5-9-2016

Building Robust Distributed Infrastructure Networks

Brendan Benshoof

BUILDING ROBUST DISTRIBUTED INFRASTRUCTURE NETWORKS

by

Brendan Lane Benshoof

Under the Direction of Robert Harrison

Abstract

Many competing designs for Distributed Hash Tables exist exploring multiple models of addressing, routing and network maintenance. Designing a general theoretical model and implementation of a Distributed Hash Table allows exploration of the possible properties of Distributed Hash Tables. We will propose a generalized model of DHT behavior, centered on utilizing Delaunay triangulation in a given metric space to maintain the networks topology. We will show that utilizing this model we can produce network topologies that approximate existing DHT methods and provide a starting point for further exploration. We will use our generalized model of DHT construction to design and implement more efficient Distributed Hash Table protocols, and discuss the qualities of potential successors to existing DHT technologies.

INDEX WORDS: P2P, Distributed Hash Table, Decentralized, Networking

BUILDING ROBUST DISTRIBUTED INFRASTRUCTURE NETWORKS

by

Brendan Lane Benshoof

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

Doctor of Philosophy

in the College of Arts and Sciences

Georgia State University

2016

BUILDING ROBUST DISTRIBUTED INFRASTRUCTURE NETWORKS

by

Brendan Lane Benshoof

Committee Chair Robert Harrison

Committee Raj Sunderraman

Anu G. Bourgeois

Valerie Miller

Electronic Version Approved:

Office of Graduate Studies

College of Arts and Sciences

Georgia State University

May 2016

Dedication

This work is dedicated to those who have supported me in pursuing my education out of a desire to see me grow:

- My patient and encouraging spouse, Erin Gillman, who's capacity to care for all mankind inspires me daily to try and make the world a better place.
- My parents, Bill and Denise Benshoof, who have always encouraged me to learn,to work hard, and to find new ways to grow.

Acknowledgements

- Dr. Michael Weeks, who introduced me to the possibility of graduate school, teaching, and research.
- Dr. Valerie Miller, who encouraged my curiosity and has always pushed me to do more.
- Dr. Raj Sunderraman, who encouraged me to attend Georgia State and helped my find my role here.
- Dr. Anu Bourgeois, who fostered my interest in networking and distributed systems.
- Dr. Robert Harrison, who has been my valued teacher, sponsor and mentor through this grand adventure.
- Andrew Rosen, who has been my serial co-author, Master of the Written Word, Dragon Slayer, and Good Friend.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

As more and more systems are growing using existing Peer to Peer (P2P) technology, we are building services usable by the public (and exploring even more possibilities) based on a P2P system called a Distributed Hash Table(DHT). While current DHTs are a incredibly effective and a robust solution to maintain a shared state in a P2P system in a scalable fashion, little to no improvement has been provided over the initial Chord[56] and Kademlia[42] DHTs. This dissertation explores improvements and applications of Distributed Hash Tables to provide the capacity for future robust and distributed systems infrastructures and services.

1.1 The Ship of Theseus

> The ship wherein Theseus and the youth of Athens returned from Crete had thirty oars, and was preserved by the Athenians down even to the time of Demetrius Phalereus, for they took away the old planks as they decayed, putting in new and stronger timber in their places, in so much that this ship became a standing example among the philosophers, for the logical question of things that grow; one side holding that the ship remained the same, and the other contending that it was not the same.
>
> -Plutarch, Theseus

What has attracted me to the design and applications of Distributed Hash Tables is that they encourage thinking on a global scale and require designs that operate with minimal human intervention on time scales beyond those normally considered by computer science. In traditional engineering and design we conceive of an object, that while having parts

replaced in the course of maintenance, will largely remain a single static object comprised of the same material of which it was formed until it is disposed of. When considering a Distributed Hash Table, we are intentionally designing a system like the Ship of Theseus, where on the scale of millions of devices are added and lost on a daily basis, often amounting to up to half of the constituent parts of a globe spanning system being replaced in a day. It is our task to design a pattern of behavior for each of these parts that allows the system to respond to queries despite constant in-progress failure and replacement and to constantly propagate the information stored in the system onward to new storage before the current storage fails.

Despite the challenge involved in building such a system, barring dramatic change to our expectations of the future, the existing Distributed Hash Tables will be able to persist as a public utility indefinitely. I present that the answer to the Paradox of the Ship of Theseus is, "I don't care as long as the ship still floats".

1.2 What is a DHT?

At the core of maintaining a distributed system is establishing a shared state in the form of a table of key-value pairs. DHTs provide a mechanism for agreeing upon a very large shared state with tolerable inconsistency (records are sometimes lost, and if mutable they may be inconsistent in value). While other techniques of consensus provide higher confidence, DHTs scale to awesome levels of storage capacity because they are highly tolerant of the failure of nodes within themselves. In practice, DHTs and similar distributed systems' performance are bound by the CAP Theorem[14].

1.3 What are the currently existing DHTs?

Chord [56] and Kademlia [42] are the most commonly used DHTs in practice. Chord has been favored by researchers because it was designed with a series of proofs to show its consistency in the face of churn. These proofs have been shown to be incorrect[63] without modification

to the established protocol.

1.3.1 Chord

Chord is a peer-to-peer (P2P) protocol for file sharing and distributed storage that guarantees a high probability $\log_2 N$ lookup time for a particular node or file in a network with N nodes. It is highly fault-tolerant to node failures and churn, the constant joining and leaving of nodes. It scales extremely well and the network requires little maintenance to handle individual nodes. Files in the network are distributed evenly among its members.

As a Distributed Hash Table (DHT), each member of the network and the data stored on the network is mapped to a unique m-bit key or ID, corresponding to one of 2^m locations on a ring. The ID of a node and the node itself are referred to interchangeably.

In a traditional Chord network, all messages travel in one direction- upstream hopping from one node to another with a greater ID until it wraps around. A node in the network is responsible for all the data with keys *above or upstream* his predecessor, up through and including its own ID. If a node is responsible for some key, it is referred to as being the successor of that key.

Robustness in the network is accomplished by having nodes backup their contents to their s (often 1) immediate successors, the closest nodes upstream. This is done because when a node leaves the network or fails, the most immediate successor would be responsible for its content.

Each node maintains a table of m shortcuts to other peers, as shown in Figure 1.1, 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 $n + 2^{i-1} \mod 2^m$. Nodes route messages to the finger that is closest to the sought key without going past it, until it is received by the responsible node. This provides Chord with a highly scalable $\log_2(N)$ lookup time for any key[56].

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. Full details on Chord's

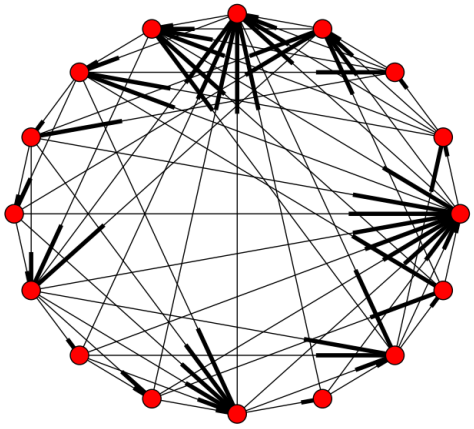


Figure 1.1: A Chord ring 16 nodes where $m = 4$. The bold lines are incoming edges. Each node has a connection to its successor, as well as 4 fingers, some of which are duplicates.

maintenance cycle are beyond the scope of this paper and can be found here[56].

1.3.2 Kademlia

Kademlia is currently the most popular DHT methodology. It powers the trackerless bittorent mainline DHT, and the C implementation related to that project is likely the greatest cause of its popularity. Many other distributed systems utilize modified versions of Kademlia as a means of peer management and as a key-value store.

Kademlia is built in a non-Euclidian metric space. Locations are represented by a large integer (160 bit is most common) and the distance between locations is calculated by the XOR metric. This means Kademlia's metric space is a generalization of a binary tree, where the locations are mapped to leaf nodes and distance between nodes is the distance required to traverse between them on that tree.

Because of the geometric awkwardness of its metric, Kademlia uses a modified k-nearest neighbors approach to approximate a node's Voronoi regions and Delaunay peers. If nodes are evenly distributed through the space, Kademlia's metric provides an $O(log(n))$ diameter network.

1.4 What are DHTs used for

DHTs are designed to be used to store data in a distributed system that would normally be centrally stored in other systems, like a database or other records. In practice, they also double as a mechanism for peers discovery and network management. Many P2P services use a DHT as part of their infrastructure:

- Bittorrent [30] uses a DHT to store tracking information, which allows users to download a file using an identifier hash.
- CJDNS [25] uses the Chord DHT to provide an efficient routing structure for packets.
- I2P [62] uses a DHT as a shared data store for many P2P applications.
- IPFS [9] uses a DHT to store Merkel tree hashes of shared files and peering information similar to Bittorrent. FreeNet [17] uses a DHT as a global datastore similarly to I2P.

1.5 Dissertation Roadmap

This dissertation tracks my exploration though appreciations of and improvements to Distributed Hash Tables.

Chapter 2 describes ChordReduce [48] which demonstrates the application of a Chord DHT to organizing a large scale distributed computation in the MapReduce [21] paradigm. This exploration lead to considerations of how to generalize the concept of DHTs. Where there were 3 distinct DHT mechanisms (Chord [56], Kademlia [42], and CAN [46]) I was interested in find a more generic model of DHT behavior.

In search of a tool to allow the abstraction of DHT behavior I developed DGVH [10] (described in Chapter 3) which permitted me to easily build a DHT protocol to operate using any system of coordinates and distance function that I wished.

In order to build practical systems for testing, we needed a software system that could easily be modified to implement new DHT behaviors. UrDHT [49] is an open source software

project documented in Chapter 4, which allows us to rapidly prototype and test new DHT methods of organization.

Intended for use with UrDHT, Chapters 5 and 6 discuss mutually exclusive optimizations of DHT behavior. Chapter 6 discusses a mechanism for reactively storing copies of data such that it is highly unlikely data will ever be lost.

Chapter 7 discusses a method of assigning participants to specific points in a DHT such that the latency between peers and across the network is minimized

Chapter 2

MapReduce on a Chord Distributed Hash Table

Google's MapReduce [21] paradigm has rapidly become an integral part in the world of data processing and is capable of efficiently executing numerous Big Data programming and data-reduction tasks. By using MapReduce, a user can take a large problem, split it into small, equivalent tasks and send those tasks to other processors for computation. The results are sent back to the user and combined into one answer. MapReduce has proven to be an extremely powerful and versatile tool, providing the framework for using distributed computing to solve a wide variety of problems, such as distributed sorting and creating an inverted index [21].

Popular platforms for MapReduce, such as Hadoop [1], are explicitly designed to be used in large datacenters [2] and the majority of research has been focused there. These MapReduce platforms are highly centralized and tend to have single points of failure[54] as a result. A centralized design assumes that the network is relatively unchanging and does not usually 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. Finally deploying these systems and developing programs for them has an extremely steep learning curve.

We were motivated to start exploring a more abstract deployment for MapReduce, one that could be deployed in a much wider variety of contexts, from peer-to-peer frameworks to datacenters. Our framework cannot and does not rely on many common assumptions, such

Previously published at ICA Conference 2014

as a dedicated and static network of homogeneous machines. Nor do we assume that failed nodes will recover [2]. Finally, our framework needed to be easy to deploy and simple to use.

We used Chord[56], a peer-to-peer distributed hash table, as the backbone for developing our system. This chapter presents these contributions:

- We define the architecture and components of ChordReduce and demonstrate how they fit together to perform MapReduce jobs over a distributed system without the need of a central scheduler or coordinator, avoiding a central point of failure. We also demonstrate how to create programs to solve MapReduce problems using ChordReduce (Section 2.3).

- We built a prototype system implementing ChordReduce and deployed it on Amazon's Elastic Cloud Compute. We tested our deployment by solving Monte-Carlo computations and word frequency counts on our network (Section 2.4).

- We prove that ChordReduce is scalable and highly fault tolerant, even under high levels of churn and can even benefit from churn under certain circumstances. Specifically, we show it is robust enough to reassign work during runtime in response to nodes entering and leaving the network (Section 2.5).

- We contrast ChordReduce with similar architectures and identify future areas of fruitful research using ChordReduce (Sections 2.6 and 2.7).

2.1 Background

ChordReduce takes its name from the two components it is built upon. Chord[56] provides the backbone for the network and the file system, providing scalable routing, distributed storage, and fault-tolerance. MapReduce runs on top of the Chord network and utilizes the underlying features of the distributed hash table. This section provides background on Chord and MapReduce.

2.1.1 MapReduce

At its core, MapReduce [21] 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 key 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 counting the occurrence of each word in a collection of documents, called WordCount. 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 [1]. Hadoop is an open-source Java implementation developed by Apache and Yahoo! [45]. Hadoop has two components, the Hadoop Distributed File System (HDFS) [13] and the Hadoop MapReduce Framework [34]. 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 [54] in the network, as well as a potential bottleneck for the system [55].

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.

2.2 ChordReduce

Popular platforms for MapReduce, such as Hadoop, are extremely powerful, but have some inherent limitations. These platforms are designed to be deployed in a data center. Their architecture relies on multiple nodes with specific roles to coordinate the work, such as the NameNode and JobTracker. These nodes perform necessary scheduling and distribution tasks and help provide fault-tolerance to the network as a whole, but in doing so become single points of failure themselves.

ChordReduce is designed as a more abstract framework for MapReduce, able to run on any arbitrary distributed configuration. ChordReduce leverages the features of distributed hash tables to handle distributed file storage, fault tolerance, and lookup. We designed ChordReduce to ensure that no single node is a point of failure and that there is no need for any node to coordinate the efforts of other nodes during processing.

Our central design philosophy was to implement additions to the Chord protocol by leveraging the existing features of Chord. By treating each task or target computation as an object of data, we can distribute them in the same manner as files and rely on the protocol to route them and provide robustness.

We have 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 (Figure 2.1). 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.

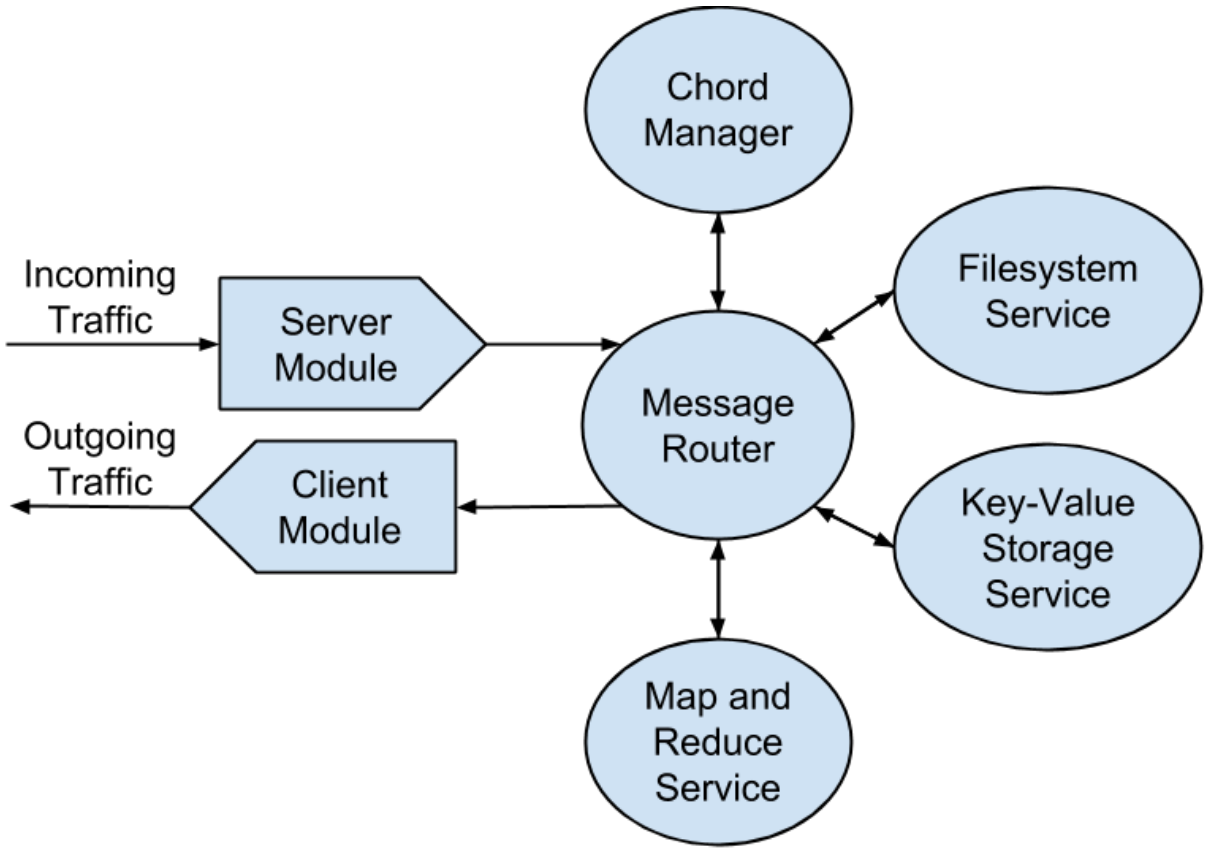


Figure 2.1: The basic architecture of a node in ChordReduce. MapReduce runs as a service on top of each node.

Marozzo et al. [41] 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.

2.2.1 File Storage

The design of a distributed file system is closely tied to the design of the accompanying implementation of MapReduce [23] [13]. Our system uses CFS [20], short for Cooperative File System, to store files. Everything in Chord, be it data or a node, is given a hash identifier or key. The ID of a node is the hash of their IP address and port, while the key for a file is the hash of its filename. In the initial version of Chord, the entire file would be stored in the node with the ID equal or closest upstream to the file's key.

