Towards a Framework for DHT Distributed Computing

by

Andrew Benjamin Rosen

Under the Direction of Dr. Anu G. Bourgeois, PhD

Abstract

Distributed Hash Tables (DHTs) are protocols and frameworks used by peer-to-peer (P2P) systems. They are used as the organizational backbone for many P2P file-sharing systems due to their scalability, fault-tolerance, and load-balancing properties. These same properties are highly desirable in a distributed computing environment, especially one that wants to use heterogeneous components.

We show that DHTs can be used not only as the framework to build a P2P file-sharing service, but as a P2P distributed computing platform. We propose creating a P2P distributed computing framework using distributed hash tables, based on our prototype system ChordReduce. This framework would make it simple and efficient for developers to create their own distributed computing applications. Unlike Hadoop and similar MapReduce frameworks, our framework can be used both in both the context of a datacenter or as part of a P2P computing platform. This opens up new possibilities for building platforms to distributed computing problems.

One advantage our system will have is an autonomous load-balancing mechanism. Nodes will be able to independently acquire work from other nodes in the network, rather than sitting idle. More powerful nodes in the network will be able use the mechanism to acquire more work, exploiting the heterogeneity of the network.

By utilizing the load-balancing algorithm, a datacenter could easily leverage additional P2P resources at runtime on an as needed basis. Our framework will allow MapReduce-like or distributed machine learning platforms to be easily deployed in a greater variety of contexts.

Index Words: Distributed Hash Tables, P2P, Voronoi, Delaunay, Networking

Towards a Framework for DHT Distributed Computing

by

Andrew Benjamin Rosen

A Dissertation Submitted in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy in Computer Science

in the College of Arts and Sciences

Georgia State University

2016

Towards a Framework for DHT Distributed Computing

by

Andrew Benjamin Rosen

Committee Chair Anu G. Bourgeois

Committee Robert Harrison

Yingshu Li

Michael Stewart

Electronic Version Approved:

Office of Graduate Studies

College of Arts and Sciences

Georgia State University

May 2016

Dedication

I would like to take the time to thank Annie-Rae Rosen, without whom, I would not be who I am today.

To my mother, who gave me a name that became a self-fulfilling prophecy.

Acknowledgments

There were some people who cared about what I did. I'm not particularly sure why.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Distributed Hash Tables (DHTs) are protocols and frameworks used by peer-to-peer (P2P) systems. They are used as the organizational backbone for many P2P file-sharing systems due to their scalability, fault-tolerance, and load-balancing properties. These same properties are highly desirable in a distributed computing environment, especially one that wants to use heterogeneous components. We will show that DHTs can be used not only as the framework to build a P2P file-sharing service, but a more generic distributed computing platform.

1.1 Objective

Our goal is to create a framework to further generalize Distributed Hash Tables (DHTs) to be used for distributed computing. Distributed computing platforms need to be scalable, fault-tolerant, and load-balancing. We will discuss what each of these mean and why they are important in section 1.3.1, but briefly:

- The system should be able to work effectively no matter how large it gets. As the system grows in size, we can expect the overhead to grow in size as well, but at an extremely slower rate.
- The more machines integrated into the system, the more we can expect to see hardware failures. The system needs to be able to automatically handle these hardware failures.
- Having a large number of machines to use is worthless if the amount of work is divided

unevenly among the system. The same is true if the system hands out larger jobs to less powerful machines or smaller jobs to the more powerful machines.

These are many of the same challenges that Peer-to-peer (P2P) file sharing applications have. Many P2P applications use DHTs to address these challenges, since DHTs are designed with these problems in mind. We propose that DHTs can be used to create P2P distributed computing platforms that are completely decentralized. There would be no need for some central organizer or scheduler to coordinate the nodes in the network. Our framework would not be limited to only a P2P context, but could be applied in data centers, a normally centrally organized context.

A successful DHT-based computing platform would need to address the problem of dynamic load-balancing. This is currently an unsolved[1] problem. If an application can dynamically reassign work to nodes added at runtime, this opens up new options for resource management. Similarly. if a distributed computation is running too slow, new nodes can be added to the network during runtime or idle nodes can boot up more virtual nodes.

Chapter 2 will delve into how DHTs work and examine specific DHTs. The remainder of the dissertation will then discuss the work we have completed and plan on doing to demonstrate the viability of using DHTs for distributed computing and other non-traditional tasks.

1.2 Applications of Distributed Hash Tables

Distributed Hash Tables have been used in numerous applications:

- *P2P file sharing* is by far the most prominent use of DHTs. The most well-known application is BitTorrent [9], which is built on Mainline DHT [28].
- DHTs have been used for *distributed storage* systems [13].
- *Distributed Domain Name Systems* (DNS) have been built upon DHTs [11] [35]. Distributed DNSs are much more robust that DNS to orchestrated attacks, but otherwise require more overhead.
- DHT was used as the name resolution layer of a large *distributed database* [31].

[1] As far as we know.

- Distributed *machine learning* [26].
- Many *botnets* are now P2P based and built using well established DHTs [44]. This is because the decentralized nature of P2P systems means there is no single vulnerable location in the botnet.
- *Live video streaming* (BitTorrent live) [34].

We can see from this list that DHTs are primarily used in P2P applications, but other applications, such as botnets, use DHTs for their decentralization. We want to use DHTs primarily for their intuitive way of organizing a distributed system.

Our goal was to further extend the use of DHTs. In previous work [40], we showed that a DHT can be to create a distributed computing framework. We used the same mechanism used in P2P applications that assigns nodes their location in the network to evenly distribute work among members of a DHT. The most direct application of a DHT distributed computing framework is a quick and intuitive way to solve embarrassingly parallel problems, such as:

- Brute force cryptography.
- Genetic algorithms.
- Markov chain Monte Carlo methods.
- Random forests.
- Any problem that could be phrased as a MapReduce problem.

Unlike the current distributed applications that utilize DHTs, we want to create a complete framework that can be used to build decentralized applications. We have found no existing projects that provide a means of building your own DHT or DHT based applications.

1.3 Why Use Distributed Hash Tables in Distributed Computing

Using distributed hash tables for distributed computing is not necessarily the most intuitive step. To understand why we want to use DHTs for distributed computing, we will first examine some of the more prominent challenges in distributed computing.

1.3.1 General Challenges of Distributed Computing

As we mentioned earlier, distributed computing platforms need to be scalable, fault-tolerant, and load-balancing. We will look at these individually:

Scalability - Distributed computing platforms should not be completely static and should grow to accommodate new needs. However, as systems grow in size, the cost of keeping that system organized grows too. The challenge of scalability is designing a protocol that grows this organizational cost at an extremely slow rate. For example, a single node keeping track of all members of the system might be a tenable situation up to a certain point, but eventually, the cost becomes too high for a single node. We want this organizational cost spread among many nodes to the point where this cost is insignificant.

Fault Tolerance The quality of fault-tolerance or *robustness* means that the system still works even after a component breaks (or many components break). We want our platform to gracefully handle failures during runtime and be able to quickly reassign work to other workers. In addition, the network should be equally graceful in handling the introduction of new nodes during runtime.

Load-Balancing The challenge of load balancing is to evenly distribute the work among nodes in the network. This is always an approximation; rarely are there exactly enough pieces for every node to get the same amount of work. The system needs an efficient set of rules for dividing arbitrary jobs into small pieces and sending those pieces to the nodes, without incurring a large overhead.

A subproblem here is handling *heterogeneity*,[2] or how should the system should handle different pieces of hardware with different amounts of computational power.

Note that there is some crossover between these categories. For example, adding new nodes to the system needs to have a low organizational overhead (scalability) and will change the network configuration, which will need to be updated (fault-tolerance).

[2] It could even be considered a problem in its own right.

1.3.2 How DHTs Address these Challenges

Distributed Hash Tables are essentially distributed lookup tables. DHTs use a consistent hashing algorithm, such as SHA-1 [19], to associate nodes and file identifiers with keys. These keys dictate where the nodes and files will be located on the network. The connections between nodes are organized such that any node can efficiently lookup the value associated with any given key, even though the node only knows a small portion of the network. We discuss the specifics of this in Chapter 2.

Nearly every DHT was designed with large P2P applications in mind, with millions of nodes in the network and new nodes entering and leaving continuously.

Scalability The organizational responsibility in DHTs is spread among all members of the network. Each node only knows a small subset of the network,[3] but can use the nodes it knows to efficiently find any other node in the network. Because each individual node only knows a small part of the network, the maintenance costs associated with organization are correspondingly small.

Using consistent hashing allows the network to scale up incrementally, adding one node at a time [15]. In addition, each join operation has minimal impact on the network, since a node affects only its immediate neighbors on a join operation. Similarly, the only nodes that need to react to a node leaving are its neighbors. Other nodes can be notified of the missing node passively through maintenance or in response to a lookup.

There have been multiple proposed strategies for tackling scalability, and it is these strategies that play the greatest role in driving the variety of DHT architectures. Each DHT must strike a balance between the size of the lookup table and lookup time. The vast majority of DHTs choose to use $\lg(n)$ sized tables and $\lg(n)$ hops, where n is the number of nodes in the network. Chapter 2 discusses these tradeoffs in greater detail and how they affect the each DHT.

Fault-Tolerance One of the most important assumptions of DHTs is that they are deployed on a constantly changing network. DHTs are built to account for a high level of *churn*.[4] *Churn* is the disruption of routing caused by the constant joining and leaving of nodes. In other words, the network topology is assumed to always be in flux. This is mitigated by a few factors.

[3] Except for ZHT [27], which breaks this rule deliberately by giving each node a full copy of the routing table.

[4] Again, except for ZHT.

First, the network is decentralized, with no single node acting as a single point of failure. This is accomplished by each node in the routing table having a small portion of the both the routing table and the data stored on the DHT.

Second is that each DHT has an inexpensive maintenance processes that mitigates the damage caused by churn. DHTs often integrate a backup process into their protocols so that when a node goes down, one of the neighboring nodes can immediately assume responsibility. The join process also slightly disrupts the topology, as affected nodes must adjust their the list of peers they know to accommodate the joiner.

The last property is that the hashing algorithm used to distribute content evenly across the DHT also distributes nodes evenly across the DHT. This means that nodes in the same geographic region occupy vastly different locations in the network. If an entire geographic region is affected by a network outage, this damage is spread evenly across the DHT, and can be handled, rather than if a contiguous portion were lost.

The fault tolerance mechanisms in DHTs also provide near constant availability for P2P applications. The node that is responsible for a particular key can always be found, even when numerous failures or joins occur [46].

Load-Balancing Consistent hashing is also used to ensure load-balancing in DHTs. Consistent hashing algorithms associate nodes and file identifiers with keys. These keys are generated by passing the identifiers into a hash function, typically SHA-160. The chosen hash function is typically large enough to avoid hash collisions[5] and generates keys in a uniform manner.

Essentially, both nodes and data are spread about the network uniformly at random. Nodes are responsible for the files with keys "close" to their own. What "close" means depends on the specific implementation. For example, "close" might mean "closest without going over."

We found defining the meaning of "close" equivalent choosing a metric for Voronoi tessellation [6]. However, because this is a random process, not all values are evenly distributed, but enough hash keys yield a close enough approximation.

Heterogenity presents a challenge for load-balancing DHTs due to conflicting assumptions and goals. DHTs assume that members are usually going to be varied in hardware, but the load-

[5] A hash collision occurs when the hashing algorithm outputs the same hashkey for two different inputs.

balancing process defined in DHTs treats each node equally. In other words, DHTs support heterogeneity, but do not attempt to exploit it.

This does not mean that heterogeneity cannot be exploited. Nodes can be given addition responsibilities manually, by running multiple instances of the P2P application on the same machine or creating more virtual nodes. We will take advantage of this for distributing the workload automatically.

