UrDHT: A Unified Model for Distributed Hash Tables

Andrew Rosen   Brendan Benshoof   Robert W. Harrison   Anu G. Bourgeois

Department of Computer Science
Georgia State University
Atlanta, Georgia

rosen@cs.gsu.edu   bbenshoof@cs.gsu.edu   rharrison@cs.gsu.edu   anu@cs.gsu.edu

Abstract—**Distributed Hash Tables (DHTs) have an inherent set qualities, such as greedy routing, maintaining lists of peers which define the topology, and form an overlay network. Rather than having a developer be concerned with the details of a given DHT, we have constructed a new framework, UrDHT, that generalizes the functionality and implementation of various DHTs.**

UrDHT is an abstract model of a Distributed Hash Table. It maps the topologies of DHTs to the primal-dual problem of Voronoi Tessellation and Delaunay Triangulation. By completing a few simple functions, a developer can implement the topology of any DHT in any arbitrary space using UrDHT. For example, we implemented a DHT operating in a hyperbolic space, a previously unexplored nontrivial metric space with potential applications.

I. INTRODUCTION

UrDHT is an abstract model of a DHT which solves a number of problems. First, it is a unified and cohesive model for creating distributed hash tables and P2P applications based on DHTs. Second, it provides a single network for bootstrapping distributed applications.

Distributed Hash Tables have been the catalyst for the creation of many P2P applications. Among these are Redis [1], Freenet [4], and, most notably, BitTorrent [5]. All DHTs use functionally similar protocols to perform lookup, storage, and retrieval operations. Despite this, no one has created a cohesive formal specification for building a DHT.

Our primary motivation for this project was to create an abstracted model for Distributed Hash Tables based on observations we made during previous research [2]. We found that all DHTs can cleanly be mapped to the primal-dual problems of Voronoi Tessellation and Delaunay Triangulation.

UrDHT directly builds its topology using this insight. It uses a greedy distributed heuristic for approximating Delaunay triangulations. UrDHT is our specification of an abstract DHT, which can be used to build many different DHTs. We found that we could reproduce the topology of different DHTs by defining a selection heuristic and rejection algorithm for the space. For every DHT we tried, our greedy approximation of Delaunay Triangulation using a distance and midpoint function for the space produced a stable DHT. This works in non-Euclidean spaces such as XOR (Kademlia) or even a hyperbolic space represented by a Poincarè disc.

The end result is not only do we have an abstract model of DHTs, we have a simple framework that developers can use to quickly create new distributed applications. This simple framework allows generation of internally consistent implementations of different DHTs that can have their performance rigorously compared.

Another poorly addressed issue with DHTs and DHT-based P2P applications we wish to address with UrDHT is the what we have termed the *bootstrapping problem*. Simply put, a node can only join the network if it knows another node that is already a member of the network it is trying to join.

The general way this works is by having a potential user manually look up at a centralized source, such as the project or application's website, the bootstrapping information. It is a philosophical conflict requiring a distributed application using a centralized source of information to build a distributed network.

UrDHT has the potential to be a distributed source for bootstrapping information for other distributed networks. This would make new distributed applications easier to adopt by creating a network to bootstrap *other networks*. UrDHT does this by making it easy to add other networks as a service.

To summarize our contributions:

- We give a formal specification for what needs to be defined in order to create a functioning DHT. While there has long existed a well known protocol

for distributed hash tables, these define what a DHT needs to be able to do. It does not describe what a DHT is.

We show that DHTs cleanly map to the primal-dual problem of Delaunay triangulations and Voronoi tessellations. We list a set of simple functions that, once defined, allow our Distributed Greedy Voronoi Heuristic (DGVH) to be run in any space, creating a DHT overlay for that space (Section II).

- We present UrDHT as an abstract DHT and show how a developer can tweak the functions we defined to create an arbitrary new DHT topology (Section III).
- We show how to reproduce the topology of Chord and Kademlia using UrDHT, which we call Ur-Chord and UrKademlia. We also implement a DHT in a hyperbolic space represented by a Poincarè disc (Section IV).
- We conduct experiments showing that UrChord sufficiently approximates a correct implementation of Chord (Section V).
- We discuss the ramifications of our work and what future work is available.

