

CS 417 – DISTRIBUTED SYSTEMS

Week 7: Distributed Lookup: Part 1: Distributed Hash Tables

Paul Krzyzanowski



© 2023 Paul Krzyzanowski. No part of this content may be reproduced or reposted in whole or in part in any manner without the permission of the copyright owner.

Distributed Lookup

Interface:

```
store(key, value)  
value = lookup(key)
```

Obect storage	File storage
Flat namespace	Hierarchical namespace
Read/write entire content	Partial read/writes
Atomic access	Concurrent access (usually)

Distributed lookup: cooperating set of nodes store & retrieve data

Ideally:

- **Peer-to-peer:** no central coordinator – all nodes have the same capabilities
- **Efficient:** route requests to the node that holds the data
- **Fault-tolerant:** some nodes can be down
- **Scalable:** Easy to add or remove nodes as capacity changes

Approaches

1. Central coordinator

- Napster

2. Flooding

- Gnutella

3. Distributed hash tables

- CAN, Chord, Amazon Dynamo, Tapestry, Kademlia, ...

1. Central Coordinator

Example: Napster

- Central directory
- Identifies content (names) and the servers that host it
- *lookup(name) → {list of servers}*
- Download from any of available servers
 - Pick the best one by pinging and comparing response times

Another example: GFS

- Controlled environment compared to Napster
- Content for a given key is broken into chunks
- Master handles all queries ... but not the data

1. Central Coordinator - Napster

- Pros

- Super simple
- Search is handled by a single server
- The directory server is a single point of control
 - Provides definitive answers to a query

- Cons

- Master has to maintain state of all peers
- Server gets all the queries
- The directory server is a single point of control
 - No directory, no service!

2. Query Flooding

Example: Gnutella distributed file sharing

- Each node joins a group – but only knows about some group members
 - Well-known nodes act as **anchors**
 - New nodes with files inform an anchor about their existence
 - Nodes use other nodes they know about as peers

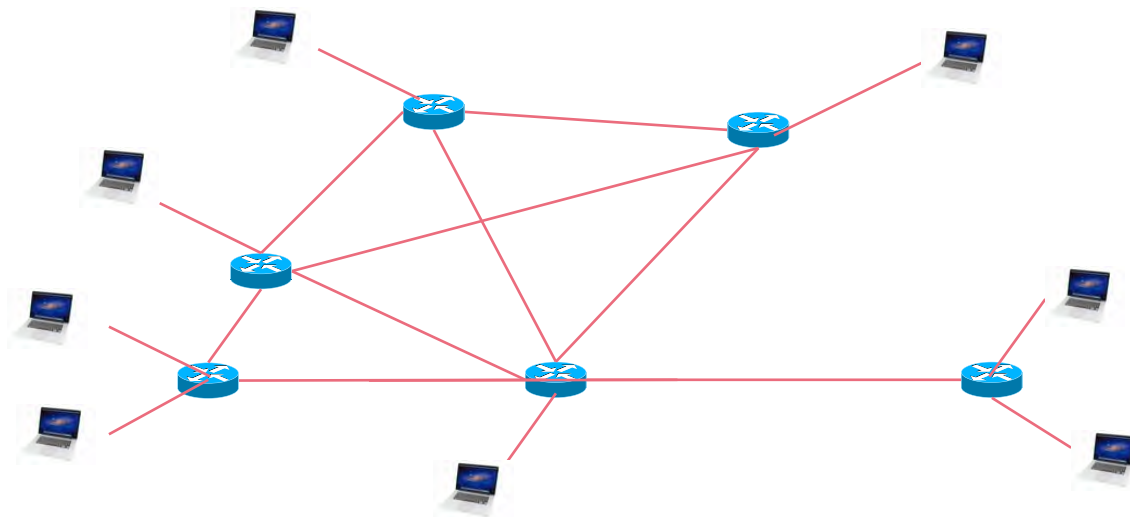
2. Query Flooding

- Send a query to peers if a file is not present locally
 - Each request contains:
 - Query key
 - Unique request ID
 - Time to Live (TTL, maximum hop count)
- Peer either responds or routes the query to its neighbors
 - Repeat until $TTL = 0$ or if the request ID has been processed
 - If found, return a response containing the node address to the requestor
 - **Back propagation**: response hops back to reach originator

Overlay network

An **overlay network** is a virtual network formed by **peer connections**

- Any node might know about a small set of machines
- “Neighbors” may not be physically close to you