Dabek et. al found that by splitting the file into blocks and storing each block in a different node greatly improved the system's load balancing when compared to storing the entire file on a single node[20]. ChordReduce implements the same system. Files are split into approximately equally sized blocks. Each block is treated as an individual file and is assigned a key equal to the hash of its contents. The block is then stored at the node responsible for that key.

The node which would normally be responsible for the whole file instead stores a *keyfile*. The keyfile is an ordered list of the keys corresponding to blocks containing portions of the file and is created as the blocks are assigned their respective keys. When the user wants to retrieve a file, they first obtain the keyfile and then request each block specified in the keyfile.

2.2.2 Decentralized MapReduce and Data Flow

In ChordReduce's 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 acts as both a client and a server. To start a job, the user contacts a node at a specified hash address and provides it with the tasks. 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 the stager is to divide the work into *data atoms*, which define the smallest individual units that work can be done on. This might represent a block of text, 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 also contain the Map and Reduce functions defined by the user.

If the user wants to perform a MapReduce job over data on the network, the stager locates the keyfile for the data and creates a data atom for each block in the file. Each data atom is then sent to the node responsible for their corresponding block. When the data atom reaches its destination node, that node retrieves the necessary data and applies the Map

function. The results are stored in a new data atom, 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.

MapReduce jobs don't rely on a file stored on the network, such as a Monte-Carlo approximation, to create data atoms specified by the user in the 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. From there, the execution is identical to the above scenario.

Once all the Reduce tasks 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.

2.2.3 Fault-Tolerance

Due to the potentially volatile nature of a peer-to-peer network, ChordReduce has to be able to handle an arbitrary amount of churn. When a node fails or leaves Chord, the failed node's successor will become responsible for all of the failed node's keys. As a result, each node in the ChordReduce network relies on their successor to act as a backup.

To prevent data from becoming irretrievable, each 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. If the node fails, the successor is able to take his place almost immediately. This scheme is used to not only backup files, but the Map and Reduce tasks and data atoms as well.

This procedure prevents any single node failure or sequences of failures from harming the network. Only the failure of multiple neighboring nodes poses a threat to the network's integrity. Furthermore, since a node's ID in the network does not map to a geographical location, any failure that affects multiple nodes simultaneously will be spread uniformly throughout, rather than hitting successive nodes. This means if successive nodes do fail simultaneously, they did so independently.

This concept can be extended to provide additional robustness. Suppose that each node

has failure rate $r < 1$ and that the each node backs up their data with s successive nodes downstream. If one of these nodes fails, the next successive node takes its place and the next upstream node becomes another backup. This ensures there will always be s backups. The integrity of the ring would only be jeopardized if $s + 1$ successive nodes failed almost simultaneously, before the maintenance cycle would have a chance to correct for the failed nodes. The chances of such an event would be r^{s+1}, as each failure would be independent.

A final consequence of this is load-balancing during runtime. As new nodes enter the network, they change their successor as the maintenance cycle guides them into the correct location in the ring. 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 maintains this data as a backup for n. Because Map tasks are backed up in the same manner as data, a node can take the data and corresponding tasks he's responsible for and begin performing Map tasks immediately.

2.3 Experiments

In order for ChordReduce to be a viable framework, we must 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

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

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. 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.

2.3.1 Experimental Deployment

We built a fully functional implementation of ChordReduce in Python. Our implementation implements all the routing and maintenance procedures defined by Chord[56], the file storage capabilities of CFS [20], and a MapReduce service built on top of the system. We ran our experiments using Amazon's Elastic Compute Cloud (EC2) service. Amazon EC2 allows users to purchase an arbitrary number of virtual machines and pay for the machines by the hour. Each node was an individual EC2 small instance [3] with a Ubuntu 12.04 image preconfigured with Git and a small startup script which retrieves the latest version of the code.

We can choose any arbitrary node as the stager and tell it to run a 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. However, this process was extremely tedious and time consuming.

We created an additional node to help configure the experiment, which we call the 'in-

strumentation node'. We avoid calling it the 'manager' or 'coordinator' as we don't want to create the false impression that the instrumentation node actively participates in the experiments or is a member of the Chord ring. The instrumentation node's role is to aid in the generation and collection of data.

First, the instrumentation node gives us an easy way to change experimental variables in between runs without having to manually reset each node. These variables range from the job size to defining the specific job. The instrumentation node also is responsible for managing churn. The instrumentation node keeps a list of all "active" and "failed" nodes in the network and decides when each active node fails (and abruptly drops out of the network) and when each failed node should join the ring. Finally, each node collects data on its individual CPU utilization and bandwidth usage and sends this information to the instrumentation node as part of the experiment.

2.3.2 Experiment Configuration

We tested our framework by running two different MapReduce jobs: a Monte-Carlo approximation of *pi* and a word frequency count. Both jobs were tested under multiple network configurations; we varied the initial size of the network[1], the size of the job, and the rate of churn.

Our Monte-Carlo approximation of π is largely analogous to having a square with the top-right quarter of a circle going through it (Figure 2.2), 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. [2]

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 to collect,

[1]The network size changes throughout due to churn, but is unlikely to drastically vary, as the chances of joins and failures are equal.

[2]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.

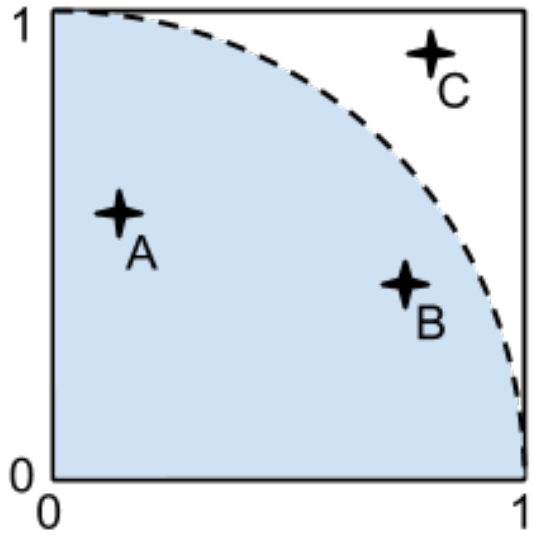


Figure 2.2: The "dartboard." The computer throws a dart by choosing a random x and y between 0 and 1. If $x^2 + y^2 < 1^2$, the dart landed inside the circle. A and B are darts that landed inside the circle, while C did not.

we could double the amount of work each node gets without having to store new files on the network. 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.

Our word frequency experiment counts the occurrence of each word in a corpus stored on the Chord network using CFS [20]. We also varied the block size used for CFS to see what effect that had on computation.

To perform a word frequency count, the stager obtains the keyfile for the desired corpus and creates a data atom containing the map and reduce functions for each key listed in the keyfile. Each node receives a data atom for each block they are responsible for and create a word frequency count for each of their specified blocks. Those results are reduced by simply combining the word frequency tables.

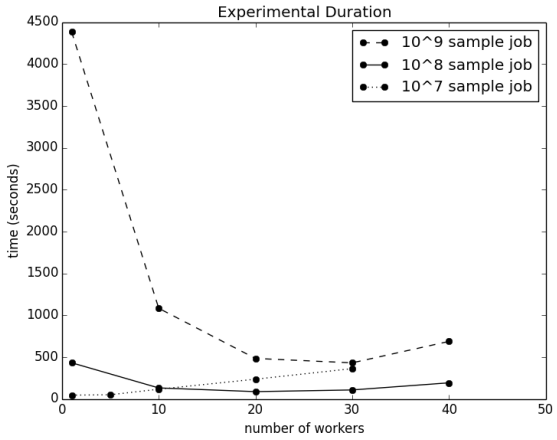


Figure 2.3: For a sufficiently large job, it was almost always preferable to distribute it. When the job is too small, such as with the 10^7 data set, our runtime is dominated by the overhead. Our results are what we would expect when overhead grows logarithmically to the number of workers.

2.4 Results

Figure 2.3 and Figure 2.4 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 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. 2.4 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

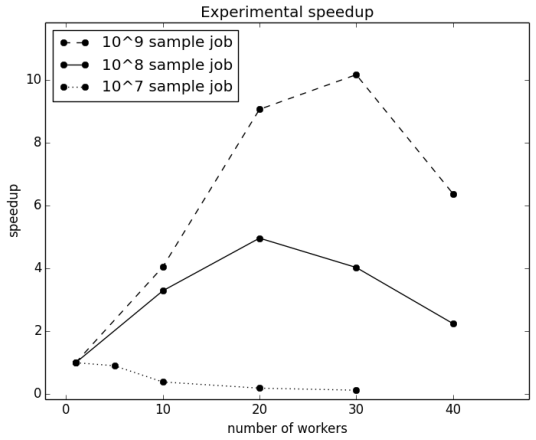


Figure 2.4: The larger the size of the job, the greater the gains of distributing with ChordReduce. In addition, the larger the job, the more workers can be added before we start seeing diminishing returns. This demonstrates that ChordReduce is scalable.

of distributing the work using ChordReduce.

The 10^7 sample set's run time grows with logarithmic behavior. 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. 2.5 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. 2.6 further emphasizes this. Note that as the jobs become larger, the expected speedup from ChordReduce approaches linear behavior.

Table 2.1 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

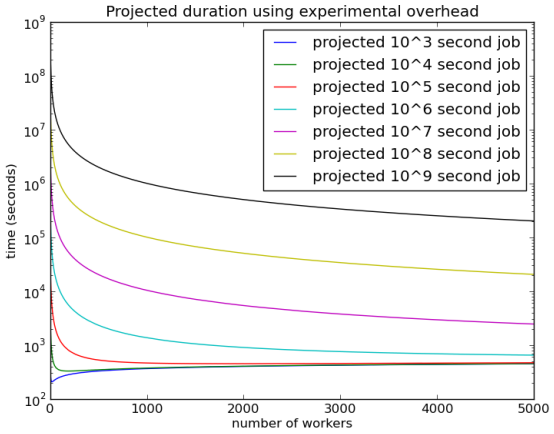


Figure 2.5: The projected runtime using ChordReduce for differently sized jobs. Each curve projects the expected behavior for job that takes a single worker the specified amount of time.

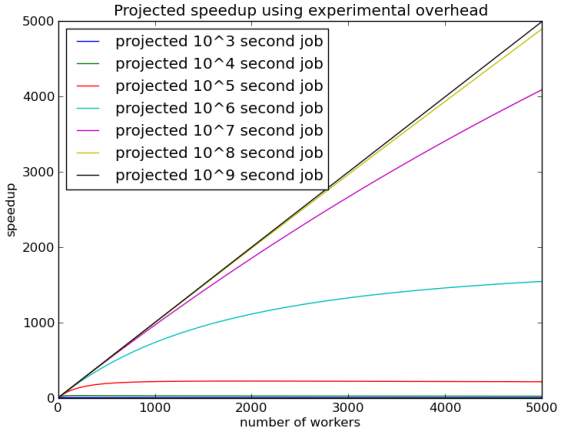


Figure 2.6: The projected speedup for different sized jobs.

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 2.1: Runtime and Speedup versus Churn Rate

generated 10^8 samples while experiencing different rates of churn, as specified in Table 2.1. 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 [41] [52].

Our testing rates for churn are an order of magnitude higher than the rates used in the P2P-MapReduce simulation [41]. 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.

2.5 Related Work

Marozzo et al. [41] 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 [24] 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 [41].

Lee et al.'s work [35] 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 [38], 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.

Both of these papers have promising results and confirm the capability of our own framework and both solely examine P2P networks for the purpose of routing data and organizing the network. ChordReduce uses Chord as a means of efficiently distributing responsibility throughout the network and uses its existing features to add robustness to nodes working on

Map and Reduce tasks, in addition to the routing and organizing capabilities. Our framework was successfully deployed and tested, operating under high rates of churn without a centralized source for organization.

2.6 Conclusion and Future Work

We presented ChordReduce, a framework for MapReduce that is completely decentralized, scalable, load balancing, and highly tolerant to churn and node failure at any point in the network. We implemented a fully functional version of ChordReduce and performed detailed experiments to test its performance. These experiments confirmed that ChordReduce is robust and effective. ChordReduce is based on Chord, which is traditionally viewed as a P2P framework for distributing and sharing files. Instead, we demonstrated that it can also be used as a platform for distributed computation. Chord provides $\log_2 n$ connectivity throughout network and has built-in mechanisms for handling backup, automatically assigning responsibility, routing, and load balancing.

Using Chord as the middleware for ChordReduce establishes its effectiveness for distributed and concurrent computation. The effectiveness of Chord opens up new approaches for tackling other distributed problems, such as supporting databases and machine learning for Big Data, and exascale computations. We intend to further optimize the performance of ChordReduce and extend the middleware to other applications.

Chapter 3

A Distributed Greedy Heuristic for Computing Voronoi Tessellations With Applications Towards Peer-to-Peer Networks

Voronoi diagrams [5] have been used in distributed and peer-to-peer (P2P) applications for some time. They have a wide variety of applications. Voronoi diagrams can be used as to manage distributed hash tables [60], or in coverage detection for wireless networks [15]. Additionally, Massively Multiplayer Online games (MMOs) can use them to distribute game states and events between players at a large scale [27] [26] [6].

Computing the Voronoi tessellation along with its coprime problem, Delaunay Triangulation, is a well-analyzed problem. There are many algorithms to efficiently compute a Voronoi tessellation given all the points on a plane, such as Fortune's sweep line algorithm [22]. However, many network applications are distributed and many of the algorithms to compute Voronoi tessellations are unsuited to a distributed environment.

In addition, complications occur when points are located in spaces with more than two dimensions. Computing the Voronoi tessellation of n points in a space with d dimensions takes $O(n^{\frac{2d-1}{d}})$ time [61]. Distributed computations often have to resort to costly Monte-Carlo calculations [8] in order to handle more than two dimensions.

Rather than exactly solving the Voronoi tessellation, we instead present a fast and accurate heuristic to approximate each of the regions of a Voronoi tessellation. This enables fast and efficient formation of P2P networks, where nodes are the Voronoi generators of those

Previously published at DPDNS Workshop IPDPS 2015

region. A P2P network built using this heuristic would be able to take advantage of its available fault-tolerant architecture to route along any inaccuracies that arise. This chapter presents the following contributions:

- We present our Distributed Greedy Voronoi Heuristic (DGVH). The DGVH is a fast, distributed, and highly accurate method, whereby nodes calculate their individual regions described by a Voronoi tessellation using the positions of nearby nodes. DGVH can work in an arbitrary number of dimensions and can handle non-Euclidean distance metrics. Our heuristic can also handle toroidal spaces. In addition, DGVH can accommodate the calculation of nodes moving their positions and adjust their region accordingly, while still maintaining a high degree of accuracy (Section 3.1). Even where small inaccuracies exist, DVGH will create a fully connected graph.

- We discuss how P2P networks and distributed applications can use DGVH (Section 3.2). In particular, we show how we can use DGVH to build a distributed hash table with embedded minimal latency.

- We present simulations demonstrating DGVH's efficacy in quickly converging to the correct Voronoi tessellation. We simulated our heuristic in networks ranging from size 500 nodes to 10000 nodes. Our simulations show that a distributed network running DGVH accurately determines the region a randomly chosen point falls in 90% of the time within 20 cycles and converges with near 100% accuracy by cycle 30 (Section 3.3).

- We present the related work we have built upon to create our heuristic and what improvements we made with DGVH (Section 3.4).

3.1 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

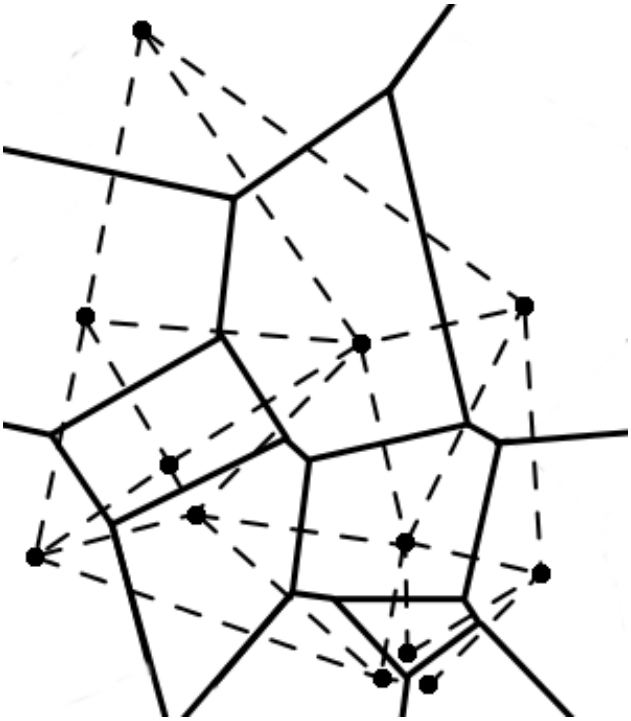


Figure 3.1: An example Voronoi diagram for objects on a 2-dimensional space. The black lines correspond to the borders of the Voronoi region, while the dashed lines correspond to the edges of the Delaunay Triangulation.

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 of a Voronoi diagram is shown in Figure 3.1. For additional information, Aurenhammer [5] provides a formal and extremely thorough description of Voronoi tessellations, as well as their applications.