1.4 Roadmap

In this section, we give a brief overview of our previous work. We go into further detail of our previous work in Chapter **??** and present the proposed work of our dissertation in Chapter **??**.

1.4.1 Completed Work

One of our first projects was to create a distributed computing platform using the Chord DHT [40]. Our goal here was to create a completely decentralized distributed computing framework that was fault-tolerant during job execution. We did this by implementing MapReduce over Chord. We then tested our prototype's fault-tolerance by executing MapReduce jobs under churn.

Our experiments with excessively high levels of churn created an anomaly in the runtime of our computations. Under beyond practical levels of experimental churn, we found that our computation was quicker than our experiments without churn. We hypothesized that this is because the random churn is acting as a (inefficient) process for autonomous load-balancing. This phenomena is described in detail in Chapter **??**, but suggested to us that there was a way to dynamically load-balance during execution.

Our second project was to develop VHash [5] [6], a distributed hash table based on Delaunay Triangulation. VHash is unique due to the way it could work in multidimensional spaces. Other DHTs typically use a space with a single dimension and optimize for the number of hops. VHash can optimize for whatever attributes are used to define the space. Our experiments showed that VHash outperforms Chord in terms of routing latency.

Our third project which analyzed the amount of effort that would be required to attack a DHT using a method known as the Sybil attack [41]. The Sybil attack [16] is a well known attack against distributed systems, but it had not been fully analyzed from the perspective of an attacker. Our results showed that attackers required relatively few resources to compromise a much larger network. We believe that some of the components that are used to perform a Sybil attack can be used for autonomous load balancing.

Publications

- Andrew Rosen, Brendan Benshoof, Robert W. Harrison, Anu G. Bourgeois "MapReduce on a Chord Distributed Hash Table" Poster at IPDPS 2014 PhD Forum [40]
- Andrew Rosen, Brendan Benshoof, Robert W. Harrison, Anu G. Bourgeois "MapReduce on a Chord Distributed Hash Table" Presentation ICA CON 2014
- Brendan Benshoof, Andrew Rosen, Anu G. Bourgeois, Robert W. Harrison "VHASH: Spatial DHT based on Voronoi Tessellation" Short Paper ICA CON 2014 [6]
- Brendan Benshoof, Andrew Rosen, Anu G. Bourgeois, Robert W. Harrison "VHASH: Spatial DHT based on Voronoi Tessellation" Poster ICA CON 2014
- Brendan Benshoof, Andrew Rosen, Anu G. Bourgeois, Robert W. Harrison "A Distributed Greedy Heuristic for Computing Voronoi Tessellations With Applications Towards Peer-to-Peer Networks" IEEE IPDPS 2015 - Workshop on Dependable Parallel, Distributed and Network-Centric Systems [5]

The following papers are in progress:

- Brendan Benshoof, Andrew Rosen, Anu G. Bourgeois, Robert W. Harrison "UrDHT: A Generalized DHT"
- Andrew Rosen, Brendan Benshoof, Robert W. Harrison, Anu G. Bourgeois "The Sybil Attack on Peer-to-Peer Networks From the Attacker's Perspective"
- Chaoyang Li, Andrew Rosen, Anu G. Bourgeois "On Minimum Camera Set Problem in Camera Sensor Networks"

Below are publications with other authors not relevant to the proposed work.

- Erin-Elizabeth A. Durham, Andrew Rosen, Robert W. Harrison "A Model Architecture for Big Data applications using Relational Databases" 2014 IEEE BigData - C4BD2014 - Workshop on Complexity for Big Data [17]

- Chinua Umoja, J.T. Torrance, Erin-Elizabeth A. Durham, Andrew Rosen, Dr. Robert Harrison "A Novel Approach to Determine Docking Locations Using Fuzzy Logic and Shape Determination" 2014 IEEE BigData - Poster and Short Paper [47]

- Erin-Elizabeth A. Durham, Andrew Rosen, Robert W. Harrison "Optimization of Relational Database Usage Involving Big Data" IEEE SSCI 2014 - CIDM 2014 - The IEEE Symposium Series on Computational Intelligence and Data Mining [18]

1.4.2 Summary of Proposal

We divide the proposed work into three distinct, but mutually dependent parts. One of these parts, the DHT framework, is a part that will be done jointly with Brendan Benshoof. The specifics are given in Chapter **??**.

DHT Framework

The goal of the DHT framework is to create a ready-to-use framework for creating DHT applications. We will then use this to create the DHT applications for DHT distributed computing. While developing VHash, we discovered the closeness metric used by DHTs to determine which node is responsible for what data is analogous to the metric used to create a Voronoi tessellation. This means the neighbors of a node map to Delaunay triangulations. To the best of our knowledge, no other party has inferred the relationship between Voronoi tessellations, Delaunay triangulations and DHTs. These properties give us a way to postulate an *ur*-DHT, a progenitor DHT which could be used to define all other DHTs.

UrDht is an open source project which we created. We will use UrDHT to implement and test multiple DHTs and applications. Using the same base framework allows us to minimize implementation differences when comparing DHTs in experiments, but also allows us to create applications quickly.

DHT Distributed Computing

This portion will be the bulk of the experimental work and data gathering. Using our created framework, we will create implement and test distributed computing problems on different DHT implementations, such as Chord [46] and Kademlia [32].

Autonomous Load-Balancing

Our goal is to develop a new and efficient algorithm for balancing the workload among members of the DHT. Load balancing schemes do exist for file storage, but none exist for computation. Furthermore, we want to develop a system that takes into account the heterogeneity of a given system, allowing more powerful nodes to take on more responsibility.

Chapter 2

Background

This chapter gives a broad overview of the concepts and implementations of Distributed Hash Tables (DHTs). This will provide context for our completed and future work.

DHTs have been a vibrant area of research for the past decade, with several of the concepts dating further back [9] [32] [38] [39] [36] [46] [43]. 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 tradeoffs in maintenance schemes, topology, and memory, with no choice being strictly better than any other.

2.1 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 [46] 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 2.1.

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 2.2.

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.

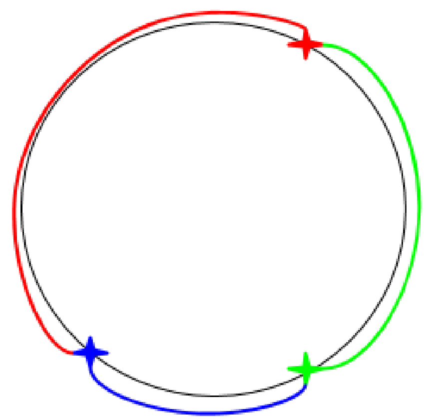

Figure 2.1: A Voronoi diagram for a Chord network, using Chord's definition of closest.

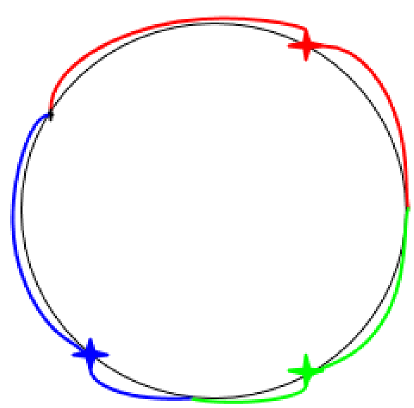

Figure 2.2: A Voronoi diagram for a Chord network, where closest is defined by the node being the closest in either direction.

2.1.1 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[1] node or file. SHA-1, which generates 160-bit hashes, is typically used as a hashing algorithm.[2]

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 proposal, 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 [43] [51] 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 [46], 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.

[1] Unique with extremely high probability. The probability of a hash collision is extremely low.

[2] Due to the research into hash collisions [45], and the glut of hardware that currently exists to perform SHA hash collisions, SHA1 is being depreciated by 2017.

`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.

`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[3].

When analyzing the DHTs in this chapter, 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.

[3]check this statement for consistency

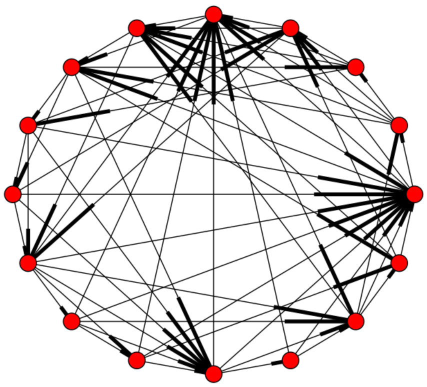

Figure 2.3: A Chord ring with 16 nodes. The fingers (long hop connections) are shown cutting across the ring.

2.2 Chord

Chord [46] 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 [50]. However, recent research has demonstrated that there have been no correct implementations of Chord in over a decade [50].

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 2.3.

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 [46], 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 [46].

2.3 Kademlia

Kademlia [32] 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.

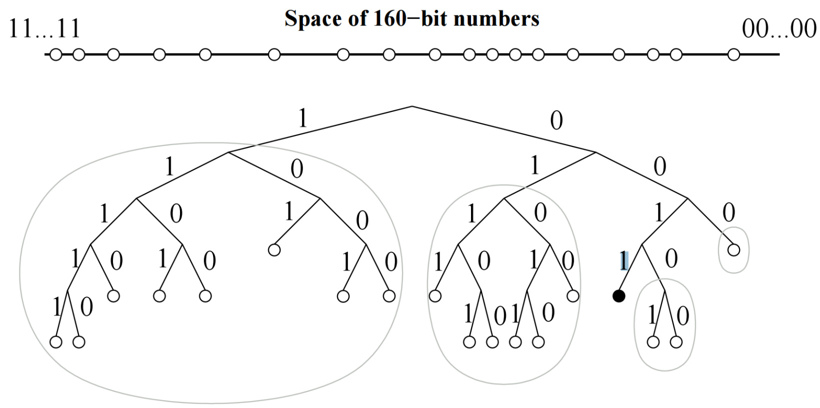

Figure 2.4: An example Kademlia network from the original paper [32]. The ovals are the node's k-buckets.

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 2.4.

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.

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

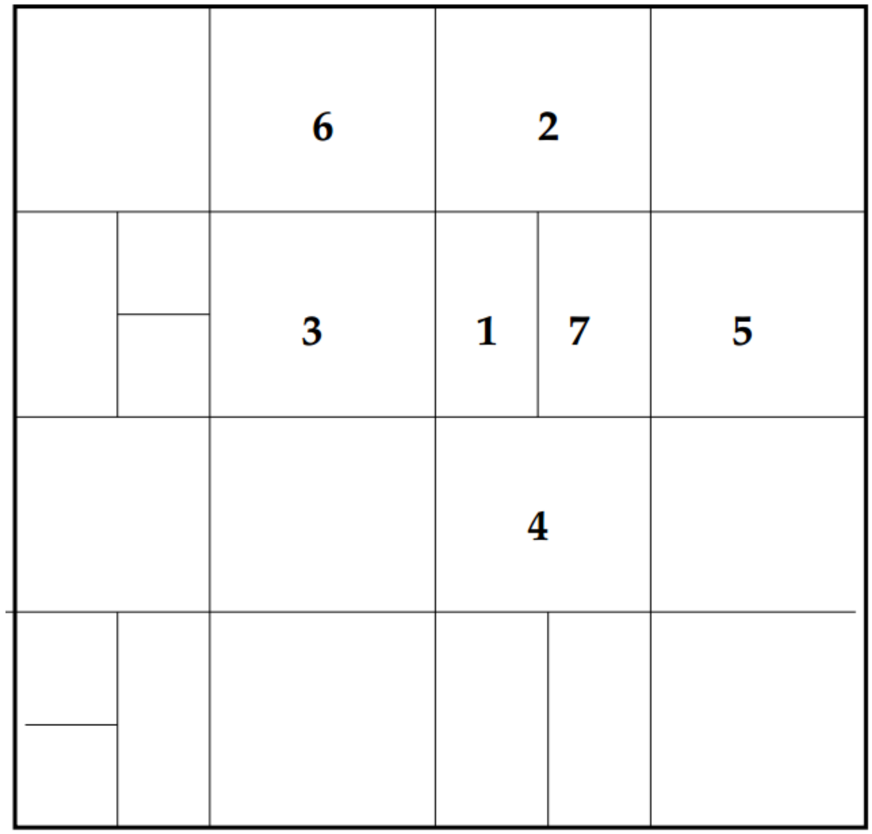

Figure 2.5: An example CAN network from [38].

