P2P DHT lecture

History

- MIT folks invented consistent hashing
  - a way to partition keys across nodes, and minimize repartition costs when add/remove nodes
  - used as a way of partitioning content across CDN or Web cache nodes
  - simple design
    - hash all URLs onto unit circle
    - hash all servers onto unit circle
    - all URLs that preceed server are mapped onto it
  - tree implementation does it
    - insert server hashes into binary tree
    - lookup URL hash in tree to find successor
- P2P file-sharing happened
  - inspired people to look for decentralized layers for systems
    - why?
      - notion that large-scale systems are fundamentally how we’re going to build things
      - notion that decentralization is good for a bunch of reasons: fault tolerance, privacy/anonymity, scalability
      - notion that centralized systems create lock-in similar to proprietary OSs, and that the only way to foster academic or approachable innovation for this class of system is to allow decentralized, p2p systems to do it
      - if so, need some abstractions and building blocks to simplify / build on top of
      - storage and routing seem to be important
  - hash table quickly settled on as on potentially interesting layer
    - why?
      - is essentially an indirection scheme
        - provides rendezvous inside network
      - and it provides associative/content-based lookup
        - abstracts location away from value
        - scalable storage independent of location
    - storage: insertion, lookup
    - anycast:
      - insert(group_name, node1) insert(group_name, node2)
        -
      - to do anycast, pick_one(lookup(group_name))
    - a pub-sub subscribe list
    - mobile IP
      - virtual name in DHT, physical name is lookup
      - a lot like a TLB
  - research “land rush” to build scalable, consistent distributed hash tables
• Chord, Pastry, CAN
  o Chord: distributed version of consistent hashing
  o Pastry: distributed plaxton tree – also ring based
  o CAN: geometric routing
  o interestingly, all seem to have similar properties and problems
    ▪ Properties:
      • log(n) routing table storage for N participants
      • log(n) hops to route to destination
      • significant fraction of nodes need to fail to disrupt reachability
      • concurrent distributed joins possible
        o “eventual convergence” as nodes come and go
        o no strong consistency bounds [ugh]
    ▪ Problems:
      • path inflation – “relative delay penalty” to IP route
        o locality awareness as key
          ▪ flexibility in choosing neighbors and routes as solution to getting there
      • need to defend against malicious nodes
        o sybil attacks:
          ▪ blockades: take away ability to choose virtual node ID
        o node harvesting
        o data harvesting
        o misrouting: need to recover
        o returning false values: “detect” at higher level, solve with redundancy
      • load balancing
        o # of routes that flow through node
          ▪ spread requests across node IDs, assume right thing happens
        o # of keys that reside on node
          ▪ lots of keys, virtual servers, caching, etc.
      • “scruffiness” in consistency/coherence semantics born from churn
        o can’t promise much; especially if caching or replication turned on
          ▪ no cache consistency semantics defined
          ▪ apps usually require non-mutable data as result, or no caching
      • easy to handle popular part of popularity curve, hard to handle tail
        o caching and natural replication in particular
major challenge: what apps can you build using this layer?

- this papers – storage systems
- other papers – the other ideas. backup, multicast/anycast video, file sharing, disappearing data.

Personally, I think the question is backwards

- what apps are best built on this layer, as opposed to some other abstraction or technique?

popular P2P systems using DHTs

- Kademlia, Overnet, eDonkey

idea seems to be cooling off now…though if you squint, the data center storage papers have elements of DHTs.

Chord overview

- ring
  - node IDs – hashed into ring
  - key – kashed into ring
  - successor(key) stores the key
- linked list around Chord for correctness
  - insertion – split linked list
  - removal – patch linked list
    - challenge: dealing with silent failures
      - R successors maintained to recover
      - stabilization algorithm; periodically ask neighbors who their neighbors are, exchange, converge
- finger pointers – tunnel through ring space
  - log(N) fingers
  - ith entry contains identity of first node that succeeds node n by at least $2^i$ on ring
    - what it looks like
    - why “first node”? can have better flexibility than that; can be anywhere in the interval $[2^i, 2^i+1)$.
    - this is tremendous freedom – allows to make highly locality aware
  - populate finger table by querying existing node and stealing the plum entries from it
    - as nodes come/go: if finger table entry stale, re-acquire from other node. if joined, need to insert into other nodes’ finger tables – must find them. deterministic in practice.
- routing
  - worst case average N/2 using successor list
  - finger tables, assuming correct, $\log(n)$
  - algorithm:
    - fetch routing table from current node
      - pick next hop from routing table
      - set as current node
    - pick next hop
• q: what are you trying to optimize?
  o hop count?
  o network latency?
  o something else?

• smallest # of chord hops: max such that node is a predecessor
• might have terrible RDP
• CFS: proposes compromise between chord distance and network distance

• node authentication
  o nodeid = hash(IP + virtual node #)
  o is remotely verifiable
  o prevents attacker from controlling nodeid

• load balancing
  o virtual servers lets you pick # of nodes per physical server
  o is this enough?
    ▪ massive heterogeneity possible in participants – bandwidth, disk capacity, CPU
    ▪ moderate variation possible in key assignment – normal distribution, implying each with k + sqrt(k) keys.
    ▪ significant variation possible in key load
    ▪ significant variation possible in value size
    ▪ virtual server idea to smooth out imbalances – mostly to deal with heterogeneity in participants and keyspace issues.

Applications

• CFS
  o idea: disk and DHT have the same interface
    ▪ can in principle map file system directly onto DHT
    ▪ CFS == SFSRO mapped onto DHT rather than disk blocks
  o issue with this?
    ▪ reliability different – network/node failures vs. block failures
      ▪ need replication to make OK
      ▪ replication expensive under high churn
    ▪ trust very different – malicious nodes out there, disk probably not
      ▪ name blocks by hash of their content; self-verifying
    ▪ latency very different –
      ▪ 5-6 hops * 20-50ms/hop == 100-200ms/fetch
      ▪ caching very important to achieve good performance
      ▪ same problem as all P2P systems – only helps with head of popularity curve [“natural replication”]
    ▪ bandwidth very different – 40MB/s disk vs. what??
      ▪ no real notion of “sequential bandwidth” like a disk has
• blocksize increase is only way to improve – 8KB for CFS(?!)
  ▪ administrative boundaries different
  • hard for you to control quality of storage of your files
    o best you can do is manually replicate [insert same file with multiple names]
  • how do you do debugging in this kind of system? who is allowed to “fix” system if it breaks?
  o why do decentralized storage in the first place?
    ▪ thought experiment: “because we can”
    ▪ CFS as backup system
    ▪ popular content distribution mechanism
      • think of CFS kind of having Akamai-like functions built in