CS 686: Special Topics in Big Data

Distributed Hash Tables

Lecture 8

Today's Agenda

- Recap: Distributed Lookups
- Distributed Hash Tables
- Chord
- DHT Issues and Attacks

Today's Agenda

- **Recap: Distributed Lookups**
- Distributed Hash Tables
- Chord
- DHT Issues and 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

A Better Approach: 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 Agenda

- Recap: Distributed Lookups
- **Distributed Hash Tables**
- Chord
- DHT Issues and 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

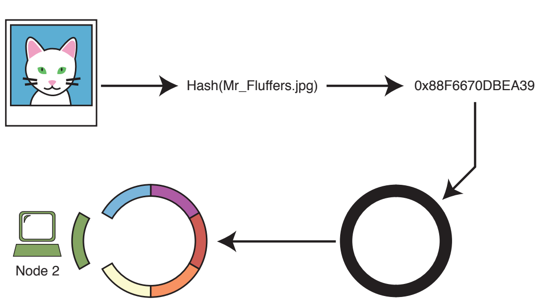
The Hash Space

- 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

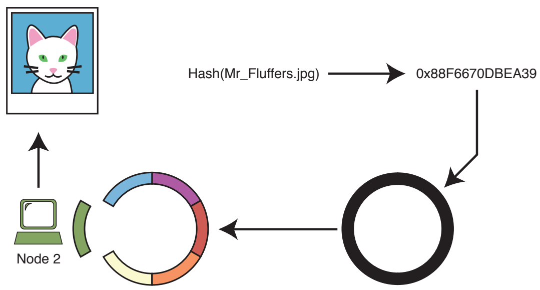
Consistent Hashing

- 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) mod n to determine file destinations

DHT Overview: Storage



DHT Overview: Retrieval



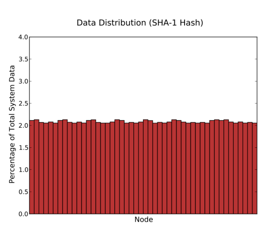
DHTs: Strengths vs. Weaknesses

- Highly scalable
  - Finding data takes O(log *n*) hops, where *n* = # of nodes
- Uniform load distribution
- Decentralized
  - No bottlenecks
  - Any node can handle incoming requests

- Dispersion: storing an entire directory of files
- Exact key required for retrieval
- Queries on values not possible
  - Bad for document-oriented databases

Data Placement

- In a pure DHT, file placement is more or less 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
- Galileo
  - Geohashes (spatial proximity)

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…

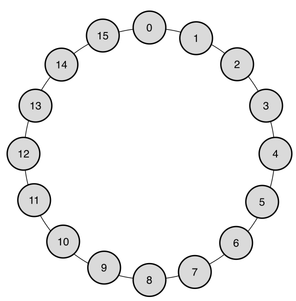
Today's Agenda

- Recap: Distributed Lookups
- Distributed Hash Tables
- **Chord**
- DHT Issues and Attacks

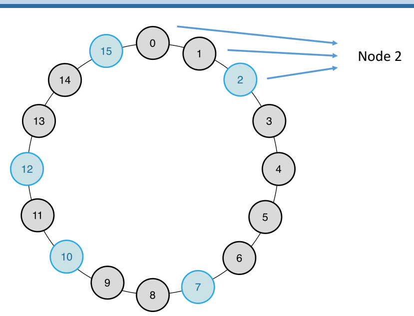
Chord

- 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)
- We also track **N** – number of nodes in the system

2^4 Network



2^4 Network: Populated

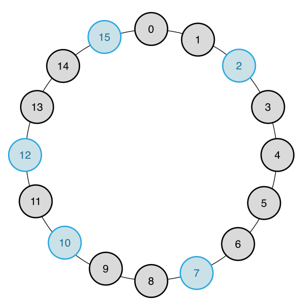


Joining the Network

- 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 do this: the successor and the predecessor

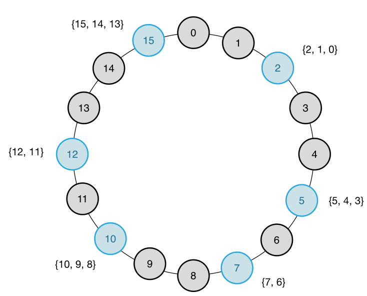
Joining the Network, ID = 5

- First, get(our_id)
  - Returns node **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, hasn't a care in the world
- What about routing tables?



Updating Routing Tables

- We do need to keep the routing tables (**finger tables**) up to date
- However, remember our rule: **no overshooting**!
- In the worst case scenario (no routing information), our DHT becomes a ring topology
- 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] = successor(node_id + 2^i)

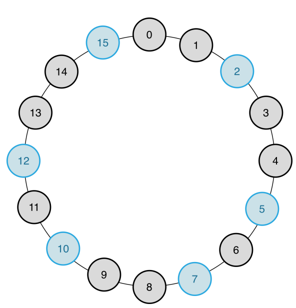
Demo Finger Table, 2^4 Network

Route[0] =
successor(5 + 2^0) = 7

Route[1] =
successor(5 + 2^1) = 7

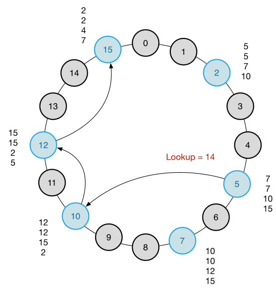
Route[2] =
successor(5 + 2^2) = 10

Route[3] =
successor(5 + 2^3) = 15



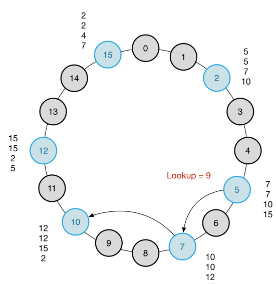
*See why we shouldn't overshoot?

Routing Requests – ID 14



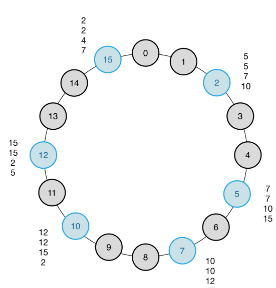
9/11/17 CS 686: Big Data 31

Routing Requests – ID 9



9/11/17 CS 686: Big Data 32

Routing Requests



9/11/17 CS 686: Big Data 33

Today's Agenda

- Recap: Distributed Lookups
- Distributed Hash Tables
- Chord
- **DHT Issues and Attacks**

Other Approaches (1/2)

- 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

Other Approaches (2/2)

- It's also possible to build a hierarchy of DHTs
- Coral CDN – used a hierarchy to load balance between clusters
- Galileo – allows the use of two hash functions, one for a **group** and another for the physical node

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

9/11/17 CS 686: Big Data 37

Avoiding Hotspots

- Along the same lines, we can run into hotspots or imbalances if our network is too **spaced out**
- 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
  - Makes dealing with heterogeneity easier as well
    - New node could take on 1.2 nodes' worth of keys

9/11/17 CS 686: Big Data 38

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

9/11/17 CS 686: Big Data 39