2.4 CAN

Unlike the previous DHTs presented in this chapter, the Content Addressable Network (CAN) [38] 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.

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 2.5

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 [38].

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

[5] Around the same time CAN was being developed, Kleinberg was doing research into small world networks [22]. He proved similar properties for lattice networks with a single shortcut. What makes this network remarkable is lack of shortcuts.

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 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.

2.5 Pastry

Pastry [43] and Tapestry [51] are extremely similar use a prefix-based routing mechnism introduced by Plaxton et al. [37]. 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 [10], 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

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

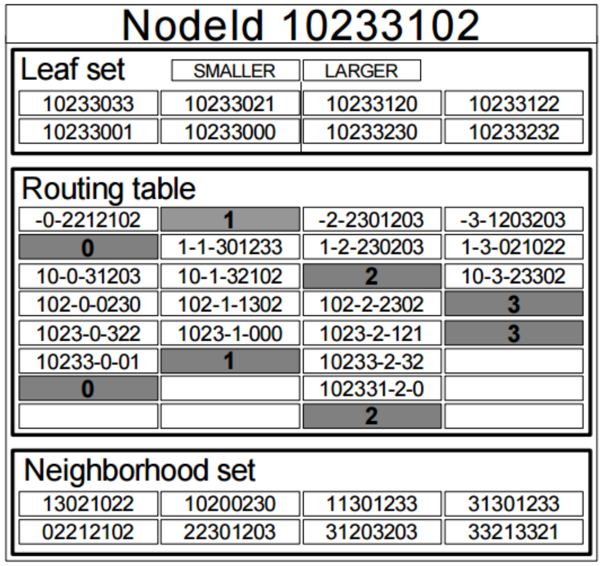

Figure 2.6: An example peerlist for a node in Pastry [43].

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.
- 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 2.6 shows an example peerlist of a node in PAST.

[7]0 is the 0th level.

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.

2.6 Symphony and Small World Routing

Symphony [30] is a $1d$ ring-based DHT similar to Chord [46], but is constructed using the properties of small world networks [22]. 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 [33]. 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 [23] 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.

[8] Randomly chosen from a specified distribution.

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.

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.

2.7 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

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

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 [27] is an example of such a system, as is Amazon's Dynamo [15]. 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.

2.8 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 2.1 summarizes the information presented in this chapter.

DHT	Routing Table Size	Lookup Time	Join/Leave	Comments
Chord [46]	$O(\log n)$, 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 [32]	$O(\log n)$, maximum $m \cdot k$	$(\lceil \log n \rceil) + c$	$O(\log(n))$	This is without considering optimization
CAN [38]	$\Omega(2d)$	$O(n^{\frac{1}{d}})$, average $\frac{d}{4} \cdot n^{\frac{1}{d}}$	Affects $O(d)$ nodes	d is the number of dimensions
Plaxton-based DHTs, Pastry [43], Tapestry [51]	$O(\log_\beta n)$	$O(\lceil \log_{2^\beta} \rceil)$	$O(\log_\beta n)$	NodeIDs are base β numbers
Symphony [30]	$2k + 2$	average $O(\frac{1}{k}\log^2 n)$	$O(\log^2 n)$ messages, constant < 1	$k \geq 1$, fingers are chosen at random
ZHT [27]	$O(n)$	$O(1)$	$O(n)$	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 2.1: The different ratios and their associated DHTs

Chapter 3

ChordReduce

Google's MapReduce [14] paradigm has rapidly become an integral part in the world of data processing and is capable of efficiently executing numerous Big Data programming and data-reduction tasks. By using MapReduce, a user can take a large problem, split it into small, equivalent tasks and send those tasks to other processors for computation. The results are sent back to the user and combined into one answer. Many popular platforms for MapReduce, such as Hadoop [2], utilize a central source of coordination and organization to store and operate on data. The hierarchical structure of Hadoop results in a single point of failure at the node that concentrates the results and also requires a complicated scheme for handling node failures.

We developed a system, called ChordReduce, which employs a less hierarchical structure. It is a system that can scale, is fault tolerant, has a minimal amount of latency, and distributes tasks evenly. ChordReduce leverages the underlying protocol from Chord [46] to distribute Map and Reduce tasks to nodes evenly, provide greater data redundancy, and guarantee a greater amount of fault tolerance. Rather than viewing Chord solely as a means for sharing files, we see it as a means for distributing work. This paper establishes the effectiveness of using Chord as a framework for distributed programming. At the same time we avoid the architectural and file system constraints of systems like Hadoop.

3.1 Background

ChordReduce takes its name from the two components it is built upon. Chord [46] provides the backbone for the network and the file system, providing scalable routing, distributed storage, and fault-tolerance. MapReduce runs on top of the Chord network and utilizes the underlying features of the distributed hash table. This section provides background on Chord and MapReduce.

3.1.1 Chord

Chord [?] is a P2P protocol for file sharing that uses a hash function to assign addresses to nodes and files for a ring overlay. The Chord protocol takes in some key and returns the identity (ID) of the node responsible for that key. These keys are generated by hashing a value of the node, such as the IP address and port, or by hashing the filename of a file. The hashing process creates a m-bit hash identifier.

The nodes are then arranged in a ring from the lowest hash-value to highest. Chord takes the files and places each in the node that has the same hashed identifier as it. If no such node exists, the node with the first identifier that follows this value is selected. Since the overlay is a circle, this assignment is computed in modulo 2^m space.

The node responsible for the key κ is called the *successor* of κ, or $successor(\kappa)$. For example, if there were some portion of the network with nodes 20, 25, and 27, node 25 would be responsible for the files with the keys (21,22,23,24,25). If node 25 were to decide to leave the network, its absence would be detected by node 27, who would then be responsible for all the keys node 25 was covering, in addition to its own keys.

With this scheme, we can reliably find the node responsible for some key by asking the next node in the circle for the information, who would then pass the request through the circle until the successor was found. We can then proceed to directly connect with the successor to retrieve the file. This naive approach is largely inefficient, and is a simplification of the lookup process, but it is the basis of how Chord theoretically works.

To speed up the lookup time, each node builds and maintains a *finger table.* The *finger table* contains the locations of up to m other nodes in the ring. The ith entry of node n's *finger table* corresponds to the node that is the $successor(n + 2^{i-1})$ mod 2^m. Hash values are not perfectly

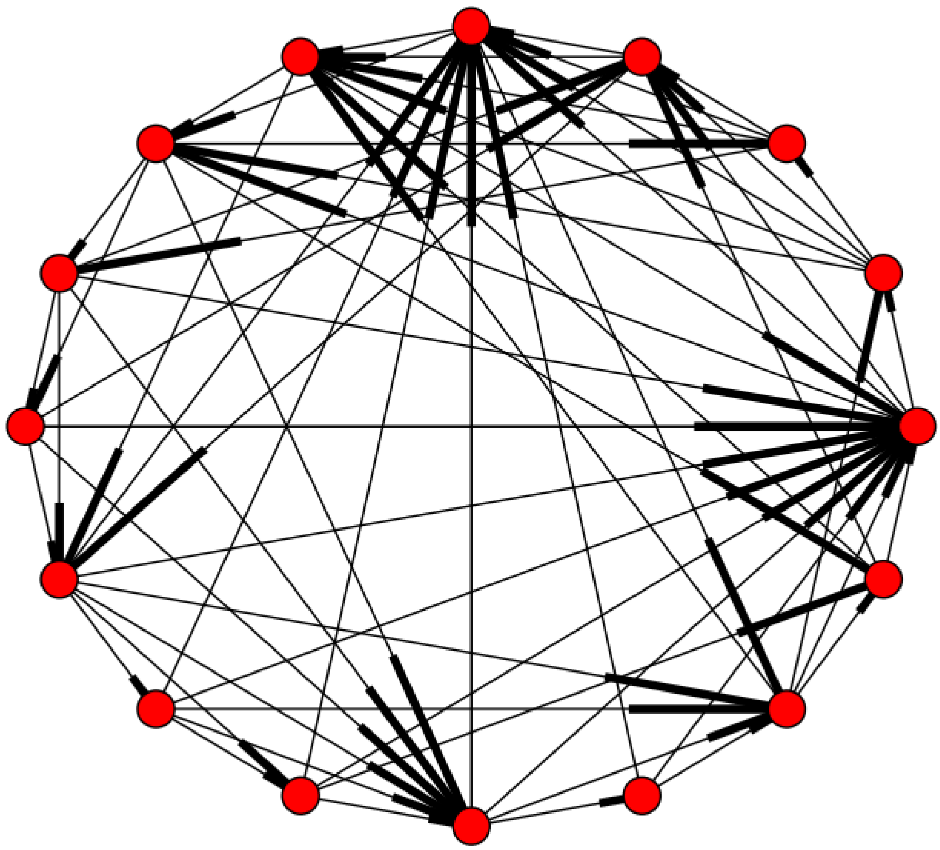

Figure 3.1: A Chord ring with 16 nodes. The bold lines are incoming edges. Each node has a connection to its successor, as well as 4 fingers, some of which are duplicates.

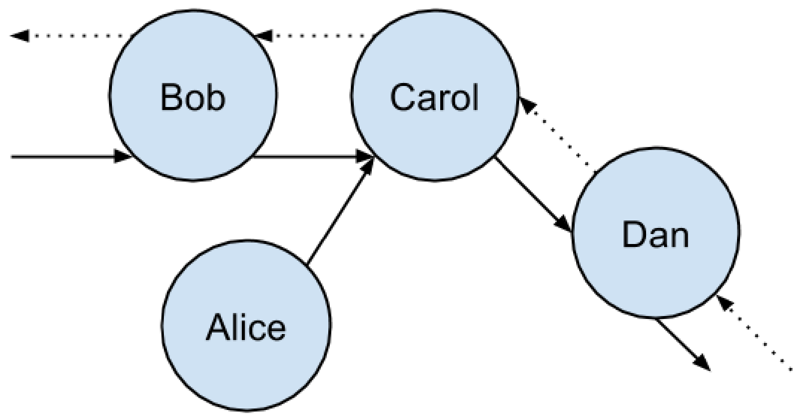

Figure 3.2: Alice has incorrectly determined that Carol is her appropriate successor. When Alice stabilizes, Carol will let her know about Bob.

