CS 677: Big Data

Distributed Hash Tables

Lecture 7

Today's Schedule

- Distributed Lookups
- Distributed Hash Tables
- Chord
- Zero-Hop DHTs, Eventual Consistency
- Replication Strategies
- Hotspots, Heterogeneity, Sybil Attacks

Today’s Schedule

- **Distributed Lookups**
- Distributed Hash Tables
- Chord
- Zero-Hop DHTs, Eventual Consistency
- Replication Strategies
- Hotspots, Heterogeneity, Sybil Attacks

Recap: Distributed Lookups

- We've discussed a few approaches for finding data in our system
- HDFS: The NameNode
  - Or in our DFS, the controller
- Napster: central catalog
  - Implemented as a database
- Gnutella: completely decentralized, flood to peers
- We need some way to map: file => node

Shortcomings

- A central index component means a single point of failure
  - Failover schemes can help
- Scalability is an issue for both approaches
  - Single index: all requests funneled through
  - Flooding: excessive communication
- Security implications
  - Paint a giant target on your central component

An Alternative: Hierarchies

- Spreading global state across multiple nodes helps alleviate these issues
  - No single point of failure, better scalability, etc.
  - Lots of real-world examples
- The downside: this can be difficult!
  - How do we keep state consistent?
  - Do we still keep a "root" node that contains a copy of everything? Why or why not?
  - There is another alternative!

Today's Schedule

- Distributed Lookups
- **Distributed Hash Tables**
- Chord
- Zero-Hop DHTs, Eventual Consistency
- Replication Strategies
- Hotspots, Heterogeneity, Sybil Attacks

Distributed Hash Tables

- Another alternative is **Distributed Hash Tables**
  - **DHTs**
- Decentralized
- Storage and retrieval are handled by the same **deterministic** algorithm
  - Supports `put(k, v)` and `get(k)`
  - Also used to place replicas
- Near-uniform load balancing

DHTs in a Nutshell

- DHTs are just like the hash table data structures we use (and abuse) all the time
  - Except when you `put()` something into the DHT, it's being stored on one of the nodes in the cluster
- We take a **hash algorithm** such as MD5 or SHA-1 and look at its complete **hash space**
  - MD5: 128 bits = 2^{128} unique keys
  - SHA-1: 160 bits = 2^{160} unique keys

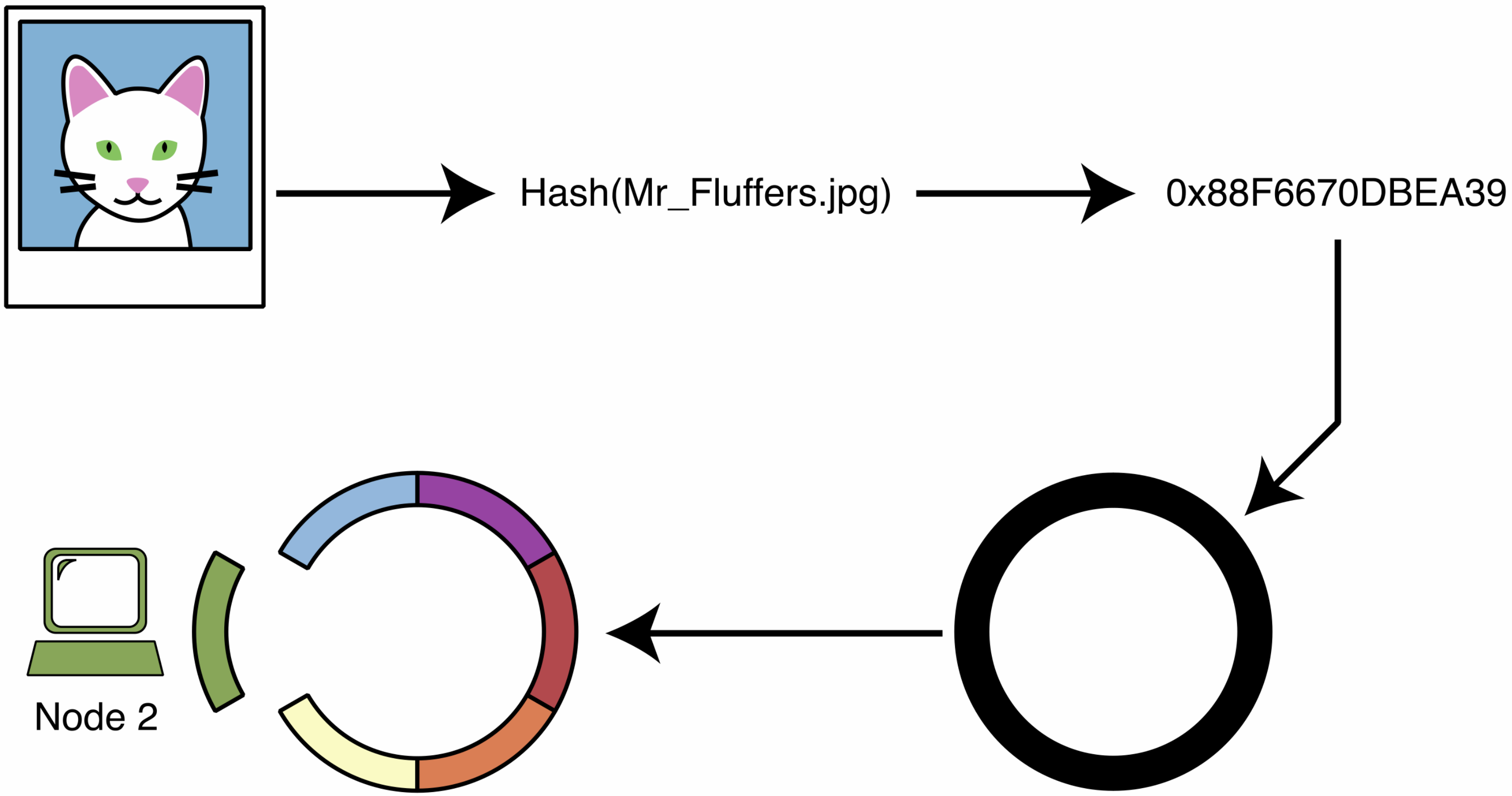
The Hash Space

- We represent our hash algorithm's **hash space** as a circle
  - In a DHT, there isn't really a "start" or "end" of the hash space
- Next, we assign nodes to be responsible for particular portions of the hash space
  - Each file is mapped to the hash space and falls under a single node's purview
  - Creates an overlay network – like our ring topology