3.1.1 Our Heuristic

The Distributed Greedy Voronoi Heuristic (DGVH) is a fast method for nodes a P2P network to define their individual Voronoi region (Figure 6.1). If each node is assigned a location, DGVH builds an approximation of the nodes it is required to connect to such that it can

Figure 3.2: Distributed Greedy Voronoi Heuristic

```
1: Given node n and its list of candidates.
2: Given the minimum table_size
3: short_peers ← empty set that will contain n's one-hop peers
4: long_peers ← empty set that will contain n's two-hop peers
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 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: if |long_peers| > table_size^2 then
21:    long_peers ← random subset of long_peers of size table_size^2
22: end if
```

accurately discern it's Voronoi region. This is done by selecting the nearby nodes that would 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 [8]. 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 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.

This heuristic has the benefit of being fast and scalable into any geometric space where a distance function and midpoint can be defined. The distance metric used for this chapter is the minimum distance in a multidimensional unit toroidal space. Where $\vec{a}$ and $\vec{b}$ are locations in a d-dimensional unit toroidal space:

$$distance = \sqrt{\sum_{i \in d} (\min(|a_i - b_i|, 1 - |a_i - b_i|))^2}$$

Whether or not distance corresponds to actual physical distance or some virtual distance on an overlay depends on the application.

Our heuristic can be overaggressive in removing candidate nodes. For example, if a node is located between two other nodes, such that their midpoint does not fall upon the shared face of their Voronoi regions, then this heuristic will not link the blocked peers. This is demonstrated in Figure 3.3. Our algorithm handles these cases via our method of peer management (Section 3.1.2).

3.1.2 Peer Management

Nodes running the heuristic maintain two peer lists: *Short Peers* and *Long Peers*. This is done to mitigate the error induced by DGVH and provide robustness against churn[1] in a

[1]The disruption caused to an overlay network by the continuous joining, leaving, and failing of nodes.

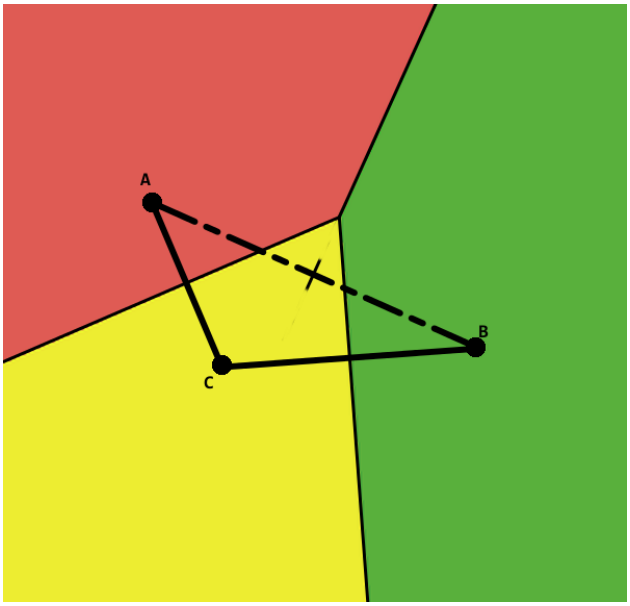


Figure 3.3: The edge between A and B is not detected by DGVH, as node C is closer to the midpoint than B is. This is mitigated by peer management polices.

distributed system.

Short Peers are the set of peers DGVH judged to have Voronoi regions adjacent to the node's own. Using a lower bound on the length of *Short Peers* corrects for errors in the approximation as it forces nodes to include peers that would otherwise be omitted. Previous work by Beaumont *et al.* [8] has found a useful lower bound on short peers to be $3d + 1$. Should the number of short peers generated by DGVH be less than the lower bound, the nearest peers not already included in *Short Peers* are added to it, until *Short Peers* is of sufficient size.

There is no upper bound to the number of short peers a node can have. This means in contrived cases, such as a single node surrounded by other nodes forming a hypersphere, this number can grow quite high. Bern *et al.* [11] found that the expected maximum degree of a vertex in a Delaunay Triangulation is

$$\Theta(\frac{\log n}{\log \log n})$$

where n is the number of nodes in the Delaunay Triangulation. This bound applies to a

Delaunay Triangulation in any number of dimensions. Thus, the maximum expected size of *Short Peers* is bounded by $\Theta(\frac{\log n}{\log \log n})$, which is a highly desirable number in many distributed systems [56] [42].

Long Peers is the list of two-hop neighbors of the node. When a node learns about potential neighbors, but are not included in the short peer list, they may be included in the long peer list. *Long Peers* has a maximum size of $(3d+1)^2$, although this size can be tweaked to the user's needs. For example, if *Short Peers* has a minimum size of 8, then *Long Peers* has a maximum of 64 entries. We recommend that members of *Long Peers* are not actively probed during maintenance to minimize the cost of maintenance. A maximum size is necessary, as leaving it unbounded would result in a node eventually keeping track of all the nodes in the network, which would be counter to the design of a distributed and scalable system.

How nodes learn about peers is up to the application. We experimented using a gossip protocol, whereby a node selects peers from *Short Peers* at random to "gossip" with. When two nodes gossip with each other, they exchange their *Short Peers* with each other. The node combines the lists of short peers[2] and uses DGVH to determine which of these candidates correspond to its neighbors along the Delaunay Triangulation. The candidates determined not to be short peers become long peers. If the resulting number of long peers exceeds the maximum size of *Long Peers*, a random subset of the maximum size is kept.

The formal algorithm for this process is described in Figure 3.4. This maintenance through gossip process is very similar to the gossip protocol used in Beaumont et al.'s RayNet [8].

3.1.3 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,

[2]Nodes remove themselves and repetitions from the candidates they receive.

Figure 3.4: Gossiping

1: Node n initiates the gossip.
2: $neighbor \leftarrow$ random node from $n.short\_peers$
3: $n\_candidates \leftarrow n.short\_peers \cup n.long\_peers \cup neighbor.short\_peers$
4: $neighbor\_candidates \leftarrow neighbor.short\_peers \cup neighbor.long\_peers \cup n.short\_peers$.
5: n and $neighbor$ each run Distributed Greedy Voronoi Heuristic using their respective $candidates$

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 $\Theta(\frac{\log N}{\log \log N})$ [11] (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 $\Omega(1)$.

Previous work [8] claims constant time approximation. The reality is that RayNet's leading constant is in the order of thousands. Our algorithm has a greater asymptotic worst case cost, but for all current realistic network sizes it will be more time efficient then RayNet's approximation.

3.2 Applications

As we previously discussed in Section 3, Voronoi tessellations have many applications for distributed systems [15] [27] [26] [6]. We focus our discussion on the two extremes of applications: DHTs, which work with overlay networks, and wireless networks, which need to

Figure 3.5: Lookup in a Voronoi-based DHT

1: Given node n
2: Given m is a message addressed for loc
3: $potential\_dests \leftarrow n \cup n.short\_peers \cup n.long\_peers$
4: $c \leftarrow$ node in $potential\_dests$ with shortest distance to loc
5: **if** $c == n$ **then return** n
6: **elsereturn** $c.lookup(loc)$
7: **end if**

take literal physical constraints into account.

3.2.1 Distributed Hash Tables and Voronoi tessellation

Arguably all Distributed Hash Tables (DHTs) are built on the concept of a Voronoi tessellation. 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 [56] [42] [46] [50], but other metrics such as geographic location can be used as well [47].

These DHTs have carefully chosen metric spaces such that these regions are very simple to calculate. For example, Chord [56] and similar ring-based DHTs [39] 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 3.5) is extremely similar to the one used in Ratnasamy et al.'s Content Addressable Network (CAN) [46], which would be $O(n^{\frac{1}{d}})$ hops.

DGVH can be used in a DHT to quickly and scalably construct both the Voronoi tessel-

lation and links to peers. In addition, gossip-based peer management policies are extremely efficient in proactively handling node joins and failures. The act of joining informs other nodes of the joiner's existence. This can be done by the joiner contacting each peer in the peer lists of the node previously responsible for the joiner's location.

Node failures can be handled without much effort. When a node attempts to route to or gossip with a node and discovers it no longer exists, it should remove the node from its peer lists and inform all of the nodes it knows to do the same. Since routing is how a node would make a decision on whether a point belongs to a particular Voronoi region, failed nodes don't have any impact on the network's accuracy.

Because the algorithm is defined in terms of midpoint and distance functions, it is not bound to any particular topology or metric space. Our heuristic can be used to create a DHT that uses any arbitrary coordinate system which defines a midpoint and distance definition.

For example, we could use latency as one of these metrics by using it to approximate node locations in the network. This would allow messages to be routed along minimum latency paths, rather than along minimal hop paths. We intend to model this using a distributed spring model.

3.2.2 Wireless Coverage

Cărbunar et al. [15] demonstrated how Voronoi tessellations could be used to solve the *coverage-boundary* problem in wireless ad-hoc networks. The coverage-boundary problem asks which nodes are on the physical edge of the network. This knowledge provides useful information to networks. For example, ad-hoc networks operating in no infrastructure or have been set up temporarily can use this knowledge to define the reach of the network's coverage.

The authors showed how a Voronoi tessellation using the nodes as Voronoi generators could solve the coverage-boundary problem. They proved that the node's Voronoi region was not completely covered by the node's sensing radius if and only if the node was on the

network's boundary [15].

Chen et al. [16] extended this property for sleep scheduling in wireless sensor networks. A node is allowed to go to sleep and conserve power so long as the area it is detecting can be covered by other nodes. Using Voronoi tessellations, a node knows it can conserve power and not affect the network if and only if it is not on the boundary of the network and its Voronoi vertices[3] are covered by other nodes.

Cărbunar et al.'s algorithm relies on a centralized computation for Voronoi tessellation, as the distributed computation they examined only relied on forming the Delaunay triangulation between nodes within a certain radius, nor could it handle moving nodes. DGVH is not limited to handling nodes within a specified radius, since the peer management spreads information about nodes by gossiping. In addition, so long as a node moving to a new location is treated as an entirely new node by the rest of the network, DGVH can handle moving nodes.[4]

3.3 Experiments

We implemented two sets of experiments for DGVH. The first compares the Voronoi tessellations created by DGVH to an actual Voronoi tessellation. Our second set of experiments demonstrates that any errors computed by DGVH are negligible when building distributed and fault-tolerant systems.

3.3.1 Experiment 1: Voronoi Accuracy

To test the accuracy of our heuristic, we have generated the graphs produced DGVH and a Delaunay triangulation generated by the Triangle [53] library. We test in 2-dimensional Euclidean space and measured the number of edges in the graph generated by DGVH that

[3]Voronoi vertices are the points at which the edges of 3 or more Voronoi cells converge.

[4]A specific node and a specific location must be bound together into a single identity. This means when a node tries to route to a node that has moved using the node's previous location, it should fail, as though that specific node no longer exists. Failure to do this would cause a node to incorrectly determines what falls within its Voronoi region.

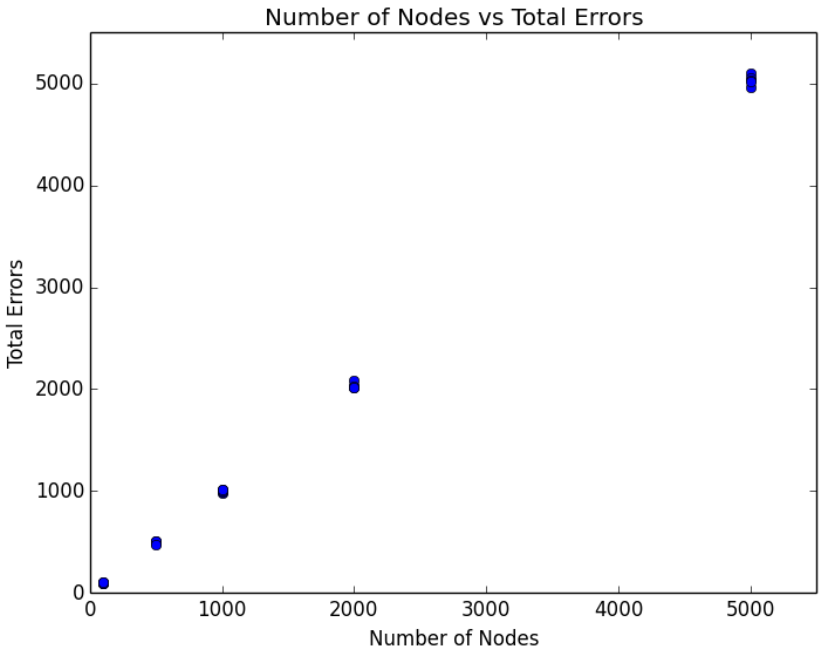


Figure 3.6: As the size of the graph increases, we see approximately 1 error per node.

differ from the graph generated by a Delaunay Triangulation. We test networks of 100, 500, 1000, 2000, and 5000 nodes and found that the graphs by DGVH differed from the graph generated Delaunay triangulation by approximately 1 edge per node. Our results are summarized in Figure 3.6.

3.3.2 Experiment 2: P2P Convergence and Routing

Our second set of experiments examines how DGVH could be used to create a DHT and how well it would perform in this task. Our simulation demonstrates how DGVH can be used to create a stable overlay from a chaotic starting topology after a sufficient number of gossip cycles. We do this by showing that the rate of successful lookups approaches 1.0. We compare these results to RayNet [8], which proposed that a random k-connected graph would be a good, challenging starting configuration for demonstrating convergence of a DHT to a stable network topology.

During the first two cycles of the simulation, each node bootstraps its short peer list by

Figure 3.7: 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
```

appending 10 nodes, selected uniformly at random from the entire network. In each cycle, the nodes gossip (Figure 3.4) and run DGVH using the new information. We then calculate the hit rate of successful lookups by simulating 2000 lookups from random nodes to random locations, as described in Figure 3.7. A lookup is successful when the network correctly determines which Voronoi region contains a randomly selected point.

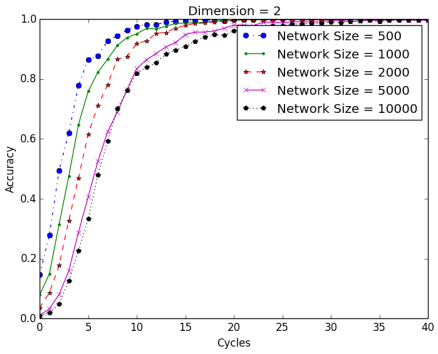
Our experimental variables for this simulation were the number of nodes in the DGVH generated overlay and the number of dimensions. We tested network sizes of 500, 1000, 2000, 5000, and 10000 nodes each in 2, 3, 4, and 5 dimensions. The hit rate at each cycle is $\frac{hits}{2000}$, where *hits* are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

Our results are shown in Figures 3.8a, 3.8b, 3.8c, and 3.8d for each dimension. Our graphs show that the created overlay quickly constructs itself from a random configuration and that our hit rate reached 90% by cycle 20, regardless of dimension. 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 [8]. As the network size and number of dimensions each increase, convergence slows.

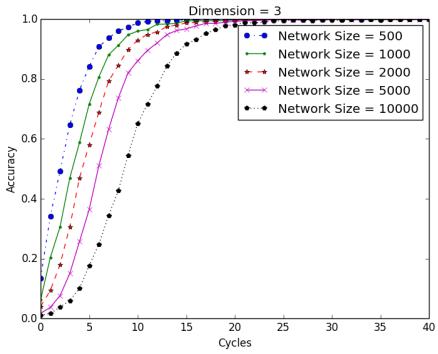
3.4 Related Work

While there has been previous work on applying Voronoi regions to DHTs and peer-to-peer (P2P) applications, we have found no prior work on how to perform embedding of an inter-node latency graph.

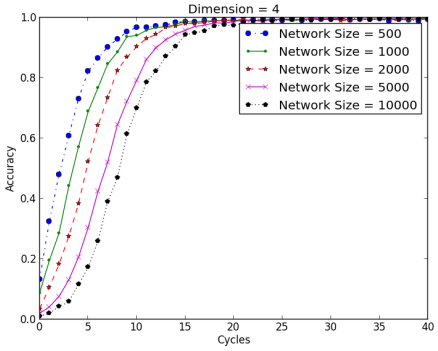
Backhaus et al.'s VAST [6] is a Voronoi-based P2P protocol designed for handling event messages in a massively multiplayer online video game. Each node finds its neighbors by con-



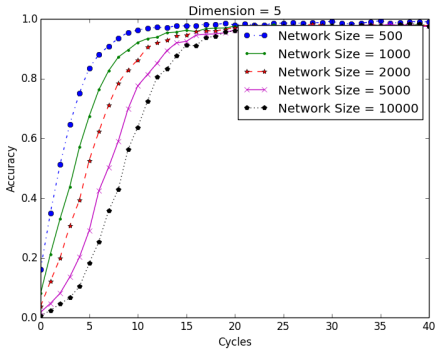
(a) This plot shows the accuracy rate of lookups on a 2-dimensional network as it self-organizes.



(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 3.8: 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.

structing a Voronoi diagram using Fortune's sweepline algorithm [22]. VAST demonstrated that Voronoi diagrams could be used as the backbone to large-scale applications, although their work focused specifically on using 2-dimensional Voronoi diagrams.

The two DHT protocols developed by Beumont et al., VoroNet [7] and RayNet [8] are two-dimension DHTs that we intend to extend using our technique. VoroNet is based off Kleinberg's small world model [32] and achieves polylogarithmic lookup time. Each node in VoroNet solves its Voronoi region to determine its neighbors and also maintains a link to a randomly chosen distant node. VoroNet focused specifically on the two-dimensional Voronoi computations and the techniques used would be too expensive in higher dimensions and were not resilient to churn [8].

RayNet [8] was based on the work done on VoroNet and used a heuristic to calculate Voronoi tessilations. Like our DGVH, RayNet's heuristic does not solve for Voronoi regions, as that is prohibitively expensive. RayNet uses a Monte-Carlo method to approximate the volume of a node's Voronoi region in constant time. While effective at estimating the Voronoi region, the volume-based Monte-Carlo approximation is expensive and requires multiple samples. This gives the runtime of RayNet's heuristic an enormous leading constant. RayNet does mention the idea of mapping attributes to each axis, but how this can be exploited is left as future work.

3.5 Conclusion and Future Work

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 DGVH'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 next step is to create a formal protocol and implementation for a Voronoi tessellation-based distributed hash table using DGVH. We can use this DHT to choose certain metrics we want to measure, such as latency, or trust, and embed that information as part of a node's identity. By creating an appropriate distance measurement, we can route along some path that minimizes or maximizes the desired metric. Rather than create an overlay that minimizes hops, we can have our overlay minimize latency, which is the actual goal of most routing algorithms.

Chapter 4

UrDHT: A Unified Model for Distributed Hash Tables

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

Distributed Hash Tables have been the catalyst for the creation of many P2P applications. Among these are Redis [51], Freenet [17], and, most notably, BitTorrent [18]. 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 [10]. We found that all DHTs can cleanly be mapped to the primal-dual problems of Voronoi Tessellation and Delaunay Triangulation.

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

The end result is not only do we have an abstract model of DHTs, we have a simple

Under consideration by ICPP 2016 at time of writing

framework that developers can use to quickly create new distributed applications. This simple framework allows generation of internally consistent implementations of different DHTs that can have their performance rigorously compared.

To summarize our contributions:

- We give a formal specification for what needs to be defined in order to create a functioning DHT. While there has long existed a well known protocol shared by distributed hash tables, this defines what a DHT does. It does not describe what a DHT is.

  We show that DHTs cleanly map to the primal-dual problem of Delaunay Triangulation and Voronoi Tessellation. We list a set of simple functions that, once defined, allow our Distributed Greedy Voronoi Heuristic (DGVH) to be run in any space, creating a DHT overlay for that space (Section 4.1).

- We present UrDHT as an abstract DHT and show how a developer would modify the functions we defined to create an arbitrary new DHT topology (Section 4.2).

- We show how to reproduce the topology of Chord and Kademlia using UrDHT. We also implement a DHT in Euclidean geometry and a hyperbolic geometry represented by a Poincarè disc (Section 4.3).

- We conduct experiments that show building DHTs using UrDHT produced efficiently routable networks, regardless of the underlying geometry(Section 4.4).

- We present some efforts and projects that are similar to our own (Section 4.5).

- We discuss the ramifications of our work and what future work is available (Section 4.6).

4.1 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.

4.1.1 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 [32]. A DHT can still function without long peers.

Interestingly, despite the diversity of DHT topologies and how each DHT organizes short

[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.

Figure 4.1: 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
```

and long peers, all DHTs use functionally identical greedy routing algorithms (Algorithm 4.1):