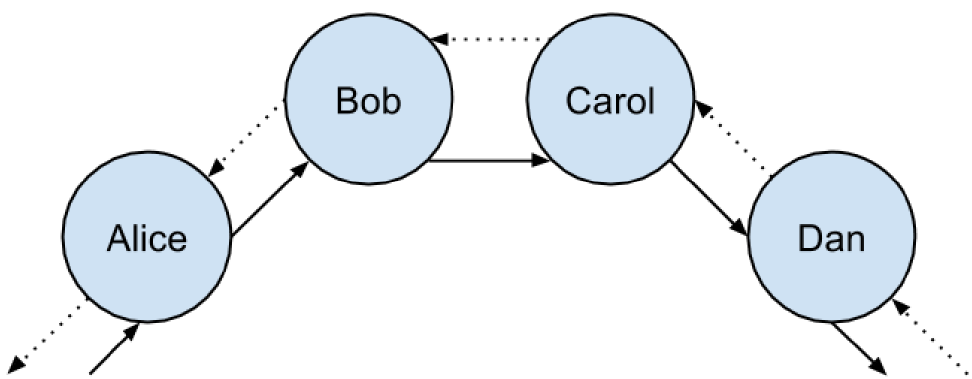

Figure 3.3: After completing stabilize, Alice makes Bob her successor and notifies him. Bob then made Alice as his predecessor.

distributed, it is possible to have duplicate entries in the *finger table*. An example Chord network with fingers is shown in in Fig. **??**.

When a node n is told to find some key, n looks to see if the key is between n and $successor(n)$ and return $successor(n)$'s information to the requester. If not, it looks for the entry in the finger table for the closest preceding node n' it knows and asks n' to find the successor. This allows each step to skip up to half the nodes in the network, giving a $\log_2(n)$ lookup time. Because nodes can constantly join and leave the network, each entry in the table is periodically checked and updated during a finger maintenance period.

To join the network, node n first asks n' to find $successor(n)$ for it. Node n uses the information to set his successor, but the other nodes in the ring will not acknowledge n's presence yet. Node n relies on the stabilize routine to fully integrate into the ring.

The stabilize routine helps the network integrate new nodes and route around nodes who have left the network. Each node periodically checks to see who their successor's predecessor is. In the case of a static network, this would be the checking node. However, if the checking node gets back a different node, it looks at that returned node's hash value and changes its own successor if needed. Regardless of whether the checking node changes its successor, that node then notifies the (possibly) new successor, who then checks if he needs to change his predecessor based on this new information. While complex, the stabilization process is no more expensive than a heartbeat function. A more concrete example:

Suppose Alice, Bob, Carol, and Dan are members of the ring and everyone is ordered alphabetically (Fig. **??**). Alice is quite sure that Carol is her successor. Alice asks Carol who her predecessor is and Carol says Bob is. Since Bob is closer than Carol, Alice changes her successor to Bob and notifies him.

When Bob sees that notification, he can see Alice is closer than whoever his previous predecessor is and sets Alice to be his predecessor. During the next stabilization cycle, Alice will see that she is still Bob's predecessor and notify him that she's still there (Fig. 3.3).

To prevent loss of data due to churn, each node sends a backup of their data to their successor. Section IV discusses the implementation of the backup process in ChordReduce and expands upon it for backing up Map and Reduce tasks.

3.1.2 Extensions of Chord

The Cooperative File System (CFS) is an anonymous, distributed file sharing system built on top of Chord [13]. In CFS, rather than storing an entire file at a single node, the file is split up into multiple chunks around 10 kilbytes in size. These chunks are each assigned a hash and stored in nodes corresponding to their hash in the same way that whole files are. The node that would normally store the whole file instead stores a *key block*, which holds the hash address of the chunks of the file.

The chunking allows for numerous advantages. First, it promotes load balancing. Each piece of the overall file would (ideally) be stored in a different node, each with a different backup or backups. This would prevent any single node from becoming overwhelmed from fulfilling multiple requests for a large file. It would also prevent retrieval from being bottlenecked by a node with a relatively low bandwidth. Finally, when Chord uses some sort of caching scheme like that described in CFS [13], caching chunks as opposed to the entire file resulted in about 1000 times less storage overhead.

Mutable files and IRM, which is short for Integrated File Replication and Consistancy Maintenence, has nodes keep track of file requests they initiate or forward. If they find they are frequently forwarding a request for a particular file, they store that file locally until it is no longer requested frequently. What makes IRM unqiue is that it combines caching with a

Chunking also opens up the options for implementing additional redundancy such as erasure codes [**?**]. With erasure codes, redundant chunks are created but any combination of a particular number of chunks is sufficient to recreate the file. For example, a file that would normally be split into 10 chunks might be split into 15 encoded chunks. The retreival of any 10 of those 15 chunks is enough to recreate the file. Implementing erasure codes would presumably make the network more fault tolerant, but that is an exercise left for future work.

Generally, related files should be kept together; Chord, however, just hashes the filename to find the responsible node and sends it to that location without any thought to organization. Our solution to this is to use allow the file owner to select first 80 bits of a file's hash, then generating the remaining least signifcant bits by hashing the filename. It does not matter if a file owner, in some infintesimally small coincidence, chooses the same 80 bit prefix as another file owner, as the purpose is to keep related files together.

As we have previously discussed, Distributed Hash Tables (DHTs) have an inherent set of qualities, such as greedy routing, maintaining lists of peers which define the topology, and forming 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 that implements a self-organizing web of computational units. 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 geometry, a previously unexplored nontrivial metric space with potential applications, such as latency embedding.

3.2 Introduction

We present UrDHT, an abstract model of a distributed hash table (DHT). It is a unified and cohesive model for creating DHTs and P2P applications based on DHTs.

Distributed Hash Tables have been the catalyst for the creation of many P2P applications. Among these are Redis [3], Freenet [8], and, most notably, BitTorrent [9]. All DHTs use functionally similar protocols to perform lookup, storage, and retrieval operations. Despite this, no one has created a cohesive formal DHT specification.

Our primary motivation for this project was to create an abstracted model for Distributed Hash Tables based on observations we made during previous research [5]. We found that all DHTs can cleanly be mapped to the primal-dual problems of Voronoi Tessellation and Delaunay Triangulation.

UrDHT builds its topology directly upon this insight. It uses a greedy distributed heuristic for approximating Delaunay Triangulations. We found that we could reproduce the topology of different DHTs by defining a selection heuristic and rejection algorithm for the geometry the DHT. For every DHT we implemented, our greedy approximation of Delaunay Triangulation produced a stable DHT, regardless of the geometry. This works in non-Euclidean geometries such as XOR (Kademlia) or even a hyperbolic geometry 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.

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 shared by distributed hash tables, this defines what a DHT does. It does not describe what a DHT is.

  We show that DHTs cleanly map to the primal-dual problem of Delaunay Triangulation and Voronoi Tessellation. 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 3.3).

- We present UrDHT as an abstract DHT and show how a developer would modify the functions we defined to create an arbitrary new DHT topology (Section 3.4).

- We show how to reproduce the topology of Chord and Kademlia using UrDHT. We also implement a DHT in a Euclidean geometry and a hyperbolic geometry represented by a Poincarè disc (Section 3.5).

- We conduct experiments that show building DHTs using UrDHT produced efficiently routable networks, regardless of the underlying geometry(Section 3.6).

- We present some efforts and projects that are similar to our own (Section 3.7).

- We discuss the ramifications of our work and what future work is available (Section 3.8).

3.3 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 DHT are assigned unique[1] keys via a consistent hashing algorithm. To make it easier to intuitively understand the context, we will call the key associated with a node its ID and refer to nodes and their IDs interchangeably.

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

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. The `store` function stores that key/value pair in the DHT, while `get` returns the value associated with that key.

However, these operations define the functionality of a DHT, but do not define the requirements for implementation. We define the necessary components that comprise DHTs. We show that these components are essentially Voronoi Tessellation and Delaunay Triangulation.

3.3.1 DHTs, Delaunay Triangulation, and Voronoi Tessellation

Nodes in different DHTs have, what appears at the first glance, wildly disparate ways of keeping track of peers - the other nodes in the network. However, peers can be split into two groups.

The first group is the *short peers*. These are the closest peers to the node and define the range of keys the node is responsible for. A node is responsible for a key if and only if its ID is closest to the given key in the geometry of the DHT. Short peers define the DHTs topology and guarantee that the greedy routing algorithm shared by all DHTs works.

Long peers are the nodes that allow a DHT to achieve faster routing speeds than the topology would allow using only short peers. This is typically $O(\log(n))$ hops, although polylogarithmic time is acceptable [22]. A DHT can still function without long peers.

Interestingly, despite the diversity of DHT topologies and how each DHT organizes short and long peers, all DHTs use functionally identical greedy routing algorithms (Algorithm 2):

Algorithm 2 The DHT Generic Routing algorithm

function n.LOOKUP((key))
    if $key \in n$'s range of responsibility **then**
        return n
    end if
    if One of n's short peers is 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)
end function

The algorithm is as follows: 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

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

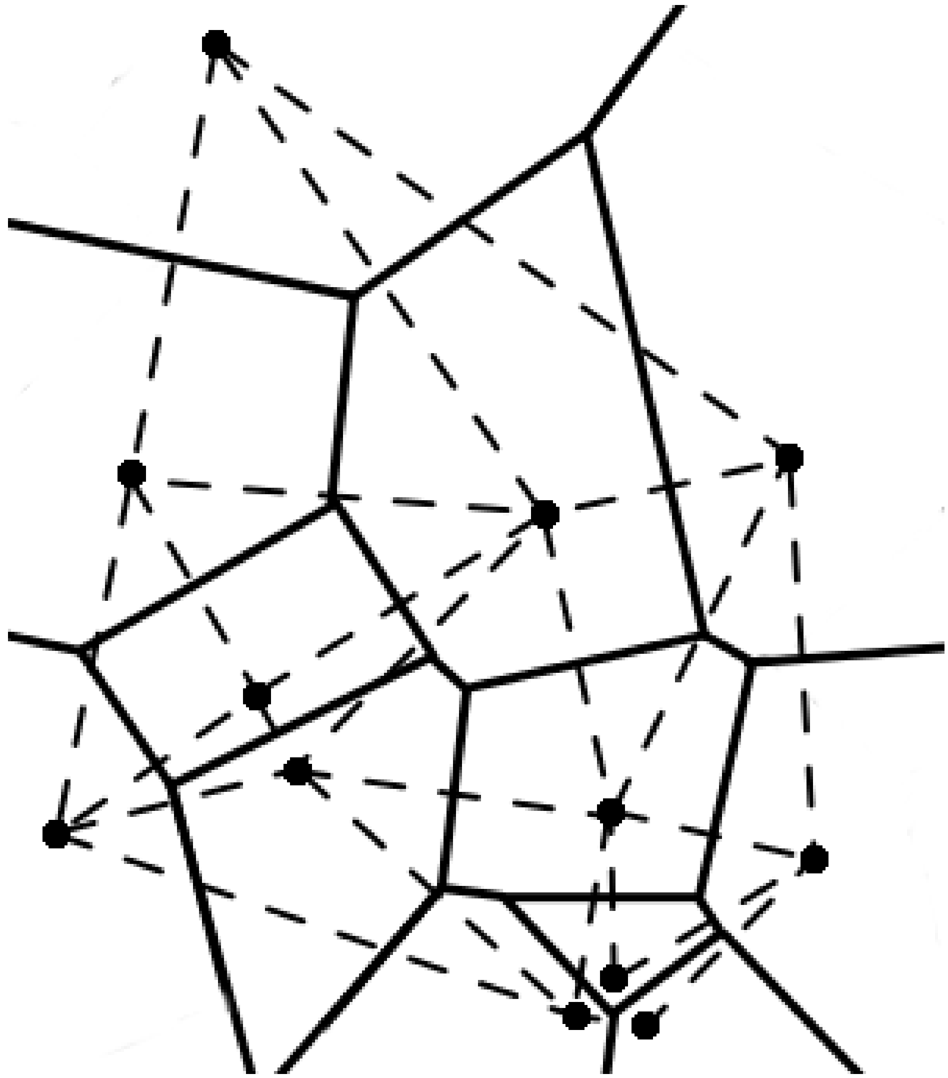

Figure 3.4: An example Voronoi diagram for objects on a 2-dimensional space. The black lines correspond to the borders of the Voronoi region, while the dashed lines correspond to the edges of the Delaunay Triangulation.

this query to the node I know with shortest distance from the node to the desired key.[3]

Depending of the specific DHT, 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 node that no longer exists. This algorithm may possibly be run in parallel, such as in Kademlia [32]. The base greedy algorithm is always the same regardless of the implementation.

