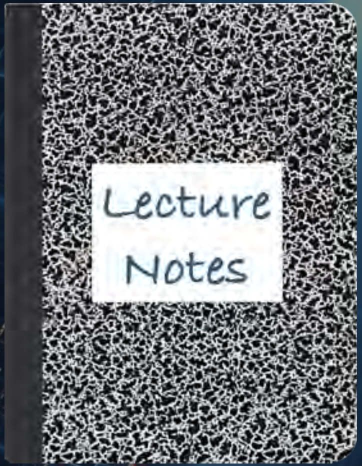
CS 417 – DISTRIBUTED SYSTEMS

Week 7: Decentralized Storage
Part 2: Distributed Hash Tables



Paul Krzyzanowski

© 2026 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)
delete(key)
```

Key-Value Storage vs. File Storage

Distributed lookup: cooperating set of nodes store & retrieve data

Ideally:

- **Peer-to-peer**
- **Efficient**
- **Fault tolerant**
- **Scalable**

- With distributed file systems, we needed to support a relatively small number of huge files, so we broke them up in chunks.
- With key-value storage, we want to support a huge number (billions) of relatively small objects. An object won't get broken into chunks.

Approaches

1. **Central coordinator (example: Napster)**
   - Users registered their files (usually music) with a coordinator
   - The coordinator becomes a bottleneck for lookups

2. **Query Flooding (examples: Gnutella, Kazaa)**
   - Peer-to-peer file sharing services flooded the network with queries
   - If a node didn't have the content, it forwarded the query to other nodes
   - Lots of hops and lots of network bandwidth

3. **Distributed hash tables**
   - CAN, Chord, Amazon Dynamo, Tapestry, Kademlia, …
   - *We'll look at CAN, Chord, and Dynamo*

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 … 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 for hash table use
  - We want a **uniform** distribution of all values of *keys* over the space *0 … 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

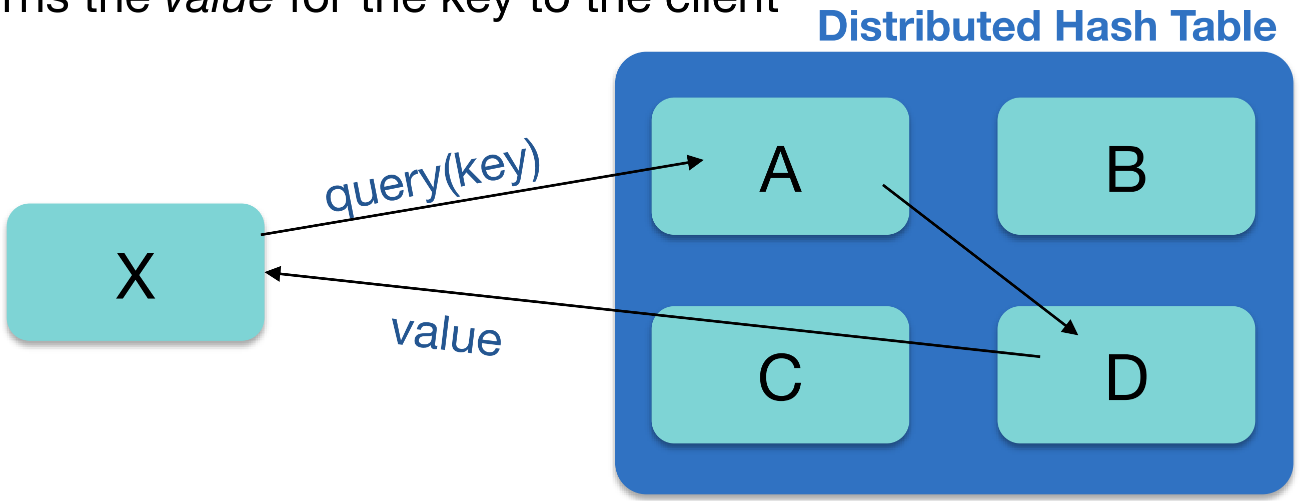
- Think about a hash table that grows or shrinks
  - If we add or remove buckets → need to rehash keys and move items

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): ***lookup(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



Keep it efficient!
- No flooding
- Minimal forwarding

Consistent hashing

Conventional hashing

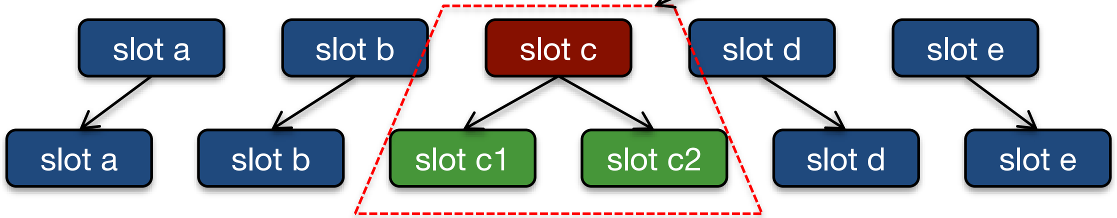
- Practically all keys must be remapped if the table size changes
- Lots of data relocation!

Consistent hashing

- On average, only K/n keys will need to be remapped

  K = # keys, n = # of buckets

Only the keys in slot c get remapped



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?

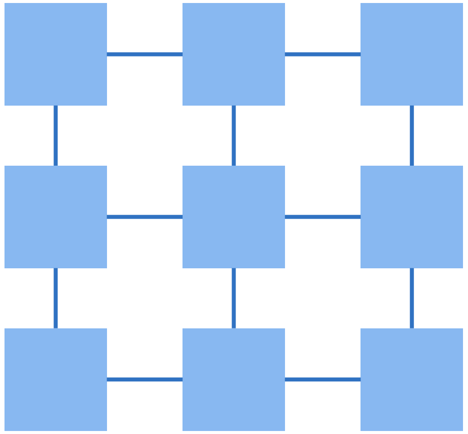
& fault tolerant?

