Attributes of Distributed Hash Tables and Their Ramifications

Andrew Rosen

July 4, 2014

Chapter 1

Introduction

Let's begin with a discussion of definitions: what is a Distributed Hash Table? The goal of a Distributed Hash Table, or DHT, is to allow nodes to lookup and route to other nodes in the network

One node in a DHT a can, without any single node knowing all the routing information for the network[1]

Structured P2P overlays and DHTs are so closely related that they often can be talked about interchangeably [5].

1.1 History

1.2 The Coordinate System, Responsibility and Neighbors

The vast majority of overlays only use a single coordinate (and therefore a single dimension). This coordinate is assigned essentially at random through a hash function, typically by hashing the IP address and port. If more than one dimension is used, coordinates can be chosen to map to a particular physical feature (geographical location or available bandwidth) or a metric (trust or influence).

All DHTs assign responsibility the same way: the node with an ID that is "closest" to a particular key is responsible for that key. What differs is the metric being used to determine which node is closest. One metric used by Chord (others) is "the node with the closest id larger than the key" (presumably, this is much easier to code than "the closest without going over.").

Almost every DHT operates in a modulus space. The space can be either bidirectional (symphony) or unidirectional (Chord). The overlay shape/layout

[1] Two exceptions here. The first is in a sufficiently small network, nodes will naturally know all node in the network. This scenario can largely be ignored. The second case is ZHT, which assumes a network with very specific conditions

is actually as important as the peer list or routing algorithm. There's typically a distance function, but it's not very interesting.

The domain of the keyspace can be m-bit identifiers, or floats from 0-1. So long as m is the same it is possible to map from one to another.

Hashing Algorithms

SHA1, LSH

1.2.1 Closest Metrics

The advantage of using a successor or predecessor metric that changes in responsibility due to node failure are recognized immediately. In a literal "closest to the key" metric, there is computation involved.

Literal

As in literally the closest. The cost is computation, but the benefit is that the metric is natural and easy to understand

Successors

This is the way Chord works; from the node's perspective, a node is responsible for the keys between its ID and that of its predecessor. Fault tolerance

Predecessors

This is the opposite of how the successor scheme works and interestingly enough, easier to describe. The node with the ID closest without going over a key is responsible for that key. Fault tolerance here works identically to the successor scheme, just in the other direction.

Groups

See Pastry

Regions

1.3 Finding Peers

We can (can we?) break this down in a couple of ways. Gossip vs nongossip (security)

Some DHTs build their peer lists by using suggestions from other nodes. The issue with this is that this opens up a new vector for attacks. A malicious node can lie to another node about what peers to use, but even worse, these lies can then propagate to other nodes.

Others DHTs choose their peers by finding the node closest to a particular ID. For example, a node in Chord chooses fingers by finding the node closest to its ID plus some power of 2. In Symphony, peers are chosen by finding the node closest to some ID, chosen randomly over a probability distribution.

1.4 Routing

All routing algorithms follow the same scheme: the next hop brings me closest to my destination. The very nature of DHTs makes the routing process greedy, and the node chooses the best path based on its peer list (It is possible to use a 1-ahead lookup, thereby keeping a record of your peer's peerlist. The cost becomes quite large for more than two, but routing does speed up.). Specifics aside, there are two ways to look at routing. The first way answers the question "Who is responsible for this particular key?" but rather than asking the entire network, it assumes the entire network consists only of it and the peer list. The other way to look at it that routing is the process of greedily eliminating possible recipients, until the final node must be the node responsible.

Almost every DHT has a $lg(n)$ routing time, eliminating half of the remaining nodes each step. However, Kleinberg Small World Networks have inspired networks that have a polylogarithmic time in exchange for smaller peerlists and routing tables.

1.5 Churn and Fault Tolerance

DHTs can choose various ways to implement fault tolerance. First, a DHT must have some mechanism for neighbors to take over for node failure (although it doesn't have to). Next is peerlist maintenance. How are failed lookups handled? How do you handle churn?

