BUILDING ROBUST DISTRIBUTED INFRASTRUCTURE NETWORKS

by

Brendan Lane Benshoof

Under the Direction of Dr. Robert Harrison, PhD

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 Delunay triangulation in a given metric space to maintain the networks topology. We will show that utilizing this model, we can produce network typologies that approximate existing DHT methods and provide a starting point for further exploration. Previous research has explored utilizing a hyperbolic metric space to provide efficient greedy routing for overlay networks. We will use our generalized model of DHT construction to design and implement more efficient distributed hash table protocol, and discuss the qualities of potential successors to existing DHT technologies.

TBUILDING 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 Computer Science

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

Dedicated to the people who I owe my success, to be enumerated in the future.

Acknowledgements

A lot of people helped me do all of this. I should stop pretending I did it all for a second and mention them.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 The Problem

I am setting forth to solve two problems in the space of decentralized services. That such services compete for infrastructures and provide high barriers to adoption even for those technically inclined and that current decentralized service overlay topologies are wasteful in management fo latency on congestion.

Most p2p systems require/leverage a core infrastructure for peer selection and maintaining shared state. The most commonly used such system is a Distributed Hash Table[**?**]. It allows networks to discover other peers in an organized fashion and to provide the network a shared "key-value" store. Because many networks implement and run their own DHT networks and protocols, they cannot benefit from a shared infrastructure beyond that of the Internet itself. This is reasonable cost because each network has differing use cases and makes different assumptions about the capabilities of every node in its own network.

Current technologies utilize network topologies built on metric spaces intended to facilitate a O(log(n)) diameter, with the partially correct assumption that this will also minimize the amount of time required to preform a lookup on the network. Because these topologies are constructed without regard to the underlying network, they often provide grossly

inefficient paths through the underlay network when following a short overlay network path.

1.2 My Proposal

While the problems presented are not strictly connected, they are bothing being considered because they share a solution. Changing the foundational assumptions of distributed hash tables give us opportunity to change their practical and perceived utility. I intend to research, propose, and develop a successor technology to current 'Distributed Hash Tables' (Kademlia, Chord, CAN) in the form of a 'Decentralized Infrastructure Network' (working name). This technology will provide increased performance, facilitate the use of current and new p2p distributed systems, and provide a distributed infrastructure for new p2p client-side applications in Server, PC, Browser, and Mobile environments.

This successor technology would be similar to current Distributed Hash tables in many ways: It will build a p2p overlay routing network, it will utilize a 'metric space' and hash functions to assign locations to nodes and records, it will Utilize a similar but simplified network protocol and maintenance cycle.

The important differences from established will be:

- Differentiation between 'Clients' and 'Servers' which allows users to utilize the network without contributing
- Abstraction of the 'Peer selection metrics' which will allow us to research and examine optimal solutions to our use case.
- Support for service specific 'subnetworks' that will allow existing p2p systems to leverage the DIN.
- Provide a multiple-use 'reusable' network infrastructure to allow for easier entry for software developers into the p2p distributed systems space.

- Utilize the abstract 'metrics' to minimize lookup latency while preserving low maintenance costs, particularly Hyperbolic metrics.

Chapter 2

Technical Background

2.1 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 concensus provide higher confidance, 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 system's performance are bound by the CAP Theorem[11].

2.1.1 CAP Theorem/ Brewers Theorem

CAP Theorem describes a trade-off between three Attributes of distributed systems. It states, that any distributed system is a compromise between consistency, availability and partition tolerance and that no distributed store of state can posses all of these qualities at a time.

- Consistency: Everybody agrees on the contents of shared data
- Availability: Everybody can quickly and consistently access all the data.

- Partition Tolerance: The network can handle loss of nodes and connectivity between nodes without loosing the system's cohesion or data.

As partition tolerance is required in a DHT for long term operation, this leaves us a trade-off between Consistency and Availability. In DHTs consistency is limited to assure availability of records to users, however such trade-offs are not binary, and we can can seek a balance between availability and consistency to best suit the needs of applications.

2.2 What are the currently existing DHTs?

Chord and Kademlia 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 it's consistency in the face of churn. Sadly these proofs have been shown to be incorrect[] without serious modification to the established protocol.

2.2.1 Kademlia

Kademlia is 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 it's 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-eucldian 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 it's metric, Kademlia uses a modified k-nearest neighbors approach to approximate node's voronoi regions and Delaney peers. If nodes are evenly distributed through the space, kademlia's metric provides an $O(log(n))$ diameter

network.

2.2.2 Chord

Chord is a familiy of ring-based DHT's. Locations are represented by a large integer similar to kademlia. The metric is a unidirectional (bidirectional in some variants) modulus ring.

Chord tracks the immediate peers in either direction on this ring to maintain the networks and calculating delaunay triangulation and voronoi regions in this metric is trivial.

This metric alone would give chord's topology an $O(n)$ diameter and to mitigate this, each node maintains $O(log(n))$ "fingers" distributed in such a way that the diameter of the network is reduced to $O(log(n))$.

2.3 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: Bitorrent[24], CJDNS[21], and I2P[45]

Chapter 3

MapReduce on a Chord Distributed Hash Table

Andrew Rosen Brendan Benshoof Robert W. Harrison Anu G. Bourgeois

3.1 Introduction

Google's MapReduce [17] 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 [17].

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[40] 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-2-peer frameworks to datacenters. Our framework cannot and does not rely on many common assumptions, such 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 [42], a peer-2-peer distributed hash table, as the backbone for developing our system. Our paper 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 III).

- 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 IV).

- 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 V).

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

3.2 Background

ChordReduce takes its name from the two components it is built upon. Chord [42] provides the backbone for the network and the file system, providing scalable routing, distributed storage, and fault-tolerance. MapReduce runs on top of the Chord network and utilizes the underlying features of the distributed hash table. This section provides background on Chord and MapReduce.

3.2.1 Chord

Chord [42] 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 the network. 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 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 or fail, the most immediate successor would be responsible for the content its content.

Each node maintains a table of m shortcuts to other peers, called the finger table. The

ith entry of a node n's finger table corresponds to the node that is the successor of the key $n + 2^{i-1} \mod 2^m$. 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 [42].

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 maintenance cycle are beyond the scope of this paper and can be found here [42].

3.2.2 MapReduce

At its core, MapReduce [17] 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.

Insert psuedocode or picture, code for word count here!

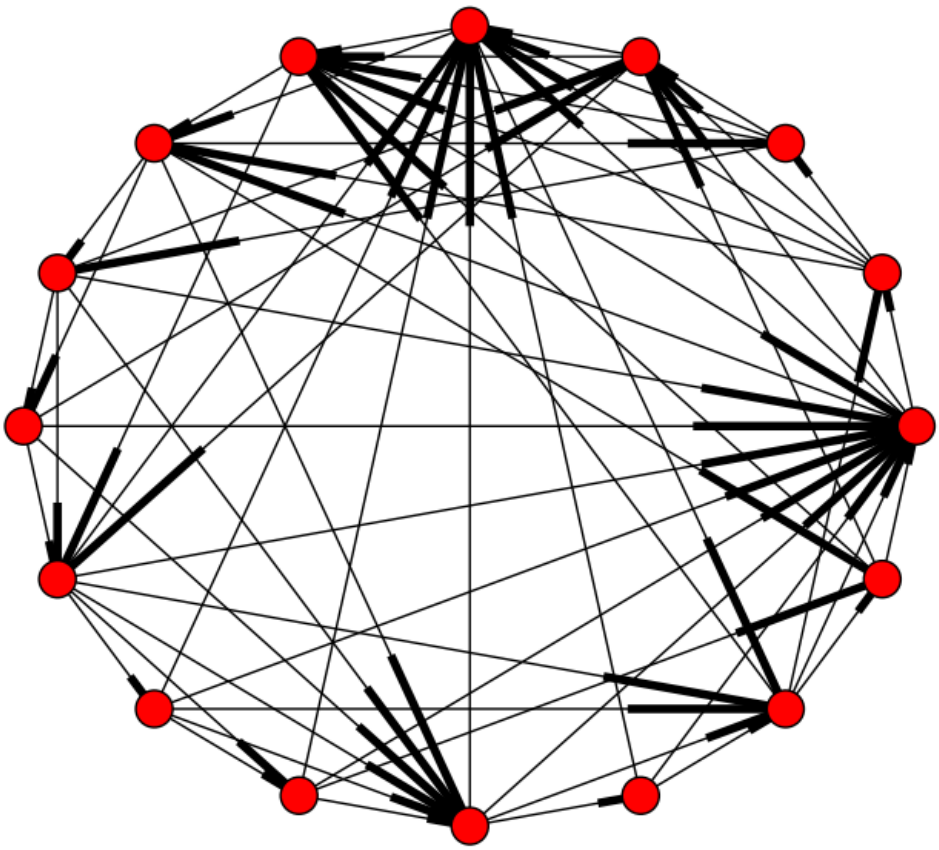


Figure 3.1: A Chord ring 16 nodes where $m = 4$. The bold lines are incoming edges. Each node has a connection to its successor, as well as 4 fingers, some of which are duplicates.

The most popular platform for MapReduce is Hadoop [1]. Hadoop is an open-source Java implementation developed by Apache and Yahoo! [33]. Hadoop has two components, the Hadoop Distributed File System (HDFS) [10] and the Hadoop MapReduce Framework [27]. 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 [40] in the network, as well as a potential bottleneck for the system [41].

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

3.3 ChordReduce

While the subsections in this section are complete, this intro is a mess of cut and pasted text 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

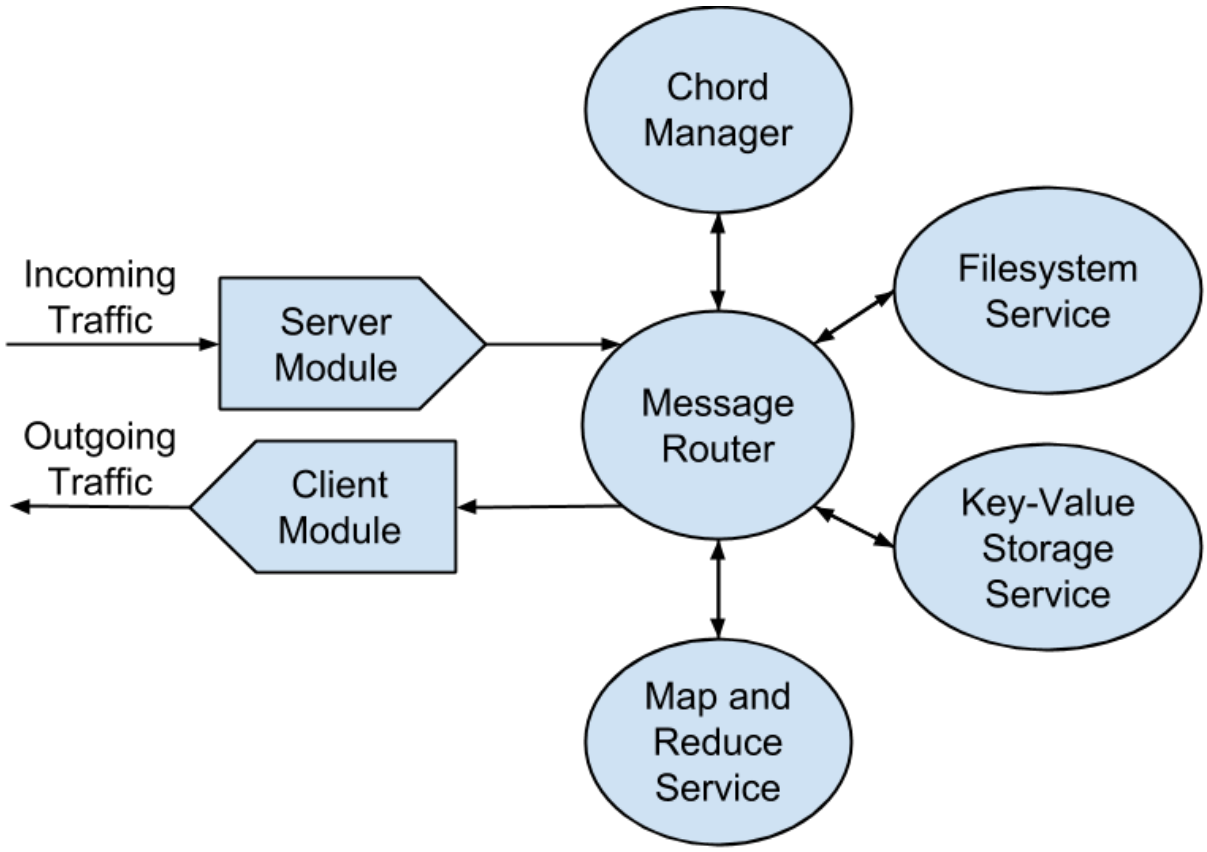


Figure 3.2: The basic architecture of a node in ChordReduce. MapReduce runs as a service on top of each node.