With the 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. An example

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

Delaunay Triangulation and Voronoi Tessellation is show in Figure 3.4.

We can map a given node's ID to a point in a space and the set of short peers to the Delaunay Triangulation. This would make the range of keys a node is responsible correspond to the node's Voronoi region. Long peers serve as shortcuts across the mesh formed by Delaunay Triangulation.

Thus, if we can calculate the Delaunay Triangulation between nodes in a DHT, we have a generalized means of creating the overlay network. This can be done with any algorithm that calculates the Delaunay Triangulation.

Computing the Delaunay Triangulation and/or the Voronoi Tessellation 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 [20].

However, DHTs are completed decentralized, with no single node having global knowledge of the topology. Many of the algorithms to compute Delaunay Triangulation and/or Voronoi Tessellation are unsuited to a distributed environment. In addition, the computational 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 [49].

Is there an algorithm we can use to efficiently calculate Delaunay Triangulation for a distributed system in an arbitrary space? We created an algorithm called the Distributed Greedy Voronoi Heuristic (DGVH), explained below [5].

3.3.2 Distributed Greedy Voronoi Heuristic

The Distributed Greedy Voronoi Heuristic (DGVH) is an efficient method for nodes to approximate their individual Voronoi region (Algorithm 3). DGVH selects nearby nodes that would correspond to points connected to it within a Delaunay Triangulation. Our previous implementation relied on a midpoint function [5]. We have refined our heuristic to render a midpoint function unnecessary.

The heuristic is described in Algorithm 3. Every maintenance cycle, nodes exchange their peer lists with their short peers. A node creates a list of candidates by combining their peer lists with their neighbor's peer lists.[4] Sort the list of peers from closest to furthest distance. The node then initializes a new peer list, initially containing the closest candidate. For each of the remaining

[4]In our previous paper, nodes exchange short peer lists with a single peer. Calls to DGVH in this paper use both short and long peer information from all of their short peers.

candidates, the node compares the distance between the current short peers and the candidate. If the new peer list does not contain any short peers closer to the candidate than the node, the candidate is added to the new peer list. Otherwise, the candidate is set aside.

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 3 Distributed Greedy Voronoi Heuristic

Given node n and its list of *candidates*.
Given the minimum *table_size*
short_peers $\leftarrow$ empty set
long_peers $\leftarrow$ empty set
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**
    if any node in *short_peers* is closer to c 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*)

Candidates are gathered via a gossip protocol as well as notifications from other peers. How long peers are handled depends on the particular DHT implementation. This process is described more in Section 3.4.1.

The expected maximum size of *candidates* corresponds to the expected maximum degree of a vertex in a Delaunay Triangulation. This is $\Theta(\frac{\log n}{\log \log n})$, regardless of the number of the dimensions [7]. We can therefore expect *short peers* to be bounded by $\Theta(\frac{\log n}{\log \log n})$.

The expected worst case cost of DGVH is $O(\frac{\log^4 n}{\log^4 \log n})$ [5], regardless of the dimension [5].[5] In most cases, this cost is much lower. Additional details can be found in our previous work [5].

We have tested DGVH on Chord (a ring-based topology), Kademlia (an XOR-based tree topology), general Euclidean spaces, and even in a hyperbolic geometry. This is interesting because not

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

only can we implement the contrived topologies of existing DHTs, but more generalizable topologies like Euclidean or hyperbolic geometries. We show in Section 3.6 that DGVH works in all of these spaces. DGVH only needs the distance function to be defined in order for nodes to perform lookup operations and determine responsibility. We will now show how we used this information and heuristic to create UrDHT, our abstract model for distributed hash tables.

3.4 UrDHT

The name UrDHT comes from the German prefix *ur*, which means "original." The name is inspired by UrDHT's ability to reproduce the topology of other distributed hash tables.

UrDHT is divided into 3 broad components: Storage, Networking, and Logic. Storage handles file storage and Networking dictates the protocol for how nodes communicate. These components oversee 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 site [42].

Most of our discussion will focus on the Logic component. The Logic component is what dictates the behavior of nodes 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.

3.4.1 The DHT Protocol

The DHT Protocol (`LogicClass.py`) [42] is the shared functionality between every single DHT. It consists of the node's information, the short peer list that defines the minimal 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.

Many of the function calls should be familiar to anyone who has study DHTs. We will discuss a few new functions we added and the ones that contribute to node maintenance.

The first thing we note is the absence of `lookup`. In our efforts to further abstract DHTs, we have replaced `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 can perform `lookup` by iteratively calling `seek` until it receives the same answer twice. We do this because we make no assumptions as to how a client using a DHT would want to perform lookups and handle errors that can occur. It also means that a single client implementing `lookup` using iterative `seek` operations could traverse any DHT topology implemented with UrDHT.

Maintenance is done via gossip. Each maintenance cycle, the node recalculates its Delaunay (short) peers using its neighbors' peer lists and any nodes that have notified it since the last maintenance cycle. Short peer selection are done using DGVH by default. While DGVH has worked in every single space we have tested, this is not proof it will work in every single case. It is reasonable and expected that some spaces may require a different Delaunay Triangulation calculation or approximation method.

Once the short peers are calculated, the node handles modifying its long peers. This is done using the `handleLongPeers` function described in Section 3.4.2.

3.4.2 The Space Math

The Space Math consists of the functions that define the DHT's topology. It requires a way to generate short peers to form a routable overlay and a way to choose long peers. Space Math requires the following functions when using DGVH:

- The `idToPoint` function takes in a node's ID and any other attributes needed to map an ID onto a point in the space. The ID is generally a large integer generated by a cryptographic hash function.

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

- We use the above functions to implement `getDelaunayPeers`. Given a set of points, the *candidates*, and a center point *centers*, `getDelaunayPeers` calculates a mesh that approximates the Delaunay peers of *center*. We assume that this is done using DGVH (Algorithm 3).

- The function `getClosest` returns the point closest to *center* from a list of *candidates*, measured by the distance function. The `seek` operation depends on the `getClosest` function.

- The final function is `handleLongPeers`. `handleLongPeers` takes in a list of *candidates* and a *center*, much like `getDelaunayPeers`, and returns a set of peers to act as the routing shortcuts.

  The implementation of this function should vary greatly from one DHT to another. For example, Symphony [30] and other small-world networks [23] choose long peers using a probability distribution. Chord has a much more structured distribution, with each long peer being increasing powers of 2 distance away from the node [46]. The default behavior is to use all candidates not chosen as short peers as long peers, up to a set maximum. If the size of long peers would exceed this maximum, we instead choose a random subset of the maximum size, creating a naive approximation of the long links in the Kleinberg small-world model [23]. Long peers do not greatly contribute to maintenance overhead, so we chose 200 long peers as a default maximum.

3.5 Implementing other DHTs

3.5.1 Implementing Chord

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

Chord uses a unidirectional distance function. Given two integer keys a and b and a maximum value 2^m, the `distance` from a to b in Chord is:

$$distance(a,b) = \begin{cases} 2^m + b - a, & \text{if } b - a < 0 \\ b - a, & \text{otherwise} \end{cases}$$

Short peer selection is trivial in Chord, so rather than using DGVH for `getDelaunayPeers`, each node chooses from the list of candidates the candidate closest to it (predecessor) and the candidate to which it is closest (successor).

Chord's finger (long peer) selection strategy is emulated by `handleLongPeers`. For each of the ith bits in the hash function, we choose a long peer p_i from the candidates such that

$$p_i = getClosest\,(candidates, t_i)$$

where

$$t_i = (n + 2^i) \mod 2^m$$

for the current node n. The `getClosest` function in Chord should return the candidate with the shortest distance from the candidate to the point.

This differs slightly from how selects its long peers. In Chord, nodes actively seek out the appropriate long peer for each corresponding bit. In our emulation, this information is propagated along the ring using short peer gossip.

3.5.2 Implementing Kademlia

Kademlia uses the exclusive or, or XOR, metric for distance. This metric, while non-euclidean, is perfectly acceptable for calculating distance. For two given keys a and b

$$distance(a, b) = a \oplus b$$

The `getDelaunayPeers` function uses DGVH as normal to choose the short peers for node n. We then used Kademlia's k-bucket strategy [32] for `handleLongPeers`. The remaining candidates are placed into buckets, each capable holding a maximum of k long peers.

To summarize briefly, node n starts with a single bucket containing itself, covering long peers for the entire range. When attempting to add a candidate to a bucket already containing k long peers, if the bucket contains node n, the bucket is split into two buckets, each covering half of that bucket's range. Further details of how Kademlia k-buckets work can be found in the Kademlia protocol paper [32].

3.5.3 ZHT

ZHT [27] leads to an extremely trivial implementation in UrDHT. Unlike other DHTs, ZHT assumes an extremely low rate of churn. It bases this rationale on the fact that tracking $O(n)$ peers in memory is trivial. This indicates the $O(\log n)$ memory requirement for other DHTs is overzealous and not based on a memory limitation. Rather, the primary motivation for keeping a number of peers in memory is more due to the cost of maintenance overhead. ZHT shows, that by assuming low rates of churn (and infrequent maintenance messages as a result), having $O(n)$ peers is a viable tactic for faster lookups.

As a result, the topology of ZHT is a clique, with each node having an edge to all other nodes. This yields $O(1)$ lookup times with an $O(n)$ memory cost. The only change that needs to be made to UrDHT is to accept all peer candidates as short peers.

3.5.4 Implementing a DHT in a non-contrived Metric Space

We used a Euclidean geometry as the default space when building UrDHT and DGVH [5]. For two vectors $\vec{a}$ and $\vec{b}$ in d dimensions:

$$distance\left(\vec{a}, \vec{b}\right) = \sqrt{\sum_{i \in d} (a_i - b_i)^2}$$

We implement `getDelaunayPeers` using DGHV and set the minimum number of short peers to $3d + 1$, a value we found through experimentation [5].

Long peers are randomly selected from the left-over candidates after DGVH is performed [5]. The maximum size of long peers is set to $(3d + 1)^2$, but it can be lowered or eliminated if desired and maintain $O(\sqrt[d]{n})$ routing time.

Generalized spaces such as Euclidean space allow the assignment of meaning to arbitrary dimension and allow for the potential for efficient querying of a database stored in a DHT.

We have already shown with Kademlia that UrDHT can operate in a non-Euclidean geometry. Another non-euclidean geometry UrDHT can work in is a hyperbolic geometry.

We implemented a DHT within a hyperbolic geometry using a Poincarè disc model. To do this, we implemented `idToPoint` to create a random point in Euclidean space from a uniform distribution. This point is then mapped to a Poincarè disc model to determine the appropriate

Delaunay peers. For any two given points a and b in a Euclidean vector space, the `distance` in the Poincarè disc is:

$$distance(a,b) = \operatorname{arcosh}\left(1 + 2\frac{\|a-b\|^2}{(1-\|a\|^2)(1-\|b\|^2)}\right)$$

Now that we have a `distance` function, DGVH can be used in `getDelaunayPeers` to generate an approximate Delaunay Triangulation for the space. The `getDelaunayPeers` and `handleLongPeers` functions are otherwise implemented exactly as they were for Euclidean spaces.

Implementing a DHT in hyperbolic geometry has many interesting implications. Of particular note, embedding into hyperbolic spaces allows us to explore accurate embeddings of internode latency into the metric space [24] [12]. This has the potential to allow for minimal latency DHTs.

