6.824 2018 Lecture 19: P2P, DHTs, and Chord

Reminders:
  course evaluations
  4B due friday
  project reports due friday
  project demos next week
  final exam on May 24th

Today's topic: Decentralized systems, peer-to-peer (P2P), DHTs
  potential to harness massive [free] user compute power, network b/w
  potential to build reliable systems out of many unreliable computers
  potential to shift control/power from organizations to users
  appealing, but has been hard in practice to make the ideas work well

Peer-to-peer
  [user computers, files, direct xfers]
  users computers talk directly to each other to implement service
    in contrast to user computers talking to central servers
  could be closed or open
  examples:
    bittorrent file sharing, skype, bitcoin

Why might P2P be a win?
  spreads network/caching costs over users
  absence of central server may mean:
    easier/cheaper to deploy
    less chance of overload
    single failure won't wreck the whole system
    harder to attack

Why don't all Internet services use P2P?
  can be hard to find data items over millions of users
  user computers not as reliable as managed servers
  if open, can be attacked via evil participants

The result is that P2P has been limited to a few niches:
  [Illegal] file sharing
    Popular data but owning organization has no money
  Chat/Skype
    User to user anyway; privacy and control
  Bitcoin
    No natural single owner or controller

Example: classic BitTorrent
  a cooperative, popular download system
  user clicks on download link for e.g. latest Linux kernel distribution
    gets torrent file w/ content hash and IP address of tracker
  user's BT app talks to tracker
    tracker tells it list of other users w/ downloaded file
  user't BT app talks to one or more users w/ the file
  user's BT app tells tracker it has a copy now too
  user's BT app serves the file to others for a while
  the point:
    provides huge download b/w w/o expensive server/link

But: the tracker is a weak part of the design
  makes it hard for ordinary people to distribute files (need a tracker)
  tracker may not be reliable, especially if ordinary user's PC
  single point of attack by copyright owner, people offended by content

BitTorrent can use a DHT instead of a tracker
  this is the topic of today's readings
  BT apps cooperatively implement a "DHT"
    a decentralized key/value store, DHT = distributed hash table
  the key is the torrent file content hash ("infohash")
  the value is the IP address of an BT app willing to serve the file
    Kademlia can store multiple values for a key
  app does get(infohash) to find other apps willing to serve
    and put(infohash, self) to register itself as willing to serve
    so DHT contains lots of entries with a given key:
      lots of peers willing to serve the file
  app also joins the DHT to help implement it

Why might the DHT be a win for BitTorrent?
  more reliable than single classic tracker

```
    keys/value spread/cached over many DHT nodes
    while classic tracker may be just an ordinary PC
  less fragmented than multiple trackers per torrent
    so apps more likely to find each other
  maybe more robust against legal and DoS attacks

How do DHTs work?

Scalable DHT lookup:
  Key/value store spread over millions of nodes
  Typical DHT interface:
    put(key, value)
    get(key) -> value
  weak consistency; likely that get(k) sees put(k), but no guarantee
  weak guarantees about keeping data alive

Why is it hard?
  Millions of participating nodes
  Could broadcast/flood each request to all nodes
    Guaranteed to find a key/value pair
    But too many messages
  Every node could know about every other node
    Then could hash key to find node that stores the value
    Just one message per get()
    But keeping a million-node table up to date is too hard
  We want modest state, and modest number of messages/lookup

Basic idea
  Impose a data structure (e.g. tree) over the nodes
    Each node has references to only a few other nodes
  Lookups traverse the data structure -- "routing"
    I.e. hop from node to node
  DHT should route get() to same node as previous put()

Example: The "Chord" peer-to-peer lookup system
  Kademlia, the DHT used by BitTorrent, is inspired by Chord

Chord's ID-space topology
  Ring: All IDs are 160-bit numbers, viewed in a ring.
  Each node has an ID, randomly chosen, or hash(IP address)
  Each key has an ID, hash(key)

Assignment of key IDs to node IDs
  A key is stored at the key ID's "successor"
    Successor = first node whose ID is >= key ID.
    Closeness is defined as the "clockwise distance"
  If node and key IDs are uniform, we get reasonable load balance.

Basic routing -- correct but slow
  Query (get(key) or put(key, value)) is at some node.
  Node needs to forward the query to a node "closer" to key.
    If we keep moving query closer, eventually we'll hit key's successor.
  Each node knows its successor on the ring.
    n.lookup(k):
      if n < k <= n.successor
        return n.successor
      else
        forward to n.successor
  I.e. forward query in a clockwise direction until done
  n.successor must be correct!
    otherwise we may skip over the responsible node
    and get(k) won't see data inserted by put(k)

Forwarding through successor is slow
  Data structure is a linked list: O(n)
  Can we make it more like a binary search?
    Need to be able to halve the distance at each step.

log(n) "finger table" routing:
  Keep track of nodes exponentially further away:
    New state: f[i] contains successor of n + 2^i
    n.lookup(k):
      if n < k <= n.successor:
        return successor
      else:
        n' = closest_preceding_node(k) -- in f[]
        forward to n'
```

