Proposal
Towards a Framework for DHT Distributed Computing

Andrew Rosen

July 5, 2015

Contents

Abstract

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

Our framework is a completely decentralized model for organizing heterogeneous units for distributed computation. It incorporates a load-balancing algorithm that is capable of injecting additional nodes during runtime to speed up existing jobs. This algorithm also provides a means of redistributing the load among existing workers during runtime.

Unlike Hadoop and similar MapReduce frameworks, our framework can be used both in both the context of a datacenter or as part of a P2P computing platform. This opens up new possibilities for building platforms to distributed computing problems. By utilizing the load-balancing algorithm, a datacenter could easily leverage additional P2P resources at runtime on an as-needed basis. Our framework also allows MapReduce-like or distributed machine learning platforms to be easily deployed in a greater variety of contexts.

Chapter 1

Introduction

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

1.1 Objective

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

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

These are many of the same challenges that Peer-to-peer (P2P) file sharing applications have. Many P2P applications use DHTs to address these challenges, since DHTs are designed with these problems in mind. We propose that DHTs can be used to create P2P distributed computing platforms that are completely

decentralized. There would be no need for some central organized or scheduler to coordinate the nodes in the network. Our framework would not be limited to only a P2P context, but could be applied in data centers, a normally centrally organized context.

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

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

1.2 Applications of Distributed Hash Tables

Distributed Hash Tables have been used in numerous applications:

- *P2P file sharing* is by far the most prominent use of DHTs. The most well-known application is BitTorrent [11], which is built on Mainline DHT [30].
- DHTs have been used for *distributed storage* systems [15].
- *Distributed Domain Name Systems* (DNS) have been built upon DHTs [14] [37]. Distributed DNSs are much more robust that DNS to orchestrated attacks, but otherwise require more overhead.
- DHT was used as the name resolution layer of a large *distributed database* [33].
- Distributed *machine learning* [28].
- Many *botnets* are now P2P based and built using well established DHTs [46]. This is because the decentralized nature of P2P systems means there is no single vulnerable location in the botnet.
- *Live video streaming* (BitTorrent live) [36].

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

Our goal was to further extend the use of DHTs. In previous work [43], we showed that a DHT can be to create a distributed computing framework. We used the same mechanism used in P2P applications that assigns nodes their

[1] As far as we know.

location in the network to evenly distribute work among members of a DHT. The most direct application of a DHT distributed computing framework is a quick and intuitive way to solve embarrassingly parallel problems, such as:

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

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

1.3 Why Use Distributed Hash Tables in Distributed Computing

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

1.3.1 General Challenges of Distributed Computing

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

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

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

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

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

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

1.3.2 How DHTs Address these Challenges

Distributed Hash Tables are essentially distributed lookup tables. DHTs use a consistent hashing algorithm, such as SHA-1 [21], to associate nodes and file identifiers with keys. These keys dictate where the nodes and files will be located on the network. The connections between nodes are organized such that any node can efficiently lookup the value associated with any given key, even though the node only knows a small portion of the network. We discuss the specifics of this in Chapter 2.

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

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

Using consistent hashing allows the network to scale up incrementally, adding one node at a time [17]. In addition, each join operation has minimal impact on the network, since a node affects only its immediate neighbors on a join operation. Similarly, the only nodes that need to react to a node leaving are its neighbors. Other nodes can be notified of the missing node passively through maintenance or in response to a lookup.

There have been multiple proposed strategies for tackling scalability, and it is these strategies that play the greatest role in driving the variety of DHT architectures. Each DHT must strike a balance between the size of the lookup

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

[3] Except for ZHT [29], which breaks this rule deliberately by giving each node a full copy of the routing table.

table and lookup time. The vast majority of DHTs choose to use $\lg(n)$ sized tables and $\lg(n)$ hops. Chapter 2 discusses these tradeoffs in greater detail and how they affect the each DHT.

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

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

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

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

The fault tolerance mechanisms in DHTs also provide near constant availability for P2P applications. The node that is responsible for a particular key can always be found, even when numerous failures or joins occur [51].

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

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

We found defining the meaning of "close" equivalent choosing a metric for Voronoi tessellation [9]. However, because this is a random process, not all values are evenly distributed, but enough hash keys yield a close enough approximation.

[4] Again, except for ZHT.

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

Heterogenity presents a challenge for load-balancing DHTs due to conflicting assumptions and goals. DHTs assume that members are usually going to be varied in hardware, but the load-balancing process defined in DHTs treats each node equally. In other words, DHTs support heterogeneity, but do not attempt to exploit it.

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

1.4 Roadmap

In this section, we give a brief overview of our previous work. We go into further detail of our previous work in Chapter 3 and present the proposed work of our dissertation in Chapter 4.

1.4.1 Completed Work

One of our first projects was to create a distributed computing platform using the Chord DHT [43]. Our goal here was to create a completely decentralized distributed computing framework that was fault-tolerant during job execution. We did this by implementing MapReduce over Chord. We then tested our prototype's fault-tolerance by executing MapReduce jobs under churn.

Our experiments with excessively high levels of churn created an anomaly in the runtime of our computations. Under beyond practical levels of experimental churn, we found that our computation was quicker than our experiments without churn. We hypothesized that this is because the random churn is acting as a (inefficient) process for autonomous load-balancing. This phenomena is described in detail in Chapter 3.3, but suggested to us that there was a way to dynamically load-balance during execution.

Our second project was to develop VHash [8] [9], a distributed hash table based on Delaunay Triangulation. VHash is unique due to the way it could work in multidimensional spaces. Other DHTs typically use a space with a single dimension and optimize for the number of hops. VHash can optimize for whatever attributes are used to define the space. Our experiments showed that VHash outperforms Chord in terms of routing latency.

Our third project which analyzed the amount of effort that would be required to attack a DHT using a method known as the Sybil attack [44]. The Sybil attack [18] is a well known attack against distributed systems, but it had not been fully analyzed from the perspective of an attacker. Our results showed that attackers required relatively few resources to compromise a much larger network. We believe that some of the components that are used to perform a Sybil attack can be used for autonomous load balancing.

Publications

- Andrew Rosen, Brendan Benshoof, Robert W. Harrison, Anu G. Bourgeois "MapReduce on a Chord Distributed Hash Table" Poster at IPDPS 2014 PhD Forum [43]
- Andrew Rosen, Brendan Benshoof, Robert W. Harrison, Anu G. Bourgeois "MapReduce on a Chord Distributed Hash Table" Presentation ICA CON 2014
- Brendan Benshoof, Andrew Rosen, Anu G. Bourgeois, Robert W. Harrison "VHASH: Spatial DHT based on Voronoi Tessellation" Short Paper ICA CON 2014 [9]
- Brendan Benshoof, Andrew Rosen, Anu G. Bourgeois, Robert W. Harrison "VHASH: Spatial DHT based on Voronoi Tessellation" Poster ICA CON 2014
- Brendan Benshoof, Andrew Rosen, Anu G. Bourgeois, Robert W. Harrison "A Distributed Greedy Heuristic for Computing Voronoi Tessellations With Applications Towards Peer-to-Peer Networks" IEEE IPDPS 2015 - Workshop on Dependable Parallel, Distributed and Network-Centric Systems [8]

The following papers are in progress:

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

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

- Erin-Elizabeth A. Durham, Andrew Rosen, Robert W. Harrison "A Model Architecture for Big Data applications using Relational Databases" 2014 IEEE BigData - C4BD2014 - Workshop on Complexity for Big Data [19]
- Chinua Umoja, J.T. Torrance, Erin-Elizabeth A. Durham, Andrew Rosen, Dr. Robert Harrison "A Novel Approach to Determine Docking Locations Using Fuzzy Logic and Shape Determination" 2014 IEEE BigData - Poster and Short Paper [52]
- Erin-Elizabeth A. Durham, Andrew Rosen, Robert W. Harrison "Optimization of Relational Database Usage Involving Big Data" IEEE SSCI 2014 - CIDM 2014 - The IEEE Symposium Series on Computational Intelligence and Data Mining [20]

1.4.2 Summary of Proposal

The work we want to do can be divided into three distinct, but mutually dependent parts. One of these parts, the DHT framework, is a part that will be done jointly with Brendan Benshoof. The specifics are given in Chapter 4.

DHT Framework

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

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

DHT Distributed Computing

This portion will be the bulk of the experimental work and data gathering. Using our created framework, we will create implement and test distributed computing problems on different DHT implementations, such as Chord [51] and Kademlia [34].

Autonomous Load-Balancing

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

Chapter 2

Background

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

DHTs have been a vibrant area of research for the past decade, with several of the concepts dating further back [11] [34] [41] [42] [39] [51] [45]. Numerous DHTs have been developed over the years and each of the major topologies have had multiple implementation and derivatives. This is partly because the process of designing DHTs involves making tradeoffs in maintenance schemes, topology, and memory, with no choice being strictly better than any other.

2.1 What is Needed to Define a DHT

There are a couple of ways to define what a DHT is. A distributed hash table assigns each node and data object in the network a unique key. The key corresponds to the identifier for the node or the data in question, typically IP/port combination or filename. This mapping is consistent, so that even though the keys are distributed uniformly at random, the key is always the same for the same input.

DHTs are traditionally used to form a peer-to-peer overlay network, in which the DHT defines the network topology. Any member of the network can efficiently find the node that corresponds to a particular key. Data can be stored in the network and can be retrieved by finding the node that is responsible for that key.