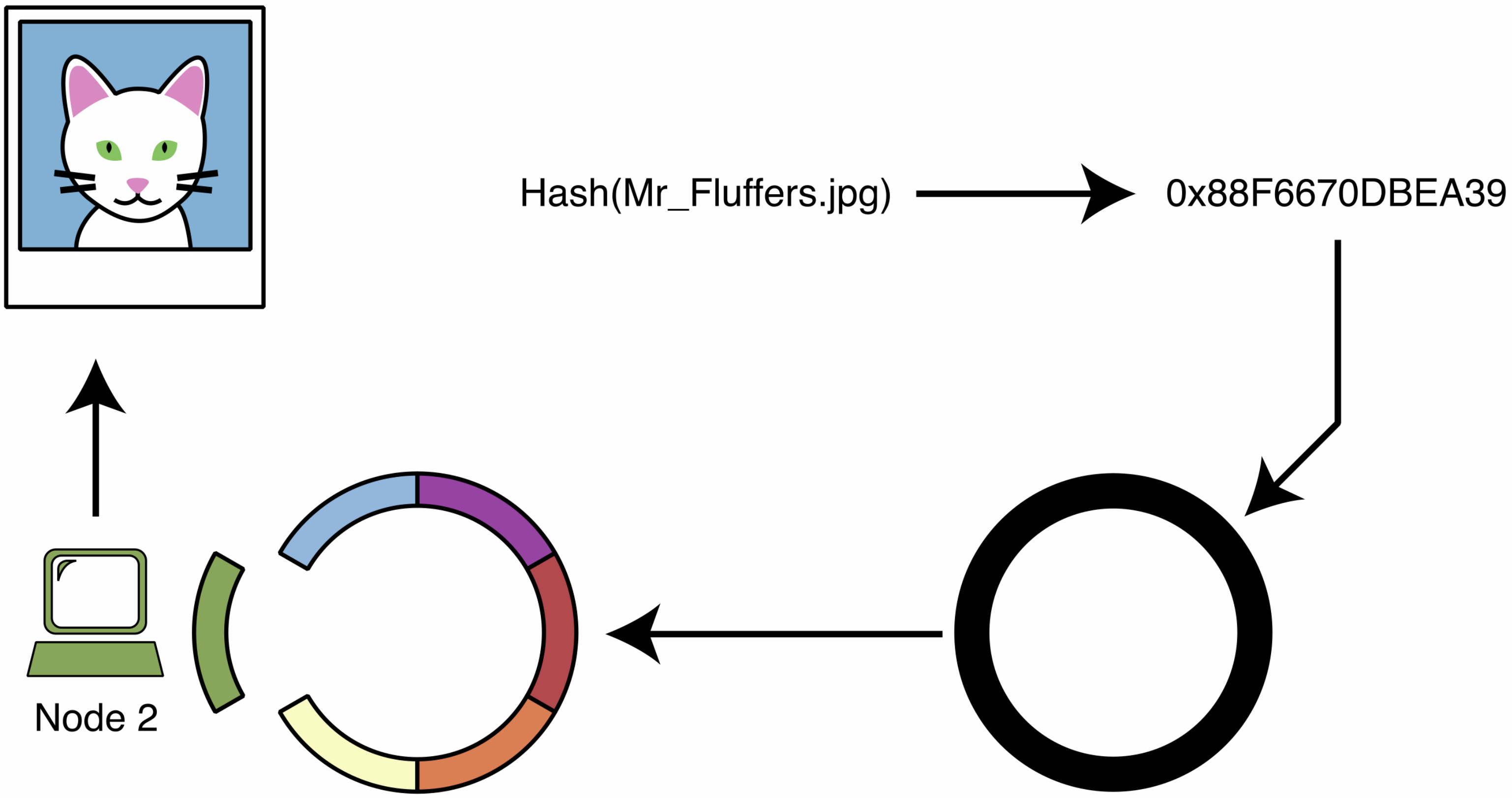
Consistent Hashing

- Breaking up the hash space in this way is a form of **consistent hashing**
- When the hash table is resized (adding or removing a node), generally K/n keys must be remapped:
  - K – number of keys
  - n – number of nodes
- Contrasts with basic hashing schemes, such as using $hash(o)\%n$ to determine file destinations

DHT Overview: Storage



DHT Overview: Retrieval

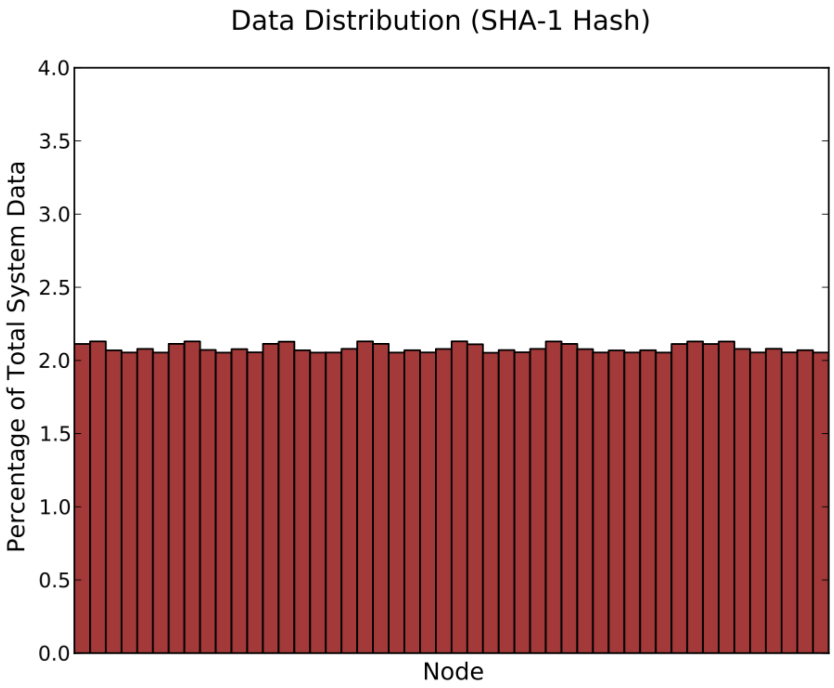


DHTs, The Good and Bad

- Good:
  - Highly scalable, decentralized, no bottlenecks
  - Finding data takes $O(log\ n)$ hops, where n is the number of nodes
  - Uniform load distribution
- Bad:
  - Exact key required for retrieval
  - Queries on values not possible
    - (bad for document-oriented databases)

Data Placement

- In a pure DHT, file placement is basically random
  - Great for keeping things balanced
- Alternatives:
  - Design a hash function that maintains order (user 2 comes after user 1)
  - Use just a portion of the file name / path



Routing Content in a DHT [1/2]

- Chord, Pastry
  - Prefix routing: Routes for delivery of messages based on values of GUIDs to which they are addressed
- CAN
  - Uses distance in a d-dimensional hyperspace into which nodes are placed
- Kademlia
  - Uses XOR of pairs of GUIDs as a metric for distance between nodes

Routing Content in a DHT [2/2]

- Cassandra
  - A variety of hash functions are supported:
  - MD5
  - Order-preserving
  - ...and the initial placement of nodes can be balanced

Basic Routing Strategy

- No matter what algorithm, there are generally two key rules to follow when routing in a DHT:
  1. Each hop through the network gets you a bit closer
     - In other words, ***do not overshoot***
     - Remember, our hash space wraps back around
  2. Routing goes **one way** only
     - Can be clockwise or counter-clockwise, but not both!

Routing Table Terminology

- Each node in a DHT maintains a **routing table** with a limited view of the network
  - Only a small amount of state is maintained
- In some systems the routing table is also called the *finger table*
- Predecessor – previous **active** node in the overlay
- Successor – next **active** node in the overlay

