

CS262, Spring 2007 (Hellerstein/Brewer): Chord, DHT Geometries

- Chord
 - mapping keys to nodes. dynamically, in general. what is this for? why dynamic?
 - vs. DNS. (what is DNS for? What did it replace?)
 - vs. p2p "search" (what is that for? what did it replace?)
 - what is the relationship between content, names, and addresses? Internet philosophy? Database philosophy? Sensornet philosophy?
 - Note: this is not just "p2p" in the narrow sense of filesharing.
 - 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. ϵ 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 $(\text{NodeID} + 2^i)$. 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 $\text{finger}[i]$ and $\text{finger}[i+1]$ 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 should point to us: the node at $n-2^i$ 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

CS262, Spring 2007 (Hellerstein/Brewer): Chord, DHT Geometries

- 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).
- Gummadi²
 - Fair bit of the lit on DHTs proposes a whole package, without reflecting on impact of topology choice beyond $\log n$. Goal here is look at many topologies, and evaluate two kinds of FLEXIBILITY: neighbor selection, and route selection. These can affect both resilience and locality (latency).
 - Routing-level design is the flexibilities above. System-level design includes iterative v recursive, caching, etc.
 - They refer to the "geometry" of the routing, which is kind of ill defined. Their point is to distinguish the fundamental topology space from the algorithm that chooses how to instantiate a pointn in that space (in terms of neighbors and routing choices.)
 - Neighbor selection: not possible in vanilla Chord, but a very good idea, and easy to do in Chord
 - sequential neighbors (the ring). always give progress.
 - many topologies add this on top of their regular stuff as needed for a hybrid.
 - Next-Hop selection:
 - needed for failure handling
 - good for lookup latency
 - Significant confusion in this paper on how to characterize the "geometries":
 - Plaxton/Tapestry/Pastry geometry is the corners of a hypercube -- route via "bit fixing" from high-order to low-order. But it's constrained so that you correct bits from high to low order. "Prefix hypercube". Gives a tree metaphor. Note that when you pick a "bit-correcting" neighbor, you can choose the lower-order bits arbitrarily. So neighbor selection here is actually quite flexible -- many possible routing tables per node. But, once picked, NO routing flexibility.
 - CAN -- slightly revisionist history. Here CAN is described as a hypercube where neighbors fix precisely one bit. Hence can fix bits in any order, because changing a low-order bit doesn't twiddle high-order bits. Hence flexible route selection ($\log N$ choices), but fixed neighbor selection.
 - Butterfly network: constant state with \log routing. Think of $\log n$ "stages". All nodes at stage i can fix bit i . links back and forward in stages, and up and down a stage. No flexibility in neighbors, last $\log N$ (inrastage) routing has no route flexibility -- seems like a price you pay for constant state.
 - Ring (generalized Chord): can pick fingers within the appropriate range $[2^i, 2^{i+1})$, rather than at 2^i per se. neighbor selection flexibility akin to plaxton. also $\log N$ route flexibility (fix one bit again).
 - XOR: $\log n$ neighbors, pick one whose XOR distance is in $[2^i, 2^{i+1})$. Works out much like plaxton, but on failure can fix any bit, albeit other bits may get "unfixed" in doing so. Essentially back to the flexible hypercube.
 - Hybrid: Pastry is Tree+Ring.
 - Resilience
 - routing recovery: repopulating the routing table
 - static resilience: how well can you do without repopulating routing tabel?

CS262, Spring 2007 (Hellerstein/Brewer): Chord, DHT Geometries

- static resilience corresponds to route selection flexibility! in terms of both reachability and DHT-level "stretch" (% increase in average pathlength over no-failure)
- sequential neighbors (multiple successors) help a LOT. However, results in significant stretch. Are more successors better than more fingers? Not clear -- they say similar but smaller increased resilience, with much better stretch numbers.
- Latency
 - Proximity Neighbor Selection (PNS)
 - Proximity Route Selection (PRS) (really proximity next-hop selection)
 - Proximity ID Select (PIS). This is hard, and hard to load-balance. Forget it.
 - Obviously not all geometries can do PNS, not all PRS.
 - Can use random sampling for PNS.
 - For PRS, try to preserve the pathlength. Works for ring and hypercube, and (of course) tree. Not XOR unfortunately. So for XOR do the greedy thing unless improved latency of a non-greedy hop is more than average latency in the network (measured magically).
- Latency Experiments
 - hard to come up with a good representative latency distribution on a real topology. Use a single-node view of the network as representative (matches symmetric DHT design intuition)
 - So, which is better: PNS or PRS? XOR and Ring can do both. Ring can do both with fixed-length paths. PNS is clear winner -- because for high bits, linear choices of neighbor, whereas each hop has only log choices for next hop.
 - Most important: pick a geometry that enables PNS, PRS is nice too. Beyond that, more minor details (e.g. XOR vs. Ring) don't matter so much.
 - BTW, proximity can be done pretty well in an absolute sense!
- Local Convergence
 - E.g. for aggregation or multicast trees, for caching up the path
 - Want nodes that are close to converge quickly when routing to a common dest
 - Picture: a "domain", with lots of faraway stuff.
 - PNS again a clear winner over PRS: number of next hops to choose in PRS is too small, likely to be outside domain.
 - Sampling for PNS plays a big role here though.
 -