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 [**?**], which is built on Mainline DHT [**?**].
- DHTs have been used for *distributed storage* systems [**?**].
- *Distributed Domain Name Systems* (DNS) have been built upon DHTs [**?**] [**?**]. 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* [**?**].

[1] As far as we know.

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

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 [**?**], 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 [**?**], 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 [**?**]. 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 [**?**], 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 [**?**].

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 [**?**]. 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 Outline

In this section, we give a brief overview of our work and the contents of this tome. Chapter 2 lays out the prerequisite knowledge for Distributed Hash Tables.

1.4.1 Completed Work

One of our first projects was to create a distributed computing platform using the Chord DHT [**?**]. 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 8, but suggested to us that there was a way to dynamically load-balance during execution.

Our second project was to develop VHash [**?**] [**?**], 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 [**?**]. The Sybil attack [**?**] 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 [**?**]
- 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 [**?**]
- 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 [**?**]
- Brendan Benshoof, Andrew Rosen, Anu G. Bourgeois, Robert W. Harrison "Distributed Decentralized Domain Name Service" IEEE IPDPS 2016 - Workshop on Dependable Parallel, Distributed and Network-Centric Systems

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 work discussed in this dissertation.

- 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 [**?**]
- 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 [**?**]
- 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 [**?**]

1.4.2 Summary of Dissertation

The dissertation is divided into distinct, but mutually dependent parts. Chapter 2 covers the requisite background material. Each subsequent chapter, summarized below, covers a specific project. Chapter 9, closes with remarks on our work.

ChordReduce - DHT Distributed Computing

We present our first project, ChordReduce [**?**], in Chapter 3. ChordReduce utilized Chord [**?**] to create a distributed computing platform using the MapReduce [**?**] platform. The novel contribution of ChordReduce was it's ability to perform completely decentralized MapReduce operations in either a P2P environment or a datacenter.

Using our created framework, we will create implement and test distributed computing problems on different DHT implementations, such as Chord [**?**] and Kademlia [**?**].

DGVH and VHash

Chapter 4 covers the origin of our Distributed Greedy Voronoi Heuristic (DGVH). This provides an efficient way to create (with some error) Voronoi Tessellations and the corresponding Delaunay

Triangulation. While DGVH produces only approximates the solution for Voronoi Tessellations, it can do so in any geometric space with distance function in any number of dimensions. In addition, the computational complexity is independent of the number of dimensions and can be performed locally at each node, rather than requiring global knowledge of the network's state.

This directly leads into Chapter 5, which using DGVH to abstract DHTs.

UrDHT – an Abstract DHT Framework

In Chapter 5, we found that DHTs can be mapped to the constructs of Delaunay Triangulation and Voronoi Tessellation. Thus, creating the topology of a DHT can be done by solving for the appropriate Delaunay Triangulation. UrDHT is an open source project for creating DHTs using this principle.

Distributed Decentralized Domain Name Service

D^3NS (Chapter 6) is a system to replace the current top level DNS system and certificate authorities, offering increased scalability, security and robustness. D^3NS is based on a distributed hash table and utilizes a domain name ownership system based on the Bitcoin blockchain [**?**] and addresses previous criticism that a DHT would not suffice as a DNS replacement.

Our system provides solutions to current DNS vulnerabilities such as DDOS attacks, DNS spoofing and censorship by local governments. It eliminates the need for certificate authorities by providing a decentralized authenticated record of domain name ownership. Unlike many other DNS replacement proposals, D^3NS is reverse compatible with DNS and allows for incremental implementation within the current system.

Sybil Analysis

Chapter 7 analyses the cost of performing a Sybil attack from the perspective of an adversary. Previous analyses have all focused on defending from the aforementioned adversary, but little to no work has been performed quantifying the attacker's capabilities. Our analysis places reasonable constraints on the attacker and evaluates the resources the attacker needs to effectively own a target network.

Autonomous Load-Balancing

Chapter 8 8 presents strategies for nodes to balance the workload among members of the DHT. Load balancing schemes do exist for file storage, but none exist for computation. Furthermore, we wanted 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 [?] [?] [?] [?] [?] [?] [?]. 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 trade-offs 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 [**?**] 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.

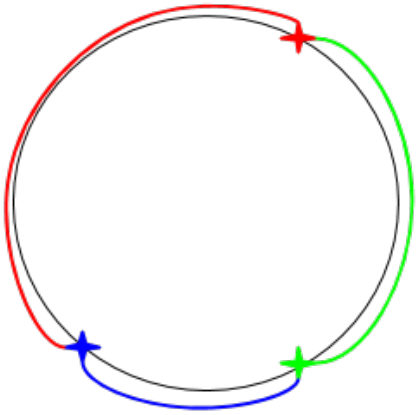


Figure 2.1: A Voronoi diagram for a Chord network, using Chord's definition of closest.

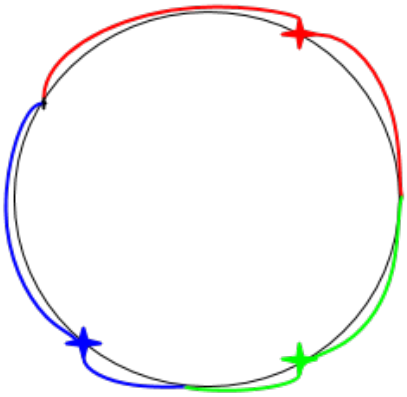


Figure 2.2: A Voronoi diagram for a Chord network, where closest is defined by the node being the closest in either direction.

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 dissertation, 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 [**?**] [**?**] 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 [**?**], 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.

[1] Unique with extremely high probability. The probability of a hash collision is extremely low and are ignored in most formal specifications for DHTs. This could be resolved for any file by using any number of the collision resolution strategies, such as chaining or linear probing. However, resolving a collision of two nodes is much more problematic with no canonical solution other than praying it won't happen.

[2] Due to the research into hash collisions [**?**], and the glut of hardware that currently exists to perform SHA hash collisions, SHA1 is being depreciated by many companies in 2017. This will undoubtedly lead to some kind of security flaw in a decade or so, when some entrepreneuring hacker figures out a way to force websites to accept a forged SHA1 key.

n **nodes** - The number of nodes in the network.

Similarly, All DHTs perform the same operations with minor variation.

`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

[3]check this statement for consistency

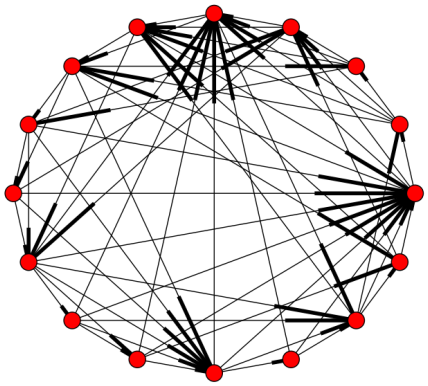


Figure 2.3: A Chord ring with 16 nodes. The fingers (long hop connections) are shown cutting across the ring.

politely leave the network but always abruptly fail, since a `leave()` operation is fairly trivial and has minimal impact.

2.2 Chord

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

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 [**?**], 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 [**?**].

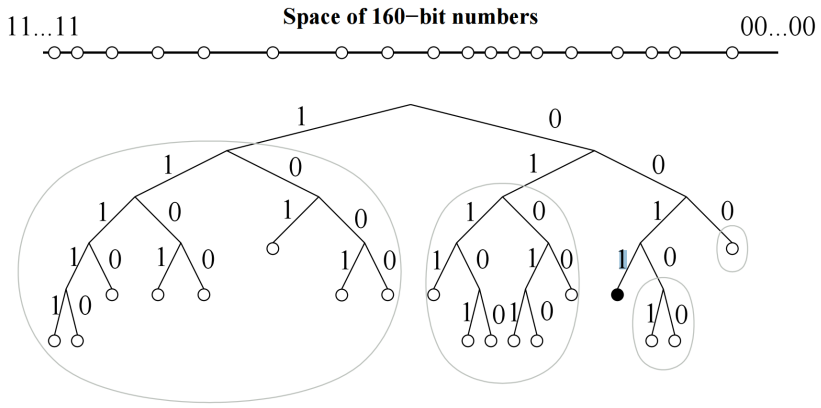


Figure 2.4: An example Kademlia network from the original paper [**?**]. The ovals are the node's k-buckets.

2.3 Kademlia

Kademlia [**?**] 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.

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.

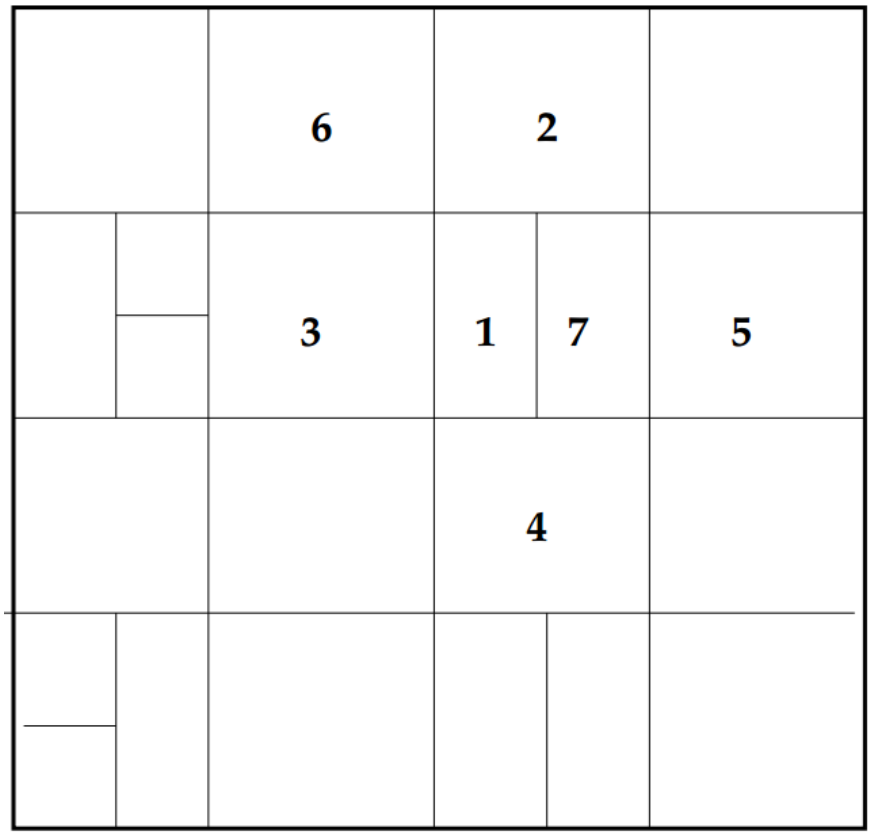


Figure 2.5: An example CAN network from [**?**].

2.4 CAN

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

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 [**?**]. 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 [**?**] and Tapestry [**?**] are extremely similar use a prefix-based routing mechnism introduced by Plaxton et al. [**?**]. 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 [**?**], 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.

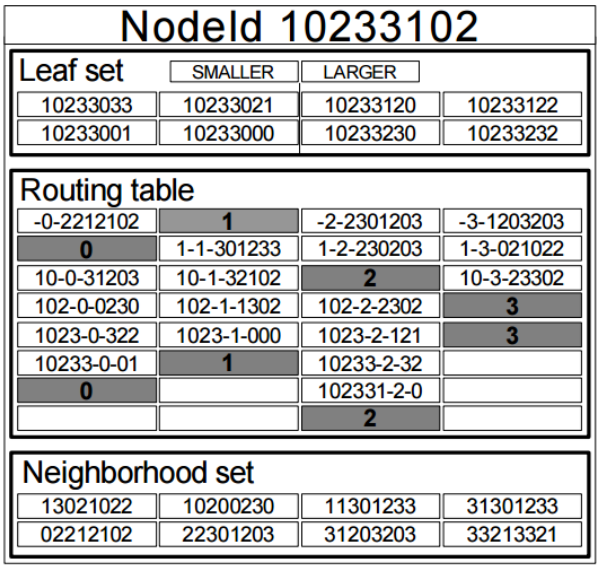


Figure 2.6: An example peerlist for a node in Pastry [**?**].

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 [**?**] is a $1d$ ring-based DHT similar to Chord [**?**], but is constructed using the properties of small world networks [**?**]. 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 [**?**]. 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 [**?**] 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 [**?**] is an example of such a system, as is Amazon's Dynamo [**?**]. 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 [?]	$O(\log n)$, 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 [?]	$O(\log n)$, maximum $m \cdot k$	$(\lceil \log n \rceil) + c$	$O(\log(n))$	This is without considering optimization
CAN [?]	$\Omega(2d)$	$O(n^{\frac{1}{d}})$, average $\frac{d}{4} \cdot n^{\frac{1}{d}}$	Affects $O(d)$ nodes	d is the number of dimensions
Plaxton-based DHTs, Pastry [?], Tapestry [?]	$O(\log_\beta n)$	$O(\lceil \log_{2^\beta} \rceil)$	$O(\log_\beta n)$	NodeIDs are base β numbers
Symphony [?]	$2k + 2$	average $O(\frac{1}{k}\log^2 n)$	$O(\log^2 n)$ messages, constant < 1	$k \geq 1$, fingers are chosen at random
ZHT [?]	$O(n)$	$O(1)$	$O(n)$	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

Distributed computing is a current trend and will continue to be the approach for intensive applications. We see this in the development of cloud computing [**?**], volunteer computing frameworks like BOINC [**?**] and Folding@Home [**?**], and MapReduce [**?**]. Google's MapReduce in particular has rapidly become an integral part in the world of data processing. A user can use MapReduce to 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.

Popular platforms for MapReduce, such as Hadoop [**?**] [**?**], are explicitly designed to be used in large datacenters [**?**] and the majority of research has been focused there. However, as we have previously mentioned, there are notable issues with a centralized design.

First and foremost is the issue of fault-tolerance. Centralized designs have a single point of failure [**?**]. So long as all computing resources are located in one geographical area or rely on a particular node, a power outage or catastrophic event could interrupt computations or otherwise disrupt the platform [**?**].

A centralized design assumes that the network is relatively unchanging and may not have mechanisms to handle node failure during execution or, conversely, cannot speed up the execution of a job by adding additional workers on the fly. Many environments also anticipate a certain degree in homogeneity in the system. Finally deploying these systems and developing programs for them has an extremely steep learning curve.

There is no reason that these assumptions need to be the case for MapReduce, or for many distributed computing frameworks in general. Moving away from the data center context opens up

more possibilities for distributed computing, such as P2P clouds [**?**]. However, without a centralized framework, the network needs some kind of protocol to organize the various components in the network. As part of our research, we developed a highly robust and distributed MapReduce framework based on Chord, called ChordReduce [**?**].

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 [**?**] 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. We established 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 [**?**] 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 an extensive and expanded background on Chord and MapReduce.

3.1.1 Chord

We introduced Chord in Chapter 2, but we present it here again in greater depth. 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.

As we have mentioned discussed in Chapter 2, these keys can be 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,

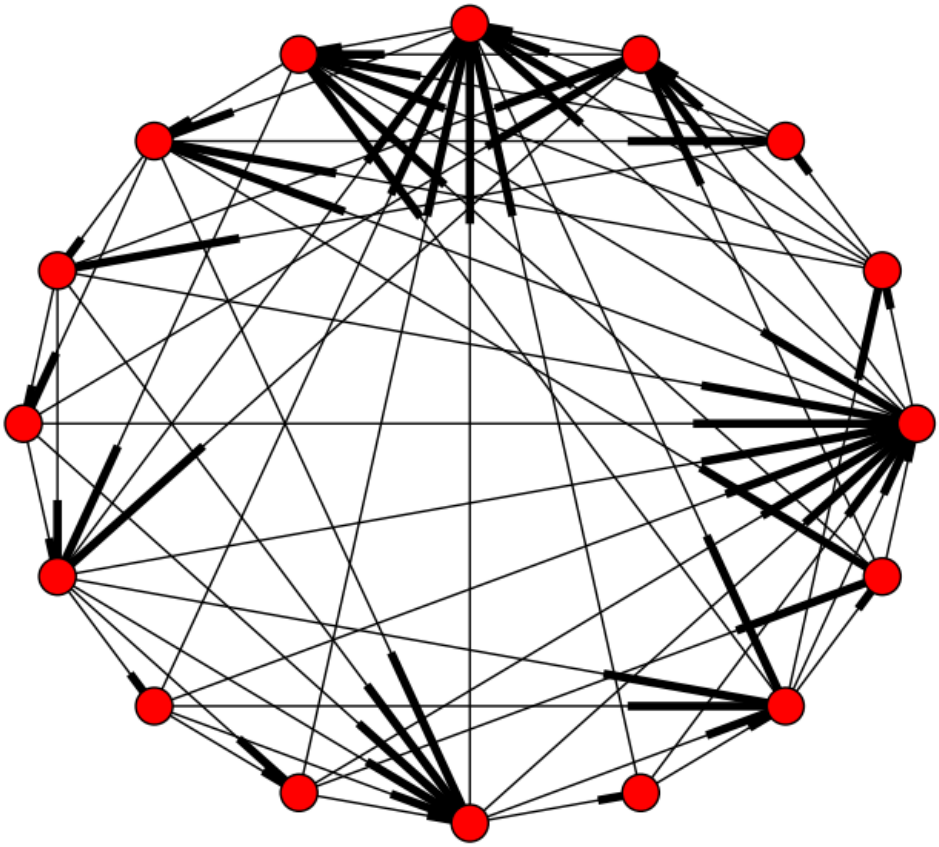


Figure 3.1: A Chord ring with 16 nodes. The bold lines are incoming edges. Each node has a connection to its successor, as well as 4 fingers, some of which are duplicates.

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

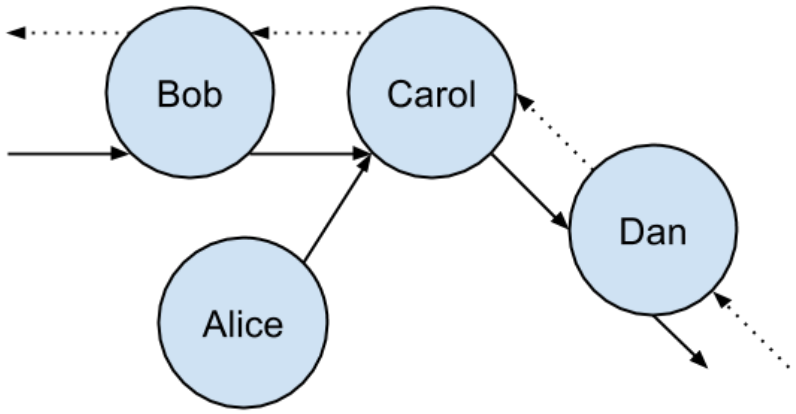


Figure 3.2: Alice has incorrectly determined that Carol is her appropriate successor. When Alice stabilizes, Carol will let her know about Bob.

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 distributed, it is possible to have duplicate entries in the *finger table*. An example Chord network with fingers is shown in in Fig. 3.1.

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

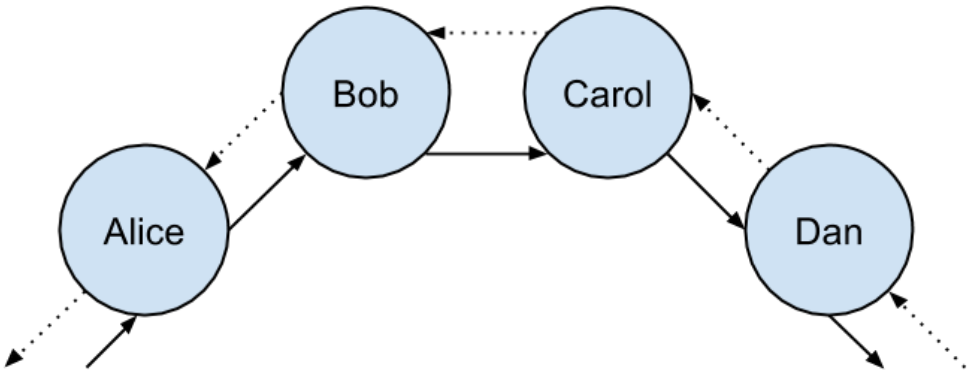


Figure 3.3: After completing stabilize, Alice makes Bob her successor and notifies him. Bob then made Alice as his predecessor.