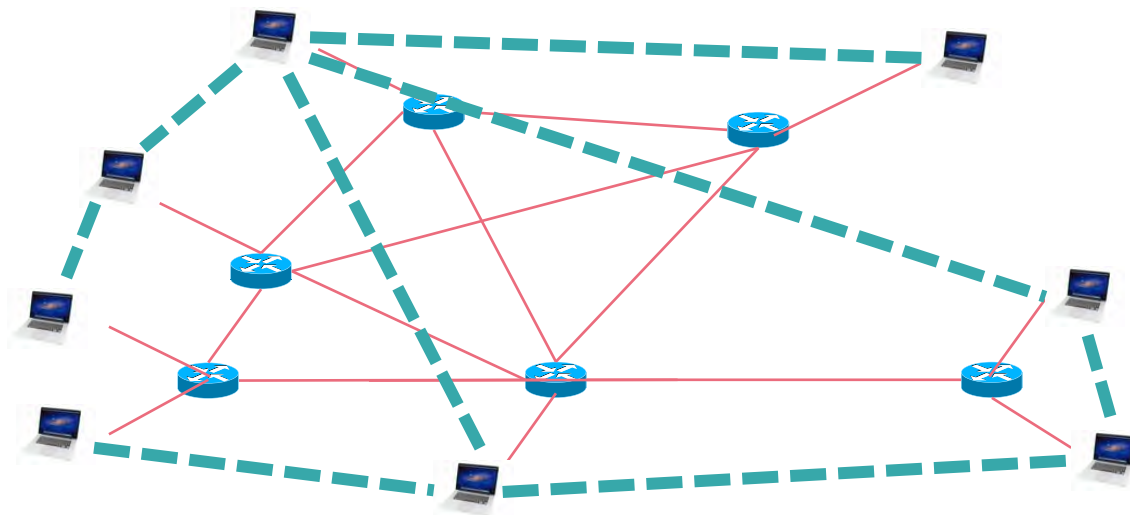


Underlying IP Network

Overlay network

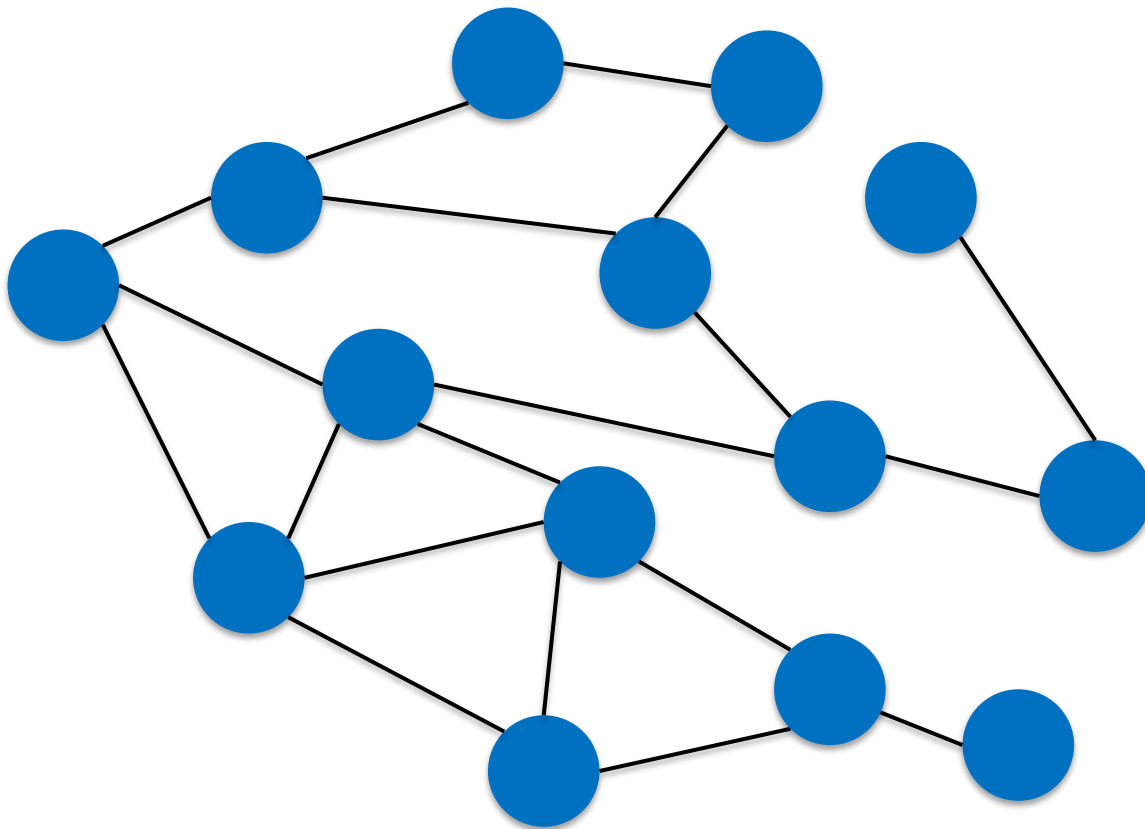
An **overlay network** is a virtual network formed by **peer connections**

- Any node might know about a small set of machines
- “Neighbors” may not be physically close to you