Reactive vs periodic

1.5.1 Design Considerations of Churn

Any comparison of DHTs have to be done with respect to churn. Any analysis that focuses on a unchanging network is sorely lacking (unless the DHT is designed to work solely in an essentially static network. [2] evaluates DHTs in terms of cost vs performance, cost being the cost of keeping the routing tables up to date and performance being latency, not hop count. [2] examined Tapestry, Chord, Kademlia, and Kelips and simulated a size1024 network in a $2^6 4$ keyspace.

They found no optimal configuration of DHT settings, but for a given cost, there is an optimal latency, and vice-versa. This is represented by convex hulls on a graph (Chord had the best looking convex hull, but it should be noted among the authors are people involved in Chord). The authors tested each parameter to find the effect each had on the cost/performance ratio.

Chord was found to use bandwidth efficiently due to stabilizing successors/neighbors rather than peers/fingers (this is good behavior for the simulation, as this means while there may be more hops, there won't be penalty from trying to talk to a dead node). In other words, because Chord has two maintenance procedures, one that makes sure the network remains healthy and the other that periodically makes sure shortcuts are properly spaced, the network maintains a high level of correctness, compared to the other three.

Increasing the number of parallel lookups from Kademlia decreased the latency, but increased traffic. Increasing the maintenance rate in Kademlia increased the traffic but produced no noticeable improvement on lookup time. This is because more dead entries are cleaned from the routing table, but the parallel lookup prevented bad entries from having a detrimental impact in the first place. "Base" was not varied in Kademlia.

Chord modifications:

Responsibility falls to the predecessor rather than successor. (uninteresting) When looking for a node to put in finger f, Chord first finds the node appropriate for that entry, then pings it and it's successors. The fastest node is put into that finger. (interesting) There is a "base" parameter, which is different than the base used to describe the size of the keyspace; a base of b means each node holds $(b-1)(log_b n)$ fingers. The network being tested is size 1024, but in a $2^6 4$ keyspace; Chord would typically use 64 fingers. Unsurprisingly and not discussed, as $(b-1)(log_b n)$ approaches 64, the shape of the convex hull improves and slowly degrades as b increases, as unnecessary cost is introduced (uninteresting but necessary for the paper's conclusion).

1.5.2 Induced Churn

Other authors have examined using manually causing churn to help with network performance. Inducing churn keeps the network in continuous flux, making it hard to keep tables poisoned. This isn't a good idea where data needs to be stored, but for anything else, like query processing and monitoring, there's considerable benefits.

The motivation of this paper was to create a defense against route poisoning that didn't rely on a trusted central authority and create a defence against eclipse attacks that helped boost performance. It expands on the idea of keeping two routing tables, one that is optimized for speed, the other for robustness and security.

The goal is to keep the average level of table poisoning down. Rather than using the secure table as a fallback, the optimized table is always used, but periodically resets to the secure table. Optimizing the table then starts again.

Furthermore, nodes choose a new ID periodically, thus making the network more unpredictable to the attacker. The authors suggest adding a deterministic nocne to the ip/port hashing (and they seem to describe the measure that Brendan described to to predict where a node is, but there's no use of public keys, as far as I can tell)

Their implementation is called maewlstrom, and it is built on top of the Bamboo DHT.

1.6 Security

I believe there are two very broad categories for thwarting adversaries: a) preventing malicious nodes from joining the DHT b) preventing what malicious nodes can do when they have joined the DHT. The two are intractably mixed. In the vast majority of cases, there is no way to prevent all possible malicious nodes from joining, so steps must be taken to mitigate the actions of malicious nodes in the network so that a single node or fraction of nodes cannot destroy the network. On the flip side, no matter what limits you place on individual nodes, if attacker can become the majority/plurality of the network, he gets to do whatever he wants.

The problem of detection might be considered a third category; after all, once you detect a malicious node or attempt to join, you can act on that information (hopefully).

All of the below is from the following survey [4]

So first off, what are general statements that are true about all DHTs? DHTs effectively act as a decentralized lookup service. Or a routing service. It really depends. But in essence each member is assigned a key, used as an identifier and for determining responsibility. DHTs can be quite large. DHTs have to load balance. Each node will (probably) only know a small subset of nodes in the network. DHTs have to deal with churn (flash crowds too but that's solved by IRM and other replication techniques) [4] says any distributed system is vulnerable to DDOS and exploits specific to a particular implementation (duh), but in particular note these two closely related exploits.