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.

[28] says that latency is similar. Marozzo et al. [31] 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.

insert architecture layer diagram and *insert work flow diagram. Chord tree of stage, map , reduce.*

3.3.1 File Storage

The design of a distributed file system is closely tied to the design to the implementation of MapReduce [19] [10]. Our system uses CFS [16], 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[16]. 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 it's 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 the files' block 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.

3.3.2 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 divide the work into *data atoms*, which define the smallest individual units that work can be done on. This might represent 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 it's 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, create data atoms specified by the user stage function 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.

3.3.3 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 nodes 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 to 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 fail, 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 *I think the citation is CFS.*

A final consequence of this is load-balancing during runtime. As new nodes enter the network, they change their successor as the maintenance ycle 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.

3.4 Experiments

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

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

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

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

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

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

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

3.4.1 Experimental Deployment

We built a fully functional implementation of ChordReduce in Python. Our implementation implements all the routing and maintenance procedures defined by Chord [42], the file storage capabilities of CFS [16], and a MapReduce service built ontop 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 'instrumentation 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 roll is to aid in the generation and collection of data.

First, the instrumentation node's 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 instrumentation node as part of the experiment.

3.4.2 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

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

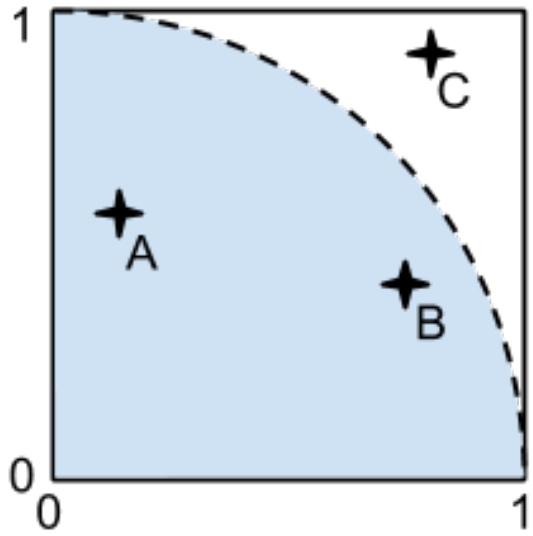


Figure 3.3: The "dartboard." The computer throws a dart by choosing a random x and y between 0 and 1. If $x^2 + y^2 < 1^2$, the dart landed inside the circle. A and B are darts that landed inside the circle, while C did not.

top-right quarter of a circle going through it (Fig. 3.3), 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, 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

[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.

on the Chord network using CFS [16]. *We created word frequency counts over a giant list of text from Project Gutenberg. There are going to be 3 corpus's of text, large enough to show scaling, not so large as to take forever.* 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 their specified blocks. Those results are reduced by simply combining the word frequency tables.

3.5 Results

For pi when we varied x, y happened. Here's a paragraph an a pretty picture. Do this a couple more times and you have a results section For wordcount, when we varied the blocksize we found this

Fig. 3.4 and Fig. 3.5 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. 3.5 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

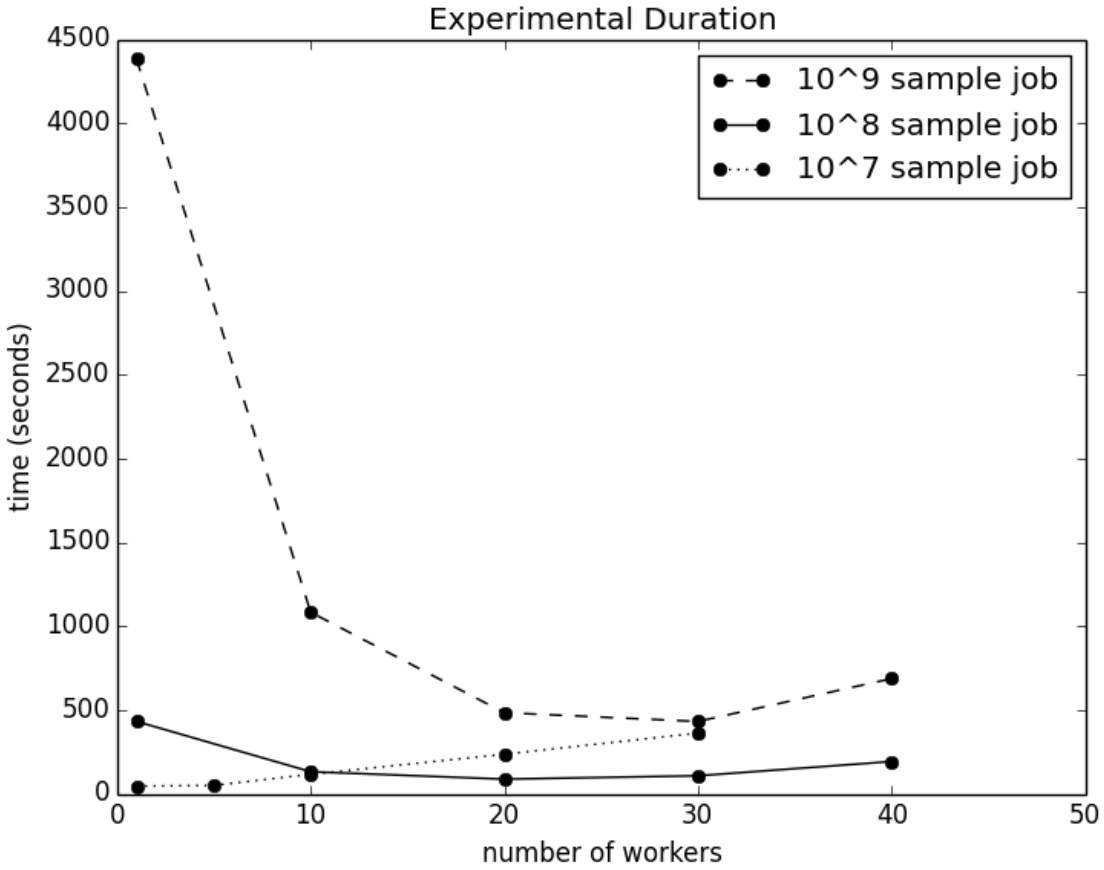


Figure 3.4: For a sufficiently large job, it was almost always preferable to distribute it. When the job is too small, such as with the 10^7 data set, our runtime is dominated by the overhead. Our results are what we would expect when overhead grows logarithmically to the number of workers.

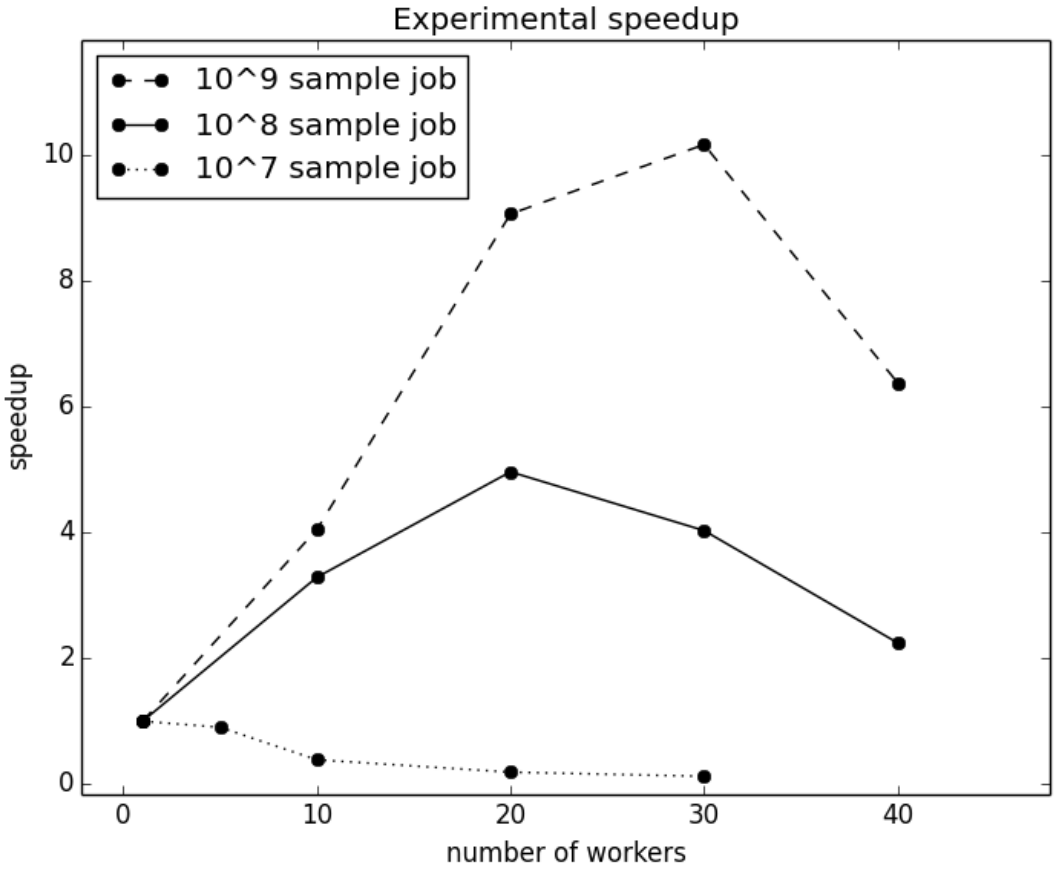


Figure 3.5: The larger the size of the job, the greater the gains of distributing with ChordReduce. In addition, the larger the job, the more workers can be added before we start seeing diminishing returns. This demonstrates that ChordReduce is scalable.

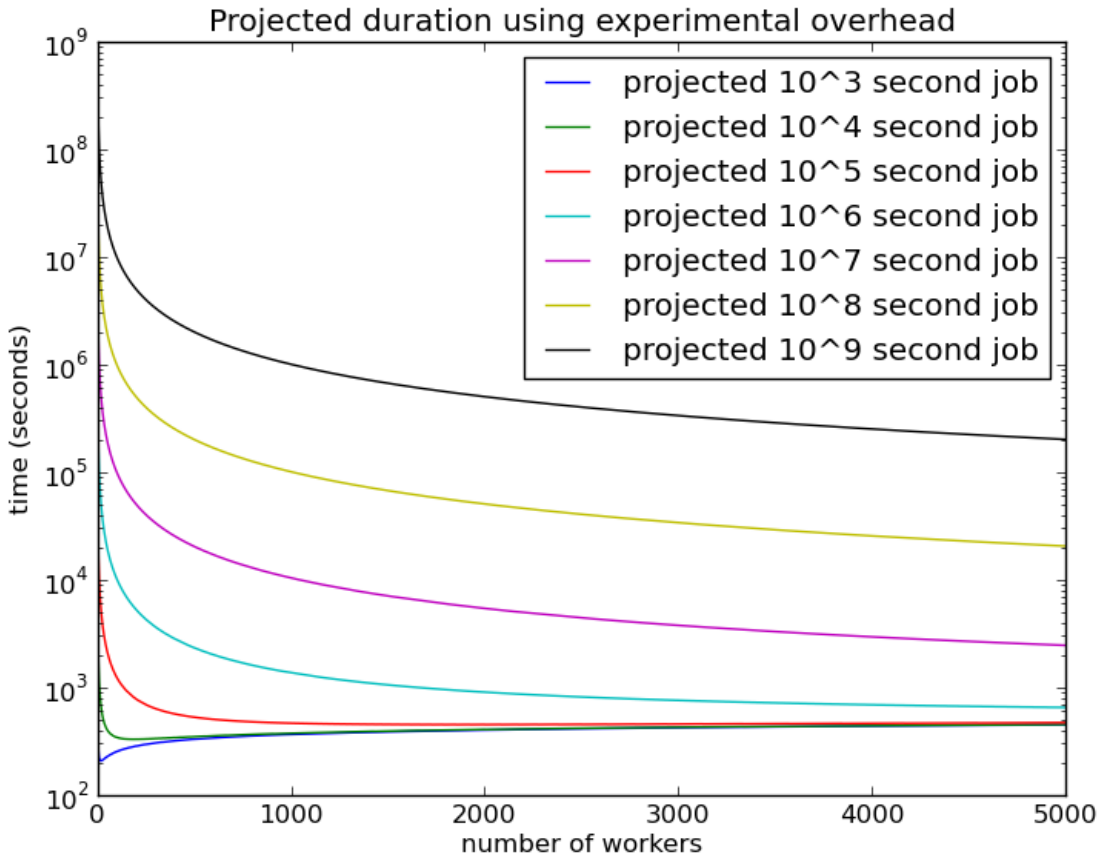


Figure 3.6: The projected runtime using ChordReduce for differently sized jobs. Each curve projects the expected behavior for job that takes a single worker the specified amount of time.

addition, the gains of distributing work further increased at 30 workers and only began to decay at 40 workers, compared with the 10^8 data set, which began its drop off at 30 workers. This behavior demonstrates that the larger the job being distributed, the greater the gains of distributing the work using ChordReduce.

