# 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 UrChord 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.

- **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. This is not necessarily the case with UrDHT (see Section IV-E).
- **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

```
Given node $n$ and a message being sent to $key$
function $n$.lookup($key$)
if If $key \in n$'s range of responsibility then
    return $n$
end if
if One of $n$'s short peers are responsible for $key$ then
    return the responsible node
end if
$candidates$ = $short\_peers$ + $long\_peers$
$next \leftarrow \min(n.\text{distance}(candidates, key))$
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

[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.

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 [8]. 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. However there is a plethora of DHTs with highly diverse spaces. The majority of well-known algorithms for solving this problem, such as Fortune's Sweepline [6], are designed to be run centralized.

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].

From the perspective of the node, the condidates are the only nodes in that exist. A node a set of peers, and uses DGVH to determine which of these correspond to Delaunay peers and form a Voronoi region 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.

*Algorithm Explanation:* 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. 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.

**Algorithm 2** Distributed Greedy Voronoi Heuristic

```
Given node $n$ and its list of $candidates$.
Given the minimum $table\_size$
$short\_peers \leftarrow$ empty set that will contain $n$'s one-hop peers
$long\_peers \leftarrow$ empty set that will contain $n$'s peers further than one hop.
Sort $candidates$ in ascending order by each node's distance to $n$
Remove the first member of $candidates$ and add it to $short\_peers$
for all $c$ in $candidates$ do
    $m \leftarrow \texttt{midpoint}(n, c)$
    if any node in $short\_peers$ is closer to $m$ than $n$ then
        Reject $c$ as a peer
    else
        Remove $c$ from $candidates$
        Add $c$ to $short\_peers$
    end if
end for
while $|short\_peers| < table\_size$ and $|candidates| > 0$ do
    Remove the first entry $c$ from $candidates$
    Add $c$ to $short\_peers$
end while
Add $candidates$ to the set of $long\_peers$
handleLongPeers($long\_peers$)
```

*How do we get Candidates:* Candidates are gathered via the gossip protocol.

*Algorithm complexity:*

*Sizes of short peers and long peers expected sizes and candidates:* The expected maximum degree of a vertex in a Delaunay Triangulation in any number of dimensions is $\Theta(\frac{\log n}{\log \log n})$ [3]. So we can realistically expect *short peers* to be bounded by $\Theta(\frac{\log n}{\log \log n})$.

*How do we handle long peers:*

*Lead to next:* We have tested DGVH on Chord (a ring-based topology), Kademlia (a XOR-based tree topology),

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

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. Most of our discussion will focus on the Logic component.

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 [9].

The Logic component is what dictates the behavior of 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 is the canonical operations that a DHT performs, while the space math is what effectively distinguishes one DHT from another. The `space math` package in UrDHT is what needs to be changes to create a new type of DHT. We discuss each in further detail below.

### A. The DHT Protocol

The DHT protocol 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 a bunch of other stuff.

Maintenance is gossip

### B. The Space Math

The space math consists of the functions which define the DHTs topology. Essentially this is the

Space math just needs a Voronoi tessellation/ whatever creator We provide DGVH for this, which works in every case we've tried. If you use DGVH, here's what needs to be changed.

1) *Distance:*
2) *Midpoint:*
3) *Get Closest:*
4) *Get Delaunay (short) Peers:*
5) *Long Peer Selection:*

## 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 aren't (citation), but our protocol isn't affected by these constraints

### B. Implementing Kademlia and Other Tree Based Topologies

Trees are easy to embed in a hyperbolic space.

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 [8] 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

### D. ZHT

ZHT [7] leads to an extremely trivial implementation in UrDHT. ZHT is effectively Clique

There is couple ways to solve this.

### E. DHTs in a Hyperbolic Topology

### F. 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.

### G. Wait, Nodes can move in DHTs??

Yes, they can. There's no rule against it. In fact, it helps.

### H. Services

## 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 expermiments?

### *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

Move Latency embedding section from intro to here

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] 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] 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.

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

[10] 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.