Sybil (creating malicious nodes) Eclipse (isolating real peers)

There is close overlap between these two (a large enough Sybil attack become an Eclipse attack) and both attacks are just about establishing control of the network so an actual attack can be carried out.

1.6.1 Sybil Attacks

The Sybil attack is basically one of ChordReduce's advantages, but turned up to eleven to be made into an attack. So, in ChordReduce, we said that basically if we wanted a node to do more work, it creates another instance of the program on a different port. Since we hashed the IP address and the port to get the ID of the node, we could create an arbitrary number of extra nodes! Now imagine you're an attack and you know that...

The Sybil attack was defined in 2002. Basically, if there's no way to guarantee that a single node in the network is a one-to-one mapping to some device in the outside world, we can just create as many nodes as we want with a single device. A DHT can operate when a small fraction of the nodes in the network are malicious, but this is a bit more than a small fraction.

Named after poor Sybil, a case of dissociative identity.

What Can the Sybil Attack Do

Once a Sybil attack has been launched, the adversary effectively controls the network. He could insert false records into routing table of legitimate nodes. I could just drop all traffic routed to me.

Prevention

Sybil attacks can be prevented by ensuring each node represents only a single physical entity (this naturally can limit some load-balancing capabilities of the DHT). There's several categories for Sybil defences proposed by [4]. I'll list I found pertinent.

The first and allegedly most successful is a centralized authority. Castro et al says the only way is to exclusively generate node IDs using a trusted CA (hey, didn't we propose a distributed trusted CA using a blockchain?). Certificates bind public key and ip to an ID. This doesn't work from some DHTs. If nodes and responsibility are defined by a coordinate, then it works great. If responsibility is defined by a space (say like a Voronoi region), then it's not so great. [4] states that certificates are the best defence, but then it becomes a question of preventing the adversary from gaining legitimate certificates. The best solutions they could come up with for that are a) charge money b) map meatspace ids to to node ids in a preexisting network. The big problem from our point of view is a centralized resource is single point of failure/obvious target. Take the CA down, and the network will fall apart/nodes can't join. Or even worse, an adversary could be your CA ("When confronted with an impregnable fortress, well-garrisoned and well-stocked with provisions - endeavour to be the man on the inside." - Terry pratchett).

The obvious step after that is a decentralized registration approach. This has issues stemming from the limited view of the network and knowing who to trust. (this is a problem we can solve/ improve upon)

A couple of schemes have been suggested to do away with hashing IP addresses for IDs and instead look at using router ip address and mac addresses. The original Pastry paper (before the Sybil was defined) mentioned possibly using the hash of a public key.

A couple of authors suggest incorporating puzzles: Borisov et al and Rowaihy et al. Borosov suggested that nodes need to solve hash values during the ping period. Obviously not so hard for a single node, but it prevents the number of nodes a single computer can impersonate. Rowaihy suggested a hierarchical tree structure with a root and a few trusted members. Prospective nodes solve hashes for leaf nodes and work their way up the parents, solving more puzzles, until authenticated by the root. (but depending on the difficulty and the hash algorithm involved, this might be defunct because of Bitcoin. There's cheap and powerful ASICS for SHA right now. Solving for hash values always makes me think of Bitcoin mining)

1.6.2 Eclipse Attacks

The Eclipse attack is very much related to the Sybil attack. The goal of Eclipse attack is to infect the peer lists (fingers) of a sufficient fraction of nodes, such that the vast majority of messages will be routed to malicious nodes. Naturally a large enough Sybil attack is going to cause an Eclipse attack. [4] states that networks with well defined rules for joining peers, such as Chord, are naturally immune to Eclipse attacks, although well defined is not... well, well-defined in the paper. Based on the reading I have done, I take it to mean that because Chord doesn't construct its finger table from info provided by other nodes in a broadcast, but from network queries . Although it was not examined, due to Symphony's nature of choosing fingers using a random distribution, it should be immune in the same manner.

