A Survey of DHT

Andrew Benjamin Rosen
Benjamin Levin
Anu Bourgeois

Abstract

This paper explores and summarizes the characteristics of different types of distributed hash tables (DHT). These characteristics include the routing table size, lookup time, and joining/leaving time as well as specific characteristics of individual types of DHTs. The types of DHTs analyzed in this paper are as follows: Chord [18], Kademlia [9], CAN (Content Addressable Network) [13], Pastry [16] and Tapestry [21] (grouped together due to a high level of similarity between the two), Symphony and Small World Routing [8], and ZHT (Zero-hop Hash Table) [7].

1 Introduction

This survey paper provides a broad overview of the concepts and implementations of Distributed Hash Tables (DHTs).

DHTs have been a vibrant area of research for the past decade, with several of the concepts dating further back [1] [9] [13] [14] [11] [18] [16]. Numerous DHTs have been developed over the years and each of the major topologies have had multiple implementation and derivatives. This is partly because the process of designing DHTs involves making trade-offs in maintenance schemes, topology, and memory, with no choice being strictly better than any other.

In practice, only two DHTs see heavy use. Chord [18] is heavily favored in academia, due to it's simplicity and being once of the earlier DHTs implemented.[1] Kademlia [9] is used for most real-world applications, most significantly BitTorrent [1].

[1] We personally believe the ability to easily draw or visualize Chord ring is a significant factor in its popularity, but we have no means of testing this easily.

2 Motivation

DHTs account for a massive amount of internet traffic, mainly through Bit Torrent (BT) usage. Bit Torrent uses Mainline DHT, which is based on the Kademlia DHT, for peer matching and accounted for 30-40% of uplink traffic in North America in 2012 [4]. While these numbers have decreased in recent year, BT still accounts for a huge amount of internet traffic making it one of the largest uses of DHTs on the planet. Given the massive scale of use of BT and Bit Torrent Sync (mainly in IoT devices), security exploits in the BT protocal would open billions of devices to potential exploits as well as their cloud service providers [19]. Research into DHTs also can provide huge performance gains, as MeNTor demonstrates by providing a 30-40% speed increase in BT transfers over wireless mesh networks with a simple protocol extension [15]. By showing the advantages and disadvantages of the commonly used types of DHTs, we hope to create a reference that other researchers can use to advance their work on DHTs.

3 What is Needed to Define a DHT

There are a couple of ways to define what a DHT is. A distributed hash table assigns each node and data object in the network a unique key. The key corresponds to the identifier for the node or the data in question, typically IP/port combination or filename. This mapping is consistent, so that even though the keys are distributed uniformly at random, the key is always the same for the same input.

DHTs are traditionally used to form a peer-to-peer overlay network, in which the DHT defines the network topology. Any member of the network can efficiently find the node that corresponds to a particular key. Data can be stored in the network and can be retrieved by finding the node that is responsible for that key.

A distributed hash table can also be thought of as a space with points (data) and Voronoi generators (nodes). A node is responsible for data that falls within its Voronoi region, which is defined by the peers closest to it. The peers that share a border for a Voronoi region are members of the node's Delaunay triangulation. Starting from any node in the network, we can find any particular node or the node responsible for a particular point in sublinear time. Regardless of the definitions, each DHT protocol needs to specify specific qualities:

Distance Metric There needs to be a way to establish how far things are

from one another. Once we have a distance metric, we define what we mean when we say a node is responsible for all data *close* to it.

Closeness Definition This definition of *closeness* is essential, since it defines what a node is responsible for and who its short hops are. The definition of closeness and distance are related but different.

We shall use Chord [18] as an example. The distance from a to b is defined as the shortest distance around the circle in either direction. However, a node is responsible for the points between its predecessor and it. The corresponding Voronoi diagram is showing in Figure 1.

However, say we were to use a more intuitive definition for closeness, where a node is responsible for the keys that were closer to it than any other node. In this case, we end up with the diagram in Figure 2.

A Midpoint Definition This defines the point which is the *minimal* equidistant point between two given points.

Peer Management Strategy This is the meat of the definition of a Distributed Hash Table. The peer management strategy includes how big peerlists are, what goes in it, and how often peers are checked to see if they are still alive. This is where almost all trade-offs are made.

Surprisingly, there is no need to define a routing strategy for individual DHTs. This is because all DHTs use the same overall routing strategy: forward the message to the known node closest to the destination. *How* routing is implemented depends on the protocol in question. Chord's routing can be implemented recursively or iteratively, while Kademlia's uses parallel iterative queries.

