Distributed Hash Tables and Self Organizing Networks

6.824 2014 lecture

Jonathan Perry

Announcements

- Amazon credit for projects

Intro

- Kademlia came out of research into peer-to-peer (P2P) networks in the early 2000's. The goal was to maintain applications that support millions of users, leveraging the users' systems as the infrastructure.
- In a P2P network, the relationship between participants is more *symmetric*: instead of having a separation between client and server, each node acts both as a client and a server.

P2P had a huge impact

- Napster opened in June 1999, had **80M** registered users by Febuary 2001
- In 2008, P2P constituted **54.56%** of Internet throughput in SW.Europe
- In 2012, BitTorrent had > **152M** active clients

Note: hard to get reliable figures, see wikipedia pages for references

Trends change

- "Cloud", centrally managed systems have become prevalent:
  - Netflix + Youtube > 50% of peak Internet traffic in North America
  - P2P file-sharing < 10% of peak Internet traffic in North America
    according to Sandvine's 2H 2013 report
- Q: Which service many use daily was a big P2P success? Hints: was bought by a big company in 2011; it's claimed that it's name was originally "Sky peer-to-peer"; the company was Microsoft.

Skype

- Skype's directory servers and traffic relays (to bypass NATs) were hosted by regular users that ran Skype.
- But in December 2010, they started moving these into the cloud ahead of the Microsoft acquisition (first to Skype's datacenters within EC2, then to Microsoft's datacenters)
- "The move was made in order to improve the Skype experience, primarily to improve the reliability of the platform and to increase the speed with which we can react to problems. The move also provides us with the ability to quickly introduce cool new features that allow for a fuller, richer communications experience in the future." - Mark Gillett, Skype (from same blog post)
- While research into P2P decreased, many ideas and techniques still useful! We'll see α ("opportunistic latency"), caching to avoid flash crowds, lazy replication, piggy-backing of maintenance traffic over production traffic.

But, why not centralized?

- Say three machines duplicate all data, clients have a list of all three servers. Query first. If timeout, try second or third.
- Q: Is this a viable solution for a service? If so, give an example
- A: yes, DNS
- Q: So when is this not appropriate?
- A: When there are many keys

Scaling memory via broadcast

- If we want to handle many keys in RAM, we could potentially divide keys to many machines. Then, to find a value, broadcast the request to all servers. The server with the key will respond.
- Q: does this seem feasible for a service?
- A: Google search does this, stores reverse indices on 10K machines, then aggregates results.
- Q: When is this not feasible?
- A: Broadcast makes sense if (1) each query requires a lot of work, or (2) request volume is low. If the amount of work is small, and volume is large, then communication becomes a bottleneck.
- Gnutella was built with broadcast to search for files. Queries would be flooded across the network. As the network became large, queries were only able to cover a small fraction of the network, making it hard to recover rare items. (there were other complexities such as avoiding loops when flooding queries, but the scaling issues are most important)

Consistent Hashing

- Karger, Lehman, Leighton, Panigrahy, Levine & Lewin. "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web", STOC'97
- Saw this in the Dynamo paper
- Each key and node are associated with a (128 or 160)-bit number. For keys, the key string is hashed, for nodes their IP address. The hash produces uniform-looking outputs. This forms a ring.
- Map all nodes to the ring. Map keys to the ring. Map each key to a node (or several nodes) that is "closest". In Dynamo (and Chord), a key maps to the node immediately after it. In Kademlia, maps to the closest node using the XOR metric: $d(a,b) = a \oplus b$
- Q: can you give an example of two nodes and a key, s.t. the key would map to different nodes under the two schemes (Chord & Kademlia)?
- A: nodes 1000, 0000.
  - The key 0111 is Chord-closer to 1000, but Kademlia-closer to 0000.
  - The key 0001 will also map to 1000 in Chord, 0000 in Kademlia.