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

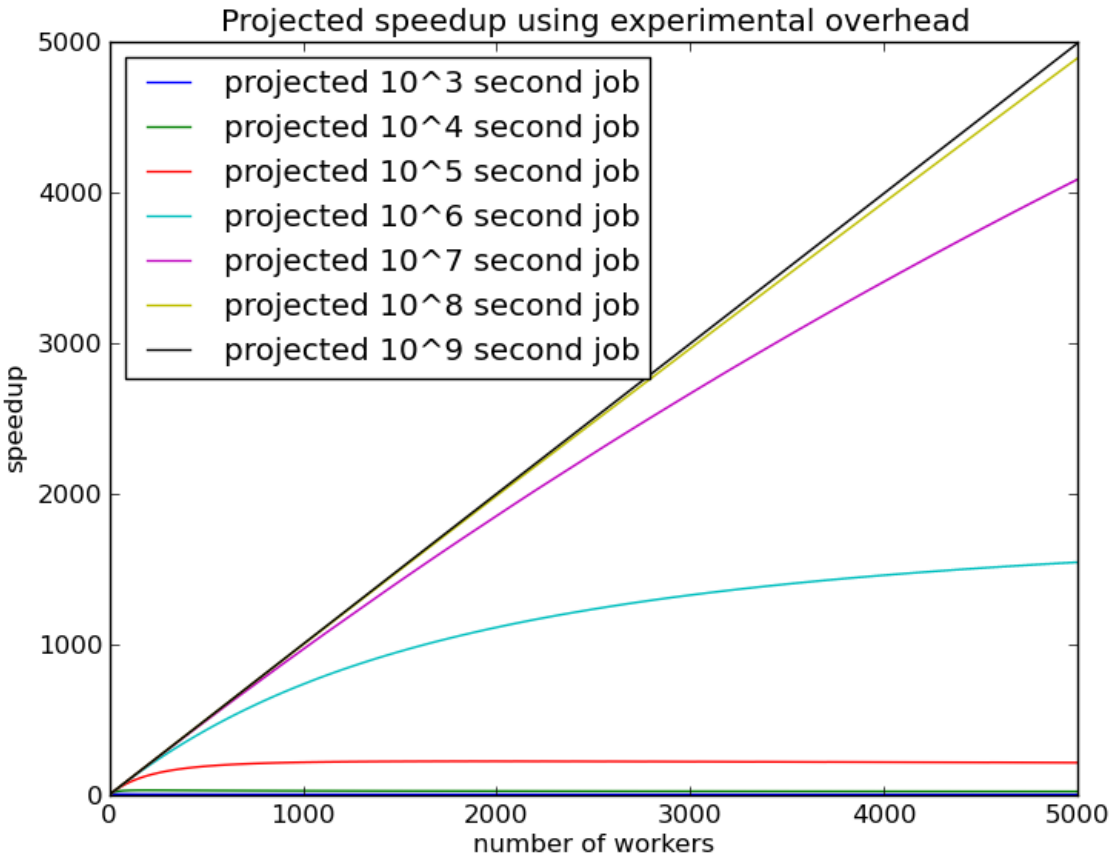


Figure 3.7: The projected speedup for different sized jobs.

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 3.1

Since we have now established that $T_n = \frac{T_1}{n} + k \cdot \log_2(n)$, we can estimate how long a job that takes an arbitrary amount of time to run on a single node would take using ChordReduce. Our data points indicated that the mean value of k for this problem was 36.5. Fig. 3.6 shows that for jobs that would take more than 10^4 seconds for single worker to complete, we can expect there would still be benefit to adding an additional worker, even when there are already 5000 workers already in the ring. Fig. 3.7 further emphasizes this. Note that as the jobs become larger, the expected speedup from ChordReduce approaches linear behavior.

Table 3.1 shows the experimental results for different rates of churn. These results show the system is relatively insensitive to churn. We started with 40 nodes in the ring and generated 10^8 samples while experiencing different rates of churn, as specified in Table 3.1. At the 0.8% rate of churn, there is a 0.8% chance each second that any given node will leave the network followed by another node joining the network at a different location. The joining rate and leaving rate being identical is not an unusual assumption to make [31] [38].

Our testing rates for churn are an order of magnitude higher than the rates used in the P2P-MapReduce simulation [31]. In their paper, the highest rate of churn was only 0.4% per minute. Because we were dealing with fewer nodes, we chose larger rates to demonstrate that ChordReduce could effectively handle a high level of churn.

Our experiments show that for a given problem, ChordReduce can effectively distribute the problem, yielding a substantial speedup. Furthermore, our results showed that the larger the problem is, the more workers could be added before diminishing returns were incurred.

During runtime, we experienced multiple instances where *plot* would fail to run and the stager would report socket errors, indicating that it had lost connection with a node in the ring. Despite this turbulence, every node managed to reestablish connection with each other and report back all the data. This further demonstrated that we were able to handle the churn in the network.

3.6 Related Work

Marozzo et al. [31] 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 [20] 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 [31].

Lee et al.'s work [28] 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 [29], 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.

3.7 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.

Here are examples of future work that we plan on doing with ChordReduce. We could do X, which is cool because duh. Y is another application of this, possibly a consequence of X.

Chapter 4

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

Brendan Benshoof Andrew Rosen Anu G. Bourgeois Robert W. Harrison

Published DPDNS Workshop IPDPS 2015

4.1 Introduction

Voronoi diagrams [4] 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 [43], or for in coverage detection for wireless networks [12]. Additionally, Massively Multiplayer Online games (MMOs) can use them to distribute game states and events between players at a large scale [23] [22] [5].

Computing the Voronoi tessellation along with its coprime problem, Delaunay Trian-

gulation, 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 [18]. However, many network applications are distributed and many of the algorithms to compute Voronoi tessellations are unsuited to a distributed environment.

In addition, complications occurs 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 [44]. Distributed computations often have to resort to costly Monte-Carlo calculations [7] 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 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. Our paper 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 5.2.2). Even where small inaccuracies exist, DVGH will create a fully connected graph.

- We discuss how P2P networks and distributed applications can use DGVH (Section 4.3). 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 near 100% accuracy by cycle 30 (Section 5.5).

- We present the related work we have built upon to create our heuristic and what improvements we made with DGVH (Section 5.6).

4.2 Distributed Greedy Voronoi Heuristic

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

The Voronoi tessellation and Delaunay triangulation are dual problems, as an edge between two objects in a Delaunay triangulation exists if and only if those object's Voronoi regions border each other. This means that solving either problem will yield the solution to both. An example Voronoi diagram is shown in Figure 5.2. For additional information, Aurenhammer [4] provides a formal and extremely thorough description of Voronoi tessellations, as well as their applications.

4.2.1 Our Heuristic

The Distributed Greedy Voronoi Heuristic (DGVH) is a fast method for nodes to define their individual Voronoi region (Algorithm 5.3). 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.

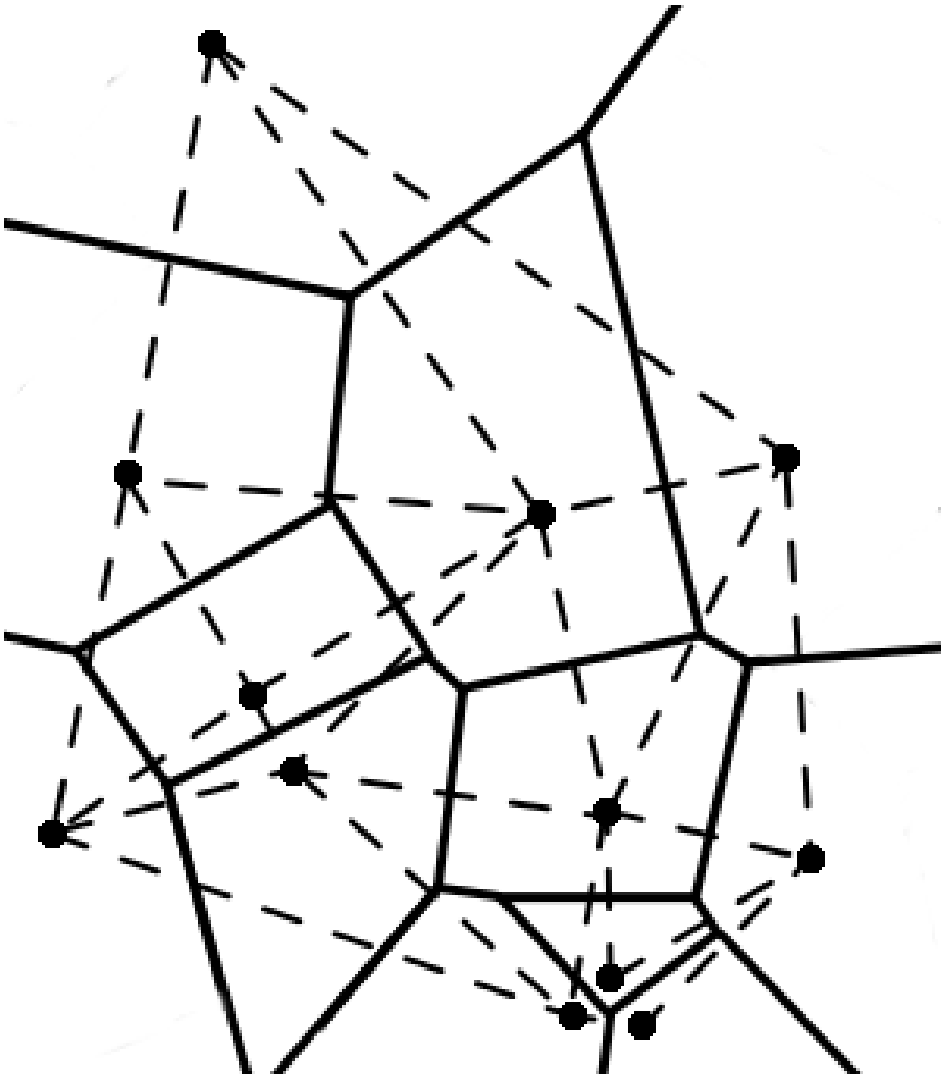


Figure 4.1: An example Voronoi diagram for objects on a 2-dimensional space. The black lines correspond to the borders of the Voronoi region, while the dashed lines correspond to the edges of the Delaunay Triangulation.

Figure 4.2: Distributed Greedy Voronoi Heuristic

1: Given node n and its list of *candidates*.
2: Given the minimum *table_size*
3: *short_peers* $\leftarrow$ empty set that will contain n's one-hop peers
4: *long_peers* $\leftarrow$ empty set that will contain n's 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: m is the midpoint between n and c
9: **if** Any node in *short_peers* is closer to m than n **then**
10: Reject c as a peer
11: **else**
12: Remove c from *candidates*
13: Add c to *short_peers*
14: **end if**
15: **end for**
16: **while** $|short\_peers| < table\_size$ AND $|candidates| > 0$ **do**
17: Remove the first entry c from *candidates*
18: Add c to *short_peers*
19: **end while**
20: Add *candidates* to the set of *long_peers*
21: **if** $|long\_peers| > table\_size^2$ **then**
22: *long_peers* $\leftarrow$ random subset of *long_peers* of size $table\_size^2$
23: **end if**

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 [7]. 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 paper 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:

$$distance = \sqrt{\sum_{i \in d} (\min(|\vec{a}_i - \vec{b}_i|, 1 - |\vec{a}_i - \vec{b}_i|))^2}$$

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 4.3. Our algorithm handles these cases via our method of peer management (Section 4.2.2).

