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 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 for a DHT. These attributes have not been formally defined. (Section III)
- We present UrDHT as an abstract DHT and show how a developer can tweak the functions we defined to create new DHT topologies. 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 [1].

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*.

III. What Defines a DHT

A distributed hash table is usually defined by its API; 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. 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.

[1]Unique with astronomically high probability, given a large enough consistant 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. This is not necessarily the case with UrDHT (see Section IV-B).
- **A `midpoint` function** - The . 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 the relatively the greedy routing algorithm (Algorithm X):

ALGORITHM BLOCK GOES HERE OR SOMETHING

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, 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 nigh impossible. LEAD INTO MULTIHASH GOES HERE

[3]This order matters, as Chord is unidirectional.

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 nodes ID to a point in a space, the range of keys a node is responsible for to that nodes Voronoi region, and the set of short peers to the Delaunay triangulation.

So how do we efficiently calculate Delaunay Triangulation and Voronoi Tesselations? We created an algorithm call the Distributed Greedy Voronoi Heuristic (DGVH).

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.

1) Put and Poll:

B. *Hyperbolic Routing*

C. *Implementing Chord and Ring Based Topology*

D. *Implementing Kademlia and Other Tree Based Topologies*

Trees are easy to embed in a hyperbolic space.

E. *ZHT*

ZHT leads to an extremely trivial implementation in UrDHT.

V. Experiments

VI. Future Work and Conclusions

References

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