BitTorrent

- BitTorrent uses its directory service to track users holding each file (key is file, value is a list of nodes that have the file).
- Given a list of users, the BT client contacts those clients directly and transfers files.
- Q ("general knowledge"): why store the users holding each file rather than the file?
- A:
  - files are large, when joining the network, nodes already have GBs of files, don't want to transfer new ones
  - Incentives. Users might tend to only download data and not upload. BT has a tit-for-tat system, where each node decides which nodes to transmit to based on how much data they sent to it. This incentive requires the nodes desiring the file to also be the nodes serving it.
- In a P2P system, the "guts" of the system are exposed to the other nodes, so the system needs to be designed to prevent abuse from other nodes. We'll only touch on this briefly (spoofing attacks).

Dynamo

- In Dynamo, all servers know all other servers in the network. A Chord-like consistent hashing scheme (mapping each key to the node immediately after it) could be implemented using a binary tree of nodes. Searching for the key in that tree will give the node.
- Q: Why can't BitTorrent use a similar scheme?
- A:
  - Too many nodes (routing tables are big)
  - Nodes in Dynamo are quite stable, but in BitTorrent nodes come and go. When nodes join and leave, too much traffic is needed to update tables.

Distributed Hash Tables

- To keep routing state small at each node, each node keeps a small routing table with pointers into other nodes in the ring.
- Nodes have progressively better knowledge of the nodes around them, but also know some nodes far away. When looking for a key, each hop shrinks the distance to the destination. First jumps are large, but get progressively smaller as the query gets closer to its destination.

Kademlia

- The node keeps k-buckets: lists of k nodes that share a prefix.
- Table starts with one empty bucket. When node U hears of node V, the bucket that would contain V might get split, but only if the bucket's prefix matches U.
- Will use k=1 for these examples.
- At node 00110, simulate these additions: 01100, 10010, 00100, 11101, 00100
- U's table is a tree whose deepest leafs are the prefixes 10010 and 10011, can you tell a prefix of U's ID?
- The buckets are arranged into a very specific type of tree: there is a single "stem", a path from the root to the deepest leaf corresponding to the prefix of U. All other leafs are handing off that stem.
- If you think about *all* nodes of the system arranged in a prefix tree, the routing tree at any given node (the one with the k-buckets) has each bucket representing a sub-tree. There is no further constraint on nodes in the k-bucket, they can be adjacent nodes or equally spaced within the subtree (unlikely, though), as long as they're from the sub-tree.

Routing table size

- The highest k-bucket contains nodes with distances in $[2^{159}, 2^{160})$ of U (its *index* is 159).
- The height of the longest path in the tree is the node's *height*.
- Q: Say each node chooses its ID from a uniform distribution, there are N nodes, a node *x* has height *h*. How many nodes are expected to be in the deepest bucket?
- A: The node at depth h has distances $[2^{160-h}, 2^{160-h+1})$.
  - This range contains 2^{160-h} numbers of the 2^{160}, so a node's probability of choosing an ID in this range is 2^{-h}.
  - E(#nodes in range) = N Pr(specific node in range) = $N\,2^{-h}$.
  - h is unlikely to be much larger than log(N), since those buckets would very likely be empty.
  - So, routing table would contain ~k log N elements.

Node lookup example

- Lookup key 11111.. at node 00110..
- At index 159 of node 00110, found 10010.
- At index 158 of node 10010, found 11100. (jumped two!)
- At index 156 of node 11100, found 11110.
- and so on, every iteration advances at least one layer down the tree.

Holes in routing table

- Q: Can there be empty buckets in the routing table?
- A: Yes. For example a node 00110 in a network that only has 00100 and 00010, nothing in bucket for prefix 1*.
- Q: How does lookup handle an empty bucket? Here, assume that there doesn't exist a node with an ID that would fall in that bucket.
- A: This would mean that the distance from the lookup key to its destination node is *at least* 2^i, where i is the index of the empty bucket along the search. So, will want to find a node with the minimum distance node among nodes that have distance $[2^i, 2^{i+1})$. So, flip the bit i and continue the search down the tree.