Moving On

Let’s take a look at **one** way to implement a DHT…

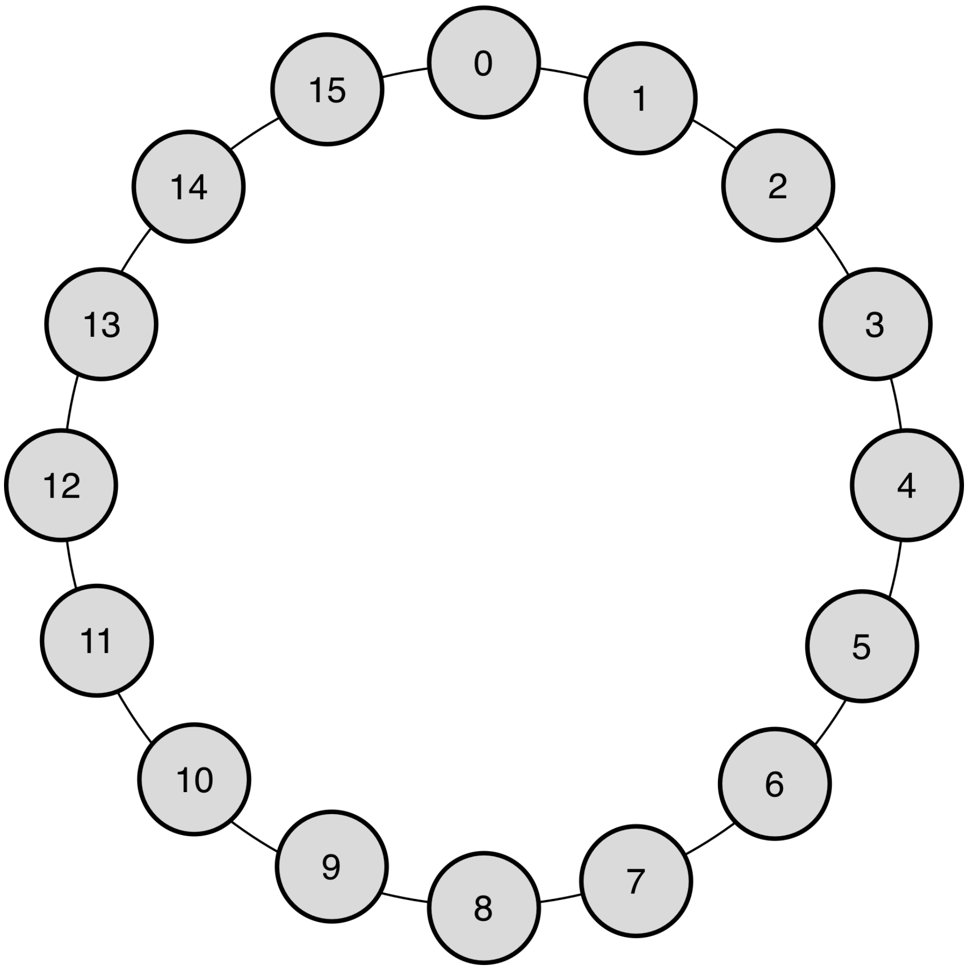
Today's Schedule

- Distributed Lookups
- Distributed Hash Tables
- **Chord**
- Zero-Hop DHTs, Eventual Consistency
- Replication Strategies
- Hotspots, Heterogeneity, Sybil Attacks

Chord

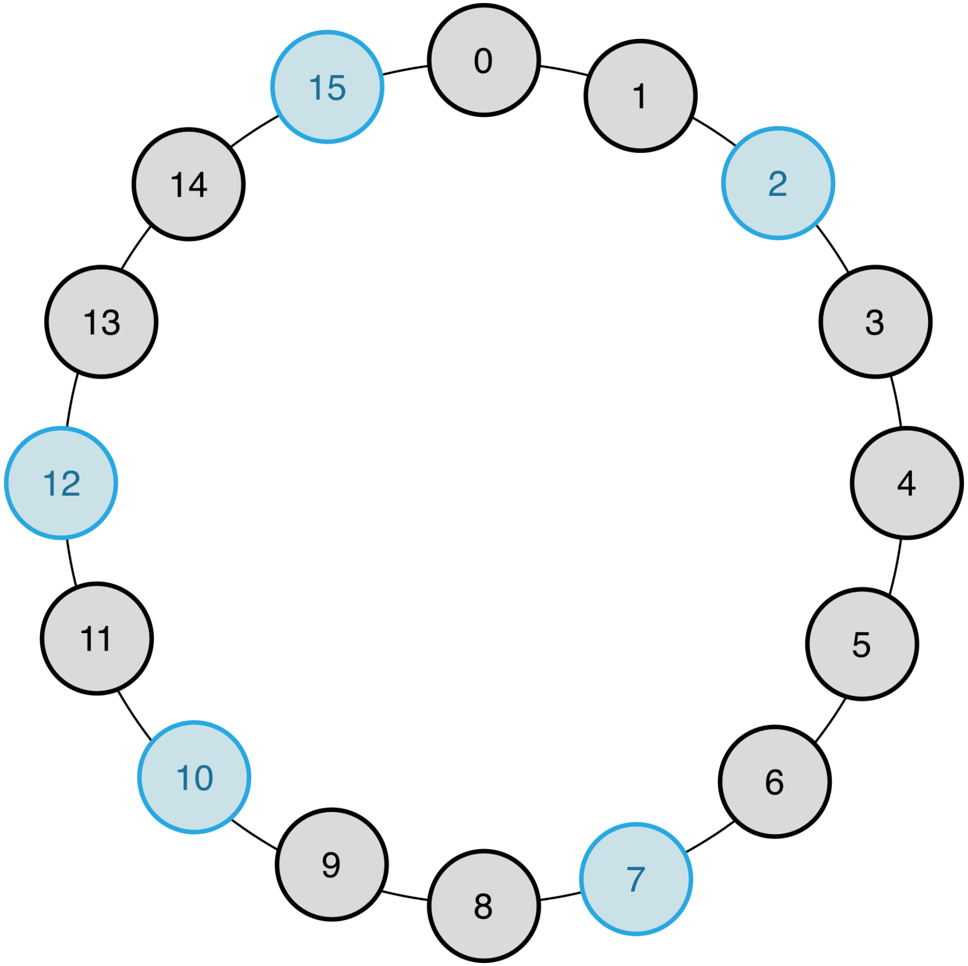
- In **Chord**, both node IDs and file IDs are mapped to the same hash space
- Each node is responsible for an ID range:
  - Its own ID up to its predecessor's ID
- When placing data with key k, locate node n where:
  - $min(id(n) >= k)$
  - (find the smallest numbered node that is greater than or equal to k)
- We also track N – number of nodes in the system

2^4 Network



2^4 Network: Populated

- What keys are node 2 responsible for?
- Node 10?

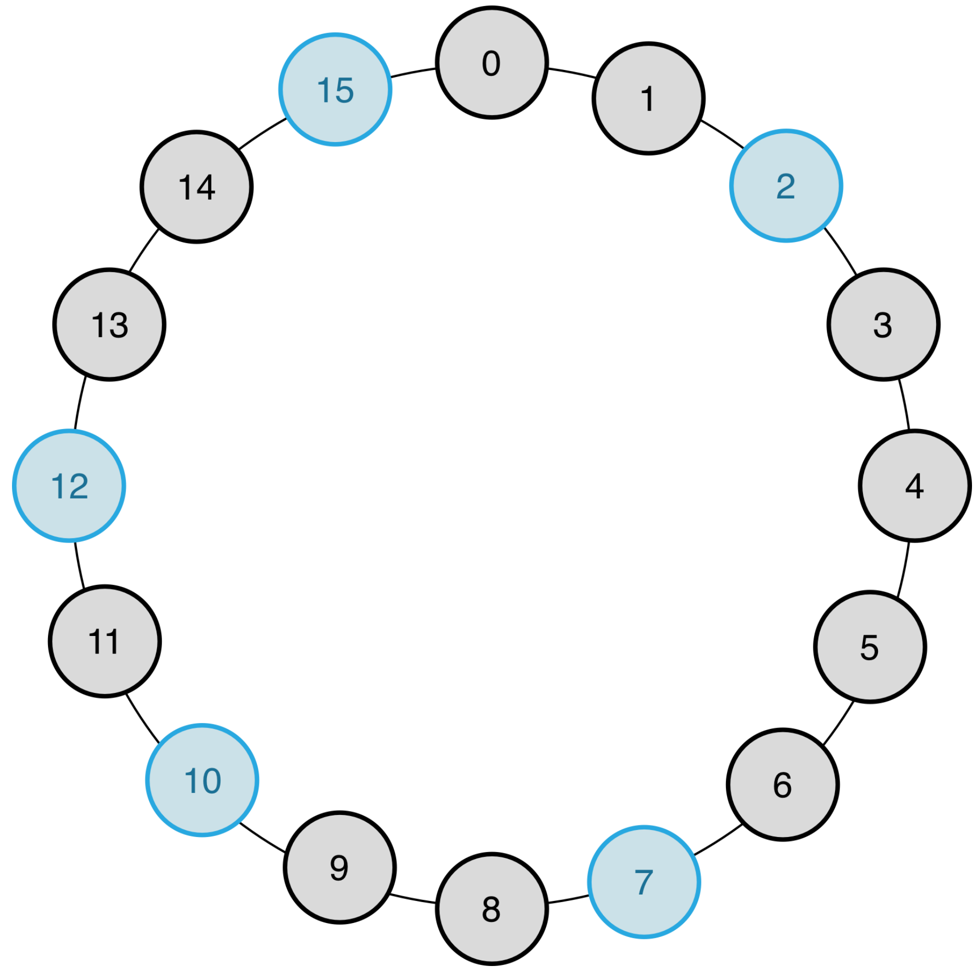


Joining the Network

- Generate an ID using the current timestamp
  - Helps reduce collisions
- An alternative: hash the hostname
  - This can lead to problems. Why?
- Let's say $hash(timestamp) = 5$
  - We need to contact **2 nodes** to join: the successor and the predecessor