4.2.2 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 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.* [7] 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.* [9] 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 [**?**] [32].

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

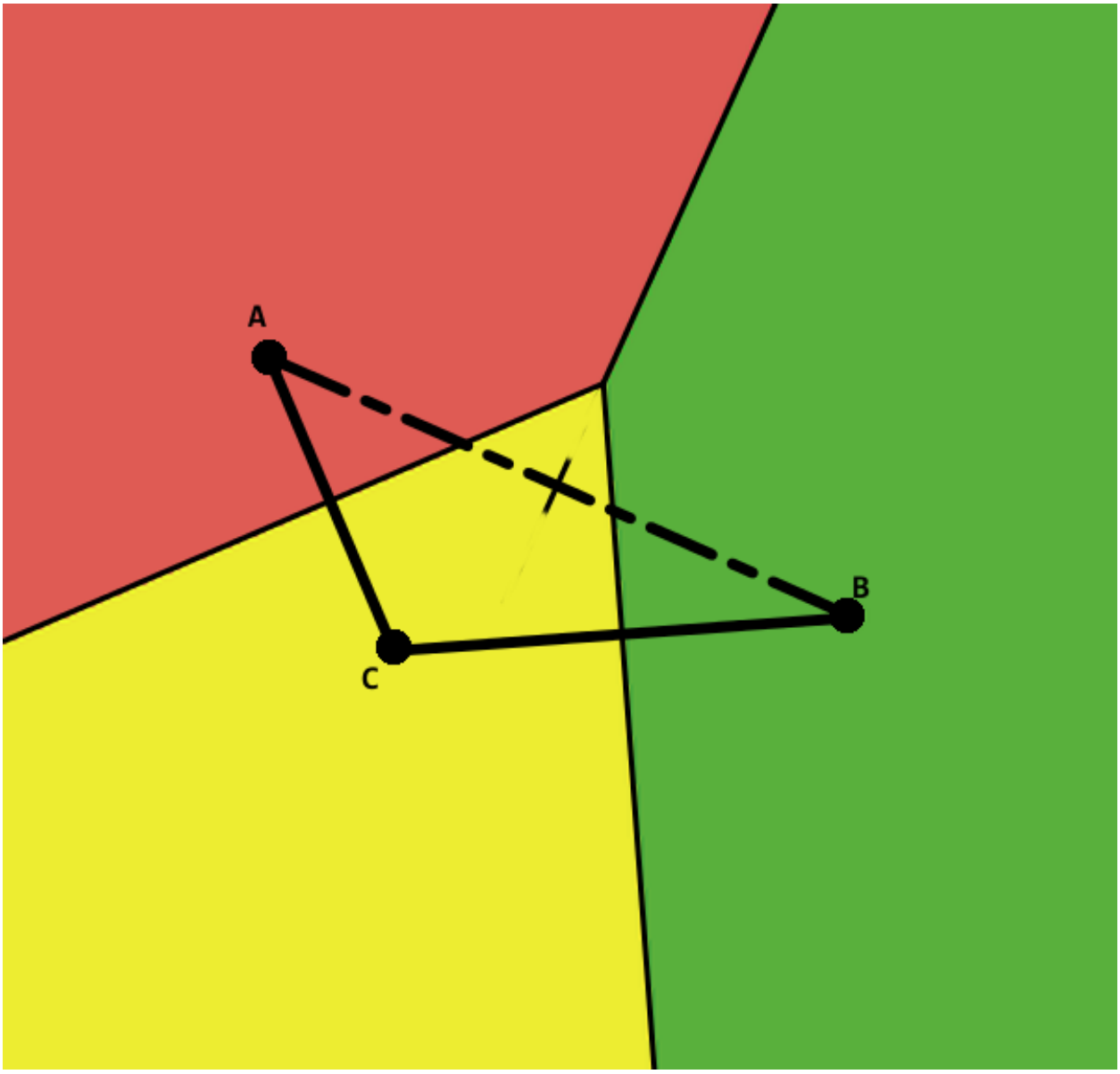


Figure 4.3: The edge between A and B is not detected by DGVH, as node C is closer to the midpoint than B is. This is mitigated by peer management polices.

Figure 4.4: 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 $candidates$

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 Algorithm 4.4. This maintenance through gossip process is very similar to the gossip protocol used in Beaumont et al.'s RayNet [7].

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

4.2.3 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, 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 $\Theta(\frac{\log N}{\log \log N})$ [9] (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 [7] 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.

4.3 Applications

As we previously discussed in Section 4.1, Voronoi tessellation have many applications for distributed systems [12] [23] [22] [5]. We focus our discussion on the two extremes of applications: DHTs, which work with overlay networks, and wireless networks, which need to take literal physical constraints into account.

Figure 4.5: 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**

4.3.1 Distributed Hash Tables

Arguably all Distributed Hash Tables (DHTs) are built on the concept of 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 [**?**] [32] [34] [37], but other metrics such as geographic location can be used as well [35].

These DHTs have carefully chosen metric spaces such that these regions are very simple to calculate. For example, Chord [**?**] and similar ring-based DHTs [30] 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 4.5) is extremely similar to the one used in Ratnasamy et al.'s Content Addressable Network (CAN) [34], which would be $O(n^{\frac{1}{d}})$ hops.

DGVH can be used in a DHT to quickly and scalably construct both the Voronoi tessellation 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.

4.3.2 Wireless Coverage

Cărbunar et al. [12] 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 [12].

Chen et al. [13] 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]

4.4 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.

4.4.1 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 [39] library. We tested in 2-dimensional euclidean space and measured the number of edges in the graph generated by DGVH that differ from the graph generated by a Delaunay Triangulation. We tested networks of 100,

[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.

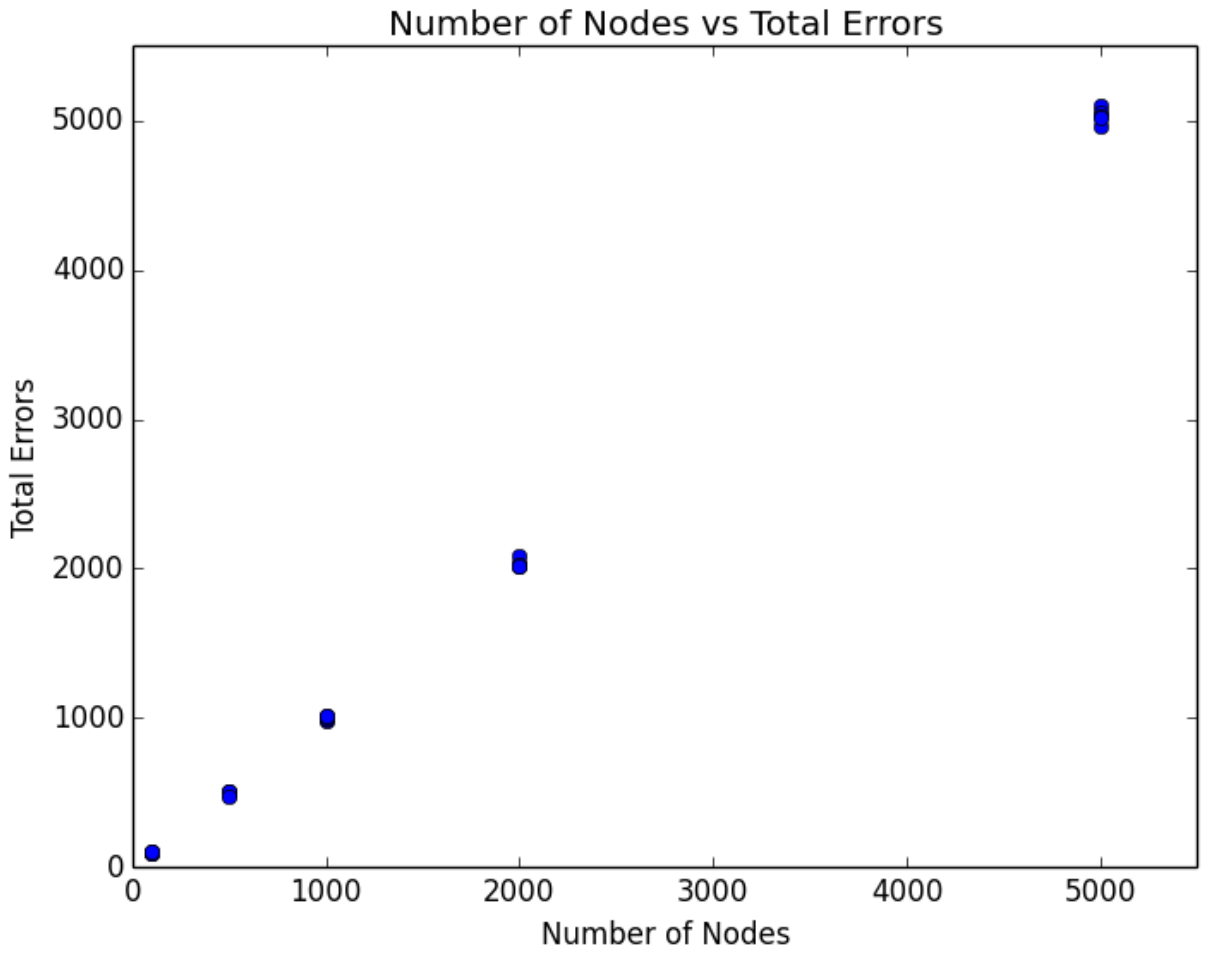


Figure 4.6: As the size of the graph increases, we see approximately 1 error per node.

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 4.6.

4.4.2 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.

Figure 4.7: Routing Simulation Sample

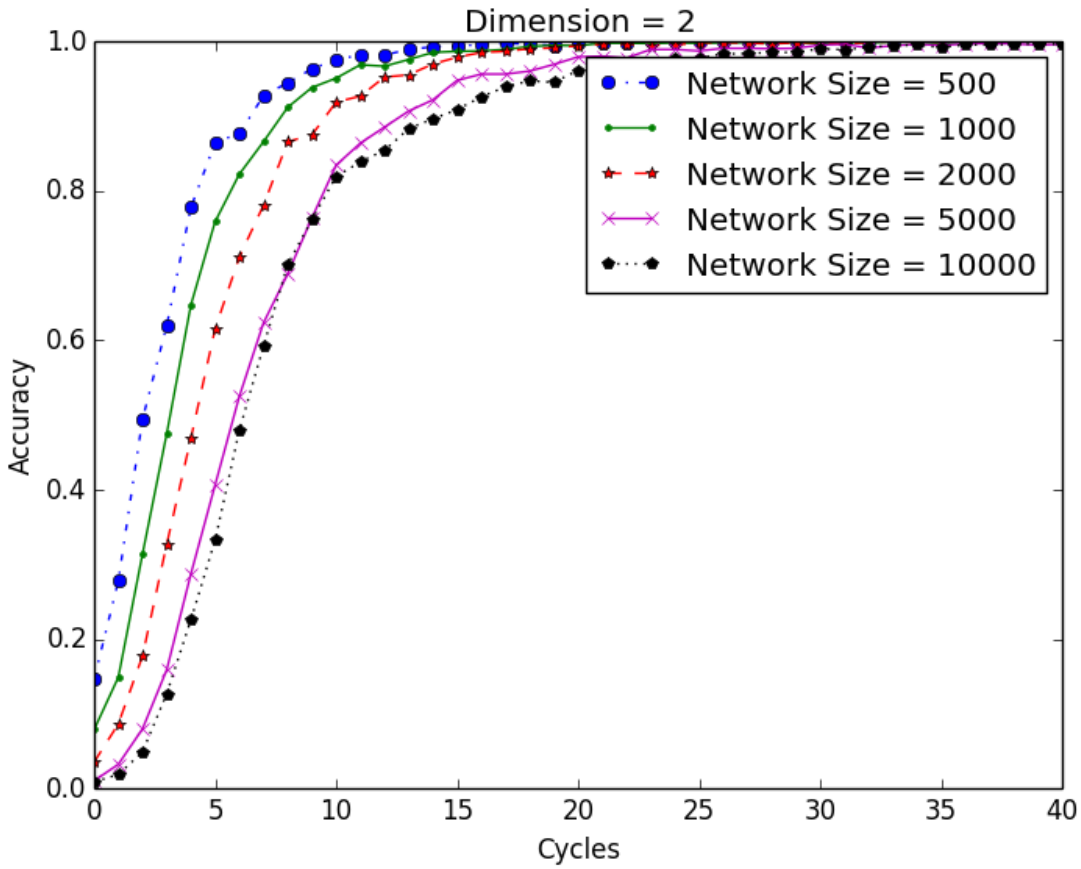
```
1: start ← random node
2: dest ← random set of coordinates
3: ans ← node closest to dest
4: if ans == start.lookup(dest) then
5:     increment hits
6: end if
```

We compare these results to RayNet [7], 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 appending 10 nodes, selected uniformly at random from the entire network. In each cycle, the nodes gossip (Algorithm 4.4) 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 Algorithm 4.7. A lookup is successful when the network correctly determines which Voronoi region contains a randomly selected point.