Distributed Hashing

CAN: Content Addressable Network

CAN design

Create a logical grid



Use a separate hash function per dimension

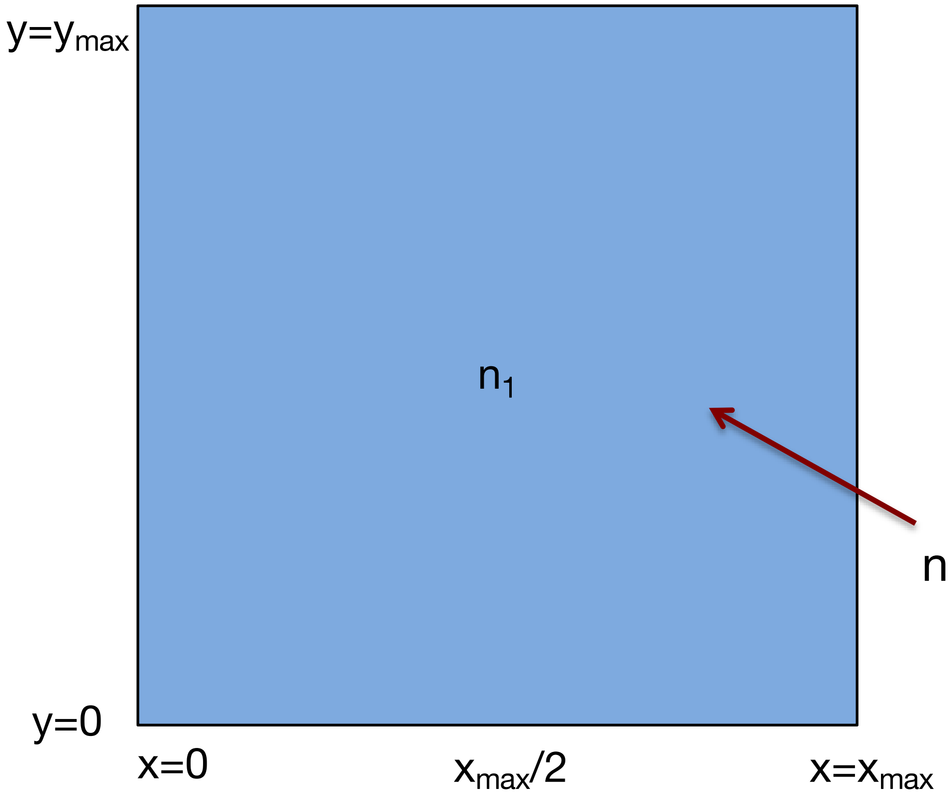
h_x(key), h_y(key)

We use cryptographic hashes (160-bit SHA1, 256-bit SHA2).
Not a different algorithm per dimension – just a different salt –
append a value, like "x" or "y"

A node

- Is responsible for a range of values in each dimension
- Knows its neighboring nodes

CAN *key→node* mapping: 1 node

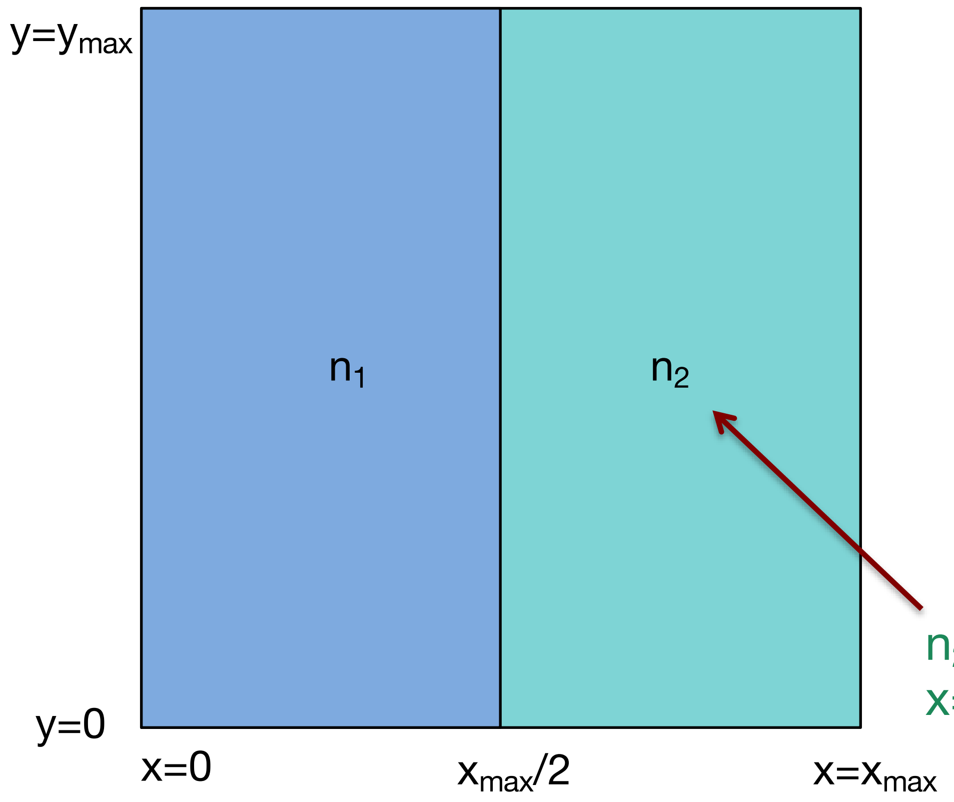


$x = \text{hash}_x(\text{key})$

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

n_1 stores everything!

CAN *key→node* mapping: 2 nodes



$x = hash_x(key)$

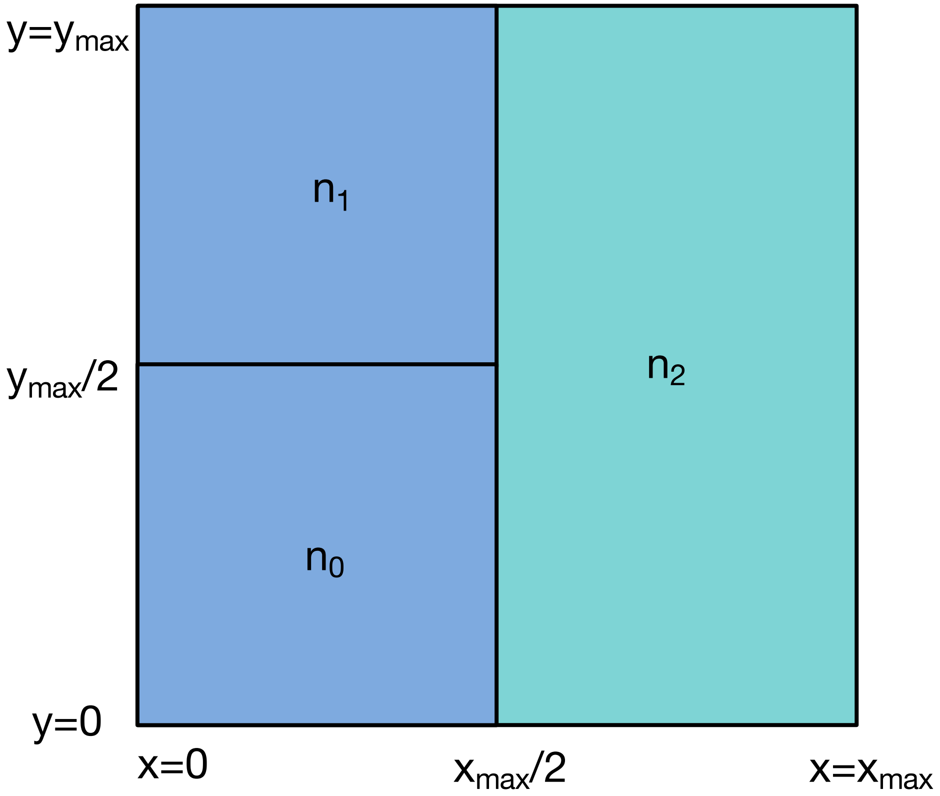
$y = hash_y(key)$