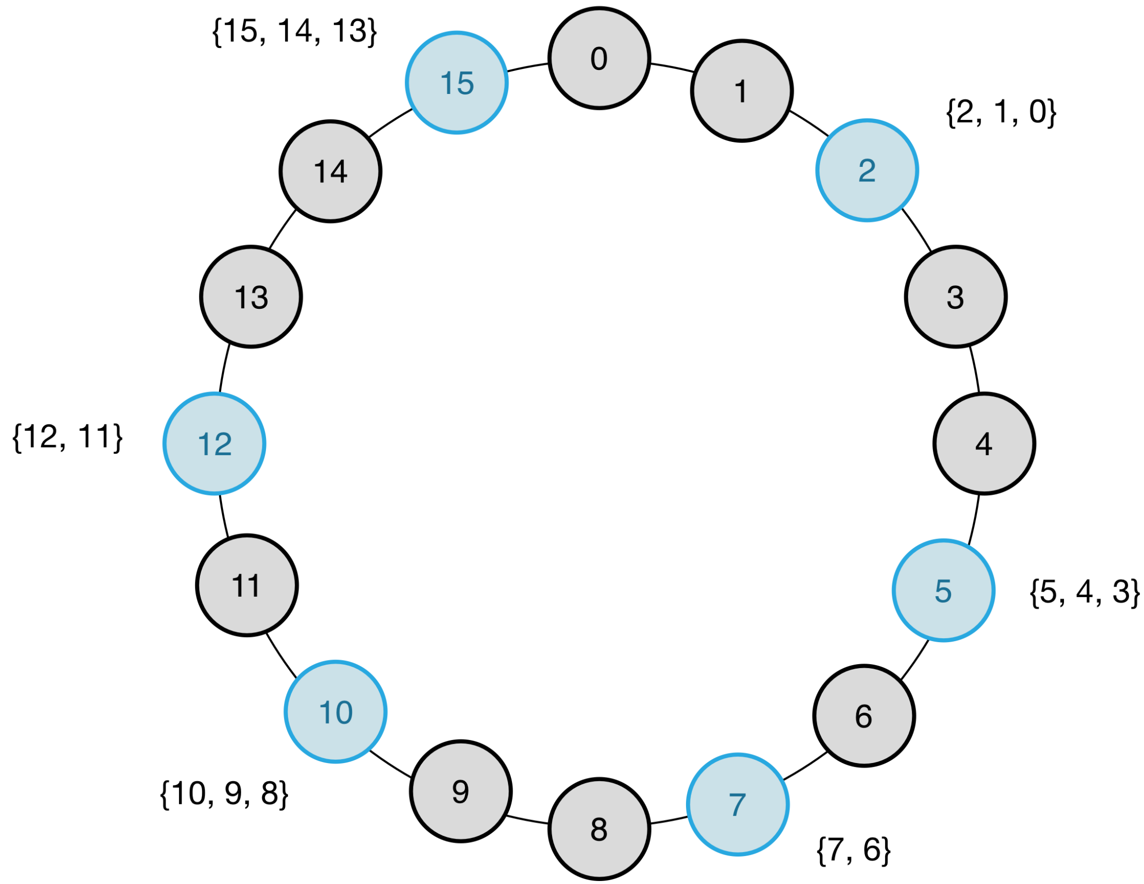
Joining the Network, $ID = 5$

- First, $lookup(our\_id)$
  - $= 7$
- Let node 7 know we're entering the network
- Ask node 7 for its predecessor
  - (2 becomes our predecessor)



Joining the Network

- This approach minimizes communication between nodes
- Node 10, for instance, was not involved at all
- What about routing tables?



Updating Routing Tables

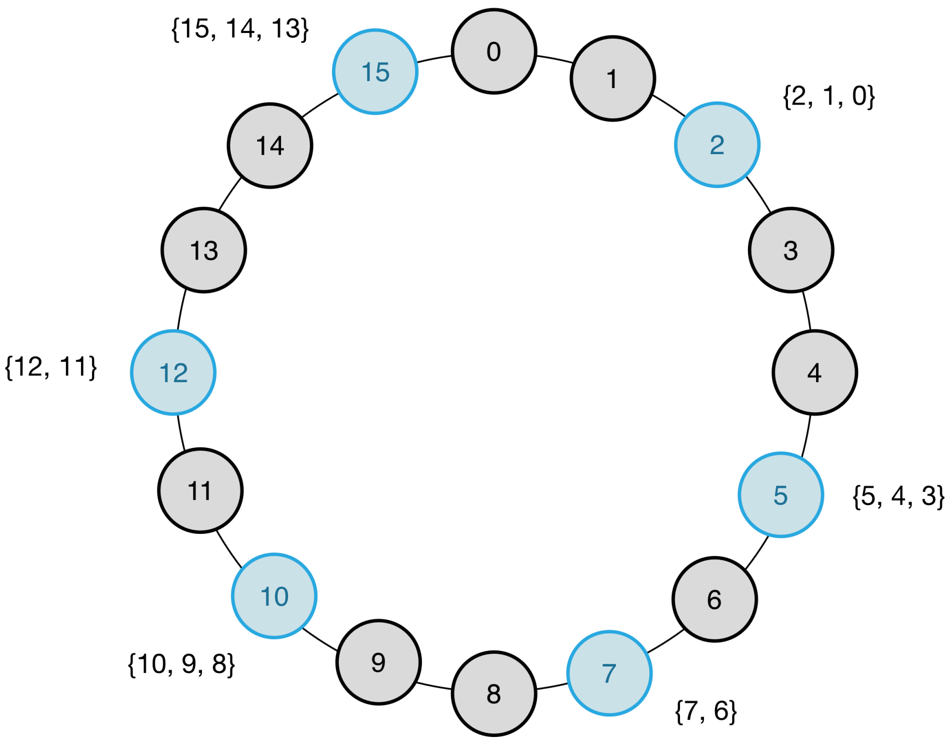
- We do need to keep the routing tables up to date
- However, remember our rule: **no overshooting!**
- In the worst case scenario (no routing information), our DHT becomes a ring topology
  - All next hops are set to your successor
- To find out where data goes, do a lookup. Then update your routing table if you discovered a new node in the process

The Finger Table

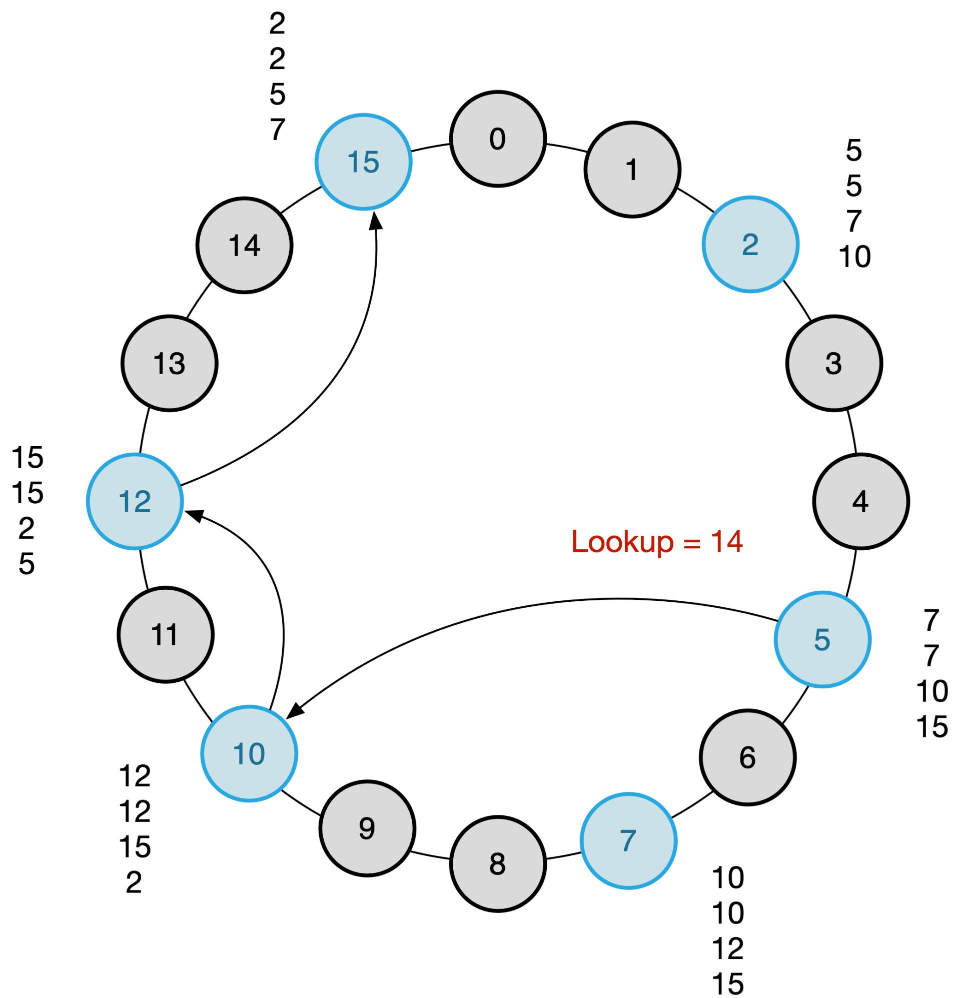
- Each node maintains a **finger table**, which contains the successor, predecessor, and a few nearby nodes
  - Maintaining more than just our direct neighbors is what makes this approach more efficient than a simple ring topology!
- If we have a **4**-bit identifier space (for a total of 2^4 = 16 nodes), each table contains 4 routing entries
- `Route[i]` = lookup(my_node_id + 2^i)