4 Terminology

The large number of DHTs have lead many papers to use different terms to describe congruent elements of DHTs, as some terms may make sense only in one context. Since this paper will cover multiple DHTs that use different terms, we have created a unified terminology:

key - The identifier generated by a hash function corresponding to a unique[2]

[2]Unique with extremely high probability. The probability of a hash collision is extremely low and are ignored in most formal specifications for DHTs. This could be resolved

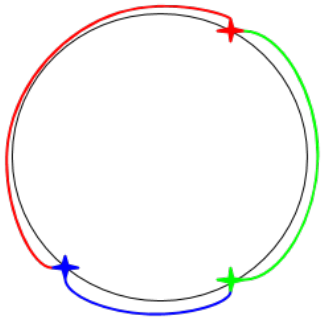

Figure 1: A Voronoi diagram for a Chord network, using Chord's definition of closest.

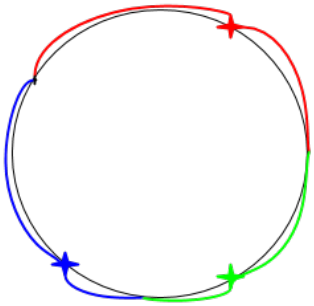

Figure 2: A Voronoi diagram for a Chord network, where closest is defined by the node being the closest in either direction.

node or file. SHA-1, which generates 160-bit hashes, is typically used as a hashing algorithm.[3]

ID - The ID is a key that corresponds to a particular node. The ID of a node and the node itself are referred to interchangeably. In this dissertation, we refer to nodes by their ID and files by their keys.

Peer - Another active member on the network. For this section, we assume that all peers are different pieces of hardware.

Peerlist - The set of all peers that a node knows about. This is sometimes referred to as the *routing table*, but certain DHTs [16] [21] overload the terminology. Any table or list of peers is a subset of the entire peerlist.

Short-hops - The subset of peers that are "closest/adjacent" to the node in the keyspace, according to the DHT's metric. In a 1-dimensional ring, such a Chord [18], this is the node's *predecessor(s)* and *successor(s)*. They may also be called *neighbors*.

Long-hops - The subset of the peerlist that the node is not adjacent to. These are sometimes referred to as fingers, long links, or shortcuts.

Root Node - The node responsible for a particular key.

Successor - Alternate name for the root node. The successor of a node is the neighbor that will assume a nodes responsibilities if that node leaves.

n **nodes** - The number of nodes in the network.

Similarly, All DHTs perform the same operations with minor variation.

`lookup(key)` - This operation finds the root node of `key`. Almost every operation on a DHT needs to leverage the `lookup` operation in some way.

---

for any file by using any number of the collision resolution strategies, such as chaining or linear probing. However, resolving a collision of two nodes is much more problematic with no canonical solution other than praying it won't happen.

[3]Due to the research into hash collisions [17], and the glut of hardware that currently exists to perform SHA hash collisions, SHA1 is being depreciated by many companies in 2017. This will undoubtedly lead to some kind of security flaw in a decade or so, when some entrepreneuring hacker figures out a way to force websites to accept a forged SHA1 key.

`put(key,value)` - Stores `value` at the root node of `key`. Unless otherwise specified, `key` is assumed be the hashkey of `value`. This assumption is broken in Tapestry.

`get(key)` - This operates like lookup, except the context is to return the value stored by a `put`. This is a subtle difference, since one could `lookup(key)` and ask the corresponding node directly. However, many implementations use backup operations and caching, which will store multiple copies of the value along the network. If we do not care which node returns the value mapped with `key`, or if it is a backup, we can express it with `get`.

`delete(key, value)` - This is self-explanatory. Typically, DHTs do not worry about key deletion and leave that option to the specific application. When DHTs do address the issue, they often assume that stored key-value pairs have a specified time-to-live, after which they are automatically removed.

On the local level, each node has to be able to *join* and perform maintenance on itself.

`join()` The join process encompasses two steps. First, the joining node needs to initialize its peerlist. It does not necessarily need a complete peerlist the moment it joins, but it must initialize one. Second, the joining node needs to inform other nodes of its existence.