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

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

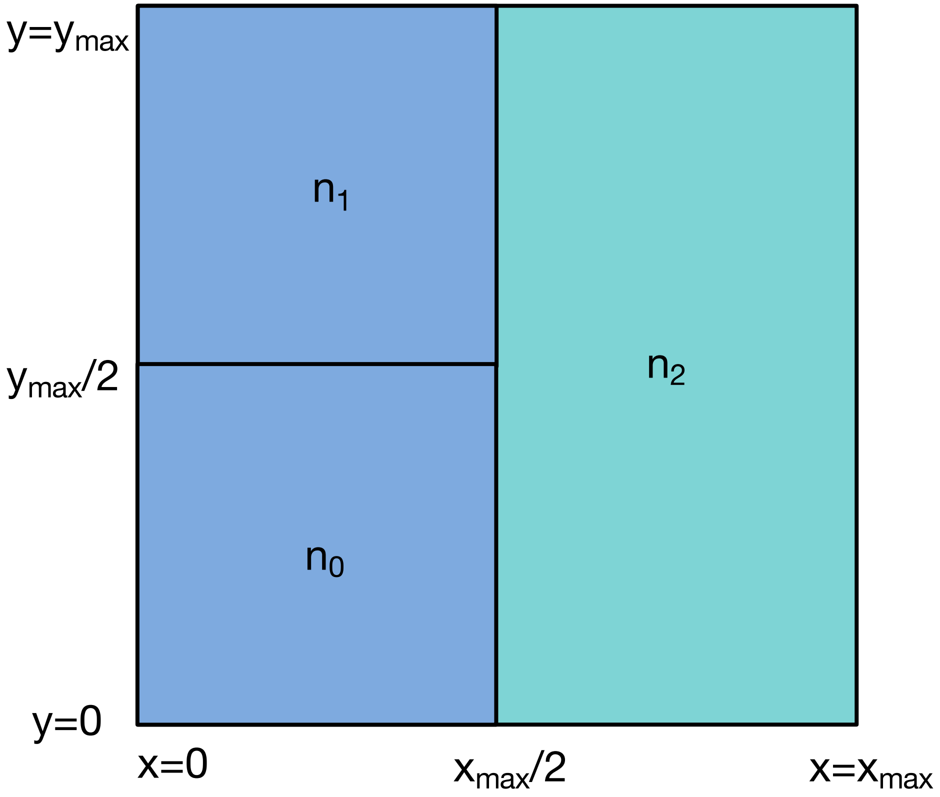
n_2 is responsible for the zone $x=(x_{max}/2 .. x_{max})$, $y=(0 .. max)$

CAN partitioning



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

CAN key→node mapping



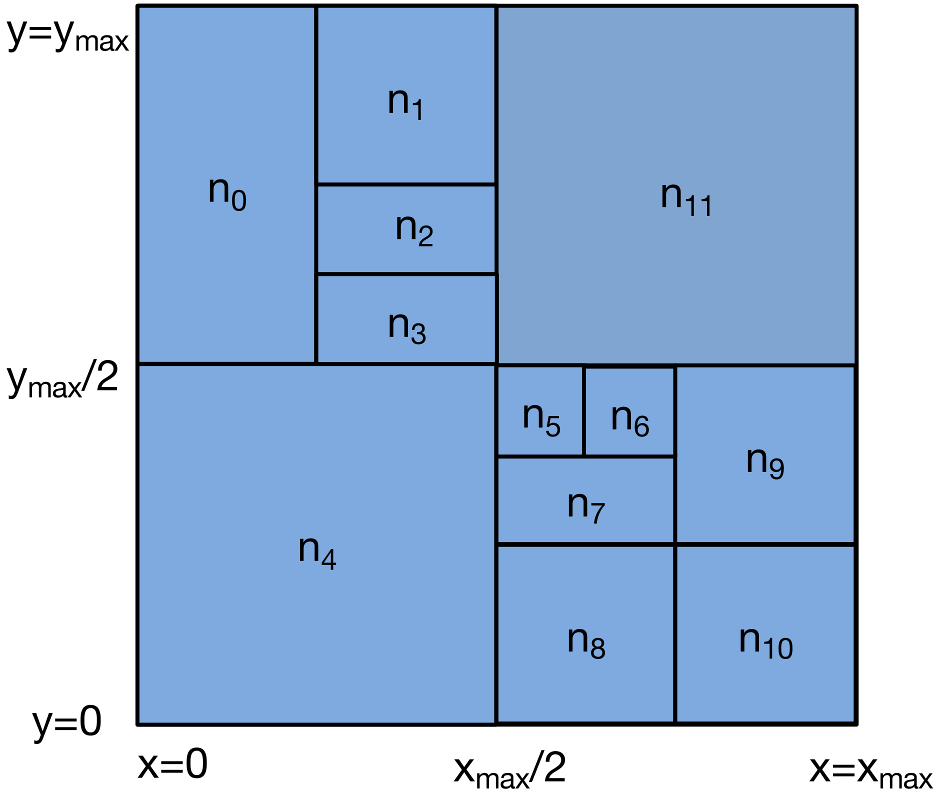
```
x = hash_x(key)

y = hash_y(key)

if x < (x_max/2) {
   if y < (y_max/2)
        n_0 has (key, value)
   else
        n_1 has (key, value)
}

if x ≥ (x_max/2)
        n_2 has (key, value)
```

CAN partitioning

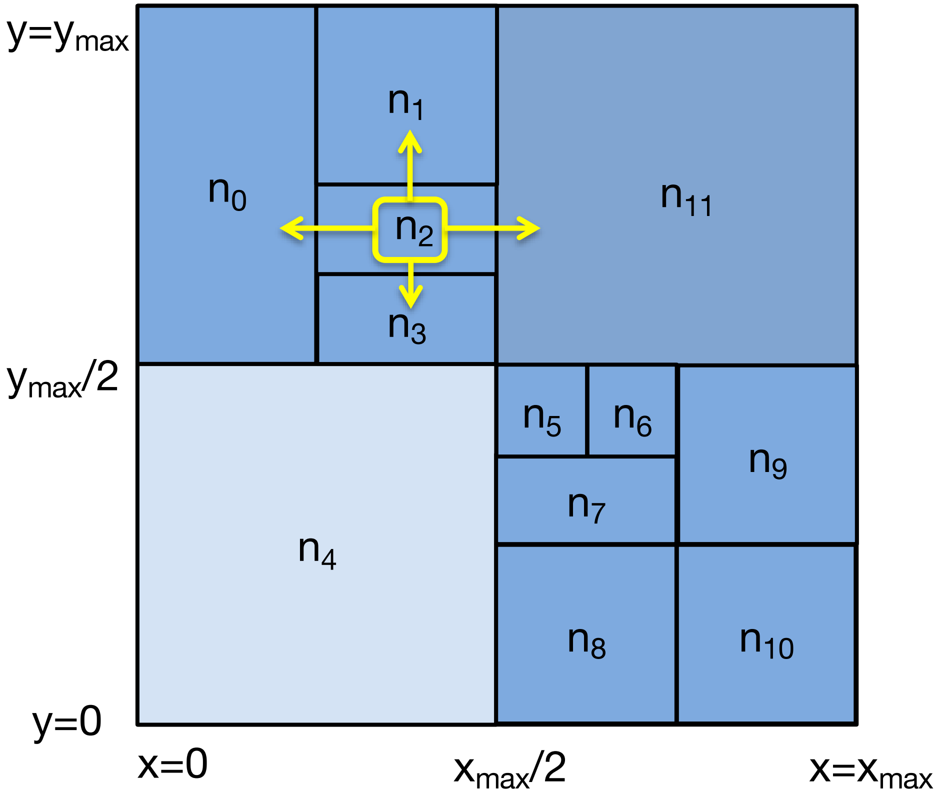


Some data must be moved to the new node based on *hash*(*key*)

Neighbors need to be made aware of the new node

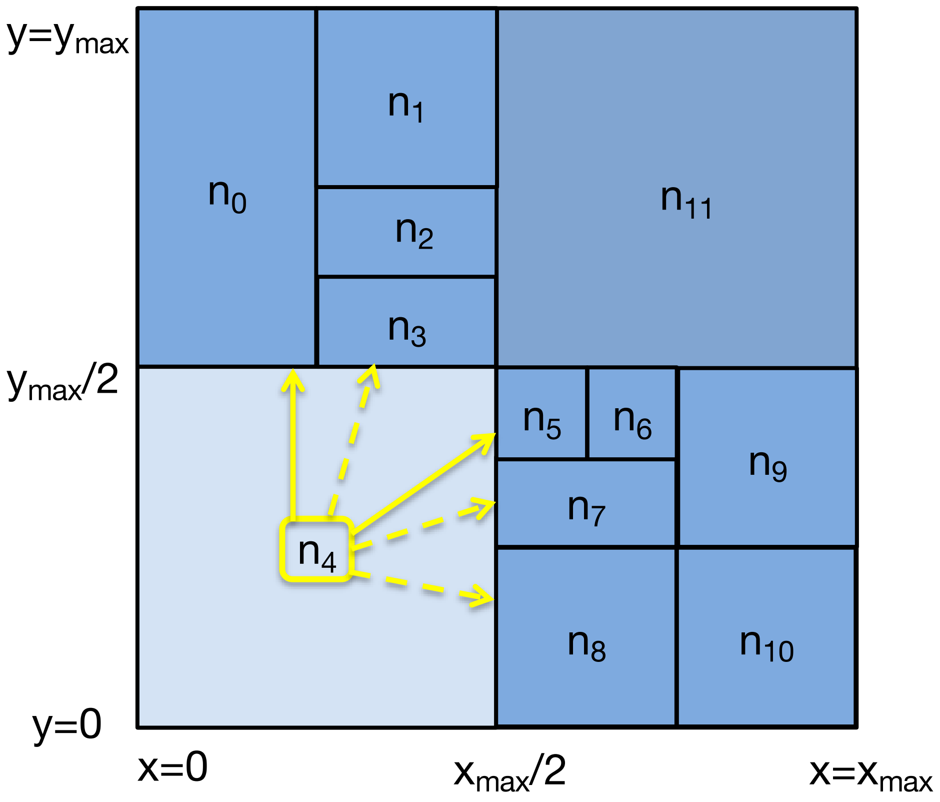