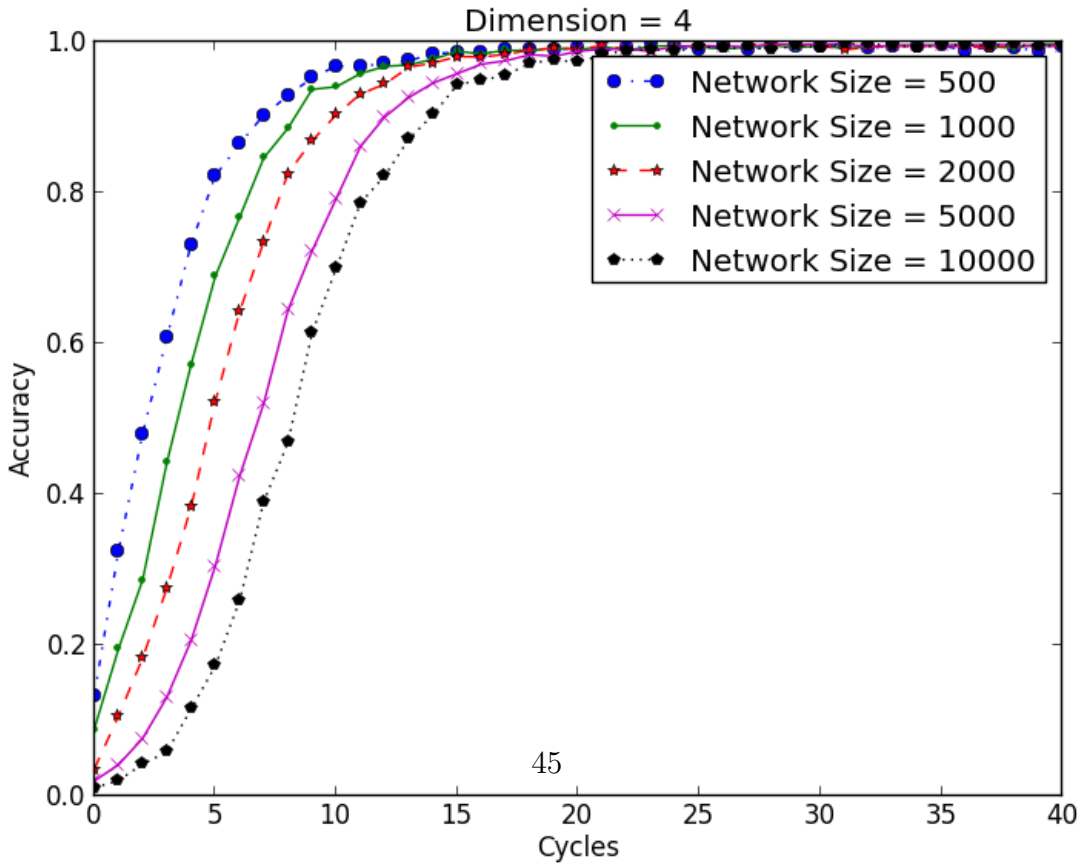
Our experimental variables for this simulation were the number of nodes in the DGVH generated overlay and the number of dimensions. We tested network sizes of 500, 1000, 2000, 5000, and 10000 nodes each in 2, 3, 4, and 5 dimensions. The hit rate at each cycle is $\frac{hits}{2000}$, where $hits$ are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

Our results are shown in Figures 4.8a, 4.8b, 4.8c, and 4.8d 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 [7]. As the network size and number of dimensions each increase, convergence slows, but not to a significant degree.



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

(b) This pl



4.5 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 [5] is a Voronoi-based P2P protocol designed for handling event messages in a massively multiplayer online video game. Each node finds its neighbors by constructing a Voronoi diagram using Fortune's sweepline algorithm [18]. 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 [6] and RayNet [7] are two-dimension DHTs that we intend to extend using our technique. VoroNet is based off Kleinberg's small world model [25] 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 [7].

RayNet [7] 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.

4.6 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 5

UrDHT: A Unified Model for Distributed Hash Tables

Andrew Rosen Brendan Benshoof Robert W. Harrison Anu G. Bourgeois

5.1 Introduction

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

Distributed Hash Tables have been the catalyst for the creation of many P2P applications. Among these are Redis [**?**], Freenet [**?**], and, most notably, BitTorrent [14]. 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 [8]. We found that all DHTs can cleanly be mapped to the primal-dual problems of Voronoi Tessellation and Delaunay Triangulation.

UrDHT builds its topology directly upon this insight. It uses a greedy distributed heuristic for approximating Delaunay Triangulations. We found that we could reproduce the

topology of different DHTs by defining a selection heuristic and rejection algorithm for the geometry the DHT. For every DHT we implemented, our greedy approximation of Delaunay Triangulation produced a stable DHT, regardless of the geometry. This works in non-Euclidean geometries such as XOR (Kademlia) or even a hyperbolic geometry represented by a Poincarè disc.

The end result is not only do we have an abstract model of DHTs, we have a simple framework that developers can use to quickly create new distributed applications. This simple framework allows generation of internally consistent implementations of different DHTs that can have their performance rigorously compared.

To summarize our contributions:

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

  We show that DHTs cleanly map to the primal-dual problem of Delaunay Triangulation and Voronoi Tessellation. We list a set of simple functions that, once defined, allow our Distributed Greedy Voronoi Heuristic (DGVH) to be run in any space, creating a DHT overlay for that space (Section 5.2).

- 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 5.3).

- We show how to reproduce the topology of Chord and Kademlia using UrDHT. We also implement a DHT in a Euclidean geometry and a hyperbolic geometry represented by a Poincarè disc (Section 5.4).

- We conduct experiments that show building DHTs using UrDHT produced efficiently routable networks, regardless of the underlying geometry(Section 5.5).

- We present some efforts and projects that are similar to our own (Section 5.6).

- We discuss the ramifications of our work and what future work is available (Section 5.7).

5.2 What Defines a DHT

A distributed hash table is usually defined by its protocol; in other words, what it can do. Nodes and data in a DHT are assigned unique[1] keys via a consistent hashing algorithm. To make it easier to intuitively understand the context, we will call the key associated with a node its ID and refer to nodes and their IDs interchangeably.

A DHT can perform the `lookup(key)`, `get(key)`, and `store(key, value)` operations.[2] The `lookup` operation returns the node responsible for a queried key. The `store` function stores that key/value pair in the DHT, while `get` returns the value associated with that key.

