

# 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**—UrDHT is an abstracted Distributed Hash Table (DHT). By completing a few simple functions, a developer can implement the topology of any DHT.

Current distributed systems suffer from fragmentation, high overhead, and an inability to scale due to difficulty of adoption. UrDHT is P2P system designed to improve the adaptability of P2P distributed serves.

## I. INTRODUCTION

Distributed Hash Tables have been extensively researched for the past decade. Despite this, no one has created a cohesive formal specification for building a DHT.

UrDHT is our specification and implementation of an abstract DHT.

- We first discuss our motivation for creating UrDHT and *creating it the way we did* (Section II).
- 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 define a set of simple functions that are needed to implement a DHT. We show that using these functions, DHTs cleanly map to the primal-dual problem of Delaunay triangulations and Voronoi tessellations (Section III).
- 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. We show how to reproduce the topology of Chord and Kademlia using UrDHT, which we call UrChord and UrKademlia.
- We conduct experiments showing that UrChord sufficiently approximates a correct implementation of Chord.

## II. MOTIVATION

Distributed Hash Tables have been the catalyst for the creation of many P2P applications. Among these are example 1, example 2, another citation, citation, and, most notably, BitTorrent [2].

### A. Bootstrapping

One issue in the adoption of new P2P applications is the bootstrapping problem. 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 other motivation is making it easier for users to create distributed applications. What topology do you use? How do we want our program to communicate over the network?

UrDHT exists to simplify this process, minimizing the distributed application development time and making it easier to adopt by creating a network to bootstrap *other networks*.

### B. Abstraction

We've see that CHord doesn't work

### C. Embedding

## III. 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<sup>1</sup> 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.<sup>2</sup> The

<sup>1</sup>Unique with astronomically high probability, given a large enough consistent hash algorithm.

<sup>2</sup>There is typically a `delete(key)` operation defined too, but it is not strictly necessary.

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. Here, we open that black box for the first time and present those components. 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-B).
- **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 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 the relatively the greedy routing algorithm (Algorithm X):

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,

---

#### Algorithm 1 The DHT routing algorithm

---

```

1: Given node  $n$  and a message being sent to  $key$ 
2: if  $key \in n$ 's range of responsibility then
3:   return  $n$ 
4: end if
5: if One of  $n$ 's short peers are responsible then
6:   return the responsible node
7: end if

```

---

I forward this query to the node I know with shortest distance from the node to the desired key.<sup>3</sup>

Between individual DHTs, this algorithm might be implemented either recursively or iteratively, the handling of dead nodes encountered during lookup might be difference, and possibly in parallel such as in case of Kademlia. 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 high impossible. LEAD INTO MULTIHASH GOES HERE

#### B. DHTs, Delaunay Triangulation, and Voronoi Tessellation

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.

So how do we efficiently calculate Delaunay Triangulations in a distributed system? We created an algorithm call the Distributed Greedy Voronoi Heuristic (DGVH) [1], shown in Algorithm 2.

### IV. URDHT

#### A. UrDHT Components (or maybe logic)

UrDHT is sectioned off into 3 components: database, network, and logic. Database handles file storage and network dictates the protocol for how nodes communicate.

The distance and midpoints functions are defined and discussed in detail in Section IV-B

<sup>3</sup>This order matters, as Chord is unidirectional.

---

**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:    $m \leftarrow \text{midpoint}(n, c)$ 
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:  $\text{handleLongPeers}(long\_peers)$ 
```

---

1) *Put and Poll:*

B. *Hyperbolic Routing*

C. *Okay, this is interesting, but why bother?*

Hyperbolic spaces allow us to cleanly embed scale free graphs

D. *Wait, Nodes can move in DHTs??*

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

E. *Implementing Chord and Ring Based Topology*

F. *Implementing Kademlia and Other Tree Based Topologies*

Trees are easy to embed in a hyperbolic space.

G. *ZHT*

ZHT leads to an extremely trivial implementation in UrDHT.

## V. EXPERIMENTS

## VI. FUTURE WORK AND CONCLUSIONS

## REFERENCES

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