A node needs to know only one neighbor in each direction

CAN neighbors



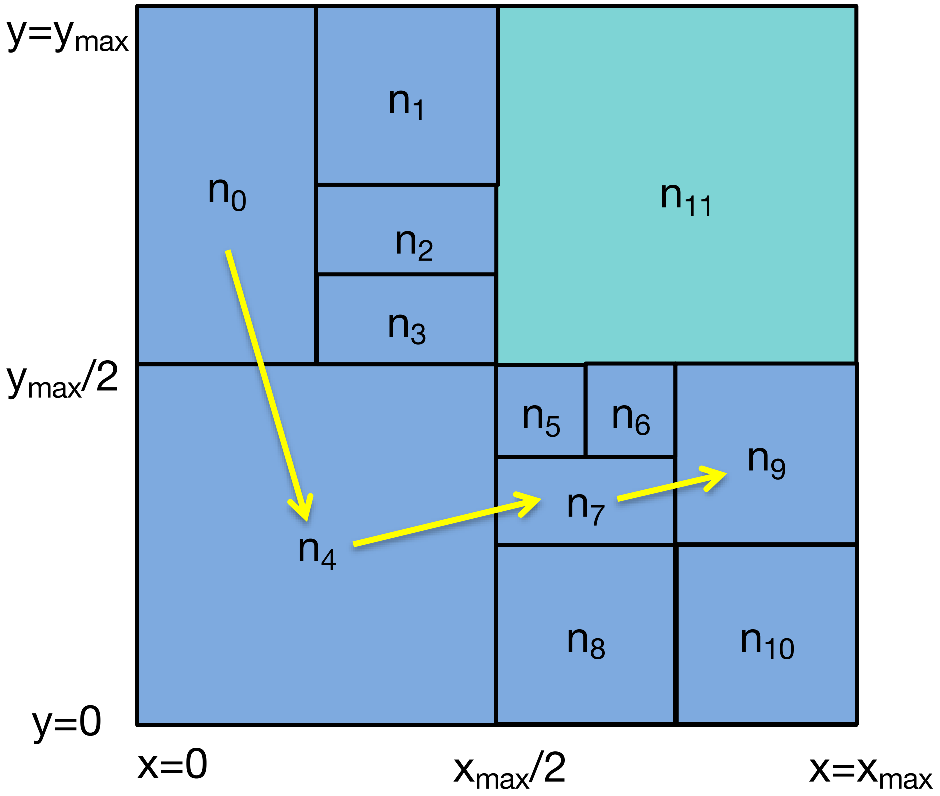
Neighbors are nodes that share adjacent zones in the overlay network

CAN neighbors



Neighbors are nodes that share adjacent zones in the overlay network

CAN routing



lookup(key):

Compute
$hash_x(key)$, $hash_y(key)$

If the node is responsible for the (x, y) value:
    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

CAN

- Performance
  - For *n* nodes in *d* dimensions
  - # neighbors = *2d*
  - Average route for 2 dimensions = $O(\sqrt{n})$ hops

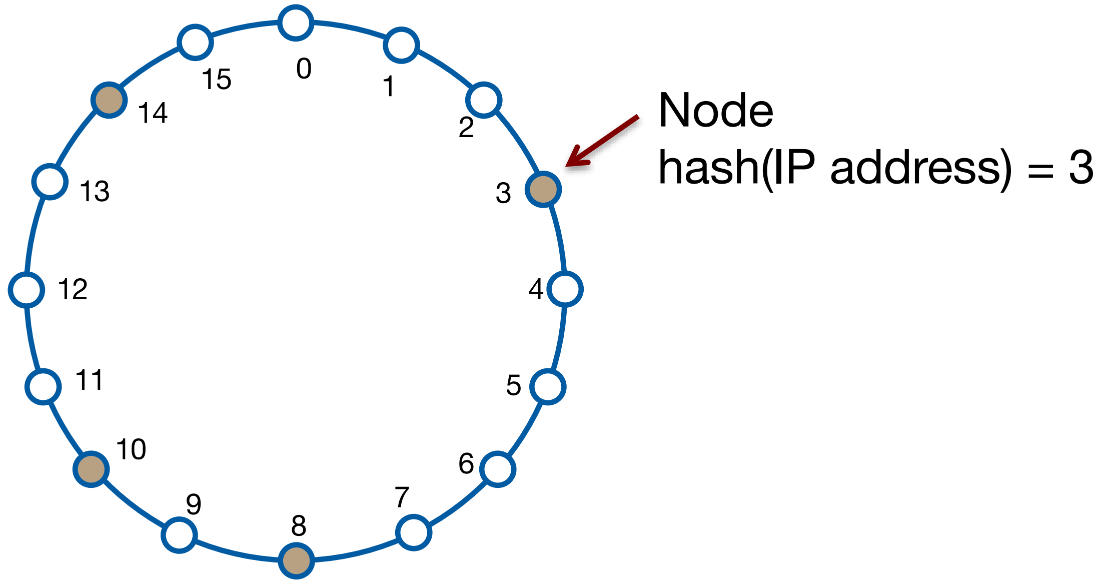
- To handle failures
  - Share knowledge of neighbor's neighbors
  - One of the node's neighbors takes over the failed zone

Distributed Hashing

Chord