A distributed hash table can also be thought of as a space with points (data) and Voronoi generators (nodes). A node is responsible for data that falls within its Voronoi region, which is defined by the peers closest to it. The peers that share a border for a Voronoi region are members of the node's Delaunay triangulation. Starting from any node in the network, we can find any particular node or the node responsible for a particular point in sublinear time. Regardless of the definitions, each DHT protocol needs to specify specific qualities:

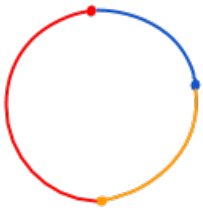


Figure 2.1: A Voronoi diagram for a Chord network, using Chord's definition of closest.

Distance Metric There needs to be a way to establish how far things are from one another. Once we have a distance metric, we define what we mean when we say a node is responsible for all data *close* to it.

Closeness Definition This definition of *closeness* is essential, since it defines what a node is responsible for and who its short hops are. The definition of closeness and distance are related but different.

We shall use Chord [51] as an example. The distance from a to b is defined as the shortest distance around the circle in either direction. However, a node is responsible for the points between its predecessor and it. The corresponding Voronoi diagram is showing in Figure 2.1.

However, say we were to use a more intuitive definition for closeness, where a node is responsible for the keys that were closer to it than any other node. In this case, we end up with the diagram in Figure 2.2.

A Midpoint Definition This defines the point which is the *minimal* equidistant point between two given points.

Peer Management Strategy This is the meat of the definition of a Distributed Hash Table. The peer management strategy includes how big peerlists are, what goes in it, and how often peers are checked to see if they are still alive. This is where almost all trade-offs are made.

Surprisingly, there is no need to define a routing strategy for individual DHTs. This is because all DHTs use the same overall routing strategy: forward the message to the known node closest to the destination. *How* routing is implemented depends on the protocol in question. Chord's routing can be implemented recursively or iteratively, while Kademlia's uses parallel iterative queries.

2.1.1 Terminology

The large number of DHTs have lead many papers to use different terms to describe congruent elements of DHTs, as some terms may make sense only in

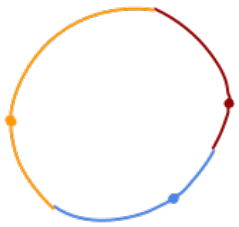


Figure 2.2: A Voronoi diagram for a Chord network, where closest is defined by the node being the closest in either direction.

one context. Since this paper will cover multiple DHTs that use different terms, we have created a unified terminology:

key - The identifier generated by a hash function corresponding to a unique[1] node or file. SHA-1, which generates 160-bit hashes, is typically used as a hashing algorithm.[2]

ID - The ID is a key that corresponds to a particular node. The ID of a node and the node itself are referred to interchangeably. In this proposal, we refer to nodes by their ID and files by their keys.

Peer - Another active member on the network. For this section, we assume that all peers are different pieces of hardware.

Peerlist - The set of all peers that a node knows about. This is sometimes referred to as the *routing table*, but certain DHTs [45] [56] overload the terminology. Any table or list of peers is a subset of the entire peerlist.

Short-hops - The subset of peers that are "closest/adjacent" to the node in the keyspace, according to the DHT's metric. In a 1-dimensional ring, such a Chord [51], this is the node's *predecessor(s)* and *successor(s)*. They may also be called *neighbors*.

Long-hops - The subset of the peerlist that the node is not adjacent to. These are sometimes referred to as fingers, long links, or shortcuts.

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

[2] Due to the research into hash collisions [50], and the glut of hardware that currently exists to perform SHA hash collisions, SHA1 is being depreciated by 2017.

Root Node - The node responsible for a particular key.

Successor - Alternate name for the root node. The successor of a node is the neighbor that will assume a nodes responsibilities if that node leaves.

n nodes - The number of nodes in the network.

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

`lookup(key)` - This operation finds the root node of `key`. Almost every operation on a DHT needs to leverage the `lookup` operation in some way.

`put(key,value)` - Stores `value` at the root node of `key`. Unless otherwise specified, `key` is assumed be the hashkey of `value`. This assumption is broken in Tapestry.

`get(key)` - This operates like lookup, except the context is to return the value stored by a `put`. This is a subtle difference, since one could `lookup(key)` and ask the corresponding node directly. However, many implementations use backup operations and caching, which will store multiple copies of the value along the network. If we do not care which node returns the value mapped with `key`, or if it is a backup, we can express it with `get`.

`delete(key, value)` - This is self-explanatory. Typically, DHTs do not worry about key deletion and leave that option to the specific application. When DHTs do address the issue, they often assume that stored key-value pairs have a specified time-to-live, after which they are automatically removed.

On the local level, each node has to be able to *join* and perform maintenance on itself.

`join()` The join process encompasses two steps. First, the joining node needs to initialize its peerlist. It does not necessarily need a complete peerlist the moment it joins, but it must initialize one. Second, the joining node needs to inform other nodes of its existence.

Maintenance Maintenance procedures generally are either *active* or *lazy*. In active maintenance, peers are periodically pinged and are replaced when they are no longer detected. Lazy maintenance assumes that peers in the peerlist are healthy until they prove otherwise, in which case they are either replaced immediately. In general, lazy maintenance is used on everything, while active maintenance is only used on neighbors[3].

When analyzing the DHTs in this chapter, we look at the overlay's geometry, the peerlist, the `lookup` function, and how fault-tolerance is performed in the DHTs. We assume that nodes never politely leave the network but always abruptly fail, since a `leave()` operation is fairly trivial and has minimal impact.

[3]check this statement for consistency

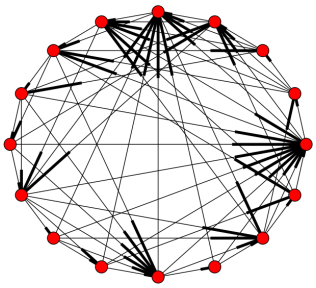


Figure 2.3: A Chord ring with 16 nodes. The fingers (long hop connections) are shown cutting across the ring.

2.2 Chord

Chord [51] is the archetypal ring-based DHT and it is impossible to create a new ring-based DHT without making some comparison to Chord. It is notable due its straightforward routing, its rules which make ownership of keys very easy to sort out, and the large number of derivatives.

Chord is extremely well known in Computer Science, and was awarded the prestigious 2011 SIGCOMM Test of Time Award [55]. However, recent research has demonstrated that there have been no correct implementations of Chord in over a decade [55].

Peerlist and Geometry

Chord is a 1-dimensional modular ring in which all messages travel in one direction - upstream, hopping from one node to another node with a greater ID until it wraps around. Each member of the network and the data stored within it is hashed to a unique m-bit key or ID, corresponding to one of the 2^m locations on a ring. An example Chord network is shown in Figure 2.3.

A node in the network is responsible for all the data with keys upstream from its predecessor's ID, up through and including its own ID. If a node is responsible for some key, it is referred to being the root or successor of that key.

Lookup and routing is performed by recursively querying nodes upstream. Querying only neighbors in this manner would take $O(n)$ time to lookup a key.

To speedup lookups, each node maintains a table of m shortcuts to other peers, called the *finger table*. The ith entry of a node n's finger table corresponds to the node that is the successor of the key $n + 2^{i-1} \mod 2^m$. During a lookup, nodes query the finger that is closest to the sought key without going past it, until it is received by the root node. Each hop essentially cuts the search space for a key in half. This provides Chord with a highly scalable $\log_2(n)$ lookup time for any key [51], with an average $\frac{1}{2}O(\log_2(n))$ number of hops.

Besides the finger tables, the peerlist includes a list of s neighbors in each direction for fault tolerance. This brings the total size of the peerlist to $log_2(2^m) + 2 \cdot s = m + 2 \cdot s$, assuming the entries are distinct.

Joining

To join the network, node n first asks n' to find `successor(`n`)`. Node n uses the information to set his successor, and maintenance will inform the other nodes of n's existence. Meanwhile, n will takeover some of the keys that his successor was responsible for.

Fault Tolerance

Robustness in the network is accomplished by having nodes backup their contents to their s immediate successors, the closest nodes upstream. This is done because when a node leaves the or fail, the most immediate successor would be responsible for the keys. In the case of multiple nodes failing all at once, having a successor list makes it extremely unlikely that any given stored value will be lost.

As nodes enter and leave the ring, the nodes use their maintenance procedures to guide them into the right place and repair any links with failed nodes. The process takes $O(\lg^2(n))$ messages. Full details on Chord's maintenance cycle can be found here [51].

2.3 Kademlia

Kademlia [34] is perhaps the most well known and most widely used DHT, as a modified version of Kademlia (Mainline DHT) is forms backbone of the BitTorrent protocol. The motivation of Kademlia was to create a way for nodes to incorporate peerlist updates with each query made.

Peerlist and Geometry

Like Chord, Kademlia uses m-bit keys for nodes and files. However, Kademlia utilizes a binary tree-based structure, with the nodes acting as the leaves of the tree. Distance between any two nodes in the tree is calculated by XORing their IDs. The XOR distance metric means that distances are symmetric, which is not the case in Chord.

Nodes in Kademlia maintain information about the network using a routing table that contains m lists, called k-buckets. For each k-bucket contains up to k nodes that are distance 2^i to 2^{i+1}, where $0 \leq i < m$. In other words, each k-bucket corresponds to a subtree of the network not containing the node. An example network is shown in Figure 2.4.