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]

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 [42]. 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 of Delaunay Triangulation and Voronoi Tessellation is show in Figure 3.1.

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 set of keys a node is responsible for correspond to the node's Voronoi region. Long peers serve as shortcuts across the mesh formed by Delaunay Triangulation.

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

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 that efficiently compute a Voronoi Tessellation for a given set of points on a plane, such as Fortune's sweep line algorithm [22].

However, DHTs are completely 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 [61]. We use DGVH as the base algorithim for building UrDHT

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 4.2.1.

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 [11]. 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 $O(\frac{\log^4 n}{\log^4 \log n})$ [10], regardless of the dimension [10].[4] In most cases, this cost is much lower. Additional details can be found in our previous work [10].

We have tested DGVH on Chord (a ring-based topology), Kademlia (an XOR-based tree topology), Euclidean space, and even on a hyperbolic surface. This is interesting because not only can we implement the contrived topologies of existing DHTs, but more generalizable topologies like Euclidean or hyperbolic geometries. We show in Section 4.4 that DGVH works in all of these spaces. DGVH only needs the distance function to be defined in order

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

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.

4.2 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 chaper, but can be found on the UrDHT Project site [49].

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.

4.2.1 The DHT Protocol

The DHT Protocol (`LogicClass.py`) [49] 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 4.2.2.

4.2.2 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 6.1).

- 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 [38] and other small-world networks [32] 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 [56]. 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 [32]. Long peers do not greatly contribute to maintenance overhead, so we chose 200 long peers as a default maximum.

4.3 Implementing other DHTs

4.3.1 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. 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.

4.3.2 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 [42] 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 [42].

4.3.3 ZHT

ZHT [36] 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.

4.3.4 Implementing a DHT in a non-contrived Metric Space

We used a Euclidean geometry as the default space when building UrDHT and DGVH [10]. For two vectors $\vec{a}$ and $\vec{b}$ in d dimensions:

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

We implement `getDelaunayPeers` using DGHV and set the minimum number of short peers to $3d+1$, a value we found through experimentation [10].

Long peers are randomly selected from the left-over candidates after DGVH is performed [10]. 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 [33] [19]. This has the potential to allow for minimal latency DHTs.

4.4 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 [42] [38] [56] [64] [8]

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.[5]

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.

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

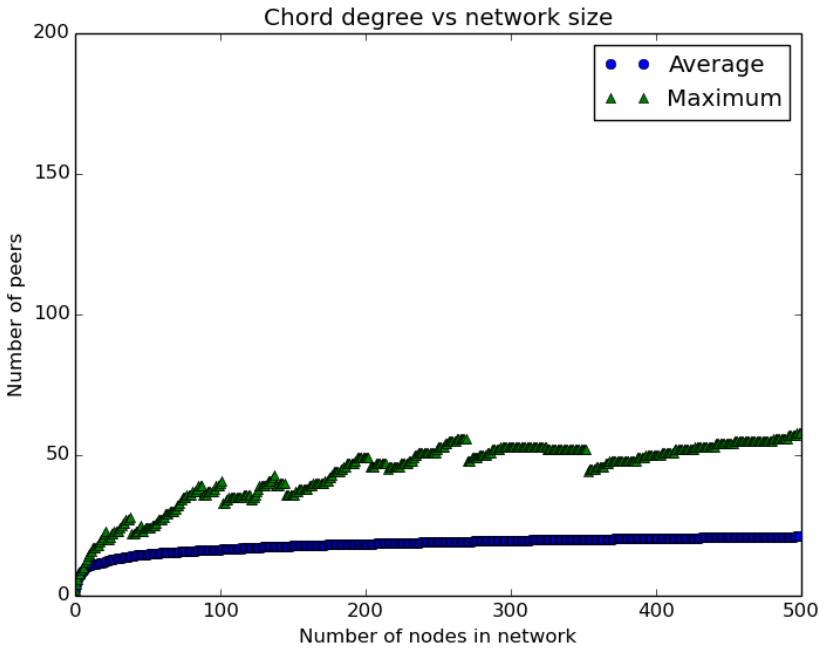


Figure 4.2: This is the average and maximum degree of nodes in the Chord network. This Chord network utilized a 120 bit hash and thus degree is bound at 122 (full fingers, predecessor and successor) when the network reaches 2^{120} nodes.

Results generated by the Chord and Kademlia simulations were in line with those from previous work [42] [56]. This demonstrates that UrDHT is capable of accurately emulating these topologies. We show these results in Figures 4.2 - 4.5.

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 4.6 - 4.9.

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 ex-

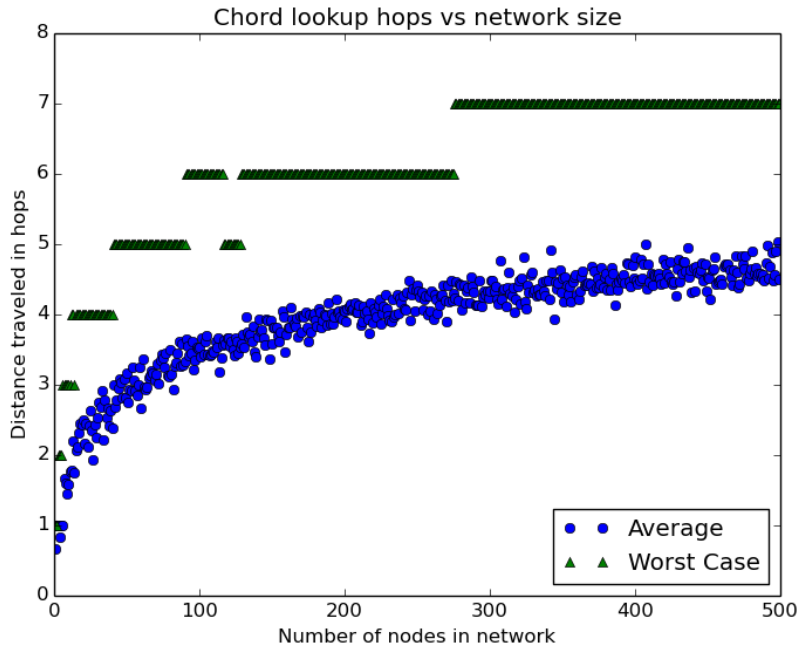


Figure 4.3: This is the number hops required for a greedy routed lookup in Chord. The average lookup between two nodes follows the expected logarithmic curve.

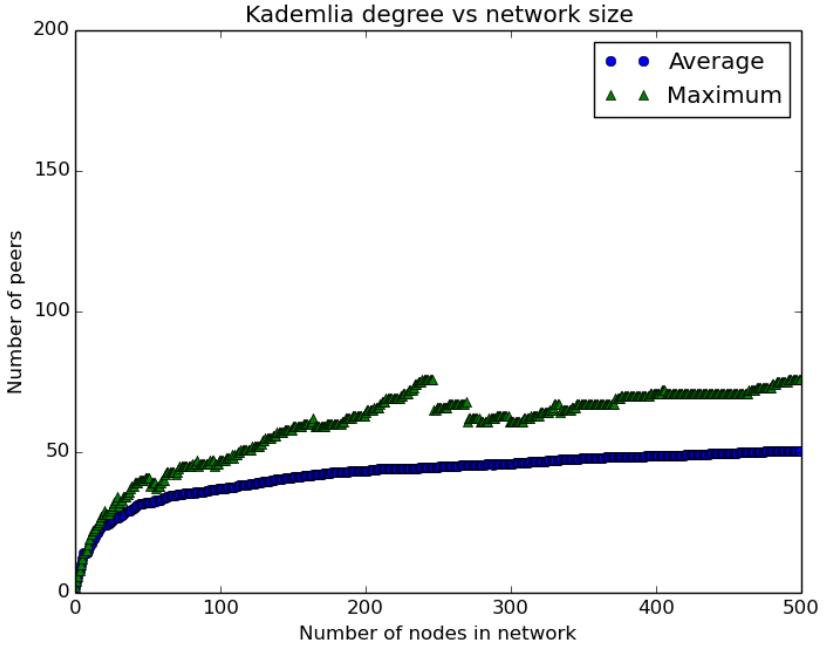


Figure 4.4: This is the average and maximum degree of nodes in the Kademlia network as new nodes are added. Both the maximum degree and average degree are $O(\log n)$.

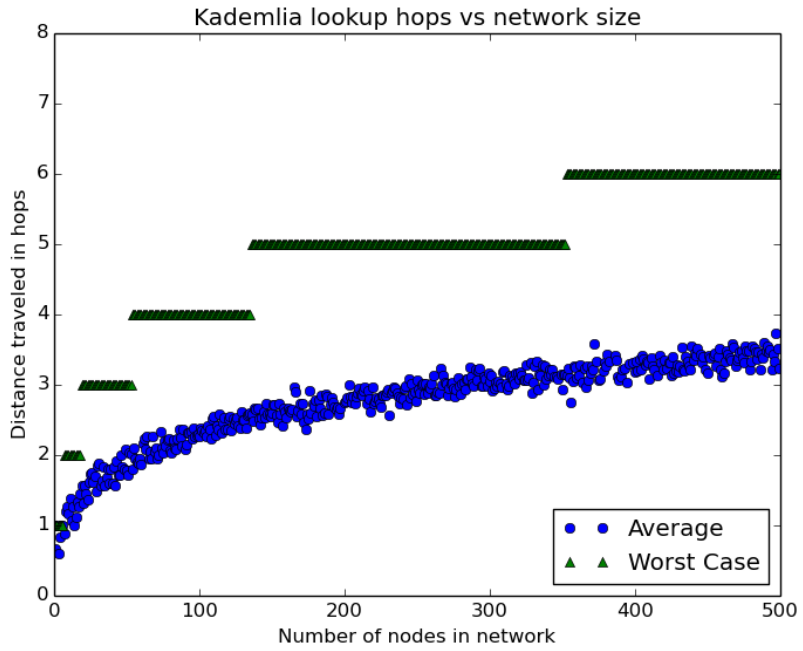


Figure 4.5: Much like Chord, the average degree follows a distinct logarithmic curve, reaching an average distance of approximately three hops when there are 500 nodes in the network.

plains 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.

4.5 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 [37]. 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.

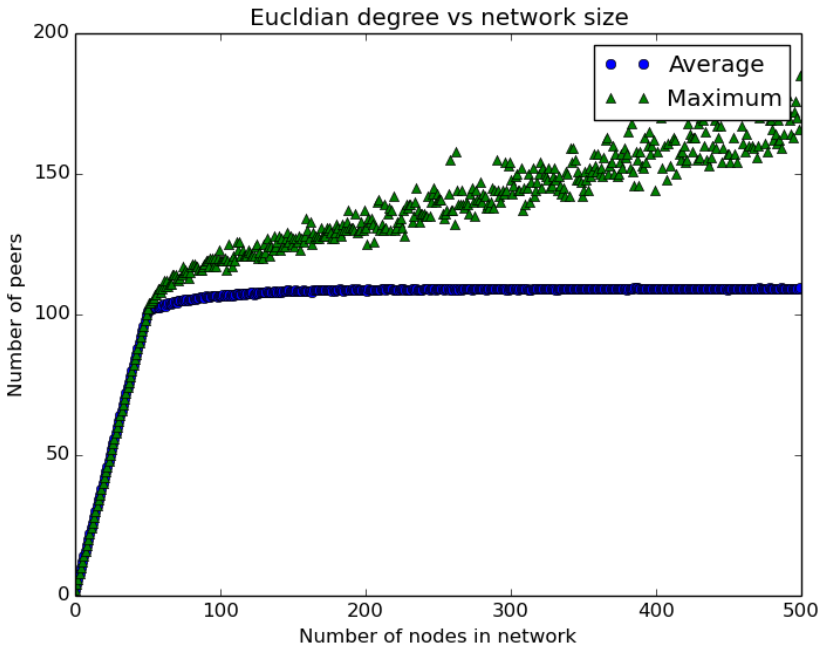


Figure 4.6: Because the long peers increase linearly to the maximum value (49), degree initially rises quickly and then grows more slowly as the number of long peers ceases to grow and the size short peers increases with network size.

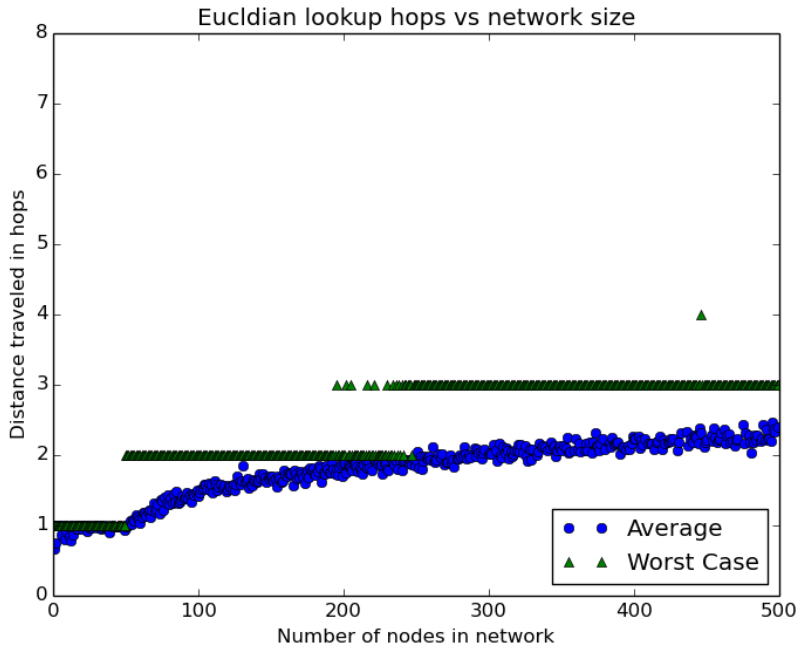


Figure 4.7: The inter-node distance stays constant at 1 until long peers are filled, then rises at the rate of a randomly connected network due to the distribution of long peers selected

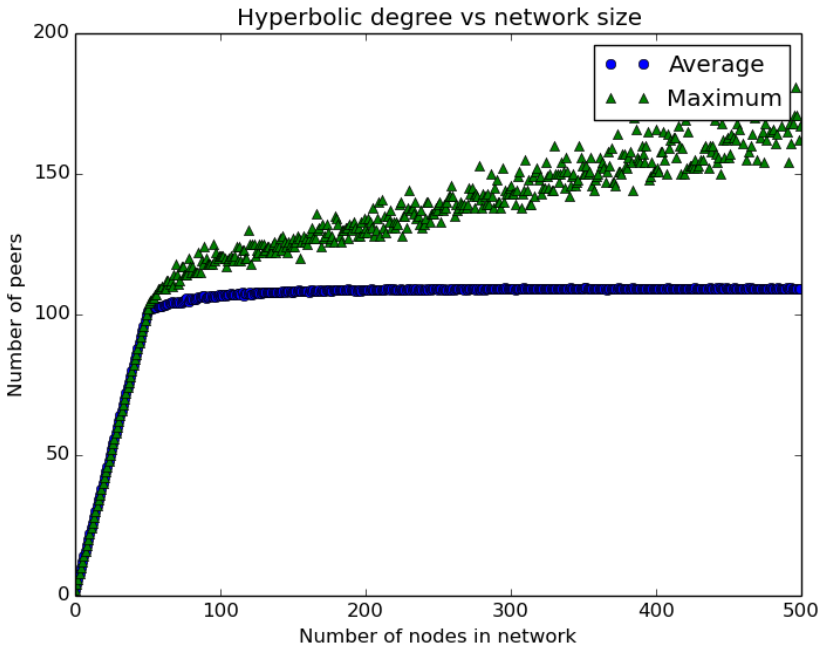


Figure 4.8: The Hyperbolic network uses the same long and short peer strategies to the Euclidean network, and thus shows similar results.