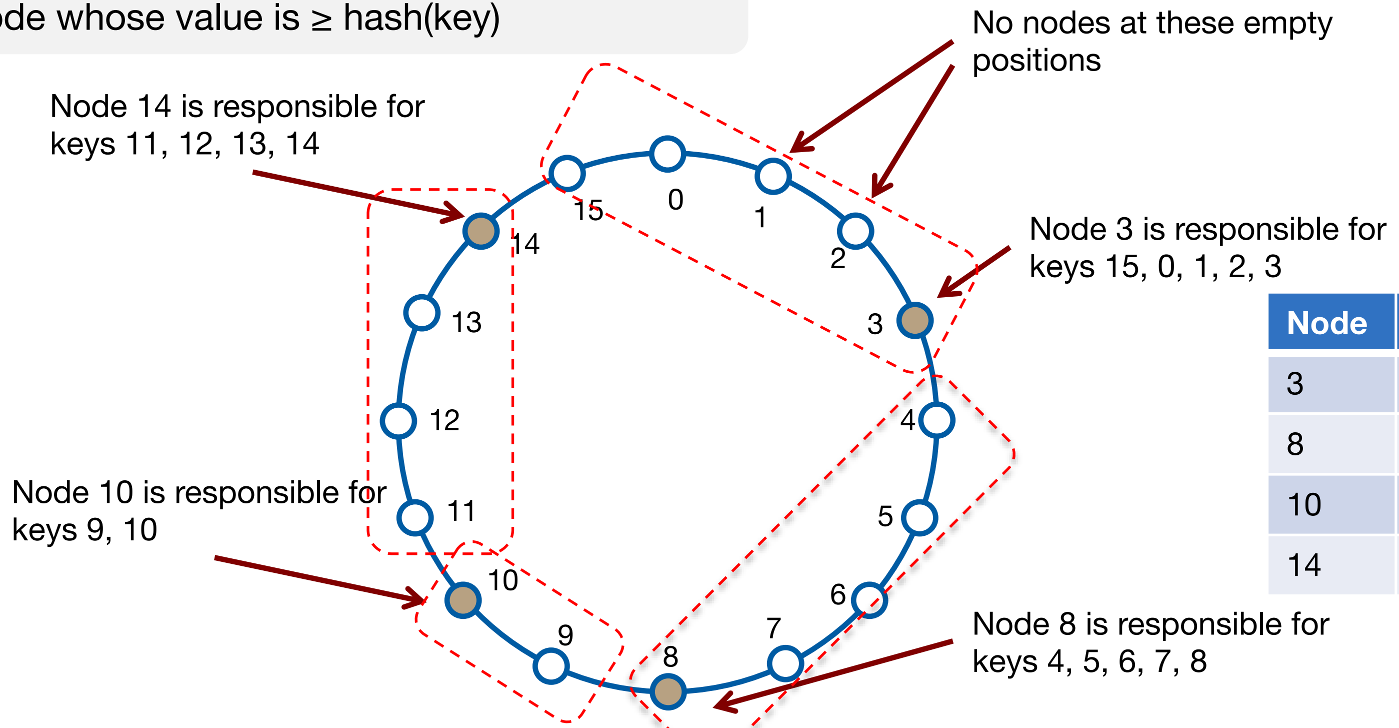
Chord & consistent hashing

- hash(key) → *m*-bit value: 0 … (2^m-1)
- Logical ring for all values 0 ... (2^m-1)
- Nodes are placed on the ring at *hash(IP address)*



Key assignment

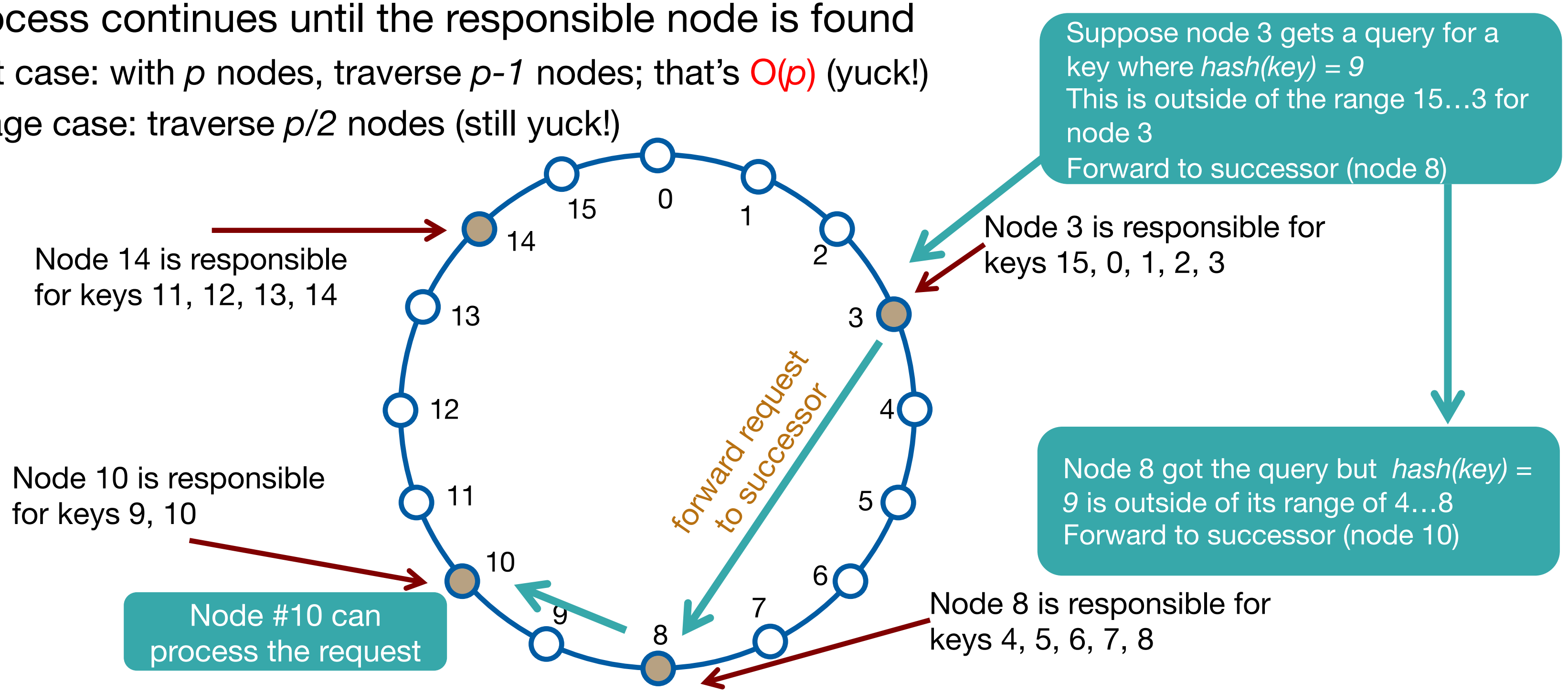
- Example: *n=16;* system with 4 nodes
- *(key, value)* data is stored at a **successor**
  - a node whose value is ≥ hash(key)


No nodes at these empty positions
Node 14 is responsible for keys 11, 12, 13, 14
Node 3 is responsible for keys 15, 0, 1, 2, 3
Node 10 is responsible for keys 9, 10
Node 8 is responsible for keys 4, 5, 6, 7, 8
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

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 *hash(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!)

