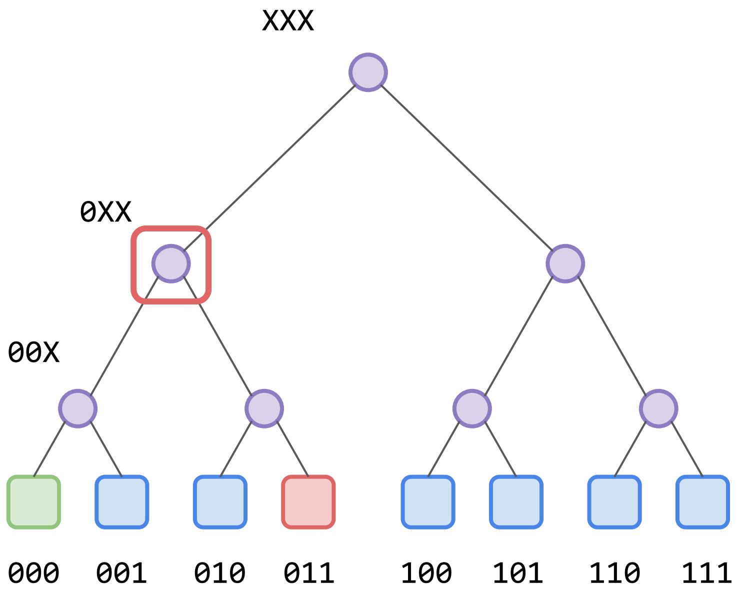
DHT Routing Geometries - Tree



Neighbor selection - one neighbor for each prefix in opposite subtree

Route selection - route to neighbor in subtree of destination

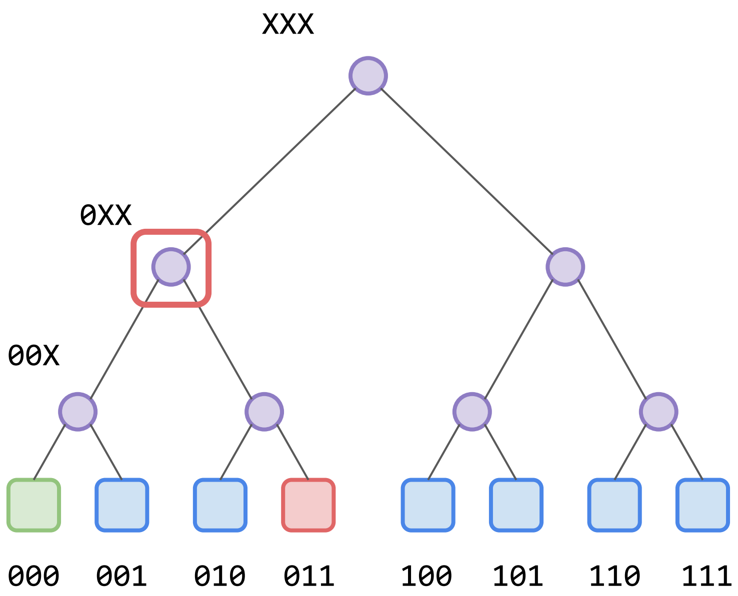
DHT Routing Geometries - Tree



Neighbor selection - one neighbor for each prefix in opposite subtree

Route selection - route to neighbor in subtree of destination

DHT Routing Geometries - Tree



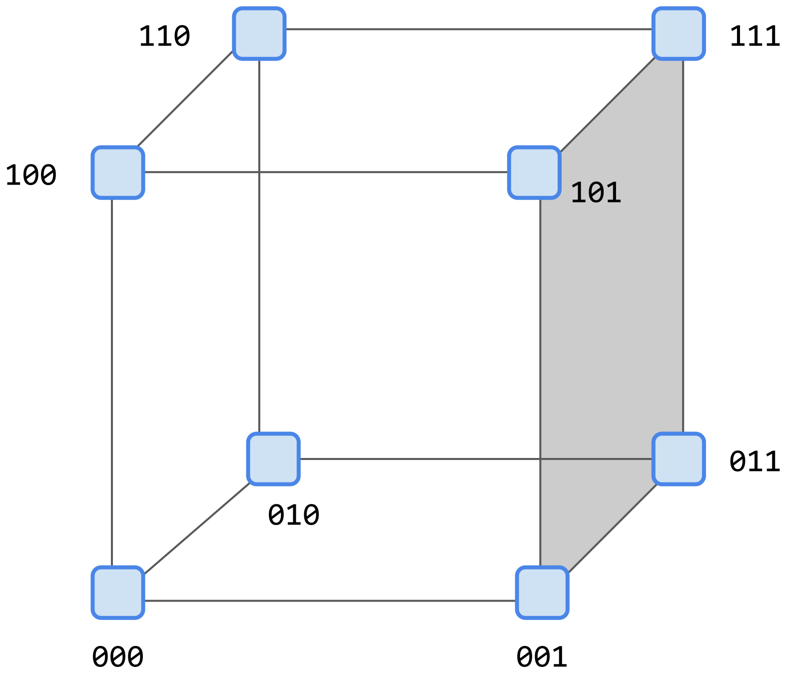
Neighbor selection - one neighbor for each prefix in opposite subtree