for a six-bit system, maybe node 8's finger table looks like this:
  0: 14
  1: 14
  2: 14
  3: 21
  4: 32
  5: 42

Why do lookups now take log(n) hops?
  One of the fingers must take you roughly half-way to target

Is log(n) fast or slow?
  For a million nodes it's 10 hops (since a hop is only needed to correct a bit).
  If each hop takes 50 ms, lookups take half a second.
  If each hop has 10% chance of failure, it's a couple of timeouts.
  So: good but not great.
    Though with complexity, you can get better real-time and reliability.

Since lookups are log(n), why not use a binary tree?
  A binary tree would have a hot-spot at the root
    And its failure would be a big problem
  The finger table requires more maintenance, but distributes the load

How does a new node acquire correct tables?
  General approach:
    Assume system starts out w/ correct routing tables.
    Add new node in a way that maintains correctness.
    Use DHT lookups to populate new node's finger table.
  New node m:
    Sends a lookup for its own key, to any existing node.
      This yields m.successor
    m asks its successor for its entire finger table.
  At this point the new node can forward queries correctly
  Tweaks its own finger table in background
    By looking up each m + 2^i

Does routing *to* new node m now work?
  If m doesn't do anything,
    lookup will go to where it would have gone before m joined.
    I.e. to m's predecessor.
    Which will return its n.successor -- which is not m.
  We need to link the new node into the successor linked list.

Why is adding a new node tricky?
  Concurrent joins!
  Example:
    Initially: ... 10 20 ...
    Nodes 12 and 15 join at the same time.
    They can both tell 10+20 to add them,
      but they didn't tell each other!
  We need to ensure that 12's successor will be 15, even if concurrent.