II. WHAT DEFINES A DHT

A distributed hash table is usually defined by its protocol; in other words, what it can do. Nodes and data in a distributed hash table are assigned unique[1] keys via a consistent hashing algorithm. To make it easier to grok the context, we will call the key associated with a node its ID and refer to nodes and their IDs interchangeably.

A DHT can perform the `lookup(key)`, `get(key)`, and `store(key, value)` operations.[2] The `lookup` operation returns the node responsible for a queried key, `get` returns the value stored with that key with the `store` function.

However, this is what a DHT *does*, viewing the DHT as a black box, not what a DHT *is* and needs to be implemented. We show that Distributed Hash Tables are just Voronoi tessellations and Delaunay triangulation.

A. DHT Components

The following functions need to be defined in order for nodes to perform lookup operations and determine responsibility.

---

[1]Unique with astronomically high probability, given a large enough consistent hash algorithm.

[2]There is typically a *delete(key)* operation defined too, but it is not strictly necessary.

- **A `distance` function** - This measures distance in the overlay formed by the Distributed Hash Table. In most DHTs, the distance in the overlay has no correlation with real-world attributes.
- **A `midpoint` function** - This calculates the minimally equidistant point between two given point. The midpoint is required for Delaunay triangulation calculation. In some spaces, such as Kademlia's XOR metric space, this can be tricky to calculate.
- **An `responsibility` definition** This defines the range of keys a node is responsible for. Not every DHT defines which node is responsible for particular keys in the same way. For example, nodes in Kademlia are responsible for the keys closest to themselves, while in Chord, nodes are responsible for the keys falling between themselves and the preceding node.

A DHT also needs a strategy to organize and maintain two lists of of other nodes in the network: *short peers* and *long peers*. Short peers are the set of peers that define the topology of the network and guarantee that greedy routing works.

Long peers allow the DHT to achieve a better than linear lookup time, typically $\log(n)$, where n is the size of the network.

Interestingly, despite the diversity of DHT topologies, all DHTs use functionally identical greedy routing algorithms (Algorithm 1):

Algorithm 1 The DHT Generic Routing algorithm

1: Given node n and a message being sent to key
2: **function** n.lookup(key)
3: **if** If $key \in n$'s range of responsibility **then**
4:     **return** n
5: **end if**
6: **if** One of n's short peers are responsible for key **then**
7:     **return** the responsible node
8: **end if**
9: $candidates = short\_peers + long\_peers$
10: $next \leftarrow \min(n\text{.distance}(candidates, key))$
11: **return** $next$.lookup(key)

If I, the node, am responsible for the key, I return myself. Otherwise, if I know who is responsible for this key, I return that node. Finally, if that is not the case, I forward this query to the node I know with shortest

distance from the node to the desired key.[3]

Between individual DHTs, this algorithm might be implemented either recursively or iteratively. It will certainly have differences in how a node handles errors, such as how to handle connecting to a failed node which no longer exists. This algorithm may possibly be run in parallel, such as in Kademlia [10]. Despite this, the base greedy algorithm is always the same between implementations.

The final component is a consistent hashing function. This function must generate keys large enough to make the chances of a hash collision nigh impossible.

B. DHTs, Delaunay Triangulation, and Voronoi Tesselation

With the following components of a DHT defined above we can now show the relationship between DHTs and the primal-dual problems of Delaunay Triangulation and Voronoi Tessellation.

We can map a given node's ID to a point in a space, the range of keys a node is responsible for to that node's Voronoi region, and the set of short peers to the Delaunay triangulation. Thus, if we can calculate the Delaunay triangulation between nodes in a DHT, we have a generalized means of created the overlay network.

This can be done with any algorithm that calculates the Delaunay Triangulation. Computing the Delaunay Triangulation and/or the Voronoi Tesselation of a set of points is a well analyzed problem. Many algorithms exist which efficiently compute a Voronoi tessellation for a given set of points on a plane, such as Fortune's sweep line algorithm [6]. However, many DHTs are distributed and many of the algorithms to compute Delaunay Triangulation and/or Delaunay Triangulation are unsuited to a distributed environment.

In addition, the computation cost increases when we move into spaces with greater than two dimensions. In general, finding the Delaunay Triangulation of n points in a space with d dimensions takes $O(n^{\frac{2d-1}{d}})$ time [13].