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

5.2.1 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

[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 5.1: 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
```

topology would allow using only short peers. This is typically $O(\log(n))$ hops, although polylogarithmic time is acceptable [**?**]. A DHT can still function without long peers.

Interestingly, despite the diversity of DHT topologies and how each DHT organizes short and long peers, all DHTs use functionally identical greedy routing algorithms (Algorithm 5.1):

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 [32]. The base greedy algorithm is always the same regardless of the implementation.

With the components of a DHT defined above, we can now show the relationship between DHTs and the primal-dual problems of Delaunay Triangulation and Voronoi Tessellation. An example Delaunay Triangulation and Voronoi Tessellation is show in Figure 5.2.

We can map a given node's ID to a point in a space and the set of short peers to the

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

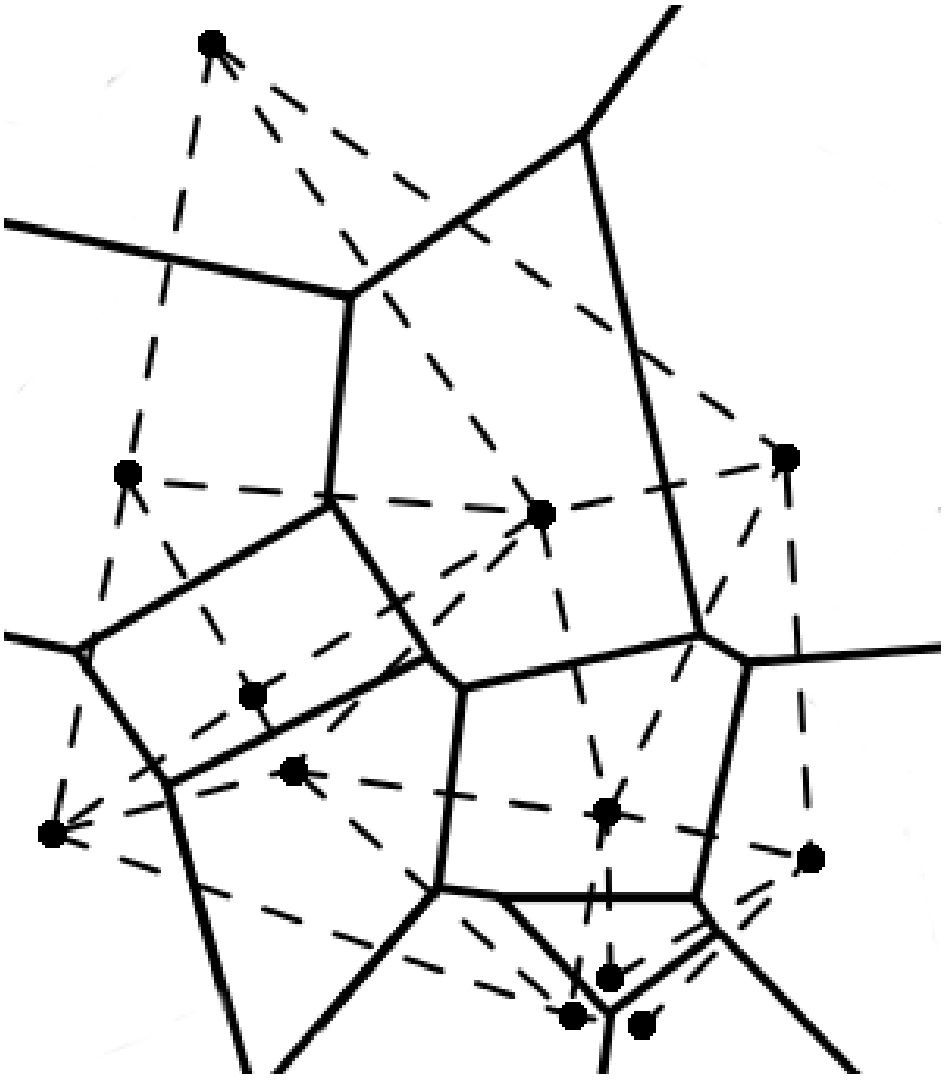


Figure 5.2: An example Voronoi diagram for objects on a 2-dimensional space. The black lines correspond to the borders of the Voronoi region, while the dashed lines correspond to the edges of the Delaunay Triangulation.

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

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

Computing the Delaunay Triangulation and/or the Voronoi Tessellation of a set of points is a well analyzed problem. Many algorithms exist which efficiently compute a Voronoi Tessellation for a given set of points on a plane, such as Fortune's sweep line algorithm [18].

However, DHTs are completed decentralized, with no single node having global knowledge of the topology. Many of the algorithms to compute Delaunay Triangulation and/or Voronoi Tessellation are unsuited to a distributed environment. In addition, the computational cost increases when we move into spaces with greater than two dimensions. In general, finding the Delaunay Triangulation of n points in a space with d dimensions takes $O(n^{\frac{2d-1}{d}})$ time [44].

Is there an algorithm we can use to efficiently calculate Delaunay Triangulation for a distributed system in an arbitrary space? We created an algorithm called the Distributed Greedy Voronoi Heuristic (DGVH), explained below [8].

5.2.2 Distributed Greedy Voronoi Heuristic

The Distributed Greedy Voronoi Heuristic (DGVH) is an efficient method for nodes to approximate their individual Voronoi region (Algorithm 5.3). DGVH selects nearby nodes that would correspond to points connected to it within a Delaunay Triangulation. Our previous implementation relied on a midpoint function [8]. We have refined our heuristic to render a midpoint function unnecessary.

The heuristic is described in Algorithm 5.3. Every maintenance cycle, nodes exchange their peer lists with their short peers. A node creates a list of candidates by combining their

Figure 5.3: Distributed Greedy Voronoi Heuristic

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

peer lists with their neighbor's peer lists.[4] Sort the list of peers from closest to furthest distance. The node then initializes a new peer list, initially containing the closest candidate. For each of the remaining candidates, the node compares the distance between the current short peers and the candidate. If the new peer list does not contain any short peers closer to the candidate than the node, the candidate is added to the new peer list. Otherwise, the candidate is set aside.

The resulting short peers are a subset of the node's actual Delaunay neighbors. A crucial feature is that this subset guarantees that DGVH will form a routable mesh.

Candidates are gathered via a gossip protocol as well as notifications from other peers. How long peers are handled depends on the particular DHT implementation. This process is described more in Section 5.3.1.

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

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

dimensions [9]. 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 $O(\frac{\log^4 n}{\log^4 \log n})$ [8], regardless of the dimension [8].[5] In most cases, this cost is much lower. Additional details can be found in our previous work [8].

We have tested DGVH on Chord (a ring-based topology), Kademlia (an XOR-based tree topology), general Euclidean spaces, and even in a hyperbolic geometry. This is interesting because not only can we implement the contrived topologies of existing DHTs, but more generalizable topologies like Euclidean or hyperbolic geometries. We show in Section 5.5 that DGVH works in all of these spaces. DGVH only needs the distance function to be defined in order for nodes to perform lookup operations and determine responsibility. We will now show how we used this information and heuristic to create UrDHT, our abstract model for distributed hash tables.

5.3 UrDHT

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

UrDHT is divided into 3 broad components: Storage, Networking, and Logic. Storage handles file storage and Networking dictates the protocol for how nodes communicate. These components oversee the lower level mechanics of how files are stored on the network and how bits are transmitted through the network. The specifics are outside the scope of the paper, but can be found on the UrDHT Project site [36].

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

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

Space Math is what effectively distinguishes one DHT from another. A developer only needs to change the details of the `space math` package in UrDHT to create a new type of DHT. We discuss each in further detail below.

5.3.1 The DHT Protocol

The DHT Protocol (`LogicClass.py`) [36] is the shared functionality between every single DHT. It consists of the node's information, the short peer list that defines the minimal overlay, the long peers that make efficient routing possible, and all the functions that use them. There is no need for a developer to change anything in the DHT Protocol, but it can be modified if so desired. The DHT Protocol depends on functions from Space Math in order to perform operations within the specified space.

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

The first thing we note is the absence of `lookup`. In our efforts to further abstract DHTs, we have replaced `lookup` using the function `seek`. The `seek` function acts a single step of `lookup`. It returns the closest node to *key* that the node knows about.

Nodes can perform `lookup` by iteratively calling `seek` until it receives the same answer twice. We do this because we make no assumptions as to how a client using a DHT would want to perform lookups and handle errors that can occur. It also means that a single client implementing `lookup` using iterative `seek` operations could traverse any DHT topology implemented with UrDHT.

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

Once the short peers are calculated, the node handles modifying its long peers. This is done using the `handleLongPeers` function described in Section 5.3.2.

5.3.2 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 5.3).

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

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

  The implementation of this function should vary greatly from one DHT to another. For example, Symphony [29] and other small-world networks [25] choose long peers using