Each k-bucket is maintained by a least recently seen eviction algorithm that skips live nodes. Whenever the node receives a message, it adds the sender's

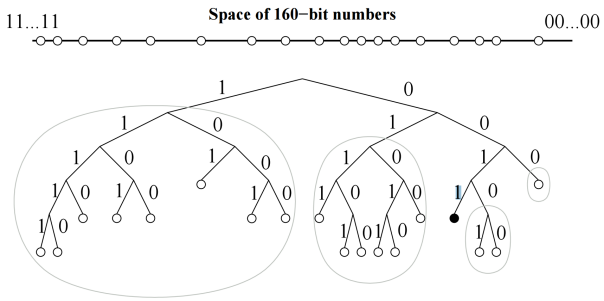


Figure 2.4: An example Kademlia network from the original paper [34]. The ovals are the node's k-buckets.

info to the tail of the corresponding k-bucket. If that info already exists, the info is moved to the tail.

If the k-bucket is full, the node starts pinging nodes in the list, starting at the head. As soon as a node fails to respond, that node is evicted from the list to make way for the new node at the tail.

If there are no modifications to a particular k-bucket after a long period of time, the node does a `refresh` on the k-bucket. A refresh is a `lookup` of a random key in that k-bucket.

Lookup

In most DHTs, `lookup(key)` sends a single message and returns the information of a single node. The `lookup` operation in Kademlia differs in both respects: `lookup` is done in parallel and each node receiving a `lookup(key)` returns the k closest nodes to `key` it knows about.

A `lookup(key)` operation begins with the seeking node sending lookups in parallel to the α nodes from the appropriate k-bucket. Each of theses α nodes will asynchronously return the k closest nodes it knows closest to `key`. As lookups return their results, the node continue to send lookups until no new nodes[4] are found.

Joining

A joining node starts with a single contact and then performs a *lookup* operation on its own ID. Each step of the *lookup* operation yields new nodes for the joining node's peerlist and informs other nodes of its existence. Finally, the joining node performs a `refresh` on each k-bucket farther away than the closest node it knows of.

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

Fault-Tolerance

Nodes actively republish each file stored on the network each hour by rerunning the `store` command. To avoid flooding the network, two optimizations are used.

First if a node receives a `store` on a file it is holding, it assumes $k-1$ other nodes got that same command and resets the timer for that file. This means only one node republishes a file each hour. Secondly, `lookup` is not performed during a republish.

Additional fault tolerance is provided by the nature of the `store(data)` operation, which `puts` the file in the k closest nodes to the key. However, there is very little in the way of frequent and active maintenance other than what occurs during `lookup` and the other operations.

2.4 CAN

Unlike the previous DHTs presented in this chapter, the Content Addressable Network (CAN) [41] works in a d-dimensional torus, with the entire coordinate space divided among members. A node is responsible for the keys that fall within the "zone" that it owns. Each key is hashed into some point within the geometric space.

Peerlist and Geometry

CAN uses an exceptionally simple peerlist consisting only of neighbors. Every node in the CAN network is assigned a geometric region in the coordinate space and each node maintains a routing table consisting each node that borders the node's region. An example CAN network is shown in Figure 2.5

The size of the routing table is a function of the number of dimensions, $O(d)$. The lower bound on the routing tables size in a populated network (eg, a network with at least $2d$ nodes) is $\Omega(2d)$. This is obtained by looking at each axis, where there is at least one node bordering each end of the axis. The size of the routing table can grow as more nodes join and the space gets further divided; however, maintenance algorithms prevent the regions from becoming too fragmented.

Lookup

As previously mentioned, each node maintains a routing table corresponding to their neighbors, those nodes it shares a face with. Each hop forwards the lookup to the neighbor closest to the destination, until it comes to the responsible node. In a space that is evenly divided among n nodes, this simple routing scheme uses only $2 \cdot d$ space while giving average path length of $\frac{d}{4} \cdot n^{\frac{1}{d}}$. The overall lookup time of in CAN is bounded by $O(n^{\frac{1}{d}})$ hops[5].

[5] Around the same time CAN was being developed, Kleinberg was doing research into small world networks [24]. He proved similar properties for lattice networks with a single shortcut. What makes this network remarkable is lack of shortcuts.

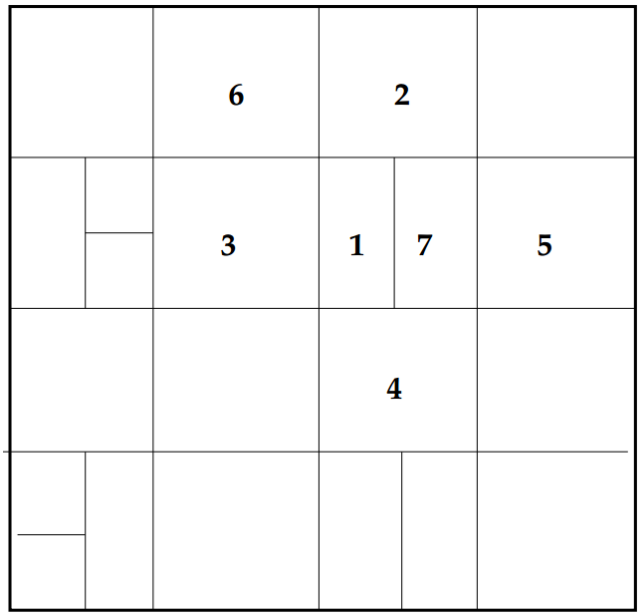


Figure 2.5: An example CAN network from [41].

If a node encounters a failure during lookup, the node simply chooses the next best path. However, if lookups occur before a node can recover from damage inflicted by churn, it is possible for the greedy lookup to fail. The fallback method is to use an expanding ring search until a candidate is found, which recommences greedy forwarding.

Joining

Joining works by splitting the geometric space between nodes. If node n with location P wishes to join the network, it contacts a member of the node to find the node m currently responsible for location P. Node n informs m that it is joining and they divide m's region such that each becomes responsible for half.

Once the new zones have been defined, n and m create its routing table from m and its former neighbors. These nodes are then informed of the changes that just occurred and update their tables. As a result, the join operation affects only $O(d)$ nodes. More details on this splitting process can be found in CAN's original paper [41].

Repairing

A node in a DHT that notifies its neighbors that its leaves usually has minimal impact to the network and in this is true for most cases in CAN. A leaving node, f, simply hands over its zone to one of its neighbors of the same size,

which merges the two zones together. Minor complications occur if this is not possible, when there is no equally-sized neighbor. In this case, f hands its zone to its smallest neighbor, who must wait for this fragmentation to be fixed.

Unplanned failures are also relatively simple to deal with. Each node broadcasts a heartbeat to its neighbors, containing its and its neighbors' coordinates. If a node fails to hear a heartbeat from f after a number of cycles, it assumes f must have failed and begins a `takeover` countdown. When this countdown ends, the node broadcasts[6] a `takeover` message in an attempt to claim f's space. This message contains the node's volume. When a node receives a `takeover` message, it either cancels the countdown or, if the node's zone is smaller than the broadcaster's, responds with its own `takeover`.

The general rule of thumb for node failures in CAN is that the neighbor with the smallest zone takes over the zone of the failed node. This rule leads to quick recoveries that affect only $O(d)$ nodes, but requires a zone reassignment algorithm to remove the fragmentation that occurs from `takeovers`.

To summarize, a failed node is detected almost immediately, and recovery occurs extremely quickly, but fragmentation must be fixed by a maintenance algorithm.

2.5 Pastry

Pastry [45] and Tapestry [56] are extremely similar use a prefix-based routing mechnism introduced by Plaxton et al. [40]. In Pastry and Tapestry, each key is encoded as a base 2^b number (typically $b = 4$ in Pastry, which yields easily readable hexadecimal). The resulting peerlist best resembles a hypercube topology [13], with each node being a vertice of the hypercube.

One notable feature of Pastry is the incorperation of a proximity metric. The peerlist uses IDs that are close to the node according to this metric.

Peerlist

Pastry's peerlist consists of three components: the routing table, a leaf set, and a neighborhood set. The routing table consists of $\log_{2^b}(n)$ rows with $2^b - 1$ entries per row. The ith level of the routing table correspond to the peers with that match first i digits of the example nodes ID.

Thus, the 0th row contains peers which don't share a common prefix with the node, the 1st row contains those that share a length 1 common prefix, the 2nd a length 2 common prefix, etc. Since each ID is a base 2^b number, there is one entry for each of the $2^b - 1$ possible differences.

For example, let is consider a node 05AF in system where $b = 4$ and the hexadecimal keyspace ranges from 0000 to FFFF.

- 1322 would be an appropriate peer for the 1st entry of level 0.

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

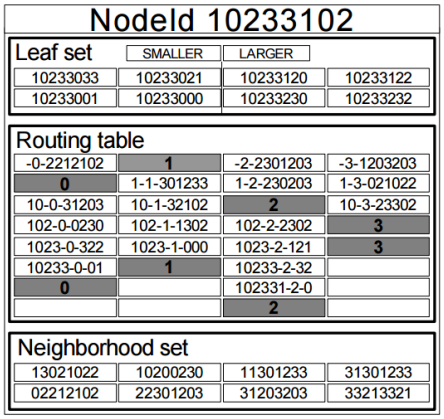


Figure 2.6: An example peerlist for a node in Pastry [45].

