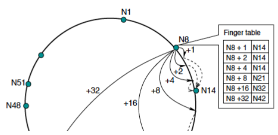


Distributed Hash Tables

Mike Freedman

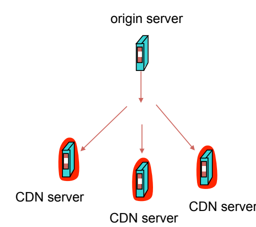
COS 461: Computer Networks

Lectures: MW 10-10:50am in Architecture N101

http://www.cs.princeton.edu/courses/archive/spr13/cos461/

Scalable algorithms for discovery

- If many nodes are available to cache, which one should file be assigned to?
- If content is cached in some node, how can we discover where it is located, avoiding centralized directory or all-to-all communication?



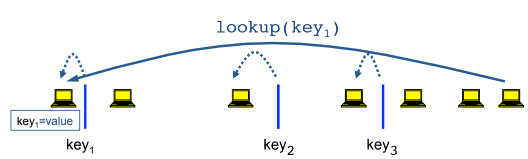
Akamai CDN: hashing to responsibility within cluster

Today: What if you don't know complete set of nodes?

Partitioning Problem

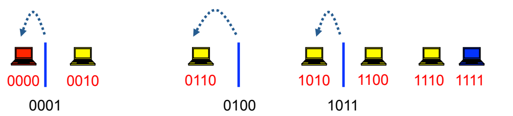
- Consider problem of data partition:
  - Given document X, choose one of k servers to use
- Suppose we use modulo hashing
  - Number servers 1..k
  - Place X on server *i = (X mod k)*
    - Problem? Data may not be uniformly distributed
  - Place X on server *i = hash (X) mod k*
    - Problem? What happens if a server fails or joins (k → k±1)?
    - Problem? What is different clients has different estimate of k?
    - Answer: All entries get remapped to new nodes!

Consistent Hashing



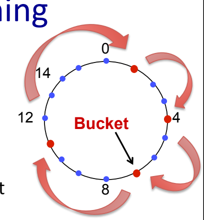
- Consistent hashing partitions key-space among nodes
- Contact appropriate node to lookup/store key
  - Blue node determines red node is responsible for key_1
  - Blue node sends lookup or insert to red node

Consistent Hashing



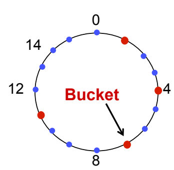
- Partitioning key-space among nodes
  - Nodes choose random identifiers: e.g., hash(IP)
  - Keys randomly distributed in ID-space: e.g., hash(URL)
  - Keys assigned to node "nearest" in ID-space
  - Spreads ownership of keys evenly across nodes

Consistent Hashing



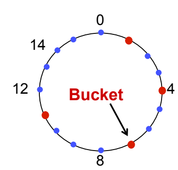
- Construction
  - Assign *n* hash buckets to random points on mod 2^k circle; hash key size = *k*
  - Map object to random position on circle
  - Hash of object = closest clockwise bucket
    - *successor* (key) → bucket
- Desired features
  - Balanced: No bucket has disproportionate number of objects
  - Smoothness: Addition/removal of bucket does not cause movement among existing buckets (only immediate buckets)

Consistent hashing and failures



- Consider network of n nodes
- If each node has 1 bucket
  - Owns $1/n^{th}$ of keyspace *in expectation*
  - Says nothing of request load per bucket
- If a node fails:
  (A) Nobody owns keyspace (B) Keyspace assigned to random node
  (C) Successor owns keyspaces (D) Predecessor owns keyspace
- After a node fails:
  (A) Load is equally balanced over all nodes
  (B) Some node has disproportional load compared to others

Consistent hashing and failures

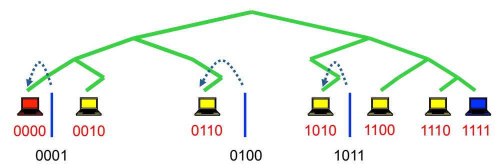


- Consider network of n nodes
- If each node has 1 bucket
  - Owns $1/n^{th}$ of keyspace *in expectation*
  - Says nothing of request load per bucket
- If a node fails:
  - Its *successor* takes over bucket
  - Achieves smoothness goal: Only localized shift, not O(n)
  - But now successor owns 2 buckets: keyspace of size *2/n*
- Instead, if each node maintains *v* random nodeIDs, not 1
  - "Virtual" nodes spread over ID space, each of size *1 / vn*
  - Upon failure, v successors take over, each now stores *(v+1) / vn*

Consistent hashing vs. DHTs

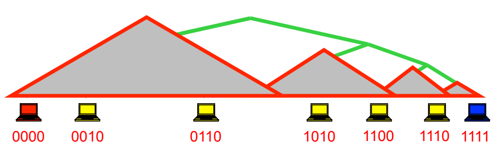
	Consistent Hashing	Distributed Hash Tables
Routing table size	O(n)	O(log n)
Lookup / Routing	O(1)	O(log n)
Join/leave: Routing updates	O(n)	O(log n)
Join/leave: Key Movement	O(1)	O(1)

Distributed Hash Table



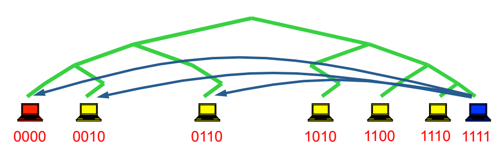
- Nodes' neighbors selected from particular distribution
  - Visual keyspace as a tree in distance from a node

