Proposal

Andrew Rosen

April 6, 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 framework 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 redistributed 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 Computing is well understood to be the approach to take to solve large problems. Many problems can be broken up into multiple parts that can be solved simultaneously, yielding a much quicker result than a single working attacking the problem. However, there are two broad obstacles in distributed computing.

The first is figuring out how the mechanics of efficiently distributing a problem to multiple workers and asynchronously coordinating their effort. The second is creating and maintaining the computation platform itself.

Some specific challenges are:

Scalability - Distributed computing platforms should not be completely static; if the platform would be improved by the addition of a new resource, it should be possible to add that resource. The addition of new workers in a distributed computing framework should be a minimally disruptive process. This ties into the network's fault tolerance

Load-Balancing This is one of the most important issues to consider when creating a framework for distributed computing. How do you split up a problem then distribute it so that no single worker is under or over-utilized? Failing that, how do you minimize the imbalance in work? Is there a way to do so at run time?

Fault Tolerance Even in a network that is expected to remain static for long periods of time, the platform still has to deal with failure. In a centralized environment, hardware failures are common given enough machines. 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.

Fortunately, these challenges are not unique to distributed computing but are also obstacles in distributed file storage. In particular, distributed file storage applications that utilize Distributed Hash Tables are designed to handle these particular challenges.

1.1 Distributed Hash Tables

Distributed Hash Tables (DHTs) are traditionally used as the backbone of structured Peer-to-Peer (P2P) file-sharing applications. The largest such application by far is Bittorrent [1], which is built using Mainline DHT [2], a derivative of Kademlia [3]. The number of users on Bittorrent ranges from 15 million to 27 million users daily, with a turnover of 10 million users a day [4].

Most research on DHTs assumes that they will be used in the context of a large P2P file-sharing application (or at least, an application *potentially* incorporating millions of nodes). This lends the DHT to having particular qualities. The network must be able to handle members joining and leaving arbitrarily. The resulting application must be agnostic towards hardware. The network must be decentralized and split whatever burden there is equally among its members.

In other words, distributed hash tables provide scalability, load-balancing, robustness, and heterogeneity to an application. More recent applications have examined leveraging these qualities, since these qualities are desirable in many different frameworks. For example, one paper [5] used a DHT as the name resolution layer of a large distributed database. Research has also been done in using DHTs as an organizing mechanism in distributed machine learning [6].

I describe each of the aforementioned qualities and their ramifications below in sections 1.1.1, 1.1.2, 1.1.3, and 1.1.2 . While these properties are be individually enumerated, they are greatly intertwined and the division between their impacts can be somewhat arbitrary.

1.1.1 Robustness and Fault-Tolerance

One of the most important assumptions of DHTs is that they are deployed on a non-static network. DHTs need to be built to account for a high level what is called *churn*. Churn refers to the disruption of routing caused by the constant joining and leaving of nodes. 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 information stored on the DHT (see the Load Balancing property below).

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 neighbors can immediately assume responsibility. The join process also causes disruption to the network, as affected nodes have adjust their peerlists to accommodating the joiner.

The last property is that the hash algorithm used to distribute content evenly across the network(again see load balancing) also distributes nodes evenly across the DHT. This means that nodes in the same geographic region occupy vastly different positions in the keyspace. If an entire geographic region is affected by a network outage, this damage is spread evenly across the DHT, which can be handled.

This property is the most important, as it deals with failure of entire sections of the network, rather than a single node. Recent research in using DHTs for High End Computing [7] shows what can happen if we remove this assumption by placing the network that is almost completely static.

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 [8].

1.1.2 Load Balancing

All Distributed Hash Tables use some kind of consistent hashing algorithm to 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 and generates keys in a uniform manner. The result of this is that as more nodes join the network, the distribution of nodes in the keyspace becomes more uniform, as does the distribution of files.