Is there an algorithm we can use to efficiently calculate Delaunay Triangulations for a distributed system in an arbitrary space? We created an algorithm call the Distributed Greedy Voronoi Heuristic (DGVH), shown in Algorithm 2 and explained below [2].

A node takes a set of peers and uses DGVH to determine which of these correspond to Delaunay peers. From the perspective of the node, the candidates are the

---

Algorithm 2 Distributed Greedy Voronoi Heuristic

1: Given node n and its list of $candidates$.
2: Given the minimum $table\_size$
3: $short\_peers \leftarrow$ empty set that will contain n's one-hop peers
4: $long\_peers \leftarrow$ empty set that will contain n's peers further than one hop.
5: Sort $candidates$ in ascending order by each node's `distance` to n
6: Remove the first member of $candidates$ and add it to $short\_peers$
7: **for all** c in $candidates$ **do**
8:    $m \leftarrow$ `midpoint`(n, c)
9:    **if** any node in $short\_peers$ is closer to m than n **then**
10:       Reject c as a peer
11:    **else**
12:       Remove c from $candidates$
13:       Add c to $short\_peers$
14:    **end if**
15: **end for**
16: **while** $|short\_peers| < table\_size$ **and** $|candidates| > 0$ **do**
17:    Remove the first entry c from $candidates$
18:    Add c to $short\_peers$
19: **end while**
20: Add $candidates$ to the set of $long\_peers$
21: `handleLongPeers`$(long\_peers)$

---

only nodes that exist. The resulting short peers are a subset of the node's actual Delaunay neighbors. A crucial feature is that this subset guarantees that DGVH will form a routable mesh.

DGVH uses the midpoint to gauge which other nodes to use as its Delaunay triangulation [2]. Every maintenance cycle, nodes exchange their peer lists with their neighbors. A node creates a list of candidates by combining their peer list with their neighbor's peer list. [4] This list of peers is then sorted from closest to furthest distance. The node then initializes a new peer list with the closest candidate. For each of the remaining candidates, the node calculates the midpoint between itself and the candidate. If new peer list does not contain any nodes closer to the midpoint than the candidate, the candidate is added to the new peer list. Otherwise, the candidate is set aside. How long peers are handled

---

[3]This order matters, as some DHTs such as Chord are unidirectional.

[4]In our previous paper, nodes exchange peer lists with a single peer. Calls to DGVH in this paper use all the

depends on the particular DHT implementation.

Candidates are gathered via a gossip protocol as well as notifications from close peers. This process is described more in Section III-A.

The expected maximum size of *candidates* corresponds to the expected maximum degree of a vertex. For a Delaunay Triangulation in any number of dimensions is $\Theta(\frac{\log n}{\log \log n})$ [3]. We can therefore expect *short peers* to be bounded by $\Theta(\frac{\log n}{\log \log n})$.

The expected worst case cost of $O(\frac{\log^4 n}{\log^4 \log n})$ [2], regardless of the dimension [2]. [5] It most cases, this cost much lower, on the order of is $O(dk \log(k) + k^2)$ for d dimensions and k candidates.

We have tested DGVH on Chord (a ring-based topology), Kademlia (a XOR-based tree topology), general Euclidean spaces, and even in a hyperbolic geometry. We show in Section V that DGVH works in all of these spaces.

III. URDHT

The name of UrDHT comes from the the German prefix *ur*, which means the original. Our name states that all DHTs can spring from UrDHT.

UrDHT is sectioned off into 3 broad components: Storage, Networking, and Logic. Storage handles file storage and network dictates the protocol for how nodes communicate. These components deal with the lower level mechanics of how files are stored on the network and how bits are transmitted through the network. The specifics are outside the scope of the paper, but can be found on the UrDHT Project [11].

Most of our discussion will focus on the Logic component. The Logic component is what dictates the behavior of node within the DHT and the construction of the overlay network. It is composed of two parts: the DHT Protocol and the Space Math.

The DHT Protocol contains the canonical operations that a DHT performs, while the Space Math is what effectively distinguishes one DHT from another. A developer only needs to change the details of the `space math` package in UrDHT to create a new type of DHT. We discuss each in further detail below.