Maintenance Maintenance procedures generally are either *active* or *lazy*. In active maintenance, peers are periodically pinged and are replaced when they are no longer detected. Lazy maintenance assumes that peers in the peerlist are healthy until they prove otherwise, in which case they are either replaced immediately. In general, lazy maintenance is used on everything, while active maintenance is only used on neighbors

When analyzing the DHTs in this paper, we look at the overlay's geometry, the peerlist, the `lookup` function, and how fault-tolerance is performed in the DHTs. We assume that nodes never politely leave the network but always abruptly fail, since a `leave()` operation is fairly trivial and has minimal impact.

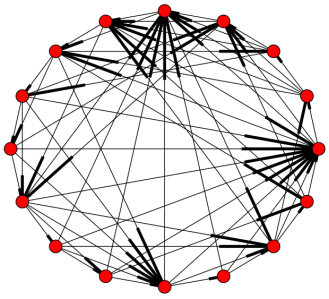

Figure 3: A Chord ring with 16 nodes. The fingers (long hop connections) are shown cutting across the ring.

5 Chord

Chord [18] is the archetypal ring-based DHT and it is impossible to create a new ring-based DHT without making some comparison to Chord. It is notable due its straightforward routing, its rules which make ownership of keys very easy to sort out, and the large number of derivatives.

Chord is extremely well known in Computer Science, and was awarded the prestigious 2011 SIGCOMM Test of Time Award. However, recent research has demonstrated that there have been no correct implementations of Chord in over a decade, and many of the properties Chord claimed to be invariants could break under certain trivial conditions [20].

Peerlist and Geometry

Chord is a 1-dimensional modular ring in which all messages travel in one direction - upstream, hopping from one node to another node with a greater ID until it wraps around. Each member of the network and the data stored within it is hashed to a unique m-bit key or ID, corresponding to one of the 2^m locations on a ring. An example Chord network is shown in Figure 3.

A node in the network is responsible for all the data with keys upstream from its predecessor's ID, up through and including its own ID. If a node is responsible for some key, it is referred to being the root or successor of that key.

Lookup and routing is performed by recursively querying nodes upstream. Querying only neighbors in this manner would take $O(n)$ time

to lookup a key.

To speedup lookups, 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 $n+2^{i-1} \mod 2^m$. During a lookup, nodes query the finger that is closest to the sought key without going past it, until it is received by the root node. Each hop essentially cuts the search space for a key in half. This provides Chord with a highly scalable $\log_2(n)$ lookup time for any key [18], with an average $\frac{1}{2}O(\log_2(n))$ number of hops.

Besides the finger tables, the peerlist includes a list of s neighbors in each direction for fault tolerance. This brings the total size of the peerlist to $log_2(2^m) + 2 \cdot s = m + 2 \cdot s$, assuming the entries are distinct.

Joining

To join the network, node n first asks n' to find `successor(`n`)`. Node n uses the information to set his successor, and maintenance will inform the other nodes of n's existence. Meanwhile, n will takeover some of the keys that his successor was responsible for.

Fault Tolerance

Robustness in the network is accomplished by having nodes backup their contents to their s 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 keys. In the case of multiple nodes failing all at once, having a successor list makes it extremely unlikely that any given stored value will be lost.

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. The process takes $O(\lg^2(n))$ messages. Full details on Chord's maintenance cycle can be found here [18].

6 Kademlia

Kademlia [9] is perhaps the most well known and most widely used DHT, as a modified version of Kademlia (Mainline DHT) is forms backbone of the BitTorrent protocol. The motivation of Kademlia was to create a way for nodes to incorporate peerlist updates with each query made.

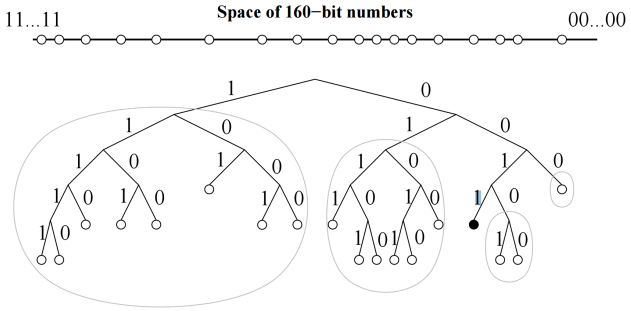

Figure 4: An example Kademlia network from the original paper [9]. The ovals are the node's k-buckets.

Peerlist and Geometry

Like Chord, Kademlia uses m-bit keys for nodes and files. However, Kademlia utilizes a binary tree-based structure, with the nodes acting as the leaves of the tree. Distance between any two nodes in the tree is calculated by XORing their IDs. The XOR distance metric means that distances are symmetric, which is not the case in Chord.