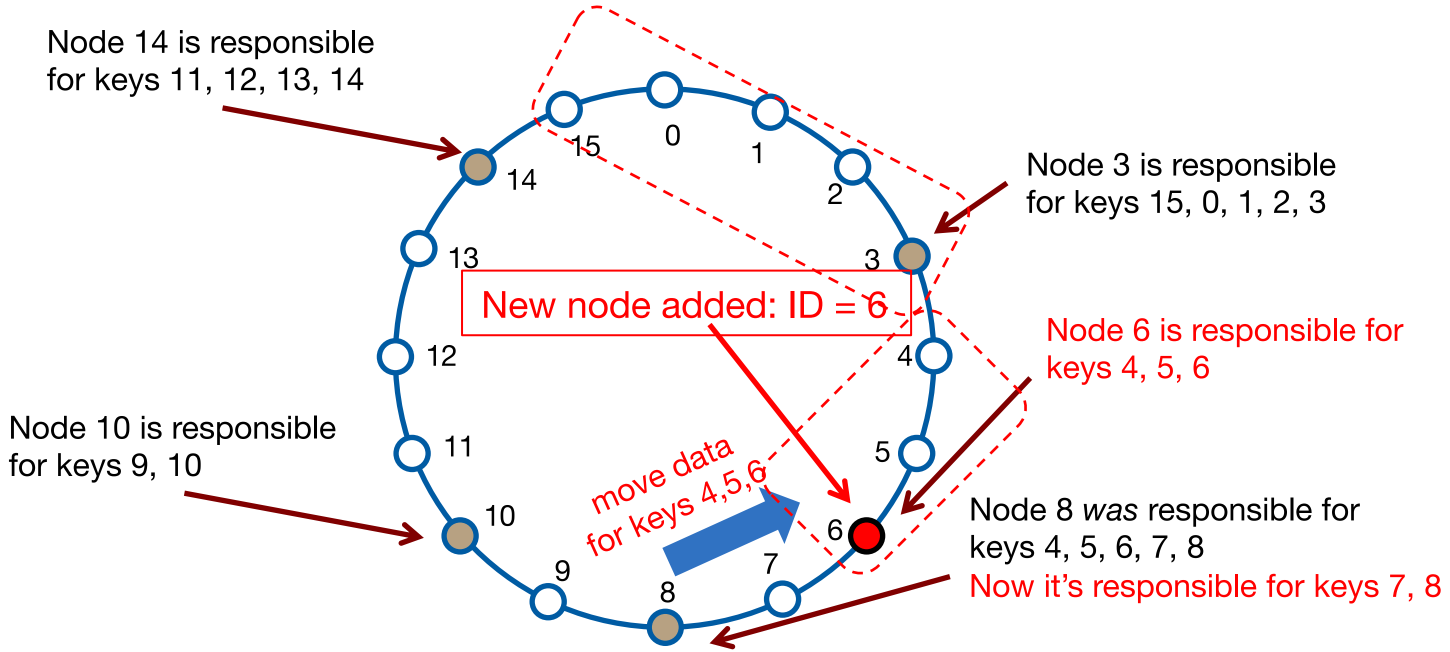

Suppose node 3 gets a query for a key where *hash(key)* = *9*
This is outside of the range 15…3 for node 3
Forward to successor (node 8)
Node 3 is responsible for keys 15, 0, 1, 2, 3
Node 14 is responsible for keys 11, 12, 13, 14
Node 10 is responsible for keys 9, 10
Node 8 got the query but *hash(key)* = *9* is outside of its range of 4…8
Forward to successor (node 10)
Node 8 is responsible for keys 4, 5, 6, 7, 8
Node #10 can process the request
forward request to successor
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Let’s figure out three more things

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

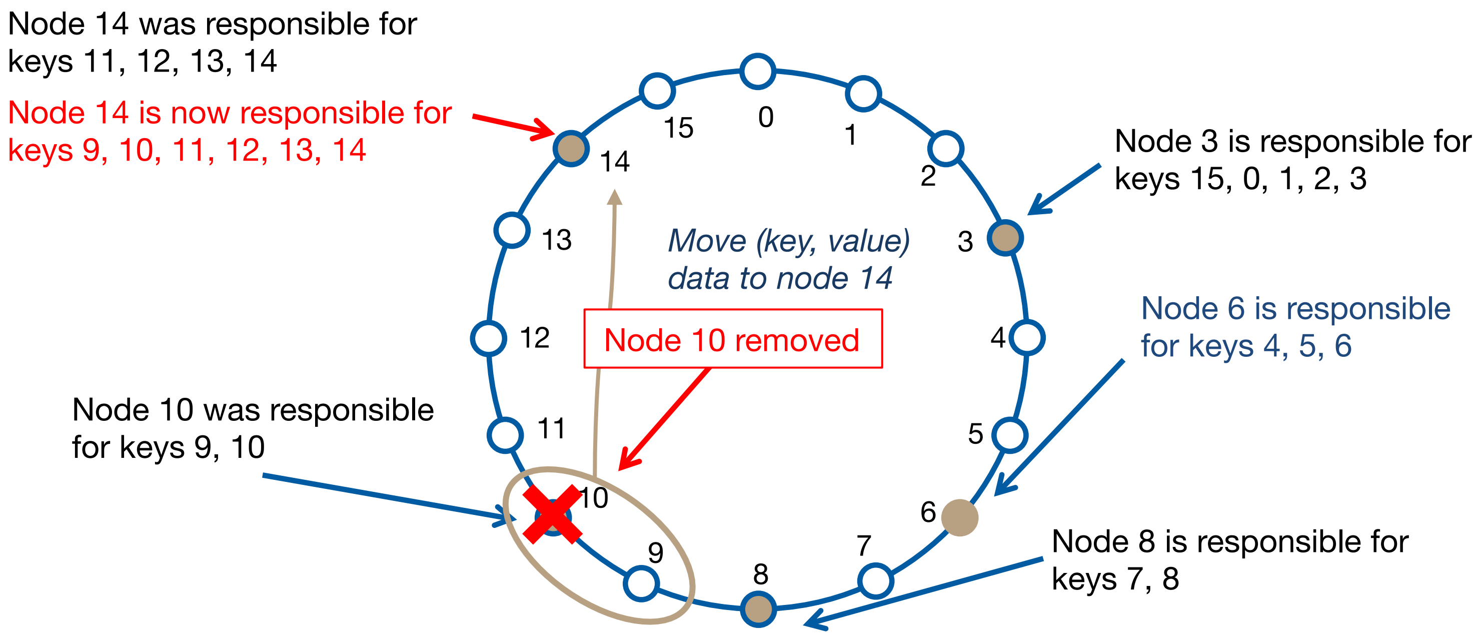
Adding a node

- 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


Node 14 is responsible for keys 11, 12, 13, 14
Node 3 is responsible for keys 15, 0, 1, 2, 3
New node added: ID = 6
Node 6 is responsible for keys 4, 5, 6
Node 10 is responsible for keys 9, 10
move data for keys 4,5,6
Node 8 *was* responsible for keys 4, 5, 6, 7, 8
Now it's responsible for keys 7, 8
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Removing a node

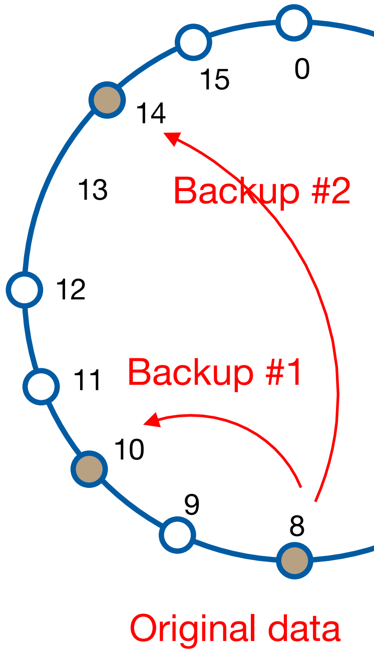
- Keys are reassigned to the node's successor
- Data for those *(key, value)* pairs must be moved to the successor



Node 14 was responsible for keys 11, 12, 13, 14
Node 14 is now responsible for keys 9, 10, 11, 12, 13, 14
15
0
1
2
14
Node 3 is responsible for keys 15, 0, 1, 2, 3
13
Move (key, value) data to node 14
3
12
Node 10 removed
4
Node 6 is responsible for keys 4, 5, 6
11
5
Node 10 was responsible for keys 9, 10
10
6
9
7
8
Node 8 is responsible for keys 7, 8

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



Finger tables

We're not thrilled about *O(N)* lookup

Finger table = partial list of nodes, progressively more distant

At each node, ith 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)*

Some uses of DHTs

- General purpose distributed object store: names, passwords, user profiles, …
- Coral CDN, Tox IM, Freenet anonymous sharing, Scribe notification
- **Amazon** – shopping carts, session info, product catalog, …
- **BitTorrent** – decebtrakuzed 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

Distributed Lookup

Amazon Dynamo

Amazon Dynamo

- Not exposed as a web service
  - Used to power parts of Amazon Web Services and internal services
  - Highly available, key-value storage system

- In an infrastructure with millions of components, something is always failing!