However, because this is a random process, it is highly unlikely that each node will be spread evenly throughout the network. This appears to be a weakness, but can be turned into an advantage in heterogeneous systems by using *virtual nodes* [9] [10]. When a node joins the network, it joins not at one position, but multiple virtual positions in the network [10]. Using virtual nodes allows load-balance optimization in a heterogeneous network; more powerful machines can create more virtual nodes and handle more of the overall responsibility in the network.

DeCandia et aldiscussed various load balancing techniques that were tested on Dynamo [10]. Each node was assigned a certain number of tokens and the node would create a virtual node for each token. The realization DeCandia et alhad was that there was no reason to use the same scheme for data partitioning and data placement. DeCandia et alintroduced two new strategies which work off assigning nodes equally sized partitions.

Under these schemes, each virtual node maps to an ID as before, but the patitions each node is responsible for are equally sized[1].

Heterogeneity

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, while the load-balancing process defined in DHTs treats

[1] Need help here

each node equally. It is much simpler to treat each node as equal unit. In other words, DHTs support heterogeneity, but do not attempt to exploit it.

This doesn't 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. However, this is not a feasible option for any kind of truly decentralized system and would need to be done automatically. There is no well-known mechanism to that exists to automatically allocate virtual nodes on the fly [2]. A few options present themselves.

One is to use adapt a request tracking mechanism, such as what is used in IRM, except instead of tracking file requests, it tracks requests that are directed to a particular (real) node. If a particular (real) node receives an inordinate amount of requests, the node doing the detecting suggests that the node obtain another token/create another virtual node. Another strategy is to use the preference lists/successor predecessor lists, and observe the distribution of the workload, adjusting the virtual nodes based on that.

Dynamic load balancing may not be essential to P2P file-sharing applications, but is absolutely essential to any kind of P2P distributed computation. In our ChordReduce experiments, we observed that just approximating dynamic load-balancing by simulating high levels of churn noticeably improved results[3].

1.1.3 Scalability

In order to maintain scalability, a DHT has to ensure that as the network grows larger:

- Churn does not have a disproportionate overhead. For example, in a 1000 node network, a joining or leaving node will affect only an extremely small subset of these nodes. [4]
- Lookup request speeds (usually measured in hops) grow by a much smaller amount, possibly not at all.

Using consistent hashing allows the network to scale up incrementally, adding one node at a time [10]. 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. This is almost instantaneous if the network is using backups. 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 which play the greatest role in driving the variety of DHT architectures. Each DHT must strikes a balance between memory cost of the

[2] citation needed, although this can be similar to IRM

[3] We found this by accident, just by testing the network's fault tolerance in regards to a high level of churn

[4] We will see that this requirement can be relaxed in very specific cases [7].

peerlist and lookup time. The vast majority of DHTs choose to use $\lg(N)$ sized routing tables and $\lg(n)$ hops[5]. Chapter 2 discusses these tradeoffs in greater detail and how they affect the each DHT.

1.2 Hypothesis: problems in distributed computing + solutions = dissertation topic

Distributed computing platforms need to be scalable, fault-tolerant, and load balancing. In addition, the ability to incorporate heterogeneous hardware is a definite benefit. Distributed Hash Tables can provide an application with all of these qualities. P2P applications have been using DHTs for large-scale distributed file sharing applications for years now and are particularly effective.

I propose that DHTs can be used to create P2P distributed computing platforms that are completely decentralized. Rather than keys being assigned to some data, we can assign keys to tasks and automatically distribute those tasks to the responsible nodes There would be no need for some central coordinator or scheduler.

A successful DHT based computing platform would need to address the problem of dynamic load-balancing. This is currently an unsolved problem and if an application can dynamically reassign work to nodes added at runtime, this opens up new options for resource management. If a computation is running too slow, new nodes can be added to the network during runtime or idle nodes can boot up more virtual nodes (now that I think of it this is two different but highly related problems: internal and external).

The next chapter will delve into how DHTs work and examine specific DHTs. The remainder of the paper will then discuss the work I plan on doing to demonstrate the viability of using DHTs for distributed computing.