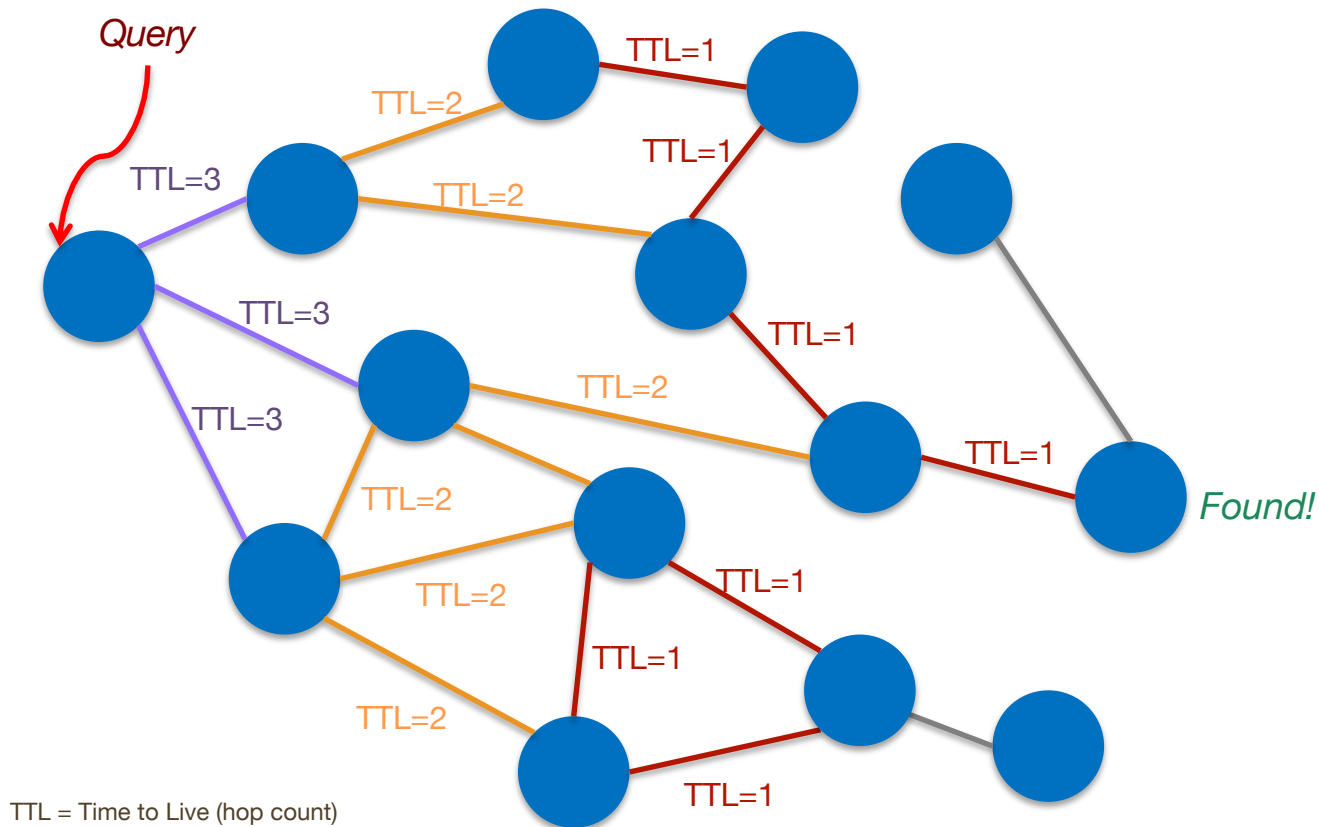


Overlay Network

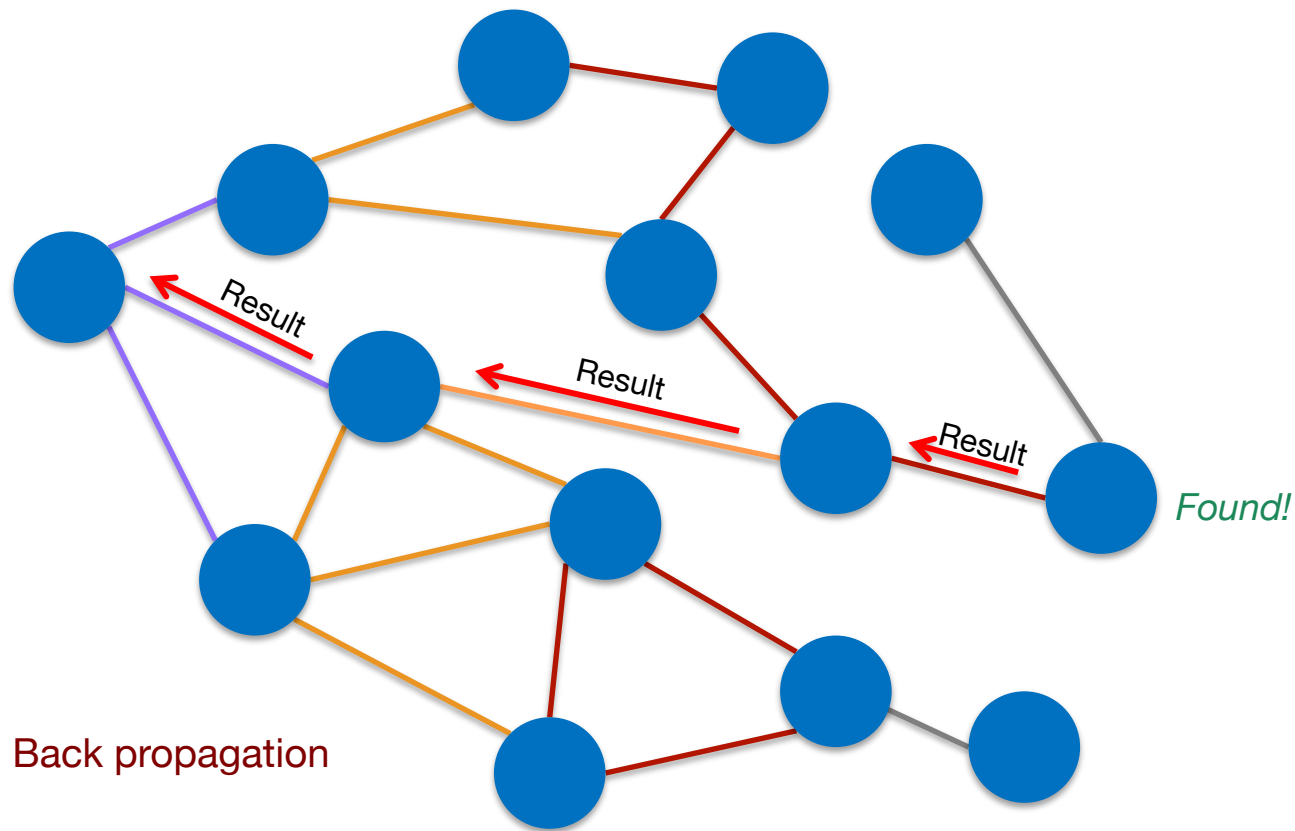
Flooding Example: Overlay Network



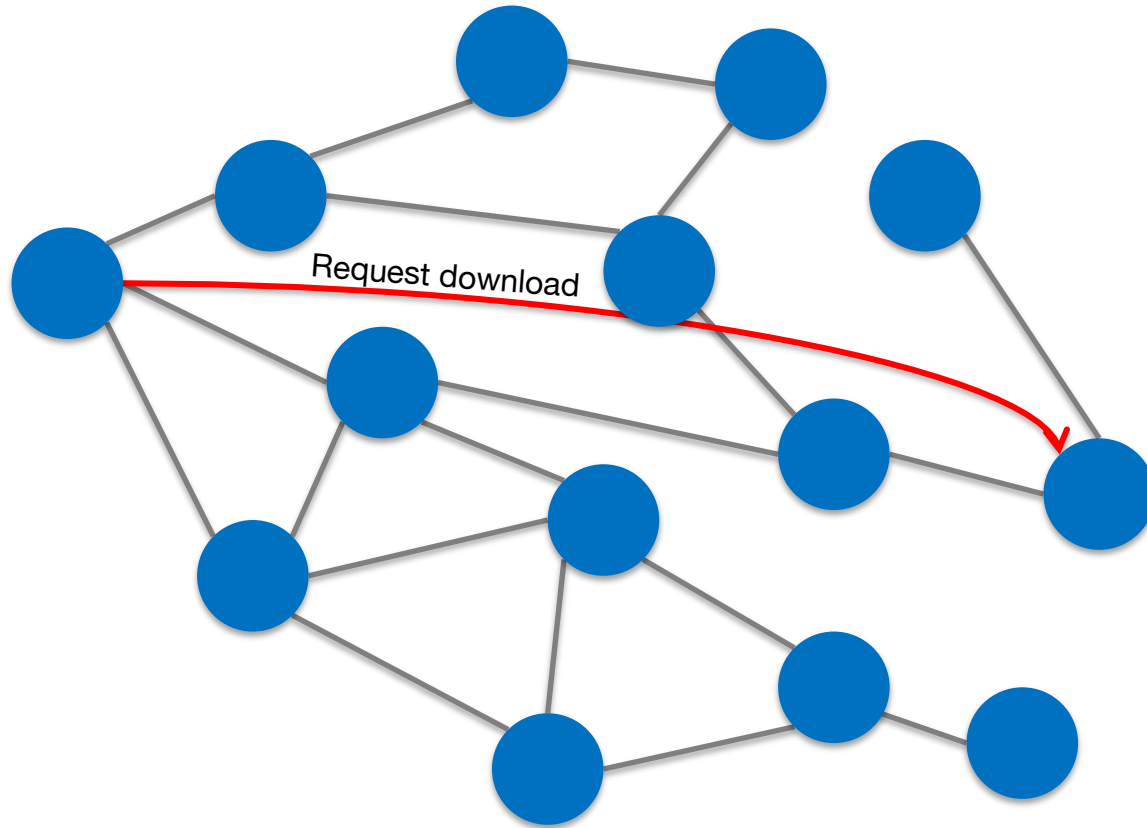
Flooding Example: Query Flood



Flooding Example: Query response



Flooding Example: Download



What's wrong with flooding?

- Some nodes are not always up, and some are slower than others
 - Kazaa was created to deal with this: **super nodes** = more powerful with better uptime
 - Gnutella later did the same, classifying some nodes as special (“**ultrapeers**”)
 - Regular nodes send all content info to ultrapeers
- Poor use of network resources
 - Lots of messages throughout the entire network (until TTL=0 kicks in)
- Potentially high latency
 - Requests get forwarded from one machine to another
 - If back propagation is used:
replies go through the same sequence of systems used in the query, increasing latency even more – useful in preserving anonymity

3. Distributed Hash Tables

Hash tables

Remember hash functions & hash tables?

- Linear search: $O(N)$
- Tree or binary search: $O(\log_2 N)$
- Hash table: $O(1)$

What's a hash function? (refresher)

Hash function

- A function that takes a variable length input (e.g., a string or any object) and generates a (usually smaller) fixed length result (i.e., an integer)

- Example: hash strings to a range 0-7:

`hash("Newark") → 1`

`hash("Jersey City") → 6`

`hash("Paterson") → 2`

Hash table

- Table of *(key, value)* tuples

- Look up a key:

Hash function maps *keys* to a range $0 \dots N-1$

Index into a table of N elements

`i = hash(key)`

`item = table[i]`

- No need to search through the table!

Considerations with hash tables (refresher)

- Picking a good hash function
 - We want uniform distribution of all values of *key* over the space $0 \dots N-1$
- Collisions
 - Multiple keys may hash to the same value
 - `hash("Paterson")` → 2
 - `hash("Edison")` → 2
 - `table[i]` is a **bucket** (**slot**) for all such (*key*, *value*) sets
 - Within `table[i]`, use a linked list or another layer of hashing
- Think about a hash table that grows or shrinks
 - If we add or remove buckets → need to rehash keys and move items

Distributed Hash Tables (DHT): Goal

Create a peer-to-peer version of a *(key, value)* data store

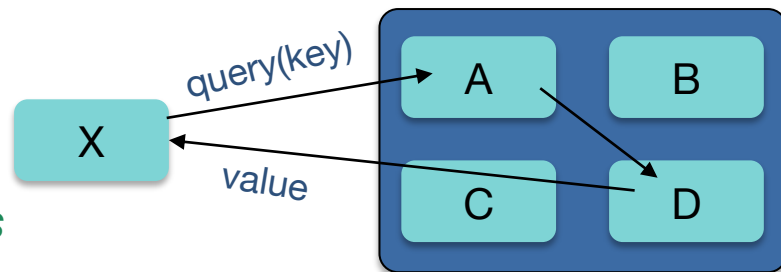
How we want it to work

1. A client (*X*) queries any peer (*A*) in the data store with a key
2. The data store finds the peer (*D*) that has the value
3. That peer (*D*) returns the *value* for the key to the client

Distributed Hash Table
Object Storage

Make it efficient!