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 ID 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. 3.2). 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 to Alice 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, or multiple successors upstream. Section 3.3.1 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 [**?**]. In CFS, rather than storing an entire file at a single node, the file is split up into multiple chunks around 10 kilobytes 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 [**?**], 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 Consistency Maintenance [**?**], has nodes keep track of file requests they initiate or forward. If nodes find they are frequently forwarding a request for a particular file, they store that file locally until it is no longer requested frequently.

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 retrieval of any 10 of those 15 chunks is enough to recreate the file. Implementing erasure codes would presumably make a DHT more fault tolerant, but that is an exercise left for future work.

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

another file owner, as the purpose is to keep related files together.

3.1.3 MapReduce

At its core, MapReduce [**?**] is a system for division of labor, providing a layer of separation between the programmer and the more complicated parts of concurrent processing. The programmer sends a large task to a master node, who then divides that task among slave nodes (which may further divide the task). This task has two distinct parts: Map and Reduce. Map performs some operation on a set of data and then produces a result for each Map operation. The resulting data can then be reduced, combining these sets of results into a single set, which is further combined with other sets. This process continues until one set of data remains. A core concept here is the tasks are distributed to the nodes that already contain the relevant data, rather than the data and task being distributed together among arbitrary nodes.

The archetypal example of using MapReduce is WordCount – the task of counting the occurrence of each word in a collection of documents. These documents have been split up into blocks and stored on the network over the distributed file system. The master node locates the worker nodes with blocks and sends the Map and Reduce tasks associated with WordCount. Each worker then goes through their blocks and creates a small word frequency list. These lists are then used by other workers, who combine them into larger and larger lists, until the master node is left with a word frequency list of all the words in the documents.

The most popular platform for MapReduce is Hadoop [**?**]. Hadoop is an open-source Java implementation developed by Apache and Yahoo! [**?**]. Hadoop has two components, the Hadoop Distributed File System (HDFS) [**?**] and the Hadoop MapReduce Framework [**?**]. Under HDFS, nodes are arranged in a hierarchical tree, with a master node, called the NameNode, at the top. The NameNode's job is to organize and distribute information to the slave nodes, called DataNodes. This makes the NameNode a single point of failure [**?**] in the network, as well as a potential bottleneck for the system [**?**].

To do work on Hadoop, the user stores their data on the network. This is handled by the NameNode, which equally apportions the data among the DataNodes. When a user wants to run some analysis on the data or some subset the data, then that function is sent by the NameNode to each of the DataNodes that is responsible for the indicated data. After the DataNode finishes

processing, the result is handled by other nodes called Reducers which collect and reduce the results of multiple DataNodes.

3.2 Related Work

We have identified two papers that focus on combining P2P concepts with MapReduce. Both papers are similar to our research, but differ in crucial ways, as described below.

3.2.1 P2P-MapReduce

Marozzo et al. [**?**] investigated the issue of fault tolerance in centralized MapReduce architectures such as Hadoop. They focused on creating a new P2P based MapReduce architecture built on JXTA [**?**] called P2P-MapReduce. P2P-MapReduce is designed to be more robust at handling node and job failures during execution.

Rather than use a single master node, P2P-MapReduce employs multiple master nodes, each responsible for some job. If one of those master nodes fails, another will be ready as a backup to take its place and manage the slave nodes assigned to that job. This avoids the single point of failure that Hadoop is vulnerable to. Failures of the slave nodes are handled by the master node responsible for it.

Experimental results were gathered via simulation and compared P2P-MapReduce to a centralized framework. Their results showed that while P2P-MapReduce generated an order of magnitude more messages than a centralized approach, the difference rapidly began to shrink at higher rates of churn. When looking at actual amounts of data being passed around the network, the bandwidth required by the centralized approach greatly increased as a function of churn, while the distributed approach again remained relatively static in terms of increased bandwidth usage.

They concluded that P2P-MapReduce would, in general, use more network resources than a centralized approach. However, this was an acceptable cost as the P2P-MapReduce would lose less time from node and job failures [**?**].

While P2P-MapReduce is decentralized, it still relies on a very definite master/slave hierarchy for organization, computations, and scaling. During simulation, 1% of the entire network was assigned as master nodes. This means for a simulation of 40000 nodes, 400 were required to

organize and coordinate jobs, rendering them unable to do any processing. In addition, a loosely-consistent Distributed Hash Table (DHT) such as JXTA can be much slower and fails to maintain the same level of guarantees as an actual DHT, such as Chord [**?**].

3.2.2 MapReduce using Symphony

Lee et al.'s work [**?**] draws attention to the fact that a P2P network can be much more than a way to distribute files and demonstrates how to accomplish different tasks using Map and Reduce functions over a P2P network. Rather than using Chord, Lee et al. used Symphony [**?**], another DHT protocol with a ring topology. To run a MapReduce job over the Symphony ring, a node is selected by the user to effectively act as the master. This ad-hoc master then performs a bounded broadcast over a subsection the ring. Each node repeats this broadcast over a subsection of that subsection, resulting in a tree with the first node at the top. Map tasks are disseminated evenly throughout the tree and their results are reduced on the way back up to the ad-hoc master node. This allows the ring to disseminate Map and Reduce tasks without the need for a coordinator responsible for distributing these tasks and keeping track of them, unlike Hadoop.

Their experimental results showed that the latency experienced by a centralized configuration is similar to the latency experienced in a completely distributed framework. However, there are no mechanisms in place to handle churn in the network. If a node joins during a MapReduce job, it will be unable to contribute any of its resources to the problem. If a node in the bounded broadcast tree fails, or worse the ad-hoc master node fails, the data that node is responsible for is lost.

3.3 ChordReduce

Marozzo et al. [**?**] shows that adding additional fault-tolerance features to a MapReduce architecture is worth the added cost of maintenance, as the time lost due to node failures is greatly reduced. However, Marozzo et al. do not explore the benefits of leveraging the properties of a P2P protocol to reduce the complexity of the architecture and completely distribute the responsibility of the task across the network. As a result, P2P-MapReduce still relies on a ratio of masters to slaves to coordinate and organize the network, meaning a percentage of the network is unable to contribute processing power to the actual solving of a problem.

Lee et al. [?] explores the benefits of building a MapReduce module to run on top of Symphony [?], a P2P protocol. Unlike Hadoop, this allows the MapReduce tasks to be executed without the need of a central source of coordination by distributing tasks over a bounded broadcast tree created at runtime. The Symphony based MapReduce architecture would be greatly improved by the addition of components to handle the failure of nodes during execution. As it stands now, if a node crashes the job will fail due to the loss of data.

While both of these papers have promising results and confirm the capability of our own framework, both solely look at P2P networks for the purpose of routing data and organizing the network. Neither examines using a P2P network as a means of efficiently distributing responsibility throughout the network and using existing features to add robustness to nodes working on Map and Reduce tasks.

ChordReduce uses Chord to act as a completely distributed topology for MapReduce, negating the need to assign any explicit roles to nodes or have a scheduler or coordinator. ChordReduce does not need to assign specific nodes the task of backing up work; nodes backup their tasks using the same process that would be used for any other data being sent around the ring. Finally, results work their way back to a specified hash address, rather than a specific hash node, eliminating any single point of failure in the network. These features help prevent a bottleneck from occurring. The result is a simple, distributed, and highly robust architecture for MapReduce.

3.3.1 Handling Node Failures in Chord

Due to the potentially volatile nature of a peer-to-peer network, Chord has to be able to handle (or at the very least, tolerate) an arbitrary amount of churn. Section 3.1.1 described how Chord gradually guides nodes into their correct locations after they join the network. The same is true for when a node leaves the network; the stabilize procedure will guide nodes to their correct successors and predecessors. However, we can exert more control over how to handle nodes leaving the network.

When a node n changes his successor, n asks if the successor is holding any data n should be responsible for. The successor looks at all the data n is responsible for and sends it to n. The successor does not have to delete this data. In fact, keeping this data as a backup is beneficial to the network as a whole, as n could decide to leave the network at any point.

Chord specifies two ways a node can leave the ring. A node can either suddenly drop out of

existence, or a node can tell the network he is about to leave, letting his successor and predecessor immediately perform the needed changes.

When a node politely quits, he informs both his successor and predecessor and gives them all the information they need to fill the resulting gap. He also sends all of the data he is responsible for to his successor, who becomes responsible for that data when the node leaves. Fingers that pointed to that node would be corrected during the finger maintenance period. This allows for the network to adjust to churn with a minimum of overhead.

It is unlikely that every time a node leaves the network, it will do so politely. If a node suddenly quits, the data it had stored is lost. To prevent data from becoming irretrievable, a node periodically sends backups to its successor. In order to prevent a cascade of backups of backups, the node only passes along what it considers itself responsible for. What a node is responsible for changes as nodes enter and leave the network. If a node's successor leaves, the node sends a backup to his new successor.

Our prototype framework does not implement a polite disconnect; when a node quits, it does so quickly and abruptly. This design ensures that the system would be able to handle churn under the worst of cases. Polite quit could be implemented quite easily.

3.3.2 Implementation

ChordReduce is a fully functional Chord implementation in Python. Our installation was designed to be as simple as possible. It consists of downloading our code [**?**] and running `chord.py`. A user can specify a port and IP of a node in the ring they wish to join. The node will automatically integrate into the ring with this minimal information. The ring as implemented is stable and well organized. We created various services to run on top the network, such as a file system and distributed web server. Our file system is capable of storing whole files or splitting the file up among multiple nodes the ring. Our MapReduce module is a service that runs on top of our Chord implementation, similar to the file system (Fig. 3.4). We avoided any complicated additions to the Chord architecture; instead we used the protocol's properties to create the features we desired in our MapReduce framework.

In our implementation of MapReduce, each node takes on responsibilities of both a worker and master, much in the same way that a node in a P2P file-sharing service will act as both a client

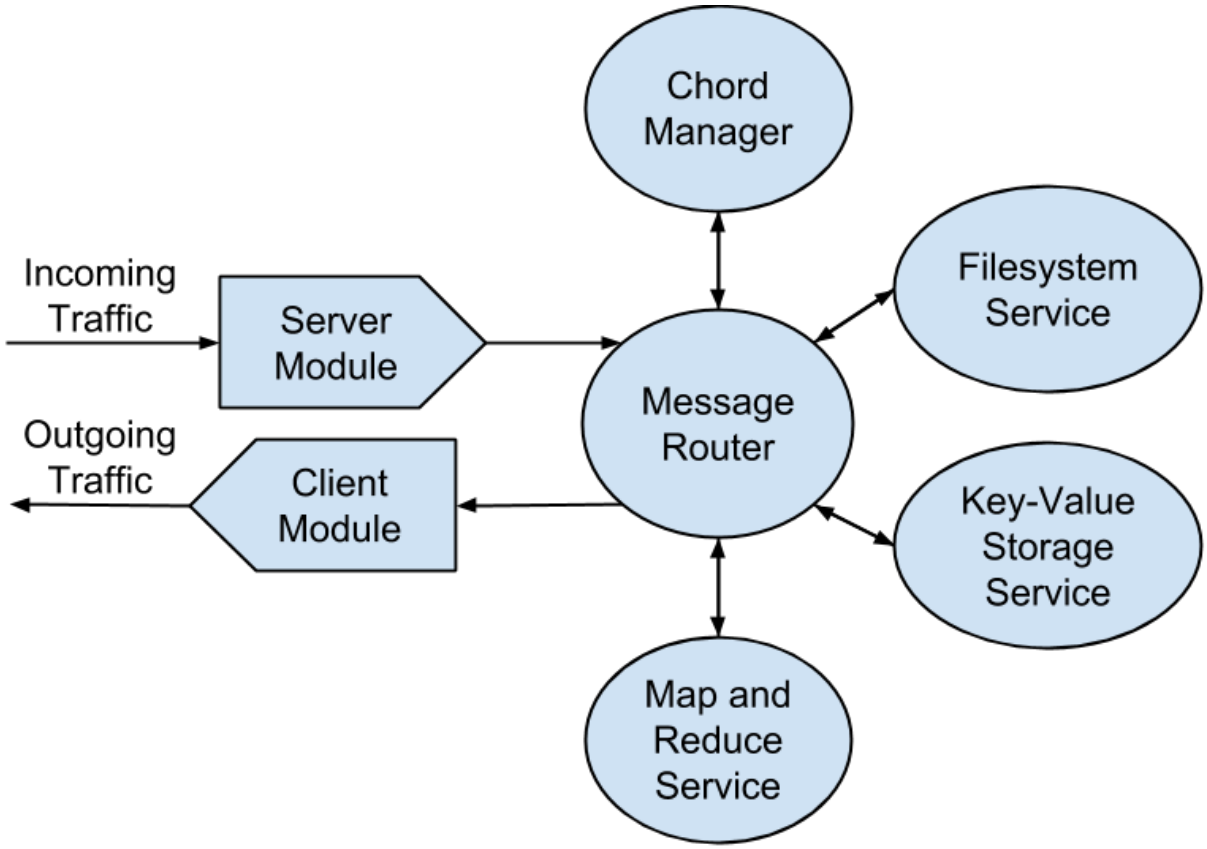


Figure 3.4: The basic architecture of a node in ChordReduce. MapReduce runs as a service on top of each node.

and a server. Jobs still must start from a single location. To start a job, the user contacts a node at a specified hash address and provides it with the tasks and data. This address can be chosen arbitrarily or be a known node in the ring. The node at this hash address is designated as the stager.

The job of this stager is to take the work and divide it into *data atoms*, which are the smallest individual units that work can be done on. This might be a line of text in a document, the result of a summation for a particular intermediate value, or a subset of items to be sorted. The specifics of how to divide the work are defined by the user in a *stage* function. The data atoms are then each given a random hash and sent to the node responsible for that hash address, guaranteeing they are evenly distributed throughout the network. The data atoms also contain the Map function and Reduce function as defined by the user. A job ID is also included, so that data atoms from different jobs can be differentiated. Once the data atoms are sent out, the stager's job is done and it behaves like any other node in the network. The staging period is the only time ChordReduce is vulnerable to churn, and only if the stager leaves the ring in the middle of sending out data atoms. The user would get some results back, but only for the data the stager managed to send out.

Nodes that receive data atoms apply the Map function to the data to create result data atoms, which are then sent back to the stager's hash address (or some other user defined address). This will take $\log_2 n$ hops traveling over Chord's fingers. At each hop, the node waits a predetermined minimal amount of time to accumulate additional results (In our experiments, this was 100 milliseconds).

Nodes that receive at least two results merge them using the Reduce function. The results are continually merged until only one remains at the hash address of the stager.

Once the reductions are finished, the user retrieves his results from the node at the stager's address. This may not be the stager himself, as the stager may no longer be in the network. The stager does not need to collect the results himself, since the work is sent to the stager's hash address, rather than the stager itself. Thus, the stager could quit the network after staging, and both the user and the network would be unaffected by the change.

Similar precautions are taken for nodes working on Map and Reduce tasks. Those tasks are backed up by a node's successor, who will run the task if the node leaves before finishing its work (e.g. the successor loses his predecessor). The task is given a timeout by the node. If the backup

node detects that the responsible node has failed, he starts the work and backs up again to *his* successor. Otherwise, the data is tossed away once the timeout expires. This is done to prevent a job being submitted twice.

An advantage of our system is the ease of development and deployment. The developer does not need to worry about distributing work evenly, nor does he have to worry about any node in the network going down. The stager does not need to keep track of the status of the network. The underlying Chord ring handles that automatically. If the user finds they need additional processing power during runtime, they can boot up additional nodes, which would automatically be assigned work based on their hash value. If a node goes down while performing an operation, his successor takes over for him. This makes the system extremely robust during runtime.

All a developer needs to do is write three functions: the staging function, Map, and Reduce. These define how to split up the work into manageable portions, the work to be performed on each portion to obtain results, and how to combine these results into a single result, respectively.

3.4 Experiments

In order for ChordReduce to be a viable framework, we had to show these three properties:

1. ChordReduce provides significant speedup during a distributed job.
2. ChordReduce scales.
3. ChordReduce handles churn during execution.

Speedup can be demonstrated by showing that a distributed job is generally performed more quickly than the same job handled by a single worker. More formally we need to establish that $\exists n$ such that $T_n < T_1$, where T_n is the amount of time it takes for n nodes to finish the job.

To establish scalability, we need to show that the cost of distributing the work grows logarithmically with the number of workers. In addition, we need to demonstrate that the larger the job is, the number of nodes we can have working on the problem without the overhead incurring diminishing returns increases. This can be stated as

$$T_n = \frac{T_1}{n} + k \cdot \log_2(n)$$

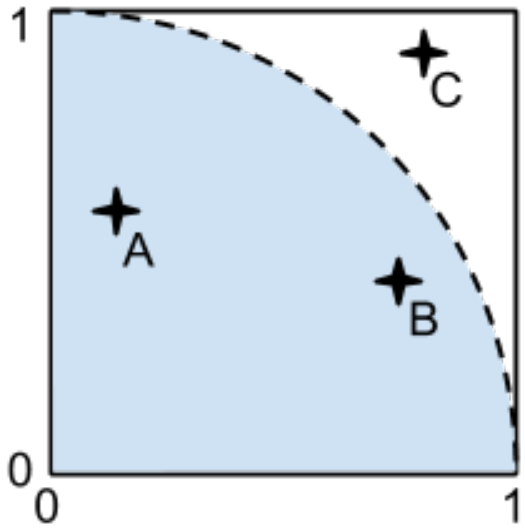


Figure 3.5: The "dartboard." The computer throws a dart by choosing a random x and y between 0 and 1. If $x^2 + y^2 < 1^2$, the dart landed inside the circle. A and B are darts that landed inside the circle, while C did not.

where $\frac{T_1}{n}$ is the amount of time the job would take when distributed in an ideal universe and $k \cdot \log_2(n)$ is network induced overhead, k being an unknown constant dependent on network latency and available processing power.

Finally, to demonstrate robustness, we need to show that ChordReduce can handle arbitrary node failure in the ring and that such failures minimally impair the overall speed of the network.

3.4.1 Setup

To stress test our framework, we ran a Monte-Carlo approximation of π. This process is largely analogous to having a square with the top-right quarter of a circle going through it (Fig. 3.5), and then throwing darts at random locations. Counting the ratio of darts that land inside the circle to the total number of throws gives us an approximation of $\frac{\pi}{4}$. The more darts thrown, i.e. the more samples that are taken, the more accurate the approximation[1].

We chose this experiment for a number of reasons. The job is extremely easy to distribute. This also made it very easy to test scalability. By doubling the amount of samples, we can double the amount of work each node gets. We could also test the effectiveness of distributing the job among

[1]This is not intended to be a particularly good approximation of π. Each additional digit of accuracy requires increasing the number of samples taken by an order of magnitude.

different numbers of workers.

Each Map job is defined by the number of throws the node must make and yields a result containing the total number of throws and the number of throws that landed inside the circular section. Reducing these results is then a matter of adding the respective fields together.

We ran our experiments using Amazons's Elastic Compute Cloud (EC2) service. Amazon EC2 allows users to purchase an arbitrary amount of virtual machines by the hour. Each node was an individual EC2 small instance [**?**] with a preconfigured Ubuntu 12.04 image. These instances were capable enough to provide constant computation, but still weak enough that they would be overwhelmed by traffic on occasions, creating a constant churn effect in the ring.