Demo Routing Table: 2^4 Network, $ID = 5$

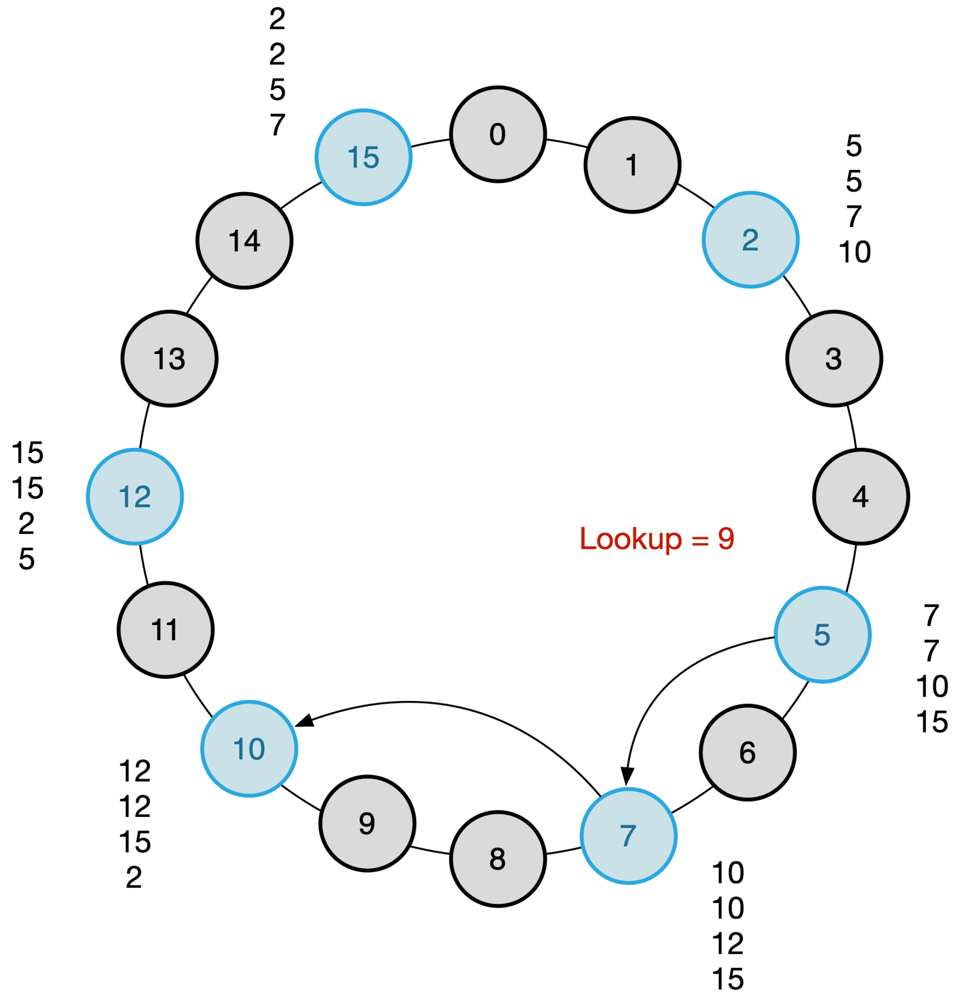
- `Route[i]`
  - $= lookup(ID + 2^i)$
- `Route[0] =`
  - $lookup(5 + 2^0) = 7$
- `Route[1] =`
  - $lookup(5 + 2^1) = 7$
- `Route[2] =`
  - $lookup(5 + 2^2) = 10$
- `Route[3] =`
  - $lookup(5 + 2^3) = 15$



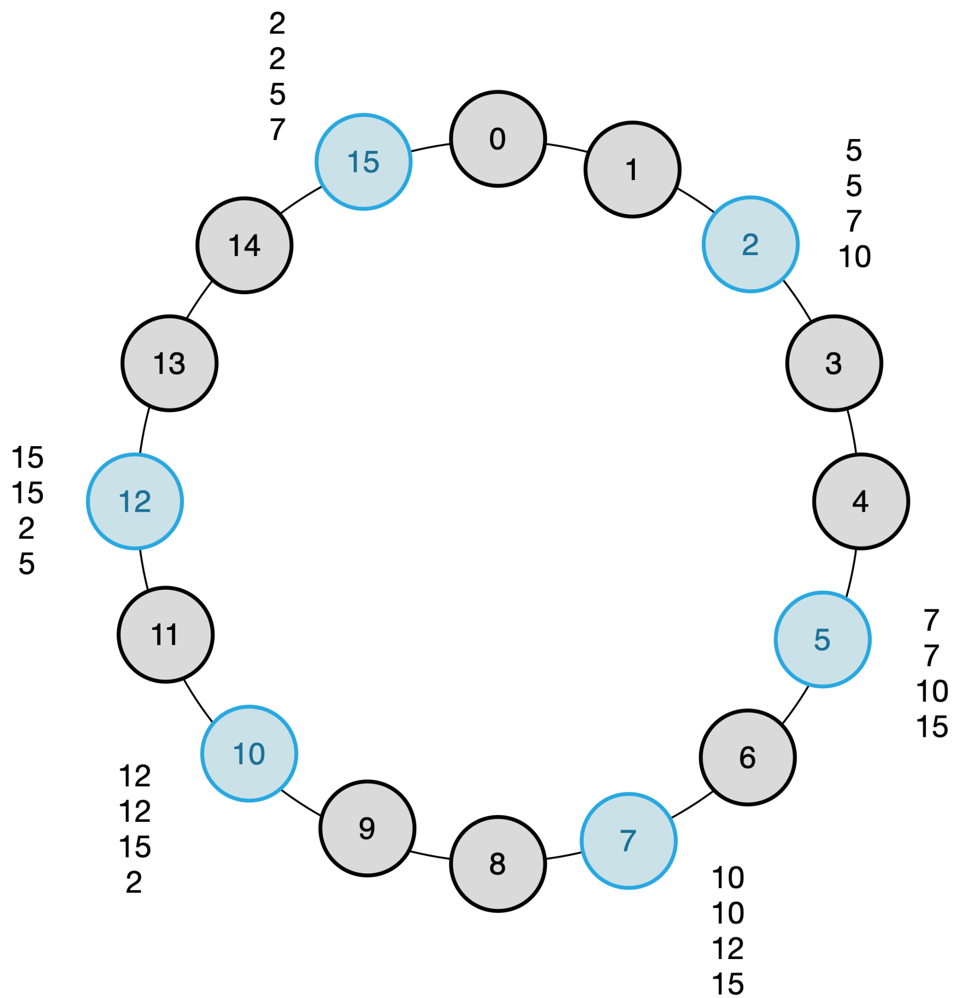
Routing Requests: $ID = 14$



Routing Requests: $ID = 9$



Routing Tables



Today's Schedule

- Distributed Lookups
- Distributed Hash Tables
- Chord
- **Zero-Hop DHTs, Eventual Consistency**
- Replication Strategies
- Hotspots, Heterogeneity, Sybil Attacks