An Eclipse attack begins with a compromised or malicious node (we'll just use malicious) [1]. In an open system, like the vast majority of use cases for DHTs, it is very reasonable to assume this can occur. Malicious nodes can only affect messages that are sent or forwarded to them, so the adversaries goal is (re)direct the majority of the overlay's traffic to malicious nodes, preventing the normal/good nodes from communicating without routing through a malicious node. Thus, the malicious nodes eclipse the good nodes.

The redirection can be done in a few ways. First and most blunt is via a Sybil attack. If the adversary creates enough virtual nodes, the majority of the network traffic will have to be directed to malicious nodes. This attack works wherever a Sybil attack works (Note: this may be a potential misclassification on my part; I have to see how more papers treat the difference between Sybils and Eclipse.).

The second type of attack exploits the routing table maintenance of DHTs that use a "gossip"- based approach. The malicious nodes exploit the peer selection process and poison the routing tables of good nodes so that traffic will be routed to malicious nodes. This could be done by falsely responding to queries or by spoofing messages and impersonating other nides. Depending the specifics of the maintenance and discovery procedures, the process could cascade with nodes unwittingly poisoning one another.

One possible way: Most DHTs store node information in a tuple: (ID, ip, port). An attack simply feeds nodes fake IDs (chosen either statistically or in response to a request) with a malicious node's ip:port. If node ID is generated via hash(ip,port) this can be mitigated someone by verification

Chapter 2

The Four Kings

Shared Attributes

2.1 Chord

2.2 Pastry

Addressing - 128 bit ID, 0 to $2^128 - 1$, assigned randomly using hash. but thought of as base 2^b numbers (typically b=4). This creates a hypercube topology [1].

Peerlist - A routing table, neighborhood set and a leaf set. The Routing table consists of log n rows and b-columns each. The 0th row contains peers which don't share a common prefix. The 1st row contains those that share a length 1 common prefix, the 2nd a length 2 common prefix, etc. Since each ID is a base 2^b number, there is one column for each possible difference. The i,j entry of the table contains an ID that shares the same first i digits, with digit i+1 having a value of j (yes, a slot is wasted in each row).

The neighborhood set hold the ID and Ip address of the closest nodes, defined by a metric. It is not used for routing. The leaf set is used to hold the numerically closest nodes. Half of it for smaller and half for larger.

The table is populated at first by a join message to the node responsible for the joining node id. As part of the join message, nodes along the path send their routing tables. After the joining node creates it's routing table, it sends a copy to each node in the table, who then can update their routing tables. Node join cost is $O(log_2^b n)$ messages with a constant coefficient of $3 * 2^b$

When a node leaves the network, its neighbor contacts it's leaf closest to the failed node for its leaf table. That information is used to repair the leaf set. A failed routing node is replaced with another appropriate node for that slot.

Who do we actively back up to? Pastry is only about routing. PAST stores a file to the k closest nodes with id's closest to the file. This allows messages to

make it to any one of the k nodes that can respond to that file lookup (most likely the closest one to the originator)

Eclipse attack would basically work like this - when a node asks the malicious one for peer info, the malicious node replies with IDs it makes up on the spot, each bound to it's IP. These IPs would be spread throughout the keyspace so that any malicious value has a good chance of being chosen.

Routing - Forwarded to (node/peer?) whose shared prefix is longer. If no one has a better shared prefix than the current node, the message is forwarded to the closest node.

Pastry's goal is to minimize the "distance " messages travel, but distance can be defined by some metric, typically the number of hops.

The leaf set is the of nodes closest to the node in the keyspace. The neighborhood set is the of nodes closest to the node according to the distance metric Guarantees routing time is ¡ lg n in typical operation. Guarantees eventual delivery except when half of the leaf nodes fail simultaneously.

Fault Tolerance - A failed node doesn't delay routing, because Pastry's routing table allows it to just send to the next closest node. Damage to the routing table is replaced by contacting other nodes and requesting a suitable replacement

2.3 Tapestry

2.4 CAN

2.5 Chord

Chord [3] is a peer-to-peer (P2P) protocol for file sharing and distributed storage that guarantees a high probability $\log_2 N$ lookup time for a particular node or file in the network. It is highly fault-tolerant to node failures and churn, the constant joining and leaving of nodes. It scales extremely well and the network requires little maintenance to handle individual nodes. Files in the network are distributed evenly among its members.