Route selection - route to neighbor in subtree of destination

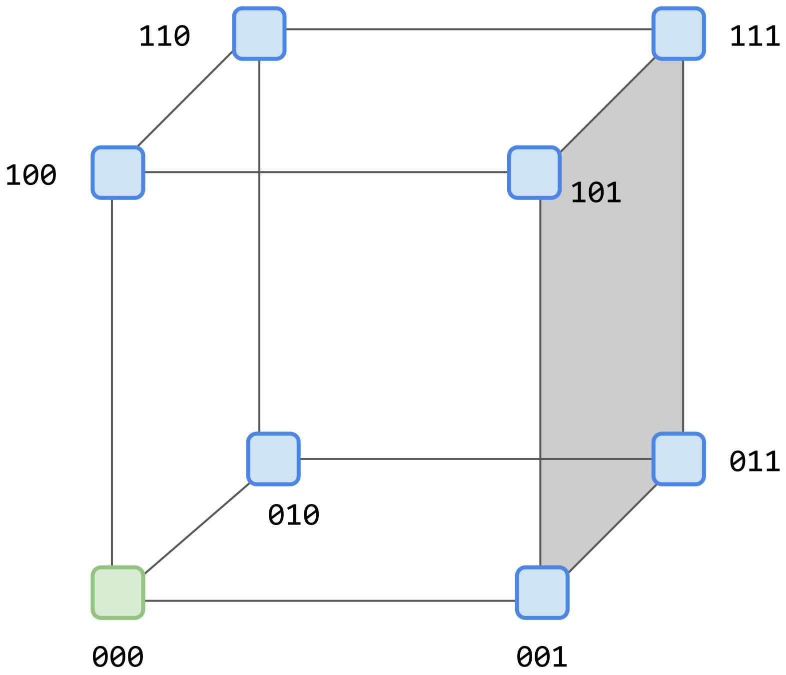
Good flexibility in neighbor selection

Poor flexibility in route selection

DHT Routing Geometries - Hypercube

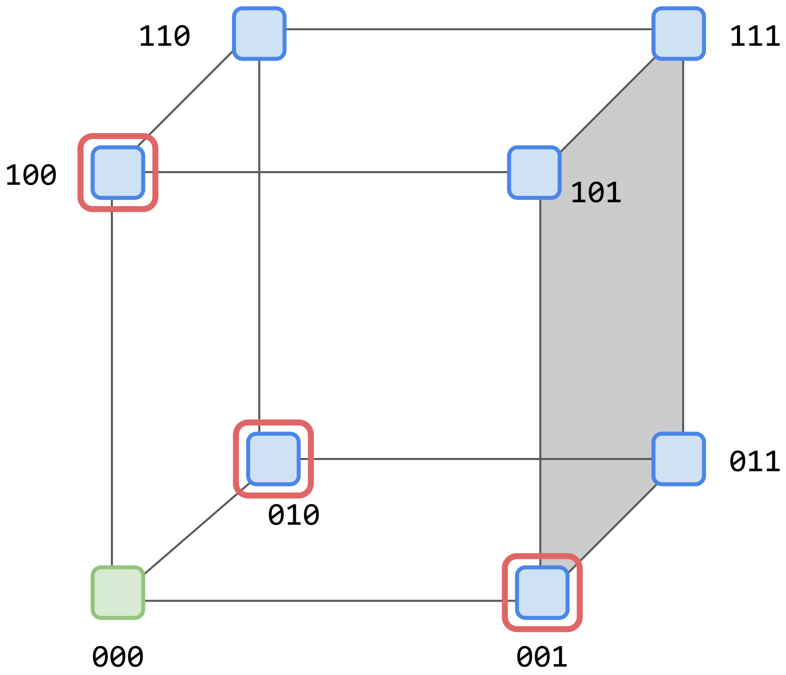


DHT Routing Geometries - Hypercube



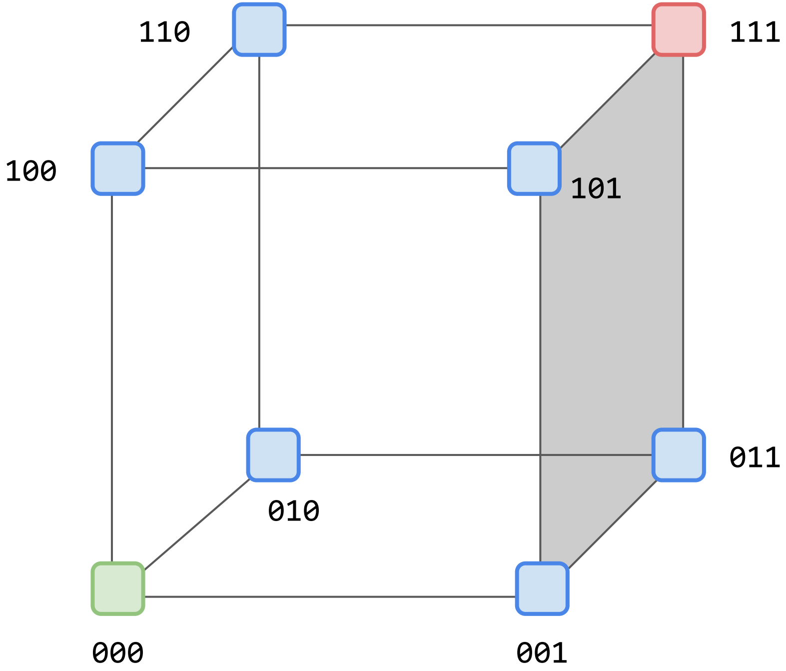
Neighbor selection - neighbor differs in the bit of one dimension