Nodes in Kademlia maintain information about the network using a routing table that contains m lists, called k-buckets. For each k-bucket contains up to k nodes that are distance 2^i to 2^{i+1}, where $0 \leq i < m$. In other words, each k-bucket corresponds to a subtree of the network not containing the node. An example network is shown in Figure 4.

Each k-bucket is maintained by a least recently seen eviction algorithm that skips live nodes. Whenever the node receives a message, it adds the sender's info to the tail of the corresponding k-bucket. If that info already exists, the info is moved to the tail.

If the k-bucket is full, the node starts pinging nodes in the list, starting at the head. As soon as a node fails to respond, that node is evicted from the list to make way for the new node at the tail.

If there are no modifications to a particular k-bucket after a long period of time, the node does a `refresh` on the k-bucket. A refresh is a `lookup` of a random key in that k-bucket.

Lookup

In most DHTs, `lookup(key)` sends a single message and returns the information of a single node. The `lookup` operation in Kademlia differs in both

respects: `lookup` is done in parallel and each node receiving a `lookup(key)` returns the k closest nodes to `key` it knows about.

A `lookup(key)` operation begins with the seeking node sending lookups in parallel to the α nodes from the appropriate k-bucket. Each of theses α nodes will asynchronously return the k closest nodes it knows closest to `key`. As lookups return their results, the node continue to send lookups until no new nodes[4] are found.

Joining

A joining node starts with a single contact and then performs a *lookup* operation on its own ID. Each step of the *lookup* operation yields new nodes for the joining node's peerlist and informs other nodes of its existence. Finally, the joining node performs a `refresh` on each k-bucket farther away than the closest node it knows of.

Fault-Tolerance

Nodes actively republish each file stored on the network each hour by rerunning the `store` command. To avoid flooding the network, two optimizations are used.

First if a node receives a `store` on a file it is holding, it assumes $k-1$ other nodes got that same command and resets the timer for that file. This means only one node republishes a file each hour. Secondly, `lookup` is not performed during a republish.

Additional fault tolerance is provided by the nature of the `store(data)` operation, which `puts` the file in the k closest nodes to the key. However, there is very little in the way of frequent and active maintenance other than what occurs during `lookup` and the other operations.

7 CAN

Unlike the previous DHTs presented in this paper, the Content Addressable Network (CAN) [13] works in a d-dimensional torus, with the entire coordinate space divided among members. A node is responsible for the keys that fall within the "zone" that it owns. Each key is hashed into some point within the geometric space.

[4] If a file being stored on the network is the objective, the `lookup` will also terminate if a node reports having that file.

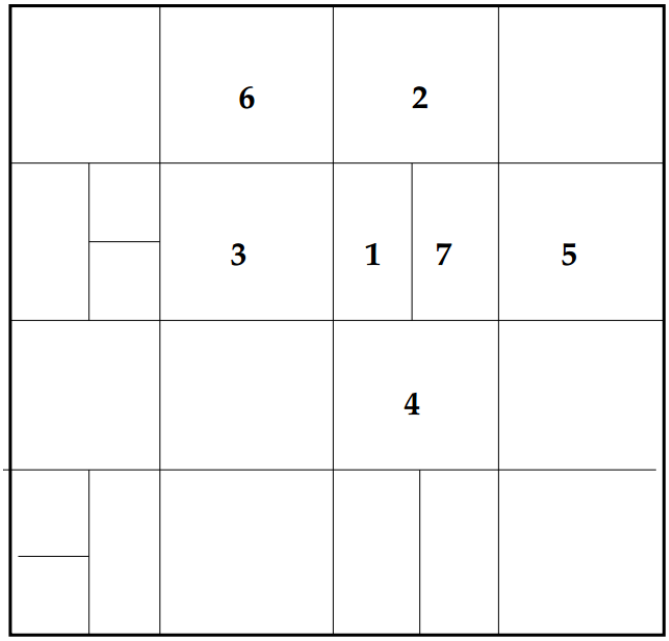

Figure 5: An example CAN network from [13].

Peerlist and Geometry

CAN uses an exceptionally simple peerlist consisting only of neighbors. Every node in the CAN network is assigned a geometric region in the coordinate space and each node maintains a routing table consisting each node that borders the node's region. An example CAN network is shown in Figure 5

The size of the routing table is a function of the number of dimensions, $O(d)$. The lower bound on the routing tables size in a populated network (eg, a network with at least $2d$ nodes) is $\Omega(2d)$. This is obtained by looking at each axis, where there is at least one node bordering each end of the axis. The size of the routing table can grow as more nodes join and the space gets further divided; however, maintenance algorithms prevent the regions from becoming too fragmented.