- 0AF2 would be an appropriate peer for the 10th[7] entry of level 1.
- 09AA would be an appropriate peer for the 9th entry of level 1.
- 05F2 would be an appropriate peer for the 2nd entry of level 3.

The leaf set is used to hold the L nodes with the numerically closest IDs; half of it for smaller IDs and half for the larger. A typical value for L is 2^b or 2^{b+1}. The leaf set is used for routing when the destination key is close to the current node's ID. The neighborhood set contains the L closest nodes, as defined by some proximity metric. It, however, is generally not used for routing. Figure 2.6 shows an example peerlist of a node in PAST.

Lookup

The `lookup` operation is a fairly straightforward recursive operation. The `lookup(key)` terminates when the `key` is falls within the range of the leaf set, which are the nodes *numerically* closest to the current node. In this case, the destination will be one of the leaf set, or the current node.

If the destination node is not immediately apparent, the node uses its routing table to select the next node. The node looks at the length l shared prefix, at examines the lth row of its routing table. From this row, the `lookup` continues with the entry that matches at least another digit of the prefix. In the case that this entry does not exist or has failed, the `lookup` continues from the closest ID chosen from the entire peerlist. This process is described by Algorithm 1. Lookup is expected to take $\lceil \log_{2^b} \rceil$, as each hop along the routing table reduces the search space by $\frac{1}{2^b}$.

[7] 0 is the 0th level.

Algorithm 1 Pastry lookup algorithm

```
Let L be the routing
function LOOKUP(key)
    if key is in the range of the leaf set then
        destination is closest ID in the leaf set or self
    else
        next ← entry from routing table that matches ≥ 1 more digit
        if next ≠ null then
            forward to next
        else
            forward to the closest ID from the entire peerlist
        end if
    end if
end function
```

Joining

To join the network, node J sends a `join` message to A, some node that is close according to the proximity metric. The `join` message is forwarded along like a `lookup` to the root of X, which we'll call *root*. Each node that received the `join` sends a copy of the their peerlist to J.

The leaf set is constructed from copying *root*'s leaf set, while ith row in the routing table routing table is copied from the ith node contacted along the `join`. The neighborhood set is copied from A's neighborhood set, as `join` predicates that A be close to J. This means A's neighborhood set would be close to A.

After the joining node creates its peerlist, it sends a copy to each node in the table, who then can update their routing tables. The cost of a `join` is $O(log_2^b n)$ messages, with a constant coefficient of $3 * 2^b$

Fault Tolerance

Pastry lazily repairs its leaf set and routing table. When node from the leaf set fails, the node contacts the node with largest or smallest ID (depending if the failed node ID was smaller or larger respectively) in the leaf set. That node returns a copy of its leaf set, and the node replaces the failed entry. If the failed node is in the routing table, the node contacts a node with an entry in the same row as the failed node for a replacement.

Members of the neighborhood set are actively checked. If a member of the neighborhood set is unresponsive, the node obtains a copy of another entry's neighborhood set and repairs from a selection.

2.6 Symphony and Small World Routing

Symphony [31] is a $1d$ ring-based DHT similar to Chord [51], but is constructed using the properties of small world networks [24]. Small world networks owe

their name to a phenomena observed by psychologists in the late 1960's.

Subjects in experiments were to route a postal message to a target person; for example the wife of a Cambridge divinity student in one experiment and a Boston stockbroker in another [35]. The messages were only to be routed by forwarding them to a friend they thought most likely to know the target. Of the messages that successfully made their way to the destination, the average path length from a subject to a participant was only 5 hops.

This lead to research investigating creating a network with randomly distributed links, but with a efficient lookup time. Kleinberg [25] showed that in a 2-dimensional lattice network, nodes could route messages in $O(\log^2 n)$ hops using only their neighbors and a single randomly chosen[8] finger. In other words, $O(\log^2 n)$ lookup is achievable with a $O(1)$ sized routing table.

Peerlist

Rather than the 2-dimensional lattice used by Kleinberg, Symphony uses a 1-dimensional ring[9] like Chord. Symphony assigns m-bit keys to the modular unit interval $[0, 1)$, instead of using a keyspace ranging from 0 to $2^n - 1$. This location is found with $\frac{hashkey}{2^m}$. This is arbitrary from a design standpoint, but makes choosing from a random distribution simpler.

Nodes know both their immediate predecessor and successor, much like in Chord. Nodes also keep track of some $k \geq 1$ fingers, but, unlike in Chord, these fingers are chosen at random. These fingers are chosen from a probability distribution corresponding to the expression $e^{ln(n)+(rand48()-1.0)}$, where n is the number of nodes in the network and `rand48()` is a C function that generates a random float?double between 0.0 and 1.0. Because n os difficult to compute due to the changing nature of P2P networks, each node uses an approximation is used based on the distance between themselves and their neighbors.

A final feature of note is that links in Symphony are bidirectional. Thus, if a node creates a finger to a peer, that peer creates a, so nodes in Symphony have a grand total of $2k$ fingers.

Joining and Fault Tolerance

The joining and fault tolerance processes in Symphony are extremely straightforward. After determining its ID, a joining node asks a member to find the root node for its ID. The joining node integrates itself in between its predecessor and successor and then randomly generates its fingers.

Failures of immediate neighbors are handled by use of successor and predecessor lists Failures for fingers are handled lazily and are replaced by another randomly generated link when a failure is detected.

[8] Randomly chosen from a specified distribution.

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

2.7 ZHT

One of the major assumptions of DHT design is that churn is a significant factor, which requires constant maintenance to handle. A consequence of this assumption is that nodes only store a small subset of the entire network to route to. Storing the entire network is not scalable for the vast majority of distributed systems due to bandwidth constraints and communication overhead incurred by the constant joining and leaving of nodes.

In a system that does not expect churn, the memory and bandwidth costs for each node to keep a full copy of the routing table are minimal. An example of this would be a data center or a cluster built for higher-performance computing, where churn would overwhelmingly be the result of hardware failure, rather than users quitting.

ZHT [29] is an example of such a system, as is Amazon's Dynamo [17]. ZHT is a "zero-hop hash table," which takes advantage of the fact that nodes in High-End Computing environments have a predictable lifetime. Nodes are created when a job begins and are removed when a job ends. This property allows ZHT to `lookup` in $O(1)$ time.

Peerlist

ZHT operates in a 64-bit ring, for a total of $N = 2^{64}$ addresses. ZHT places a hard limit of n on the maximum number of physical nodes in the network, which means the network has n partitions of $\frac{N}{n} = \frac{2^{64}}{n}$ keys. The partitions are evenly divided along the network.

The network consists of k physical nodes which each are running at least one instance (virtual nodes) of ZHT, with a combined total of i . Each instance is responsible for some span of partitions in the ring.

Each node maintains a complete list of all nodes in the network, which do not have to be updated very often due to the lack of or very low levels of churn. The memory cost is extremely low. Each instance has a 10MB footprint, and each entry for the membership table takes only 32 bytes per node. This means routing takes anywhere between 0 to 2 hops (explained below).

Joining

ZHT operates under a static or dynamic membership. In a static membership, no nodes will be joining the network once the network has been bootstrapped. Nodes can join at any time when ZHT is using dynamic membership.

To join, the joiner asks a random member for a copy of the peerlist The joiner can then determine which node is the most heavily overloaded. The joiner chooses an address in the network to take over partitions from that node.

Fault Tolerance

Fault tolerance exists to handle only hardware failure or planned departures from the network. Nodes backup their data to their neighbors.

2.8 Summary

We have seen that there are a wide variety of distributed hash tables, but they have some clearly defined characteristics that bind them all together. Table 2.1 summarizes the information presented in this chapter.

DHT	Routing Table Size	Lookup Time	Join/Leave	Comments
Chord [51]	$O(\log n)$, maximum $m + 2s$	$O(\log n)$, avg $(\frac{1}{2}\log n)$	$< O(\log n^2)$ total messages	m = keysize in bits, s is neighbors in 1 direction
Kademlia [34]	$O(\log n)$, maximum $m \cdot k$	$(\lceil \log n \rceil) + c$	$O(\log(n))$	This is without considering optimization
CAN [41]	$\Omega(2d)$	$O(n^{\frac{1}{d}})$, average $\frac{d}{4} \cdot n^{\frac{1}{d}}$	Affects $O(d)$ nodes	d is the number of dimensions
Plaxton-based DHTs, Pastry [45], Tapestry [56]	$O(\log_\beta n)$	$O(\lceil \log_{2^\beta} \rceil)$	$O(\log_\beta n)$	NodeIDs are base β numbers
Symphony [31]	$2k + 2$	average $O(\frac{1}{k}\log^2 n)$	$O(\log^2 n)$ messages, constant < 1	$k \geq 1$, fingers are chosen at random
ZHT [29]	$O(n)$	$O(1)$	$O(n)$	Assumes an extremely low churn
VHash	$\Omega(3d+1)+O((3d+1)^2)$	$O(\sqrt[d]{n})$ hops	$3d + 1$	approximates regions, hops are based least latency

Table 2.1: The different ratios and their associated DHTs

Chapter 3

Completed Work

In this chapter, we will cover our completed research. Our research has focused on novel implementation]and Distributed Hash Tables (DHTs). We have implemented and created an entirely new DHT called VHash [9], implemented MapReduce on Chord [43], and performed an analysis on an attack on DHTs.