DHT Routing Geometries - Hypercube



Neighbor selection - neighbor differs in the bit of one dimension

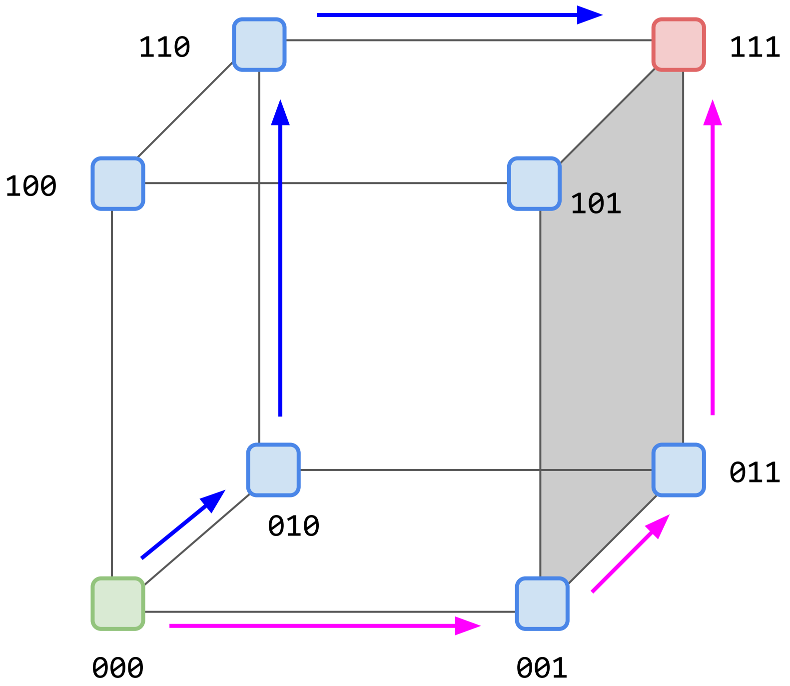
DHT Routing Geometries - Hypercube



Neighbor selection - neighbor differs in the bit of one dimension

Route selection - route to destination by correcting any differing bit