As a distributed hash table (DHT), each member of the network and the data stored on the network is mapped to a unique m-bit key or ID, corresponding to one of 2^m locations on a ring. The ID of a node and the node itself are referred to interchangeably.

In a traditional Chord network, all messages travel in one direction - upstream, hopping from one node to another with a greater ID until it wraps around. A node in the network is responsible for all the data with keys *above or upstream* his predecessor, up through and including its own ID. If a node is responsible for some key, it is referred to being the successor of that key.

Robustness in the network is accomplished by having nodes backup their contents to their s (often 1) immediate successors, the closest nodes upstream.

This is done because when a node leaves the or fail, the most immediate successor would be responsible for the content its content.

Each node maintains a table of m shortcuts to other peers, called the finger table. The ith entry of a node n's finger table corresponds to the node that is the successor of the key $n + 2^{i-1} \mod 2^m$. Nodes route messages to the finger that is closest to the sought key without going past it, until it is received by the responsible node. This provides Chord with a highly scalable $\log_2(N)$ lookup time for any key [3].

As nodes enter and leave the ring, the nodes use their maintenance procedures to guide them into the right place and repair any links with failed nodes. Full details on Chord's maintenance cycle are beyond the scope of this paper and can be found here [3].

Chapter 3

The Challengers

3.1 Kademlia

Motivation of Kademlia was to learn better routing information with each query made (the security ramifications of gossip based routing tables being ignored, I suppose).

Addressing - Nodes and files are given n-bit identifiers, typically 160 (exactly like chord Chord). Nodes are arranged in a tree and distance between any two nodes is calculated by XORing the addressed. The XOR distance metric means that distances are symmetric, which is not the case in Chord.

A node's location in the tree given by the shortest unique prefix of it's ID. For each bit in the prefix, there would be a subtree which does not contain that node. Kademlia guarantees that the node will know at least one node in each of these subtrees.

Peerlist (using 160-bit keys)- The routing table contains 160 list, called k-buckets. Each k-bucket corresponds to a subtree not containing the node. In other words, the k-bucket is a list of nodes distance 2^i to 2^{i+1} away ($0 \leq i < 160$). The list is of maximum size k and is sorted by least recently seen (note that live nodes are never evicted). Whenever the node receives a message, it puts the sender's info in the k-bucket.

(If I'm an eclipse attacker, I just keep spamming messages of different ids, but with my own ip address and port info, or with sybils)

Routing - A unique feature of Kademlia is that it performs α parallel lookups. Kademlia recursively finds the k closest nodes to the target ID, starting with α nodes from it's own routing table. Those nodes send back the k closest nodes it knows about, and the seeking node sends a new message to the closest node it now knows.

Maintenance costs- Interestingly enough, so long as $\alpha > 1$, Kademlia receives little benefit from increased maintenance of it's peers, since multiple lookups avert the penalty that would normally occur from trying to talk to a dead node.

Implementations

3.2 The Small World

Almost everyone can to relate to this phenomenon, where a complete stranger turns out not to be so strange and that this person although distant relation to you, is closer than you thought thru your friend's relations. In the 1960's researchers took an interest in this "small world" phenomena.

Kevin Bacon

The experiment

Kleinberg If this experiment can get a message across the US, how can we model this behavior mathematically and use it for computer messages.

(this section is being done from memory at the moment) Symphony closely resembles Chord, in that both use a 1-d ring structure and that the improvements made in here can be applied to Chord

Addressing: Rather than using a keyspace of 0 to $2^n - 1$, Symphony assigns node on point between 0 and 1. This is arbitrary (although it gets me thinking, is there any advantage/statistical properties that could be exploited by making the space monic).

Peerlist: (Besides a successor and predecessor list) Symphony maintains a list of f fingers, randomly distributed according to the function defined in Kleinberg's small world routing. This yields an average lookup of $O((log^2 n)/f)$. Note that when f is log(n), Symphony's degree and lookup time is the same as most other DHTs.