Lookup

As previously mentioned, each node maintains a routing table corresponding to their neighbors, those nodes it shares a face with. Each hop forwards the lookup to the neighbor closest to the destination, until it comes to the

responsible node. In a space that is evenly divided among n nodes, this simple routing scheme uses only $2 \cdot d$ space while giving average path length of $\frac{d}{4} \cdot n^{\frac{1}{d}}$. The overall lookup time of in CAN is bounded by $O(n^{\frac{1}{d}})$ hops[5].

If a node encounters a failure during lookup, the node simply chooses the next best path. However, if lookups occur before a node can recover from damage inflicted by churn, it is possible for the greedy lookup to fail. The fallback method is to use an expanding ring search until a candidate is found, which recommences greedy forwarding.

Joining

Joining works by splitting the geometric space between nodes. If node n with location P wishes to join the network, it contacts a member of the node to find the node m currently responsible for location P. Node n informs m that it is joining and they divide m's region such that each becomes responsible for half.

Once the new zones have been defined, n and m create its routing table from m and its former neighbors. These nodes are then informed of the changes that just occurred and update their tables. As a result, the join operation affects only $O(d)$ nodes. More details on this splitting process can be found in CAN's original paper [13].

Repairing

A node in a DHT that notifies its neighbors that its leaves usually has minimal impact to the network and in this is true for most cases in CAN. A leaving node, f, simply hands over its zone to one of its neighbors of the same size, which merges the two zones together. Minor complications occur if this is not possible, when there is no equally-sized neighbor. In this case, f hands its zone to its smallest neighbor, who must wait for this fragmentation to be fixed.

Unplanned failures are also relatively simple to deal with. Each node broadcasts a heartbeat to its neighbors, containing its and its neighbors' coordinates. If a node fails to hear a heartbeat from f after a number of cycles, it assumes f must have failed and begins a `takeover` countdown. When this countdown ends, the node broadcasts[6] a `takeover` message in

[5] Around the same time CAN was being developed, Kleinberg was doing research into small world networks [5]. He proved similar properties for lattice networks with a single shortcut. What makes this network remarkable is lack of shortcuts.

[6] This message is sent to all of f's neighbors.

an attempt to claim f's space. This message contains the node's volume. When a node receives a `takeover` message, it either cancels the countdown or, if the node's zone is smaller than the broadcaster's, responds with its own `takeover`.

The general rule of thumb for node failures in CAN is that the neighbor with the smallest zone takes over the zone of the failed node. This rule leads to quick recoveries that affect only $O(d)$ nodes, but requires a zone reassignment algorithm to remove the fragmentation that occurs from `takeovers`.

To summarize, a failed node is detected almost immediately, and recovery occurs extremely quickly, but fragmentation must be fixed by a maintenance algorithm.

8 Pastry

Pastry [16] and Tapestry [21] are extremely similar use a prefix-based routing mechnism introduced by Plaxton et al. [12]. In Pastry and Tapestry, each key is encoded as a base 2^b number (typically $b = 4$ in Pastry, which yields easily readable hexadecimal). The resulting peerlist best resembles a hypercube topology [2], with each node being a vertice of the hypercube.

One notable feature of Pastry is the incorperation of a proximity metric. The peerlist uses IDs that are close to the node according to this metric.

Peerlist

Pastry's peerlist consists of three components: the routing table, a leaf set, and a neighborhood set. The routing table consists of $\log_{2^b}(n)$ rows with $2^b - 1$ entries per row. The ith level of the routing table correspond to the peers with that match first i digits of the example nodes ID.

Thus, the 0th row contains peers which don't share a common prefix with the node, 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 entry for each of the $2^b - 1$ possible differences.

For example, let is consider a node 05AF in system where $b = 4$ and the hexadecimal keyspace ranges from 0000 to FFFF.

- 1322 would be an appropriate peer for the 1st entry of level 0.
- 0AF2 would be an appropriate peer for the 10th[7] entry of level 1.

[7] 0 is the 0th level.