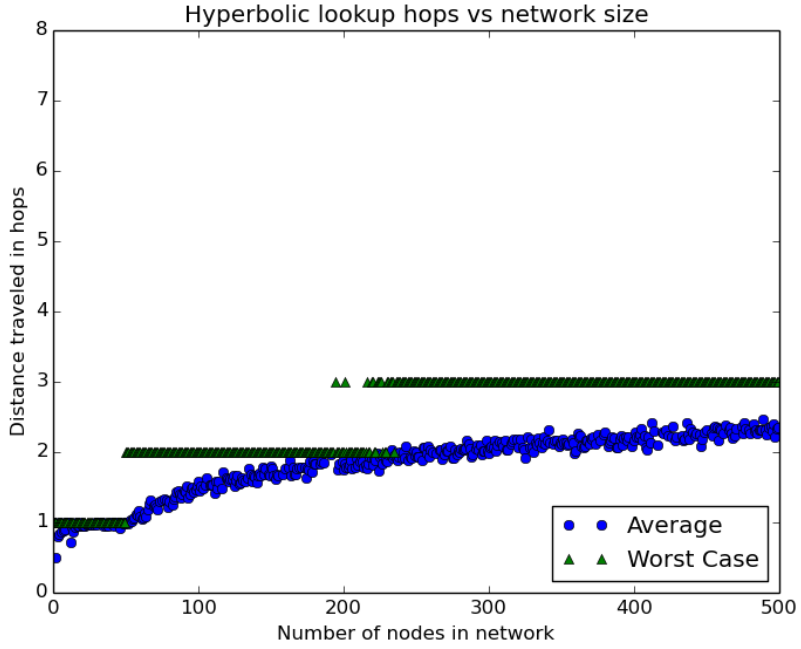


Figure 4.9: Like the Euclidean Geometry, our Poincarè disc based topology has much shorter maximum and average distances.

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

The T-Man[29] and Vicinity [58] 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.

4.6 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 [33] [19]. 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.

Chapter 5

Replication Strategies to Increase Storage Robustness in Decentralized P2P Architectures

Robustness in the face of a dynamic network is a central issue for secure and reliable storage of data in a peer-to-peer (P2P) network with a distributed hash table (DHT) infrastructure. A DHT's robustness will degrade if the churn rate (the rate of participants leaving and joining the network) increases to such a level that data is lost or the network is broken into disconnected segments. DHTs use "replication strategies" to proactively place copies of a record (called replicas) in the network to be used if the original is lost.

This chapter describes a variety of strategies that have been proposed to increase the robustness of storage in P2P systems that that use a DHT as the organization mechanism. Each strategy will be described, analyzed, and finally compared and contrasted with other strategies.

We judge the long term efficacy of each strategy by calculating a "half-life" value for records in that strategy in terms of the baseline rate of churn and the number of replicas placed using the strategy. Using half-life measurements, systems designers can consider the long term behavior of records in the network.

We propose and analyze a novel reactive update strategy where the effective half-life can be made arbitrarily long with minimal impact on DHT efficiency and overhead. Rather than sending potentially unnecessary queries and updates, we send exactly the required messages

Under consideration by IEEE MilCom 2016 at time of writing

to maintain the desired level of robustness. Our approach differs from existing approaches in that they estimate at time of implementation the frequency of replication required to attempt to maintain robustness. This strategy has the potential to practically remove record loss due to churn in P2P systems built on a DHT.

5.1 Robustness

Most DHTs focus their efforts on preventing record loss due to churn. Churn is the constant entry and exit of participants of the P2P network over time. Established systems like Mainline DHT can see a daily net fluctuation of 10,000,000 nodes per day [59] (about half the size of the network).

Churn complicates record maintenance because nodes currently hosting data are constantly leaving without warning and new nodes are constantly arriving and needing data assigned to them. Records are lost to churn under two conditions: the node hosting the record leaves the network or the node with a record ceases to be responsible for the record due to a new node joining the network and claiming that portion of the address space.

We will describe churn as a "Replacement ratio": R_c, which is measured over a period of time (most often a day). R_c is related to the more conventional churn rate metric $\frac{exits+joins}{2 \cdot size}$ but provides more information on the volatility of records. This value describes the portion of a DHT's metric space that is maintained by a different node at the end of the period. This is equivalent to the percentage of records that would ideally have new owners. Ensuring these records migrate to the appropriate new owners and thus providing robustness is the primary concern of this work.

5.1.1 Robustness and network partitions

Network partitions occur when a failure results in the network separating into multiple non-connected networks. Network partitions form from a cut over the graph inter-node connectivity. Failures in either the underlay network or in the overlay network can result in

a network partition. Unlike the churn based failures the failure of nodes is not independent with network partitioning. For example, if two previously connected regions of the Internet cease to be connected due to disaster or political intervention, there will be two new smaller networks. Each smaller network will have just lost access to all the nodes in the other partition. Therefore active robustness methods based on periodically restoring records will degrade into passive methods if the network partitioning places all of the active parties in one partition.

Underlay Partitions

An underlay partition failure can be the result of a failure in the infrastructure, manipulation of BGP, or governmental action. The effect of these, would be to isolate geographic or political regions from each other. Assuming the overlay network of the DHT has been constructed independently from the topology of the underlay network, the failures due to the underlay partitioning will occur at apparently random locations in the overlay network.

While this may resemble how failure occurs during churn, the failures will all occur almost instantaneously to a potentially large segment of the population of the network. Kademlia's topology is likely to recover from such an event, however searching the network will be impaired while new connections are established. Chord's consistent topology proof is built upon an invariant that any join or exit from the network is occurring when the majority of the network is consistent. The sudden failure of nodes in a underlay partition situation violates this assumption and may destroy the ability of the remaining Chord network to form a searchable overlay topology.

We will discuss underlay partitions in terms of a ratio R_u that describes the fraction of randomly distributed members of the network that remain after an underlay partition occurs.

Overlay Partitions

Overlay partitions occur upon the topology generated by the DHT protocol. In the case of an overlay partition, nodes are hypothetically able to communicate with each other, but are unable to do so due to a lack of knowledge of the remainder of the network. In practice, overlay partitions may occur due to eclipse attacks [57] or logic errors in how the DHT constructs the overlay (which have been documented in the Chord protocol). The most likely cause of an overlay partition is a result of updating a DHT protocol in a non-reverse-compatible manner, which will result in two non-communicating networks when only some of the participants of the network update to the new protocol.

We will discuss overlay partitions in terms of a ratio R_o that describes the size of a connected subnetwork that remains after a partition. It is worth considering, that in the case of any partition, the ideal behavior is that both resulting partitions remain connected, searchable, and retain discoverable replicas of all records, rather than the survival of only one of the partitions.

5.1.2 Half Life and Time

Traditionally in statistical discussions of databases, we consider "Up-Time" as the primary metric of reliability. However in a DHT, this metric does not mesh well with reality.

Up-Time is predicated on the idea that records will never be lost, but only temporarily unavailable. Because a DHT sees a much higher node failure rate than the data centers, where Up-Time is the primary factor of consideration, we consider any failure that would result in the temporary loss of availability of a record to simply be an increase in latency. While DHTs attempt to minimize this latency, failure is more honestly measured in the likelihood a record will be lost forever.

Because the amount of data and rate of churn is so high, we must consider that there is a likelihood that a record can be lost simply because all of the machines that held instances of the record on the network have failed (or otherwise left the network). With this in mind

we use the metrics "Half-Life" and "Mean Expected Lifetime".

Half-Life is a unit of measure commonly used in Physics and Chemistry. It describes the amount of time over which there is a 50% chance that an item has ceased to exist (in the current state). This is useful for discussing the lifetime of records in the network because we can calculate the likelihood a replica is lost due to churn over any given period of time and then use this to calculate the half life using Equation 5.1. It is important to note that the time unit for half-life is the sampling period of R.

$$\frac{-\log(2)}{\log(1 - R_{period})} \tag{5.1}$$

Given an R sampled in a given period, we can convert from the original period ($Period_A$) to a smaller or larger period ($Period_B$) by applying Equation 5.2. If we treat the event being considered as equally likely to occur any where in the period over which R was measured, the likelihood of R is equal to the likelihood of occurrence in each smaller slices of time ORed together.

$$R_{Period_B} = 1 - (1 - R_{Period_A})^{\frac{Period_B}{Period_A}} \tag{5.2}$$

This equation is based on the method of calculating a repeated OR operation on a probability. Rather than using the equation $p_0 + p_1 - p_0 p_1$ repetitively, we simplify the calculation by applying De Morgan's laws of negation and instead calculate a repeated AND operation in the form of $1 - (1 - p)^x$. Where the original OR operation could only be preformed in whole steps. This formulation allows us to perform the OR operation in partial steps. Preforming the OR operation $\frac{1}{k}$ times allows us to solve for the value, that if OR operation was preformed on it k times, would equal the originally considered value. In other words, we can consider the likelihood of failure over any time period where a record may be particularly vulnerable given an R initially calculated over a different period.

For a single record, the half-life would represent the period after which there is a 50%

chance the record has been lost. For a population of replicas, the half-life would represent the period of which half of the records are expected to have been lost. For such a population, the mean expected lifetime would describe the period required to reduce the average number of replicas to 1 (which would describe a 50% chance of total loss). For a population of K records, the mean expected lifetime in multiples of the period of R can be calculated using Equation 5.3

$$\tau = \frac{-\log(K)}{\log(1 - R_{period})} \text{where } \tau \text{ is the lifetime.} \tag{5.3}$$

5.2 Existing Passive Replication Strategies

Passive strategies are those where a client writes the record and replicas to the DHT once, after which no participant ever re-publishes the record. Because of constant churn, such records are likely doomed to be lost as the nodes to which they were stored leave the network. We describe passive strategies as a baseline of comparison for reaction to churn. We later show they are identical to more active strategies in response to partition failures.

5.2.1 K-Random Replicas

The K-random node strategy is not used by any established DHT, however it provides a simple analytic model that we can extend to the other replica strategies. In the K-random node strategy, a file is stored at K locations chosen by chaining of a cryptographic hash. That is, if a record is stored a location L, the first replica will be stored at location $hash(L)$ and the second at $hash(hash(L))$, etc., until a replica has been stored K different nodes.

This scheme allows for simple speedup and redundancy. Given a location L, any node can locate the potential K backup sites and search them by order of closeness or in parallel (effectively in order of latency). The locations of replicas are effectively random so we can simplify our analysis for churn and partition tolerance.

The half-life due to churn for the record (a time period over which, on average, the number

of replicas in the network will be halved) is $\frac{-\log(2)}{\log(1-R_c)}$. Because the number of replicas is a small integer, we can also consider an expected mean lifetime of a record in the network:

$$\frac{\log K}{\log(2)} \cdot \frac{-\log(2)}{\log(1-R_c)}$$

This value describes the time period after which the expected mean number of remaining replicas is one. This period is the mean value because there is a 50% likelihood there are less than 1 replica (or zero replicas, as the number of replicas is a natural number that cannot take partial values.)

As a result we can present the average lifetime of a record in the network in terms of the number of replicas and churn replacement rate to be $O(\frac{-\log K}{\log(1-R_c)})$. This implies that while more replicas always results in a longer expected lifetime, the return on adding additional replicas diminishes quickly since treating R_c as constant results in the expected lifetime being a slow growing $O(\log K)$ function, as can be seen in Figure 5.1

The K-random strategy preforms similarly in both underlay and overlay partition failures (as the replicas in the network are effectively random in relation to each type of failure). The expected fraction of surviving nodes is simply the R fraction of the original network that the partition represents. The likelihood that a record is totally lost is simply the odds that all replicas are not in the considered partition: $(1-R)^K$.

5.2.2 K-Nearest Replicas

K-nearest replication is a common strategy in Kademlia based networks. When storing a record, members preform a multi-beam search to discover the K closest nodes to the target location.

This has an advantage over K-random nodes in that in many cases of failure, there is no downtime. If the current owner of a record dies, an adjacent node that likely already has a backup takes over responsibility. In terms of churn resistance and underlay failure, K-

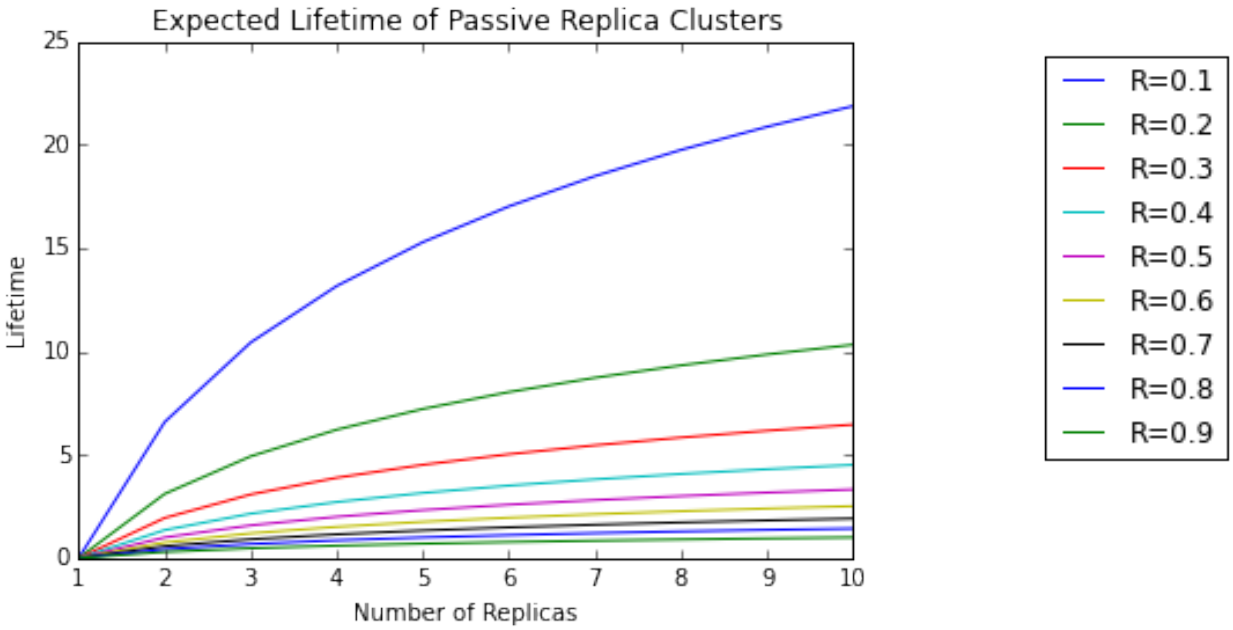


Figure 5.1: The mean expected lifetime of a passive cluster of replicas for a given number of replicas and replacement ratios.

nearest behaves identically to K-random simply because the likelihood of loss due to churn is independent of the replica's location in the network.

In the case of an overlay partition, the K-nearest strategy is less robust because it is more likely that all the nodes hosting replicas will fall on the same side of the network. Unlike the K-random strategy, the odds that all replicas are lost is proportional to the size of the surviving partition R_o. We treat the entire cluster of replicas as one point because the likelihood that the cluster will be cut by the partition is negligible as the network size becomes significantly larger than K. In this case, any $O(1)$ number of replicas would not provide tangible benefit. In the results of the simulation (Figure 5.4, we can see K-nearest in the case of overlay failure performs much worse than other strategies.

5.3 Sponsorship

While the Kademlia chapter proposes a strategy for active replication [42], however in practice only the "greeting" portion (where new nodes are given the records they are now re-

sponsible for) is implemented and Sponsorship is the strategy effectively used. Rather than members of the DHT taking any actions for ensuring reliability, the DHT behaves as originally specified and acts only as dumb storage. To ensure that a record survives churn, a sponsor outside the network periodically re-stores the record and replicas in the network. In an ideal situation, this solves the problem of reliability by placing it in the hands of the users, such that critical records will always have reliable sponsors.

From an analytic standpoint, the sponsor's continued ability to re-store the record is no different from the K-random strategy, admittedly with a much lower rate of node replacement. It is difficult to analyze the efficacy of sponsorship relative to the qualities of a given DHT network because they are intentionally meant to be independent from the DHT. Because there is no assurance that new sponsors will be created when existing sponsors are lost, Sponsorship only provides a more reliable pool of nodes to be exhausted by exponential decay. Once the sponsors have all failed, replicas will be quickly lost as discussed in passive strategies. The half-life and expected lifetime equations of the record under active sponsorship is identical to passive strategies as we assume the lifetime provided by the higher reliability will dominate the lifetime provided by passive replication. Some practical applications of DHTs like Bittorrent [18] and IFPS [9] add additional sponsors as the record is demanded, however unpopular records are essentially as likely to be lost as passive K-random replicas once the sponsors fail.

5.4 Active Replication Strategies

Unlike the passive replication strategies, which doomed even highly backed up records to eventual loss, active strategies result in much higher expected record lifetimes. In active replica strategies, records and replicas are restored as they are lost due to churn. However, there will always be a chance that records could be lost despite all efforts to back them up.

