Chord

Goal: build a “peer to peer” hash table for the wide area.

- DHT work started in response to Napster, which was a centralized search engine but p2p distribution of files
- DHTs aim to make the search engine decentralized: given a key, find a node with that key/value pair and return the value
- But... nodes come and go relatively quickly (“churn”)

Consistent Hashing

- key k assigned to node equal or just following k -- its "successor"
- gives load balancing (1+\epsilon)K/N keys each, cheap join/leave O(K/N). Based on random hashing. \epsilon is O(\log N)
- if you run O(\log N) virtual nodes per real machine, you can reduce this arbitrarily

Routing: the Ring of successor pointers is key the base case

- CHORD: number of fingers equal to number of bits in the ID space. successor of (NodeID + 2^i). First finger is the successor.
- "recursive" routing in log N steps. Can also do "iterative". tradeoffs?
- What is this: arithmetic!
  - Connection to Group Theory: Cayley and Coset graphs.Deconstructing DHTs

Join (the DHT kind, not the database kind):

- correctness invariants are on data placement and successors. rest of finger table is "just" hints for performance.
  - 1) initialize new node's state (pred and fingers). based on lookups in existing ring. do bulk lookups to save cases where finger[i] and finger[i+1] are the same, makes things scale with net size, not address space size. Also, could ask neighbor for his finger table and pred to bootstrap, reducing init time to log(N)
  - 2) update fingers and pred of existing nodes to point to new node. We basically know who shold point to us: the node at n-2^(i-1) and a contiguous run of its predecessors
  - 3) move state (or ask app to do so). this is dodged. can you make this atomic in a p2p system?

Concurrent operation!

- how does this work when lots of this is happening at once?
- lookup cases: fast, slow (bad fingers), and wrong (bad successors). Can you detect "wrong"?
- stabilization: separate correctness and performance maintenance. Stabilization updates successors. Works if network remains connected. Idea: "fix forward": periodically check your successor's pred to see if you've got the wrong successor, and if so notify new successor. Note that join can now simply point to successor, does not set up anyone

else anymore -- stabilization does that!

- theorems: once linked in, always linked in. eventual consistency of successors with quiescence (no more joins).
- fixing fingers: well, notice that a single join doesn't mess up fingers much. if finger fixing goes faster than NETWORK DOUBLING IN SIZE, things remain log lookup. So fix fingers lazily, on error perhaps.

Failure:

- to deal with this, maintain r = O(log n) successors under stabilize, not just one. upon failure, skip the successor. stabilize will take care of the rest.
- A challenge: network locality. Many many schemes proposed for this, some quite fancy. Basic idea is to choose fingers with some randomness, and maintain multiple alternatives via measurement (where distance typically equals latency).