6.824 2015 Lecture 22: P2P, DHTs, and Chord

Lecture outline:
  peer-to-peer (P2P)
  BitTorrent
  DHTs
  Chord

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:
    skype, video and music players, file sharing

Why might P2P be a win?
  spreads network/caching costs over users
  absence of server may mean:
    easier 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 than managed servers
  if open, can be attacked via evil participants

The result is that P2P has some successful niches:
  Client-client video/music, where serving costs are high
  Chat (user to user anyway; privacy and control)
  Popular data but owning organization has no money
  No natural single owner or controller (Bitcoin)
  Illegal file sharing

Example: classic BitTorrent
  a cooperative download system, very popular!
  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 client talks to tracker
    tracker tells it list of other user clients w/ downloaded file
  user't BT client talks to one or more client's w/ the file
  user's BT client tells tracker it has a copy now too
  user's BT client serves the file to others for a while
  the point:
    provides huge download b/w w/o expensive server/link

BitTorrent can also use a DHT instead of / as well as a tracker
  this is the topic of today's readings
  BT clients cooperatively implement a giant key/value store
  "distributed hash table"
  the key is the file content hash ("infohash")
  the value is the IP address of a client willing to serve the file
    Kademlia can store multiple values for a key
  client does get(infohash) to find other clients willing to serve
    and put(infohash, self) to register itself as willing to serve
  client also joins the DHT to help implement it

Why might the DHT be a win for BitTorrent?

```
  single giant tracker, less fragmented than many trackers
    so clients more likely to find each other
  maybe a classic tracker too exposed to legal &c attacks
  it's not clear that BitTorrent depends heavily on the DHT
    mostly a backup for classic trackers?

How do DHTs work?

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

Why is it hard?
  Millions of participating nodes
  Could broadcast/flood request -- but too many messages
  Every node could know about every other node
    Then hashing is easy
    But keeping a million-node table up to date is 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
  By Stoica, Morris, Karger, Kaashoek and Balakrishnan; 2001

Chord's ID-space topology
  Ring: All IDs are 160-bit numbers, viewed in a ring.
  Each node has an ID, randomly chosen

Assignment of key IDs to node IDs?
  Key stored on first node whose ID is equal to or greater than key ID.
    Closeness is defined as the "clockwise distance"
  If node and key IDs are uniform, we get reasonable load balance.
  So keys IDs should be hashes (e.g. bittorrent infohash)

Basic routing -- correct but slow
  Query 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 win.
  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 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

There's a binary lookup tree rooted at every node
  Threaded through other nodes' finger tables
  This is *better* than simply arranging the nodes in a single tree
    Every node acts as a root, so there's no root hotspot
    But a lot more state in total

Is log(n) fast or slow?
  For a million nodes it's 20 hops.
  If each hop takes 50 ms, lookups take a second.
  If each hop has 10% chance of failure, it's a couple of timeouts.
  So in practice log(n) is better than O(n) but not great.

How does a new node acquire correct tables?
  General approach:
    Assume system starts out w/ correct routing tables.
    Use routing tables to help the new node find information.
    Add new node in a way that maintains correctness.
  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.
  So, for correctness, m's predecessor needs to set successor to m.
    Each node keeps track of its current predecessor.
    When m joins, tells its successor that its predecessor has changed.
    Periodically ask your successor who its predecessor is:
      If that node is closer to you, switch to that guy.
    So if we have x m y
      x.successor will be y (now incorrect)
```

```
      y.predecessor will be m
      x will ask its x.successor for predecessor
        x learns about m
        sets x.successor to m
        tells m "x is your predecessor"
        called "stabilization"
  Correct successors are sufficient for correct lookups!

What about concurrent joins?
  Two new nodes with very close ids, might have same successor.
  Example:
    Initially 40 then 70
    50 and 60 join concurrently
    at first 40, 50, and 60 think their successor is 70!
    which means lookups for e.g. 45 will yield 70, not 50
    after one stabilization, 40 and 50 will learn about 60
    then 40 will learn about 50

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

Chord's routing is conceptually similar to Kademlia's
  Finger table similar to bucket levels
    Both halve the metric distance for each step
    Both are about speed and can be imprecise
  n.successor similar to Kademlia's requirement that
    each node know of all the nodes that are very close in xor-space
    in both cases care is needed to ensure that different lookups
      for same key converge on exactly the same node

What about node failures?
  Assume nodes fail w/o warning. Strictly harder than graceful departure.
  Two issues:
    Other nodes' routing tables refer to dead node.
    Dead node's predecessor has no successor.
  If you try to route via dead node, detect timeout, treat as empty table entry.
    I.e. route to numerically closer entry instead.
  For dead successor
    Failed node might have been just before key ID!
      So we need to know what its n.successor was
    Maintain a _list_ of successors: r successors.
    Lookup answer is first live successor >= key
      or forward to *any* successor < key

Kademlia has a faster plan for this
  send alpha (or k) lookup RPCs in parallel, to different nodes
  send more lookups as previous ones return info about nodes closer to key
  single non-responsive node won't cause lookup to suffer a timeout

Dealing with unreachable nodes during routing is extremely important
  "Churn" is very high in open p2p networks
  People close their laptops, move WiFi APs, &c pretty often
  Measurement of Bittorrent/Kademlia suggest lookups are not very fast

Geographical/network locality -- reducing lookup time
  Lookup takes log(n) messages.
    But they 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:
  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 (lower node density) as you get close in ID space.
  So last few hops likely to be very long.
  Though if you are reading, and any replica will do,
    you still have choice even at the end.

What about security?
  Self-authenticating data, e.g. key = SHA1(value)
    So DHT node can't forge data
    Of course it's annoying to have immutable data...
  Can someone cause millions of made-up hosts to join?
    They don't exist, so routing will break?
    Don't believe new node unless it responds to ping, w/ random token.
  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?
    Or "join" many times, so it is most of the DHT nodes?
    Maybe you can require (and check) that node ID = SHA1(IP address)

Why not just keep complete routing tables?
  So you can always route in one hop?
  Danger in large systems: timeouts or cost of keeping tables up to date.

How to manage data?
  Here is the most popular plan.
  DHT doesn't guarantee durable storage
    So whoever inserted must re-insert periodically if they care
    May want to automatically expire if data goes stale (bittorrent)
  DHT does replicate 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

Retrospective
  DHTs seem very promising for finding data in large p2p systems
    Decentralization seems good for load, fault tolerance
  But: the security problems are difficult
  But: churn is a serious problem, particularly if log(n) is big
  So DHTs have not had the impact that many hoped for
```