k and α

- Kademlia keeps k nodes in every k-bucket. BitTorrent uses k=8.
- Q: Why?
- A: so if some fail, can still query the subspace within the bucket.
- FIND_NODE works in iterations. In each iteration send the query to α nodes. Each queried node returns the k closest neighbors to the query in its routing tables.
- Q: Why send α queries in parallel? Why not just one?
- The node doesn't have to wait for all α queries from an iteration to return before launching new queries → latency advantage!
  - If one of the nodes failed, the query continues before hitting a time-out.
  - The physical network topology doesn't conform to the DHT topology. In fact, because of the uniform distribution of IDs, there will tend to be no correlation between physical proximity and Kademlia-metric proximity. The α parameter allows the DHT to utilize physically closer nodes to get responses faster.
- Note that α increases the bandwidth required of the system. Sending k queries in parallel might be too much. And after sending k queries, k^2 nodes might return, would we launch k^2 parallel queries? Too much.

FIND_NODE example

- Example of a FIND_NODE query: draw a line left to right. Left is close to the target, right is far. Assume k=3, α=2. Start with marking k nodes "far" (right), these would be the nodes from the querying node's routing tables. Mark Q for queried, A for answered. (note it might be easier to think about the line in log-scale rather than linear: far is *very* far, close is *very* close)
- The node doesn't have to wait for all α queries from an iteration to return before launching new queries. This is where the system is not specified completely: if one query returns, should the system send out new queries ASAP, or wait for more responses, so it might "get lucky" and the query will return a node much closer to the target?
- As more answers come in, the set of known nodes gets closer to the target. Once an iteration gets no nodes closer to the target than previous iterations, the node sends queries to all k closest nodes.
- Q: Why send to all k closest nodes?
- A: When a new node joins, it notifies the k closest nodes to it (by doing FIND_NODE). This k-neighborhood is most likely to know of all nodes closest to them (and hence, closest to the queried ID).
- Before we talk further about how nodes join, let's take a look at the k-bucket maintenance policy.

Maintaining k-buckets

- buckets are maintained in order of time last seen, most recent at bottom.
- When getting a query from node *u*:
  - if u is already in the bucket, move u to bucket's bottom.
  - If u is not in the bucket, and the bucket has less than k nodes, add u.
  - If bucket is full, and there are *questionable* nodes (nodes that have not been seen in 15 minutes), ping them in order until finding a dead node to throw away, insert u instead.
  - If all nodes are live, discard u.

Maintaining k-buckets (cont.)

- Q: When a client sees a new node u, it performs pings to try to keep its old nodes. Why not throw away an arbitrary node and put u there instead?
- A: Because it has been shown on the Gnutella network that nodes are more likely to stay up if they've already been up long
- Q: Would this be true in datacenters too?
- A: Not necessarily. For example if each node goes down for a software update every X days, the older nodes are more likely to go away than the new ones.

P2P sidenote: communication on the Internet

- BitTorrent defines a "good" node as a node that:
  - Answered our query in the last 15 minutes, or
  - Ever responded to any of our query, and sent us a query in the last 15 minutes.
- Q: Why isn't it sufficient that a node send us a query in the last 15 minutes?
- A:
  1. Firewalls. A node might be able to send requests and receive answers, but a firewall might block incoming requests.
  2. Mitigate attacks that could poison the k-buckets or issue bad STOREs. Some Internet providers allow packet spoofing: attackers send out packets that appear to originate from addresses the attackers do not own. Spoofers will not get replies to their messages -- the responses will go to the real owners of the addresses. This is why Kademlia nodes issue "tokens" in their RPCs, and expect to hear the same token in the responses, proving that the issuer of the query owns the address.