Distributed Hash Table



- Nodes' neighbors selected from particular distribution
  - Visual keyspace as a tree in distance from a node
  - At least one neighbor known per subtree of increasing size / distance from node

Distributed Hash Table

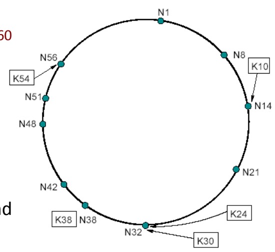


- Nodes' neighbors selected from particular distribution
  - Visual keyspace as a tree in distance from a node
  - At least one neighbor known per subtree of increasing size / distance from node
- Route greedily towards desired key via overlay hops

13

The Chord DHT

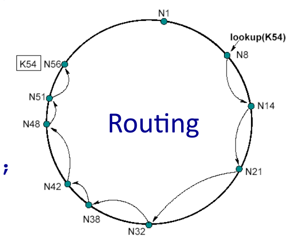
- Chord ring: ID space mod 2^{160}
  - *nodeid = SHA1 (IP address, i)*
    for i=1..v virtual IDs
  - *keyid = SHA1 (name)*
- Routing correctness:
  - Each node knows successor and predecessor on ring
- Routing efficiency:
  - Each node knows O(log n) well-distributed neighbors



14

Basic lookup in Chord

```
lookup (id):
  if ( id > pred.id &&
       id <= my.id )
    return my.id;
  else
    return succ.lookup(id);
```

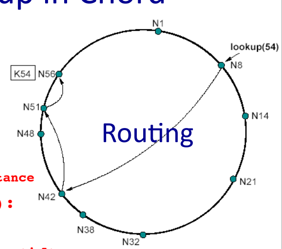


- Route hop by hop via successors
  - O(n) hops to find destination id

15

Efficient lookup in Chord

```
lookup (id):
  if ( id > pred.id &&
       id <= my.id )
    return my.id;
  else
    // fingers() by decreasing distance
    for finger in fingers():
      if id >= finger.id
        return finger.lookup(id);
    return succ.lookup(id);
```

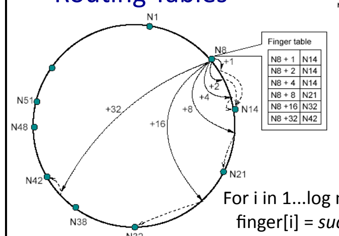


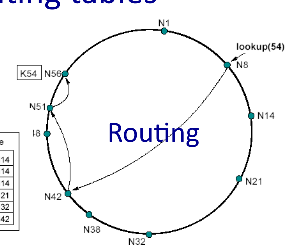
- Route greedily via distant "finger" nodes
  - O(log n) hops to find destination id

16

Building routing tables

Routing Tables

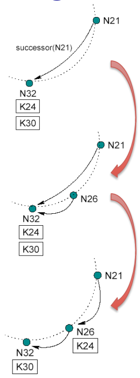




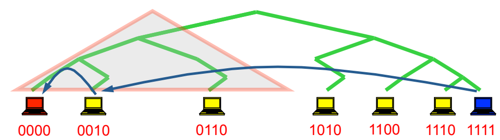
For i in 1...log n:
finger[i] = *successor* ((my.id + 2^i) mod 2^{160})

Joining and managing routing

- Join:
  - Choose nodeid
  - *Lookup (my.id)* to find place on ring
  - During lookup, discover future successor
  - Learn predecessor from successor
  - Update succ and pred that you joined
  - Find fingers by *lookup* ((my.id + 2^i) mod 2^{160})
- Monitor:
  - If doesn't respond for some time, find new
- Leave: Just go, already!
  - (Warn your neighbors if you feel like it)



Performance optimizations



- Routing entries need not be drawn from strict distribution as finger algorithm shown
  - Choose node with lowest latency to you
  - Will still get you ~ ½ closer to destination
- Less flexibility in choice as closer to destination

Consistent hashing vs. DHTs

	Consistent Hashing	Distributed Hash Tables	Distributed Hash Tables
Routing table size	O(n)	O(log n)	O(sqrt(n))
Lookup / Routing	O(1)	O(log n)	O( )
Join/leave: Routing updates	O(n)	O(log n)	O(sqrt(n))
Join/leave: Key Movement	O(1)	O(1)	O(1)

(A) sqrt (N) (B) log N (C) 1

DHT Design Goals

- An "overlay" network with:
  - Flexible mapping of keys to physical nodes
  - Small network diameter
  - Small degree (fanout)
  - Local routing decisions
  - Robustness to churn
  - Routing flexibility
  - Decent locality (low "stretch")
- Different "storage" mechanisms considered:
  - Persistence w/ additional mechanisms for fault recovery
  - Best effort caching and maintenance via soft state

Storage models

- Store *only* on key's immediate successor
  - Churn, routing issues, packet loss make lookup failure more likely
- Store on *k* successors
  - When nodes detect succ/pred fail, re-replicate
  - Use erasure coding: can recover with j-out-of-k "chunks" of file, each chunk smaller than full replica
- Cache along reverse lookup path
  - Provided data is immutable
  - …and performing recursive responses

Summary

- Peer-to-peer systems
  - Unstructured systems (next Monday)
    - Finding hay, performing keyword search
  - Structured systems (DHTs)
    - Finding needles, exact match
- Distributed hash tables
  - Based around consistent hashing with views of O(log n)
  - Chord, Pastry, CAN, Koorde, Kademlia, Tapestry, Viceroy, …
- Lots of systems issues
  - Heterogeneity, storage models, locality, churn management, underlay issues, …
  - DHTs deployed in wild: Vuze (Kademlia) has 1M+ active users