3.1 VHash

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

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

Arguably all Distributed Hash Tables (DHTs) are built on the concept of Voronoi 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 [51] [34] [41] [45], but other metrics such as geographic location can be used as well [42].

These DHTs have carefully chosen metric spaces such that these regions are very simple to calculate. For example, Chord [51] and similar ring-based DHTs [31] utilize a unidirectional, one-dimensional ring as their metric space, such that the region for which a node is responsible is the region between itself and its predecessor.

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

Messages get routed along links to neighboring nodes. This would take $O(n)$ hops in one dimension. In multiple dimensions, our routing algorithm (Algorithm 2) is extremely similar to the one used in Ratnasamy et al.'s Content Addressable Network (CAN) [41], which would be $O(n^{\frac{1}{d}})$ hops.

Algorithm 2 Lookup in a Voronoi-based DHT

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

Efficient solutions, such as Fortune's sweepline algorithm [22], are not usable in spaces with 2 more dimensions. As far as we can tell, there is no way efficient to generate higher dimension Voronoi tessellations, especially in the distributed Churn-heavy context of a DHT. Our solution is the Distributed Greedy Voronoi Heuristic.

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 3.1. For additional information, Aurenhammer [4] provides a formal and extremely thorough description of Voronoi tessellations, as well as their applications.

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

During each cycle, nodes exchange their peer lists with a current neighbor and then recalculate their neighbors. A node combines their neighbor's peer

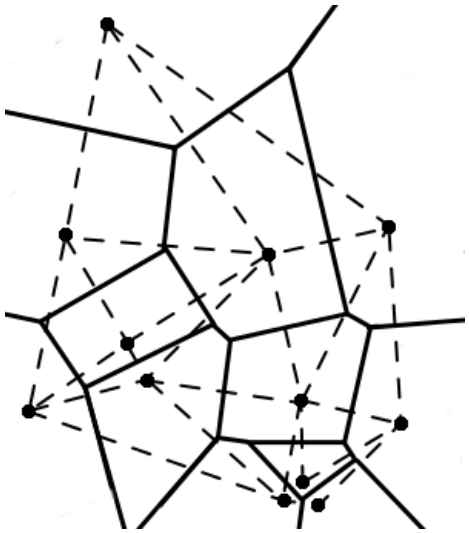


Figure 3.1: An example Voronoi diagram for objects on a 2-dimensional space. The black lines correspond to the borders of the Voronoi region, while the dashed lines correspond to the edges of the Delaunay Triangulation.

Algorithm 3 Distributed Greedy Voronoi Heuristic

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

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.

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 $\Theta(\frac{\log N}{\log \log N})$ [10] (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 $\Omega(1)$.

3.1.1 Experimental Results

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

[1] Our results are pulled directly from [8] and [9].

Convergence

Our first experiment examined how DGVH could be used to create a routing overlay and how well it performed in this task. The simulation demonstrated how DGVH formed a stable overlay from a chaotic starting topology after a number of cycles. We compared our results to those in RayNet [7]. The authors of Raynet proposed a random k-connected graph would be a challenging initial configuration for showing a DHT relying on a gossip mechanism could converge to a stable topology.

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

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

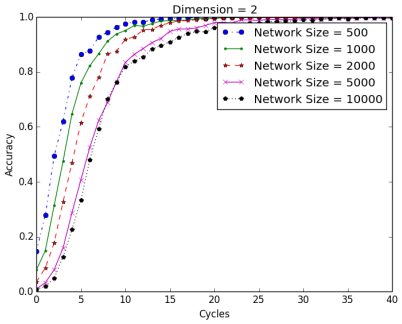
Algorithm 4 Routing Simulation Sample

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

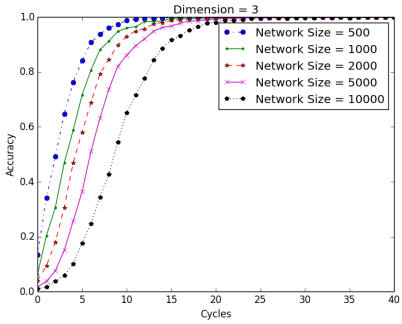
The results of our simulation are shown in Figures 3.2a, 3.2b, 3.2c, and 3.2d. Our graphs show that a correct overlay was quickly constructed from a random configuration and that our hit rate reached 90% by cycle 20, regardless of the number of dimensions. Lookups consistently approached a hit rate of 100% by cycle 30. In comparison, RayNet's routing converged to a perfect hit rate at around cycle 30 to 35 [7]. As the network size and number of dimensions each increase, convergence slows, but not to a significant degree.

Latency Distribution Test

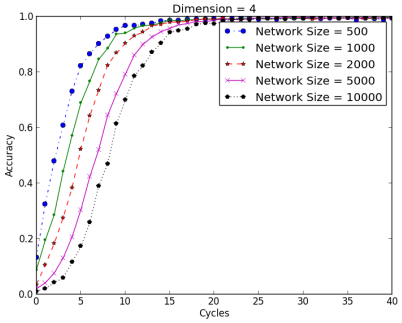
The goal of our second set of experiments was to demonstrate VHash's ability to optimize a selected network metric: latency in this case. In our simulation, we used the number of hops on the underlying network as an approximation of latency. We compared VHash's performance to Chord [51]. As we discussed



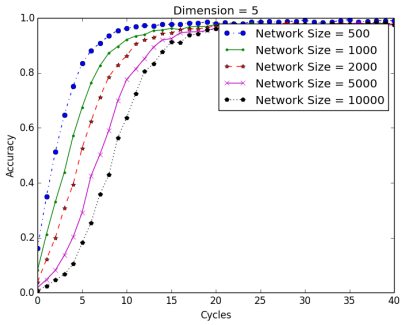
(a) This plot shows the accuracy rate of lookups on a 2-dimensional network as it self-organizes.



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



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



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

Figure 3.2: These figures show that, starting from a randomized network, DGVH forms a stable and consistent network topology. The Y axis shows the success rate of lookups and the X axis show the number of gossips that have occurred. Each point shows the fraction of 2000 lookups that successfully found the correct destination.

in Chapter 2 Chord is a well established DHT with an $O(\log(n))$ sized routing table and $O(\log(n))$ lookup time measured in overlay hops.

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

For this experiment, we constructed scale free network with 10000 nodes placed at random (which has an approximate diameter of 3 hops) as an underlay network [12] [38] [23]. We chose to use a scale-free network as the underlay, since scale free networks model the Internet's topology [12] [38]. We then chose a random subset of nodes to be members of the overlay network. Our next step was to measure the distance in underlay hops between 10000 random source-destination pairs in the overlay. VHash generated an embedding of the latency graph utilizing a distributed force directed model, with the latency function defined as the number of underlay hops between it and its peers.

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

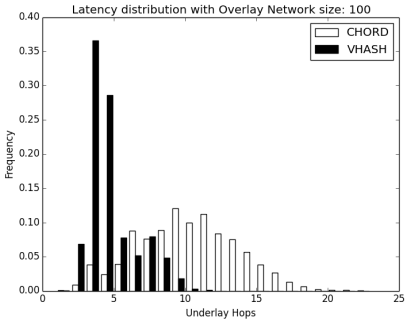
Figures 3.3a, 3.3b, and 3.3c show the distribution of path lengths measured in underlay hops in both Chord and VHash. VHash significantly outperformed Chord and considerably reduced the underlay path lengths in three network sizes.

We also sampled the lookup length measured in overlay hops for a 1000 sized Chord and VHash network. As seen in Figure 3.4, the paths measured in overlay for VHash were significantly shorter than those in Chord. In comparing the overlay and underlay hops, we find that for each overlay hop in Chord, the lookup must travel 2.719 underlay hops on average; in VHash, lookups must travel 2.291 underlay hops on average for every overlay hop traversed.

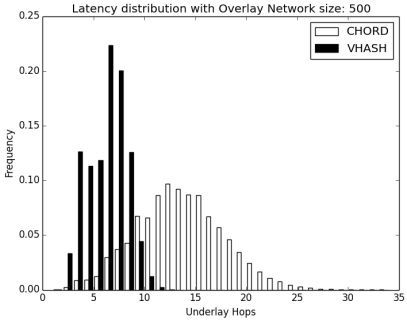
Recall that this work is based on scale free networks, where latency improvements are difficult. An improvement of 0.4 hops over a diameter of 3 hops is significant. VHash has on average less overlay hops per lookup than Chord, and for each of these overlay hops we consistently traverse more efficiently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

3.1.2 Remarks

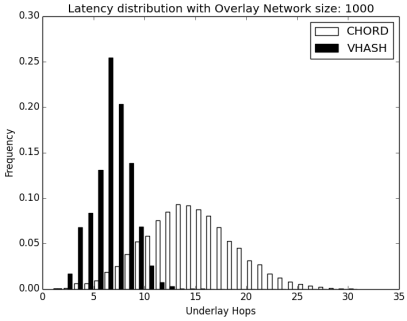
Voronoi tessellations have a wide potential for applications in ad-hoc networks, massively multiplayer games, P2P, and distributed networks. However, centralized algorithms for Voronoi tessellation and Delaunay triangulation are not applicable to decentralized systems. In addition, solving Voronoi tessellations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



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



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



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

Figure 3.3: Figures 3.3a, 3.3b, and 3.3c show the difference in the performance of Chord and VHash for 10,000 routing samples on a 10,000 node underlay network for differently sized overlays. The Y axis shows the observed frequencies and the X axis shows the number of hops traversed on the underlay network. VHash consistently requires fewer hops for routing than Chord.