Joining Kademlia

- To join, a new node u needs to know of a node w that is already connected.
- Q: How does a BitTorrent client find another node? ("bootstrapping")
  - Through a traditional tracker. The tracker ("peer") tells the DHT client ("node") the addresses of nodes participating in the tracker. This assumes that the user is able to find a tracker, and that the tracker knows of a valid peer.
  - Node list is persisted to disk, so can be re-read next time.
- (1)u performs a FIND_NODE on itself. This triggers key migration (keys are not deleted from their source, to keep k copies of each k/v pair).
- (2)u refreshes all k-buckets farther than closest neighbor (in BT, refresh is done querying for a random node within the bucket). This populates u's k-buckets and inserts u into other nodes' k-buckets
- Building the pointers of the consistent hash using XOR rather than Euclidian distance pays off here. In both these queries, the nodes that u is looking for are the same nodes that need to be notified of u's existence (compare with Chord).
- Q: Kademlia tries to keep old live nodes in its k-buckets. If no node will be willing to insert the new node into its k-buckets, the new node would be unreachable (bad). Is that possible?
- A: Nodes always keep the k closest nodes to themselves (mentioned in the long version of the paper), so the new node would at least join the k-buckets of its closest neighbors.

Storing keys

- The publisher finds the closest k nodes to the key by using FIND_NODE, then issues STORE to all of them.
- To limit stale information in the network (also to remove excessive replication), the k/v pair expires after 24 hours. To keep it in the network, the publisher needs to actively re-publishes the file. This scheme is less appropriate for other systems, e.g. BitTorrent: if the seeder of a torrent disappears, the torrent shouldn't go away. BitTorrent must have some mechanism for this, maybe when a peer announces itself, the timeout is updated.
- Every hour nodes re-publish keys to their k neighbors (calls STORE again on all keys).
- Q:Why is the hourly re-publishing necessary?
  - If nodes leave, keep replication factor at k.
  - Also needed if STOREs can be dropped by the network. If some of the k nodes didn't get the STORE, someone needs to retry.
- Q: Why choose an hour as the re-publishing interval and not 10 minutes or 10 hours?
- A: k and the re-publishing interval are chosen such that keys are not lost when faults occur: it should be very unlikely for all k nodes to fail between the republishing interval. It's a tradeoff between communication and storage overhead. Frequent re-publishing interval means higher communication overhead, but lower storage overhead (k can be lower). Infrequent means low communication overhead and higher storage overhead (large k).

Searching for keys and Consistency

- Q: How does FIND_VALUE differ from FIND_NODE?
- A: If a node has the key stored, it replies with the key instead of a list of nodes. Basically, searching for a key returns the first value it finds.
- Q: What kind of consistency guarantees does Kademlia provide when there are multiple writes to the same key?
- A: Kademlia doesn't even try to handle multiple writes. No read-my-writes. No consistency across updates to different keys. The latest write is not even guaranteed to survive, as re-publishing might overwrite a fresh value with an old value -- no conflict resolution between multiple values.
- BitTorrent uses the DHT to store addresses of peers that are downloading a torrent. An announce_peer adds the peer's address to a list associated with a torrent, so BitTorrent has multiple updates.
- Q: BitTorrent doesn't mention re-publishing. How does it make sure queries can find peers that announce themselves?
- A: Probably relies on peers to re-announce themselves.

Caching and "over-caching"

- Kademlia uses caching to deal with *flash crowds*, when some key becomes extremely popular and receives many queries over a short period of time.
- Whenever a key is queried, the node getting the key STOREs it in the closest node to where the key was found, with expiry time determined by its distance from the original (non-cached) value.
- Assume all nodes are of height h (there are approx. 2^h nodes in the system). If k nodes have the value, then queries would take h hops and will hopefully get load balanced among the k nodes. If 2k nodes store the value, queries are h-1 hops and balanced on ~2k nodes. If 2^jk nodes, queries will find after h-j hops...
- Would like the number of nodes holding the key to grow according to the key's popularity.
- Q: Given expiry time as function of distance (monotinically decreasing), and the frequency of queries for a key, can you give an intuitive upper bound on how many nodes will store the key?
- A: Look at where expiry times become shorter than the key's access frequency. Nodes much farther than the boundary are unlikely to have the key, since even if the key reaches the boundary, it will expire from the boundary node before the next query to the key.

Caching vs. LRU

- Paper mentions LRU schemes (Least Recently Used). In this scheme, the amount of storage is limited. Once space runs out and a new entry needs to be cached, the entry that was least recently accessed is discarded to make room.
- Q: The whole point of using expiry times instead of LRU is not to decide the size of the cache. But what problem is solved with LRU that is not solved with expiry times?
- A: If the access pattern changes, caches could grow beyond node capacity. LRU has a limit on the cache.