Other Approaches

- Taking multiple hops through the network can incur varying amounts of latency
  - Some applications want to hit more constant latencies
- In an internal system (completely administered by one organization), it's possible to know more about the network layout
- In these cases a **Zero-Hop** DHT works in the same way, except every node has the entire routing table
- Coral CDN – uses a hierarchy of DHTs to load balance between clusters

Zero-Hop DHTs [1/2]

- When nodes enter and leave the network in a controlled manner, **zero-hop** DHTs may be a good fit
- $O(1)$ routing hops rather than $O(log\ n)$
- Every node must maintain an entire copy of the routing table
  - Synchronous updates are not required
  - If an old route is used, just forward the request to the correct node
  - Node down? Try the predecessor

Zero-Hop DHTs [2/2]

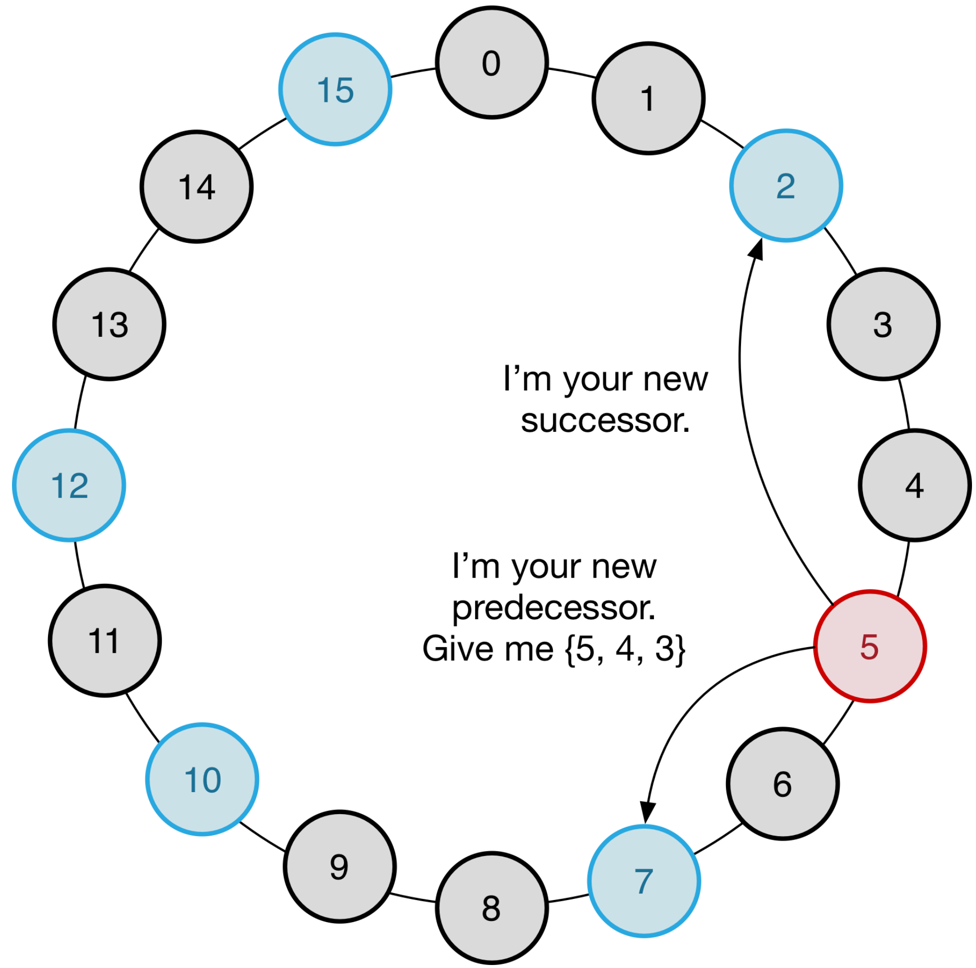
- Zero-Hop DHTs are a great example of finding a compromise in the middle
- Retain many good aspects of regular DHTs, but are also easier to implement
  - May sacrifice some scalability, but in general they target a different use case
- Some implementations: Dynamo, Cassandra, Riak
  - Dynamo: Amazon & SLAs

GlusterFS

- Unlike most of the distributed file systems we've surveyed, GlusterFS is actually ***mountable*** as a Unix FS
  - Backed by Zero-Hop DHT
- Hashes directory ID + file ID to place/locate files
- When we use a regular file system, move operations are common
  - When the usual lookup fails, broadcast to everyone
- Supports **linkfiles**, which are essentially a symlink to redirect lookup requests to another node
  - Great for dealing with file migrations

Eventual Consistency [1/2]

- Joining or leaving the Chord network causes **inconsistency**
- In this example, it may take a bit for node **15** to learn about node **5**
- The system will eventually reach a steady state (usually in ms)



Eventual Consistency [2/2]

- Eventual consistency is a mainstay of distributed systems
- It's easier to accept that things will be inconsistent (sometimes) rather than trying to prevent it
  - Amazon: shopping cart vs billing
- You can often achieve much better performance if you relax consistency
  - But remember to ask yourself: are your customers/clients okay with that?

Today’s Schedule

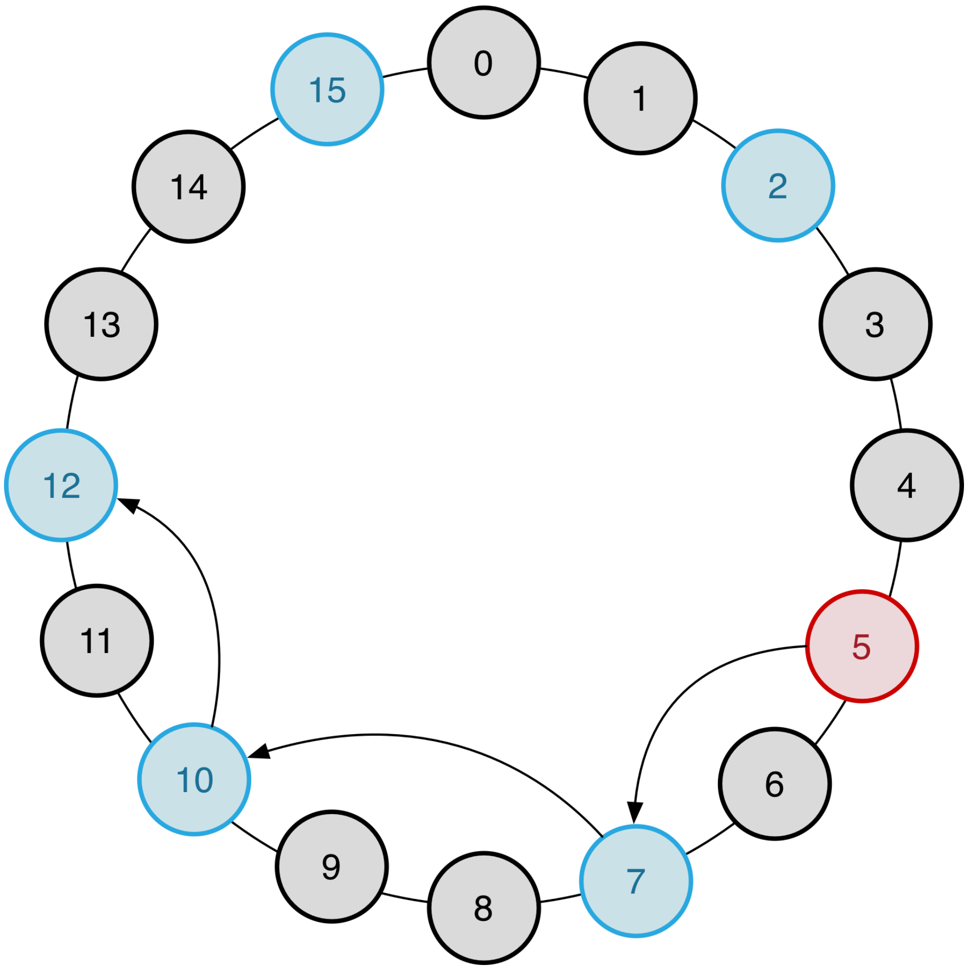
- Distributed Lookups
- Distributed Hash Tables
- Chord
- Zero-Hop DHTs, Eventual Consistency
- **Replication Strategies**
- Hotspots, Heterogeneity, Sybil Attacks

Replication

- We've seen from the HDFS paper that maintaining **3** total copies of each file is our gold standard
  - In some situations, 5 is warranted
  - ...And sometimes having 0 copies is the way to go 😄
- It's always worth thinking about the cost of maintaining these, though
- How do we do replication in DHTs?