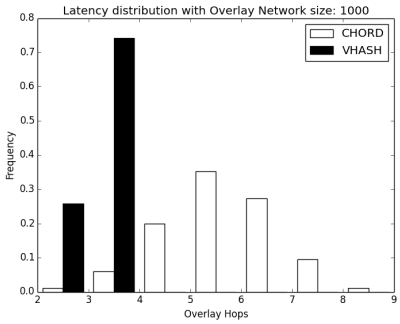


Figure 3.4: Comparison of Chord and VHash in terms of overlay hops. Each overlay has 1000 nodes. The Y axis denotes the observed frequencies of overlay hops and the X axis corresponds to the path lengths in overlay hops.

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

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

3.2 ChordReduce

DHTs have received a great deal of research due to their popularity as the backbone for structured P2P system primarily used for file-sharing. There are two recent and fairly open questions that we want to examine.

1. How and in what contexts can DHTs effectively be used for distributed computations?

2. How can nodes in a DHT autonomously detect and redistribute imbalances in the network load?

This section will talk about ChordReduce, in which we have made preliminary attempts to answer the first question by implementing MapReduce on a DHT. The next section, Section 3.3, discusses the our preliminary work in answering the second.

3.2.1 Background and Motivation

Distributed computing is a current trend and will continue to be the approach for intensive applications. We see this in the development of cloud computing [6],

volunteer computing frameworks like BOINC [3] and Folding@Home [26],and MapReduce [16]. Google's MapReduce in particular has rapidly become an integral part in the world of data processing. A user can use MapReduce to take a large problem, split it into small, equivalent tasks and send those tasks to other processors for computation. The results are sent back to the user and combined into one answer.

Popular platforms for MapReduce, such as Hadoop [1] [48], are explicitly designed to be used in large datacenters [2] and the majority of research has been focused there. However, as we have previously mentioned, there are notable issues with a centralized design.

First and foremost is the issue of fault-tolerance. Centralized designs have a single point of failure [48]. So long as all computing resources are located in one geographical area or rely on a particular node, a power outage or catastrophic event could interrupt computations or otherwise disrupt the platform [5].

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

There is no reason that these assumptions need to be the case for MapReduce, or for many distributed computing frameworks in general. Moving away from the data center context opens up more possibilities for distributed computing, such as P2P clouds [6]. However, without a centralized framework, the network needs some kind of protocol to organize the various components in the network. As part of our research, we developed a highly robust and distributed MapReduce framework based on Chord, called ChordReduce [43].

There a number of reasons to used a DHT as the protocol for a distributed computing platform. First, nodes ID and their location in the network are strongly bound to what data they are responsible for, such that any node can lookup which node is responsible a particular piece of data. This obviates the need for a centralized organizer to maintain this bit of metadata or assign backups for data, as nodes can do this autonomously. DHTs assume that the network is heterogeneous, rather than homogeneous. They have been used for over a decade for P2P file-sharing applications for these reasons.

3.2.2 What is ChordReduce?

ChordReduce [43] is designed as a more abstract framework for MapReduce, able to run on any arbitrary distributed configuration. ChordReduce leverages the features of distributed hash tables to handle distributed file storage, fault tolerance, and lookup. We designed ChordReduce to ensure that no single node is a point of failure and that there is no need for any node to coordinate the efforts of other nodes during processing.

File System

Our central design philosophy was to leverage as many features of the underlying DHT as possible. For example, we do not need to create a new distributed file system, as we can just use the DHT to hash file identifiers and use the DHT to store the file at the node responsible for that key.

If the file is large, we can instead use Dabek et al.'s Cooperative File System or CFS [15]. In CFS, files are split into approximately equally sized blocks. Each block is treated as an individual file and is assigned a key equal to the hash of its contents. The block is then stored at the node responsible for that key. The node that 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.

Computation

ChordReduce treats each task or target computation as an object of data. This means we can distribute them in the same manner as files and rely on the protocol to route them and provide robustness.

In ChordReduce, each node takes on responsibilities of both a worker node and master node, in the same way that a node in a P2P file-sharing service acts as both a client and a server. A user starts a job, 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. We call this node the *stager* for this particular job.

The job of the stager is to divide the work into *data atoms*, the smallest units of work. This might represent a block of text, the result of a summation for a particular intermediate value, or a subset of items to be sorted. The specifics of how to divide the work are defined by the user in a *stage* function. The data atoms also contain user created Map and Reduce functions.

If the user wants to perform a MapReduce job on a particular file on the network, the stager locates the keyfile for the data and creates a data atom for each block in the file. Each data atom is then sent to the node responsible for their corresponding block. When the data atom reaches its destination node, that node retrieves the necessary data and applies the Map function. The results are stored in a new data atom, which are then sent back to the stager's hash address (or some other user defined address). This will take $O(\lg n)$ 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.

Some MapReduce jobs do not rely on a file stored on the network, such as a Monte-Carlo approximation. In this situation, the data atoms define a task

to be run multiple times. In this case, the created 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 the 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. This eliminates the need for a single specific root node and provides fault tolerance.

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 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.

Robustness

Since the system is distributed, we need to assume that any member of the network can go down at any time. When a node fails or leaves Chord, the failed node's successor will become responsible for all of the failed nodes keys. Likewise, 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 sends data that it is currently responsible for. This 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 computational tasks 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. Recall that a node's ID in the network does not map to a geographical locations. Any failure that affects multiple nodes

simultaneously would be spread uniformly throughout the network. This means if successive nodes to fail simultaneously, they do so independently.

Assume 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 simultaneously. The chances of this would be $r^s + 1$, as each failure would be independent.

A final consequence of this is load-balancing during runtime. When a joining node n find 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 is responsible for and begin performing Map tasks immediately.

3.2.3 Experiments

We created a prototype of ChordReduce in order to demonstrate it was a viable framework [43]. To acheive this, we had to show ChordReduce had these three properties:

1. ChordReduce provided significant speedup during a distributed job.
2. ChordReduce scaled.
3. ChordReduce handled churn during execution.

We needed to demonstrate speedup by showing that a job handled by multiple workers generally finished sooner 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 needed to show that the cost of distributing the work grows logarithmically with the number of workers. We needed to demonstrate that the larger the job is, the more nodes can work on the problem before we begin experiencing diminishing returns. 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 had to show that ChordReduce can handle arbitrary node failure in the ring and that such failures minimally impacted the overall speed of computation

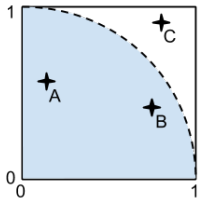


Figure 3.5: The "dartboard." The computer throws a dart by choosing a random x and y between 0 and 1. If $x^2 + y^2 < 1^2$, the dart landed inside the circle. A and B are darts that landed inside the circle, while C did not.

Setup

To stress test our framework, we ran a Monte-Carlo approximation of π. This process is analogous to having a square containing the top-right quadrant of a circle (Fig. 3.5), and then throwing darts at random locations. Counting the ratio of darts that land inside the circle to the total number of throws yields an approximation of $\frac{\pi}{4}$. The more darts we throw, 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, we can double the amount of work each node gets. We could also test the effectiveness of distributing the job among different numbers of workers.

Each Map job was defined as the number of throws the node must make and yielded total number of throws and the number of throws that landed inside the circular section. Reducing these results was then a matter of adding the respective fields together.

We ran our experiments using Amazons's Elastic Compute Cloud (EC2) service. Amazon EC2 allows users to purchase an arbitrary amount of virtual machines by the hour. Each node was an individual EC2 small instance with a preconfigured Ubuntu 12.04 image.

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

We ran our experiments on groups of 1, 10, 20, 30, and 40 workers, which

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

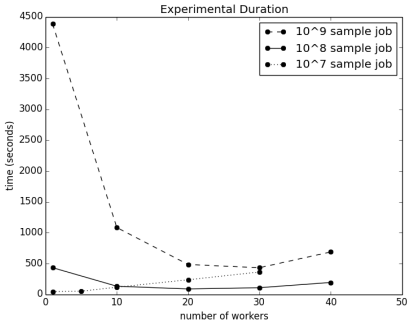


Figure 3.6: Our results show that for a sufficiently large job, it was almost always preferable to distribute it. When the job is too small, such as with the 10^7 data set, our runtime is dominated by the overhead. Our results are what we would expect when overhead grows logarithmically to the number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

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

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

Fig. 3.6 and Fig. 3.7 summarize the experimental results of job duration and speedup. Our default series was the 10^8 samples series. On average, it took a single node 431 seconds, or approximately 7 minutes, to generate 10^8 samples. Generating the same number of samples using ChordReduce over 10, 20, 30, or 40 nodes was always quicker. The samples were generated fastest when there were 20 workers, with a speedup factor of 4.96, while increasing the number of workers to 30 yielded 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, which experienced a speedup of only 2.25.