[5] log n or log N, matters

Chapter 2

Background

DHTs have been a vibrant area of research for the past decade, with some of the concepts dating further back. Numerous DHTs have been developed over the years. This is partly because the process of designing DHTs involves making tradeoffs, with no choice being strictly better than any other.

The large number of DHTs have lead many papers use different terms to describe congruent elements of DHTs, as some terms may make sense only in one context. Since this paper will cover multiple DHTs that would use different terms, I've created a unified terminology:

key - The identifier generated by a hash function corresponding to a unique[1] node or file.

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 paper, I try to 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 [11] [12] overload the terminology. Any table or list of peers is a subset of the entire peerlist.

Neighbors - 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 [8], this is the node's *predecessor(s)* and *successor(s)*.

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

Root Node - The node responsible for a particular key.

[1] Unique with extremely high probability. SHA-1, which generates 160-bit hashes, is typically used as a hashing algorithm.

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 it directly. However, many implementations use backup operations and caching which will store multiple copies of the value along the network If we don't 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 doesn't 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[2].

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.

[2] check this statement for consistency

2.1 Chord

Chord [8] 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.

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 with a greater ID until it wraps around. Each member of the network and the data stored is hashed to a unique m-bit key or ID, corresponding to one of the 2^m locations on a ring.

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. However, querying only neighbors 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 [8], 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 [8].

2.2 Kademlia

Kademlia [3] is perhaps the most well known and definately the most widely DHT, as a modified version of Kademlia (Mainline DHT) is useds as the 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.

Each k-bucket is maintained by a least recently seen eviction algorithm that skips live nodes. Whenever the node receives a message, it adds the sender's info to the tail of the corresponding k-bucket. If that info already exists, the info is moved to the tail.

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

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

Lookup

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

A `lookup(key)` operation begins with the seeking node sending lookups in parallel to the α nodes from the appropriate k-bucket. Each of theses α nodes will asynchronously return the k closest nodes it knows closest to `key`. As lookups return their results, the node continue to send lookups until no new nodes[3] are found.

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

4

Joining

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

Fault-Tolerance

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

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

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

2.3 CAN

Unlike the previous DHTs presented in this chapter, the Content Addressable Network (CAN) [13] 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.

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

[4] I would argue that this lookup operation is not recursive as claimed by the paper, but iterative, since the initiator sends all the messages.

divided; however, maintenance algorithms prevent the regions from becoming too fragmented.

Lookup

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

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

Joining

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

Once the new zones have been defined, n and m create its routing table from m and its former neighbors. These nodes are then informed of the changes that just occurred and update their tables. As a result, the join operation affects only $O(d)$ nodes. More details on this splitting process can be found in CAN's original paper [13].

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

[5] Around the same time CAN was being developed, Kleinberg was doing research into small world networks [14]. He proved similar properties for lattice networks with a single shortcut. What makes this network remarkable is lack of shortcuts.

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.

2.4 Pastry

Pastry [12] and Tapestry [11] are extremely similar use a prefix-based routing mechnism introduced by Plaxton et al. [15]. 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 [16], 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.
- 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.

[6] This message is sent to all of f's neighbors; I assume that nodes must keep track of their neighbors' neighbors.

[7] 0 is the 0th level. It's easier that way.

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.