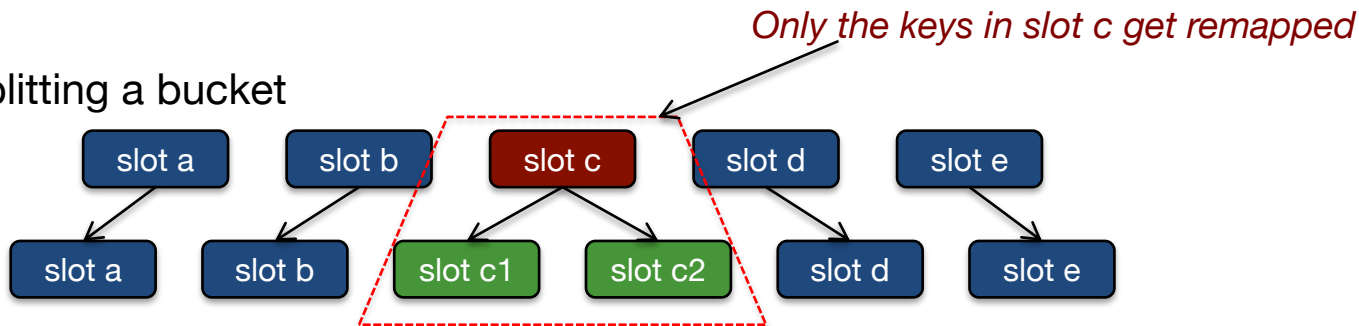
*A query should not generate a flood
or go be forwarded through too many nodes*



Consistent hashing

- Conventional hashing
 - Practically all keys must be remapped if the table size changes
- **Consistent hashing**
 - Most keys will hash to the same value as before
 - On average, K/n keys will need to be remapped
 $K = \# \text{ keys}, n = \# \text{ of buckets}$

Example: splitting a bucket



Designing a distributed hash table

- Spread the hash table across multiple nodes (peers)
- Each node stores a portion of the key space – it's a bucket

lookup(key) → node ID that holds (key, value)

lookup(node_ID, key) → value

Questions

How do we partition the data & do the lookup?

& keep the system decentralized?

& make the system scalable (lots of nodes with dynamic changes)?

& fault tolerant (replicated data)?

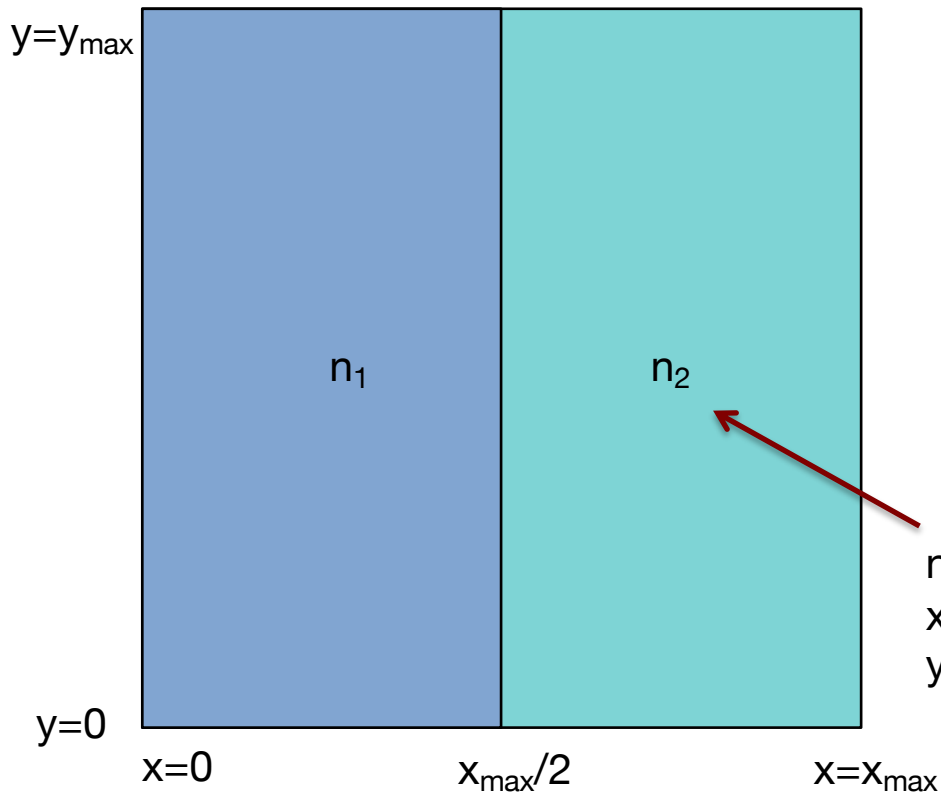
Distributed Hashing

CAN: Content Addressable Network

CAN design

- Create a logical grid
 - x-y in 2-D (but not limited to two dimensions)
- Separate hash function per dimension
 - $h_x(\text{key})$, $h_y(\text{key})$
- A node
 - Is responsible for a range of values in each dimension
 - Knows its neighboring nodes

CAN *key*→*node* mapping: 2 nodes



$x = \text{hash}_x(\text{key})$

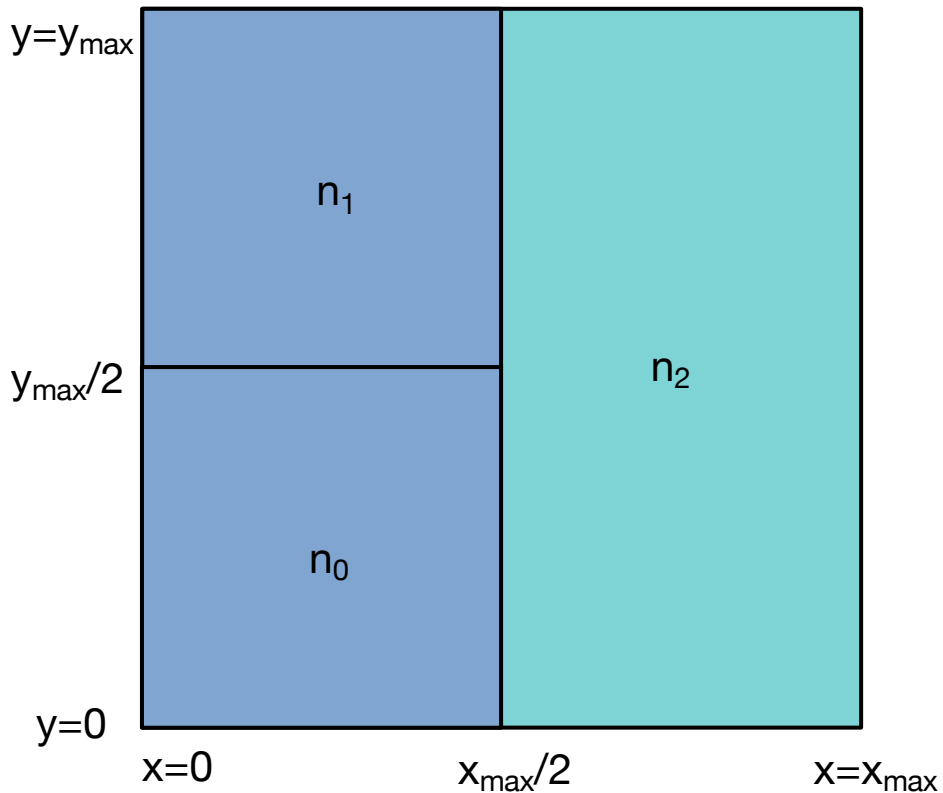
$y = \text{hash}_y(\text{key})$

if $x < (x_{\max}/2)$
 n_1 has (*key*, *value*)

if $x \geq (x_{\max}/2)$
 n_2 has (*key*, *value*)