  - Failure is the normal case

- A lot of services within Amazon only need primary-key access to data
  - Best seller lists, shopping carts, preferences, session management, sales rank, product catalog
  - No need for complex querying or management offered by an RDBMS
    - Full relational database is overkill: limits scale and availability
    - Still not efficient to scale or load balance RDBMS on a large scale

Core Assumptions & Design Decisions

- Two operations: **get** and **put**
  - Binary objects (data) identified by a unique key
  - Objects tend to be small (typically < 1MB)
- Strongly consistent distributed databases provide poor availability
  - Use weaker consistency for higher availability
- Apps should be able to configure Dynamo for desired latency & throughput
  - Balance performance, cost, availability, and durability guarantees
- At least 99.9% of read/write operations must be performed within a few hundred milliseconds:
  - Avoid routing requests through multiple nodes
- Dynamo can be thought of as a **zero-hop DHT**

Core Assumptions & Design Decisions

- Incremental scalability
  - System should be able to grow by adding a storage host (node) at a time
- Symmetry
  - Every node has the same set of responsibilities
- Decentralization
  - Favor decentralized techniques over central coordinators
- Heterogeneity (mix of slow and fast systems)
  - Workload partitioning should be proportional to capabilities of servers

Consistency & Availability

Strong consistency & high availability cannot be achieved simultaneously

- Optimistic replication techniques – *eventually consistent* model
  - Propagate changes to replicas in the background – they will *eventually* be updated
  - This can lead to conflicting changes that have to be detected & resolved

- When do you resolve conflicts?
  - **During writes**: the traditional approach
    - Reject write if cannot reach all (or majority) of replicas – ***but don't deal with conflicts***
  - **Resolve conflicts during reads**: Dynamo approach
    - Design for an "always writable" data store - highly available
    - read/write operations can continue even during network partitions
    - Rejecting customer updates won't be a good experience
      - Example: a customer should always be able to add or remove items in a shopping cart

Consistency & Availability

Who resolves conflicts?

Choices: the *data store system* or the *application*?