Lookup

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

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

Algorithm 1 Pastry lookup algorithm

```
Let $L$ be the routing
function LOOKUP($key$)
    if $key$ is in the range of the leaf set then
        destination is closest ID in the leaf set or self
    else
        $next \leftarrow$ entry from routing table that matches $\geq 1$ more digit
        if $next \neq null$ 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.

2.5 Symphony and Small World Routing

Symphony [17] is a $1d$ ring-based DHT similar to Chord [8], but is constructed using the properties of small world networks [14]. 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 [18]. 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 [19] 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.

[8] Randomly chosen from a specified distribution.

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

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

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

Joining and Fault Tolerance

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

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

2.6 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 [7] is an example of such a system, as is Amazon's Dynamo [10]. 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.

No idea how membership table is updated

Fault Tolerance

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

2.7 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 [8] [3] [13] [12], but other metrics such as geographic location can be used as well [**?**].

These DHTs have carefully chosen metric spaces such that these regions are very simple to calculate. For example, Chord [8] and similar ring-based DHTs [**?**] utilize a unidirectional, one-dimensional ring as their metric space, such that the region for which a node is responsible is the region between itself and its predecessor.

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

Messages get routed along links to neighboring nodes. This would take $O(n)$ hops in one dimension. In multiple dimensions, our routing algorithm (Algorithm 2) is extremely similar to the one used in Ratnasamy et al.'s Content Addressable Network (CAN) [13], which would be $O(n^{\frac{1}{d}})$ hops.

Algorithm 2 Lookup in a Voronoi-based DHT

```
Given node $n$
Given $m$ is a message addressed for $loc$
$potential\_dests \leftarrow n \cup n.short\_peers \cup n.long\_peers$
$c \leftarrow$ node in $potential\_dests$ with shortest distance to $loc$
if $c == n$ then
    return $n$
else
    return $c.lookup(loc)$
end if
```

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

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

The Distributed Greedy Voronoi Heuristic (DGVH) is a fast method for nodes to define their individual Voronoi region (Algorithm 3). This is done by selecting the nearby nodes that would correspond to the points connected

Figure 2.1: 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.

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.

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
```

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

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

Nodes do this by calculating the distance of the given point to itself and other nodes it knows 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 $\Theta(\frac{\log N}{\log \log N})$ [**?**] (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 $\Omega(1)$.

2.8 Summary

DHT	Routing Table Size	Lookup Time	Join/Leave	Comments
Chord [8]	$O(\log n)$, 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 [3]	$O(\log n)$, maximum $m \cdot k$	$\lceil \log n \rceil) + c$	$O(\log(n))$	This is without considering optimization
CAN [13]	$\Omega(2d)$	$O(n^{\frac{1}{d}})$, average $\frac{d}{4} \cdot n^{\frac{1}{d}}$	Affects $O(d)$ nodes	d is the number of dimensions
Plaxton-based DHTs, Pastry [12], Tapestry [11]	$O(\log_\beta n)$			NodeIDs are base β numbers
Symphony [17]	$2k + 2$	average $O(\frac{1}{k}\log^2 n)$	$O(\log^2 n)$ messages, constant < 1	$k \geq 1$, fingers are chosen at random
ZHT [7]	$O(n)$	$O(1)$	$O(n)$	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

Proposal

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 I want to examine.

1. How can DHTs effectively be used for distributed computations? In what contexts is this feasible: only in the data center, or can a large-scale worldwide P2P network be used to do distributed computation? Is it better to use DHTs as an organizing mechanism, or as the actual platform for computation.

2. How can nodes autonomously load balance?

3.1 DHT Distributed Computing

Distributed computing is a current trend and is the future trend. We see this in the development of cloud computing [**?**], volunteer computing frameworks like BOINC [26] and Folding@Home [**?**],and MapReduce [30]. 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 [**?**] [32], are explicitly designed to be used in large datacenters [31] and the majority of research has been focused there. However, as we've previously noted, 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 [32]. So long as all computing resources located in one geographical area or rely on a particular node, a power outage or catastrophic event could interrupt computations or otherwise disrupt the platform [**?**].

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

There is no reason that these assumptions need to be the case for MapReduce, or for many distributed computing frameworks in general.

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

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 network is heterogeneous, rather than homogeneous.

They have been used for over a decade for P2P file-sharing applications for these reasons. My goal is to further develop ChordReduce and create a DHT based platform for solving embarrassingly parallel problems using DHTs.