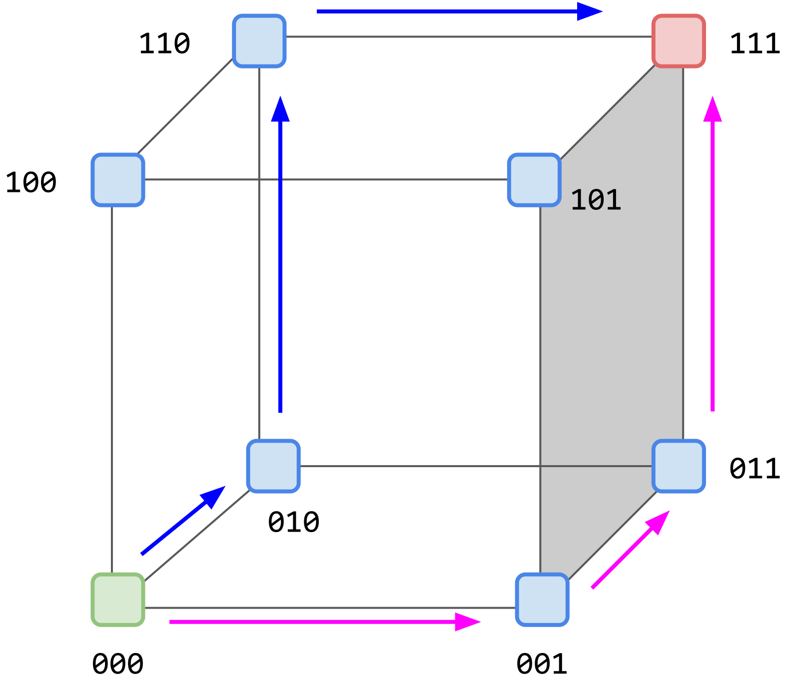
DHT Routing Geometries - Hypercube



Neighbor selection - neighbor differs in the bit of one dimension

Route selection - route to destination by correcting any differing bit

DHT Routing Geometries - Hypercube



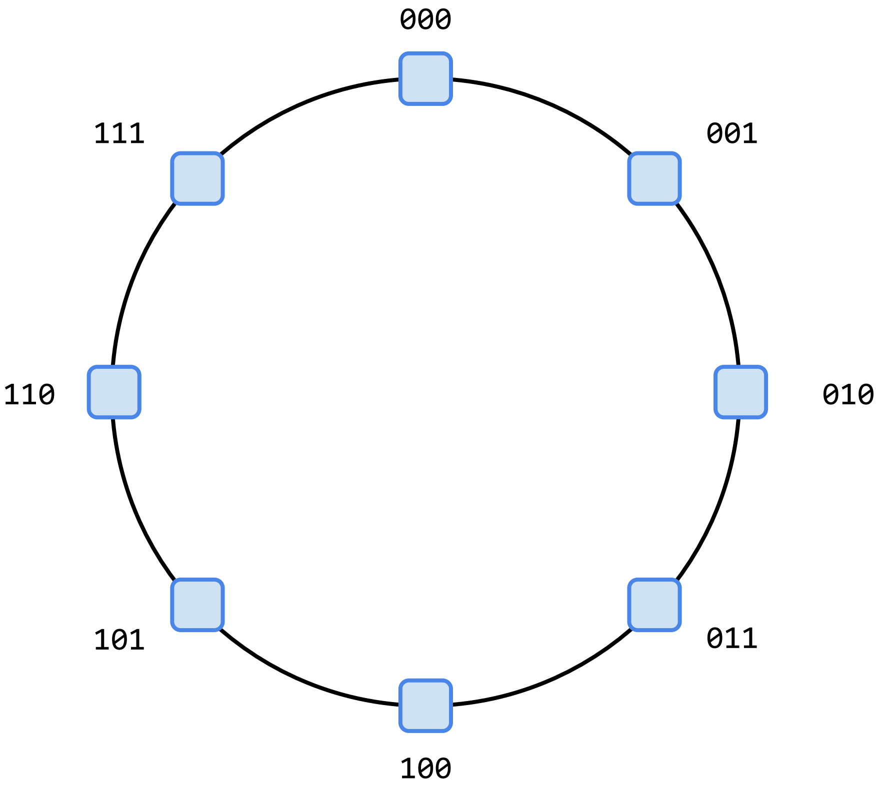
Neighbor selection - neighbor differs in the bit of one dimension