A. The DHT Protocol

The DHT Protocol (`LogicClass.py`) is the shared functionality between every single DHT. It consists of

the node's information, the short peer list to define the overlay, the long peers that make efficient routing possible, and all the functions that use them. There is no need for a developer to change anything in the DHT Protocol, but it can be modified if so desired. The DHT Protocol depends on functions from Space Math in order to perform operations within the specified space.

In our efforts to further abstract DHTs, we have implemented `lookup` using the function `seek`. The `seek` function acts a single step of `lookup`. It returns the closest node to *key* that the node knows about. Nodes perform `lookup` by iteratively calling `seek` until it receives the same answer twice.

The `join` operation takes in a set of bootstrap nodes, called *candidates*. The joining node randomly selects one of these *candidates* and finds the "parent" node currently responsible for the space. The joining node then populates its short peers using the "parent" node's short peers. The node uses the parent to populate its short peer list and then makes it aware of its existence using `notify`. Once that has been finished, the joining node starts its maintenance thread.

Maintenance is done via gossip. Each maintenance cycle, the node recalculates its Delaunay (short) peers using its neighbor's peer list and any nodes that have notified it since the last maintance cycle. This is done using DGVH by default, but could be done with some other algorithm. Once those are calculated, the node handles modifying it's long peers, as dictated by the `handleLongPeers` function described in Section III-B.

B. The Space Math

The space math consists of the functions which define the DHT's topology. It requires a way to generate short peers to form a routable overlay and a way to generate long peers. We provide DGVH for generating short peers, which works in every space we have tried. Space Math requires the following functions when using DGVH:

1) IDToPoint: The `idToPoint` function takes in a node's ID and any other attributes needed to map an ID onto a point in the space.

In the vast majority of DHTs, this `idToPoint` function needs nothing more than the ID as input. The ID is directly translated into a large integer and used as a coordinate in a one dimensional space.

2) Distance: The `distance` function takes in two points, a and b, and outputs the shortest distance from a to b. This distinction matters since distance is not

[5]As mentioned in the previous footnote, if we are exchanging peers with a single neighbor rather than all our neighbors, the cost lowers to $O(\frac{\log^2 n}{\log^2 \log n})$.

symmetric in every DHT. The prime example of this is Chord, which is a unidirectional toroidal ring.

3) Midpoint: Similarly to `distance`, `midpoint` takes in the points a and b and returns the closest midpoint[6] between the two.

4) Get Closest: The function `getClosest` returns the point closest to *center* from a list of *candidates*, measured by the distance function. Depending on what you want to measure, `getClosest` might measure the distance from *center* to each of the candidates or from each of the candidates to the *center*.

5) Get Delaunay (short) Peers: We then use the above functions to implement `getDelaunayPeers`. Given a set of points, the *candidates*, and a center point *centers*, `getDelaunayPeers` calculates a mesh that of the Delaunay peers of *center*.

We assume that this is done using DGVH, shown the Python code in Listing 1

Listing 1: getDelaunayPeers()

```python
def getDelaunayPeers(candidates,center):
        if len(candidates) < 2:
                return candidates
        sortedCandidates = sorted(candidates,
            key=lambda x: distance(x, center))
        peers = [sortedCandidates[0]]
        sortedCandidates = sortedCandidates
            [1:]
        for c in sortedCandidates:
                m = midpoint(c, center)
                accept = True
                for p in peers:
                        if distance(m,p) <
                            distance(m, center
                            ):
                                accept = False
                                break
                if accept:
                        peers.append(c)
        return peers
```

6) Handle Long Peers: The final function is `handleLongPeers`. `handleLongPeers` takes in a *candidates* and a *center*, much like `getDelaunayPeers`, and returns .

The implementation of this function should vary greatly from one DHT to another. For example, long peers in Symphony [9] and other small-world [7] networks choose long peers using a probability distribution. Chord has a much more structured distribution, with each long peer being an increasing power of 2 distance away from the node [12].

---

[6]Toroidal spaces have more than one midpoint.

In some case it may more convenient implement `handleLongPeers` as part of `getDelaunayPeers`.

IV. IMPLEMENTING OTHER DHTs

A. Implementing Chord and Ring-Based Topology

Ring topologies are fairly straightforward since they act as are one dimensional Voronoi Tesselations, splitting up what is effectively a modular number line among multiple nodes.