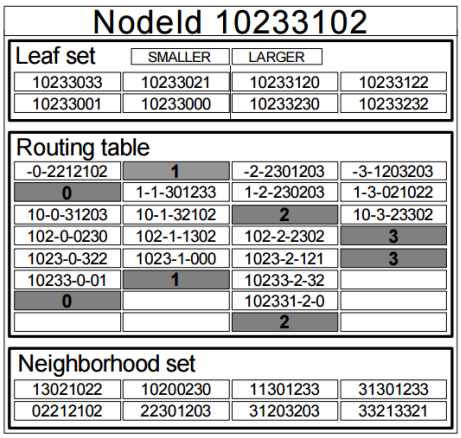

Figure 6: An example peerlist for a node in Pastry [16].

- 09AA would be an appropriate peer for the 9th entry of level 1.
- 05F2 would be an appropriate peer for the 2nd entry of level 3.

The leaf set is used to hold the L nodes with the numerically closest IDs; half of it for smaller IDs and half for the larger. A typical value for L is 2^b or 2^{b+1}. The leaf set is used for routing when the destination key is close to the current node's ID. The neighborhood set contains the L closest nodes, as defined by some proximity metric. It, however, is generally not used for routing. Figure 6 shows an example peerlist of a node in PAST.

Lookup

The `lookup` operation is a fairly straightforward recursive operation. The `lookup(key)` terminates when the `key` is falls within the range of the leaf set, which are the nodes *numerically* closest to the current node. In this case, the destination will be one of the leaf set, or the current node.

If the destination node is not immediately apparent, the node uses its routing table to select the next node. The node looks at the length l shared prefix, at examines the lth row of its routing table. From this row, the `lookup` continues with the entry that matches at least another digit of the prefix. In the case that this entry does not exist or has failed, the `lookup` continues from the closest ID chosen from the entire peerlist. This process is described by Algorithm 1. Lookup is expected to take $\lceil \log_{2^b} \rceil$, as each hop along the routing table reduces the search space by $\frac{1}{2^b}$.

Algorithm 1 Pastry lookup algorithm

```
Let L be the routing
function LOOKUP(key)
    if key is in the range of the leaf set  then
        destination is closest ID in the leaf set or self
    else
        next ← entry from routing table that matches ≥ 1 more digit
        if next ≠ null then
            forward to next
        else
            forward to the closest ID from the entire peerlist
        end if
    end if
end function
```

Joining

To join the network, node J sends a `join` message to A, some node that is close according to the proximity metric. The `join` message is forwarded along like a `lookup` to the root of X, which we'll call *root*. Each node that received the `join` sends a copy of the their peerlist to J.

The leaf set is constructed from copying *root*'s leaf set, while ith row in the routing table routing table is copied from the ith node contacted along the `join`. The neighborhood set is copied from A's neighborhood set, as `join` predicates that A be close to J. This means A's neighborhood set would be close to A.

After the joining node creates its peerlist, it sends a copy to each node in the table, who then can update their routing tables. The cost of a `join` is $O(log_2^b n)$ messages, with a constant coefficient of $3 * 2^b$

Fault Tolerance

Pastry lazily repairs its leaf set and routing table. When node from the leaf set fails, the node contacts the node with largest or smallest ID (depending if the failed node ID was smaller or larger respectively) in the leaf set. That node returns a copy of its leaf set, and the node replaces the failed entry. If the failed node is in the routing table, the node contacts a node with an entry in the same row as the failed node for a replacement.

Members of the neighborhood set are actively checked. If a member of the neighborhood set is unresponsive, the node obtains a copy of another

entry's neighborhood set and repairs from a selection.

9 Symphony and Small World Routing

Symphony [8] is a $1d$ ring-based DHT similar to Chord [18], but is constructed using the properties of small world networks [5]. Small world networks owe their name to a phenomena observed by psychologists in the late 1960's.

Subjects in experiments were to route a postal message to a target person; for example the wife of a Cambridge divinity student in one experiment and a Boston stockbroker in another [10]. The messages were only to be routed by forwarding them to a friend they thought most likely to know the target. Of the messages that successfully made their way to the destination, the average path length from a subject to a participant was only 5 hops.

This lead to research investigating creating a network with randomly distributed links, but with a efficient lookup time. Kleinberg [6] showed that in a 2-dimensional lattice network, nodes could route messages in $O(\log^2 n)$ hops using only their neighbors and a single randomly chosen[8] finger. In other words, $O(\log^2 n)$ lookup is achievable with a $O(1)$ sized routing table.