- Build a highly configurable and easy to use DHT framework based off the DHT abstractions we have discovered.
- Use this framework to implement a few of the more popular DHTs.
- Create a processor scoring mechanism for creating virtual nodes.
- Do computing tasks over these implemented DHTs.
    - The emphasis is our framework is optimized for robustness over everything.
    - Do we want to build a IaaS or PaaS?

3.1.1 ChordReduce

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

File System

Our central design philosophy was to leverage as many features of the underlying DHT as possible. For example, we don't 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 [**?**]. 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 it's contents. The block is then stored at the node responsible for that key. The node which would normally be responsible for the whole file instead stores a *keyfile*. The keyfile is an ordered list of the keys corresponding to the files' block and is created as the blocks are assigned their respective keys. When the user wants to retrieve a file, they first obtain the keyfile and then request each block specified in the keyfile.

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 divide the work into *data atoms*, the smallest units of work. This might represent block of text, the result of a summation for a particular intermediate value, or a subset of items to be sorted. The specifics of how to divide the work are defined by the user in a *stage* function. The data atoms also contain 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 it's destination node, that node retrieves the necessary data and applies the Map function. The results are stored in a new data atom, which are then sent back to the stager's hash address (or some other user defined address). This will take $\log_2 n$ hops traveling over Chord's fingers. At each hop, the node waits a predetermined minimal amount of time to accumulate additional results (In our experiments, this was 100 milliseconds). Nodes that receive at least two results merge them using the Reduce function. The results are continually merged until only one remains at the hash address of the stager.

Some MapReduce jobs don't rely on a file stored on the network, such as a Monte-Carlo approximation 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 his results from the node at the stager's address. This may not be the stager himself, as the stager may no longer be in the network. The stager does not need to collect the results himself, since the work is sent to the stager's hash address, rather than the stager itself. Thus, the stager could quit the network after staging, and both the user and the network would be unaffected by the change.

Similar precautions are taken for nodes working on Map and Reduce tasks. Those tasks are backed up by a node's successor, who will run the task if the node leaves before finishing its work (e.g. the successor loses his predecessor). The task is given a timeout by the node. If the backup node detects that the responsible node has failed, he starts the work and backs up again to *his* successor. Otherwise, the data is tossed 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 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.

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.

Let each node has failure rate $r < 1$ and that the each node backs up their data with s successive nodes downstream. If one of these nodes fail, the next successive node takes its place and the next upstream node becomes another backup. This ensures there will always be s backups. The integrity of the ring would only be jeopardized if $s + 1$ successive nodes failed 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's responsible for and begin performing Map tasks immediately.

3.1.2 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.

This is more difficult in a heterogeneous system. 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 I need to answer is "how?" I need to create some unit of measurement for a distributed computing system and research if any other researchers have asked this problem (I haven't encountered any in my reading). 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, which is the other primary focus of my proposal.

3.2 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 [20] [21] 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 being buggy and was only able to handle smallish networks, we were fairly certain of it's ability to handle churn.

Marozzo et al. [20] 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.[1] This could effectively be thought of as each node flipping a

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

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 rubustness 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 robustness of the system. Since we were testing ChordReduce on Amazon's EC2 and paying per instance per hour, we didn't use any more nodes than necessary. 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. This completely mystified us. Churn is a disruptive force; how can it be aiding the network?

3.2.1 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 distributio. 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 a bunch 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:

- An 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 larger regions are larger and new joining nodes have a greater "chance" of hashing to that location). 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 are doesn't acquire any new jobs. This has no impact on the network (Case I).

    - The node rejoins in a region that has a 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 deal with) (Case III).
    - A possible counterbalance in load balancing by acquiring new jobs off a busy node (Case "It probably evens out").

Now here's the trick. 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 that Case I and II occur.

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 horrible 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.

The questions and goals here are straight forward:

- Further establish the phenomena exists.

- We stumbled across this phenomena with a brute force method and still got promising results. Can we create a more accurate and mean

- Can this phenomena be stochastically modeled or otherwise predicted via theoretical analysis?

- In what contexts can this be used for DHTs? Distributed computing? Replication for file sharing?