3.6 Experiments

We use simulations to test our implementations of DHTs using UrDHT. Using simulations to test the correctness and relative performance of DHTs is standard practice for testing and analyzing DHTs [32] [30] [46] [51] [4] [25].

We tested four different topologies: Chord, Kademlia, a Euclidean geometry, and a Hyperbolic geometry. For Kademlia, the size of the k-buckets was 3. In the Euclidean and Hyperbolic geometries, we set a minimum of 7 short peers and a maximum of 49 long peers.

We created 500 node networks, starting with a single node and adding a node each maintenance cycle.[6]

For each topology, at each step, we measured:

- The average degree of the network. This is the number of outgoing links and includes both short and long peers.
- The worst case degree of the network.
- The average number of hops between nodes using greedy routing.

[6]We varied the amount of maintenance cycles between joins in our experiments, but found it had no effect upon our results.

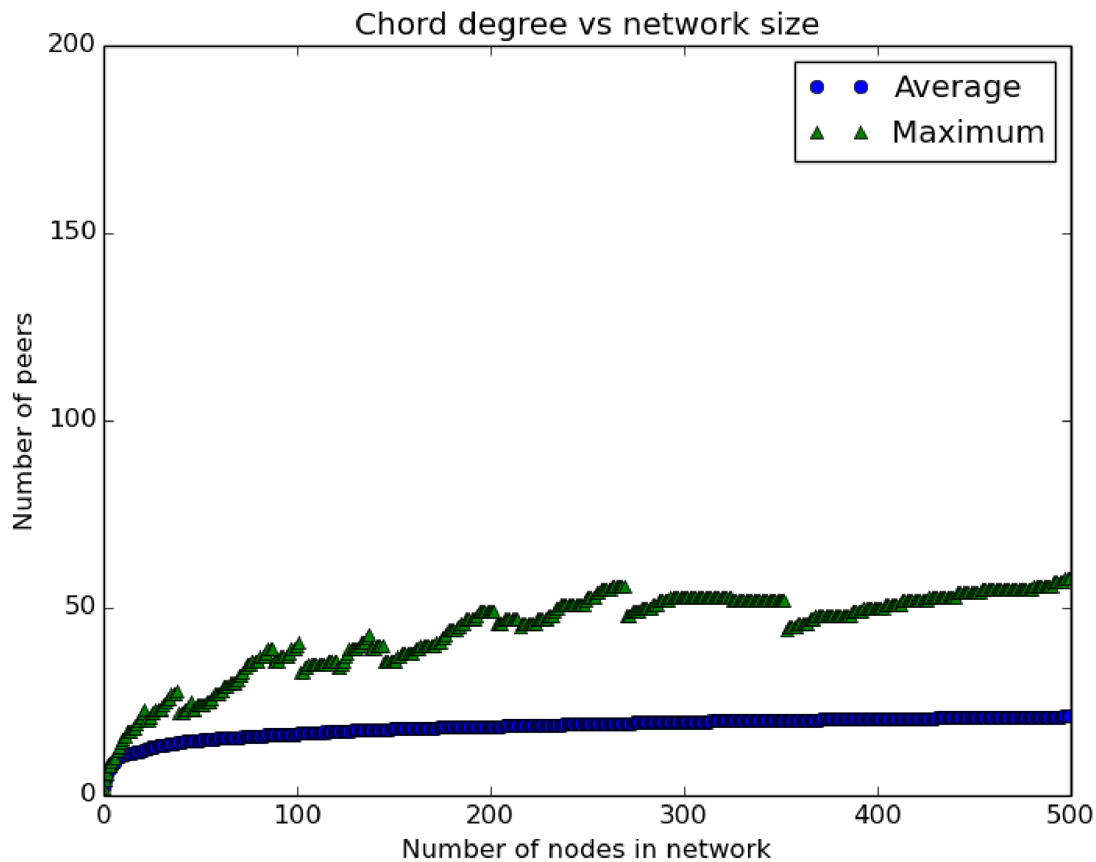

Figure 3.5: This is the average and maximum degree of nodes in the Chord network. This Chord network utilized a 120 bit hash and thus degree is bound at 122 (full fingers, predecessor and successor) when the network reaches 2^{120} nodes.

- The diameter of the network. This is the worst case distance between two nodes using greedy routing.

We also tested the reachability of nodes in the network. At every step, the network is fully reachable.

Results generated by the Chord and Kademlia simulations were in line with those from previous work [32] [46]. This demonstrates that UrDHT is capable of accurately emulating these topologies. We show these results in Figures 3.5 - 3.8.

The results of our Euclidean and Hyperbolic geometries indicate similar asymptotic behavior: a higher degree produces a lower diameter and average routing. However, the ability to leverage this trade-off is limited by the necessity of maintaining an $O(\log n)$ degree. These results are shown

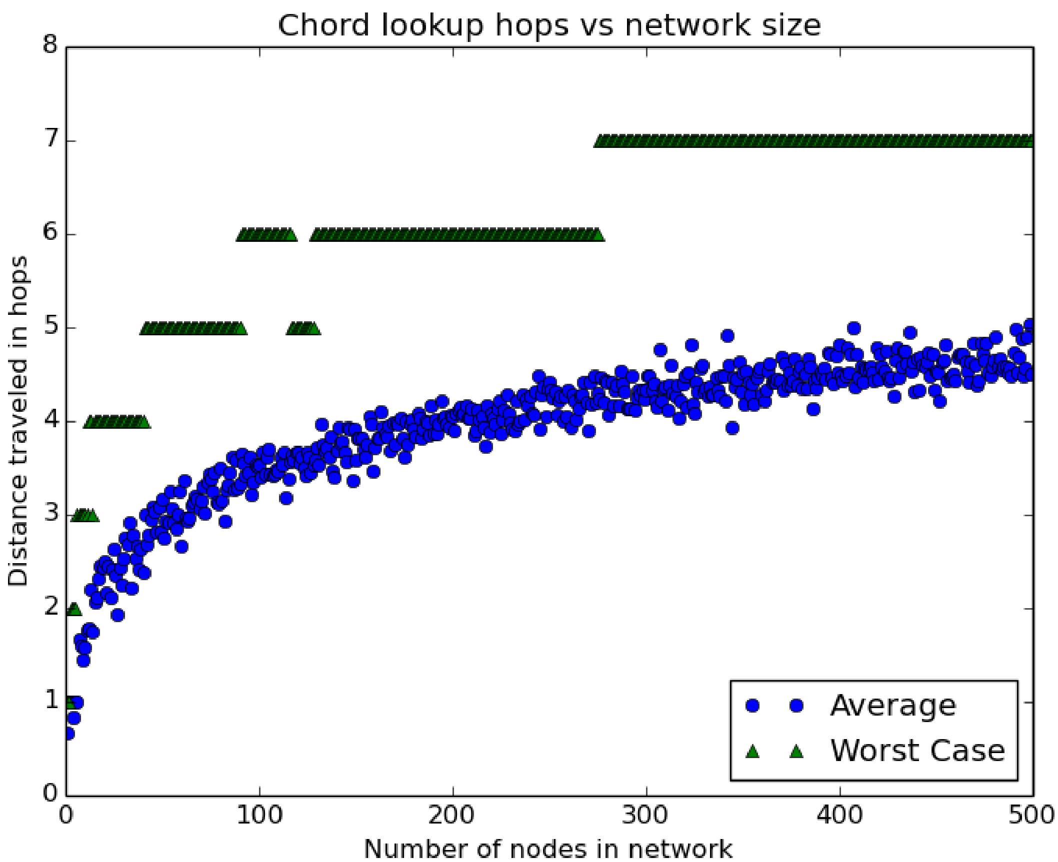

Figure 3.6: This is the number hops required for a greedy routed lookup in Chord. The average lookup between two nodes follows the expected logarithmic curve.

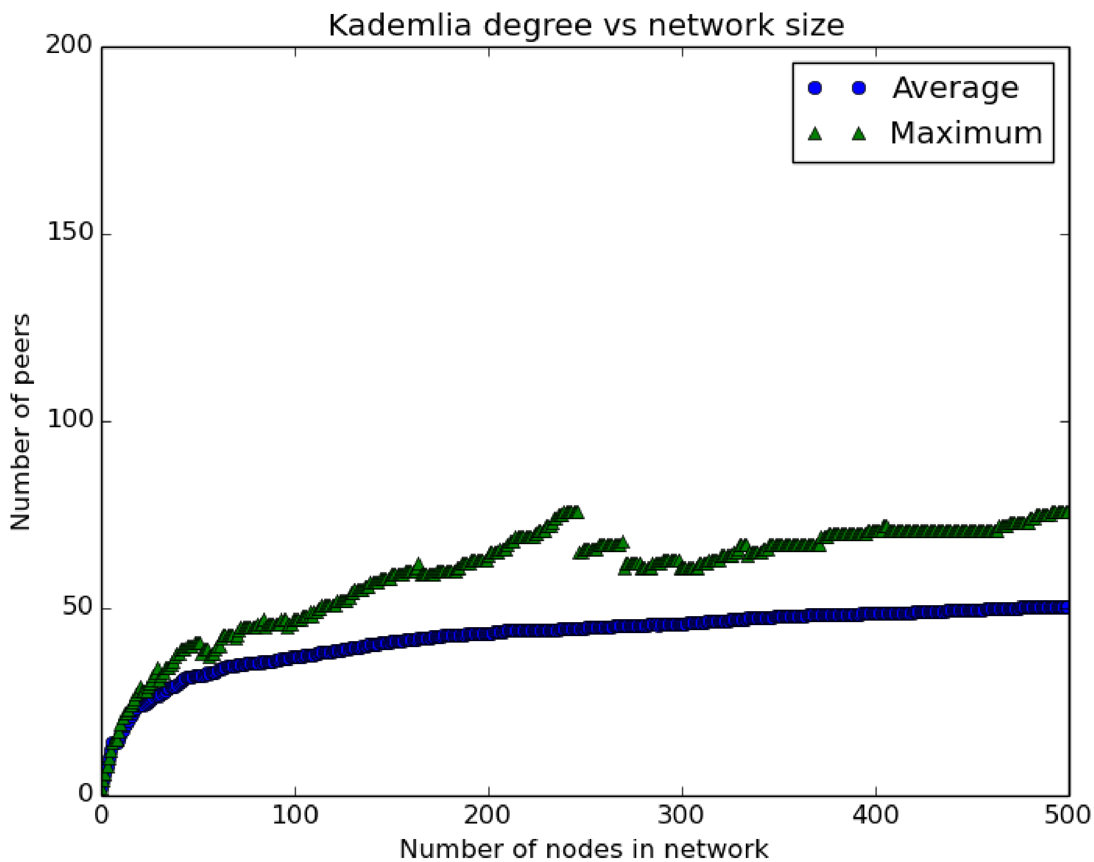

Figure 3.7: This is the average and maximum degree of nodes in the Kademlia network as new nodes are added. Both the maximum degree and average degree are $O(\log n)$.