Peerlist

Rather than the 2-dimensional lattice used by Kleinberg, Symphony uses a 1-dimensional ring[9] like Chord. Symphony assigns m-bit keys to the modular unit interval $[0, 1)$, instead of using a keyspace ranging from 0 to $2^n - 1$. This location is found with $\frac{hashkey}{2^m}$. This is arbitrary from a design standpoint, but makes choosing from a random distribution simpler.

Nodes know both their immediate predecessor and successor, much like in Chord. Nodes also keep track of some $k \geq 1$ fingers, but, unlike in Chord, these fingers are chosen at random. These fingers are chosen from a probability distribution corresponding to the expression $e^{ln(n)+(rand48()-1.0)}$, where n is the number of nodes in the network and `rand48()` is a C function that generates a random float?double between 0.0 and 1.0. Because n os difficult to compute due to the changing nature of P2P networks, each node uses an approximation is used based on the distance between themselves and their neighbors.

[8] Randomly chosen from a specified distribution.

[9] This is technically a 1-dimensional lattice.

A final feature of note is that links in Symphony are bidirectional. Thus, if a node creates a finger to a peer, that peer creates a, so nodes in Symphony have a grand total of $2k$ fingers.

Joining and Fault Tolerance

The joining and fault tolerance processes in Symphony are extremely straightforward. After determining its ID, a joining node asks a member to find the root node for its ID. The joining node integrates itself in between its predecessor and successor and then randomly generates its fingers.

Failures of immediate neighbors are handled by use of successor and predecessor lists Failures for fingers are handled lazily and are replaced by another randomly generated link when a failure is detected.

10 ZHT

One of the major assumptions of DHT design is that churn is a significant factor, which requires constant maintenance to handle. A consequence of this assumption is that nodes only store a small subset of the entire network to route to. Storing the entire network is not scalable for the vast majority of distributed systems due to bandwidth constraints and communication overhead incurred by the constant joining and leaving of nodes.

In a system that does not expect churn, the memory and bandwidth costs for each node to keep a full copy of the routing table are minimal. An example of this would be a data center or a cluster built for higher-performance computing, where churn would overwhelmingly be the result of hardware failure, rather than users quitting.

ZHT [7] is an example of such a system, as is Amazon's Dynamo [3]. ZHT is a "zero-hop hash table," which takes advantage of the fact that nodes in High-End Computing environments have a predictable lifetime. Nodes are created when a job begins and are removed when a job ends. This property allows ZHT to `lookup` in $O(1)$ time.

Peerlist

ZHT operates in a 64-bit ring, for a total of $N = 2^{64}$ addresses. ZHT places a hard limit of n on the maximum number of physical nodes in the network, which means the network has n partitions of $\frac{N}{n} = \frac{2^{64}}{n}$ keys. The partitions are evenly divided along the network.

The network consists of k physical nodes which each are running at least one instance (virtual nodes) of ZHT, with a combined total of i . Each instance is responsible for some span of partitions in the ring.

Each node maintains a complete list of all nodes in the network, which do not have to be updated very often due to the lack of or very low levels of churn. The memory cost is extremely low. Each instance has a 10MB footprint, and each entry for the membership table takes only 32 bytes per node. This means routing takes anywhere between 0 to 2 hops (explained below).

Joining

ZHT operates under a static or dynamic membership. In a static membership, no nodes will be joining the network once the network has been bootstrapped. Nodes can join at any time when ZHT is using dynamic membership.

To join, the joiner asks a random member for a copy of the peerlist The joiner can then determine which node is the most heavily overloaded. The joiner chooses an address in the network to take over partitions from that node.

Fault Tolerance

Fault tolerance exists to handle only hardware failure or planned departures from the network. Nodes backup their data to their neighbors.

11 Summary

We have seen that there are a wide variety of distributed hash tables, but they have some clearly defined characteristics that bind them all together. Table 1 summarizes the information presented in this paper.

References

[1] Bram Cohen. Incentives build robustness in bittorrent. In *Workshop on Economics of Peer-to-Peer systems*, volume 6, pages 68–72, 2003.