Since our data showed that approximating π on one node with 10^8 samples took approximately 7 minutes, collecting 10^9 samples on a single node would take 70 minutes at minimum. Fig. 3.7 shows that the 10^9 set gained greater benefit from being distributed than the 10^8 set, with the speedup factor at 20 workers being 9.07 compared to 4.03. In addition, dimishing returns only showed up at 40 workers, compared with the 10^8 data set, which began its drop off at 30 workers. This behavior showed 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

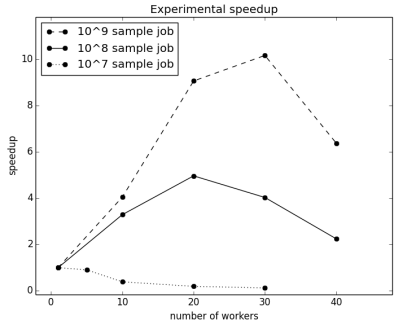


Figure 3.7: The larger the size of the job, the greater the gains of distributing with ChordReduce. In addition, the larger the job, the more workers can be added before we start seeing diminishing returns. This demonstrates that ChordReduce is scalable.

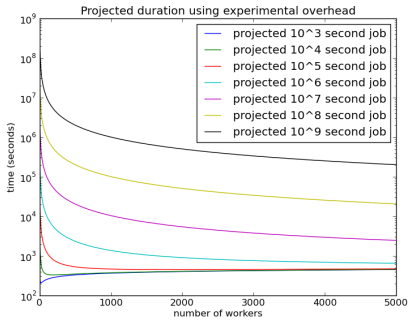


Figure 3.8: The projected runtime using ChordReduce for differently sized jobs. Each curve projects the expected behavior for job that takes a single worker the specified amount of time.

overheard acting as the dominant factor in runtime. This matches the behavior predicted by our equation, $T_n = \frac{T_1}{n} + k \cdot \log_2(n)$. For a small T_1, $\frac{T_1}{n}$ approaches 0 as n gets larger, while $k \cdot \log_2(n)$, our overhead, dominates the sample. The samples from our data set fit this behavior, establishing that our overhead increases logarithmically with the number of workers.

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

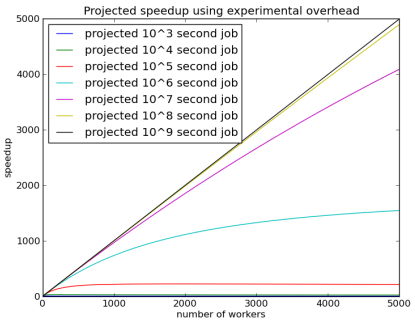


Figure 3.9: The projected speedup for different sized jobs.

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

jobs become larger, the expected speedup from ChordReduce approaches linear behavior.

Table 3.1 shows the experimental results for different rates of churn. We discus these experimental results and their significance in Section 3.3.

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.

3.2.4 Heterogeneity Calculation

One of the advantages to using homogeneous hardware is that each machine, each core, each node is the same. To evenly distribute the workload, you just have to give each machine the same amount of work. While a

This is more difficult in a heterogeneous system, such as ChordReduce, as each machine can shoulder a different amount of work. How do we distribute

work evenly across a heterogeneous system?

We can solve this by adjusting the amount of nodes representing each machine in the network. Machines that can handle a larger load create more nodes in the network. Besides solving the heterogeneous load-balancing problem, increasing the number of nodes in the system increases the overall load-balancing of the system.

The question we must answer is "how?" We need to create some unit of measurement for a distributed computing system and research if any other researchers have asked this problem. Furthermore, this measurement might need to be relative to other nodes in the network, since the only basis for comparison are the scores of the peers. Finally, this process needs to be handled autonomously by each node. This is part of the proposed work discussed in Chapter 4.

3.3 Autonomous Load Balancing

During our experiments testing the capabilities of ChordReduce, we experienced a significant and completely unexpected anomaly while testing churn. One of the things previous research [32] [27] in the same area we felt we needed to explore better was how a completely decentralized computation could handle churn. Now, despite our initial prototype having numerous bugs and only able to handle small networks, we were fairly certain of it's ability to handle churn.

Marozzo et al. [32] tested their network using churn rates of 0.025%, 0.05%, 0.1%, 0.2%, and 0.4% per minute. The churn rate of $cr << 1$ per minute means that each minute on average, $cr \cdot n$ nodes leave the network and $cr \cdot n$ new nodes join the network.[3] This could effectively be thought of as each node flipping a weighted coin every minute. When the coin lands on tails, the node leaves. A similar process happens for nodes wanting to join the network.

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

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

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

Hypothesis

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

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

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

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

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

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

This leads us to two hypotheses:

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

The first hypothesis is mentally pleasing to anyone who has tried to create a distributed system, but lacks rigor. We still have to verify the existence of this phenomena in an independent experiment, and establish that it does is part of the proposed work (Chapter 4).

Once we have established that it does exist, we need a better load-balancing strategy than randomly inducing. We want nodes to have a precomputed list of locations in which they can insert nodes to perform load-balancing on an ad-hoc basis during runtime. This precomputed list ties directly into the security research on DHTs we have done [44].

3.4 Sybil Attacks and Injection

One of the key properties of structured peer-to-peer (P2P) systems is the lack of a centralized coordinator or authority. P2P systems remove the vulnerability of a single point of failure and the susceptibility to a denial of service attack [18], but in doing so, open themselves up to new attacks.

Completely decentralized P2P systems are vulnerable to *Eclipse attacks*, whereby an attacker completely occludes healthy nodes from one another. This prevents them from communicating without being intercepted by the adversary. Once an Eclipse attack has taken place, the adversary can launch a variety of crippling attacks, such as incorrectly routing messages or returning malicious data [49].

One way to accomplish this attack is to perform a *Sybil attack* [18]. In a Sybil attack, the attacker masquerades as multiple nodes, effectively over-representing the attacker's presence in the network to maximize the number of links that can be established with healthy nodes. If enough malicious nodes are injected into the system, the majority of the nodes will be occluded from one another, successfully performing an Eclipse attack.

This vulnerability is well known [53]. Extensive research has been done assessing the damage an attacker can do after establishing themselves [49]. Little focus has been done on examining how the attacker can establish himself in the first place and precisely how easily the Sybil attack can be accomplished.

We did a project that focused on looking at the computational and memory costs of performing the Sybil attack. The computation costs turn out to be fairly trivial and can be precomputed based on how IDs are assigned, a process we named *mashing*. If a node obtains their ID via an IP/Port combination, and we limit an attacker to using only ephemeral IP addresses (16383 total), the per

node cost of mashing is quite low. Per node, it takes 48 milliseconds to mash 16383 IP/Port combinations and only 352 kilobytes to store this information after precomputing it.

An attacker would do this for each of his nodes, then join the network and insert as many Sybils as possible. We calculated that it would take only 1221 IP addresses to compromise 50% of the links in a 20,000,000 node network [44].

3.4.1 Experiments

The primary experiment of our project was simulating the complete eclipse a network using a Sybil attack, starting with a single malicious node [44]. We simulated a network of n nodes, each represented by an ID generated by SHA1 of a random IP/port combination.

The goal of the attacker was to mash as many pairs of adjacent nodes as possible. We call this the *Nearest Neighbor Eclipse* since the attacker seeks to become the nearest neighbors of each node.

The attacker was given num_ips randomly generated IP addresses, but could use any port between 49152 and 65535. This means attacker had $16383 \cdot num\_ips$ Sybils at his disposal. Each of these addresses could be precomputed by the attacker and stored in a sorted list, requiring only 352 kilobytes per IP.

The adversary in this attack chooses any random hash key as a starting point to "join" the network. This is their first Sybil and the join process provides information about a number of other nodes. Most importantly, nodes provide information about other nodes that are close to it. The adversary uses this information to inject Sybils in between successive healthy nodes. For example, in Pastry, a joining node typically learns about the 16 nodes closest to it for fault tolerance, in addition to all the other nodes it learns about [45]. In Chord, this number is a system configuration value r [51].

We simulated this attack on networks of up to 20 million nodes. We chose 20 million since it falls neatly into the 15-27 million user range seen on Mainline DHT [54]. We gave the attacker access to up to 19 IP addresses. Our results are in Figures 3.10 and 3.11 and are taken from our paper on this attack [44].

Our results show that an adversary, given only modest resources, can inject a Sybil in between the vast majority of successive nodes in moderately sized networks. In a large network, modest resources still can be used to compromise more that a third of the network, an important goal if the adversary wishes to launch a Byzantine attack.

3.4.2 Ramifications

Our analysis and experiments show that an adversary with limited resources can easily compromise a P2P system and occlude the majority of the paths between nodes. We can turn this attack around and use to benefit a DHT. Some nodes will be responsible for larger regions than others and therefore will be responsible for a larger portion of the data. If a node can detect when a peer is overloaded, the node can inject a virtual node into the region to shoulder

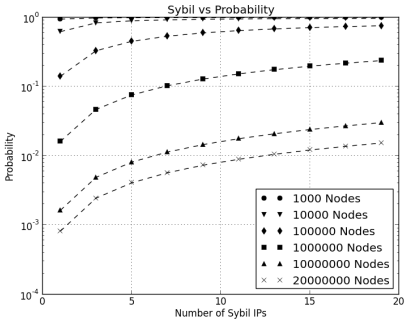


Figure 3.10: Our simulation results. The x-axis corresponds to the number of IP addresses the adversary can bring to bear. The y-axis is the probability that any chosen region has been mashed. Each line maps to a different network size of n. The dashed line corresponds to values $P_{bad\_neighbor} = \frac{num\_ips \cdot 16383}{num\_ips \cdot 16383 + n - 1}$, which is the probability a node has a malicious neighbor

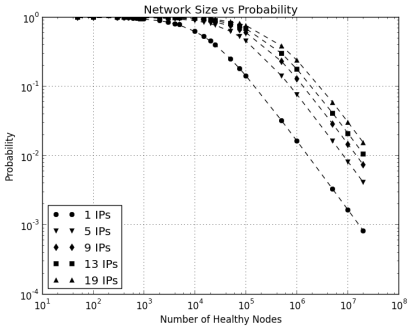


Figure 3.11: These are the same as results shown in Figure 3.10, but our x-axis is the network size n in this case. Here, each line corresponds to a different number of unique IP addresses the adversary has at their disposal.

some of the load. The load could be defined by the size of the region or by the volume of traffic.