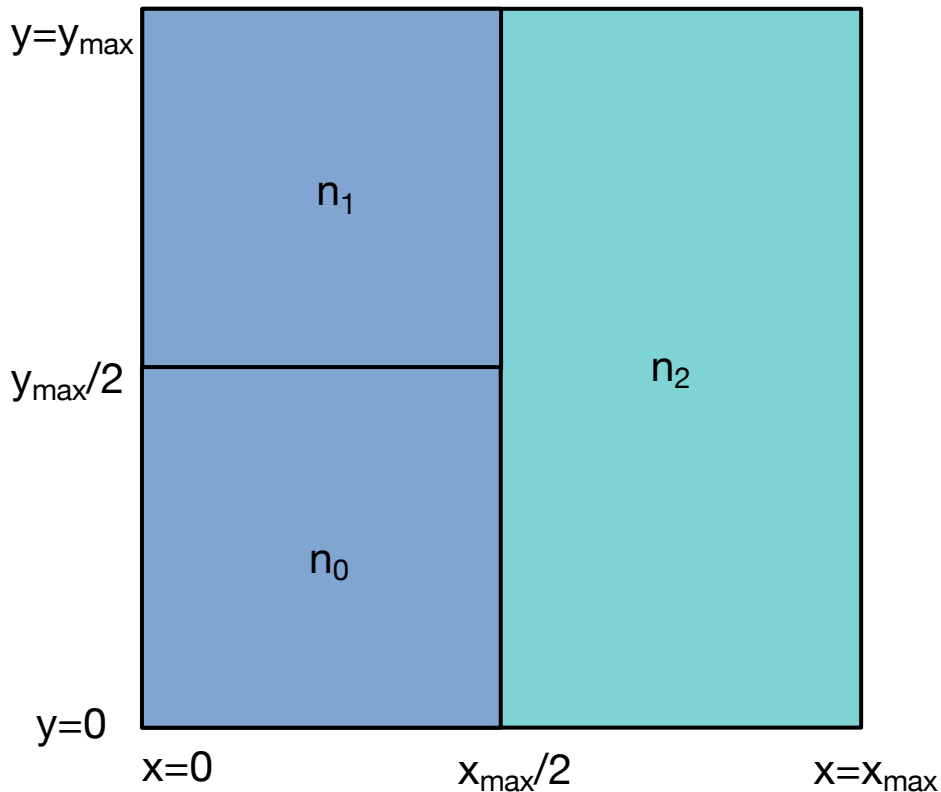
n_2 is responsible for a **zone**
 $x=(x_{\max}/2 \dots x_{\max})$,
 $y=(0 \dots y_{\max})$

CAN partitioning



Any node can be split in two – either horizontally or vertically

CAN key→node mapping



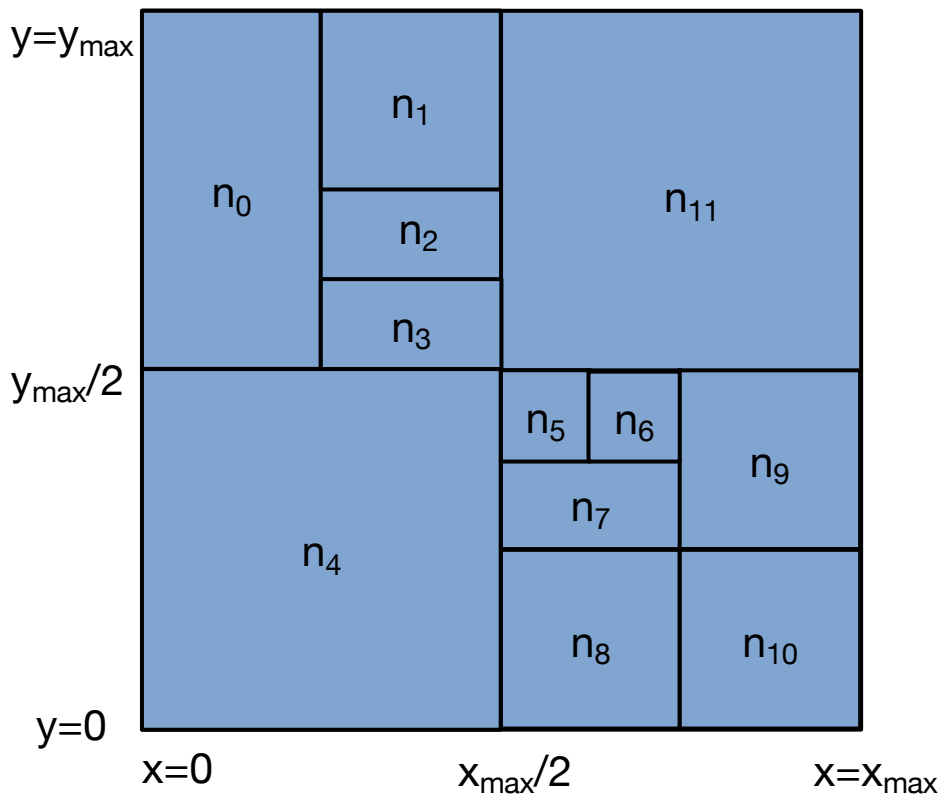
$x = \text{hash}_x(\text{key})$

$y = \text{hash}_y(\text{key})$

```
if  $x < (x_{\max}/2)$  {  
  if  $y < (y_{\max}/2)$   
     $n_0$  has (key, value)  
  else  
     $n_1$  has (key, value)  
}
```

```
if  $x \geq (x_{\max}/2)$   
   $n_2$  has (key, value)
```

CAN partitioning



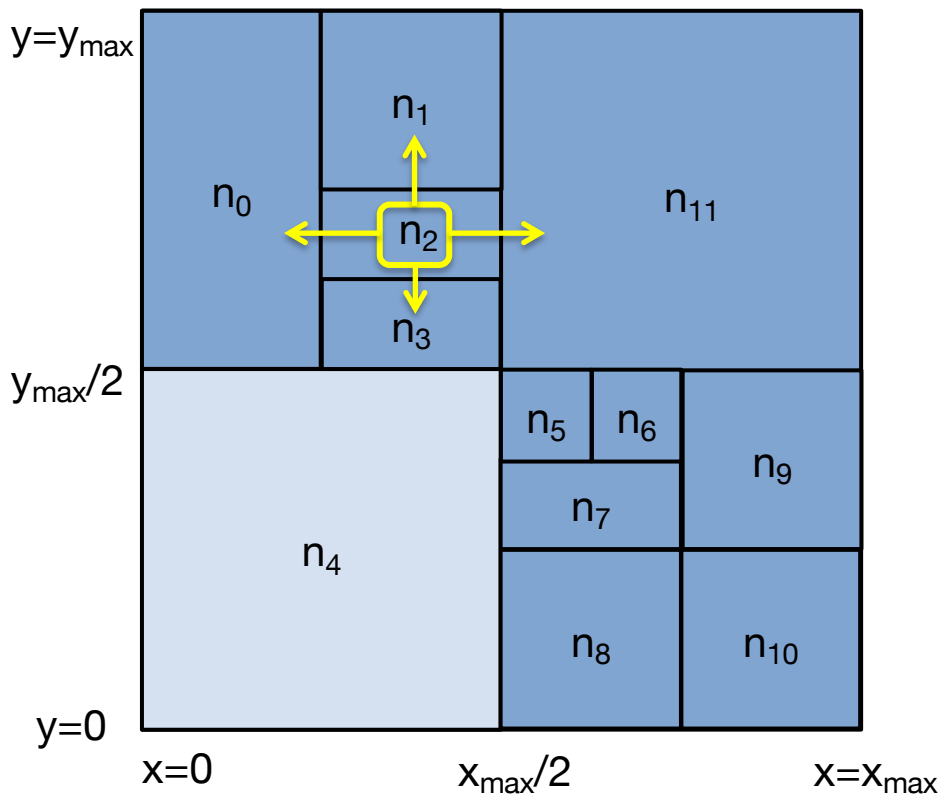
Any node can be split in two:
either horizontally or vertically