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

DHT	Routing Table Size	Lookup Time	Join/Leave	Comments
Chord [18]	$O(\log n)$, maximum $m + 2s$	$O(\log n)$, avg $(\frac{1}{2}\log n)$	$< O(\log n^2)$ total messages	m = keysize in bits, s is neighbors in 1 direction
Kademlia [9]	$O(\log n)$, maximum $m \cdot k$	$(\lceil \log n \rceil) + c$	$O(\log(n))$	This is without considering optimization
CAN [13]	$\Omega(2d)$	$O(n^{\frac{1}{d}})$, average $\frac{d}{4} \cdot n^{\frac{1}{d}}$	Affects $O(d)$ nodes	d is the number of dimensions
Plaxton-based DHTs, Pastry [16], Tapestry [21]	$O(\log_\beta n)$	$O(\lceil \log_{2^\beta} \rceil)$	$O(\log_\beta n)$	NodeIDs are base β numbers
Symphony [8]	$2k + 2$	average $O(\frac{1}{k}\log^2 n)$	$O(\log^2 n)$ messages, constant < 1	$k \geq 1$, fingers are chosen at random
ZHT [7]	$O(n)$	$O(1)$	$O(n)$	Assumes an extremely low churn
VHash	$\Omega(3d+1)+O((3d+1)^2)$	$O(\sqrt[d]{n})$ hops	$3d + 1$	approximates regions, hops are based least latency

Table 1: A comparison of and summary of the various DHTs.

[3] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 205–220. ACM, 2007.

[4] R. Farahbakhsh, N. Crespi, Á. Cuevas, R. Cuevas, and R. González. Understanding the evolution of multimedia content in the internet through bittorrent glasses. *IEEE Network*, 27(6):80–88, November 2013.

[5] Jon Kleinberg. The small-world phenomenon: An algorithmic perspective. In *Proceedings of the thirty-second annual ACM symposium on Theory of computing*, pages 163–170. ACM, 2000.

[6] Jon M Kleinberg. Navigation in a small world. *Nature*, 406(6798):845–845, 2000.

[7] Tonglin Li, Xiaobing Zhou, Kevin Brandstatter, Dongfang Zhao, Ke Wang, Anupam Rajendran, Zhao Zhang, and Ioan Raicu. Zht: A light-weight reliable persistent dynamic scalable zero-hop distributed hash table. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 775–787. IEEE, 2013.

[8] Gurmeet Singh Manku, Mayank Bawa, Prabhakar Raghavan, et al. Symphony: Distributed Hashing in a Small World. In *USENIX Symposium on Internet Technologies and Systems*, page 10, 2003.

[9] Petar Maymounkov and David Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. In *Peer-to-Peer Systems*, pages 53–65. Springer, 2002.

[10] Stanley Milgram. The small world problem. *Psychology today*, 2(1):60–67, 1967.

[11] C. Greg Plaxton, Rajmohan Rajaraman, and Andréa W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '97, pages 311–320, New York, NY, USA, 1997. ACM.

[12] C Greg Plaxton, Rajmohan Rajaraman, and Andrea W Richa. Accessing nearby copies of replicated objects in a distributed environment. *Theory of Computing Systems*, 32(3):241–280, 1999.

[13] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. 2001.

[14] Sylvia Ratnasamy, Brad Karp, Li Yin, Fang Yu, Deborah Estrin, Ramesh Govindan, and Scott Shenker. Ght: a geographic hash table for data-centric storage. In *Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, pages 78–87. ACM, 2002.

[15] Michael Rethfeldt, Benjamin Beichler, Peter Danielis, Felix Uster, Christian Haubelt, and Dirk Timmermann. Mentor: A wireless-mesh-network-aware data dissemination overlay based on bittorrent. *Ad Hoc Networks*, 79:146 – 159, 2018.

[16] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware 2001*, pages 329–350. Springer, 2001.

[17] Marc Martinus Jacobus Stevens et al. *Attacks on hash functions and applications*. Mathematical Institute, Faculty of Science, Leiden University, 2012.

[18] 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.

[19] T.Y. Yang, A. Dehghantanha, K.-K.R. Choo, and Z. Muda. Forensic investigation of p2p cloud storage: Bittorrent sync as a case study. *Computers & Electrical Engineering*, 58:350–363, February 2017. © 2017 Elsevier. This is an author produced version of a paper subsequently published in Computers and Electrical Engineering. Uploaded in accordance with the publisher's self-archiving policy. Article available under the terms of the CC-BY-NC-ND licence (https://creativecommons.org/licenses/by-nc-nd/4.0/).

[20] Pamela Zave. Using lightweight modeling to understand chord. *ACM SIGCOMM Computer Communication Review*, 42(2):49–57, 2012.

[21] Ben Y Zhao, Ling Huang, Jeremy Stribling, Sean C Rhea, Anthony D Joseph, and John D Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *Selected Areas in Communications, IEEE Journal on*, 22(1):41–53, 2004.