We know Chord's invariants are not (citation), but our protocol isn't affected by these constraints

B. Implementing Kademlia and Other Tree Based Topologies

The largest complication in implementing UrKademlia is defining the exclusive or, or XOR, metric which is used for distance. This metric, while non-euclidean, is perfectly acceptable for calculating distance in Kademlia [10] However, XOR does not have an intuitive midpoint we could use for DGVH.

To solve this, we used the XOR metric defined by Kademlia as the distance function and the midpoint function.

We then implemented handle long peers.

C. Implementing A Euclidean Space

Euclidean space was the default space we used when building UrDHT.

D. ZHT

ZHT [8] leads to an extremely trivial implementation in UrDHT. Unlike other DHTs, ZHT assumes an extremely low rate of churn. As a result, the topology of ZHT is a clique, with each node having an edge to all other nodes, yielding $O(1)$ lookup times.

The only change that needs to be made is to `handleLongPeers`. We just place no upper bound on the size of the list of Long Peers.

E. DHTs in a Hyperbolic Topology

1) Okay, this is interesting, but why bother?: Because we it was difficult

Because it shows that UrDHT and DGVH both work in arbitrary geometries. For example, handling geographic coordinates.

F. Services

`put poll`

V. Experiments

We use simulations to test our DHTs. Using simulations to test the correctness and relative performance of DHTs is standard practice in both whitepapers and applications Citations are all major DHTs Citations for analysis works that use simulations Citations for applications of DHTs that were peer reviewed and accepted using simulations

What are our experiments?

A. UrDHT Cohesion Euclidean Space

B. Cohesion in hyperbolic space

We showed in worked in DGVH, it should work in a hyperbolic space.

C. Performance of Chord on our network module vs UrChord

D. UrKademlia Works

VI. Future Work and Conclusions

UrDHT

There are numerous routes

Creating a DHT overlay that operates in a hyperbolic geometry One of the other features shared by nearly every DHT is that routing works by minimizing the number of hops across the overlay network, with all hops treated as the same length. This is done because it is assumed that DHTs know nothing about the state of actual infrastructure the overlay is built upon.

However, this means that most DHTs will happily route a message from one continent to another and back. This is obviously undesirable, but it is the status quo in DHTs. This stems from the generation of node IDs in DHTs. Nodes are typically assigned a point in the range of a cryptographic hash funtion, called the keyspace. This ID corresponds to the hash of some identifier or given a point randomly. This is done for purposes of load balancing and fault tolerance.

We want to see if there is a means of embedding latency into the DHT, while still maintaining the system's fault tolerance. Doing so would mean that the hops traversed to a destination are, in fact, the shortest path to the destination.

We believe we can embed a latency graph in a hyperbolic space and define UrDHT such that it operates within this space. The end result would be a DHT with latency embedded into the overlay. Nodes would respond to changes in latency and the network by rejoining the network at new positions.

Hyperbolic spaces allow us to cleanly embed scale free graphs

References

[1] Redis. http://redis.io.

[2] Brendan Benshoof, Andrew Rosen, Anu G. Bourgeois, and Robert W Harrison. A distributed greedy heuristic for computing voronoi tessellations with applications towards peer-to-peer networks. In *Dependable Parallel, Distributed and Network-Centric Systems, 20th IEEE Workshop on*.

[3] Marshall Bern, David Eppstein, and Frances Yao. The expected extremes in a delaunay triangulation. *International Journal of Computational Geometry & Applications*, 1(01):79–91, 1991.

[4] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Designing Privacy Enhancing Technologies*, pages 46–66. Springer, 2001.

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

[6] Steven Fortune. A sweepline algorithm for voronoi diagrams. *Algorithmica*, 2(1-4):153–174, 1987.

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

[8] 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.

[9] 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.

[10] 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.

[11] Andrew Rosen, Brendan Benshoof, Robert W Harrison, and Anu G. Bourgeois. Urdht. https://github.com/UrDHT/.

[12] 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.

[13] David F Watson. Computing the n-dimensional delaunay tessellation with application to voronoi polytopes. *The computer journal*, 24(2):167–172, 1981.