Once started, nodes retrieve the latest version of the code and run it as a service, automatically joining the network. We can choose any arbitrary node as the stager and tell it to run the MapReduce process. We found that the network was robust enough that we could take a node we wanted to be the stager out of the network, modify its MapReduce test code, have it rejoin the network, and then run the new code without any problems. Since only the stager has to know how to create the Map tasks, the other nodes do not have to be updated and execute the new tasks they are given.

We ran our experiments on groups of 1, 10, 20, 30, and 40 workers, which generated a 10^8 sample set and a 10^9 sample set. Additionally, we gathered data on a 10^7 sample set using 1, 5, 10, 20, 30 workers. To test churn, we ran an experiment where each node had an equal chance of leaving and joining the network and varied the level of churn over multiple runs.

We also utilized a subroutine we wrote called *plot*, which sends a message sequentially around the ring to establish how many members there are. If *plot* failed to return in under a second, the ring was experiencing structural instability.

3.4.2 Results

Fig. 3.6 and Fig. 3.7 summarize the experimental results of job duration and speedup. Our default series was the 10^8 samples series. On average, it took a single node 431 seconds, or approximately 7 minutes, to generate 10^8 samples. Generating the same number of samples using ChordReduce over 10, 20, 30, or 40 nodes was always quicker. The samples were generated fastest when there were 20 workers, with a speedup factor of 4.96, while increasing the number of workers to 30 yielded

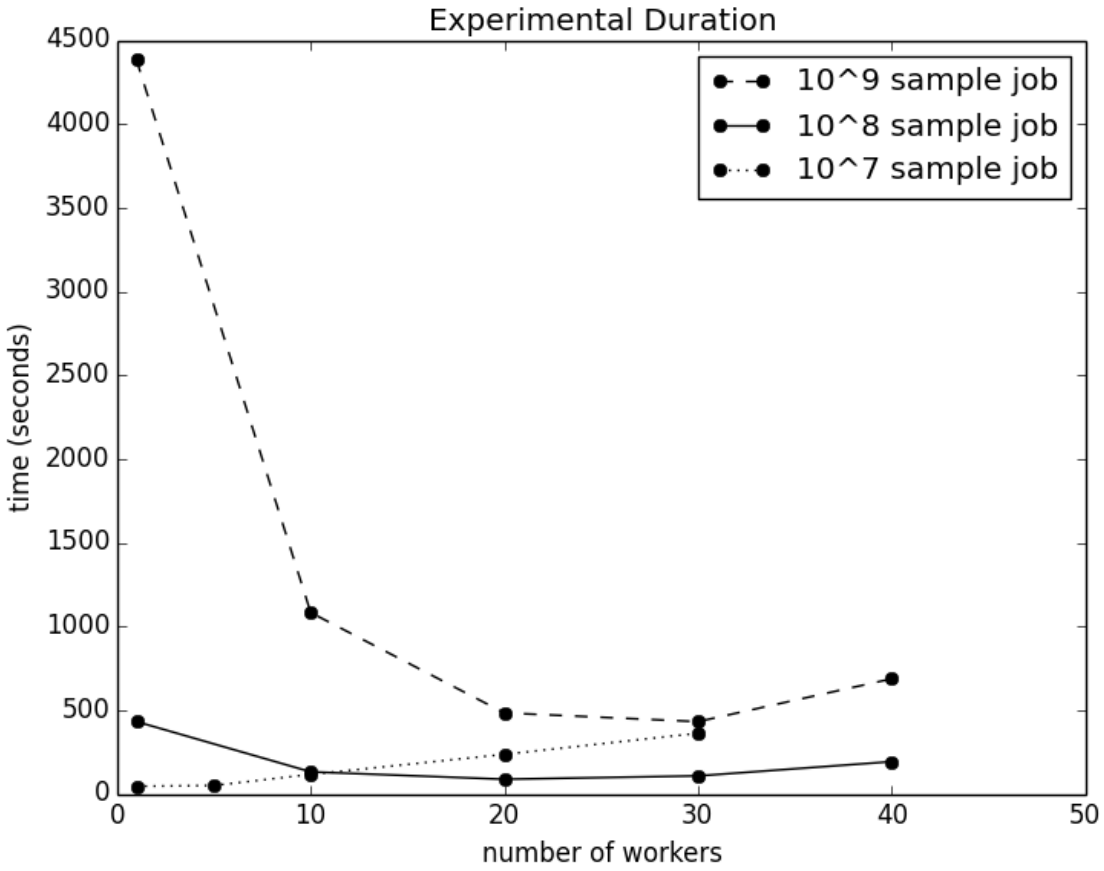


Figure 3.6: For a sufficiently large job, it was almost always preferable to distribute it. When the job is too small, such as with the 10^7 data set, our runtime is dominated by the overhead. Our results are what we would expect when overhead grows logarithmically to the number of workers.

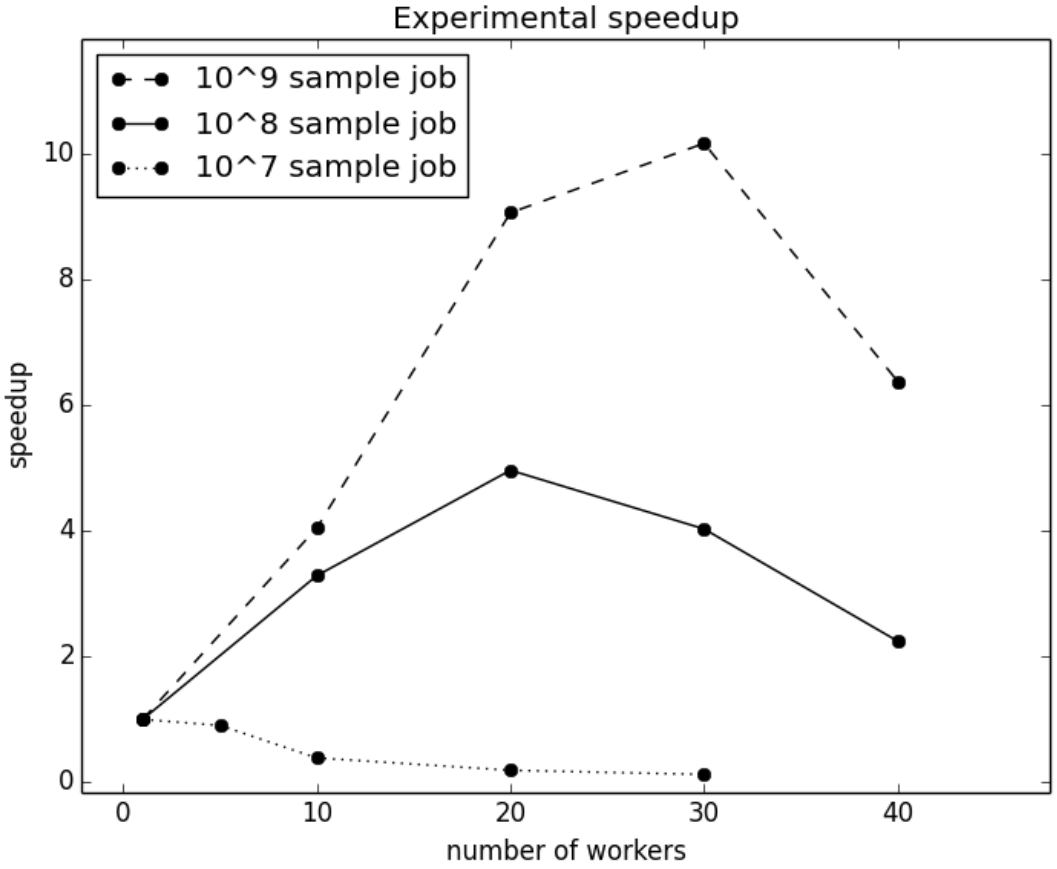


Figure 3.7: The larger the size of the job, the greater the gains of distributing with ChordReduce. In addition, the larger the job, the more workers can be added before we start seeing diminishing returns. This demonstrates that ChordReduce is scalable.

Churn rate per second	Average runtime (s)	Speedup vs 0% churn
0.8%	191.25	2.15
0.4%	329.20	1.25
0.025%	431.86	0.95
0.00775%	445.47	0.92
0.00250%	331.80	1.24
0%	441.57	1.00

Table 3.1

a speedup of only 4.03. At 30 nodes, the gains of distributing the work were present, but the cost of overhead ($k \cdot \log_2(n)$) had more of an impact. This effect is more pronounced at 40 workers, with a speedup of 2.25.

Since our data showed that approximating π on one node with 10^8 samples took approximately 7 minutes, collecting 10^9 samples on a single node would take 70 minutes at minimum. Fig. 3.7 shows that the 10^9 set gained greater benefit from being distributed than the 10^8 set, with the speedup factor at 20 workers being 9.07 compared to 4.03. In addition, the gains of distributing work further increased at 30 workers and only began to decay at 40 workers, compared with the 10^8 data set, which began its drop off at 30 workers. This behavior demonstrates that the larger the job being distributed, the greater the gains of distributing the work using ChordReduce.

The 10^7 sample set confirms that the network overhead is logarithmic. At that size, it is not effective to run the job concurrently and we start seeing overheard acting as the dominant factor in runtime. This matches the behavior predicted by our equation, $T_n = \frac{T_1}{n} + k \cdot \log_2(n)$. For a small T_1, $\frac{T_1}{n}$ approaches 0 as n gets larger, while $k \cdot \log_2(n)$, our overhead, dominates the sample. The samples from our data set fit this behavior, establishing that our overhead increases logarithmically with the number of workers.

Since we have now established that $T_n = \frac{T_1}{n} + k \cdot \log_2(n)$, we can estimate how long a job that takes an arbitrary amount of time to run on a single node would take using ChordReduce. Our data points indicated that the mean value of k for this problem was 36.5. Fig. 3.8 shows that for jobs that would take more than 10^4 seconds for single worker to complete, we can expect there would still be benefit to adding an additional worker, even when there are already 5000 workers already in the ring. Fig. 3.9 further emphasizes this. Note that as the jobs become larger, the expected speedup from ChordReduce approaches linear behavior.

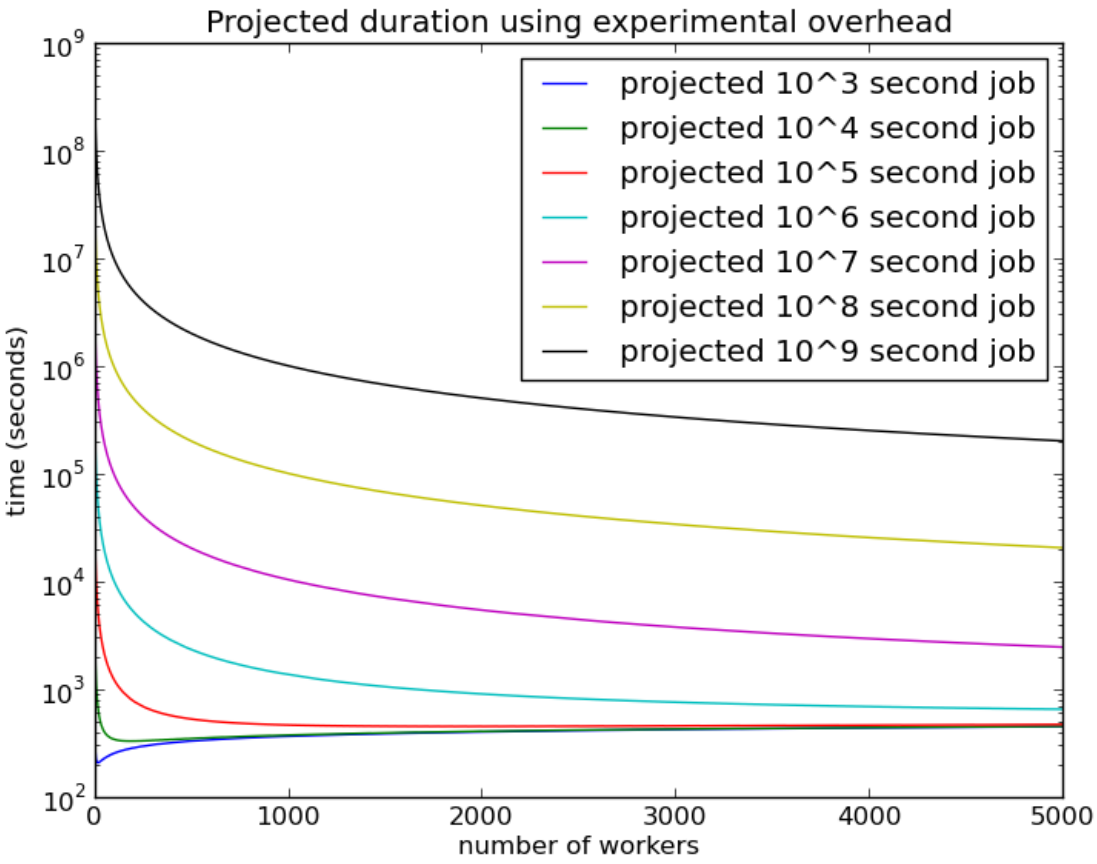


Figure 3.8: The projected runtime using ChordReduce for differently sized jobs. Each curve projects the expected behavior for job that takes a single worker the specified amount of time.

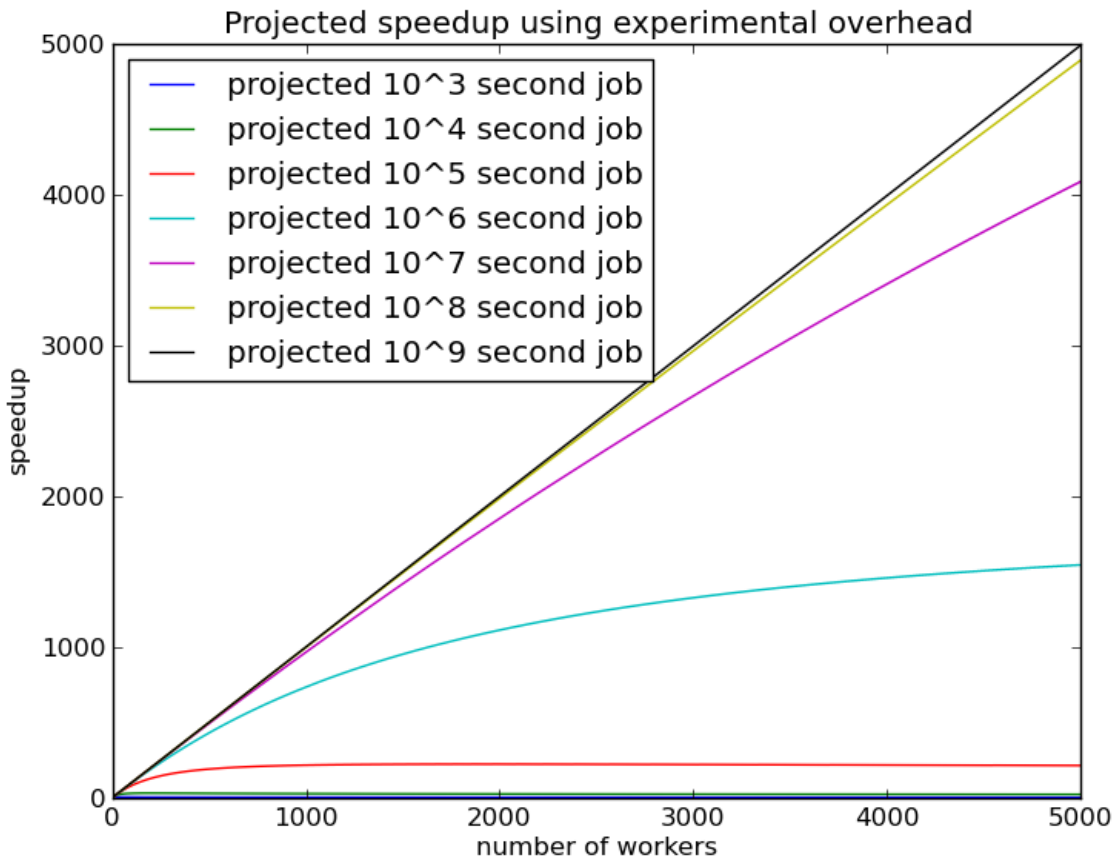


Figure 3.9: The projected speedup for different sized jobs.

Table 3.1 shows the experimental results for different rates of churn. These results show the system is relatively insensitive to churn. We started with 40 nodes in the ring and generated 10^8 samples while experiencing different rates of churn, as specified in Table 3.1. At the 0.8% rate of churn, there is a 0.8% chance each second that any given node will leave the network followed by another node joining the network at a different location. The joining rate and leaving rate being identical is not an unusual assumption to make [**?**] [**?**].

Our testing rates for churn are an order of magnitude higher than the rates used in the P2P-MapReduce simulation [**?**]. In their paper, the highest rate of churn was only 0.4% per minute. Because we were dealing with fewer nodes, we chose larger rates to demonstrate that ChordReduce could effectively handle a high level of churn.

Our experiments show that for a given problem, ChordReduce can effectively distribute the problem, yielding a substantial speedup. Furthermore, our results showed that the larger the problem is, the more workers could be added before diminishing returns were incurred. During runtime, we experienced multiple instances where *plot* would fail to run and the stager would report socket errors, indicating that it had lost connection with a node in the ring. Despite this turbulence, every node managed to reestablish connection with each other and report back all the data. This further demonstrated that we were able to handle the churn in the network.

3.5 Remarks

During our experiments testing the capabilities of ChordReduce, we experienced a significant and completely unexpected anomaly while testing churn. One of the things previous research [**?**] [**?**] in the same area we felt we needed to explore better was how a completely decentralized computation could handle churn. Now, despite our initial prototype having numerous bugs and only able to handle small networks, we were fairly certain of it's ability to handle churn.

Marozzo et al. [**?**] tested their network using churn rates of 0.025%, 0.05%, 0.1%, 0.2%, and 0.4% per minute. The churn rate of $cr << 1$ per minute means that each minute on average, $cr \cdot n$ nodes leave the network and $cr \cdot n$ new nodes join the network.[2] This could effectively be thought of as each node flipping a weighted coin every minute. When the coin lands on tails, the node leaves.

[2]It is standard practice to assume the joining rate and leaving rate are equal.

A similar process happens for nodes wanting to join the network.

We wanted the robustness of our system to be beyond reproach, so we tested at rates from 0.0025% to 0.8% ***per second***, 120 times the fastest rate used to test P2P-MapReduce. This is an absurdly fast and unrealistic speed, the only purpose of which was to cement the fault tolerance of the system. Since we were testing ChordReduce on Amazon's EC2 and paying per instance per hour, we limited the number of nodes. Rather than having a pool of nodes waiting to join the network, we conserved our funds by having leaving nodes immediately rejoin the network under a new IP/port combo. The meant our churn operation was essentially a simultaneous leave and join.

What we found was that jobs on ChordReduce finished twice as fast under the unrealistic levels churn (0.8% per second) than no churn (Table 3.1). This completely mystified us. Churn is a disruptive force; how can it be aiding the network?

Hypothesis

We hypothesize this was due to the number of data pieces (larger) vs the number of workers (smaller). There were more workers than there were pieces of data, so some workers ended up with more data than others in the initial distribution. This means that there was some imbalance in the way data was distributed among nodes. This was *further* exacerbated by small number of workers distributed over a large hash space, leading some nodes to have larger swaths of responsibility than others.

Given this setup, without any churn, the operation would be: Workers get triggered, they start working, and the ones with little work finish their work quickly, and the network waits for the node with higher loads of work.

Its important to note here that the work in ChordReduce was performed atomically, a piece at a time. When a node was working on a piece, it informed it's successor, then informed them when it finished. These pieces of work were also small, possibly too small.