Some data must be moved to
the new node based on
hash(key)

Neighbors need to be made
aware of the new node

A node needs to know only
**one neighbor in each
direction**

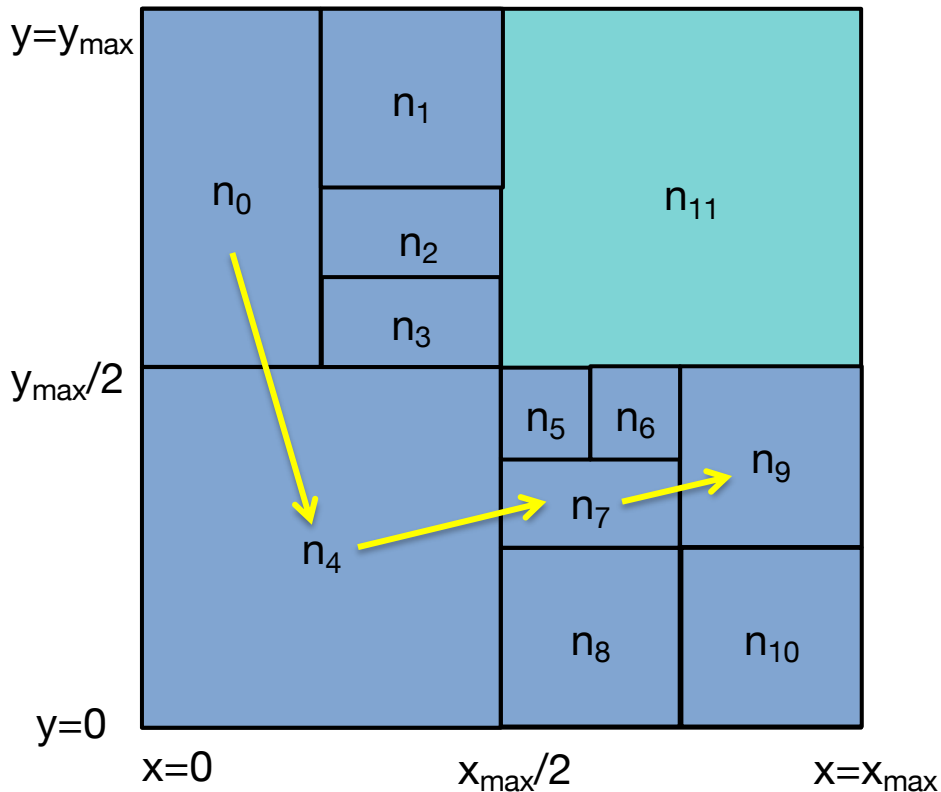
CAN neighbors



Neighbors refer to nodes that share **adjacent zones** in the **overlay network**

n_4 only needs to keep track of n_5 , n_7 , or n_8 as its right neighbor.

CAN routing



lookup(key):

Compute

hash_x(key), hash_y(key)

If the node is responsible for the (x, y) value then look up the key locally

Otherwise route the query to a neighboring node

CAN

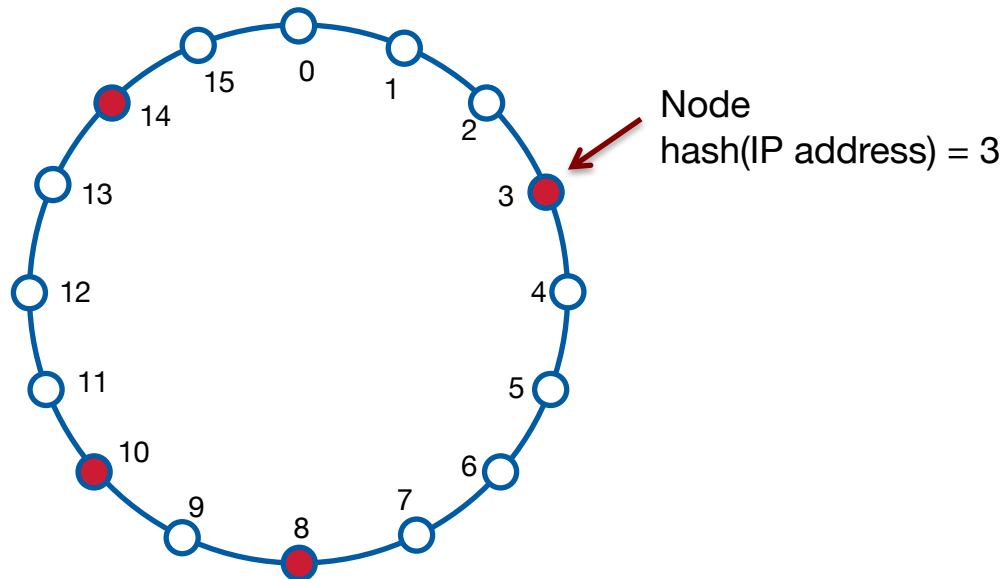
- Performance
 - For n nodes in d dimensions
 - # neighbors = $2d$
 - Average route for 2 dimensions = $O(\sqrt{n})$ hops
- To handle failures
 - Share knowledge of neighbor's neighbors
 - One of the node's neighbors takes over the failed zone