Route selection - route to destination by correcting any differing bit

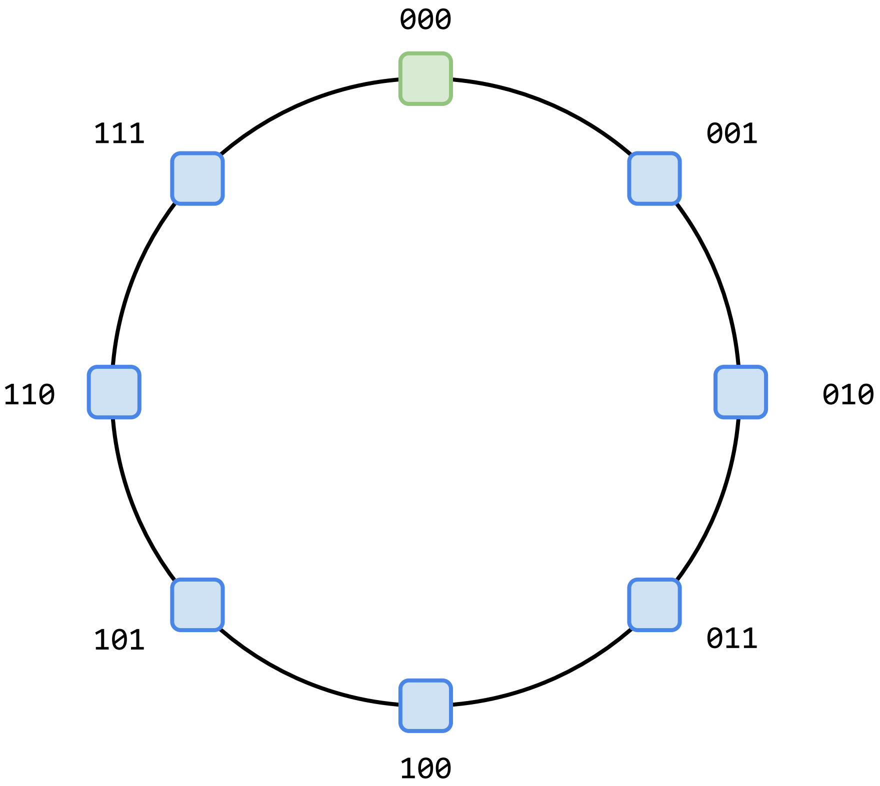
Poor flexibility in neighbor selection

Good flexibility in route selection