As mentioned previously, under our induced experimental churn, we had the nodes randomly fail and immediately join under a new IP/port combination, which yields a new hash. The failure rates were orders of magnitude higher than what would be expected in a "real" (nonexperimental) environment. The following possibilities could occur:

- A node without any active jobs leaves. It dies and and comes back with a new port chosen. This new ID has a higher chance of landing in a larger region of responsibility (since new joining nodes have a greater chance of hashing to a larger region than a smaller). In other words, it has a (relatively) higher chance of moving into an space where it becomes acquires responsibility for enqueued jobs. The outcomes of this are:
  - The node rejoins in a region and does not acquire any new jobs. This has no impact on the network (Case I).
  - The node rejoins in a region that has jobs waiting to be done. It acquires some of these jobs. This speeds up performance (Case II).
- A node with active jobs dies. It rejoins in a new space. The jobs were small, so not too much time is lost on the active job, and the enqueued jobs are backed up and the successor knows to complete them. However, the node can rejoin in a more job-heavy region and acquire new jobs. The outcomes of this are:
  - A minor negative impact on runtime and load balancing (since the successor has more jobs to handle) (Case III).
  - A possible counterbalance in load balancing by acquiring new jobs off a busy node (Case IV).

The longer the nodes work on the jobs, the more nodes finish and have no jobs. This means as time increases, so do the occurrences of Case I and II.

This leads us to two hypotheses:

- Deleting nodes motivates other nodes to work harder to avoid deletion (a "beatings will continue until morale improves" situation).
- Our high rate of churn was dynamically load-balancing the network. It appears even the smallest effort of trying to dynamically load balance, such as rebooting random nodes to new locations, has benefits for runtime. Our method is a poor approximation of dynamic load-balancing, and it still shows improvement.

The first hypothesis is mentally pleasing to anyone who has tried to create a distributed system, but lacks rigor. Verification, analysis, and exploitation of this phenomena is the subject of (Chapter 8).

Once we have established that it does exist, we need a better load-balancing strategy than randomly inducing. We want nodes to have a precomputed list of locations in which they can insert nodes to perform load-balancing on an ad-hoc basis during runtime. This precomputed list ties directly into the security research on DHTs we have done [**?**] and is the subject of Chapter 7.

Chapter 4

VHash And DGVH

DHTs all seek to minimize lookup time for their respective topologies. This is done by minimizing the number of overlay hops needed for a lookup operation. This is a good approximation for minimizing the latency of lookups, but does not actually do so, as each hop has a different amount of latency. Furthermore, a network might need to minimize some arbitrary metric, such as energy consumption.

VHash is a multi-dimensional DHT that minimizes routing over some given metric. It uses a fast approximation of a Delaunay Triangulation to compute the Voronoi tessilation of a multi-dimensional space.

Arguably all Distributed Hash Tables (DHTs) are built on the concept of Voronoi tessellations. In all DHTs, a node is responsible for all points in the overlay to which it is the "closest" node. Nodes are assigned a key as their location in some keyspace, based on the hash of certain attributes. Normally, this is just the hash of the IP address (and possibly the port) of the node [**?**] [**?**] [**?**] [**?**], but other metrics such as geographic location can be used as well [**?**].

These DHTs have carefully chosen metric spaces such that these regions are very simple to calculate. For example, Chord [**?**] and similar ring-based DHTs [**?**] utilize a unidirectional, one-dimensional ring as their metric space, such that the region for which a node is responsible is the region between itself and its predecessor.

Using a Voronoi tessellation in a DHT generalizes this design. Nodes are Voronoi generators at a position based on their hashed keys. These nodes are responsible for any key that falls within its generated Voronoi region.

Messages get routed along links to neighboring nodes. This would take $O(n)$ hops in one dimension. In multiple dimensions, our routing algorithm (Algorithm 2) is extremely similar to the one used in Ratnasamy et al.'s Content Addressable Network (CAN) [**?**], which would be $O(n^{\frac{1}{d}})$ hops.

Algorithm 2 Lookup in a Voronoi-based DHT

```
1: Given node n
2: Given m is a message addressed for loc
3: potential_dests ← n ∪ n.short_peers ∪ n.long_peers
4: c ← node in potential_dests with shortest distance to loc
5: if c == n then
6:     return n
7: else
8:     return c.lookup(loc)
9: end if
```

Efficient solutions, such as Fortune's sweepline algorithm [**?**], are not usable in spaces with 2 more dimensions. As far as we can tell, there is no way efficient to generate higher dimension Voronoi tessellations, especially in the distributed Churn-heavy context of a DHT. Our solution is the Distributed Greedy Voronoi Heuristic.

4.1 Distributed Greedy Voronoi Heuristic

A Voronoi tessellation is the partition of a space into cells or regions along a set of objects O, such that all the points in a particular region are closer to one object than any other object. We refer to the region owned by an object as that object's Voronoi region. Objects which are used to create the regions are called Voronoi generators. In network applications that use Voronoi tessellations, nodes in the network act as the Voronoi generators.

The Voronoi tessellation and Delaunay triangulation are dual problems, as an edge between two objects in a Delaunay triangulation exists if and only if those object's Voronoi regions border each other. This means that solving either problem will yield the solution to both. An example Voronoi diagram is shown in Figure 4.1. For additional information, Aurenhammer [**?**] provides a formal and extremely thorough description of Voronoi tessellations, as well as their applications.

The Distributed Greedy Voronoi Heuristic (DGVH) is a fast method for nodes to define their individual Voronoi region (Algorithm 6). This is done by selecting the nearby nodes that would

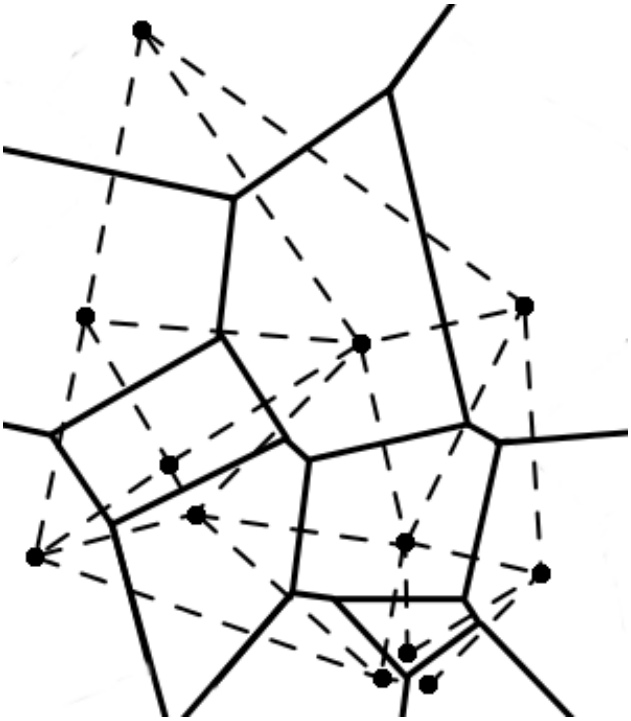


Figure 4.1: An example Voronoi diagram for objects on a 2-dimensional space. The black lines correspond to the borders of the Voronoi region, while the dashed lines correspond to the edges of the Delaunay Triangulation.

correspond to the points connected to it by a Delaunay triangulation. The rationale for this heuristic is that, in the majority of cases, the midpoint between two nodes falls on the common boundary of their Voronoi regions.

During each cycle, nodes exchange their peer lists with a current neighbor and then recalculate their neighbors. A node combines their neighbor's peer list with its own to create a list of candidate neighbors. This combined list is sorted from closest to furthest. A new peer list is then created starting with the closest candidate. The node then examines each of the remaining candidates in the sorted list and calculates the midpoint between the node and the candidate. If any of the nodes in the new peer list are closer to the midpoint than the candidate, the candidate is set aside. Otherwise the candidate is added to the new peer list.

DGVH never actually solves for the actual polytopes that describe a node's Voronoi region. This is unnecessary and prohibitively expensive [**?**]. Rather, once the heuristic has been run, nodes can determine whether a given point would fall in its region.

Nodes do this by calculating the distance of the given point to itself and other nodes it knows

Algorithm 3 Distributed Greedy Voronoi Heuristic

Given node n and its list of *candidates*.
Given the minimum *table_size*
short_peers $\leftarrow$ empty set that will contain n's one-hop peers
long_peers $\leftarrow$ empty set that will contain n's two-hop peers
Sort *candidates* in ascending order by each node's distance to n
Remove the first member of *candidates* and add it to *short_peers*
for all c in *candidates* **do**
    m is the midpoint between n and c
    if Any node in *short_peers* is closer to m than n **then**
        Reject c as a peer
    else
        Remove c from *candidates*
        Add c to *short_peers*
    end if
end for
while $|short\_peers| < table\_size$ **and** $|candidates| > 0$ **do**
    Remove the first entry c from *candidates*
    Add c to *short_peers*
end while
Add *candidates* to the set of *long_peers*
if $|long\_peers| > table\_size^2$ **then**
    long_peers $\leftarrow$ random subset of *long_peers* of size $table\_size^2$
end if

about. The point falls into a particular node's Voronoi region if it is the node to which it has the shortest distance. This process continues recursively until a node determines that itself to be the closest node to the point. Thus, a node defines its Voronoi region by keeping a list of the peers that bound it.

4.1.1 Algorithm Analysis

DVGH is very efficient in terms of both space and time. Suppose a node n is creating its short peer list from k candidates in an overlay network of N nodes. The candidates must be sorted, which takes $O(k \cdot \lg(k))$ operations. Node n must then compute the midpoint between itself and each of the k candidates. Node n then compares distances to the midpoints between itself and all the candidates. This results in a cost of

$$k \cdot \lg(k) + k \text{ midpoints} + k^2 \text{ distances}$$

Since k is bounded by $\Theta(\frac{\log N}{\log \log N})$ [**?**] (the expected maximum degree of a node), we can translate the above to

$$O(\frac{\log^2 N}{\log^2 \log N})$$

In the vast majority of cases, the number of peers is equal to the minimum size of *Short Peers*. This yields $k = (3d + 1)^2 + 3d + 1$ in the expected case, where the lower bound and expected complexities are $\Omega(1)$.

4.2 Experimental Results

We evaluated the effectiveness of VHash and DGVH in creating a set of experiments.[1] The first experiment showed how VHash could use DGVH to create a routing mesh. Our second showed how optimizing for latency yielded better results than optimizing for least hops.

[1] Our results are pulled directly from [**?**] and [**?**].

4.2.1 Convergence

Our first experiment examined how DGVH could be used to create a routing overlay and how well it performed in this task. The simulation demonstrated how DGVH formed a stable overlay from a chaotic starting topology after a number of cycles. We compared our results to those in RayNet [**?**]. The authors of Raynet proposed a random k-connected graph would be a challenging initial configuration for showing a DHT relying on a gossip mechanism could converge to a stable topology.

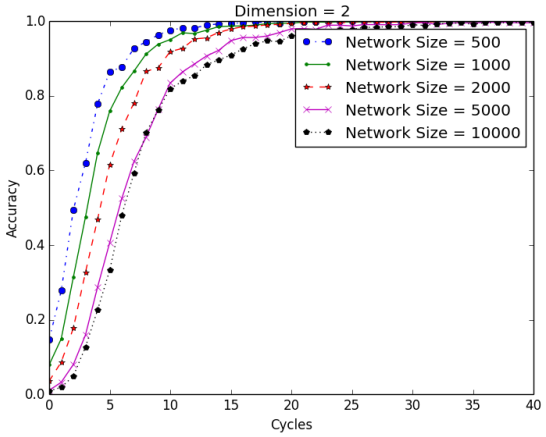
In the initial two cycles of the simulation, each node bootstrapped its short peer list by appending 10 nodes, selected uniformly at random from the entire network. In each cycle, the nodes gossiped , swapping peer list information. They then ran DGVH using the new information. We calculated the hit rate of successful lookups by simulating 2000 lookups from random nodes to random locations, as described in Algorithm 4. A lookup was considered successful if the network was able to determine which Voronoi region contained a randomly selected point.

Our experimental variables for this simulation were the number of nodes in the DGVH generated overlay and the number of dimensions. We tested network sizes of 500, 1000, 2000, 5000, and 10000 nodes each in 2, 3, 4, and 5 dimensions. The hit rate at each cycle is $\frac{hits}{2000}$, where $hits$ are the number of successful lookups.

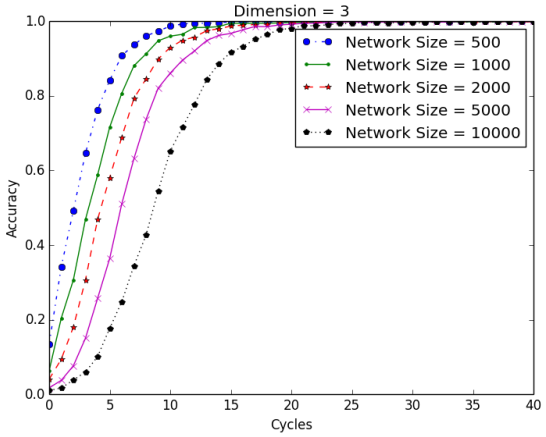
Algorithm 4 Routing Simulation Sample

```
1: start ← random node
2: dest ← random set of coordinates
3: ans ← node closest to dest
4: if ans == start.lookup(dest) then
5:     increment hits
6: end if
```

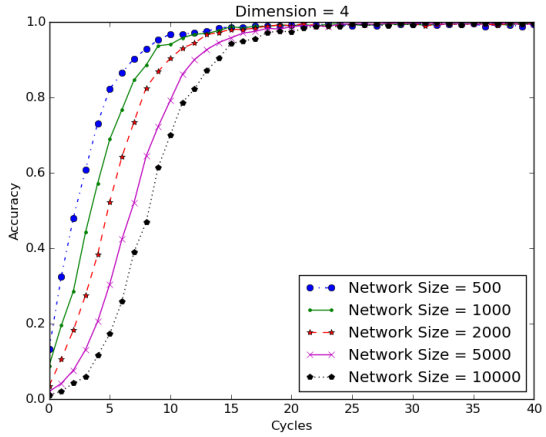
The results of our simulation are shown in Figure 4.2. Our graphs show that a correct overlay was quickly constructed from a random configuration and that our hit rate reached 90% by cycle 20, regardless of the number of dimensions. Lookups consistently approached a hit rate of 100% by cycle 30. In comparison, RayNet's routing converged to a perfect hit rate at around cycle 30 to 35 [**?**]. As the network size and number of dimensions each increase, convergence slows, but not to a significant degree.



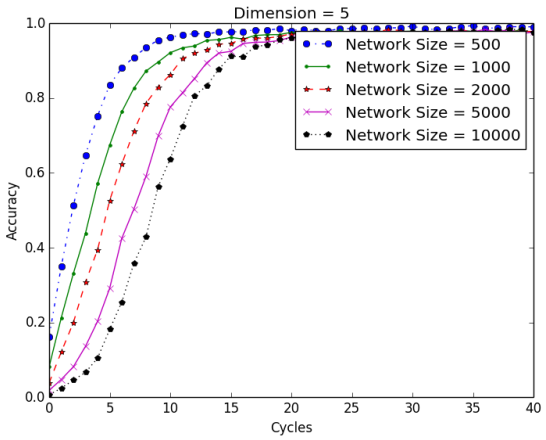
(a) This plot shows the accuracy rate of lookups on a 2-dimensional network as it self-organizes.



(b) This plot shows the accuracy rate of lookups on a 3-dimensional network as it self-organizes.



(c) This plot shows the accuracy rate of lookups on a 4-dimensional network as it self-organizes.



(d) This plot shows the accuracy rate of lookups on a 5-dimensional network as it self-organizes.

Figure 4.2: These figures show that, starting from a randomized network, DGVH forms a stable and consistent network topology. The Y axis shows the success rate of lookups and the X axis show the number of gossips that have occurred. Each point shows the fraction of 2000 lookups that successfully found the correct destination.

4.2.2 Latency Distribution Test

The goal of our second set of experiments was to demonstrate VHash's ability to optimize a selected network metric: latency in this case. In our simulation, we used the number of hops on the underlying network as an approximation of latency. We compared VHash's performance to Chord [**?**]. As we discussed in Chapter 2 Chord is a well established DHT with an $O(\log(n))$ sized routing table and $O(\log(n))$ lookup time measured in overlay hops.

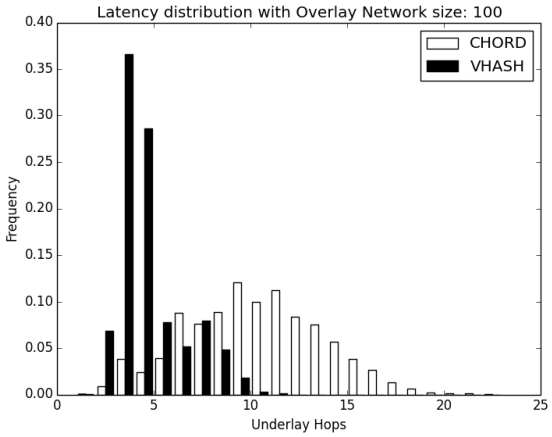
Instead of using the number of hops on the overlay network as our metric, we are concerned with the actual latency lookups experience traveling through the *underlay* network, the network upon which the overlay is built. Overlay hops are used in most DHT evaluations as the primary measure of latency. It is the best approach available when there are no means of evaluating the characteristics of the underlying network. VHash is designed with a capability to exploit the characteristics of the underlying network. With most realistic network sizes and structures, there is substantial room for latency reduction in DHTs.

For this experiment, we constructed a scale free network with 10000 nodes placed at random (which has an approximate diameter of 3 hops) as an underlay network [**?**] [**?**] [**?**]. We chose to use a scale-free network as the underlay, since scale free networks model the Internet's topology [**?**] [**?**]. We then chose a random subset of nodes to be members of the overlay network. Our next step was to measure the distance in underlay hops between 10000 random source-destination pairs in the overlay. VHash generated an embedding of the latency graph utilizing a distributed force directed model, with the latency function defined as the number of underlay hops between it and its peers.

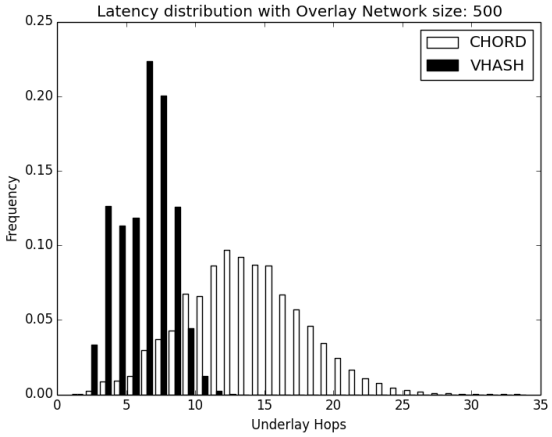
Our simulation created 100, 500, and 1000 node overlays for both VHash and Chord. We used 4 dimensions in VHash and a standard 160 bit identifier for Chord.

Figure 4.3 shows the distribution of path lengths measured in underlay hops in both Chord and VHash. VHash significantly outperformed Chord and considerably reduced the underlay path lengths in three network sizes.