a probability distribution. Chord has a much more structured distribution, with each long peer being increasing powers of 2 distance away from the node [**?**]. The default behavior is to use all candidates not chosen as short peers as long peers, up to a set maximum. If the size of long peers would exceed this maximum, we instead choose a random subset of the maximum size, creating a naive approximation of the long links in the Kleinberg small-world model [25]. Long peers do not greatly contribute to maintenance overhead, so we chose 200 long peers as a default maximum.

5.4 Implementing other DHTs

5.4.1 Implementing Chord

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

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

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

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

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

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

where

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

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

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

5.4.2 Implementing Kademlia

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

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

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

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

5.4.3 ZHT

ZHT [**?**] leads to an extremely trivial implementation in UrDHT. Unlike other DHTs, ZHT assumes an extremely low rate of churn. It bases this rationale on the fact that tracking $O(n)$ peers in memory is trivial. This indicates the $O(\log n)$ memory requirement for other

DHTs is overzealous and not based on a memory limitation. Rather, the primary motivation for keeping a number of peers in memory is more due to the cost of maintenance overhead. ZHT shows, that by assuming low rates of churn (and infrequent maintenance messages as a result), having $O(n)$ peers is a viable tactic for faster lookups.

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

5.4.4 Implementing a DHT in a non-contrived Metric Space

We used a Euclidean geometry as the default space when building UrDHT and DGVH [8]. For two vectors $\vec{a}$ and $\vec{b}$ in d dimensions:

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

We implement `getDelaunayPeers` using DGHV and set the minimum number of short peers to $3d + 1$, a value we found through experimentation [8].

Long peers are randomly selected from the left-over candidates after DGVH is performed [8]. 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 [26] [15]. This has the potential to allow for minimal latency DHTs.

5.5 Experiments

We use simulations to test our implementations of DHTs using UrDHT. Using simulations to test the correctness and relative performance of DHTs is standard practice for testing and analyzing DHTs [32] [29] [**?**] [**?**] [7] [**?**].

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

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

For each topology, at each step, we measured:

- The average degree of the network. This is the number of outgoing links and includes both short and long peers.

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

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

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

Results generated by the Chord and Kademlia simulations were in line with those from previous work [32] [**?**]. This demonstrates that UrDHT is capable of accurately emulating these topologies. We show these results in Figures 5.4 - 5.7.

The results of our Euclidean and Hyperbolic geometries indicate similar asymptotic behavior: a higher degree produces a lower diameter and average routing. However, the ability to leverage this trade-off is limited by the necessity of maintaining an $O(\log n)$ degree. These results are shown in Figures 5.8 - 5.11.

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

5.6 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

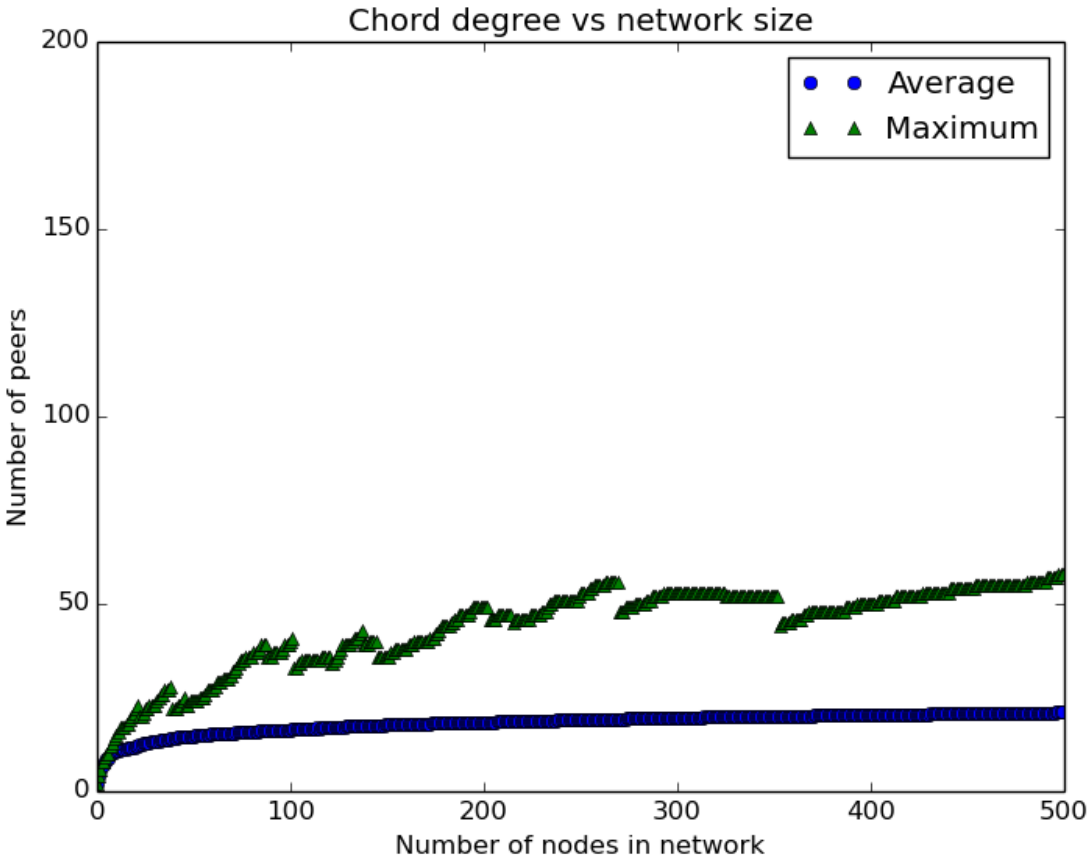


Figure 5.4: This is the average and maximum degree of nodes in the Chord network. This Chord network utilized a 120 bit hash and thus degree is bound at 122 (full fingers, predecessor and successor) when the network reaches 2^{120} nodes.

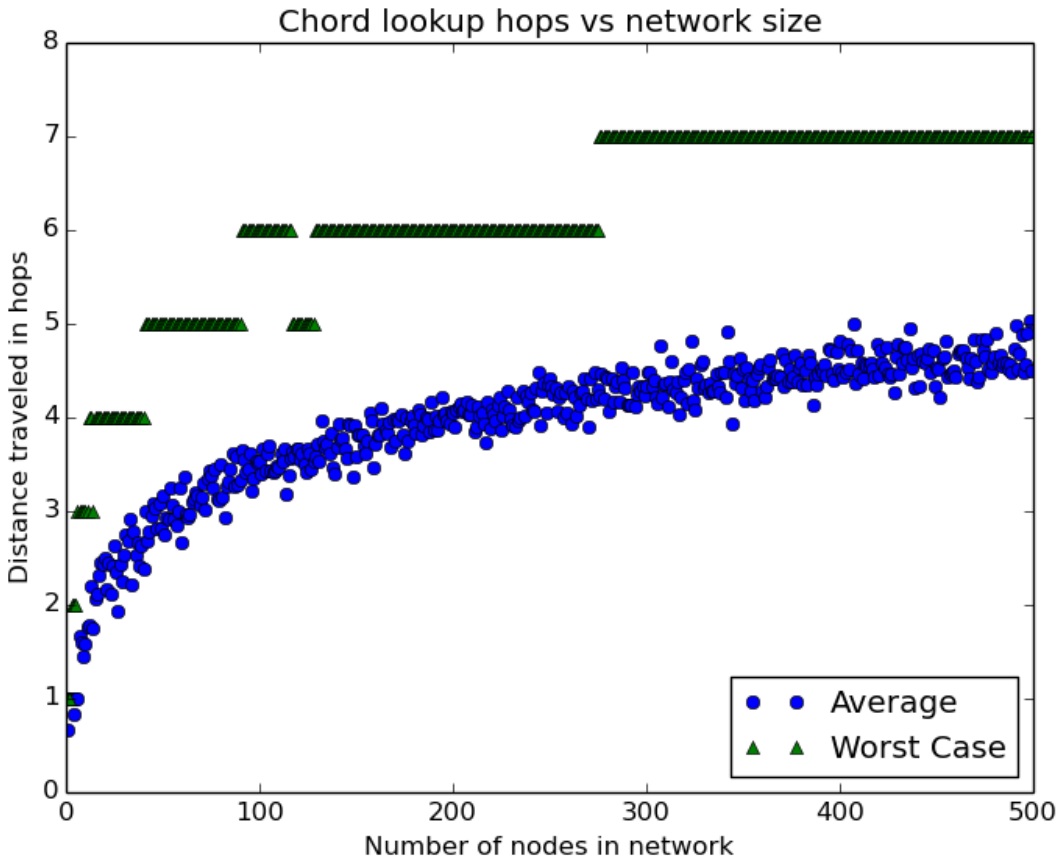


Figure 5.5: This is the number hops required for a greedy routed lookup in Chord. The average lookup between two nodes follows the expected logarithmic curve.

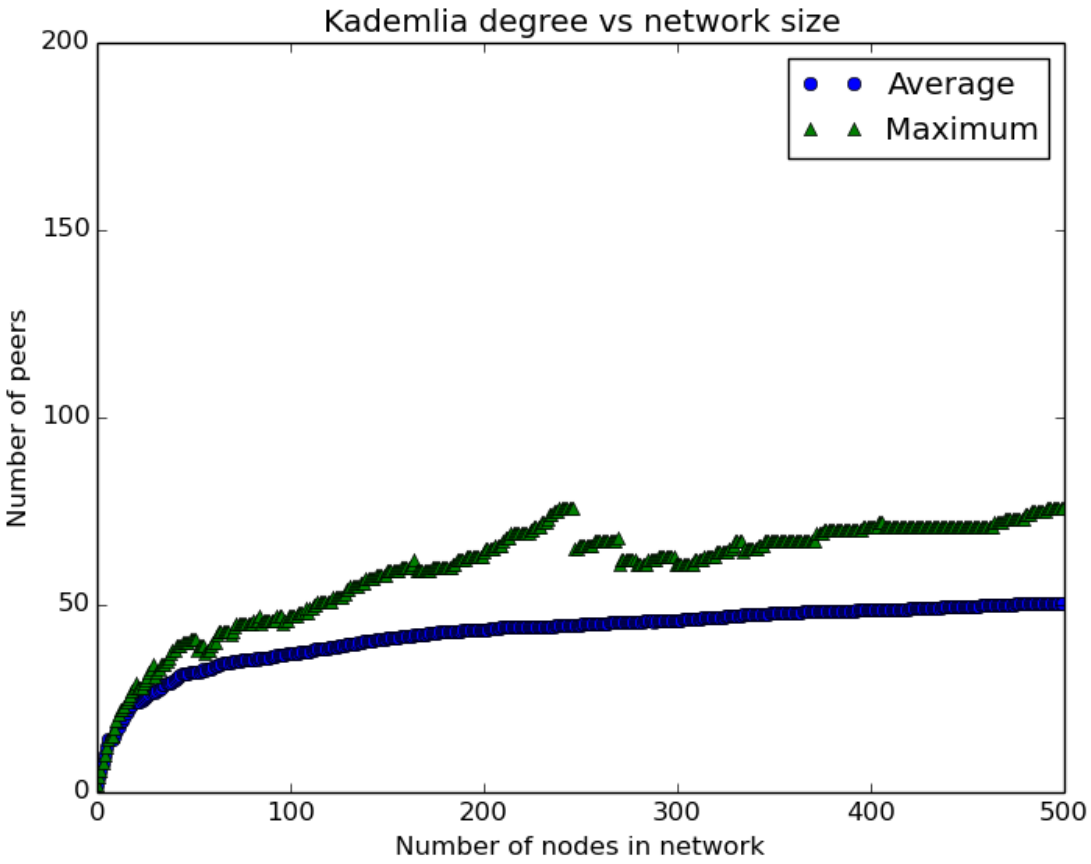


Figure 5.6: This is the average and maximum degree of nodes in the Kademlia network as new nodes are added. Both the maximum degree and average degree are $O(\log n)$.

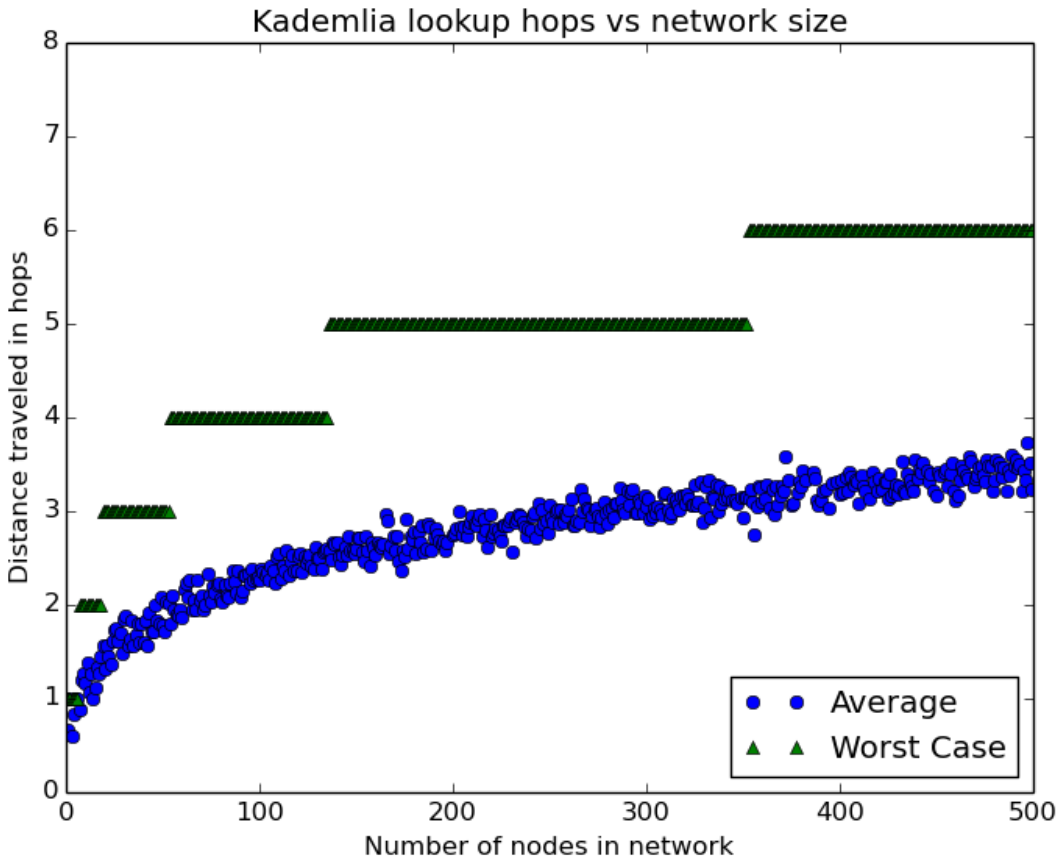


Figure 5.7: Much like Chord, the average degree follows a distinct logarithmic curve, reaching an average distance of approximately three hops when there are 500 nodes in the network.

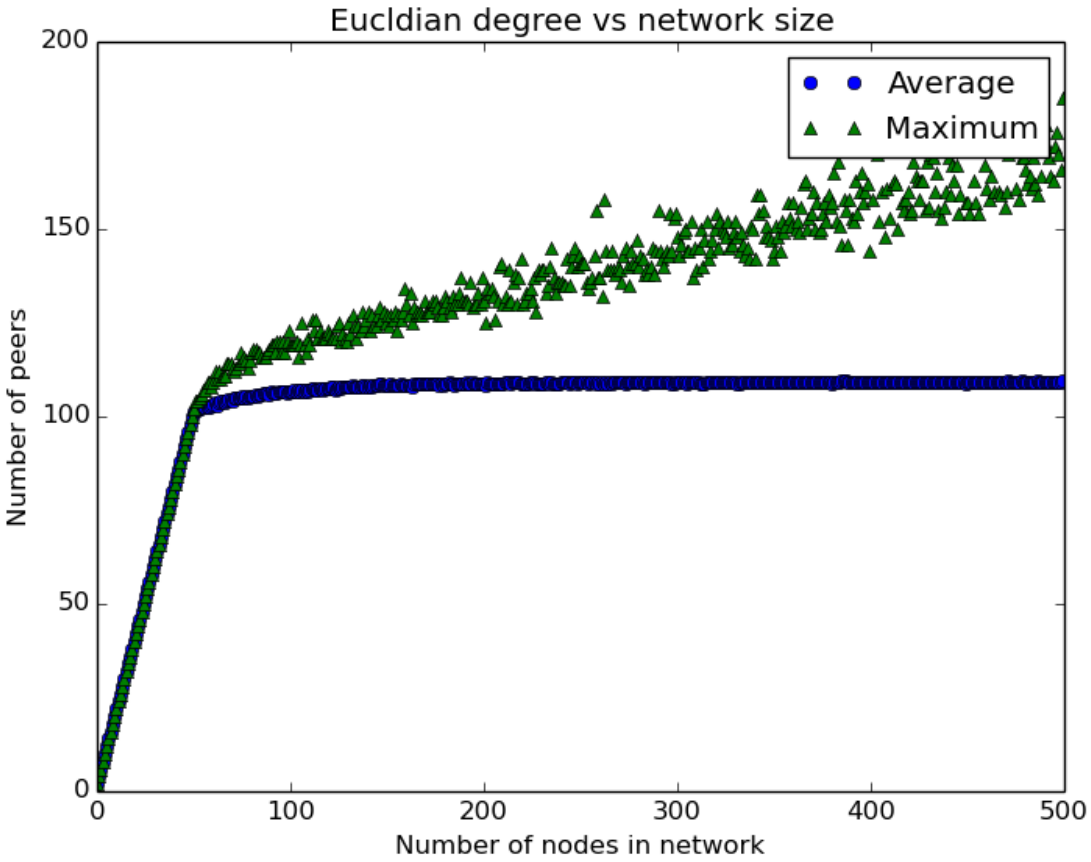


Figure 5.8: Because the long peers increase linearly to the maximum value (49), degree initially rises quickly and then grows more slowly as the number of long peers ceases to grow and the size short peers increases with network size.

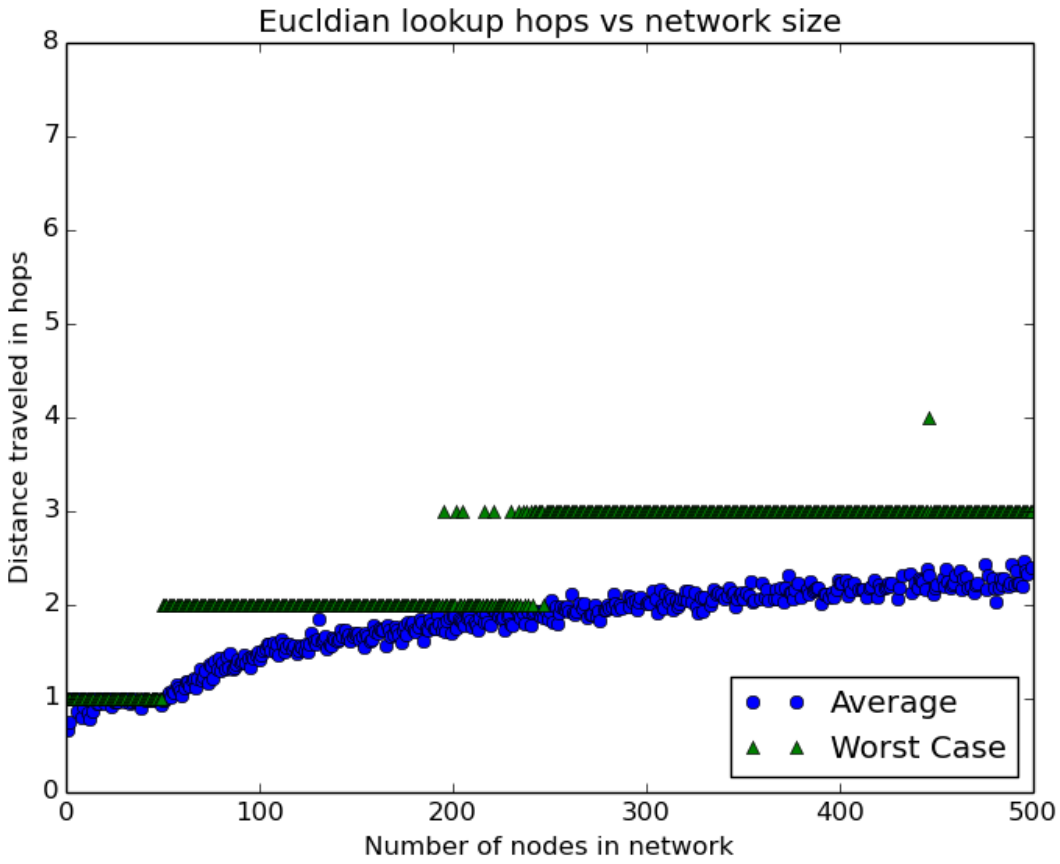


Figure 5.9: The inter-node distance stays constant at 1 until long peers are filled, then rises at the rate of a randomly connected network due to the distribution of long peers selected

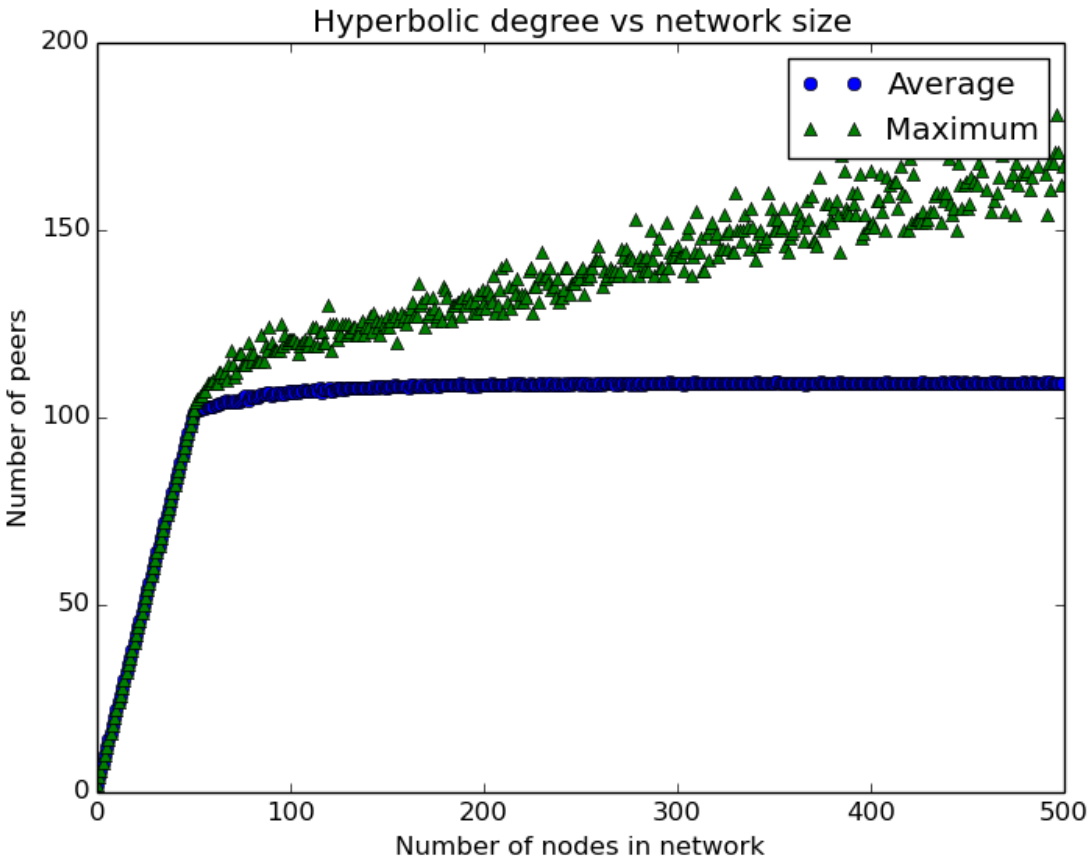


Figure 5.10: The Hyperbolic network uses the same long and short peer strategies to the Euclidean network, and thus shows similar results.

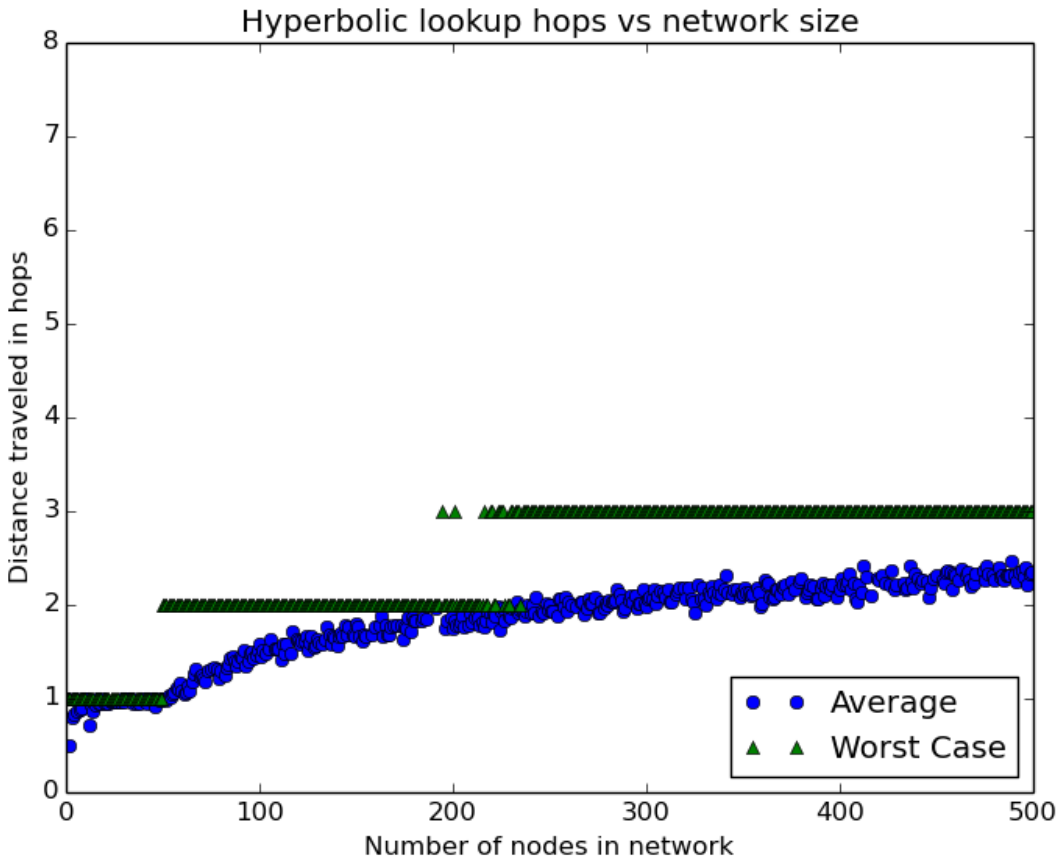


Figure 5.11: Like the Euclidean Geometry, our Poincarè disc based topology has much shorter maximum and average distances.

using system called P2 [**?**] [**?**]. P2 is a network engine for constructing overlays which uses the Overlog declarative logic language. Writing programs for P2 in Overlog yields extremely concise and modular implementations of for overlay networks.

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

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

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

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

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

5.7 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 [26] [15]. 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 6

Replication Strategies to Increase Storage Robustness in Decentralized P2P Architectures

Brendan Benshoof

Abstract

Classic P2P key-value storage systems use a small section of strategies to ensure that records are maintained in the system despite constant churn. These strategies see effective levels of robustness to churn, however they often do not provide effective robustness against systemic failures due to natural disasters and network partitioning in the Internet or the overlay topology of the P2P systems. We explore and evaluate the status quo of replica robustness strategies in the face of partition failures and propose new techniques to improve over the established methods.

6.1 Introduction

In this paper, we will consider a selection of strategies to increase robustness of storage in a p2p network. Each strategy will be described, analyzed and finally compared and contrasted with other strategies.