DHT Routing Geometries - Ring

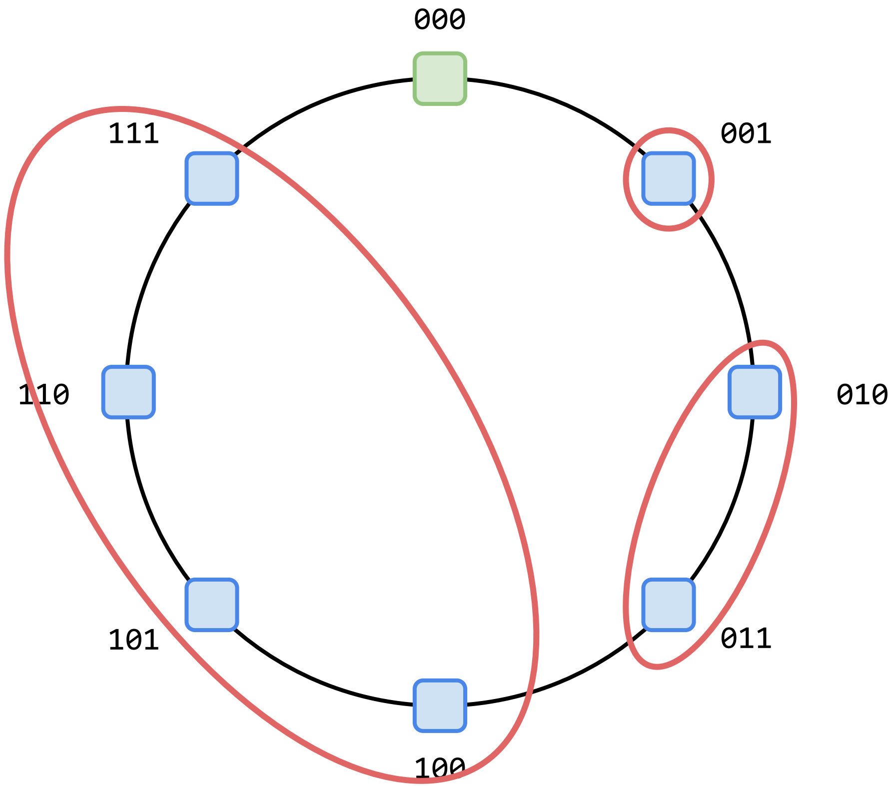


DHT Routing Geometries - Ring



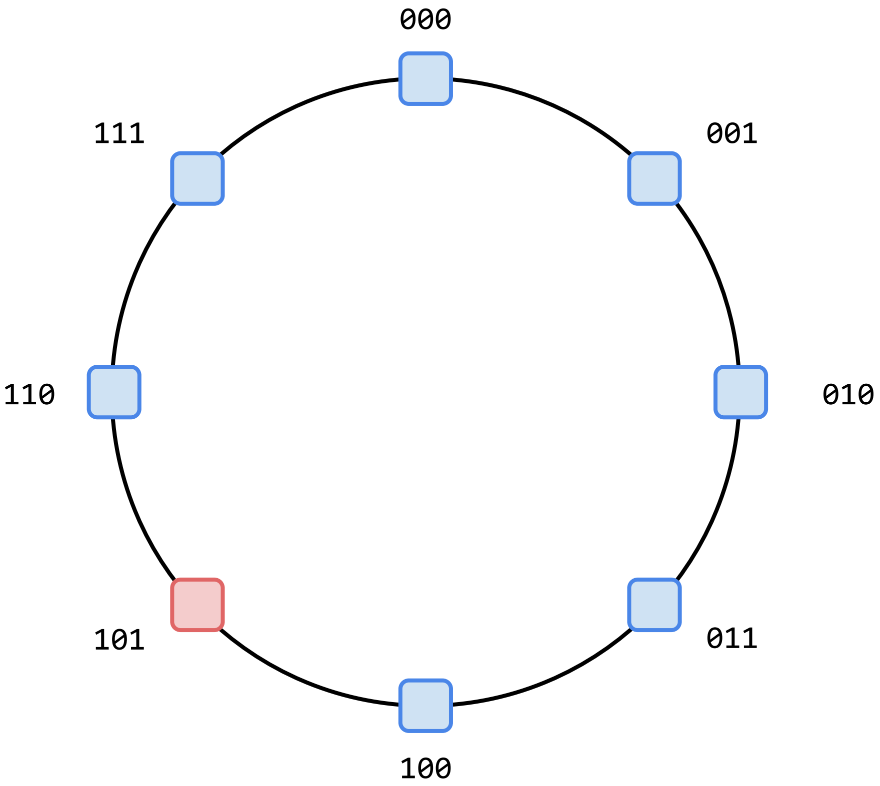
Neighbor selection - one neighbor in each finger interval