My proposed scheme works like this: a node that determines that it can carry more of the network's weight. How it does this is a problem that needs to be solved. Regardless, once it has made this determination, the node choose an area in the keyspace for it to inject a replica into.

This ties into the security research on DHTs I have done.

3.2.2 Sybil Attacks and Injection

We discovered injecting replicas is easy and simple in P2P networks, we use a Sybil attack. This was the focus of my Data Security project I hypothesize we can Sybil attacks for improving load balancing on demand.

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 [22], 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 [23].

One way to accomplish this attack is to perform a *Sybil attack* [22]. 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 [24]. Extensive research has been done assessing the damage an attacker can do after establishing themselves [23]. 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.

I focused on looking at the computational and memory costs of creating as many replicas as possible. The computation costs turn out to be fairly trivial and can be precomputed based on how IDs are assigned, a process I call *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. I calculated that it would take only 1221 IP addresses to compromise 50% of the links in a 20,000,000 node network.

An altruistic member of the network could only inject replicas where they are needed.

So given that we want nodes to insert nodes Why not let nodes choose keys manually or at random? First, this is bad practice which makes a Sybil attacks trivially easy to perform. Second, one of the primary benefits of using consistent hashing is that it allows a selection of conitguious keys to map to a uniform distribution. Most importantly for us, this distribution can be precomputed and reproduced as needed.

Perhaps we need an entirely new node type, a sentinel that exists merely for checking traffic conditions and load. This also breaks how we think of normal nodes.

One assertion is why not use some kind of mechanism for ensuring even distribution of nodes? The issue there is that our solutions for doing this are centralized distribution or a spring-model mechanism (or the binning mechanism of Dynamo, but the paper is a bit roundabout on how to do that).

Chapter 4

Proposed Work

4.1 Distributed DHT Computing

4.1.1 Abstract Framework

This portion has been done jointly with Brendan.

4.1.2

4.2 Autonomous load balancing

Bibliography

[1] B. Cohen, "Incentives build robustness in bittorrent," in *Workshop on Economics of Peer-to-Peer systems*, vol. 6, pp. 68–72, 2003.

[2] A. Loewenstern and A. Norberg, "BEP 5: DHT Protocol." `http://www.bittorrent.org/beps/bep_0005.html`, March 2013.

[3] P. Maymounkov and D. Mazieres, "Kademlia: A peer-to-peer information system based on the xor metric," in *Peer-to-Peer Systems*, pp. 53–65, Springer, 2002.

[4] L. 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*, pp. 1–10, Sept 2013.

[5] G. Mateescu, W. Gentzsch, and C. J. Ribbens, "Hybrid computingwhere {HPC} meets grid and cloud computing," *Future Generation Computer Systems*, vol. 27, no. 5, pp. 440 – 453, 2011.

[6] M. Li, L. Zhou, Z. Yang, A. Li, F. Xia, D. G. Andersen, and A. Smola, "Parameter server for distributed machine learning,"

[7] T. Li, X. Zhou, K. Brandstatter, D. Zhao, K. Wang, A. Rajendran, Z. Zhang, and I. 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*, pp. 775–787, IEEE, 2013.

[8] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications," *SIGCOMM Comput. Commun. Rev.*, vol. 31, pp. 149–160, August 2001.

[9] P. B. Godfrey and I. Stoica, "Heterogeneity and load balance in distributed hash tables," in *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, vol. 1, pp. 596–606, IEEE, 2005.

[10] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, “Dynamo: amazon’s highly available key-value store,” in *ACM SIGOPS Operating Systems Review*, vol. 41, pp. 205–220, ACM, 2007.

[11] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz, “Tapestry: A resilient global-scale overlay for service deployment,” *Selected Areas in Communications, IEEE Journal on*, vol. 22, no. 1, pp. 41–53, 2004.

[12] A. Rowstron and P. Druschel, “Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems,” in *Middleware 2001*, pp. 329–350, Springer, 2001.