Replicate to Successors

- Send a copy to R **successors**
- If Node 5 goes down, Node 7 will take its load
  - Great! Promote replica to primary file
- Doesn't account for query traffic, physical locations, etc.



Query Paths

- Rather than replicating immediately to a certain set of nodes, wait for queries to come in
- Cache the replicas at nodes that forwarded the query
  - Reduces the latency of frequent queries that originate at the same node
  - Let's say my client always contacts the node in San Francisco, which then retrieves from a node in Texas
    - Store a replica in SF
- Better for query performance, not absolute safety

Salting

- For each file, add a **salt**
  - Random data used as an additional input to the hash function
  - SALT_REPLICA1 = “Hi there!”
  - SALT_REPLICA2 = “What what what”
- `put(key + SALT_REPLICA1, value)`
- Now we can deterministically locate the replicas associated with a key

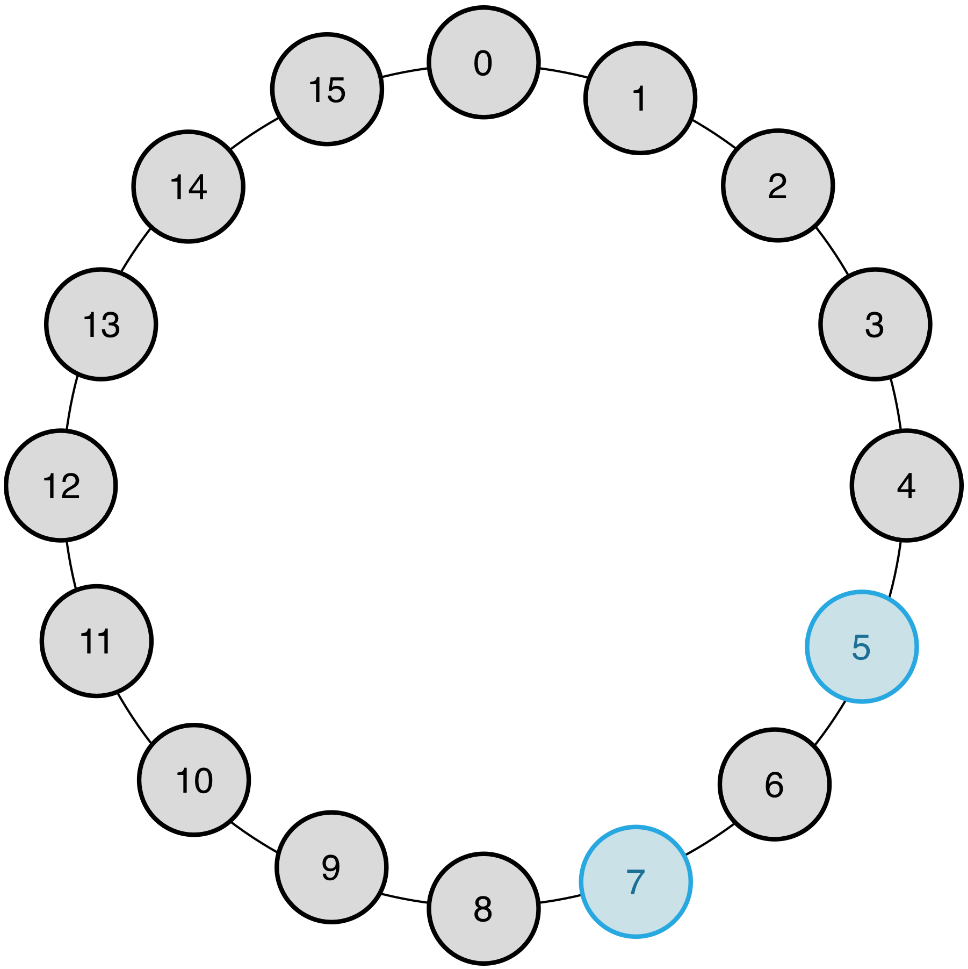
Today's Schedule

- Distributed Lookups
- Distributed Hash Tables
- Chord
- Zero-Hop DHTs, Eventual Consistency
- Replication Strategies
- **Hotspots, Heterogeneity, Sybil Attacks**

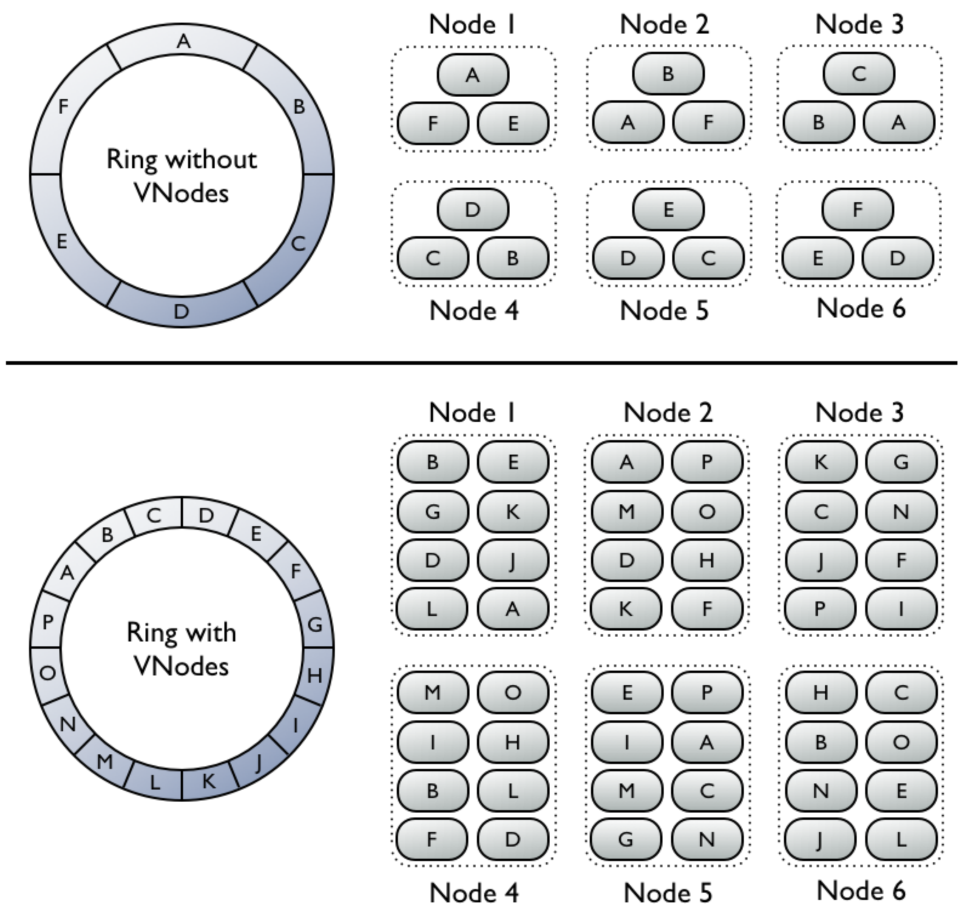
Avoiding Hotspots

- Our cluster may be heterogeneous or have **hotspots** that receive a disproportionate amount of load
- To help fill in the gaps and even out the load, nodes may be required to initially represent several IDs
  - Used frequently in large deployments – hundreds of IDs are assigned to each node
  - Allows variations on the default load level: new node could take on 1.2 nodes' worth of keys