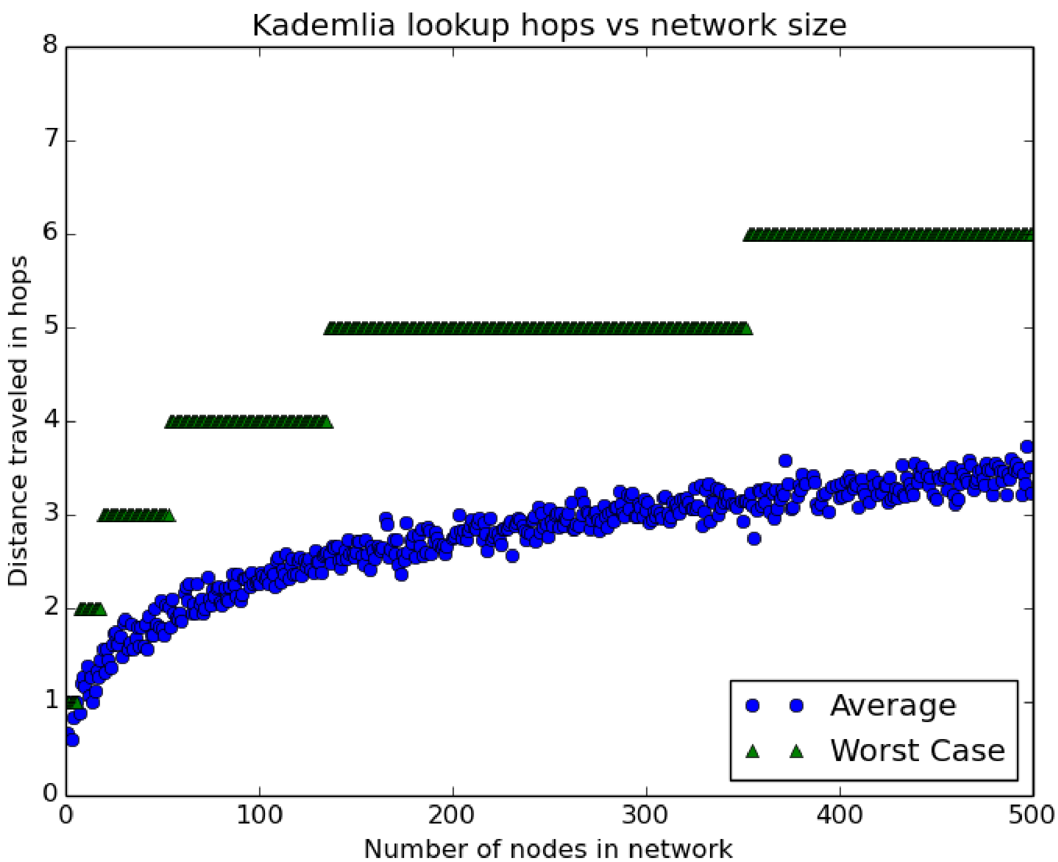

Figure 3.8: Much like Chord, the average degree follows a distinct logarithmic curve, reaching an average distance of approximately three hops when there are 500 nodes in the network.

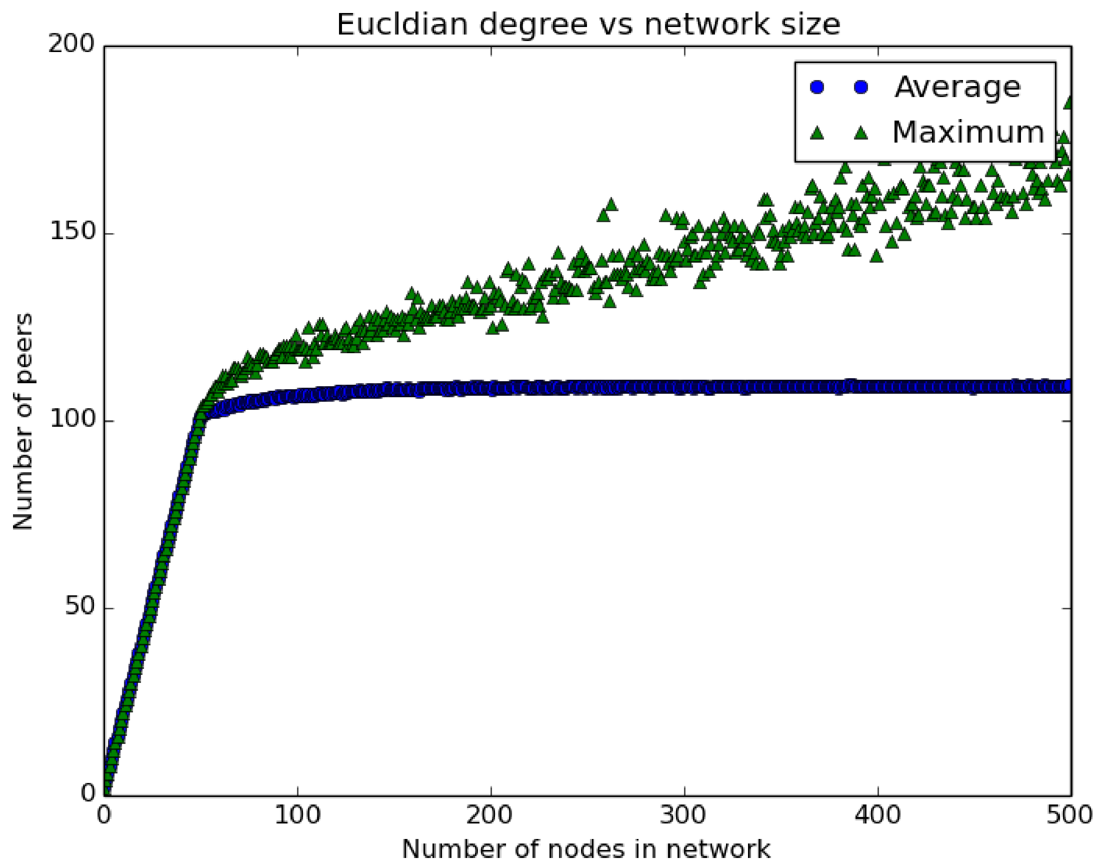

Figure 3.9: Because the long peers increase linearly to the maximum value (49), degree initially rises quickly and then grows more slowly as the number of long peers ceases to grow and the size short peers increases with network size.

in Figures 3.9 - 3.12.

While we maintain the number of links must be $O(\log n)$, all DHTs practically bound this number by a constant. For example, in Chord, this is the number of bits in the hash function plus the number of predecessors/successors. Chord and Kademlia fill this bound asymptotically. The long peer strategy used by the Euclidean and Hyperbolic metrics aggressively filled to this capacity, relying on the distribution of long peers to change as the network increased in size rather than increasing the number of utilized long peers. This explains why the Euclidean and Hyperbolic spaces have more peers (and thus lower diameter) for a given network size. This presents a strategy for trade-off of the network diameter vs. the overhead maintenance cost.

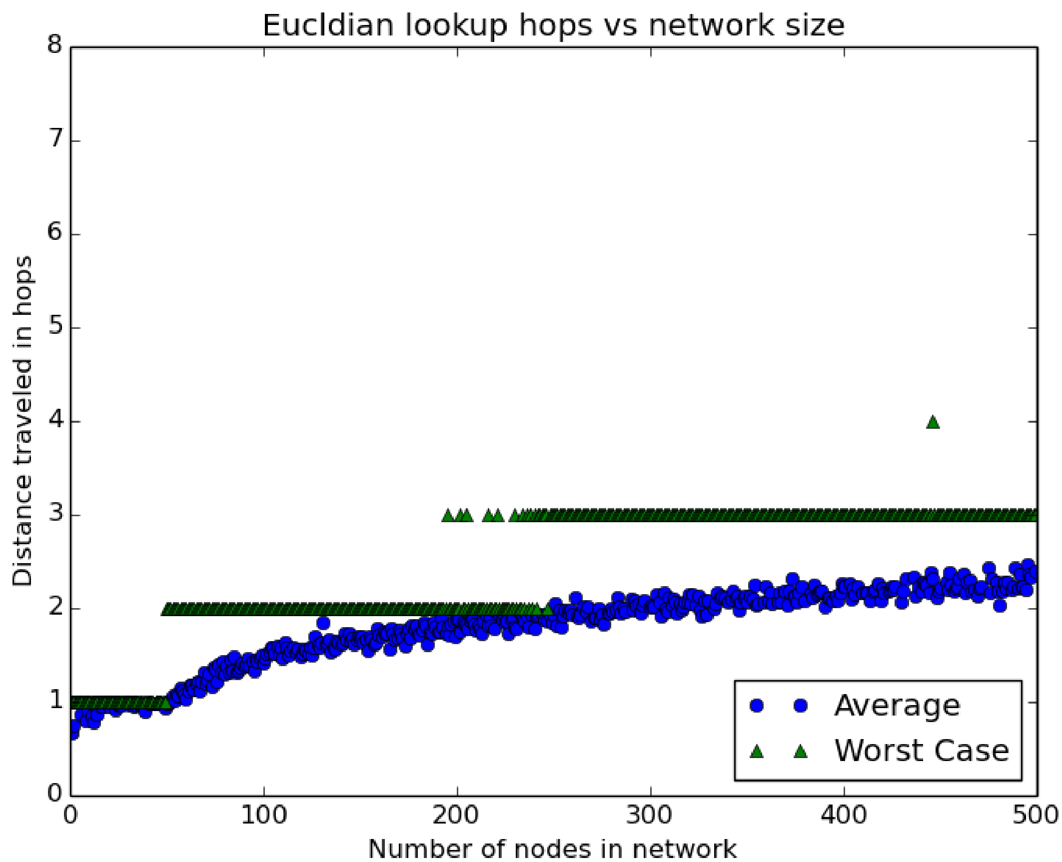

Figure 3.10: The inter-node distance stays constant at 1 until long peers are filled, then rises at the rate of a randomly connected network due to the distribution of long peers selected

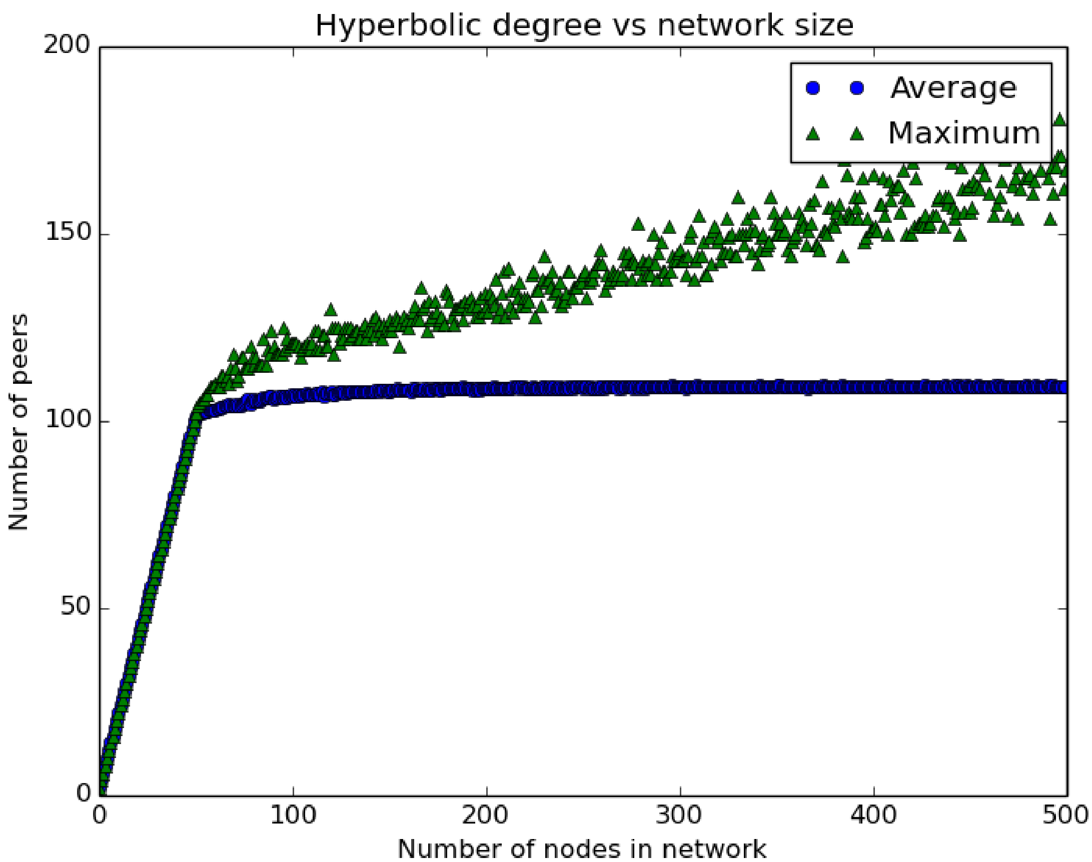

Figure 3.11: The Hyperbolic network uses the same long and short peer strategies to the Euclidean network, and thus shows similar results.