5.4.1 Active K-Random Replicas

The active K-random replicas strategy requires a slight modification to how the DHT stores information. In addition to recording the key value pair, if a replica is being stored, the original location of storage must be accessible to the node. In this strategy, all nodes in the DHT iterate over the values stored in them (likely spread out over the span of hours or a day) and ensures the record and all k replicas are periodically restored.

The failure state for this strategy requires that all nodes hosting the records to be removed from the network before any one of them can restore the record. This means, likelihood of failure over the restoration period is R_c^K. This essentially dramatically increases the efficacy of replicas in increasing the half-life of the record: $\tau_{1/2} = \frac{-\log 2}{\log(1-R_c^K)}$. We can simplify this half-life into a format more comparable using the bound $log_2(1+x) \leq x$ which results in:

$$\tau_{1/2} \geq (\frac{1}{R_c})^K \tag{5.4}$$

Unlike the passive strategy, the addition of more replicas exponentially increases the half-life of the record in the network. In addition, the base $\frac{1}{R_c}$ indicates that lowering the restoration period, and thus decreasing R_c, also causes dramatic improvements to the reliability. We can see in figure 5.2 that while a higher rate of failure increases the number of required replicas for a desired half-life, there is always increasing improvement from adding more replicas.

Active K-random replicas behaves identically to passive K-random replicas in terms of its partition tolerance (both underlay and overlay). In the K-random strategy, for each record stored with a node, that node is required to periodically communicate with $K-1$ random other nodes. Because the number of records would far exceed the number of nodes in the network, in effect each node would be required to regularly communicate with every other node in the network. The basic premise of the scalability of a DHT is that each node is only required to regularly communicate with a limited number of peers, so it is clear that

while there are benefits to active K-random replication in terms of expected lifetime, the maintenance cost is prohibitive.

As an example, let us consider a network with R_{day} likelihood of a record being lost per day. If $R_{day} = 0.99$ then the likelihood of loss in 1 minute is $R_{minute} = 1 - (1 - R_{day})^{\frac{1}{24*60}}$ or $R_{minute} = 0.0031929$.

Given this R_{minute} and the requirement for K failures in the period to occur, the expected half-life of the record in the network is $\frac{-\log(2)}{\log(1-R_{minute}^{K})}$. If K is set to 10, the expected half-life in minutes of the record with 10 active replicas in a network where 99% of the nodes are replaced in a day is $6.29 \cdot 10^{24}$ minutes or $8.7 \cdot 10^{8}$ times the age of the universe.

In practice, half-lives this large indicate that it is reasonable to expect the likelihood of the record's loss to be dominated by partition failures and the eventual obsolescence of the network. Some examples of this behavior can be seen in Figure 5.2. We can see, as the rate of replacement due to churn rises, the efficacy of replicas to prevent failures falls, requiring more replicas of similar guarantees of robustness.

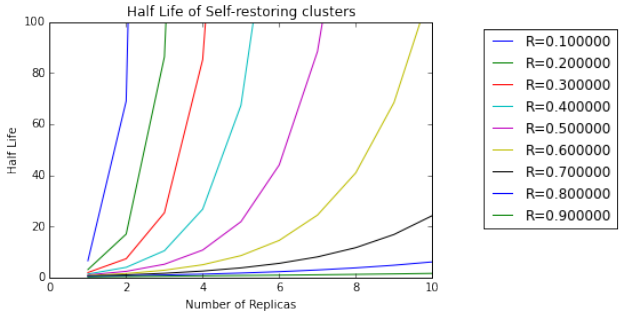


Figure 5.2: This graph shows the expected half-life of active clusters of replicas in response to churn. Note that the value for R dramatically effects behavior.

5.4.2 Active K-nearest Replicas

The basis of this strategy can be found in the Kademlia [42] paper, however no analysis on reliability due to the replica strategy is offered in that paper. The Active K-nearest replicas provides a churn robustness identical to the Active K-random strategy. However it provides an opportunity for a significantly lower maintenance overhead than K-random

replicas. Where the random nature of backup site location effectively required the network to act as a clique, K-nearest associates backups with those nearby to the record. These nodes will be the peers a node is already required to communicate with regularly. This way, the number of maintenance messages is dramatically reduced to K nearby peers (however the size of these messages would still be determined by the number of replicas).

5.5 Summary of Comparisons

The methods discussed show 3 major classes of behavior, passive, active but inefficient, and active and efficient. Passive strategies are very easy to implement and require essentially no maintenance, but under realistic churn conditions records should not be expect to last more than a day or two. All of the active strategies discussed provide the desirable $\frac{-\log 2}{\log(1-R^K)}$ half life. However the Active K-Random strategy is intolerably inefficient. Because the number of records is likely much larger than the number of participants in the DHT, it is reasonable to expect K-Random will be responsible at least one record or replica that is also stored with every other member of the DHT. Thus nodes utilizing a K-Random strategy will need to be constantly seeking and re-storing records to keep up with restoring the records for which they are responsible. Compared to Active K-Random, Active K-Nearest is much more efficient because a node will very rarely have to search for the owner of a replica, rather it restoring backups to nodes it already knows and already periodically communicates with as part of the DHT.

Sponsorship stands out as largely incomparable with the other methods. It is intentionally designed to make record replication independent of the state of the DHT itself. Unlike the passive or active methods, its efficacy is a function of the actions of human beings, rather than a deterministic system and thus the lifetime of records when it is used will vary wildly.

When considering partition tolerance in these strategies, active strategies do not provide improvement to handling partition failures over passive strategies. Because we only consider survival over the partition event, which occupies a very small amount of time, only the

```
1: procedure On Peer Exit(p)
2:     for r in Backups received from p do
3:         if I own r now then
4:             Mark r as owned
5:         end if
6:     end for
7:     Identify the now Kth closest peer NewPeer
8:     Backup all records marked owned to NewPeer
9: end procedure
10: procedure On Peer Join(p)
11:     for r in records marked owned do
12:         if p owns r then
13:             Send r to p
14:             Mark r replica issued from p
15:         end if
16:     end for
17: end procedure
18: procedure On Self Join(p)
19:     for Records r received from each peer p do
20:         if I own r then
21:             Mark r as owned
22:         else
23:             Mark r as backup from p
24:         end if
25:     end for
26: end procedure
```

Figure 5.3: Reactive K-nearest Replication Procedures

locations of replicas in the network is going to be relevant to the record's survival.

Similarly to the passive nearest K replicas strategy, the active K-nearest strategy is identical to K-random replicas in the case of an underlay partition and like the passive K-replicas strategy, it is equivalent to having no backups at all in the case of an overlay partition because the backups are clustered together. This indicates that while the active K-nearest strategy is incredibly robust to churn and tolerably robust to underlay failure, it is comparatively vulnerable to overlay partition failure.

5.6 Proposed Reactive K-nearest Replication Strategy

We present Reactive K-nearest replicas as a set of modifications to active K-nearest replicas that allows us to gain the same benefit with a smaller window of vulnerability and with less maintenance overhead. Reactive K nearest has the same $\frac{-\log 2}{\log(1-R_c^K)}$ half-life as active strategies, but has the potential for a much shorter vulnerable windows in calculating R_c. As a result of the smaller window of vulnerability, we have a low likelihood of loss during that window (as seen in Equation 5.2) and greater returns on adding replicas to the network.

When replicas are associated with nearby peers, it is reasonable to expect that records will only be lost when those peers exit the network or are displaced by newly joining nodes. Because of this, rather than attempt to track the existence of each replica, we can track the peers with which many replicas have been stored. The DHT protocol already implements tracking the state of nearby peers so this adds no additional overhead.

This means rather than a periodic re-store of the large number of replicas, we store the replica once when the record is initially stored, and we react to the change in our peer list by re-issuing the affected records. This means, the amount of overhead is exactly that required to maintain the desired level of robustness.

We can further restrict maintenance cost by limiting responsibility of re-storing of replicas to the initial owner of the record, and yet keep the same benefit of all K active replicas. We do this by giving the replica owning node the responsibility to act upon the failure of the responsible node by instantly assuming responsibility for the records closest to them (which they already have) upon the failure of the originally responsible node. As shown in the procedures in Figure 5.3, we only react to changes in topology, rather than periodically re-storing replicas. Thus, if no updates are required, no resources are used to make them. If peers are joining and exiting often, only enough messages to ensure there are always k backups are sent.

5.7 Experimental Validation

For each of the strategies, we simulate partition failures to validate our formal analysis. We simulate underlay and overlay failure on the Chord DHT and report the resulting record survival rates for each strategy. We only consider the survival of a single record and its replicas, as this simplifies computation.

We have chosen the Chord DHT, because it is the only DHT to even theoretically experience an overlay failure during operation [63]. We simulate a partition along the chord ring, where a continuous portion of the chord ring of size r is removed. Underlay failures are considerably simpler to simulate, as we only randomly select the appropriate portion of nodes and remove them from the network. A record is only considered totally lost if it and all replicas have been removed from the network by the partition.

For each simulation we construct a simulated 10,000 node DHT. We will consider underlay partitions of sizes 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%, 90% Each data point presented is the average of 1000 trials run with those parameters. All simulations are run with a K of 5 replicas.

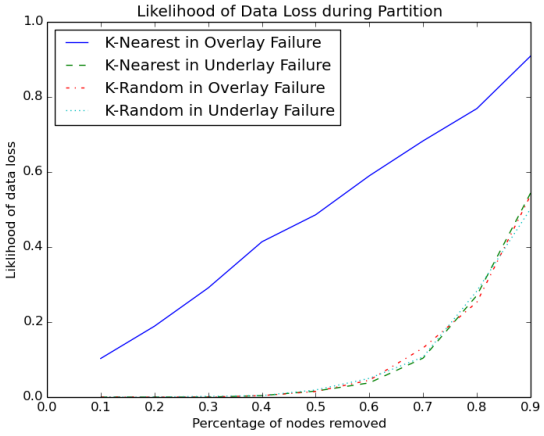


Figure 5.4: The total loss likelihood in cases of network partition

5.8 Conclusions

We presented a novel system for increasing robustness in decentralized P2P networks implemented with a DHT. This strategy dramatically improves the robustness of the system as measured by the half-life of data records. The half-life is the time for 50% of the copies of data to be lost and describes the characteristic time scale of data loss. Using modifications of strategies presented in 2002 by Kademlia [42], which have remained largely unimplemented for the past 14 years, we can build DHT networks that can reasonably be expected to never lose records due to churn during their active use lifetime. We achieve this with lower overhead than the currently used sponsorship strategy with a resulting half-life of $\frac{-\log 2}{\log(1-R_c^K)}$ and the potential for lower values of R_c than are available to active and sponsorship strategies.

A DHT is not designed to be repaired in the traditional sense, but instead to consist of a constantly changing set of computers around the world. Like the philosophical Ship of Theseus, it is constantly in the process of being replaced and renewed. If we can make a DHT a highly reliable store of data and ensure it is actively participated in then a DHT has the potential to persist longer than any other medium of storage because it is constantly self-healing. This allows us to begin to consider the use cases of a system for storing information that could reliably last generations.

The ability to retain information for an indefinite lifetime opens up other use cases for new robust P2P systems based on DHTs such as a distributed credential store [40], an archival database, a public library, or a set of reliable terrain and mapping data with an associated historical context. It defines a mechanism to preserve any critical data for arbitrarily long periods of time while allowing access from many locations. It also opens up potential problems. If records do not have a fixed lifetime, then the data stored in the DHT can grow without bound. Therefore future research should examine rigorous and secure mechanisms for data revocation and expiry.

Chapter 6

Online Hyperbolic Latency Graph Embedding as Synthetic Coordinates for Latency Reduction in Distributed Hash Tables

Overlay topologies are networks built between a group of computers which are all part of a larger network (underlay) that route messages between each other. One of the purposes of a Distributed Hash Table (DHT) is to build and maintain a scalable overlay network for a distributed system. A direct connection between two nodes in an overlay network in fact uses a route on the underlay network to actually send the messages. Often, an overlay network is constructed with little regard for the topology of the underlay network, such that while routes may seem efficient in the overlay network, they are inefficient in the underlay network.

Previous work by Kleinberg [33] and Papadopoulos [44] has explored the possibility of using a hyperbolic space to assign a node's location in a DHT, such that paths efficient in the overlay network are also efficient in the underlay network. Because current DHTs make no consideration of latency in building their overlay typologies, there is much room for improvement in that regard.

Kleinberg [33] and Papadopoulos propose that future work should devise a general method for maintaining the topology in spite of node failures and joins, also referred to as churn. We utilize our previous work with the Distributed Greedy Voronoi Heurisitc (DGVH) [10] to solve this problem.

Under consideration by IEEE CNS 2016 at time of writing

In addition to lacking a maintenance mechanism, previous explorations lacked an efficient method for new nodes to accurately insert themselves into the network. We present a simple, first effort, technique for placing nodes in the hyperbolic space that while not optimal, is sufficient to produce dramatically improved results.

We combine these mechanisms to show that a DHT built to minimize latency in a Hyperbolic Space is a viable technique that provides dramatic latency reduction over traditional DHT techniques. We provide simulation results to demonstrate the achived latency reduction and we show that concerns raised by Kleinberg concerning high congestion in the network can be solved using DGVH.

6.1 Background

There is an inherent trade-off between availability and latency minimization in DHT design. A design that always ensures minimal latency routes is a clique [36] that provides one hop lookups. However this design requires all members of the network to maintain $O(n)$ connections, which limits the scalability of such a system.

A Distributed Hash Table (DHT) is designed to ensure the scalability of the system by requiring nodes to track only $O(\log n)$ nodes in the network. This necessitates higher latency than a direct connection. We present a technique to minimize the maintenance overhead and query latencies within the constraint of maintaining the scalability of the network.

Distributed Hash Tables form the core of many Peer-To-Peer (P2P) systems like BitTorrent [30], CJDNS [25], and I2P [62]. Improving the response time and efficiency of DHTsand similar systems will provide these systems increased performance and capacity to scale to more users.

6.1.1 Geographic Routing

Geographic routing is used in latency optimization of Content Delivery Networks and Wireless Sensor Networks [31]. Unlike traditional network routing mechanisms, messages can be

routed in a network based on the location of the current node and the location of the destination. While the ideal is that messages could be routed greedily, shortening the physical distance to the destination with each step, the realities of radio and connectivity limitations have resulted in a number of proposed algorithms of increasing complexity.

There are problems with the application of simple greedy geographic routing in practice. Geographic locations may not accurately represent network connectivity and latency. Often "holes" or "lakes" in the network prevent greedy routing. This requires more complex and stateful routing methods to act as an auxiliary to greedy routing. The fundamental problem being that due to the inability to make connections past an obstacle, we cannot create a greedy spanning graph.

While we will take inspiration from the mechanisms of greedy geographic routing, DHTs are not vulnerable to the problem of untraversable terrain that plagues wireless sensor networks. In practice, as long as a greedily traversable overlay is maintained, we can strictly use the greedy geographic forwarding strategy to route messages. This leaves us with the problem that locations and routes in space often do not match the throughput and latency reality of the network. The focus of this work is building a coordinate system alternative to geographic locations that provides the ability to leverage greedy geographic routing for both successful and efficient routes.

6.1.2 Greedy Traversable Graphs

The idea of a Greedy Traversable Graph is that, given a distance function, a route can be found between between any two nodes by greedy best-first search without any backtracking. This provides efficient routing through the network without maintaining any state in the packet or maintaining routing tables.

The property that ensures the ability to route greedily is that for every two nodes a and b that are not adjacent in the graph, there exists some node c that is adjacent to a and c is closer to b than a is. Essentially this ensures that if I am not adjacent to my target I can

Figure 6.1: DGVH Algorithm

```
1: Given node n and its list of candidates.
2: Given the minimum table_size
3: short_peers ← empty set that will contain n's one-hop peers
4: long_peers ← empty set that will contain n's two-hop peers
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 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: if |long_peers| > table_size^2 then
21:    long_peers ← random subset of long_peers of size table_size^2
22: end if
```

always take a step closer to my target.

DGVH

DGVH [10] is a technique for building a minimal Greedy Traversable Graph over points in arbitrary metric spaces.

The DGVH Algorithm (Figure 6.1) directly ensures the Greedy Traversable property by creating links to any node to which we do not have an intermediary step. DGVH tracks two hop peers up to a constant limit to ensure that the overlay network is rapidly repaired when node joins and failures occur.

DGVH essentially handles the basic tasks of ensuring network reachability and maintenance. Utilizing DGVH as a basis for our system, we can focus on implementing an effective

embedding mechanism with insurance that we cannot damage the basic function of a DHT.

6.1.3 Distributed Hash Tables

Distributed Hash Tables are a scalable system for decentralized key-value storage. DHTs use a mapping from the key to a location in the network using a cryptographic hash and a coordinate space. Nodes each maintain a peer list of limited size and maintain a topology that can be greedily searched for any record. The coordinate spaces and their metrics vary wildly between implementations of DHTs.

Importantly, DHTs are an overlay network, so we work under the assumption that an underlying routing system will allow any node to connect to any other node. This eliminates the issue of "holes" that was found in traditional geographic routing. Attempting to minimize latency in a DHT means attempting to leverage the latency minimization work already done by the underlay network.

An important refinement we present on the discussion of DHTs is the observation that all of them work under the same basic principle. Nodes are responsible for their Voronoi region in the given metric space and they must maintain peer links with the corresponding Delaunay triangulation in order to ensure there is consensus on who owns which records. In addition, DHTs form "long" connections across the metric space in addition to their Delaunay peers to ensure that the resulting network has Kleinberg "small world" [32] properties.

Chord [56] and Kademlia [42] are the most common methods of implementing a DHT. We use them as a baseline comparison for our improvements. Both systems maintain $O(\log n)$ connections to peers and promise $O(\log n)$ hops between any two nodes. Because neither attempts to minimize latency, the latency stretch is $O(\log n)$.