Distributed Hashing Case Study

Chord

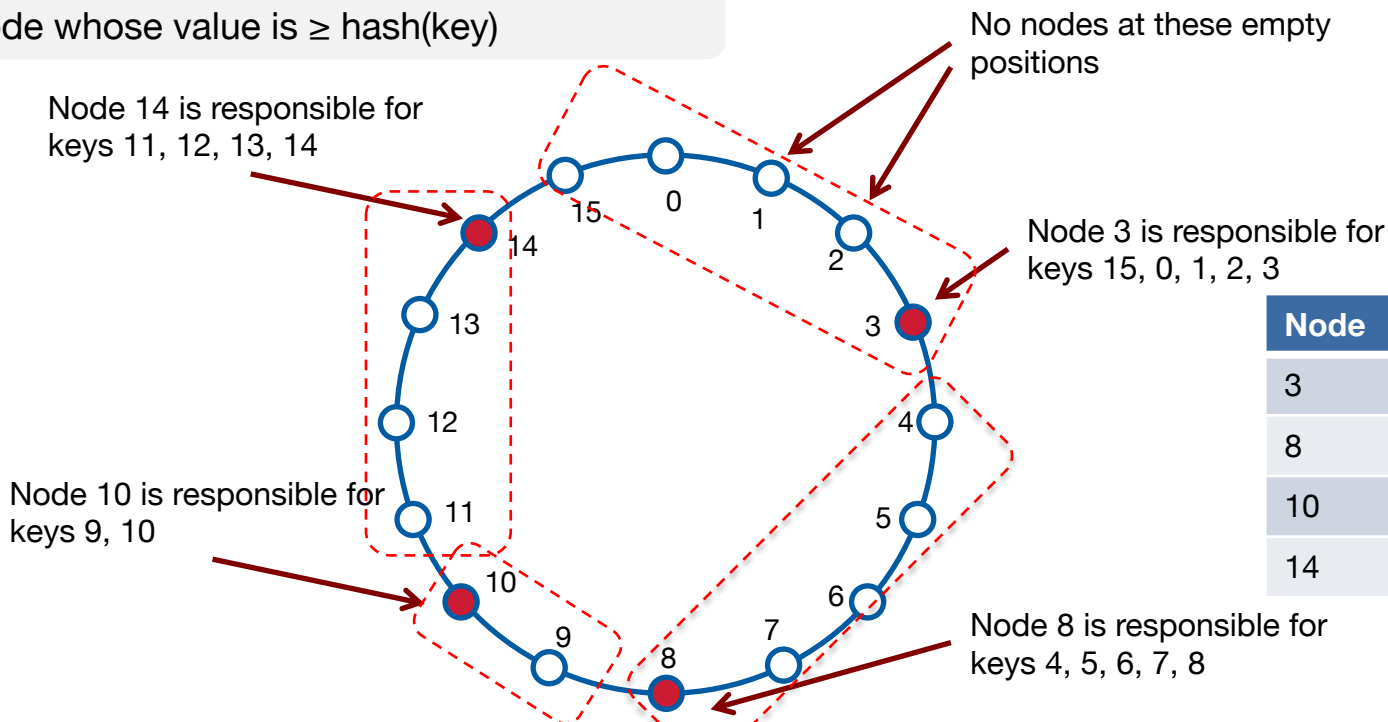
Chord & consistent hashing

- A key is hashed to an m -bit value: $0 \dots (2^m-1)$
- A logical ring is constructed for the values $0 \dots (2^m-1)$
- Nodes are placed on the ring at *hash(IP address)*



Key assignment

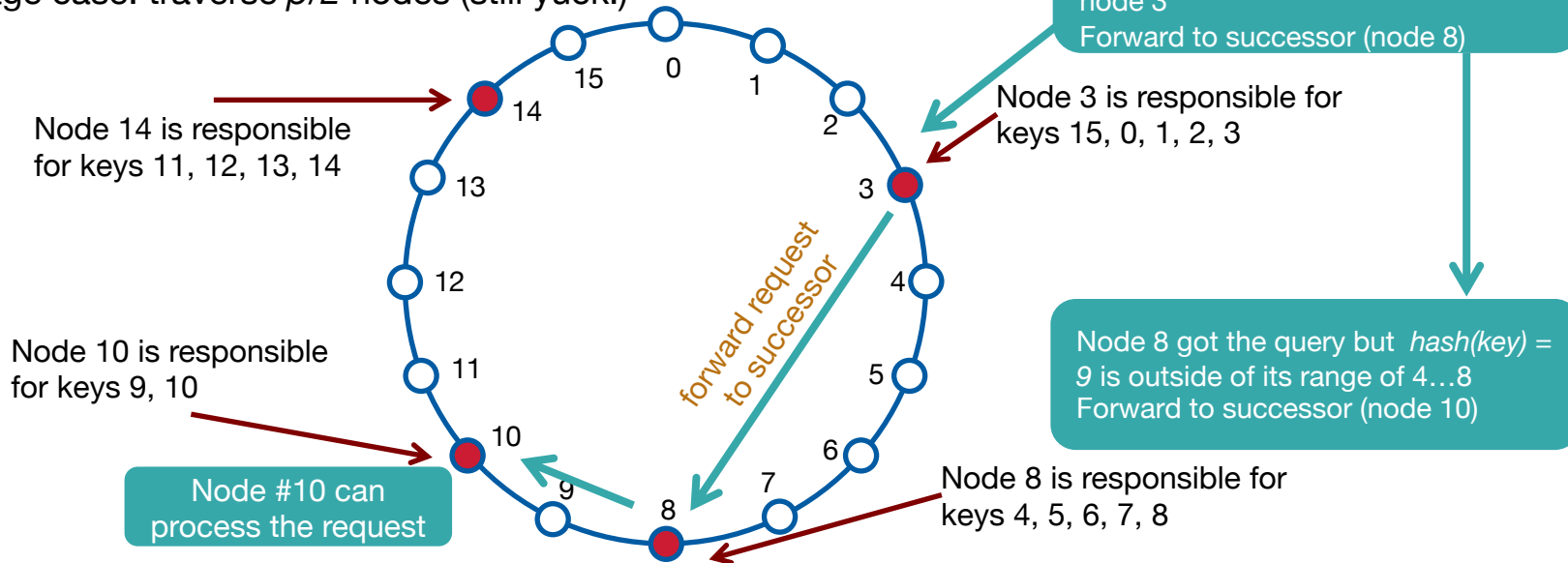
- Example: $n=16$; system with 4 nodes (so far)
- $(key, value)$ data is stored at a **successor**
 - a node whose value is $\geq \text{hash}(key)$



Node	Hash range
3	0-3, 15
8	4-8
10	9-10
14	11-14

Handling *insert* or *query* requests

- Any peer can get a request (*insert* or *query*).
- If the $\text{hash}(\text{key})$ is not for its ranges of keys, it forwards the request to the successor.
- The process continues until the responsible node is found
 - Worst case: with p nodes, traverse $p-1$ nodes; that's $O(p)$ (yuck!)
 - Average case: traverse $p/2$ nodes (still yuck!)

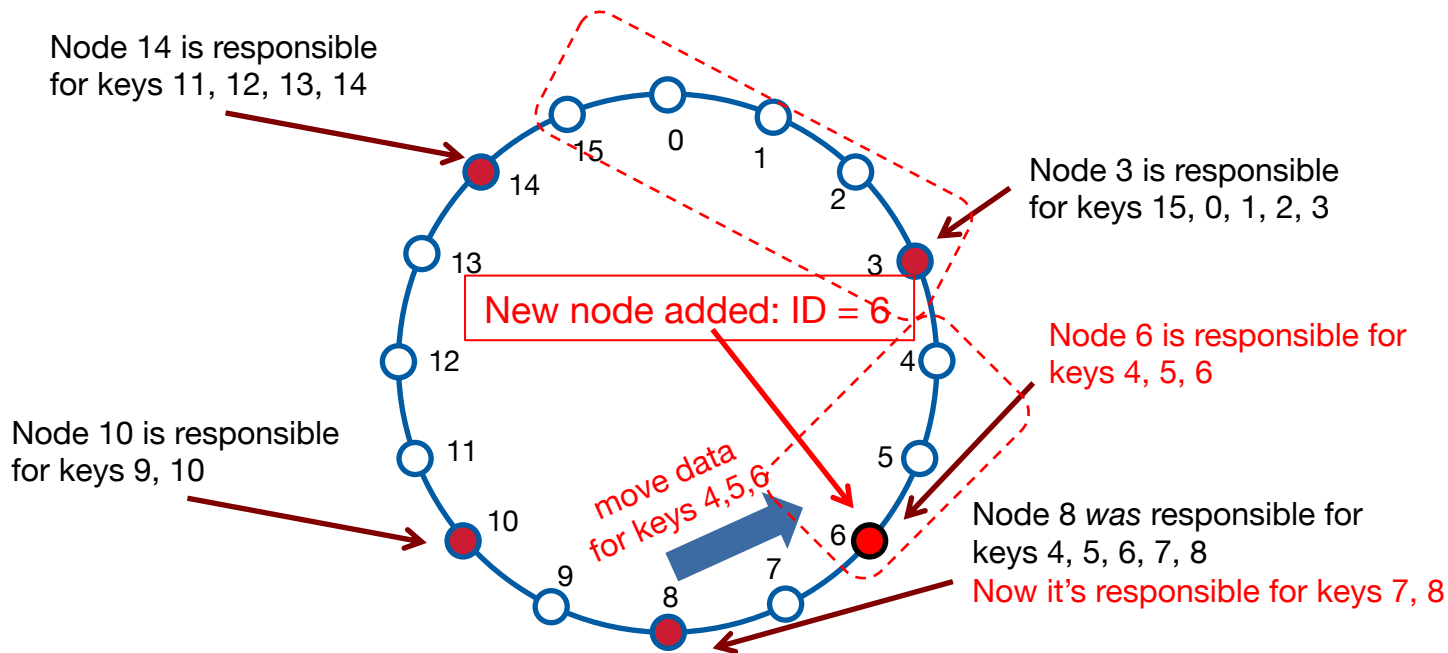


Let's figure out three more things

1. Adding/removing nodes
2. Improving lookup time
3. Providing fault tolerance

Adding a node

- Some keys that were assigned to a node's successor now get assigned to the new node
- Data for those *(key, value)* pairs must be moved to the new node