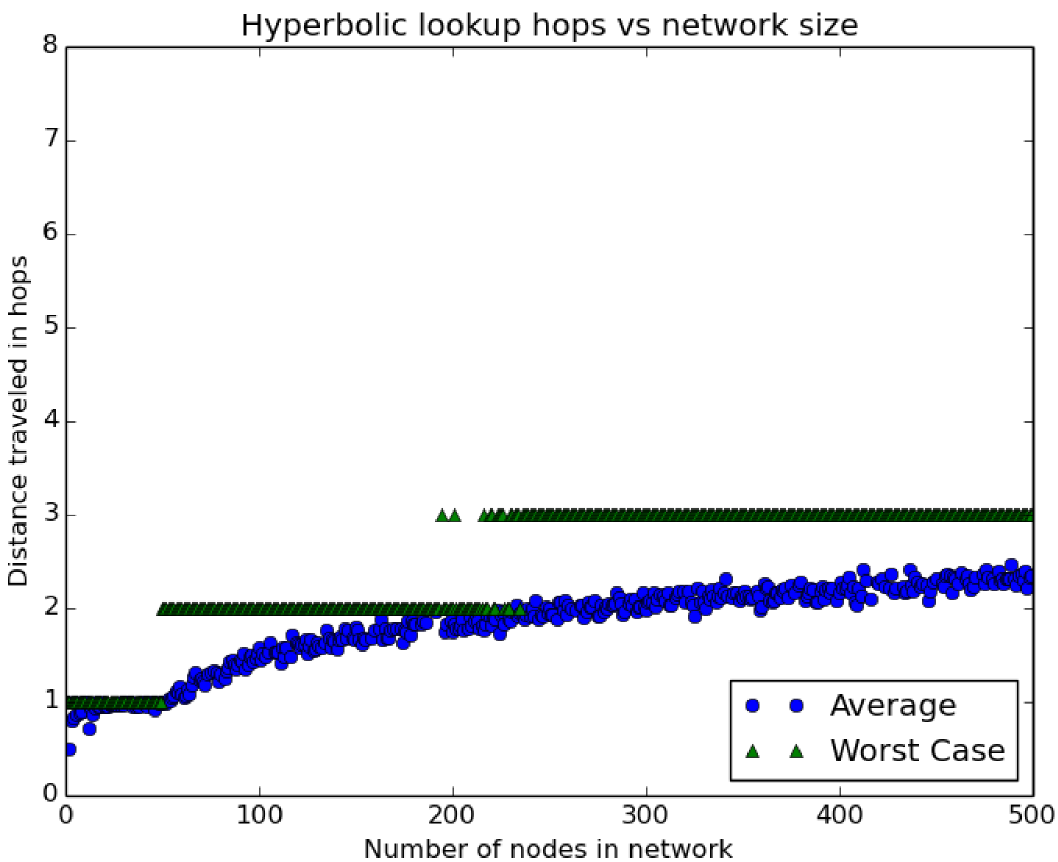

Figure 3.12: Like the Euclidean Geometry, our Poincarè disc based topology has much shorter maximum and average distances.

3.7 Related Work

There have been a number of efforts to either create abstractions of DHTs or ease the development of DHTs. One area of previous work focused on constructing overlay networks using system called P2 [1] [29]. P2 is a network engine for constructing overlays which uses the Overlog declarative logic language. Writing programs for P2 in Overlog yields extremely concise and modular implementations of for overlay networks.

Our work differs in that P2 attempts to abstract overlays and ease construction by using a language and framework. while UrDHT focuses on abstracting the idea of a structured overlay into Voronoi Tessellations and Delaunay Triangulations. This allows developers to define the overlays they are building by mathematically defining a short number of functions.

Our use case is also subtly different. P2 focuses on overlays in general, all types of overlays. UrDHT concerns itself solely with distributed hash tables, specifically, overlays that rely on hash functions to distribute the load of the network and assign responsibility in an autonomous manner.

One difficulty in using P2 is that it is no longer supported as a project [1]. P2's concise Overlog statements also present a sharp learning curve for many developers. These present challenges not seen with UrDHT.

The T-Man[21] and Vicinity [48] protocols both present gossip-based methods for organizing overlay networks. The idea behind T-Man is similar to UrDHT, but again it focuses on overlays in general, while UrDHT applies specifically to DHTs. The ranking function is similar to the metrics used by UrDHT using DGVH, but DGVH guarantees full connectivity in all cases and is based on the inherent relationship between Voronoi Tessellations, Delaunay Triangulations, and DHTs.

UrDHT uses a gossiping protocol similar to the ones presented by T-Man and Vicinity due to they gossip protocol's ability to rapidly adjust changes in the topology.

3.8 Applications and Future Work

We presented UrDHT, a unified model for DHTs and framework for building distributed applications. We have shown how it possible to use UrDHT to not only implement traditional DHTs such as Chord and Kademlia, but also in much more generalized spaces such as Euclidean and Hyperbolic geometries. The viability of UrDHT to utilize Euclidean and Hyperbolic metric spaces indicates that further research into potential topologies of DHTs and potential applications of these topologies is warranted.

There are numerous routes we can take with our model. Of particular interest are the applications of building 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 could happily route a message from one continent to another and back. This is obviously undesirable, but it is the status quo in DHTs. The reason for this stems from the generation of node IDs in DHTs. Nodes are typically assigned a point in the range of a cryptographic hash function. The ID corresponds to the hash of some identifier or given a point randomly. This is done for purposes of load balancing and fault tolerance.

For future work, 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 [24] [12]. 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. This approach would maintain the decentralized strengths of DHTs, while reducing overall delay and communication costs.

Bibliography

[1] Getting started with overlog and p2. `https://github.com/declarativitydotnet/p2/blob/master/doc/tutorial`.

[2] Hadoop. http://hadoop.apache.org/.

[3] Redis. `http://redis.io`.

[4] Olivier Beaumont, Anne-Marie Kermarrec, and Étienne Rivière. Peer to peer multidimensional overlays: Approximating complex structures. In *Principles of Distributed Systems*, pages 315–328. Springer, 2007.

[5] 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*.

[6] Brendan Benshoof, Andrew Rosen, Anu G. Bourgeois, and Robert W Harrison. Vhash: Spatial dht based on voronoi tessellation. In *2nd International IBM Cloud Academy Conference*.

[7] 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.

[8] 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.

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

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

[11] Russ Cox, Athicha Muthitacharoen, and Robert T Morris. Serving dns using a peer-to-peer lookup service. In *Peer-to-Peer Systems*, pages 155–165. Springer, 2002.

[12] Andrej Cvetkovski and Mark Crovella. Hyperbolic embedding and routing for dynamic graphs. In *INFOCOM 2009, IEEE*, pages 1647–1655. IEEE, 2009.

[13] Frank Dabek, M Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-Area Cooperative Storage with CFS. *ACM SIGOPS Operating Systems Review*, 35(5):202–215, 2001.

[14] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[15] 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.

[16] John R Douceur. The sybil attack. In *Peer-to-peer Systems*, pages 251–260. Springer, 2002.

[17] Erin-Elizabeth A Durham, Andrew Rosen, and Robert W Harrison. A model architecture for big data applications using relational databases. In *Big Data (Big Data), 2014 IEEE International Conference on*, pages 9–16. IEEE, 2014.

[18] Erin-Elizabeth A Durham, Andrew Rosen, and Robert W Harrison. Optimization of relational database usage involving big data a model architecture for big data applications. In *Computational Intelligence and Data Mining (CIDM), 2014 IEEE Symposium on*, pages 454–462. IEEE, 2014.

[19] Donald Eastlake and Paul Jones. Us secure hash algorithm 1 (sha1), 2001.

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

[21] Márk Jelasity and Ozalp Babaoglu. T-man: Gossip-based overlay topology management. In *Engineering Self-Organising Systems*, pages 1–15. Springer, 2005.

[22] 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.

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

[24] Robert Kleinberg. Geographic routing using hyperbolic space. In *INFOCOM 2007. 26th IEEE International Conference on Computer Communications. IEEE*, pages 1902–1909. IEEE, 2007.

[25] Jinyang Li, Jeremy Stribling, Thomer M Gil, Robert Morris, and M Frans Kaashoek. Comparing the performance of distributed hash tables under churn. In *Peer-to-Peer Systems III*, pages 87–99. Springer, 2005.

[26] Mu Li, Li Zhou, Zichao Yang, Aaron Li, Fei Xia, David G Andersen, and Alexander Smola. Parameter server for distributed machine learning.

[27] 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.

[28] Andrew Loewenstern and Arvid Norberg. BEP 5: DHT Protocol. `http://www.bittorrent.org/beps/bep_0005.html`, March 2013.

[29] Boon Thau Loo, Tyson Condie, Joseph M Hellerstein, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Implementing declarative overlays. In *ACM SIGOPS Operating Systems Review*, volume 39, pages 75–90. ACM, 2005.

[30] 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.

[31] Gabriel Mateescu, Wolfgang Gentzsch, and Calvin J. Ribbens. Hybrid computingwhere {HPC} meets grid and cloud computing. *Future Generation Computer Systems*, 27(5):440 – 453, 2011.

[32] 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.

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

[34] Jacob Jan-David Mol, Arno Bakker, Johan A Pouwelse, Dick HJ Epema, and Henk J Sips. The design and deployment of a bittorrent live video streaming solution. In *Multimedia, 2009. ISM'09. 11th IEEE International Symposium on*, pages 342–349. IEEE, 2009.

[35] Vasileios Pappas, Daniel Massey, Andreas Terzis, and Lixia Zhang. A comparative study of the dns design with dht-based alternatives. In *INFOCOM*, volume 6, pages 1–13, 2006.

[36] 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.

[37] 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.

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

[39] 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.

[40] Andrew Rosen, Brendan Benshoof, Robert W Harrison, and Anu G. Bourgeois. Mapreduce on a chord distributed hash table. In *2nd International IBM Cloud Academy Conference*.

[41] Andrew Rosen, Brendan Benshoof, Robert W Harrison, and Anu G. Bourgeois. The sybil attack on peer-to-peer networks from the attacker's perspective.

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

[43] 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.

[44] Sherif Saad, Issa Traore, Ali Ghorbani, Bassam Sayed, David Zhao, Wei Lu, John Felix, and Payman Hakimian. Detecting p2p botnets through network behavior analysis and machine learning. In *Privacy, Security and Trust (PST), 2011 Ninth Annual International Conference on*, pages 174–180. IEEE, 2011.

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

[46] 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.

[47] Chinua Umoja, JT Torrance, Erin-Elizabeth A Durham, Andrew Rosen, and Robert W Harrison. A novel approach to determine docking locations using fuzzy logic and shape determination. In *Big Data (Big Data), 2014 IEEE International Conference on*, pages 14–16. IEEE, 2014.

[48] Spyros Voulgaris and Maarten Van Steen. Epidemic-style management of semantic overlays for content-based searching. In *Euro-Par 2005 Parallel Processing*, pages 1143–1152. Springer, 2005.

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

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

[51] 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.