6.1.4 Scale Free Networks

Scale free networks are a family of tree-like networks defined by an having exponential degree distribution and a tendency for high degree nodes to be linked to other high degree nodes.

These properties describe a network with a $O(\log n)$ diameter, which can often be considered $O(1)$ in large systems.

Scale free networks are of interest to us because they are a generalization of the topology of human-built digital networks. The latency distribution of a representative sample of a computer in the global internet should show a latency similar to that of a representative subset of a scale free graph.

6.2 Previous Works

6.2.1 Kleinberg's Hyperbolic Embedding

A distributed and dynamic hyperbolic embedding of latency suitable for optimizing a DHT was envisioned by Robert Kleinberg in 2007 [33].

Greedy Embedding

The Greedy Embedding discussed by Kleinberg is inverse to the "DGVH" method of generating a Greedy Traversable Graph. Rather than being given a set of points and generating a Greedy Traversable graph, we are given a graph and metric, then solve for points for each node such that the resulting graph is Greedy Traversable when using those points in that metric.

Initially discussed by Papadimitriou *et al* [43], a Greedy Embedding can be formally defined as a distance function between two points $d(a, b)$ in a given metric space and a function $f(v)$ that maps each vertex in the given graph such that for every pair of non-adjacent vertices $a, b \in G$ there exists a third vertex c adjacent to a such that $d(c, b) < d(a, b)$. Essentially, the result of this definition is that for any vertices not directly connected, there exists a path of nodes that are iterative steps closer to the target that could be followed by Greedy Traversal. Interestingly, graphs produced by DGVH are forced by the algorithm to fulfill this property.

The Geometry of the Hyperbolic Plane

Hyperbolic space in two dimensions is defined as the surface of a hyperbola in three dimensions where all paths between points are taken along the surface of the hyperbola. This hyperbola is defined by the equation $z^2 = x^2 + y^2 + 1$. Note that for any (x, y) pair there exists two solutions for the value of z, one positive and the other negative. These two solutions form two disconnected "sheets" mirrored across the xy-plane. By convention, we only consider points on the $-z$ sheet. All calculations and processes work effectively on either sheet as long as all considered points are on the same sheet.

The hyperbolic plane has many differing qualities from the Euclidean plane. Most importantly here is that of "relativity". On the Euclidean plane, we can treat any point as the "origin" and calculate new locations for every point or figure on that plane by translation, while having all inter-point angles and distances remain the same. The Euclidean distance equation $\sqrt{(x_1 - x_0)^2) + (y_1 - y_0)^2)}$ can be interpreted as translating (x_1, y_1) into the reference frame where (x_1, y_1) is the center and using the distance metric $\sqrt{x^2 + y^2}$ to determine the distance. Unlike this, the hyperbolic plane has a defined center. While points can be rotated and mirrored freely over this center without changing inter-point distance and angle, they cannot be translated. The distance between two points in the hyperboloid model is $\operatorname{arcosh}(z_0 z_1 - x_0 x_1 - y_0 y_1)$ where $\operatorname{arcosh} x$ is defined as $\ln(x + \sqrt{x^2 + 1})$.

Unlike Euclidean space, a hyperbolic plane has a Greedy Embedding for any graph. Kleinberg presents his technique for building an arbitrary Graph Embedding in hyperbolic spaces by building a spanning tree of the graph, then embedding the tree into the hyperbolic space. This is effective because the circumference of the disk increases exponentially with radius. Therefore we can trivially embed trees into the space as the available space on the disk increases with depth correspondingly with the number of leaves in a full tree. While the result is Greedy Traversable the resulting embedding does not provide desirable qualities in that greedy routes are not necessarily the shortest and central nodes can receive high levels of congestion.

6.2.2 Greedy Hyperbolic Embedding

Further work by Papadopoulos *et al.* [44] shows a greedy centralized technique for managing the creation of a dynamic network in a hyperbolic space. Papadopoulos *et al.* improves on Kleinberg's initial work by presenting a simple strategy for handling node insertion in the network to simulate a scale free graph and routing methods to handle node and edge failure.

Growing Network with Greedy Embedding

Papadopoulos *et al.* provides a sophisticated method for accurate generating a growing scale free network in a hyperbolic space. It is important to note that they are creating a new topology and network from scratch, rather than attempting to use existing latency information to build a greedy embedding that represents a network topology.

Papadopoulos *et al.* shows that graphs with similar properties to those found in implemented network topologies can be generated and have desirable greedy routing properties. This is accomplished by randomly selecting points in the hyperbolic space such that all points are constrained within a given disk that represents the network radius and a distribution of radius that represents the branching factor of the scale free graph intended to be simulated. Points are then connected together randomly, using a model weighted by inter-node distance and angle.

Utility in building a Hyperbolic DHT

While interesting, the statistical method of peer selection described is insufficient for ensuring a greedy embedding that can be navigated under churn. The proposed gravity-pressure routing method requires packets maintain a nonviable amount of state information without reasonable benefit. It begins to follow the path of geographic routing techniques with growing sophistication in response to an environment that is not greedy routable. We are better served by ensuring the environment is greedy routable than attempting to add overhead to routing to manage failures of topology maintenance.

6.3 Proposed Hyperbolic DHT

We present a technique for building a Distributed Hash Table on a hyperbolic metric space to minimize look-up and maintenance latency within the constraints of ensuring the scalability of the system. We show a simple greedy method for inserting nodes into the network such that latency is congruent with the distance metric. The result of this is that nodes closer together in the DHT overlay have shorter latency to each other. This causes maintenance and queries to to require less latency then if peers had been selected randomly.

We show that unlike previous works indicate, no special updating of node location is required in response to the joining or exiting of new nodes and, in fact, a constant churn rate will help the system respond to changes in global latency distribution. We have found that if we augment the network topology with a Greedy Traversable Graph, holes introduced via node loss are accurately repaired and new nodes are greedily inserted at latency-ideal locations in the network.

6.3.1 The Value of Approximation

Unlike previous work, in presenting a practical DHT description, we are dealing with different premises than previous work with hyperbolic embedding. The largest divergence from Kleinberg and Papadopoulos *et al.*'s works is that, rather than handling the embedding a of an entire Internet sized scale free graph, we are embedding a smaller effectively random sampling of that graph. Secondly, we will be handling constant churn of members of the network and changes in inter-node latency over time. The consequence of this is that the exact methods presented in previous work, which promise essentially optimal routes, cannot be extended into reality in that fashion. Because we are bound to the scalability limitations of a Distributed Hash Table, we will be required to induce stretch simply because we cannot connect all nodes in the overlay network as a clique.

6.3.2 Hyperbolic DHT Model

We establish a Hyperbolic DHT Network in the hyperbolic plane using the hyperboloid model. While any hyperbolic plane representation will work effectively, for accurate internal representation we utilize the x, y, z coordinates of points in the 3D hyperbolic sheet.

We chose this rather than Poincare disk or similar representations due to the inability of floating point numbers to accurately represent values at the extremes of those models. Differences in location on the Poincare disk of the order of ϵ (where $1. + \epsilon/2 = 1.$ in floating point arithmetic) correspond to large differences in location in the hyperbolic plane. $1. - \epsilon$ is the largest radius that can be represented in single precision and corresponds to to a radius of about 12 in the hyperbolic plane.

Using a direct application of DGVH's capacity to build Greedy Traversable overlay networks in arbitrary metric spaces, we can extend the contrived metric spaces of Chord and Kademlia into a more general model. This would allow DHTs to be constructed in any metric with the triangle inequality and symmetric distance. Conveniently, as it is the focus of this chapter, the hyperbolic plane metric space is easy for DGVH to utilize.

We will use points on the hyperbolic plane in the hyperboloid representation [28] (3D coordinates of points on the bottom sheet of the hyperbola) with the distance metric $\operatorname{arcosh}(z_0 \cdot z_1 - x_0 \cdot x_1 - y_0 \cdot y_1)$

Joining and Embedding

The only divergence from a traditional DHT's operation is in the assignment of points in the coordinate space to joining nodes. To preserve the accuracy of the embedding, we must place nodes in the network at a point where they have low local latency. Given that the goal of the embedding is that paths can be routed with minimal latency, we leverage the inverse to greedily place nodes into the network.

Given any arbitrary node in the network as a starting point, the joining has two sequential greedy searches:

First the joining node greedily searches for the location $0, 0, -1$ which represents the "center" of the network. Once we have a reference to a node at the center of the network, we perform a second greedy best first search similar to before. In the second search, rather than looking up the next hop, we query a node for its peers. We then ping each peer of the current best hop. If we find a best hop, such that all peers of that hop have higher latency to us than it, we have found our best insertion position. We select a location subordinate to this node and perform a traditional UrDHT join at this location. This search starts at the center of the scale free graph, and navigates towards the ideal point of insertion.

6.3.3 Routing and Message Passing in the Hyperbolic DHT

Routing uses a modified "recursive" method. It is important to note that usage of the overlay network does not provide any increase in efficiency, rather it provides an efficient mechanism for finding the owner of a given location with minimal overhead while preserving the capacity of the network to scale to arbitrary size efficiently.

Generally a message will be targeted to a specific location in the metric space, rather than a specific server, and whichever server is responsible for that location will handle the query. Often a message will originate from a user who is not a member of the DHT, but is querying to store a value or retrieve a stored value.

A message will begin in the network at a selected "sponsor", who hopefully is chosen for low latency with the user. This is not required and a sponsor can simply be any known member of the DHT. The message is passed between members of the DHT using a greedy best first strategy, that forwards the message to the peer closest to the destination location, in this case using the distance along the geodesic. No trace-back or hop count information is required to ensure delivery and a node will consider itself the destination of the message when it is closer to the destination location than any of its peers. Once the query message is handled (often storing or retrieving a value), the response (a success notification or the requested data) will be sent directly to the user rather than using the overlay network.

6.3.4 Storing Records on the Hyperbolic Surface

The traditional mechanism of mapping keys to cryptographic hashes is less intuitive when locations in the space are actual points rather than integers. The most straightforward method is to design a pseudo-random point generator that can be seeded using the more traditional cryptographic hash. Care must be taken to ensure that the results of this process are evenly distributed. Using a classical pseudo-random number generator like mersenne-twister with classical 32-bit or 64-bit data types will bound the maximum number of unique locations to the number of unique integers the PRNG can generate and, as the distributed system grows in size, increases to the size and format of these identifiers may be required.

As we are using the hyperboloid model of representing points in hyperbolic plane, we must map our hash values onto the hyperbolic plane with the goal of distributing the resulting points evenly over the portions of the plane actually occupied by nodes in the system. Given a centered and bounded disk on the hyperbolic plane in which all nodes fall, we can expect the distribution of nodes to be linear over the polar angle and exponentially distributed over the radius.

Using the algorithm in Figure 6.2 to produce points will evenly distribute the random points over the disk out to a maximum hyperbolic radius of MAX_R. A maximum radius is required because a higher share of points will be distributed to locations distant from the origin of the space as the circumference of the space increases exponentially in response to radius. While it would be possible to distribute points over an unbounded space statistically, in this case the majority of points would be assigned to portions of the hyperbolic disk increasingly distant from the embedding of the DHT nodes.

While it is perfectly possible that the network would either be smaller or larger in radius than a pre-chosen MAX_R, the disparity in load due to this is likely a preferable problem than attempting to vary MAX_R at runtime. If there is an expectation of the network's size at the time of establishment, an appropriate MAX_R can be chosen in respect to that.

No matter what MAX_R is chosen, all keys will be assigned to responsible nodes. However,

Figure 6.2: Pseudo-random location selection Algorithm

1: $SEED(HASH(key))$
2: $ANGLE = 2\pi RANDOM()$
3: $H_R = MAX_R + \frac{\log(1-RANDOM())}{MAX_R}$
4: $P_R = \sqrt{\cosh^2(H_R) - 1}$
5: $X = P_R \sin(ANGLE)$
6: $Y = P_R \cos(ANGLE)$
7: $Z = -\sqrt{X^2 + Y^2 + 1}$
8: Return (X, Y, Z)

if MAXRADIUS is smaller than the network radius, then it is likely nodes on the periphery will not be assigned records. This may be ideal behavior in more general systems than a DHT as these nodes will likely have high latency.

6.4 Analysis

We present two independent arguments that the diameter of a Greedy Traversable Graph Embedding of a scale free graph in the hyperbolic plane is $O(\log n)$ or better.

Given that the diameter of a scale free graph is $O(\frac{\log n}{\log\log n})$ [12] and an efficient Greedy Embedding has $O(1)$ stretch over ideal latency, then the diameter of the Greedy Embedding must be $O(\frac{\log n}{\log\log n})$ or better.

The greedy insertion algorithm attempts to uniformly insert nodes the hyperbolic plane such that the Voronoi regions of these nodes are approximately equal in area. The area on a hyperbolic plane is thus $O(n)$, and a path drawn across it is length $O(r)$. In terms of radius, area of a disk on the hyperbolic plane is $O(e^r)$. Because of this, we can assume an arbitrary path drawn on the surface will cross a number of regions proportional to its length, which is $O(\ln n)$.

6.4.1 Expected Path Stretch

The ideal stretch ratio ($\frac{ActualPathLength}{OptimalPathLength}$) for the hyperbolic embedding is $O(1)$. However simulation shows our stretch ratio to be $O(\log^2(n))$. This stretch is caused by errors in the embedding causing non-optimal routes to be taken. Even if the accuracy of the hyperbolic embedding fails due to unforeseen technical problems or active attack, the properties of the hyperbolic space ensure that the stretch factor is no worse than if nodes were connected randomly as in a traditional DHT.

The stretch ratio observed in existing DHTs is the number of hops required to complete a lookup. In practice, the distance between any successive hops in the lookup is expected to be the average inter-node distance, thus the expected average stretch is $O(\log_2(n))$ times the average inter-node distance.

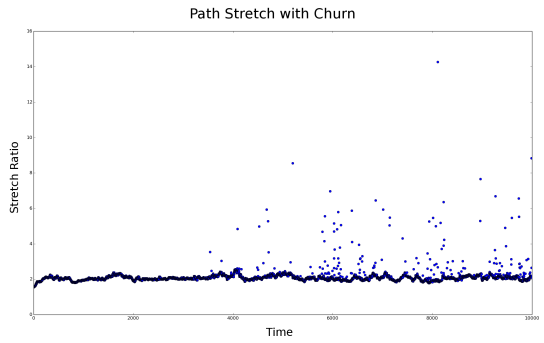


Figure 6.3: Here we see the stretch factor over time as nodes exit and join the network. Removal of central nodes can often require a short period of readjustment, but stretch remains stable over time.

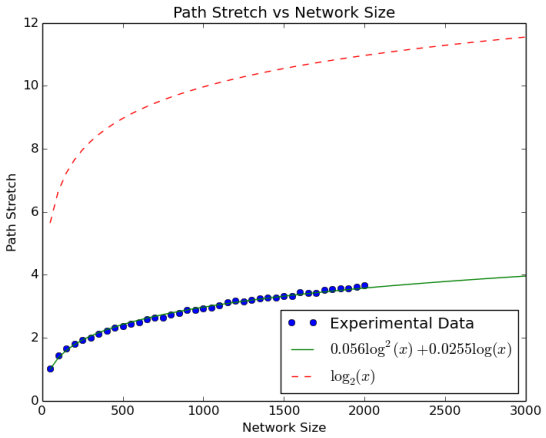


Figure 6.4: This data shows the experimental data points along with best fit curve and $\log_2(x)$ curve as comparison.

6.4.2 Congestion and Route Diversity

Problematically, the latency reduction provided by hyperbolic embedding has an inherent disadvantage [33]. This latency reduction comes from having the connectivity of the DHT congruent to the connectivity of the underlying scale free graph. The degree distribution and low number of central nodes in a scale free graph forces most routing paths through high degree central nodes.

While we cannot easily decrease the maintenance overhead due to high degree, we can manage congestion using a simple mechanism. Because DGVH maintains a list of "long-peers", (a size limited subset of all the "short-peers" of my "short-peers") every node connected to a central node has a random sampling of short-cuts across the network that bypass central nodes. Only when the long-peers fail to provide a reasonable alternative is a message routed to a higher centrality node. While this dramatically reduces the network throughput required for central nodes, it should still be expected that central nodes will have a higher throughput requirement than those on the periphery of the network, but less so than the concerns of previous works [33]. This indicates for future work reducing the diameter of the network reduces the overall work the system has to do and building short-cuts over high degree nodes reduces the concentration of congestion around central nodes.

Additional congestion avoidance behavior is trivial to implement because the DGVH greedy routing can effectively route around many holes. When a node is reaching congestion saturation, it can begin to respond to routing queries with a failure message, causing the forwarding algorithm to bypass the overloaded node. If the overloaded node is the only viable path to the destination, then the resulting loop that the packets follow will act as an ad-hoc buffer. The routing loop would storing and re-trying to send messages to congested peers until they can be accepted, effectively using the network as a memory in much the same sense as a mercury delay line [4].

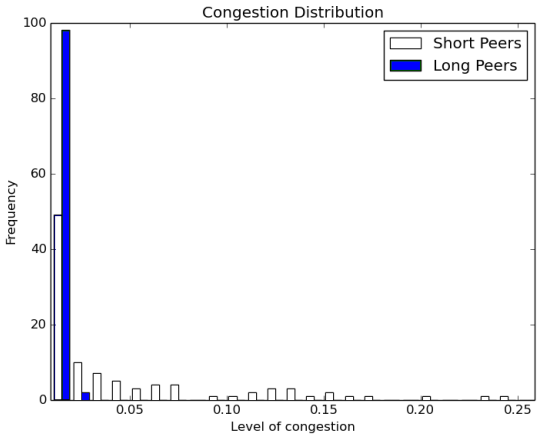


Figure 6.5: Using only short peers forces more nodes to take on higher congestion versus using long peers which globally reduces congestion.

