Distributed Hash Tables

15-415 (Fall 2011)

Adapted from a presentation by Jeff Pang in 15-744, Spring 2007

DHTs

- Like it sounds – a distributed hash table
- Put(Key, Value)
- Get(Key) -> Value

Interface vs. Implementation

- Put/Get is an abstract interface
  - Very convenient to program to
  - Doesn't require a "DHT" in today's sense of the world.
  - e.g., Amazon's S^3 storage service
    - /bucket-name/object-id -> data
- We'll mostly focus on the back-end log(n) lookup systems like Chord
  - But researchers have proposed alternate architectures that may work better, depending on assumptions!

DHTs

- Two options:
  - lookup(key) -> node ID
  - lookup(key) -> data
- When you know the nodeID, you can ask it directly for the data, but specifying interface as -> data provides more opportunities for caching and computation at intermediaries
- Different systems do either. We'll focus on the problem of *locating the node responsible for the data*. The solutions are basically the same.

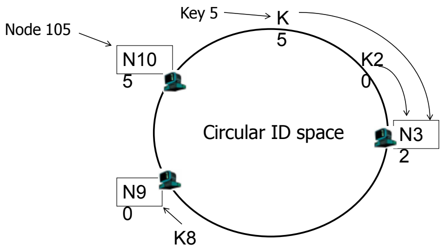
Algorithmic Requirements

- Every node can find the answer
- Keys are load-balanced among nodes
  - Note: We're not talking about *popularity* of keys, which may be wildly different. Addressing this is a further challenge...
- Routing tables must adapt to node failures and arrivals
- How many hops must lookups take?
  - Trade-off possible between state/maint. traffic and num lookups...

Consistent Hashing

- How can we map a key to a node?
- Consider ordinary hashing
  - func(key) % N -> node ID
  - What happens if you add/remove a node?
- Consistent hashing:
  - Map node IDs to a (large) circular space
  - Map keys to same circular space
  - Key "belongs" to nearest node

DHT: Consistent Hashing

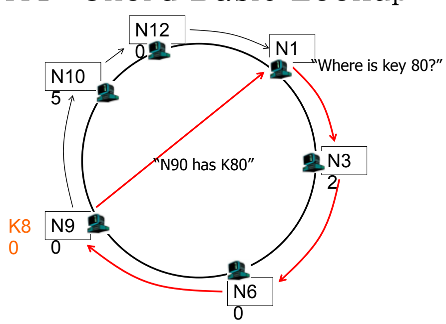


A key is stored at its successor: node with next higher ID

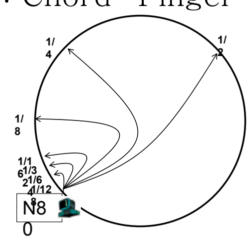
Consistent Hashing

- Very useful algorithmic trick outside of DHTs, etc.
  - Any time you want to not greatly change object distribution upon bucket arrival/departure
- Detail:
  - To have good load balance
  - Must represent each bucket by log(N) "virtual" buckets

DHT: Chord Basic Lookup



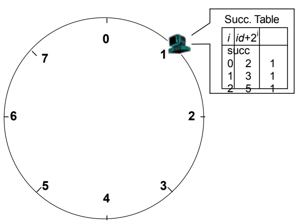
DHT: Chord "Finger Table"



- Entry i in the finger table of node n is the first node that succeeds or equals $n + 2^i$
- In other words, the ith finger points $1/2^{n-i}$ way around the ring

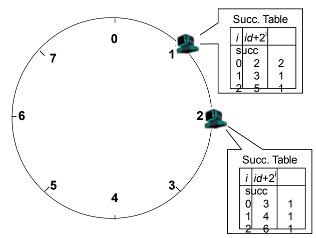
DHT: Chord Join

- Assume an identifier space [0..8]
- Node n1 joins



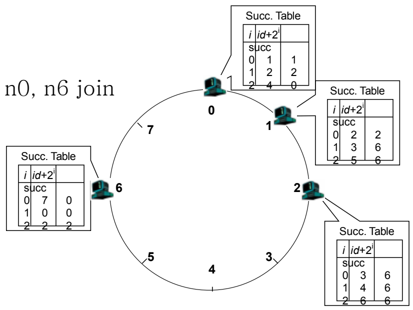
DHT: Chord Join

- Node n2 joins



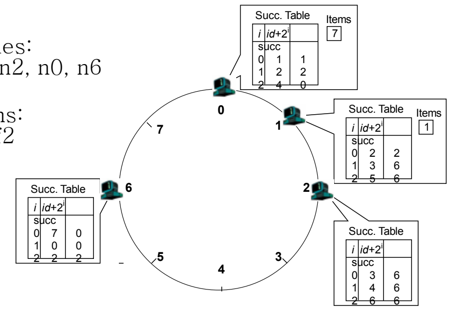
DHT: Chord Join

- Nodes n0, n6 join



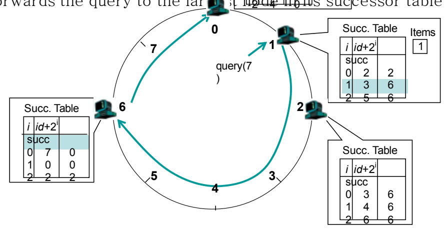
DHT: Chord Join

- Nodes:
  n1, n2, n0, n6
- Items:
  f7, f2



DHT: Chord Routing

- Upon receiving a query for item *id*, a node
- Checks whether stores the item locally
- If not, forwards the query to the largest node in its successor table that doe



DHT: Chord Summary

- Routing table size?
  - Log *N* fingers
- Routing time?
  - Each hop expects to 1/2 the distance to the desire

LH*: A Distributed Linear Hash

- Just because we spoke about *Linear Hashing* earlier in the semester during our discussion of growable hashing schemes…
- It is easy to see that Linear Hashing can be distributed.
  - Each of the buckets is a host
  - The buckets can even be RAM-only
  - A coordinator is invoked by the host of a bucket upon collision
    - The coordinator assigns a new host from a pool of available hosts
    - It then communicates with the two hosts to coordinate the split
    - After the split, the old hosts knows who it split with and can forward queries
  - A retiring host is problematic.
    - Coordinator can supply replacement host to accept bucket of storage
    - Coordinator needs to inform all hosts that cold have split with the retiring host over time, so that they can forward
      - Alternate approach: If unable to find a host, contact the coordinator to find its replacement
  - One extension of Linear Hashing to the distributed environment is called *LH**