Network partition

- Network partitions into two partitions A and B, and heals after 1.5 hours. Will a key X in A be available during and after the partition?
- Answering how network will behave depends on caching, k/v expiry scheme & republishing, k-bucket maintenance policy.
- During the partition, with sufficiently large k (e.g., k=20), the probability of all buckets containing k falling into one side of the network is small. Both sides would have a node with X, so will be able to query it. If k is small, popular keys are more likely to be available, because of caching.
- Assume: k-buckets refresh every hour, and pings all nodes within. If a node is unavailable, it is deleted. Then, after the partition heals, we are left with two disjoint Kademlia networks: all nodes in A are removed from B and vice versa. New keys in A are invisible in B.
- Q: Assume now that unavailable nodes are kept in the k-buckets, and will only be removed them if a new live node appears. Will Kademlia heal to one network?
- A: All k-buckets will have around 1/2 their entries become unavailable. Since there are still queries in both A and B, the buckets close to the root are likely to delete all entries from the other side of the partition. However, unless both A and B grow substantially during the partition, the bottom-most k-buckets should still contain nodes from the other side of the partition. Refreshing these buckets after the network heals should heal the network.

Optimizing pings

- As a query traverses the network, it is likely to hit many nodes that have never heard of the query's initiator. If there are entries in the k-bucket that hadn't been seen in a while (15 minutes in BT), this could trigger multiple pings, as the node tries to find a failed entry, one by one. The cost could be prohibitive.
- An optimization is to keep a *replacement cache* for each bucket, with nodes that have been seen in queries and could replace failed nodes. The node waits until a query has to go out the bucket anyway, and piggy-backs a ping with it. If the ping fails, the bad entry is replaced with a node from the replacement cache.
- Q: Can Kademlia overwhelm the underlying network?
- A: Yes. If there is a lot of query traffic, and all the nodes retry after a fixed timeout, then the network could reach congestion collapse: all nodes are sending, but most packets get dropped along the way, so most of the network capacity is wasted and the system becomes slow. Common solution is to have exponentially increasing timeouts ("exponential backoff").

Optimizing number of hops

- We saw that Kademlia lookups perform $\log_2 N$ hops, to find a key/node.
- Q: Could we decrease the number of hops? (hint: Dynamo that has zero-hop lookups)
- A: Can have 2^b-ary trees instead of binary trees, so instead of getting one bit closer to the target, the lookup gets b bits closer. Another way to look at this is the height of the tree decreases by a factor of b. Result: $1/b \log_2 n$

Optimizing number of hops: Super-nodes

- Q: In the BitTorrent case, we could do something better to reduce the number of hops?
- A: Yes! There is no real requirement that *every* node will participate in the DHT. Can have a small fraction of the nodes participate in the DHT, and the other nodes would use them as a service. These are called super-nodes (diagram: supernodes communicating among themselves, regular nodes talk only to supernodes).
- P2P networks select their super-nodes from among the regular nodes. Usually they would be nodes with more available bandwidth, RAM and CPU, and that were more stable (long uptimes).
- Skype was organized this way. Moving their directory service to their datacenters involved moving the supernodes.
- The number of hops decreases with the smaller network size N.

Optimizing round-trips

- Kademlia and BT use an RPC scheme to query. The query source drives the search.
- However, in other schemes (Freenet), the next hop of the query could forward the query onwards.
- Q: What advantage does this chaining provide? What breaks in current Kademlia?
- A: Advantage: for each hop, only have to pay one-way latency and not round-trip. Disadvantage: α scheme breaks: if every node along the path sends α queries out, the cost grows exponentially with number of hops. If every node only sends out one query, don't get the latency benefits from α.
- Q: So would chaining still be an option?
- A: Yes.
  - If latency is uniform (within a datacenter)
  - By meticulously maintaining RTT to each node in k-bucket, can choose lowest latency and still chain (but maintaining the metrics can be expensive)