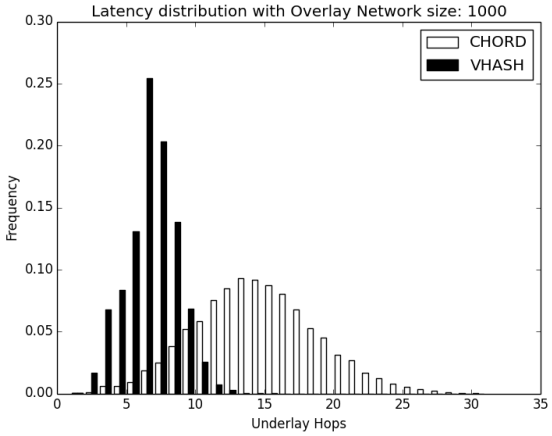
We also sampled the lookup length measured in overlay hops for a 1000 sized Chord and VHash network. As seen in Figure 4.4, the paths measured in overlay for VHash were significantly shorter than those in Chord. In comparing the overlay and underlay hops, we find that for each overlay hop in Chord, the lookup must travel 2.719 underlay hops on average; in VHash, lookups must



(a) Frequency of path lengths on Chord and VHash in a 100 node overlay.



(b) Frequency of path lengths on Chord and VHash in a 500 node overlay.



(c) Frequency of path lengths on Chord and VHash in a 1000 node overlay.

Figure 4.3: Figures 4.3a, 4.3b, and 4.3c show the difference in the performance of Chord and VHash for 10,000 routing samples on a 10,000 node underlay network for differently sized overlays. The Y axis shows the observed frequencies and the X axis shows the number of hops traversed on the underlay network. VHash consistently requires fewer hops for routing than Chord.

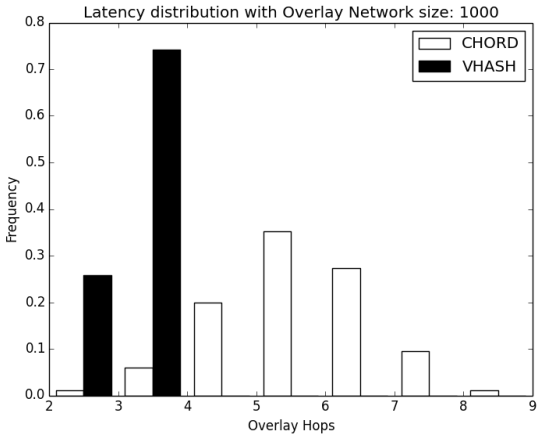


Figure 4.4: Comparison of Chord and VHash in terms of overlay hops. Each overlay has 1000 nodes. The Y axis denotes the observed frequencies of overlay hops and the X axis corresponds to the path lengths in overlay hops.

travel 2.291 underlay hops on average for every overlay hop traversed.

Recall that this work is based on scale free networks, where latency improvements are difficult. An improvement of 0.4 hops over a diameter of 3 hops is significant. VHash has on average less overlay hops per lookup than Chord, and for each of these overlay hops we consistently traverse more efficiently across the underlay network.

4.3 Remarks

Voronoi tessellations have a wide potential for applications in ad-hoc networks, massively multiplayer games, P2P, and distributed networks. However, centralized algorithms for Voronoi tessellation and Delaunay triangulation are not applicable to decentralized systems. In addition, solving Voronoi tessellations in more than 2 dimensions is computationally expensive.

We created a distributed heuristic for Voronoi tessellations in an arbitrary number of dimensions. Our heuristic is fast and scalable, with a expected memory cost of $(3d+1)^2+3d+1$ and expected maximum runtime of $O(\frac{\log^2 N}{\log^2 \log N})$.

We ran two sets of experiments to demonstrate VHash's effectiveness. Our first set of experiments demonstrated that our heuristic is reasonably accurate and our second set demonstrates that reasonably accurate is sufficient to build a P2P network which can route accurately. Our second experiment showed that VHash could significantly reduced the latency in Distributed Hash Tables.

Chapter 5

UrDHT

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. 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 [**?**]. We found that all DHTs can cleanly be mapped to the primal-dual problems of Voronoi Tessellation and Delaunay Triangulation. 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.

5.1 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.

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.

5.1.1 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 [**?**]. 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 5):

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

[1] Unique with astronomically high probability, given a large enough consistent hashing algorithm.
[2] There is typically a *delete(key)* operation too, but it is not strictly necessary.
[3] This order matters, as some DHTs such as Chord are unidirectional.

Algorithm 5 The DHT Generic Routing algorithm

```
1: function n.LOOKUP((key))
2:     if key ∈ n's range of responsibility then
3:         return n
4:     end if
5:     if One of n's short peers is responsible for key then
6:         return the responsible node
7:     end if
8:     candidates = short_peers + long_peers
9:     next ← min(n.distance(candidates, key))
10:    return next.lookup(key)
11: end function
```

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 [**?**]. 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 Delaunay Triangulation and Voronoi Tessellation is show in Figure 5.1.

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 [**?**].

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 [**?**].

Is there an algorithm we can use to efficiently calculate Delaunay Triangulation for a distributed

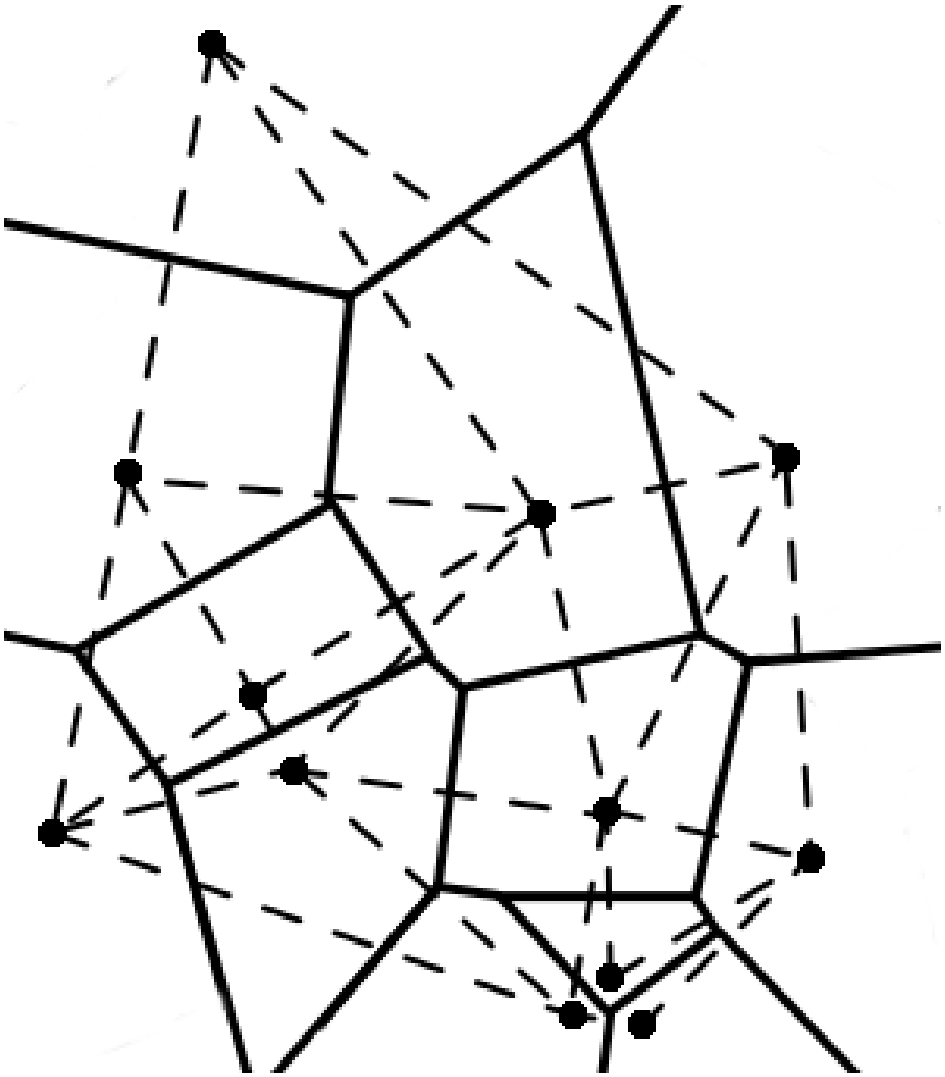


Figure 5.1: An example Voronoi diagram for objects on a 2-dimensional space. The black lines correspond to the borders of the Voronoi region, while the dashed lines correspond to the edges of the Delaunay Triangulation.

system in an arbitrary space? We created an algorithm called the Distributed Greedy Voronoi Heuristic (DGVH), explained below [**?**].

5.1.2 Distributed Greedy Voronoi Heuristic

The Distributed Greedy Voronoi Heuristic (DGVH) is an efficient method for nodes to approximate their individual Voronoi region (Algorithm 6). DGVH selects nearby nodes that would correspond to points connected to it within a Delaunay Triangulation. Our previous implementation relied on a midpoint function [**?**]. We have refined our heuristic to render a midpoint function unnecessary.

The heuristic is described in Algorithm 6. 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 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.

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 5.2.1.

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

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

[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.

[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 $O(\frac{\log^2 n}{\log^2 \log n})$.

Algorithm 6 Distributed Greedy Voronoi Heuristic

```
1: Given node n and its list of candidates.
2: Given the minimum table_size
3: short_peers ← empty set
4: long_peers ← empty set
5: Sort candidates in ascending order by each node's distance to n
6: Remove the first member of candidates and add it to short_peers
7: for all c in candidates do
8:     if any node in short_peers is closer to c than n then
9:         Reject c as a peer
10:    else
11:        Remove c from candidates
12:        Add c to short_peers
13:    end if
14: end for
15: while |short_peers| < table_size and |candidates| > 0 do
16:    Remove the first entry c from candidates
17:    Add c to short_peers
18: end while
19: Add candidates to the set of long_peers
20: handleLongPeers(long_peers)
```

only can we implement the contrived topologies of existing DHTs, but more generalizable topologies like Euclidean or hyperbolic geometries. We show in Section 5.4 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.

5.2 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 [**?**].

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.

5.2.1 The DHT Protocol

The DHT Protocol (`LogicClass.py`) [**?**] is the shared functionality between every single DHT. It consists of the node's information, the short peer list 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 5.2.2.

5.2.2 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 6).

- 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 [**?**] and other small-world networks [**?**] 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 [**?**]. 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 [**?**]. Long peers do not greatly contribute to maintenance overhead, so we chose 200 long peers as a default maximum.

5.3 Implementing other DHTs

5.3.1 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.

5.3.2 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 [**?**] 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 [**?**].

5.3.3 ZHT

ZHT [**?**] 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.

5.3.4 Implementing a DHT in a non-contrived Metric Space

We used a Euclidean geometry as the default space when building UrDHT and DGVH [**?**]. 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 [**?**].

Long peers are randomly selected from the left-over candidates after DGVH is performed [**?**]. 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 [**?**] [**?**]. This has the potential to allow for minimal latency DHTs.

5.4 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 [**?**] [**?**] [**?**] [**?**] [**?**] [**?**].

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.
- 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 [**?**] [**?**]. This demonstrates that UrDHT is capable of accurately emulating these topologies. We show these results in Figures 5.2 - 5.5.

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 in Figures 5.6 - 5.9.

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.

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

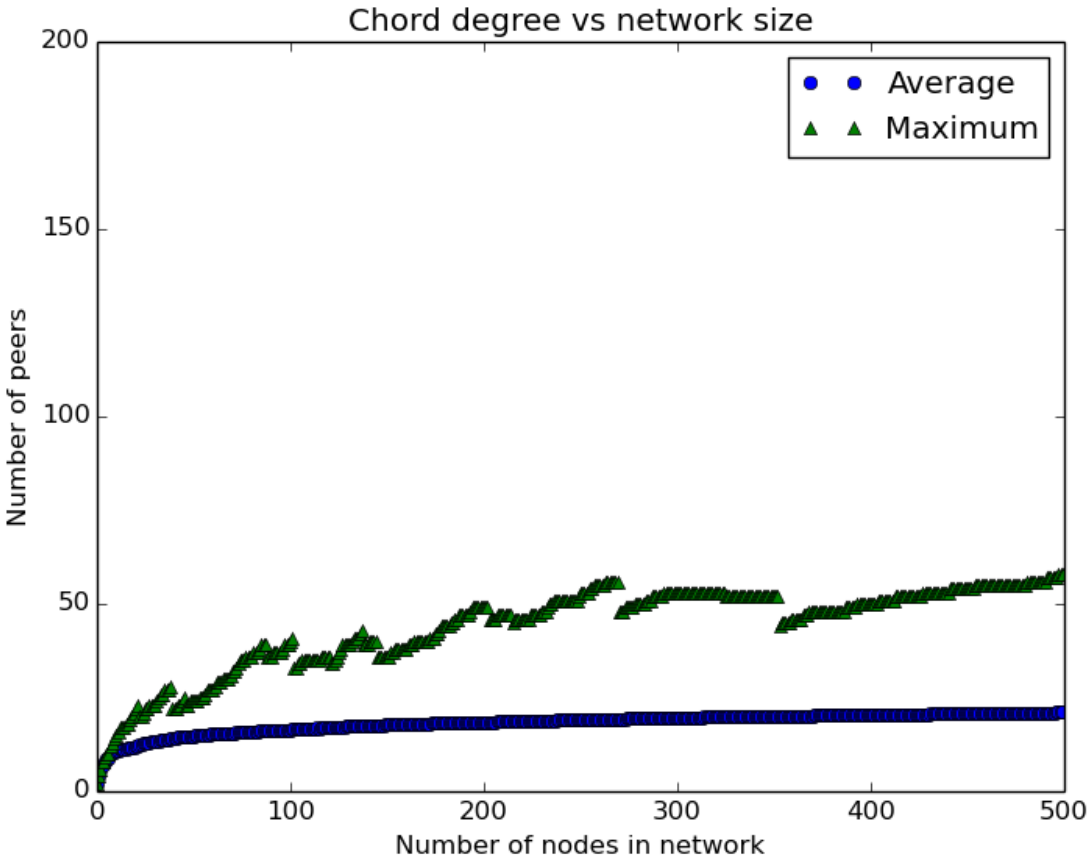


Figure 5.2: This is the average and maximum degree of nodes in the Chord network. This Chord network utilized a 120 bit hash and thus degree is bound at 122 (full fingers, predecessor and successor) when the network reaches 2^{120} nodes.

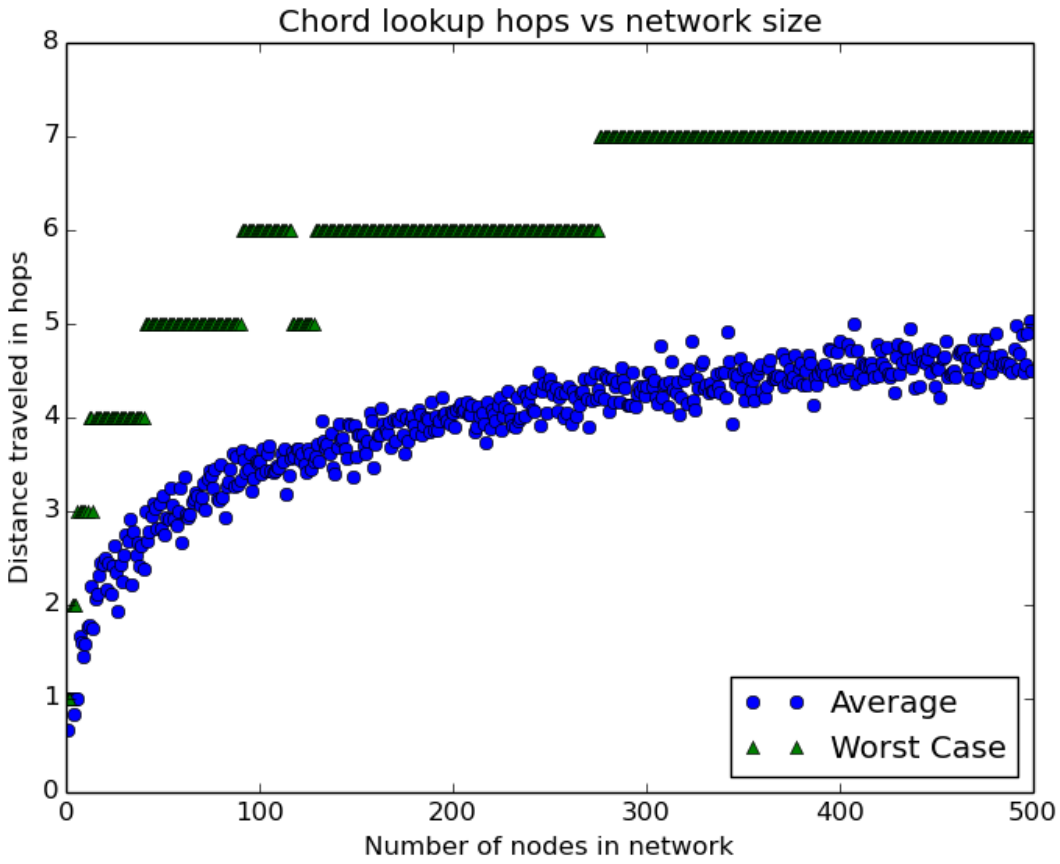


Figure 5.3: This is the number hops required for a greedy routed lookup in Chord. The average lookup between two nodes follows the expected logarithmic curve.

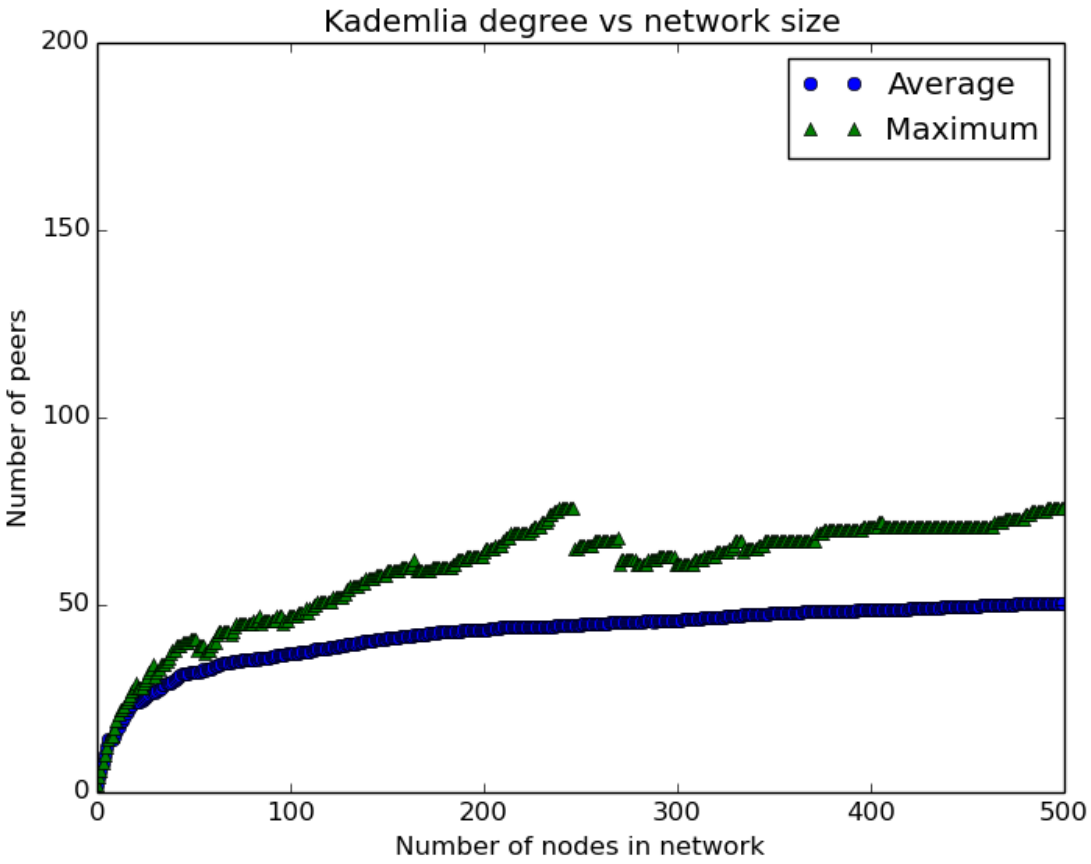


Figure 5.4: This is the average and maximum degree of nodes in the Kademlia network as new nodes are added. Both the maximum degree and average degree are $O(\log n)$.