[13] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, “A scalable content-addressable network,” 2001.

[14] J. Kleinberg, “The small-world phenomenon: An algorithmic perspective,” in *Proceedings of the thirty-second annual ACM symposium on Theory of computing*, pp. 163–170, ACM, 2000.

[15] C. G. Plaxton, R. Rajaraman, and A. W. Richa, “Accessing nearby copies of replicated objects in a distributed environment,” *Theory of Computing Systems*, vol. 32, no. 3, pp. 241–280, 1999.

[16] T. Condie, V. Kacholia, S. Sank, J. M. Hellerstein, and P. Maniatis, “Induced churn as shelter from routing-table poisoning.,” in *NDSS*, 2006.

[17] G. S. Manku, M. Bawa, P. Raghavan, *et al.*, “Symphony: Distributed Hashing in a Small World.,” in *USENIX Symposium on Internet Technologies and Systems*, p. 10, 2003.

[18] S. Milgram, “The small world problem,” *Psychology today*, vol. 2, no. 1, pp. 60–67, 1967.

[19] J. M. Kleinberg, “Navigation in a small world,” *Nature*, vol. 406, no. 6798, pp. 845–845, 2000.

[20] F. Marozzo, D. Talia, and P. Trunfio, “P2P-MapReduce: Parallel Data Processing in Dynamic Cloud Environments,” *Journal of Computer and System Sciences*, vol. 78, no. 5, pp. 1382–1402, 2012.

[21] K. Lee, T. W. Choi, A. Ganguly, D. Wolinsky, P. 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*, pp. 1602–1609, 2011.

[22] J. R. Douceur, “The sybil attack,” in *Peer-to-peer Systems*, pp. 251–260, Springer, 2002.

[23] M. Srivatsa and L. Liu, "Vulnerabilities and security threats in structured overlay networks: A quantitative analysis," in *Computer Security Applications Conference, 2004. 20th Annual*, pp. 252–261, IEEE, 2004.

[24] G. Urdaneta, G. Pierre, and M. V. Steen, "A survey of dht security techniques," *ACM Computing Surveys (CSUR)*, vol. 43, no. 2, p. 8, 2011.

[25] P. O. Boykin, J. S. A. Bridgewater, J. S. Kong, K. M. Lozev, B. A. Rezaei, and V. P. Roychowdhury, "A symphony conducted by brunet," *CoRR*, vol. abs/0709.4048, 2007.

[26] D. P. Anderson, "Boinc: A system for public-resource computing and storage," in *Grid Computing, 2004. Proceedings. Fifth IEEE/ACM International Workshop on*, pp. 4–10, IEEE, 2004.

[27] D. Thain, T. Tannenbaum, and M. Livny, "Distributed computing in practice: The condor experience," *Concurrency and Computation: Practice and Experience*, vol. 17, no. 2-4, pp. 323–356, 2005.

[28] K. Lee, D. Wolinsky, and R. J. Figueiredo, "Pond: dynamic creation of htc pool on demand using a decentralized resource discovery system," in *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*, pp. 161–172, ACM, 2012.

[29] D. Malkhi, M. Naor, and D. Ratajczak, "Viceroy: A scalable and dynamic emulation of the butterfly," 2001.

[30] J. Dean and S. Ghemawat, "Mapreduce: Simplified Data Processing on Large Clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[31] "Virtual hadoop." http://wiki.apache.org/hadoop/Virtual

[32] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pp. 1–10, IEEE, 2010.

[33] P. Triantafillou and T. Pitoura, "Towards a unifying framework for complex query processing over structured peer-to-peer data networks," in *Databases, Information Systems, and Peer-to-Peer Computing*, pp. 169–183, Springer, 2004.

[34] M. Harren, J. M. Hellerstein, R. Huebsch, B. T. Loo, S. Shenker, and I. Stoica, "Complex queries in dht-based peer-to-peer networks," in *Peer-to-Peer Systems*, pp. 242–250, Springer, 2002.