DHT Routing Geometries - Ring



Neighbor selection - one neighbor in each finger interval

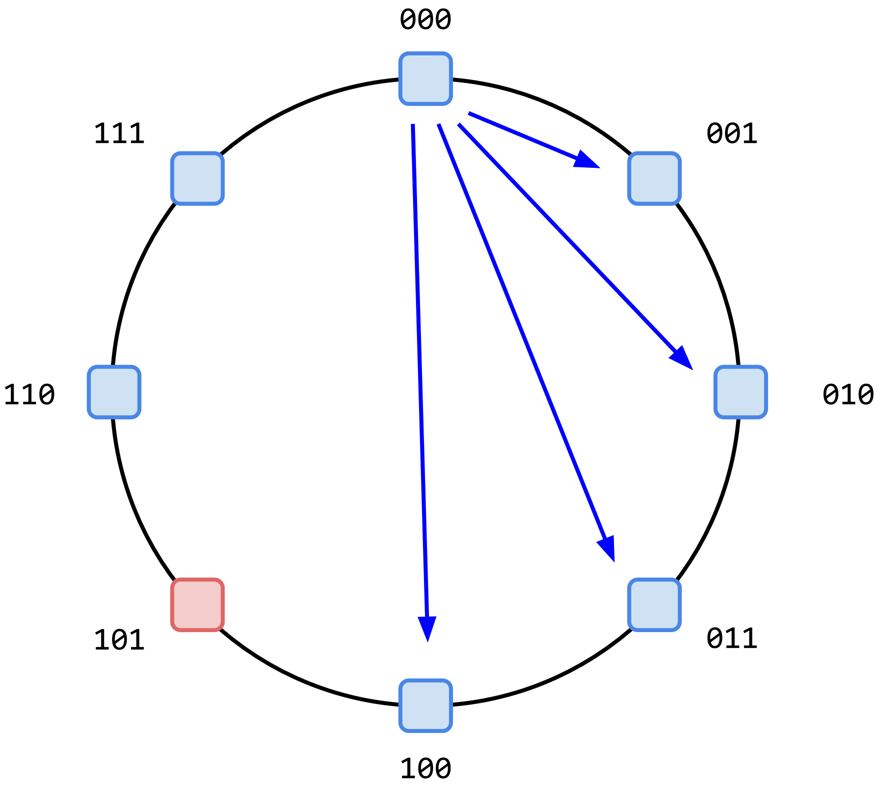
DHT Routing Geometries - Ring



Neighbor selection - one neighbor in each finger interval

Route selection - route to destination by making progress along ring

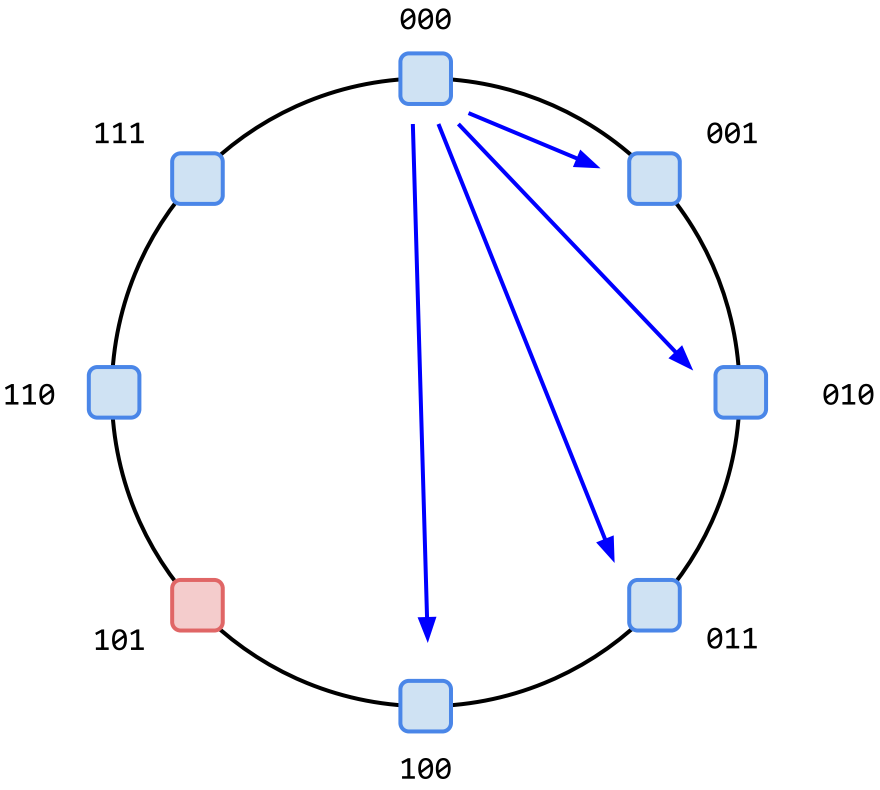
DHT Routing Geometries - Ring



Neighbor selection - one neighbor in each finger interval

Route selection - route to destination by making progress along ring

DHT Routing Geometries - Ring



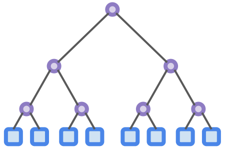
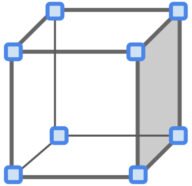
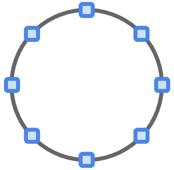
Neighbor selection - one neighbor in each finger interval

Route selection - route to destination by making progress along ring

Good flexibility in neighbor selection

Good flexibility in route selection

DHT Routing Geometries - Summary

	 Tree	 Hypercube	 Ring
Neighbor selection (Proximity)	✓		✓
Route selection (Resilience)		✓	✓