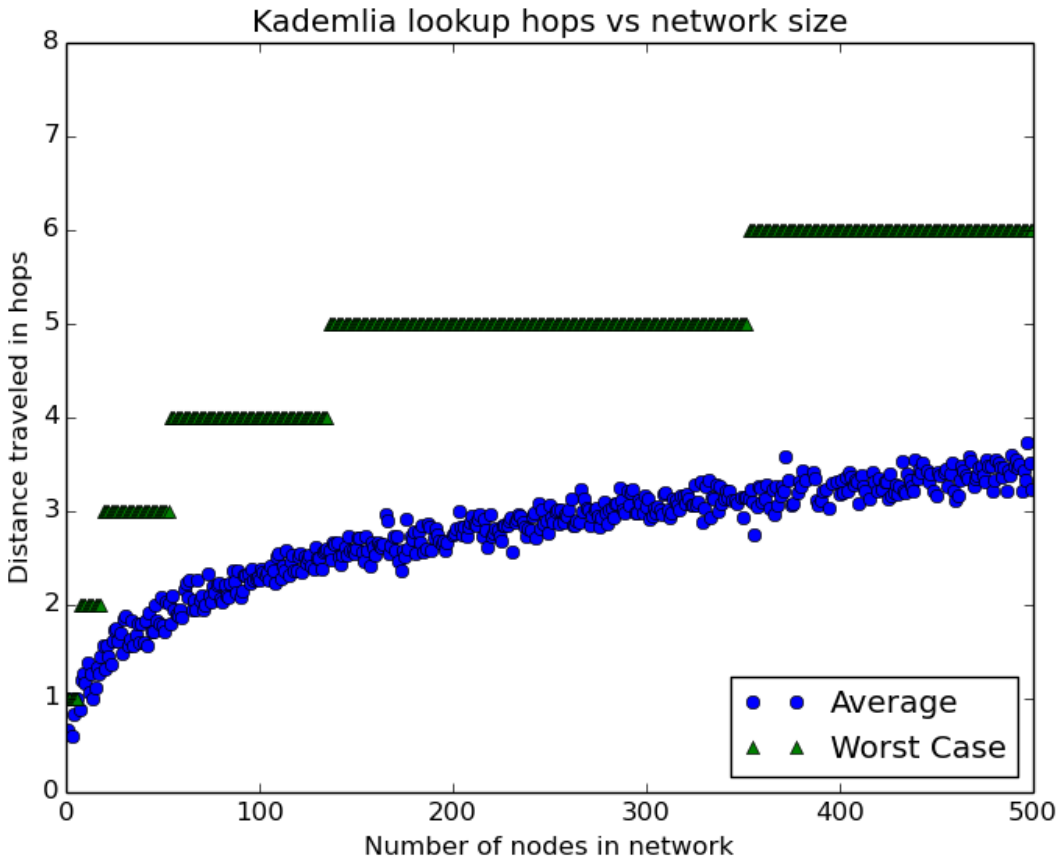


Figure 5.5: Much like Chord, the average degree follows a distinct logarithmic curve, reaching an average distance of approximately three hops when there are 500 nodes in the network.

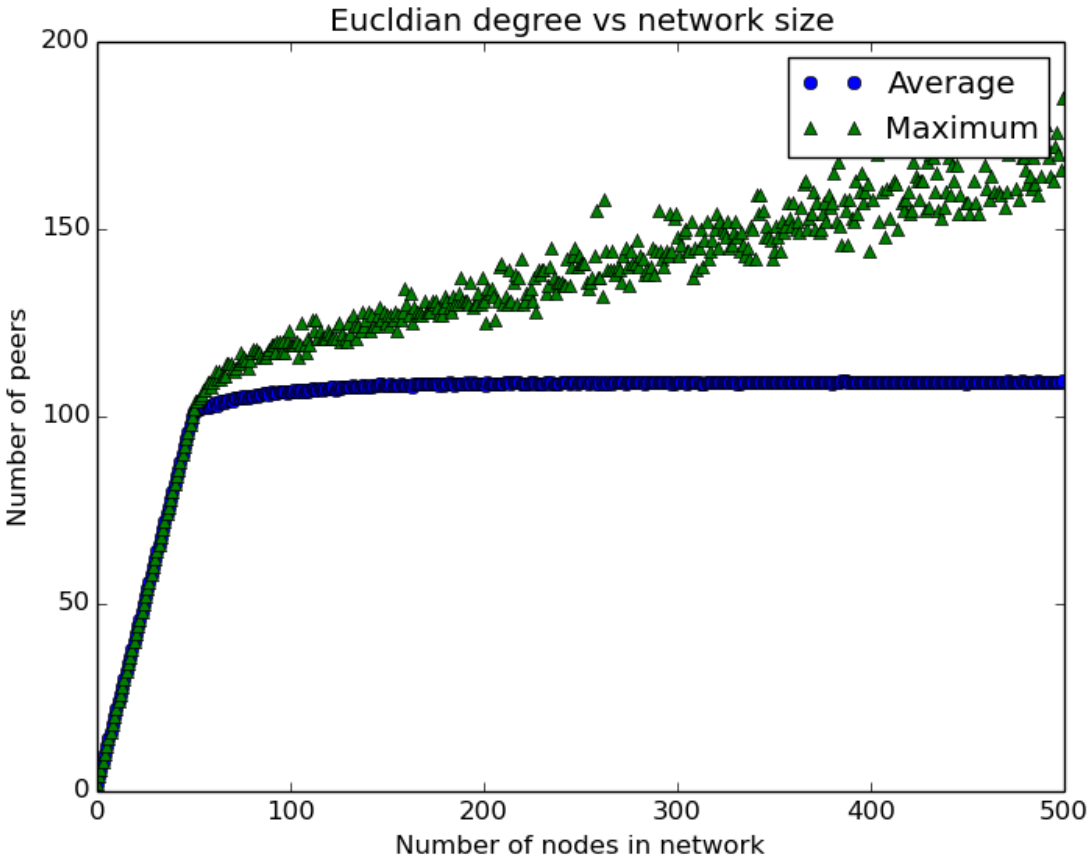


Figure 5.6: Because the long peers increase linearly to the maximum value (49), degree initially rises quickly and then grows more slowly as the number of long peers ceases to grow and the size short peers increases with network size.

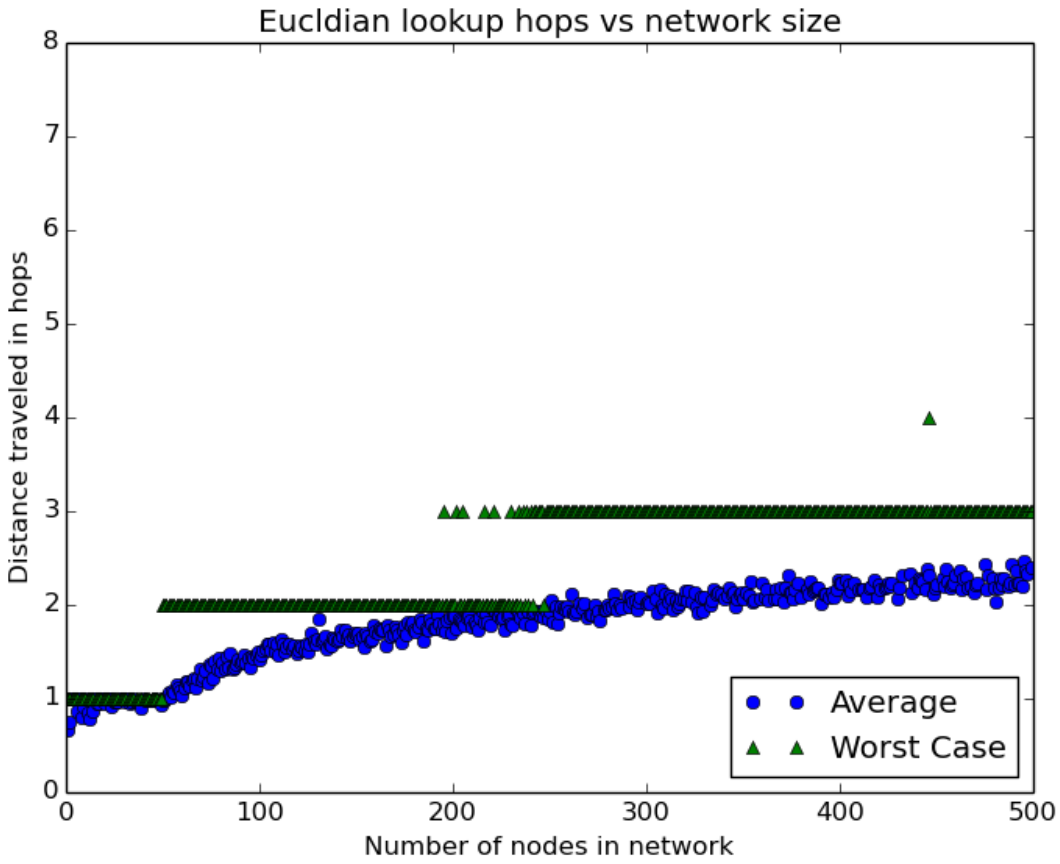


Figure 5.7: The inter-node distance stays constant at 1 until long peers are filled, then rises at the rate of a randomly connected network due to the distribution of long peers selected

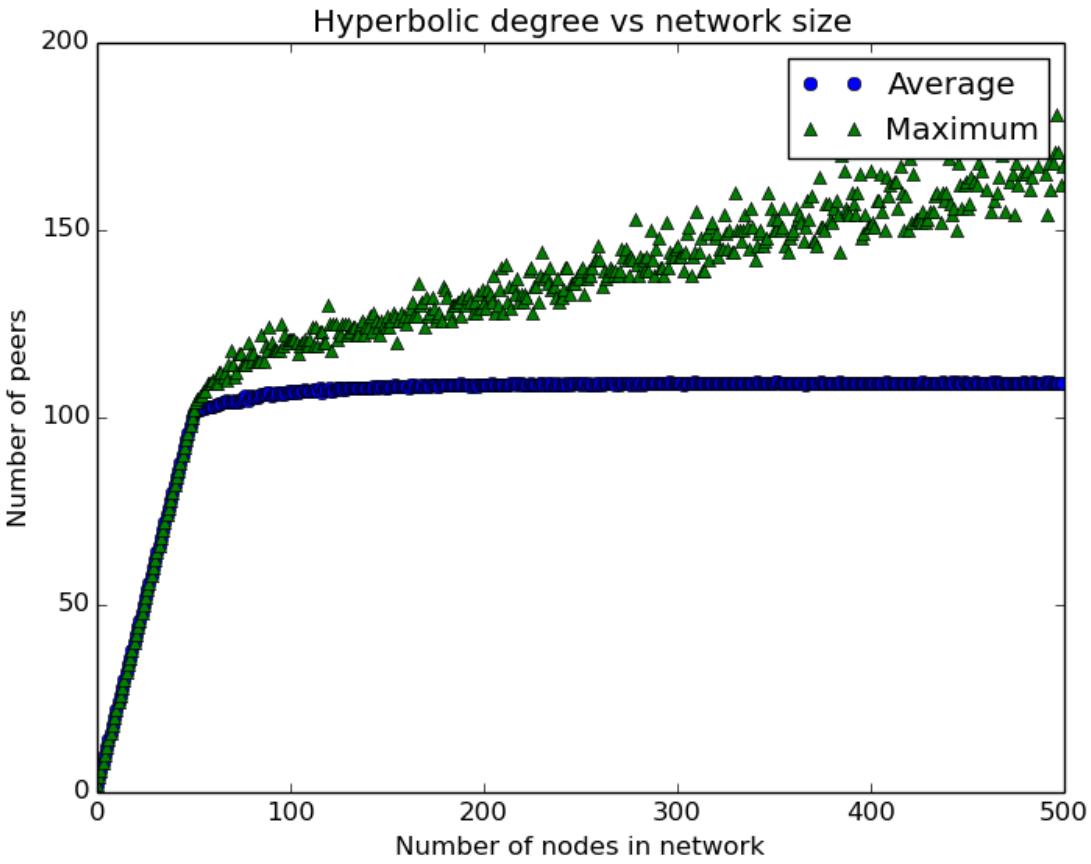


Figure 5.8: The Hyperbolic network uses the same long and short peer strategies to the Euclidean network, and thus shows similar results.

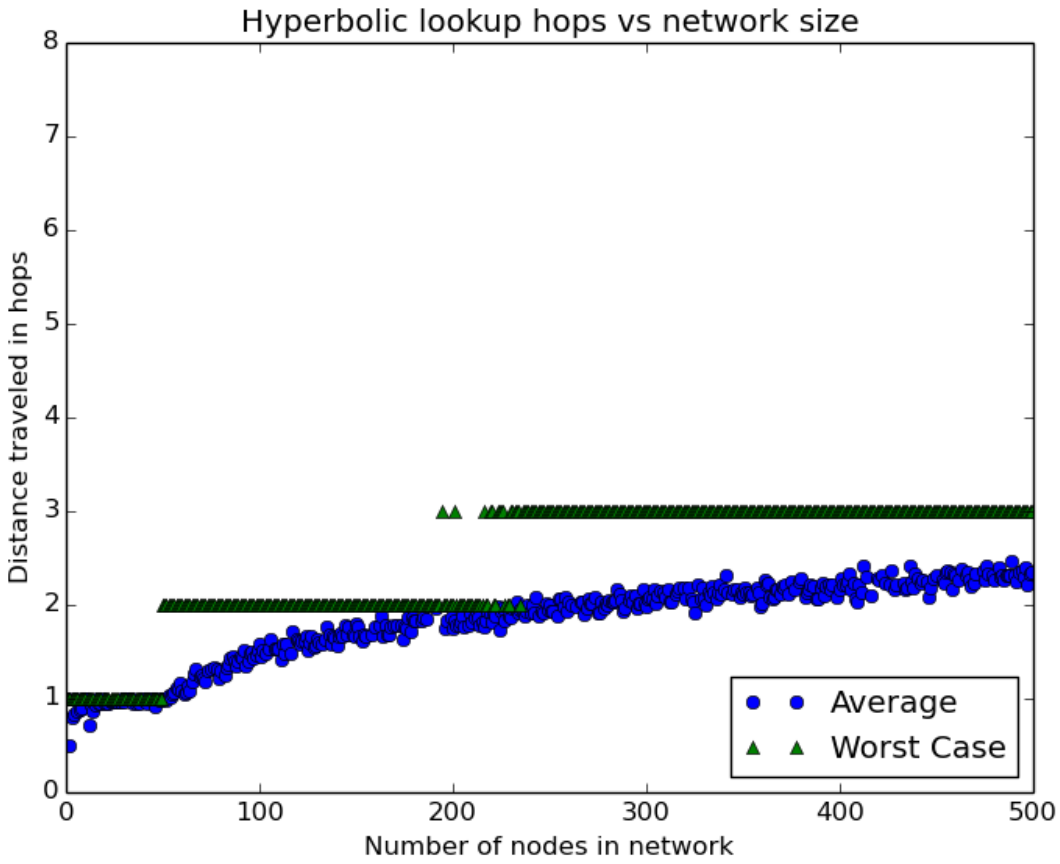


Figure 5.9: Like the Euclidean Geometry, our Poincarè disc based topology has much shorter maximum and average distances.

5.5 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 [**?**] [**?**]. 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 [**?**]. P2's concise Overlog statements also present a sharp learning curve for many developers. These present challenges not seen with UrDHT.

The T-Man[**?**] and Vicinity [**?**] 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.

5.6 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 [**?**] [**?**]. 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.

5.7 Remarks

Chapter 6

D^3DNS

6.1 Introduction

The Domain Name System, commonly referred to as DNS [**?**] [**?**], is a fundamental component of the Internet. DNS maps memorable names to the numerical IP addresses used by computers to communicate over IP.

Two recent events in the United states have brought DNS to the forefront of networking and security research. First is recent legislation proposed in the US House of Representatives and US Senate. The Stop Online Piracy Act (SOPA) [**?**] and PROTECT IP Act (PIPA) [**?**] were both introduced in 2011. There were numerous aspects to both bills, but essential to both was that DNS servers located in the US would be required filter DNS records on demand, essentially fracturing the DNS system. There would be no guarantee that DNS could serve the same information to two different users.

More recent are the leaks of classified information elucidating the extent of the NSA's spying capabilities. These leaks have raised questions about the security of SSL and TLS, as well as the level of trust that users place in certificate authorities.

These types of threats to DNS, along with security concerns, were not considered when designing the protocol, but DNS is too widely used and too integrated with the Internet as a whole to be replaced. Extensions such as DNSSEC [**?**] add authentication and data integrity, but do not alter the fundamental architecture of the DNS network.

There have been many explorations and attempts [**?**] [**?**] [**?**] to propose a DNS system based

on a distributed hash table (DHT) [**?**]. We extend on those papers by implementing a DHT which minimizes latency distance rather than hop distance and implementing a shared record of ownership based on recent developments in cryptocurrency [**?**] [**?**].

This paper proposes the Distributed Decentralized Domain Name Service, or D^{3}NS. D^{3}NS is a completely decentralized Domain Name Service operating over a DHT. D^{3}NS does not replace the DNS protocol, but rather adds robustness to the architecture as a whole. Internally, D^{3}NS signs all DNS records using public/private keys, providing additional security internal to the DNS system.

We show that D^{3}NS addresses the objections to a decentralized DNS system posed by Cox *et al.* [**?**], specifically: dramatically reducing latency compared to other DHT systems, retaining the extensibility of the original DNS system, and changing the intended scope of use to address incentive issues. We show D^{3}NS allows for new authentication methods and a means of decentralized proof of ownership beyond that of Cox *et al.*'s work.

The rest of the paper is consists of the following sections. Section II gives an overview of DNS and identifies prior research in the area of distributed DNS. Section III defines the various components of D^{3}NS. Section IV covers the modified blockchain used for record authentication in D^{3}NS. Section V presents UrDHT [**?**], a DHT that we designed and used with D^{3}NS. Section VI details our implementation of D^{3}NS and we discuss our conclusions and future work in Section VIII.

6.2 Background

This paper is intended to address concerns raised by Cox *et al.* [**?**] and propose a viable decentralized DNS replacement utilizing a DHT. Our proposed improvements on the DNS alternative presented by Cox *et al.* are fully reverse compatible with the current hierarchical DNS system and provides a shared, authenticated public record that allows for DNSSEC style authentication.

6.2.1 DNS Overview

DNS queries proceed recursively through the DNS hierarchy, beginning with a query to a root server, which then yields a record for a server for the requested top level domain. This server then directs the request to another DNS server responsible for the domain under that, which yields an

answer or another DNS server, which is queried in the same manner.

One of the key concepts of the DNS architecture is that no matter which servers end up being queried, a user can expect to receive a record consistent with what the rest of the DNS system will serve for that particular request.

Recent legislative motions reflect DNS's weakness to be influenced by local government intervention [**?**] [**?**] [**?**].

These bills would have required that servers maintained in the US filter specified domain names, preventing users from obtaining the correct IP address for the domain name in question. Multiple governments have been noted to preform systematic attacks on DNS queries [**?**]. This filtering is incompatible with the DNS Security Extensions (DNSSEC) [**?**] and DNS's intended usage.

The mandated DNS filtering would possibly drive users to unregulated DNS servers, which would create more attack vectors.

6.2.2 Related Work

Cox *et al.* devised a distributed DNS using Chord [**?**] as the storage medium for DNS records, which they called Distributed DNS (DDNS). They examined the possibility of extending DNSSEC with new options for storing keys in the DHT. They encountered the then unsolved problem of proof-of-ownership [**?**] for domain names and found no means of enhancing security.