A network implementing this load-balancing strategy would be self-adaptive. Nodes in this type of self-adaptive network would have a limited number of virtual nodes to mash. This limit would protect nodes from becoming overloaded themselves and ensure network stability. We discuss this further in Chapter 4.

3.5 Summary

Our previous work has focused primarily on various aspects of Distribute Hash Tables. We can categorize our research into three distinct, but connected parts:

- Generalizing DHTs.
- Distributed Computing on a DHT.
- Autonomous Load Balancing.

It is each of these three parts we wish to further research.

Chapter 4

Proposed Work

In this chapter, we will present the three parts we propose for my research. Each part can be completed independently of each other, but are related as to be much more beneficial to complete together.

4.1 UrDHT: Our DHT Framework

As mentioned in Chapter 1, our plan is to create a highly configurable and easy to use DHT framework based off the DHT abstractions we have discovered. Rather than making a fully-functional DHT application on our own, we will be making a minimally functional DHT framework that will be easy to fork for a variety of applications.

We call this proposed framework UrDHT, *ur-* being the Germanic prefix denoting primal, or primitive, or original. UrDHT is a project that presents a minimal and extensible implementation for all the essential components for a DHT: the different aspects for a protocol definition, the storage of values, and the networking components. Every DHT has the same components, but there has yet to be an all-encompassing framework that clearly demonstrates this.

- This will be done jointly with Brendan and anyone else who is interested in creating a completely open source framework for DHTs.
- In particular, my focus on the project will be implementing each of the DHTs I plan on using to test Distributed Computing on DHTs. This will require a formal description of each DHT and their components.
- The goal of this step is **not only** to create a DHT, but to create an easily extensible abstract framework for DHTs.
- The abstraction comes from implementing the relationship we found between DHT spheres of responsibility and Voronoi tessellations. This is the key point of the project. Our previous research [9] has led us to assert

that there is a mathematical formulation for different aspects which every DHT shares in common, such as a distance metric and closeness definition.

There are two clear goals of this project: the mathematical definitions for distributed hash tables and the UrDHT application itself. The UrDHT project is probably the stronger goal, since it is a fully fledged and novel framework. Furthermore, as an open-source application, we it will be useful to many developers since it will provide an orderly way to create new DHTs and DHT-based applications.

The mathematical formulations, on the other hand, serve as novel formulations and definitions of DHTs. They provide new insight, but do not serve as a new application or framework. However, the formulations can and should be presented as an atomic unit of research.

4.2 Distributed DHT Computing

The next step is to use the UrDHT framework to re-implement ChordReduce. Our goal is a DHT based platform for solving embarrassingly parallel problems using DHTs. The steps involved in this are listed below.

- We will use UrDHT to implement a few of the more popular DHTs.
  - We want to compare each of the DHTs to see if there is a difference between using one or another for distributed computing.
  - Using UrDHT for all the implementations will minimize the non-protocol differences between each DHT, which will allow for as fair a comparison as possible.
  - Additionally this will serve as an example of how to implement our framework.
- Implement a distributed computing mechanism on each of the implemented DHTs for solving computing tasks.
  - The emphasis of our distributed computing application is robustness and fault-tolerance.
- Test each framework using a variety of embarrassingly parallel problems, such as:
  - Brute-force cryptanalysis.
  - MapReduce problems.
  - Monte-Carlo computations.

Here there is one very clear goal. This would be a reimplementation of our original ChordReduce paper with a completely new problems and new results.

4.3 Autonomous Load Balancing

As discussed earlier, it would be highly beneficial for each node to take advantage of their processing power. The most straightforward way of doing this in a DHT context is to have more powerful nodes acquire a larger stretch of the network space. The goals here are straightforward:

- Further establish that the "strategy" of randomly inducing churn works at reducing. If we successfully establish it exists, we must determine if the process can be modeled.
- Create a processor scoring mechanism for creating virtual nodes. Essentially, we need to create a mechanism to estimate how many virtual nodes a particular piece of hardware can handle.
- Use this score in conjunction with various implemented strategies for autonomous load-balancing to find the most effective. A few of the strategies we will examine:
  - Passive load balancing - Here, each node looks at range of its Sybil locations. Based on its score, it choses the k largest locations and injects Sybils. No actual communication with other nodes is used.
  - IRM [47] based strategy.
  - Invitation based - Here we flip the strategy around. If the node detects its region is too large or it has too much work; it invites other nodes to help. The nodes look at the range in question and offer to help if they have a Sybil that fits and are not overloaded themselves. The inviter looks at the offers of help and selects the best candidate.
  - Another invitation based scheme is that each of the nodes submits itself as a potential filler to each of it's sybil locations. The nodes that would be affected by this node's entry add it to a list of fillers. When the node decides it needs help, it selects of of the fillers to join. These invitation based schemes have the advantage of nodes having control of their range. The question is it "turtles all the way down" here? Do we let the replicas also call for help?
  - Using the force spring model used in VHash [9].
- We must determine under which contexts this kind of highly responsive load-balancing can be used. Is it only useful during distributed computation, or is it useful in file-sharing as well?

Once we have created the various load balancing strategies, we would need to implement them and test them against each other. Randomly induced churn would serve as a baseline strategy. We can then present which strategy is most effective at evenly spreading the network's load.

Chapter 5

Conclusion

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

We have shown in the previous chapters that we were able to create a prototype distributed computing application [43] based on the Chord DHT [51]. ChordReduce gave us several new questions to ask, such as does it matter which DHT is used for distribute computation and how can we have nodes autonomously accept new work from overloaded nodes? To effectively answer these questions we need to create a new framework for creating DHT applications, which we dub UrDHT.

UrDHT will be an invaluable resource to any other developer who wishes to create a DHT application. By implementing a distributed computing application using UrDHT, we will create a completely decentralized framework for doing distributed computing. This will allow distributed computing to take place not just in data centers, but within P2P context as well.

Bibliography

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

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

[3] David P Anderson. Boinc: A system for public-resource computing and storage. In *Grid Computing, 2004. Proceedings. Fifth IEEE/ACM International Workshop on*, pages 4–10. IEEE, 2004.

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

[5] Ozalp Babaoglu and Moreno Marzolla. The people's cloud. *Spectrum, IEEE*, 51(10):50–55, 2014.

[6] Ozalp Babaoglu, Moreno Marzolla, and Michele Tamburini. Design and implementation of a p2p cloud system. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 412–417, New York, NY, USA, 2012. ACM.

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

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

[10] 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.

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

[12] Reuven Cohen, Keren Erez, Daniel Ben-Avraham, and Shlomo Havlin. Resilience of the internet to random breakdowns. *Physical review letters*, 85(21):4626, 2000.

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

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

[15] 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.

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

[17] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 205–220. ACM, 2007.

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

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

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

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

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

[23] Aric Hagberg, Dan Schult, Pieter Swart, D Conway, L Séguin-Charbonneau, C Ellison, B Edwards, and J Torrents. Networkx. high productivity software for complex networks. *Webová strá nka https://networkx. lanl. gov/wiki*, 2004.

[24] Jon Kleinberg. The small-world phenomenon: An algorithmic perspective. In *Proceedings of the thirty-second annual ACM symposium on Theory of computing*, pages 163–170. ACM, 2000.

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

[26] Stefan M Larson, Christopher D Snow, Michael Shirts, et al. Folding@ home and genome@ home: Using distributed computing to tackle previously intractable problems in computational biology. 2002.

[27] 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.

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

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

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

[31] 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.

[32] 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.

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

[34] 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.

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

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

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

[38] Romualdo Pastor-Satorras and Alessandro Vespignani. Epidemic spreading in scale-free networks. *Physical review letters*, 86(14):3200, 2001.

[39] C. Greg Plaxton, Rajmohan Rajaraman, and Andréa W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '97, pages 311–320, New York, NY, USA, 1997. ACM.

[40] C Greg Plaxton, Rajmohan Rajaraman, and Andrea W Richa. Accessing nearby copies of replicated objects in a distributed environment. *Theory of Computing Systems*, 32(3):241–280, 1999.

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

[42] 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.

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

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

[45] 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.

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

[47] Haiying Shen. Irm: Integrated file replication and consistency maintenance in p2p systems. In *Computer Communications and Networks, 2008. ICCCN '08. Proceedings of 17th International Conference on*, pages 1–6, Aug 2008.

[48] 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.

[49] Mudhakar Srivatsa and Ling Liu. Vulnerabilities and security threats in structured overlay networks: A quantitative analysis. In *Computer Security Applications Conference, 2004. 20th Annual*, pages 252–261. IEEE, 2004.

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

[51] 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.

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

[53] Guido Urdaneta, Guillaume Pierre, and Maarten Van Steen. A survey of dht security techniques. *ACM Computing Surveys (CSUR)*, 43(2):8, 2011.

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

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

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