6.5 Simulation

We simulate the greedy construction of a hyperbolic embedding and show that they produce very low latency stretch even as the network grows in size. Figure 6.4 shows how our stretch factor varies with network size. This data fits the curve $0.056 \log^2(x) + 0.0255 \log(x)$ with a r^2 value of .98. This shows that our stretch factor is likely $O(\log^2(n))$. For all DHT sizes less than 31,952,400,000 nodes, the best fit curve indicates our greedy hyperbolic embedding will remain more efficient than a traditional DHT. This greedy insertion method was intended to be a low sophistication first attempt to show that even low accuracy embedding results

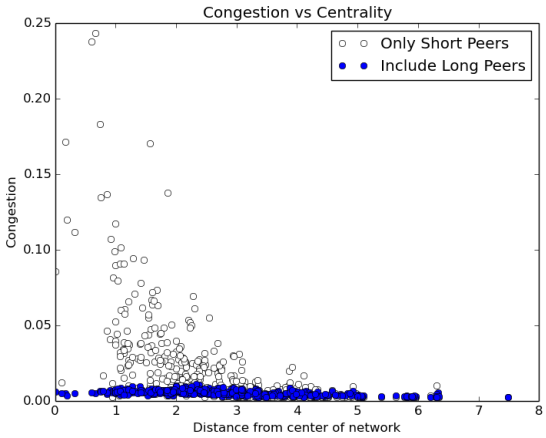


Figure 6.6: Here we see the congestion when using short peers only (approximating a scale free graph) and when utilizing randomly selected long peers. Using only short peers shows that central nodes handle disproportionate amounts of messages.

in dramatic improvements over the status quo.

Figure 6.3 shows the results when we simulate churn in a dynamic embedding. It shows that the embedding retains low latency stretch over time. We generate a 1000 node scale free graph and a size 100 overlay DHT.

For 10,000 iterations:

1. Randomly select a node from the overlay and remove it.

2. Randomly select a node from the 1000 node underlay network that is not currently being used on the overlay.

3. Greedily insert the chosen node into the network using the greedy joining method.

4. Record and log the average of all-to-all latency stretch.

The resulting network though churn has been totally replaced many times during the process of the simulations. While the quality of the simulation degrades initially from a stretch factor of 1.7, the stretch factor fluctuates around 2.0 as the simulation progresses. This shows that greedy insertion is effective at maintaining the embedding under churn.

Figure 6.6 and Figure 6.5 show the congestion in our simulated network. The values shown for congestion are calculated by finding the all-to-all greedy routed paths between nodes. The proportion of routes that pass through a given node describe how much congestion it has. For example, a congestion ratio of 0.5 indicates that half the routes in the network pass through a given node (and it should expect to see about half of all traffic).

Figure 6.6 shows, how without long peers the network is highly congested in centrally located nodes. Using long-hops most traffic can bypass the central nodes by using a long peer link to route over the potentially congested center of the network.

Figure 6.5 is a histogram that shows the distribution of load in the network. Without long-hops, most nodes have low congestion but a small amount of nodes have very high congestion. By adding long-hops, we lower the overall congestion and reduce the severity of congestion on the small portion of more central nodes.

6.6 Conclusions

We have shown that a decentralized algorithm for embedding overlay networks is possible by building a greedy traversable graph to augment a greedy embedding. This approach is possible because we work under different assumptions than previous work, as we are attempting to approximately embed a subset of a very large scale free graph rather attempting to perfectly embed the entire internet into a scale free graph. We have shown that even a poor approximation of hyperbolic embedding can dramatically decrease path stretch and that a finite amount of long-peers can dramatically reduce network congestion and disproportionate query load in the network.

Despite their poor latency optimization, DHTs have seen increasing adoption as organizational methods in new P2P systems. Application of this technique will increase the efficiency of systems already in common use and increase the viability of using a DHT to organize novel systems.

We consider application of this system a straightforward practice extending previous work

in DGVH [10], using hyperbolic embedding in a real system presents a new issue concerning methods of backing up DHT records. Previous DHTs could store replicas of data most efficently at points adjacent to the host in the DHT. Peers in the hyperbolic embedding are more likely to be physically close to one another and statistical assumptions concerning the independence of failure in adjacent nodes should be re-examined before implementation.

While we have shown that a non-optimal hyperbolic embedding brings dramatic improvement to latency reduction, future research should focus on more intelligent mechanisms for selecting an insertion location in the network. It is important to balance the capacity to select an optimal insertion location versus the amount of work required to do so in order to retain the ability of the network to scale.

Chapter 7

Conclusion

When I began to focus on DHTs early in my graduate program, I received reviews cautioning me against research into the area. It was considered a "mature" space where there was not room for further improvement. Naturally this commentary galvanized my resolve to further exploration of how DHTs could be applied and implemented.

Using ChordReduce [48], I have shown that DHTs can be utilized as a mechanism for organizing tasks other than data storage. As part of this, we found that DHTs have desirable properties for efficient one-to-all and all-to-one message passing. While the resulting software was overall inefficient (It was written in python by novice graduate students), it showed that with efficiency improvements DHTs could realistically be utilized to efficiently coordinate work on a global scale.

After we showed DHTs had strong applications, I explored methods for generalizing the behavior we saw in existing DHT mechanisms. Chord and Kademlia were generally considered totally distinct methods of accomplishing the features required of a DHT. Using DGVH and UrDHT I was able to show that DHTs all implemented specific instances of a more general behavior of owning a Voronoi Region and ensuring links along the Delaunay Triangulation. While implementing algorithms to calculate Voronoi Regions and Delaunay Triangulation in different distance metrics was difficult, DGVH provided a quick and effective method of doing so.

Using DGVH, we built UrDHT to prove our point with functional replicas of Chord and Kademlia using our abstracted logic. UrDHT has show itself to have value well beyond being

an experimental testbed for this dissertation. I look forward to seeing what the open source community builds using UrDHT, which already boasts features never considered in this work because it provides the capacity for others to explore variations on DHT behavior.

When implementing DHTs for practical purposes, especially ChordReduce, it made sense to explore mechanisms for improving robustness. The idea of the Reactive K-Nearest backup strategy was initially explored in ChordReduce and once we had a more general model for discussing DHTs it made sense to discuss the efficacy of replication strategies. Discussing DHT reliability in terms of Half-lives provides a method of accurately discussing the robustness of DHTs and similar systems against data loss. I proposed the Reactive K-Nearest backup strategy as an effective mechanism for robustness that has the potential to change how DHTs are treated in terms of reliability. Reactive K-Nearest provides a way for DHTs to go from being considered low reliability storage to high reliability and dramatically expands DHT's use cases.

Where before, reliably storing a record for any period of time without was not a viable options, Reactive K-nearest allows records to be stored at very high reliability. This provides the potential for a decentralized robust public data store that could be expected to be usable in the longer term of human development. Future work should similarly consider the reality that techniques and machines built by computer engineers and computer scientists soon need to stop being disposable. We have to begin designing and implementing infrastructure and tools in both software and hardware that will last to be utilized, maintained and understood by our ancestors.

A fundamental concern I had when first implementing and utilizing DHTs was their tendency to send a query's path waltzing back and forth across the face of the earth. During testing of ChordReduce, we linked two large populations of nodes in different Amazon AWS [3] regions, we found that a query could bounce back and forth across the country between Virgina and Washington state many times before finally resolving to be a in the local cluster and the time it spent doing that dramatically impacted our performance. Minimizing latency

while also preserving the scalability of a DHT was a challenge that proved intractable until the capacity to generalize DHTs to operate in any metric came in the form of DGVH and UrDHT.

Previous work by Robert Kleinberg [32] considered the possibility of using a hyperbolic space to minimize latency, but lacked a method of usefully embedding network latency and ensuring topology maintenance in the hyperbolic space. As I had just invented a method of topology maintenance that I had shown functioned well in hyperbolic space it seemed suitable to apply it to Kleinberg's open problem. While the greedy insertion method I present is clearly producing poorer quality embeddings than Kleinberg envisioned, it has a clear capacity to reduce the DHT queries latency at all the network scales that I expect will be witnessed by humans for the next few generations.

After considered a DHT's capacity to be a long-term surviving system, I have grown increasingly motivated by the idea that P2P systems can provide services like messaging, file sharing, and distributed computing with an expectation of continued service far beyond that of what any private organization could provide. A P2P system based on a DHT will always continue to function so long as there is public interest in supporting it. I believe that while it may be a slow process, the world's people will select for such persistent services and that they will only increase in usage and utility with time.

Bibliography

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

[2] Virtual hadoop. http://wiki.apache.org/hadoop/Virtual

[3] Amazon.com. Amazon EC2 Instances. http://aws.amazon.com/ec2/instance-types.

[4] Isaac L Auerbach, J Presper Eckert Jr, Robert F Shaw, and C Bradford Sheppard. Mercury delay line memory using a pulse rate of several megacycles. *Proceedings of the IRE*, 37(8):855–861, 1949.

[5] Franz Aurenhammer. Voronoi diagramsa survey of a fundamental geometric data structure. *ACM Computing Surveys (CSUR)*, 23(3):345–405, 1991.

[6] Helge Backhaus and Stefan Krause. Voronoi-based adaptive scalable transfer revisited: Gain and loss of a voronoi-based peer-to-peer approach for mmog. In *Proceedings of the 6th ACM SIGCOMM Workshop on Network and System Support for Games*, NetGames '07, pages 49–54, New York, NY, USA, 2007. ACM.

[7] Olivier Beaumont, A-M Kermarrec, Loris Marchal, and Etienne Rivière. Voronet: A scalable object network based on voronoi tessellations. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–10. IEEE, 2007.

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

[9] Juan Benet. Ipfs-content addressed, versioned, p2p file system. *arXiv preprint arXiv:1407.3561*, 2014.

[10] Brendan Benshoof, Andrew Rosen, Anu G. Bourgeois, and Robert W Harrison. A distributed greedy heuristic for computing voronoi tessellations with applications towards peer-to-peer networks. In *Dependable Parallel, Distributed and Network-Centric Systems, 20th IEEE Workshop on*.

[11] Marshall Bern, David Eppstein, and Frances Yao. The expected extremes in a delaunay triangulation. *International Journal of Computational Geometry & Applications*, 1(01):79–91, 1991.

[12] Béla Bollobás and Oliver Riordan. The diameter of a scale-free random graph. *Combinatorica*, 24(1):5–34, 2004.

[13] Dhruba Borthakur. The Hadoop Distributed File System: Architecture and Design. 2007.

[14] Eric Brewer. A certain freedom: thoughts on the cap theorem. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 335–335. ACM, 2010.

[15] Bogdan Carbunar, Ananth Grama, and Jan Vitek. Distributed and dynamic voronoi overlays for coverage detection and distributed hash tables in ad-hoc networks. In *Parallel and Distributed Systems, 2004. ICPADS 2004. Proceedings. Tenth International Conference on*, pages 549–556. IEEE, 2004.

[16] Xinyu Chen, Michael R Lyu, and Ping Guo. Voronoi-based sleeping configuration in wireless sensor networks with location error. In *Networking, Sensing and Control, 2008. ICNSC 2008. IEEE International Conference on*, pages 1459–1464. IEEE, 2008.

[17] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Designing Privacy Enhancing Technologies*, pages 46–66. Springer, 2001.

[18] Bram Cohen. The bittorrent protocol specification, 2008.

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

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

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

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

[23] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 29–43. ACM, 2003.

[24] Li Gong. JXTA: A Network Programming Environment. *Internet Computing, IEEE*, 5(3):88–95, 2001.

[25] Hal Hodson. Meshnet activists rebuilding the internet from scratch. *New Scientist*, 219(2929):20–21, 2013.

[26] Shun-Yun Hu, Shao-Chen Chang, and Jehn-Ruey Jiang. Voronoi state management for peer-to-peer massively multiplayer online games. In *Consumer Communications and Networking Conference, 2008. CCNC 2008. 5th IEEE*, pages 1134–1138. IEEE, 2008.

[27] Shun-Yun Hu and Guan-Ming Liao. Scalable peer-to-peer networked virtual environment. 2004.

[28] H Jansen. Abbildung der hyperbolischen geometrie auf ein zweischaliges hyperboloid. *Mitt. Math. Gesellsch. Hamburg*, 4:409–440, 1909.

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

[30] Raúl Jimenez. Kademlia on the open internet: How to achieve sub-second lookups in a multimillion-node dht overlay. 2011.

[31] Brad Nelson Karp. *Geographic routing for wireless networks.* PhD thesis, Harvard University Cambridge, Massachusetts, 2000.

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

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

[34] Kyong-Ha Lee, Yoon-Joon Lee, Hyunsik Choi, Yon Dohn Chung, and Bongki Moon. Parallel data processing with mapreduce: a survey. *ACM SIGMOD Record*, 40(4):11–20, 2012.

[35] Kyungyong Lee, Tae Woong Choi, A. Ganguly, D.I. Wolinsky, P.O. Boykin, and R. Figueiredo. Parallel Processing Framework on a P2P System Using Map and Reduce Primitives. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 1602–1609, 2011.

[36] Tonglin Li, Xiaobing Zhou, Kevin Brandstatter, Dongfang Zhao, Ke Wang, Adithya Rajendran, Zhao Zhang, and Ioan Raicu. Zht: A light-weight reliable persistent dynamic scalable zero-hop distributed hash table. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 775–787. IEEE, 2013.

[37] Boon Thau Loo, Tyson Condie, Joseph M Hellerstein, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Implementing declarative overlays. *ACM SIGOPS Operating Systems Review*, 39(5):75–90, 2005.

[38] Gurmeet Singh Manku, Mayank Bawa, Prabhakar Raghavan, et al. Symphony: Distributed Hashing in a Small World. In *USENIX Symposium on Internet Technologies and Systems*, page 10, 2003.

[39] Gurmeet Singh Manku, Mayank Bawa, Prabhakar Raghavan, et al. Symphony: Distributed hashing in a small world. In *USENIX Symposium on Internet Technologies and Systems*, page 10, 2003.

[40] Dave Longley Manu Sporny. A decentralized hashtable for the web. Draft community group report, opencreds, February 2016. http://opencreds.org/specs/source/webdht/.

[41] Fabrizio Marozzo, Domenico Talia, and Paolo Trunfio. P2P-MapReduce: Parallel Data Processing in Dynamic Cloud Environments. *Journal of Computer and System Sciences*, 78(5):1382–1402, 2012.

[42] Petar Maymounkov and David Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. In *Peer-to-Peer Systems*, pages 53–65. Springer, 2002.

[43] Christos H Papadimitriou and David Ratajczak. On a conjecture related to geometric routing. In *Algorithmic Aspects of Wireless Sensor Networks*, pages 9–17. Springer, 2004.

[44] Fragkiskos Papadopoulos, Dmitri Krioukov, M Bogua, and Amin Vahdat. Greedy forwarding in dynamic scale-free networks embedded in hyperbolic metric spaces. In *INFOCOM, 2010 Proceedings IEEE*, pages 1–9. IEEE, 2010.

[45] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J Abadi, David J DeWitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches to large-scale data

analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 165–178. ACM, 2009.

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

[47] Sylvia Ratnasamy, Brad Karp, Li Yin, Fang Yu, Deborah Estrin, Ramesh Govindan, and Scott Shenker. Ght: a geographic hash table for data-centric storage. In *Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, pages 78–87. ACM, 2002.

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

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

[50] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware 2001*, pages 329–350. Springer, 2001.

[51] Salvatore Sanfilippo and Pieter Noordhuis. Redis, 2009.

[52] Haiying Shen and Cheng-Zhong Xu. Locality-Aware and Churn-Resilient Load-Balancing Algorithms in Structured Peer-to-Peer Networks. *Parallel and Distributed Systems, IEEE Transactions on*, 18(6):849–862, 2007.

[53] Jonathan Richard Shewchuk. Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. In Ming C. Lin and Dinesh Manocha, editors, *Applied Computational Geometry: Towards Geometric Engineering*, volume 1148 of *Lecture Notes*

in Computer Science, pages 203–222. Springer-Verlag, May 1996. From the First ACM Workshop on Applied Computational Geometry.

[54] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10. IEEE, 2010.

[55] Konstantin V Shvachko. HDFS Scalability: The Limits to Growth. *login*, 35(2):6–16, 2010.

[56] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. 31:149–160, August 2001.

[57] Juan Pablo Timpanaro, Thibault Cholez, Isabelle Chrisment, and Olivier Festor. Bittorrent's mainline dht security assessment. In *New Technologies, Mobility and Security (NTMS), 2011 4th IFIP International Conference on*, pages 1–5. IEEE, 2011.

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

[59] Liang Wang and Jussi Kangasharju. Measuring large-scale distributed systems: case of bittorrent mainline dht. In *Peer-to-Peer Computing (P2P), 2013 IEEE Thirteenth International Conference on*, pages 1–10. IEEE, 2013.

[60] Wei Wang, Guisong Yang, Naixue Xiong, Xingyu He, and Wenzhong Guo. A general p2p scheme for constructing large-scale virtual environments. In *Parallel Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pages 1648–1655, May 2014.

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

[62] Bassam Zantout and Ramzi Haraty. I2p data communication system. In *Proceedings of the tenth international conference on networks*, pages 401–409, 2011.

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

[64] Ben Y Zhao, Ling Huang, Jeremy Stribling, Sean C Rhea, Anthony D Joseph, and John D Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *Selected Areas in Communications, IEEE Journal on*, 22(1):41–53, 2004.