They noted that the overlay topology of a DHT such as Chord did not take into account latency optimization and thus DNS was not a viable application of a DHT due to significantly greater latency. They also pointed out several possible improvements as a result of using a DHT, namely increased robustness, an auto-balancing structure, resistance to both DDOS attacks and packet injection based DNS spoofing.

Cox *et al.* considered the optimization and security problems solvable, but they postulated that two issues rendered the system unviable. First, they intended that DDNS would replace all DNS servers and traffic, which removed extensibility and customizability of the original DNS protocol. Second, because they considered replacing the entire DNS system rather than a meaningful subset, they presented a question of incentive. What would incentivize companies to support the new system where their servers had to share the load of other companies traffic? What would stop them from holding only their content, while rejecting any additional responsibility assigned to it?

D^{3}NS addresses both of the raised issues. We utilize a side channel method of confirming domain named ownership via a blockchain [**?**] to enable DNSSEC style security at all layers of the network. We propose a DHT structure which allows for minimum latency optimization. Our system aims only to replace authoritative Top Level Domain servers currently managed by registrars, where most records are simply a forward to an authoritative DNS server managed by the domain owner, rather than replacing all levels of DNS. This limiting of scope allows us to continue to take advantage of DNS extensions and as places responsibility of managing the network with those who have an incentive for its continued functioning.

6.3 D^{3}NS

D^{3}NS has logically discrete components which provide DNS efficient record storage, domain name ownership management and verification, and DNS backwards compatibility, all of which may be modularly replaced or have individual optimizations. D^{3}NS uses a DHT to store DNS records in a distributed fashion and a blockchain and Namecoin [**?**] analog to manage domain name ownership. D^{3}NS utilizes public and private key encryption for signing and verifying records.

6.3.1 Distributed Hash Table

Our implementation is not strictly specific to any particular distributed hash table. We examined using Chord [**?**] with DNS, similar to DDNS [**?**]. However, Chord's unidirectional ring overlay topology does not take actual network topology into account and using it for a global scale system is not viable because messages will be routed very inefficiently. D^{3}NS requires a DHT which allows the routing overlay to be optimized to the network topology and conditions in real time.

As a result we chose to develop a prototype DHT to meet this requirement to act as backend to our DNS system called UrDHT [**?**]. UrDHT is built on the idea of constructing the DHTs overlay by using Voronoi Tessellations and Delaunay Triangulation. Essentially, each node is at the center of Voronoi region and responsible for the records that fall in that region. The peers the node connects with correspond to that node's Delaunay neighbors. Because UrDHT works with the high level abstractions of Voronoi Tessellation and Delaunay Triangulation, UrDHT can easily be configured to work in any arbitrary metric space or imitate the overlay topology of any DHT, such as Chord.

For D^{3}NS, UrDHT uses a d-dimensional unit cube which wraps around the edges in a toroidal fashion as the space for the overlay. Each record in the DHT is assigned a location in that space. Each node is assigned a location and is responsible for records to which it is the closest node. The variable dimensionality is allowed so that problems can be embedded into the space with relative ease and records can be assigned locations which have meaning concerning the problem in which they are a part. This way, records that are close to each other in the problem formulation are close to each other in the DHT and are likely hosted on the same node. This offers speedup for many distributed algorithms which require traversal of data.

6.3.2 DNS Frontend

Because this system is intended to be reverse compatible with the existing DNS protocol, we serve the data provided by the DHT after it has been authenticated by the block chain to other DNS servers or clients. DNS nodes incorporated into the D^{3}NS system will not request data from other DNS servers and will only exchange data via the DHT.

6.4 Blockchain

We use a tool called a *blockchain* for maintaining and authenticating our DNS records. Blockchains have their roots in the cryptocurrency Bitcoin [**?**], where it is used to authenticate financial transactions and verify account balances. While there have been similar attempts to leverage Bitcoin's mechanisms extend DNS [**?**], they have been strictly tied to the concept of currency and not yet academically explored.

6.4.1 Blockchains in Bitcoin

Bitcoin is a decentralized electronic currency. Here we are particularly concerned with Bitcoin's blockchain. Bitcoin's blockchain consists of a shared authenticatable transaction record [**?**] [**?**] .

Bitcoin's blockchain is essentially a shared ledger. The blockchain (Figure 6.1) is a record of every single transaction made using the Bitcoin. Each transaction refers to a previous transaction, indicating the funds handled by the new transaction are in fact owned by the user initiating the transaction. The record is validated by traversing the tree of transactions and marking the refer-

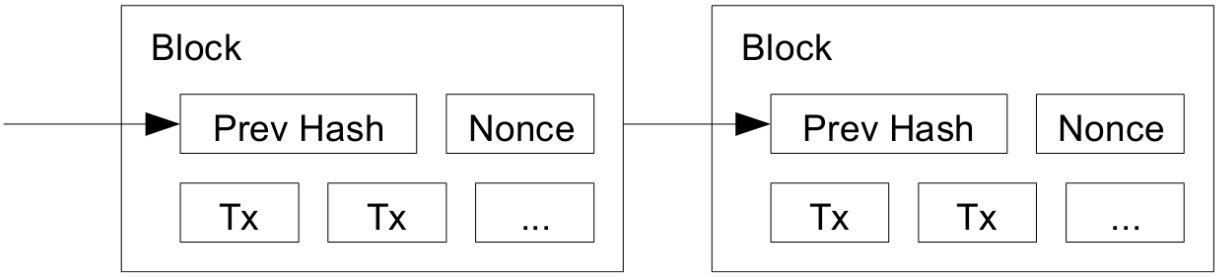


Figure 6.1: A section of the blockchain as defined by Bitcoin [**?**].

enced transactions as used. A valid blockchain has all non-leaf node transactions marked as used only once.

Transactions are grouped together and verified in a *block*, which are linked together in a chain. Each block in the chain is a series of transactions published during the time it takes to generate that block. The process of authenticating these transactions and generating a new block is called mining (Algorithm 7). Mining a block is analogous to the concept of gold mining, as the incentive for successfully mining a block is a large sum of new bitcoins.

A block is mined by generating a nonce field such that the hash of the entire block is less than a global difficulty value. This difficulty sets the rate at which new blocks are mined and is adjusted in reference to rate of mining [1]. When a block is mined, it is transmitted to the network and each transaction in it is validated by each peer. The network will then work on mining the next block.

Algorithm 7 Blockchain mining

Given previous Block B_{-1}
Given New Transaction Set T
Given Difficulty D
Given Reward destination R
New Block $B_0 = HASH(B_{-1})|T|R|Timestamp$
Block Attempt $b = B_0|Nonce$
while $HASH(b) > D$ **do**
    Block Attempt $b = B_0|Nonce$
end while
Propagate b as next block

[1] Bitcoin adjusts the difficulty rate every 2016 blocks, such that the network will then mine a block every ten minutes on average [**?**]

6.4.2 Using the Blockchain to validate DNS records

We utilize the transaction record of Bitcoin to record ownership of domain names. Rather than rewarding miners with currency, the reward and incentive for mining is a record that allots the miner the right to claim a domain name.

Algorithm 8 Blockchain Transaction Validation

```
1: A new transaction t consists of: Award domain D to user U with proof reference P and signature S
2: The transaction set T is the set of all transactions considered valid
3: if P is not marked used then
4:     if owner indicated in P matches signature S then
5:         if D matches domain referenced in P then
6:             Mark P as used
7:             Consider t valid
8:         else
9:             if P is a mining reward then
10:                P is an unclaimed mining reward
11:                if D is not yet claimed then
12:                    Mark P as used
13:                    Mark D as claimed
14:                    Consider t valid
15:                end if
16:            end if
17:        end if
18:    end if
19: end if
```

The transactions in each block indicate miners claiming a new domain or the transfer of domain ownership. Claims of new domains are validated by a reference to an unclaimed mining reward owned by the claiming user. Transfers are validated by a pointer to an unused previous transfer record or claim record indicating ownership by the transferring party. This way every domain name in the system can be associated with the owner's public key. New domains can be claimed and old domains can be transfered between owners. Algorithm 8 shows the process for validating transactions using the blockchain.

6.4.3 Using a Blockchain to Replace Certificate Authority

The shared record of the blockchain allows any participant in the mining network to act as a trusted third party to clients. This way, trust is not centralized at a single point of failure.

Internally, members of the DHT are also members of the blockchain network (as it is convenient to use the DHT overlay as the Blockchain network overlay) and thus all records pushed to the DHT and retrieved records can be confirmed as legitimate before transmission to the end user. This limits the viability of replay or injection based attacks.

6.5 UrDHT

One of the reasons we created UrDHT [**?**] was to allow for spacial representations to be mapped to hash locations, a feature lacking in many current distributed hash tables. In particular, we aimed to construct a mechanism for creating a more efficient global scale DHT built on a minimal latency overlay. Rather than focus on minimizing the amount of hops required to travel from point to point we wish to minimize the time required for a message to reach its recipient. UrDHT actually has a worse worst-case hop distance ($O(\sqrt[d]{n})$) than other comparable distributed hash tables ($O(lg(n))$). However, UrDHT can route messages as quickly as possible rather than traveling over a grand tour that an overlay network may describe in the real world.

The naive method of doing so is to assign coordinates to servers based on the geographic location of nodes. More complex approaches would approximate a minimum latency space based on inter-node latency. Algorithm 9 describes the process for performing a minimum latency embedding using UrDHT

Algorithm 9 UrDHT Minimum Latency Embedding

1: d is the dimensions of the hash space
2: seed the space with $d + 1$ nodes at random locations
3: A node n wishes to join the network
4: n pings a random subset of peers to find latencies L
5: Normalize L onto (0.0,1.0) to yield L_N
6: Choose position p such that

$$\sum_{i \in peers} (L_N[i] - dist(p, i))^2$$

is minimized
7: Re-evaluate location periodically

6.5.1 Toroidal Distance Equation

Given two vector locations $\vec{a}$ and $\vec{b}$ on a d dimensional unit toroidal hypercube:

$$distance = \sqrt[|d|]{\sum_{i \in d} (\min(|\vec{a}_i - \vec{b}_i|, 1.0 - |\vec{a}_i - \vec{b}_i|))^2}$$

6.5.2 Mechanism

UrDHT maps nodes to a d dimension toroidal unit space overlay. This is essentially a hypercube with wrapping edges. The toroidal property makes visualization difficult but allows for a space without a sparse edge, as all nodes can translate the space such that they are at the center of the space. In effect, each node views itself at the center of the graph.

Nodes in UrDHT are responsible for the address space defined by their Voronoi region. This region is defined by a list of peers maintained by the node. A minimum list of peers is maintained such that the node's Voronoi region is well defined. The links connecting the node to its peers correspond to the links of a Delaunay Triangulation. One such possible network is shown on Figure 6.2.

6.5.3 Relation to Voronoi Diagrams and Delaunay Triangulation

UrDHT does not strictly solve Voronoi diagrams [**?**] for a number of reasons. 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 [**?**].

UrDHT uses the Distributed Greedy Voronoi Heuristic [**?**], to create an approximation of Voronoi tessellation and Del. An online algorithm (Algorithm 6 maintains the set of peers defining the node's Voronoi region. The set of peers required to define a node's Voronoi Region corresponds to a solution to the dual Delaunay Triangulation.

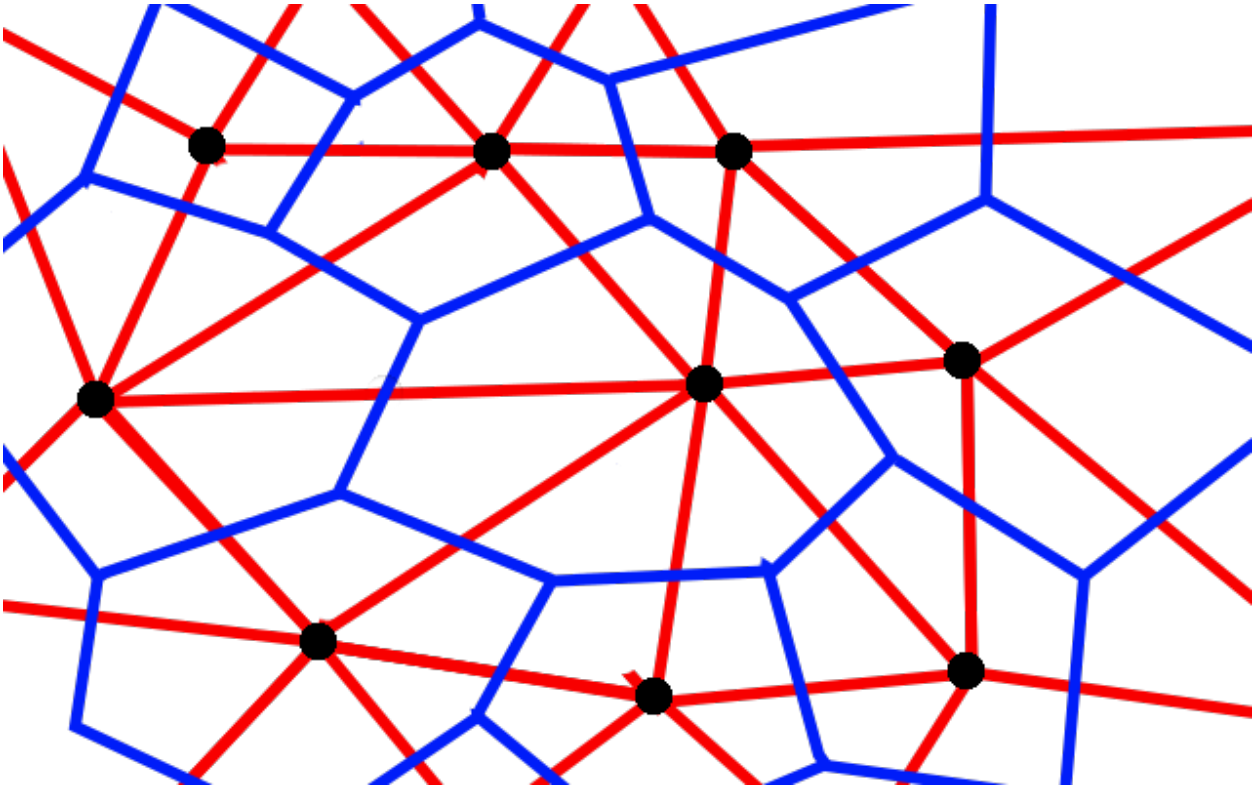


Figure 6.2: The starting network topology. The blue lines demark the Voronoi edges, while the red lines connecting the nodes correspond to the Delaunay Triangulation edges and one-hop connections.

6.5.4 Messages

Maintenance and joining are handled by a simple periodic mechanism. A notification message consisting of a node's information and active peers is the only maintenance message. All messages have a destination hash location which is used to route them to the proper server. This destination can be the hash location of a particular node or the location of a desired record or service. The message is received by the node responsible for the location. Services running on the DHT define their own message contents, such as commands to store and retrieve data.

6.5.5 Message Routing

Messages are routed over the overlay network using a simple algorithm (Algorithm 5). When routing a message to an arbitrary location, a node calculates which Voronoi region the message's destination is in amongst the itself and its peers. If the destination falls within its own region, then it is responsible and handles the message accordingly. Otherwise, the node forwards the message to the closest peer to the destination location. This process describes a precomputed and cached A* routing algorithm [**?**] .

6.5.6 Joining and Maintenance

Joining the network is a straightforward process. A new node first learns the location of at least one member of the network to join. The joining node then choses a location in the hash space either at random or based on a problem formulation (for example, based on geographic location or latency information).

After choosing a location, the joining node sends a *join* message to its own location via the known node. The message is forwarded to the current owner of that location who can be considered the "parent" node. The parent node immediately replies with a maintenance message containing its full peer list. This message is sent to the joining node, who then uses this to begin defining the space it is responsible for.

The joining node's initial peers are a subset of the parent and the parent's peers. The parent adds the new node to its own peer list and removes all his peers occluded by the new node. Then regular maintenance propagates the new node's information and repairs the overlay topology. This

process is described by Algorithm 10.

Algorithm 10 Join

1: new node N wishes to join and has location L
2: N knows node x to be a member of the network
3: N sends a request to join, addressed to L via x
4: node $Parent$ is responsible for location L and receives the join message
5: $Parent$ sends to N its own location and list of peers
6: $Parent$ integrates N into its peer set
7: N builds its peer list from N and its peers
8: regular maintenance updates other peers

Each node in the network performs maintenance periodically by a maintenance message to its peers. The maintenance message consists of the node's information and the information on that node's peer list. When a maintenance message is received, the receiving node considers the listed nodes as candidates for its own peer list and removes any occluded nodes (Algorithm 6).

When messages sent to a peer fail, it is assumed the peer has left the network. The leaving peer is removed from the peer list and candidates from the set of 2-hop peers provided by other peers move in to replace it. Maintenance is described by Algorithms 11 and 12. Figures 6.3, 6.4, and 6.5 illustrate the joining processing.

Algorithm 11 Maintenance Cycle

1: P_0 is this node's set of peers
2: T is the maintenance period
3: **while** Node is running **do**
4: **for all** node n in P_0 **do**
5: Send a Maintenance Message containing P_0 to n
6: **end for**
7: Wait T seconds
8: **end while**

There is no function for a "polite" exit from the network. UrDHT assumes nodes will fail and the difference between an intended failure and unintended failure is unnecessary. The only issue this causes is that node software should be designed to fail totally when issues arise rather then attempt to fulfill only part of its responsibilities.

Algorithm 12 Handle Maintenance Message

```
1: P_0 is this node's set of peers
2: Receive a Maintenance Message from peer n containing its set of peers:P_n
3: for all Peers p in P_n do
4:     Consider p as a member of P_0
5:     if p should join P_0 then
6:         Add p to P_0
7:         for all Other peers i in p do
8:             if i is occluded by p then
9:                 remove i from P_0
10:            end if
11:        end for
12:    end if
13: end for
```

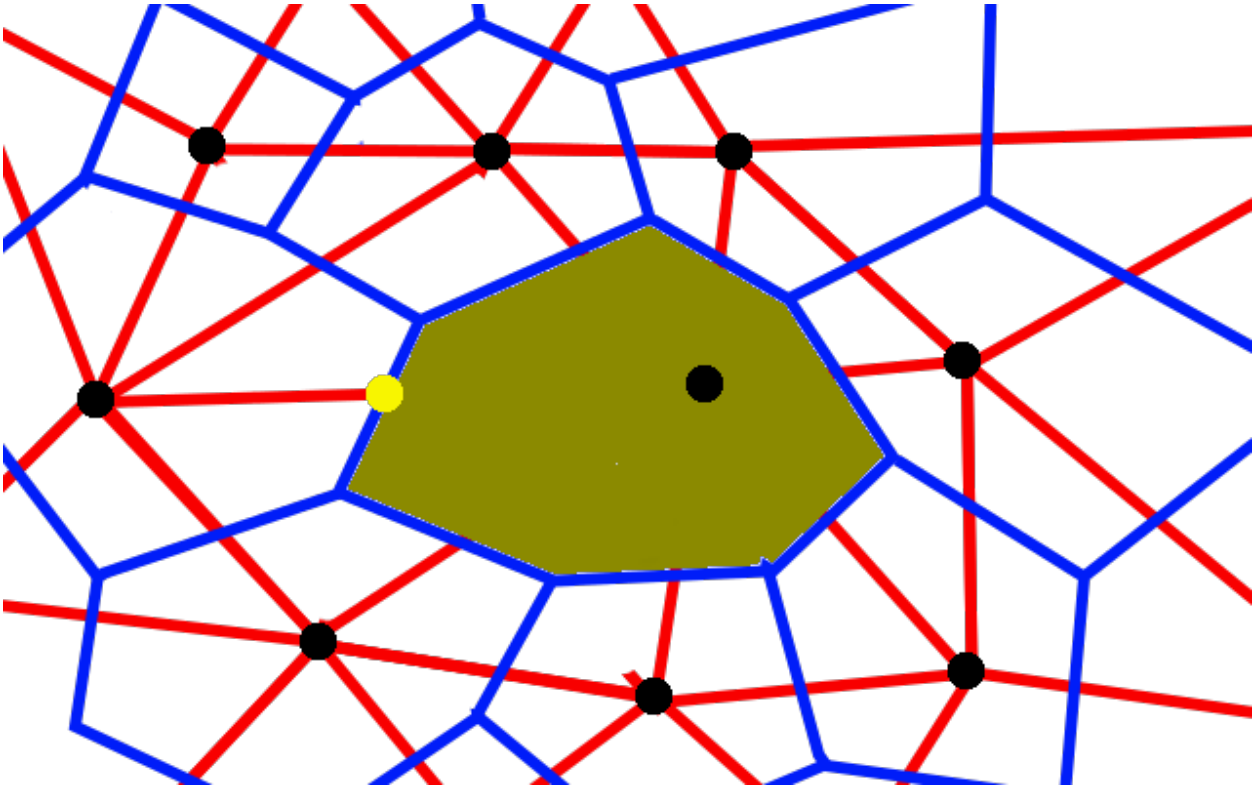


Figure 6.3: Here, a new node is joining the networks and has established that his position falls in the the yellow shaded Voronoi region.