Current DHTs are oprimized to store a large number of smaller records. While is it is perfectly viable to link together such records to store larger files (and even use it as a shared file system), it has generally been used as a layer of indirection and provides means of discovering high bandwidth protocols. For example, Bitorent uses a DHT to help find peers for particular torrent rather than storing the files directly on the DHT. If we maintain this trend of only storing small records on the DHT, it stands out as a clear choice for use a shared DNS cache, or as a mechanism for storing encrypted cryptographic keys. All of these use cases are predicated on a DHT being a reliable store of data.

6.2 Robustness

6.2.1 Robustness and Churn

Most DHTs focus their efforts on preventing record loss due to churn. Records are lost to churn under two conditions: the node hosting the replica leaves the network or the node with a replica 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). This value describes the portion of DHT's metric space that is maintained by a different node at the end of the period. This value is related to the more conventional churn rate metric $\frac{exits+joins}{2*size}$ but provides more information on the viability of records.

6.2.2 Robustness and network partitions

Network partitions are when a failure results in the network separating into multiple non-connected networks. This can be a result of failures in the underlay and the overlay networks. The result of this, is that unlike the churn based failures, failure of nodes is not independent. 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 networks, each having just lost all nodes in the other partition simultaneously. This means that robustness methods based periodically re-storing records after they are lost are likely to be insufficient as all the possessors of a record may be found only in one of the partitions.

In practice, overlay partitions may occur due to eclipse attacks and logic errors in how the DHT constructs the overlay. 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.

Underlay Partitions

An underlay partition can happen as result in a failure of infrastructure, manipulation of BGP, or governmental action. Assuming the overlay network of the DHT has been constructed randomly in relation to the underlay network, the failures due to the underlay partition will essentially occur at random locations in the network.

While this may resemble how failure occurs during churn, the failures will all occur effectively simultaneously, to a potentially very 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 networks 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.

6.3 Passive Replication Strategies

When DHTs are formalized[**?**][32], replication strategies are not discussed. Passive strategies are those where a client writes the record and replicas to the DHT once, then 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 due to churn. Because partition failures are being analyzed as specific events, rather than behavior over time, passive strategies will often fair similarly to active strategies because there will not have been sufficient time for nodes or sponsors to take action in response to the partition (If they realize a partition has occurred at all).

6.3.1 k-random nodes

The K-random node strategy is not used by any established DHT, however it provides a simple analytical model, which 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 $hash(hash(L))$ and similarly re-hashing the identifier from the previous step until K nodes have been stored.

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). Because the locations are effectively random, we can simplify our analysis for churn and partition tolerance.

The half-life of the record (a time period over which, on average, the number of replicas in the network will be halved) is $\frac{-\log(2)}{1-R_c}$

6.3.2 k-nearest nodes

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 to the target nodes. 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. Failure in such a system is limited to overlay partition failures, where only one partition of the network is likely

6.4 Active Replication Strategies

6.4.1 k-Replica

6.4.2 Adaptive Replica (IRM)

6.5 Active Replication Strategies

6.5.1 Active sponsorship

Active sponsorship is part of the method used to greatest effect in practice (it is often combined with a passive replication strategy). When a member adds records to the network, it periodically re-adds the records such that the records are ensured to exist in a reachable location. This method is specifically designed to combat churn related data loss. It's success is predicated on the idea that the member adding the file will be more reliable then most nodes in the network and will persist over time. Often the sponsorship is shared between many members who have a vested interest in the existence of the record. Uptime is a function of the rate of sponsorship, number of sponsors, churn rate and any passive strategies used to augment the system.

6.5.2 p2p backups

6.5.3 Replication within subnetworks

6.5.4 Active Sponsorship

6.5.5 Redundant Subnetworks

6.6 analysis

6.7 Conclusions

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

[5] 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.

[6] 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.

[7] 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.

[8] Brendan Benshoof, Andrew Rosen, Anu G. Bourgeois, and Robert W Harrison. A distributed greedy heuristic for computing voronoi tessellations with applications to-

wards peer-to-peer networks. In *Dependable Parallel, Distributed and Network-Centric Systems, 20th IEEE Workshop on*.

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

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

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

[16] 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.

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

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

[19] 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.

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

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

[22] 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.

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

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

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

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

[27] 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.

[28] 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.

[29] 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.

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

[31] 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.

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

[33] 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.

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

[35] 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.

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

[37] 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.

[38] 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.

[39] 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.

[40] 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.

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

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

[43] 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.

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

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