Peerlist maintenance is much simpler than Chord's. Each finger is pinged periodically. When a finger fails to respond, a new finger is chosen probabilistically . The disadvantage is that there is no guarantee of a minimum routing time.

Security: As the peerlist is chosen at random

Routing is the same as Chord, for the most part, except for the 1-ahead lookup (below)

Proposed/implemented improvements not independent to Symphony Bidirectional routing 1-ahead lookup - The node retrieves the peerlist of all it's peers and uses that in routing decisions. While more expensive, there is a significant reduction in latency (not sure if a mathematically demonstrated amount, but it is experimentally demonstrated). Using this could expose a network to an Eclipse attack.

3.3 ZHT

The world is small because here, you know everyone.

Summary: ZHT uses O(n) degree to achieve O(1) hops. It gets aways with this because it's intended for deployment in an isolated environment, so churn

and updating the routing tables essentially become a non-issue. It should be taken seriously because it has been successfully implemented.

ZHT is a DHT specifically designed for high-end computing. The paper initially discusses some areas of difficulty in scaling parallel deployments. The two big issues are 1) separating the storage and computation over a network cannot scale from peta to exa. 2) Basic metadata operations must scale. Google's GFS and Yahoo's HDFS both use a centralized metadata manager that performs worse than other options.

As ZHT is specifically designed for a massively parallel deployment, certain constraints normally necessary for DHT operation are relaxed. ZHT assumes that it is deployed on a network that is fast, trusted, and churn happens at a much much lower degree. The latter two have the most significant impact to design.

As the network can safely be assumed to be trusted, security is handed off to the local firewalls and network details. The assumption is that an adversary should not be able to interact with the DHT at all.

The largest design factor is that there is no significant churn to speak of. In general, all the nodes start up when the network starts and shutdown when the network shuts down. This allows the process of building the routing tables to be done all as part of network boot (in effect, it now doesn't matter how long it takes to build them now, or what the overhead is, because that's just part of "booting"). It also implies that the routing tables will rarely change.

These features let each node in ZHT hold a routing table with an entry with each other node in the network, which would incur much greater and overwhelming level of overhead in any other typical deployment. Implementation demonstrated that the memory cost was 32 bytes per entry in the node's routing table and they achieved a goal of ¡ 1

ZHT does work with dynamic networks, where nodes join and leave (at much lower rates than normal, joining controlled by the network admin, leaving will typically only happen because of hardware failure or maintenance). Joins and departures are handled by migrating partitions, contiguous portions of keyspace, but partitioning is only discussed in abstract terms.

The other notable feature of DHT is it's append operation, which allows stored data to be concurrently modified. ZHT has been thoroughly tested using supercomputers and large clusters and the paper is more of a discussion of implementation.

3.4 Voronoi Based Schemes

3.4.1 RayNet

Beaumont et al argues that a loose structure enough for searching. Assume a d-dimension space, each dimension tied to some attribute of an object and each object identified by a unique set of values. Objects should be linked to other objects that are close in the space.

Chapter 4

MANETS

One of the key features that make a DHT ideally suited for MANETs is the WYZYG nature of the network. When you create an overlay on computers over the Internet, each hop on the overlay is actually multiple hops on the "underlay". This is never the case on MANETs.

Bibliography

[1] Tyson Condie, Varun Kacholia, Sriram Sank, Joseph M Hellerstein, and Petros Maniatis. Induced churn as shelter from routing-table poisoning. In *NDSS*, 2006.

[2] Jinyang Li, Jeremy Stribling, Thomer M Gil, Robert Morris, and M Frans Kaashoek. Comparing the performance of distributed hash tables under churn. In *Peer-to-Peer Systems III*, pages 87–99. Springer, 2005.

[3] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. *SIGCOMM Comput. Commun. Rev.*, 31:149–160, August 2001.

[4] Guido Urdaneta, Guillaume Pierre, and Maarten Van Steen. A survey of dht security techniques. *ACM Computing Surveys (CSUR)*, 43(2):8, 2011.

[5] Jie Wu. *Handbook on theoretical and algorithmic aspects of sensor, ad hoc wireless, and peer-to-peer networks.* CRC Press, 2004.