- Data store
  - Application-unaware, so choices limited
  - Simple policy, such as "last write wins”
- Application
  - App is aware of the meaning of the data
  - Can do application-aware conflict resolution
  - **E.g., merge shopping cart versions to get a unified shopping cart.**

Fall back on "*last write wins*" if app doesn't want to bother

Reads & Writes

Two operations:

get(key) returns

1. object or list of objects with conflicting versions
2. context (resultant version per object)

put(key, context, value)

- stores replicas
- *context*: ignored by the application but includes the version of the object
- key is hashed with MD5 to create a 128-bit identifier that is used to determine the storage nodes that serve the key:

  hash(key) identifies node

Partitioning the data

- Break up the database into chunks distributed over all nodes
  - Key to scalability

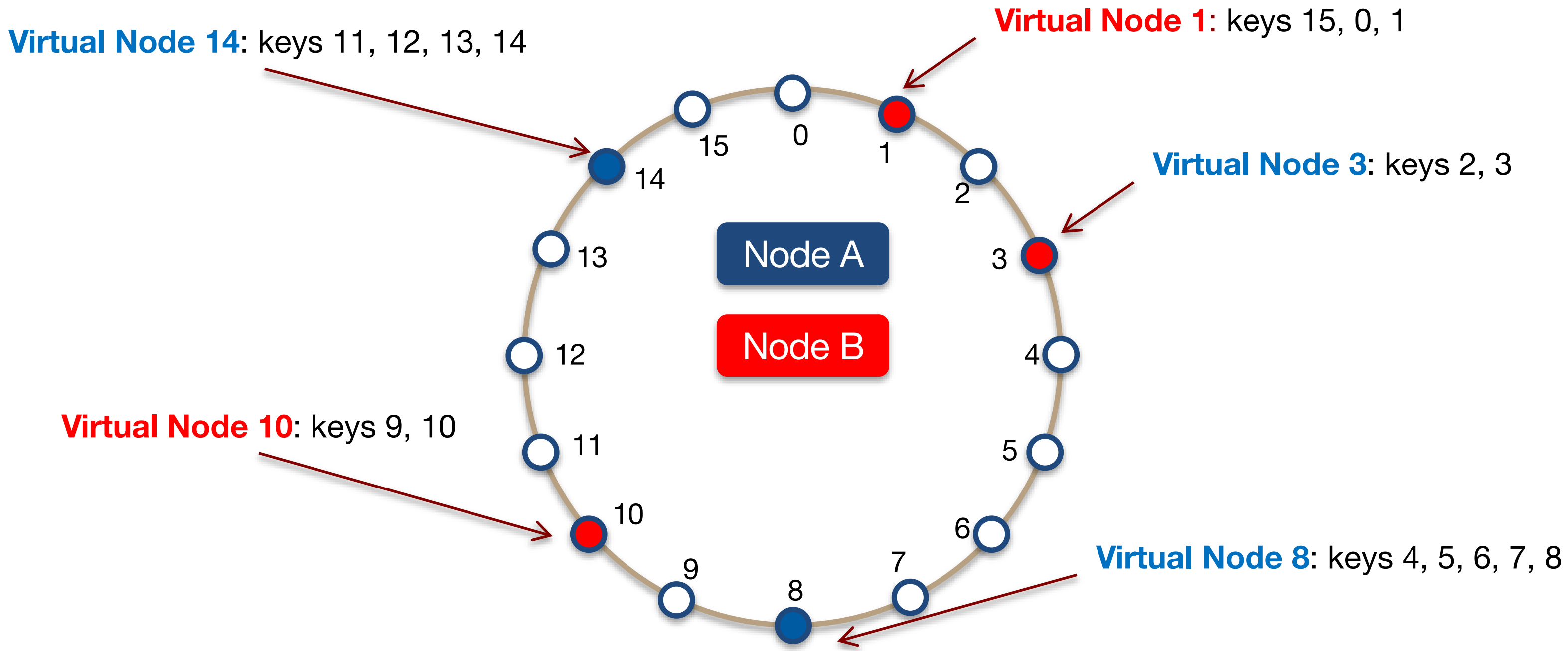
- Relies on consistent hashing
  - On average, *K*/*n* keys need to be remapped, *K* = # keys, *n* = # slots

- Logical ring of nodes: just like Chord
  - Each node is assigned a random value in the hash space: position in ring
  - Responsible for all hash values between its value and predecessor's value
  - Hash(key); then walk ring clockwise to find the first node with `position>hash`
  - Adding/removing nodes affects only immediate neighbors

Partitioning: Dynamo virtual nodes

A physical node holds contents of multiple virtual nodes at multiple points in the ring

In this example: 2 physical nodes running 5 virtual nodes



Virtual Node 14: keys 11, 12, 13, 14
Virtual Node 1: keys 15, 0, 1
Virtual Node 3: keys 2, 3
Virtual Node 10: keys 9, 10
Virtual Node 8: keys 4, 5, 6, 7, 8
Node A
Node B
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Partitioning: virtual nodes

Advantage: balanced load distribution

- If a node becomes unavailable, the load is evenly dispersed among available nodes
- If a node is added, it accepts an equivalent amount of load from other available nodes
- # of virtual nodes per system can be based on the capacity of that node
    - Makes it easy to support changing technology and addition of new, faster systems

Replication

- Storing/reading key-value data
  - Key is assigned a coordinator node (via hashing) ⇒ main node

- Replication
  - Data replicated on *N* hosts (*N* is configurable)
  - Coordinator oversees replication
  - Coordinator replicates keys at the *N-1* clockwise successor nodes in the ring

Dynamo Replication

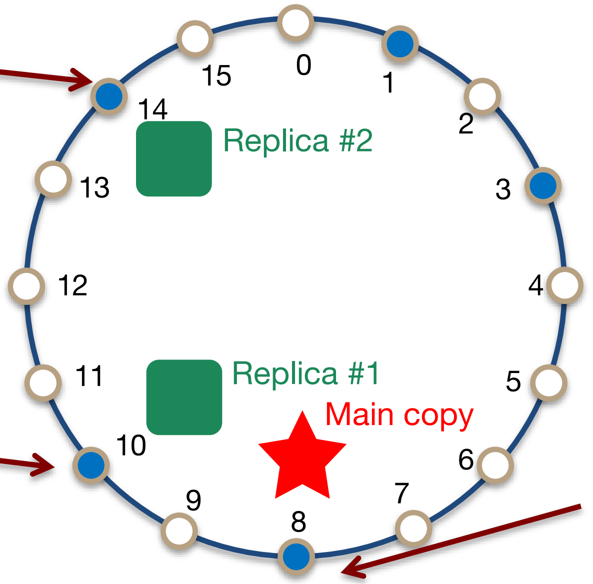
Coordinator replicates keys at the *N-1* clockwise successor nodes in the ring

Example: N=3

Node 14 holds replicas for Nodes 8 and 10

Node 10 holds replicas for Node 3 and 8

Node 8 holds replicas for Nodes 1 and 3



Availability & Consistency

- Configurable values
  - *R*: minimum # of nodes that must participate in a successful read operation
  - *W*: minimum # of nodes that must participate in a successful write operation
- Metadata to remember original destination
  - If a node was unreachable, the data is sent to another node in the ring
  - Metadata sent with the data states the original desired destination
- Data center failure
  - Each object is replicated across multiple data centers

Versioning

- Not all updates may arrive at all replicas
  - Clients may modify or read stale data
- Application-based reconciliation
  - Each modification of data is treated as a new version
- Vector clocks are used for versioning
  - Capture causality between different versions of the same object
  - Vector clock is a set of (node, counter) pairs
  - Returned as a **context** from a `get()` operation and sent via `put()`

Dynamo Storage Nodes

Each node in Dynamo has three components

1. **Request coordination**
   - Node coordinator determined by hash(key)
   - Coordinator executes *get/put* requests on behalf of requesting clients
   - State machine contains all logic for identifying nodes responsible for a key, sending requests, waiting for responses, retries, processing retries, packaging response
   - Each state machine instance handles one request
2. **Membership and failure detection**
3. **Local persistent storage**
   - Different storage engines may be used depending on application needs
     - Berkeley Database (BDB) Transactional Data Store (most popular)
     - BDB Java Edition
     - MySQL (for large objects)
     - In-memory buffer with persistent backing store

The End