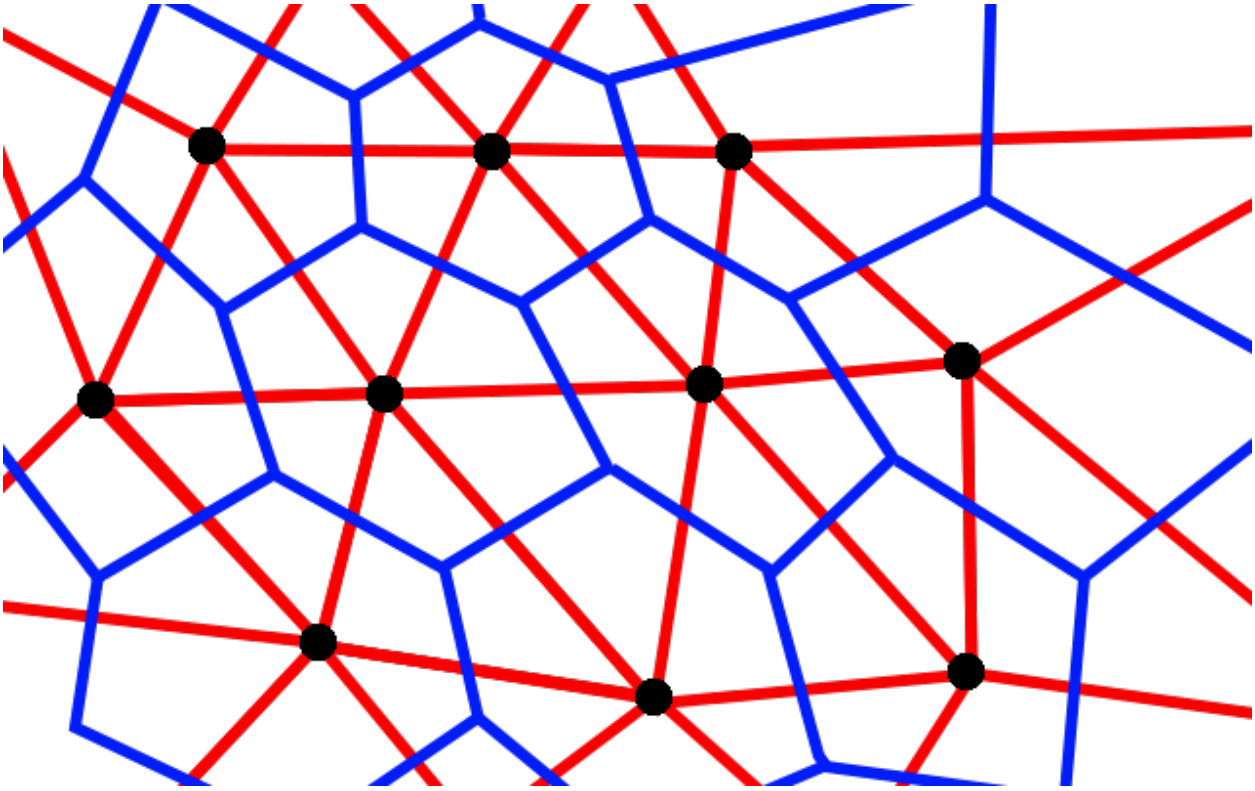


Figure 6.4: The network topology after the new node has finished joining.

6.5.7 Data Storage and Backups

The primary goal of a DHT is to provide a distributed storage medium. We extend this idea to distribute work and information among nodes using the same paradigm. Resources in the network, be it raw data or assigned tasks, are assigned hash locations. The node responsible for a given hash location is responsible for the maintenance of that resource. When a node fails, its peers take responsibility of its space. Thus it is important to provide peers with frequent backups of a node's assigned resources. That way, when a node fails, its peers can immediately assume its responsibilities.

When a resource is to be stored on the network, it is assigned a hash location. The hash locations assigned could be random, a hash of an identifier, or have specific meaning for an embedding problem. The node responsible for that resource's hash location stores the resource.

A resource is accessed by contacting the node responsible for the resource. However, the requester generally has no idea which node is responsible for any particular resource. The data request messsage is addressed to the location corresponding to the resource, rather than the node

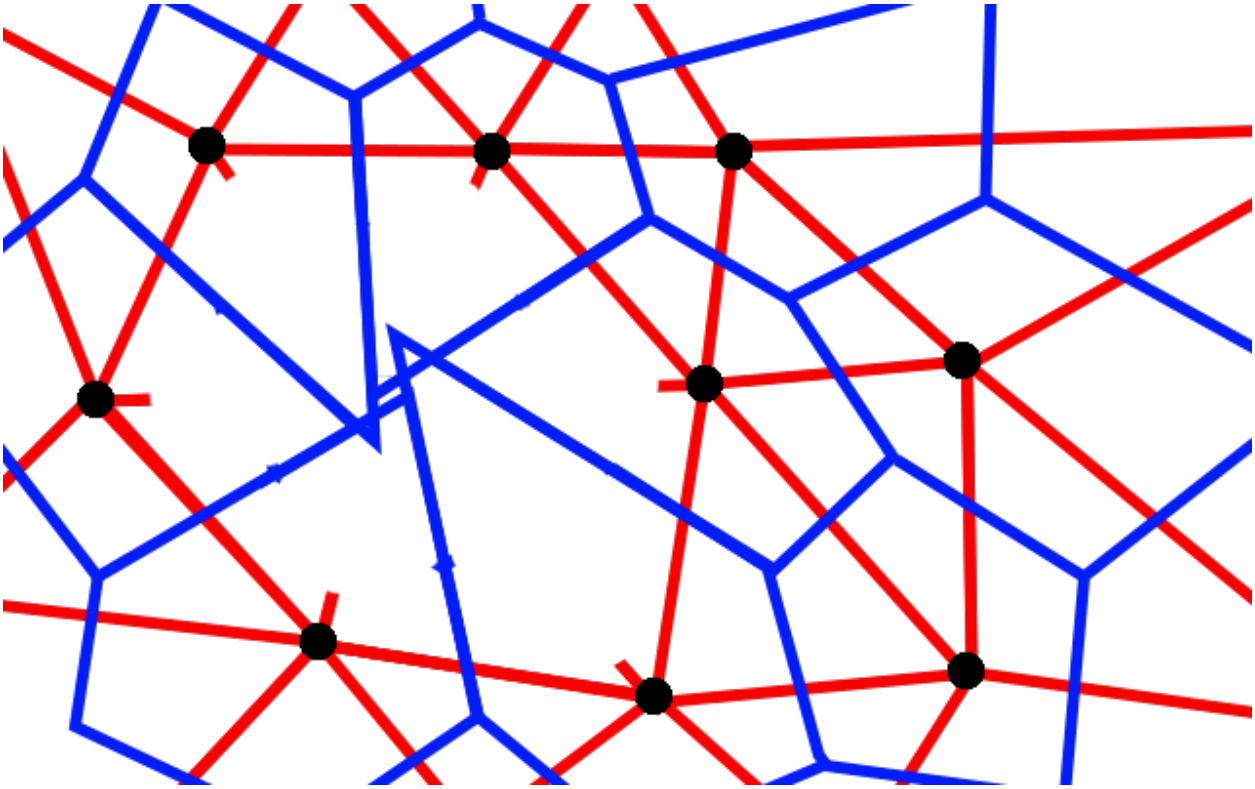


Figure 6.5: The topology immediately after the new node leaves the network. After maintenance takes place, the topology repairs itself back to the configuration shown in Figure 6.2.

responsible for that location. The message is forwarded over the overlay network, each hop bringing the node closer until it reaches the responsible node, who sends the resource or an error if the resource does not exist.

Some options are immediately apparent for dealing with wasted storage space. A system that is primarily read driven can record the time of the last read or a frequency of reads such that resources that are not read often enough are deleted after a certain period of time. If a system is write driven, allow the resource to be assigned a time to live, which can be updated as needed.

A node periodically sends a message containing backups of the resources for which it became newly responsible for to each of its peers. To minimize bandwidth and time wasted by backups, the node should only send the records changed since last backup.

6.6 Implementation

This section describes how the components of the D^{3}NS system piece together as a coherent whole. We demonstrate how this system would be used in the real world to provide a better Domain Name Service.

6.6.1 Establishment of a New Domain

Under the current DNS system, a new domain name is purchased from a company registered with the Internet Corporation for Assigned Names and Numbers (ICANN). That company adds the domain name and a record provided by the owner to the TLD servers. The owner or management company then maintains a name server to answer DNS requests for the purchased domain. In D^{3}NS, new domain names are instead awarded as part of the blockchain mining process or purchased from a previous owner, then transferred to the new owner. These assignments and transfers are both recorded in the blockchain.

A prospective domain owner can use their own mining software and mine a domain name or purchase a domain name voucher from a miner and exchange it for a domain name. In the blockchain, domain name owners are referred to by their public key. The public key is used to authenticate both records and transfers of domains. Loss of the private key of an account will result in the loss of ability to update DNS records and the ability to transfer the domain.

6.6.2 Updating Records for a Domain

A domain name record in the current DNS system is used to indicate a record on your own Name Server or to configure the record held by the TLD server that contains the address record. Using D^3NS, all records must be signed using their owner's private key and confirmed with the public key.

A properly configured D^3NS server should not accept any DNS records which have not been signed by their owner or accept a record with an older version number. To push a new DNS record for a domain, the owner must create the record set for the domain and then sign and submit it to a node on the DHT. The DHT will forward the record to the responsible party and store it after confirming its validation. The new record will begin to be broadcast to clients after old records begin to expire and caches seek new records.

6.6.3 Looking up a DNS record

In the current DNS configuration, a record is looked up using a UDP system that queries a tree of servers. clients send queries to a local DNS server which acts as a DNS resolver and cache. If a portion of the domain name is unknown, the resolver sends a request to the responsible server and looks the record up recursively.

This system is largely unchanged under D^3NS from the point of view of the DNS resolver and client. Ideally, the resolver or client has chosen a nearby member of the D^3NS network as its root domain name server (it can also maintain a large list of backup servers, should the current one fail). The resolver requests a domain's record from the D^3NS node. The node then forwards the request to the responsible party. If any node along the route has a valid cache of the required record, then that server responds and routes the message back to the entry node. All nodes along the route cache the response to aid future queries.

6.6.4 Caching

DNS needs to aggressively cache lookups. Previous investigations into optimizing caching on a DHT saw favorable results [**?**]. However, with the specific application of DNS in mind, the time-to-live field on DNS records defer the caching optimization problem onto users. While it may be tempting

to utilize the built in time-to-live field for caching on the DHT, the possible cyclical nature of cache passing may result in difficulty in propagating new records.

Integrated File Replication and Consistency Maintenance (IRM) [**?**] views the process of caching and keeping the cache up to date as components of larger problem. Nodes in the DHT keep track of how often records are requested and cache those records once a defined rate is passed. Nodes then request an update for the cache based on how often the record is requested and how often that request is expected to be changed by the owner.

In the current implementation of DNS, caching involves a time-to-live field defined by the domain owner. This means that no effective cache optimization done by the server; rather the server that trusts records to have a sensible time-to-live value. IRM [**?**] can be used to approximate the correct time-to-live value for a record.

6.7 Conclusion and Future Work

D^3NS was successfully implemented to act as a DNS server. A user would query the a computer we set up as a DNS gateway which was a member of the DHT and mining network. If the queried domain had a record stored in the DHT and an owner established in the blockchain, the server would reply with the stored DNS records. Otherwise the server would reply with a DNS failure. We ran the DHT and mining network on a cluster of computers and an artificially low difficulty such that a live demo of mining would be viable.

Using all of these components together allowed us to create a system with the following features:

- Robustness - The DHT and Blockchain are both robust to failures and attacks.
- Extensibility - The DNS reverse compatibility allows any DNS extension to be utilized, if dynamic resolution is required a name server record can be stored in the DHT to point to a user's specialized DNS servers.
- Decentralization - Both the DHT and Blockchain can operate without the support of any controlling organization, this offers security against corruption and abuse.

D^3NS successfully addresses the challenges laid down by Cox *et al.* [**?**] in regards to designing a DNS replacement running over a DHT. We encourage criticism, revision, and adoption of this new

system.

6.7.1 Unaddressed Issues and Future Work

The Blockchain does not solve all security issues relevant to DNS authentication and security. Exit nodes could lie or have packets inject to clients until the protocol from DNS server to client is improved. The DHT structure opens up unexplored disruption attacks on the overlay topology.

A series of issues relevant to UrDHT are not addressed in this paper due to limiting of scope, space, time, and a lack of actual solutions to the problems: The exact method of caching to optimize lookup time under real world usage. The overlay network being mapped onto latency results in nodes whose failure due to natural disaster to be comorbid with it's neighbors resulting to data loss. Comorbidity could be counteracted by more complex backup schemes.

Future work will explore various solutions to theses issues, as well as generalized improvements to D^3NS as whole. In particular, the technique of allying a DHT value store for real-time data and a Blockchain for ownership verification may prove a viable technique for decentralizing other web services and enabling new shared storage and computation mediums.

Chapter 7

Analysis of The Sybil Attack

Chapter 8

Autonomous Load Balancing

The following analysis and simulation relies on an important assumption about DHT behavior often assumed but not necessarily implemented. We assume that nodes are active and aggressive in creating and monitoring the backups and the data they are responsible for. Specifically, we will assume it takes T_{detect} time for a node to detect a change in their responsibility or to detect a new node to hand a backup to and that this check is performed regularly. Another assumption is that nodes do note have control in choosing their IDs from the range of hash values.

Smaller chunking results in more files spread throughout the network and a greater chance of the data being evenly spread across the network

The chances of a critical failure happening within a time interval T is the chances of some chain or cluster of nodes responsible for a single record dying within T:

$$r^s$$

Where r is the failure rate over that time interval and s is the number nodes storing that record, either as a primary system, or a backup. Incidentally, this time interval $T = T_{detect} + T_{transfer}$

8.1 The Simulation

We simulate an UrDHT Voronoi based-network in multiple dimensions. [1]

[1] UrDHT in one dimension is a Chord ring with the definition of responsibility changed to a node being responsible to all data closest to it. A 2-dimensional network will emulate the performance of CAN.

We assume that the network starts our experiments stable and the data necessary already present on the nodes and backed-up.

8.1.1 The Parameters

Constants

Time Unit In a simulation, normal measurements of time such as a second are arbitrary, so we be using the abstract *tick* to measure time. If we want to be more concrete, a tick is the amount of time it takes a node to complete one task per sybil and perform the appropriate maintanence.[2]

Maintence We assume the reactive, aggressive backups works.

Hash Functions We will be using SHA-1 [**?**], a 160-bit hash function.

Experimental Variables

Churn Measured in ticks, this can be self induced or a result of actual turbulence in the network. Like most analyses of churn [**?**], we assume churn is constant throughout the experiment and that the joining and leaving rate are equal.

Network Size How many nodes start in the network. We assume that this can grow during the experiment, either via churn or by creating sybils.

Pool Size The size of the pool of potentially joining nodes. Each tick, each node in the pool has a chance to join the network, dictated by the churn rate. Talk about relative equalibrium

Size of the job The size of the job, in tasks. This number is typically orders of maginitude greater than the network size.

[2]The shortest unit of time in the multiverse is the New York Second, defined as the period of time between the traffic lights turning green and the cab behind you honking.
– Terry Pratchett

8.2 Baseline Readings

8.3 Induced Churn

In this experiment, we rely solely on churn to perform load balancing. We use the assumption that

8.4 Random Sybil Injection

Our second series of experiments focused on nodes with low amounts of work performing a controlled and strategic Sybil attack [**?**] on the network.

No benefit waws shown by increasing maxSybils beyond 10, so we stopped increasing it there.

Chapter 9

Conclusion

Distributed Hash Tables (DHTs) are extremely powerful frameworks for distributed applications that are based off the simple and powerful hash tables. Because DHTs were designed with P2P applications in mind, DHTs are scalable, fault-tolerant, and load-balancing. These are exactly the qualities needed in a distributed computing framework.

We created a new DHT called VHash, which was based off the relationship we discovered between the DHTs and Voronoi tessellation. VHash is a DHT that operates over a multidimensional space, which allows the embedding of arbitrary metrics into this space. We showed that we can optimize latency in VHash to obtain faster lookup speeds than a traditional DHT, such as Chord. The key to VHash is our Distributed Greedy Voronoi Heuristic (DGVH). DGVH is a sufficiently accurate and fast approximation of Delaunay triangulations. Aside from its application in VHash, DGVH's applications extend to other areas, such as wireless sensor networks.

We have also shown in the previous chapters that we were able to create a prototype distributed computing application [**?**] based on the Chord DHT [**?**]. ChordReduce, as we named it, demonstrated how MapReduce could be performed on the Chord distributed hash table. As a DHT, ChordReduce is completely decentralized and fault-tolerant, able to handle nodes entering and leaving the network during churn. We demonstrated that ChordReduce can efficiently distribute a problem whilst undergoing significant churn and achieve a significant speedup.

During our experiments with ChordReduce, we found an anomaly in which a computation undergoing a significantly high level of churn finished twice as fast than when no churn was involved. This implied to us that there the churn was effectively shuffling around the nodes such that nodes

with no work were taking work from nodes with large amounts of work. We want to use this implication develop a more intelligent autonomous load-balancing mechanism. Such a mechanism would allow underworked nodes to steal work from overworked nodes in the network.

Part of autonomous load-balancing will involve exploiting heterogeneity in the network. We can do this by having more powerful nodes take a proportionally higher amount of work. This involves a process we dubbed *mashing*, which we originally used to analyze the Sybil attack on DHTs [**?**].

Based on the work we have completed, we proposed creating a framework, called UrDHT, and use it to create a distributed computing. As the name implies, UrDHT is meant to be the prototypical DHT, from which we can derive all other DHTs. This framework would make it easy for developers to create not only new DHTs, but new distributed and P2P applications. The application that we plan on creating with UrDHT is a Distributed Computing framework based on ChordReduce.

Our new framework will be able to handle more than just MapReduce problems and incorporate an autonomous load-balancing mechanism, Developers could use our framework to effortlessly organize a disparate set of nodes into a functional distributed computing system and run their own applications. Our framework could be used in numerous contexts, be it P2P or a data center.