Overloaded, Lonely Node 5



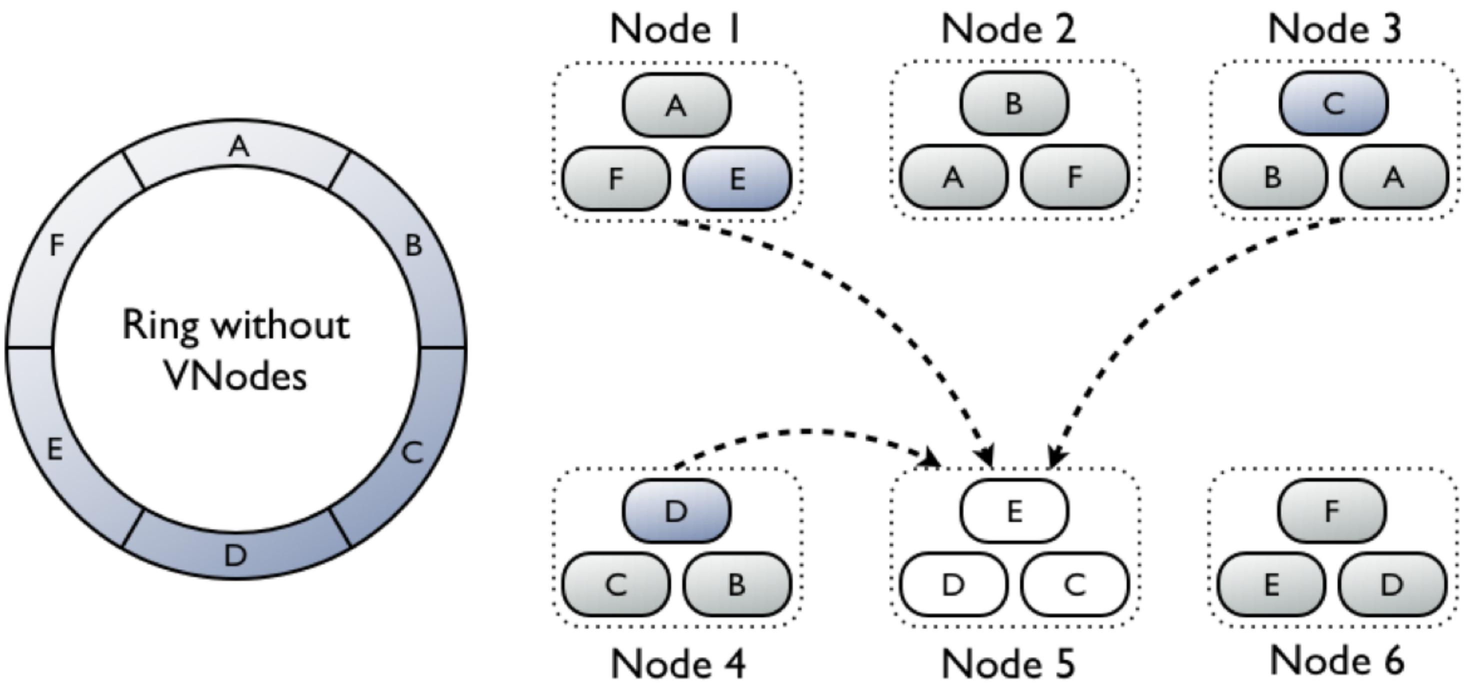
Cassandra: VNodes



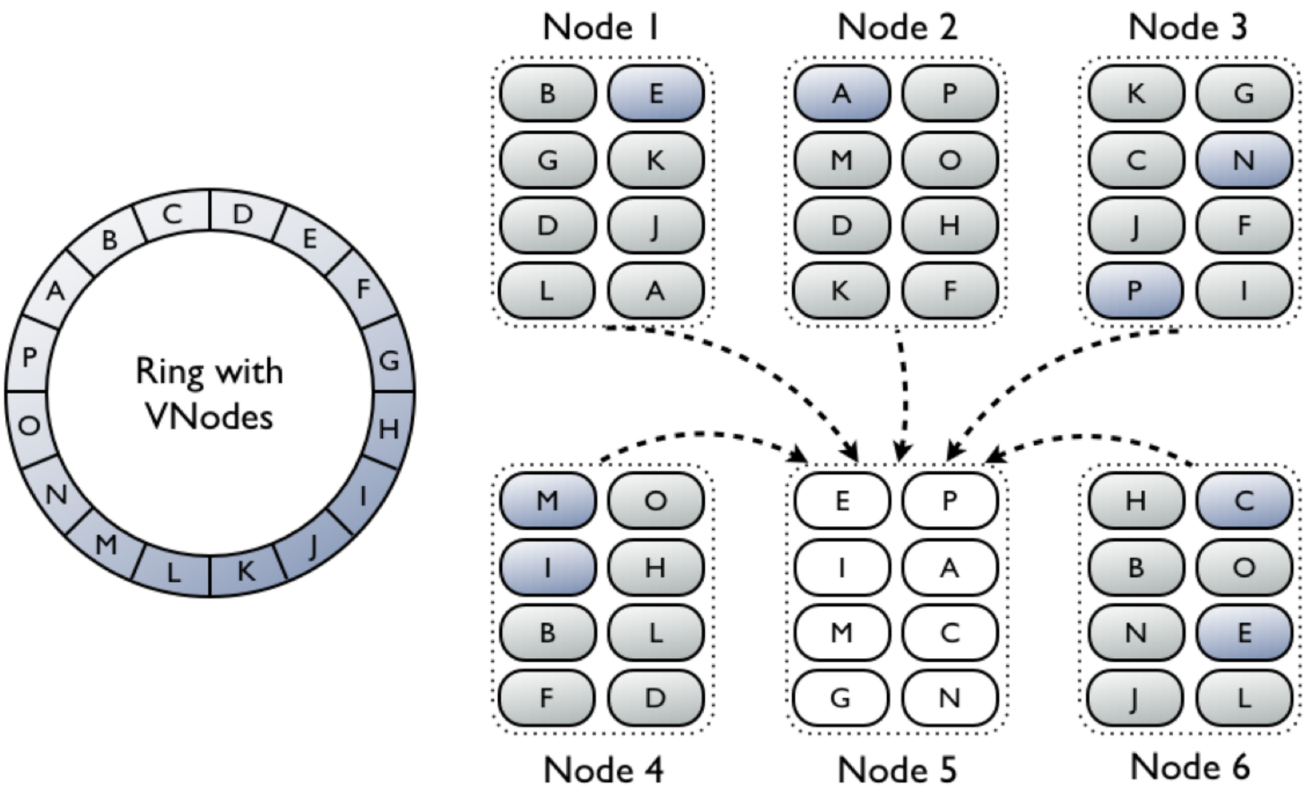
VNodes

- With virtual nodes, each physical host is responsible for many more portions of the overall hash space
- Common approach: randomize the vnode locations
- More coverage means less of a chance that one node gets stuck with too much load
- But wait, wasn't localizing network changes one of the **pros** of using DHTs?
  - Yes. But more coverage ***can*** be a good thing too.

Replacing Node 5 (No VNodes)



Replacing Node 5 (With VNodes)



VNodes: Pros and Cons

- VNode pros:
  - Better load balancing properties
  - Better parallelism when recovering
- VNode cons:
  - Less localized faults: loss of a single node is dispersed across the hash space
  - Many more nodes participating in recovery means less resources for answering queries

Dealing with Heterogeneity

- What we've discussed thus far assumes uniform hardware capabilities
- How can we account for newer, better hardware?
  - Let's not go with the HDFS approach of throwing them in the garbage ☺️
- New nodes can **advertise** as several nodes
  - Maybe the next-gen machines each get assigned two places in the hash ring

Sybil Attacks

- Outside a controlled environment, DHTs are susceptible to **Sybil Attacks**
  - Dissociative identity disorder
- Attacker masquerades as a huge number of false identities
  - Given enough control of the network, data and routing tables can be manipulated
- Prevention: central login service, reverse lookup, vouching for other nodes