Stabilization:
  Each node keeps track of its current predecessor.
  When m joins:
    m sets its successor via lookup.
    m tells its successor that m might be its new predecessor.
  Every node m1 periodically asks successor m2 who m2's predecessor m3 is:
    If m1 < m3 < m2, m1 switches successor to m3.
    m1 tells m3 "I'm your new predecessor"; m3 accepts if closer
      than m3's existing predecessor.
  Simple stabilization example:
    initially: ... 10 <==> 20 ...
    15 wants to join
    1) 15 tells 20 "I'm your new predecessor".
    2) 20 accepts 15 as predecessor, since 15 > 10.
    3) 10 asks 20 who 20's predecessor is, 20 answers "15".
    4) 10 sets its successor pointer to 15.
    5) 10 tells 15 "10 is your predecessor"
    6) 15 accepts 10 as predecessor (since nil predecessor before that).
    now: 10 <==> 15 <==> 20
  Concurrent join:
    initially: ... 10 <==> 20 ...
    12 and 15 join at the same time.
    * both 12 and 15 tell 20 they are 20's predecessor; when
      the dust settles, 20 accepts 15 as predecessor.

```
    * now 10, 12, and 15 all have 20 as successor.
    * after one stabilization round, 10 and 12 both view 15 as successor.
      and 15 will have 12 as predecessor.
    * after two rounds, correct successors: 10 12 15 20

To maintain log(n) lookups as nodes join,
  Every one periodically looks up each finger (each n + 2^i)

What about node failures?
  Nodes fail w/o warning.
  Two issues:
    Other nodes' routing tables refer to dead node.
    Dead node's predecessor has no successor.
  Recover from dead next hop by using next-closest finger-table entry.
  Now, lookups for the dead node's keys will end up at its predecessor.
  For dead successor
    We need to know what dead node's n.successor was
      Since that's now the node responsible for the dead node's keys.
    Maintain a _list_ of r successors.
    Lookup answer is first live successor >= key

Dealing with unreachable nodes during routing is important
  "Churn" is high in open p2p networks
  People close their laptops, move WiFi APs, &c pretty often
  Fast timeouts?
  Explore multiple paths through DHT in parallel?
    Perhaps keep multiple nodes in each finger table entry?
    Send final messages to multiple of r successors?
    Kademlia does this, though it increases network traffic.

Geographical/network locality -- reducing lookup time
  Lookup takes log(n) messages.
    But messages are to random nodes on the Internet!
    Will often be very far away.
  Can we route through nodes close to us on underlying network?
  This boils down to whether we have choices:
    If multiple correct next hops, we can try to choose closest.

Idea: proximity routing
  to fill a finger table entry, collect multiple nodes near n+2^i on ring
  perhaps by asking successor to n+2^i for its r successors
  use lowest-ping one as i'th finger table entry

What's the effect?
  Individual hops are lower latency.
  But less and less choice as you get close in ID space.
  So last few hops are likely to be long.
  Though if you are reading, and any replica will do,
    you still have choice even at the end.

Any down side to locality routing?
  Harder to prove independent failure.
    Maybe no big deal, since no locality for successor lists sets.
  Easier to trick me into using malicious nodes in my tables.

What about security?
  Can someone forge data? I.e. return the wrong value?
    Defense: key = SHA1(value)
    Defense: key = owner's public key, value signed
    Defense: some external way to verify results (Bittorrent does this)
  Can a DHT node claim that data doesn't exist?
    Yes, though perhaps you can check other replicas
  Can a host join w/ IDs chosen to sit under every replica of a given key?
    Could deny that data exists, or serve old versions.
    Defense: require (and check) that node ID = SHA1(IP address)
  Can a host pretend to join millions of times?
    Could break routing with non-existant hosts, or control routing.
    Defense: node ID = SHA1(IP address), so only one node per IP addr.
    Defense: require node to respond at claimed IP address.
             this is what trackerless Bittorrent's token is about
  What if the attacker controls lots of IP addresses?
    No easy defense.
  But:
    Dynamo gets security by being closed (only Amazon's computers).
    Bitcoin gets security by proving a node exists via proof-of-work.

How to manage data?
```

  Here is the most popular plan.
    [diagram: Chord layer and DHT layer]
    Data management is in DHT above layer, using Chord.
  DHT doesn't guarantee durable storage
    So whoever inserted must re-insert periodically
    May want to automatically expire if data goes stale (bittorrent)
  DHT replicates each key/value item
    On the nodes with IDs closest to the key, where looks will find them
    Replication can help spread lookup load as well as tolerate faults
  When a node joins:
    successor moves some keys to it
  When a node fails:
    successor probably already has a replica
    but r'th successor now needs a copy

Summary
  DHTs attractive for finding data in large p2p systems
    Decentralization seems good for high load, fault tolerance
  But: log(n) lookup time is not very fast
  But: the security problems are difficult
  But: churn is a problem, leads to incorrect routing tables, timeouts
  Next paper: Amazon Dynamo, adapts these ideas to a closed system.

References
Kademlia: www.scs.stanford.edu/~dm/home/papers/kpos.pdf
Accordion: www.news.cs.nyu.edu/~jinyang/pub/nsdi05-accordion.pdf
Promiximity routing: https://pdos.csail.mit.edu/papers/dhash:nsdi/paper.pdf
Evolution analysis: http://nms.csail.mit.edu/papers/podc2002.pdf
Sybil attack: http://research.microsoft.com/pubs/74220/IPTPS2002.pdf