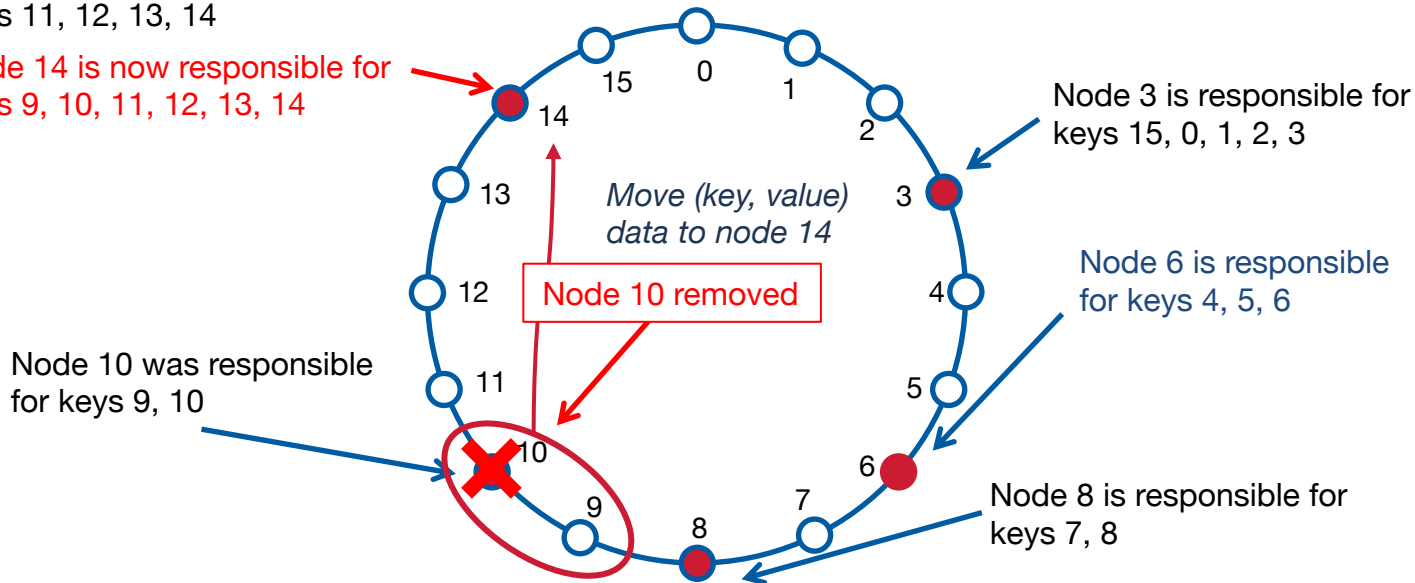


Removing a node

- Keys are reassigned to the node's successor
- Data for those (key, value) pairs must be moved to the successor

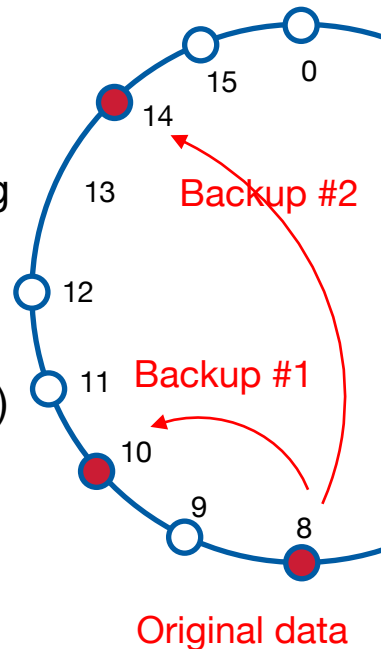
Node 14 was responsible for
keys 11, 12, 13, 14

Node 14 is now responsible for
keys 9, 10, 11, 12, 13, 14



Fault tolerance

- Nodes might die
 - (key, value) data should be replicated
 - Create R replicas, storing each one at $R-1$ successor nodes in the ring
- Need to know multiple successors
 - A node needs to know how to find its successor's successor (or more)
 - Easy if it knows all nodes!
 - When a node is back up, it needs to:
 - Check with successors for updates of data it owns
 - Check with predecessors for updates of data it stores as backups

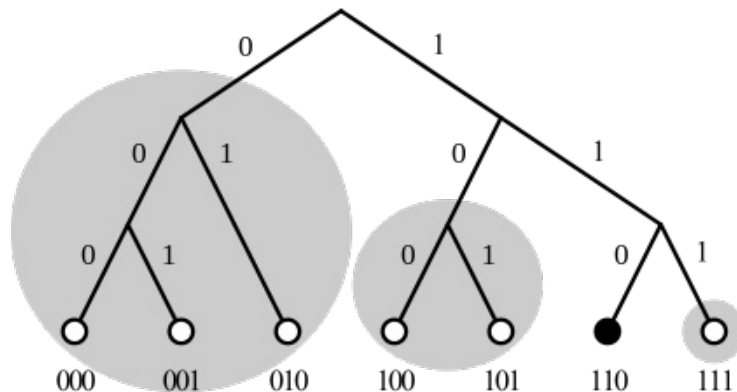


Performance

- We're not thrilled about $O(N)$ lookup
- Simple approach for great performance
 - Have all nodes know about each other
 - When a peer gets a query, it searches its table of nodes for the node that owns those values
 - Gives us $O(1)$ performance
 - Add/remove node operations must inform everyone
 - Maybe not a good solution if we have lots of peers (large tables)

BTW ... Kademlia DHT

- Similar in concept to Chord
- Uses a logical tree structure instead of a ring
- Distance = $\text{peer_1} \oplus \text{peer_2}$



Finger tables

- Compromise to avoid large tables at each node
 - Use **finger tables** to place an upper bound on the table size
- **Finger table** = partial list of nodes, progressively more distant
- At each node, i^{th} entry in finger table identifies node that succeeds it by at least 2^{i-1} in the circle
 - `finger_table[0]`: immediate (1st) successor
 - `finger_table[1]`: successor after that (2nd)
 - `finger_table[2]`: 4th successor
 - `finger_table[3]`: 8th successor
 - ...
- $O(\log N)$ nodes need to be contacted to find the node that owns a key
... not as good as $O(1)$ but way better than $O(N)$

In the kademlia DHT
finger table $\equiv$ **skip list**

Improving performance even more

- Let's revisit $O(1)$ lookup
- Each node keeps track of all current nodes in the group
 - Is that really so bad?
 - We might have thousands of nodes ... so what?
- Any node will now know which node holds a *(key, value)*
- Add or remove a node: send updates to all other nodes

Some uses of DHTs

- General purpose distributed object store: names, passwords, user profiles, ...
- Coral content delivery network, Tox instant messaging, Freenet anonymous content sharing, Scribe event notification
- **Amazon** – shopping carts, best seller lists, customer preferences, sales rank, session info, product catalog
- **BitTorrent** – distributed tracker
 - key = infohash `infohash = hash(file_contents)`
 - value = IP addresses of peers willing to serve the file
- **InterPlanetary File System (IPFS)** – 3 DHTs
 1. Find peers that have the desired file data (look up by hash of the file)
 2. Find the pathname given the file's content (hash)
 3. Get a set of addresses for a peer given its ID

The End