Proposal

Andrew Rosen

October 16, 2014

Contents

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 [2].

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 [4] 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 [5].

I decribe each of the aformentioned 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 [6] shows what can happen if we remove this assumption by placing the network that is almost completely static.

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* [7] [8]. When a node joins the network, it joins not at one position, but multiple virtual positions in the network [8]. 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 [8]. Each node was assigned a certain number of tokens and the node would create a virtual node for each token. The realization DeCandia et

alḣad 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 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 applica-

[1] Need help here

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

tions, 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.
- 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 [8]. 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 immediately. This is almost instantaneous if the network is using backups. Other nodes can be notified of the missing node passively through maintenance.

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 peerlist and lookup time. The vast majority of DHTs choose a logarithmic sized routing table to provide a simple routing scheme that keeps the number of hops logarithmically bound to the network size. Chapter 2 discusses these tradeoffs in greater detail and how they affect the each DHT.

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

1.1.4 The consequences of the Properties

So what are the consequences of these properties?

- DHTs can use constistant hashing supplemented by virtual nodes to efficiently load-balance.
- DHTs are highly resilient to damage and can handle abnormally high rates of disruption. This is extremely desirable in any kind of distributed application
- X is a desirable property in a network for distributed computing by
-

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 all of these qualities to an application.

1.2.1 The Takeaway

- DHTs are extremely good if your problem is embarrassingly parallel
- DHTs are agnostic in terms of what hardware it's running on.
-

Chapter 2

Background

2.1 Unified Terminology

Many papers use different terms to describe congruent elements of DHTs, as some terms may make sense only in one context. I shall endeavor to add to confusion by using the following unified terminology:

Peerlist - The set of all peers that a node knows about. This is sometimes referred to as the *routing table*, but certain DHTs [9] [10] overload the terminology.

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 [11], this is the node's *predecessor* and *successor*.

Fingers - The subset of the peerlist that the node is not adjacent to. These are sometimes referred to as long-hops or shortcuts. The other major distinction is that fingers aren't no

Root Node - The node responsible for a particular key.

2.2 Summary

DHT	Routing Table Size	Lookup Time	Join/Leave	Comments
Chord [11], Kademlia [3]	$O(\log n)$	$O(\log n)$		This is where most DHTs fall
CAN [12]	$\Omega(2d)$	$O(n^{\frac{1}{d}})$, average $\frac{d}{4} \cdot n^{\frac{1}{d}}$	Affects $O(d)$ nodes	d is the number of dimensions
Plaxton-based DHTs, Pastry [10], Tapestry [9]				
ZHT [6]	$O(n)$	$O(1)$		Assumes an extremely low churn

Table 2.1: The different ratios and their associated DHTs

Chapter 3

Justification and Why I Think It's Cool

3.1 Why DHTs for distributed computing?

[13] - Between congestion, cost of join/leaves, and lookup time there are trade-offs. Optimizing for two can be done but has bad cost. For example, a balanced binary tree has congestion at root.

3.1.1 DHTs well understood

3.1.2 DHTs are Highly used for their intended purposed

Bittorrent, WoW

3.2 DHTs are being effectively leveraged for other things besides file sharing already

PaaS

Load Balancing in the cloud

Resource Management in the cloud

Computing is a natural extension

3.2.1 DHTs as a volunteer Platform

Rather than rely on a centralized administrative source,

Decentralized resource discovery. The system is organized using a P2P system built on Brunet [14].

PonD [**?**]

Experiment Description

Implement and compare to Boinc.

Chapter 4

Possible Experiments and Applications

4.1 MapReduce

Google's MapReduce [15] paradigm has rapidly become an integral part in the world of data processing and is capable of efficiently executing numerous Big Data programming and data-reduction tasks. By using MapReduce, a user can take a large problem, split it into small, equivalent tasks and send those tasks to other processors for computation. The results are sent back to the user and combined into one answer. MapReduce has proven to be an extremely powerful and versatile tool, providing the framework for using distributed computing to solve a wide variety of problems, such as distributed sorting and creating an inverted index [15].

At it's core, MapReduce [15] is a system for process key/value pairs, a that statement that equally describes DHTs. However, MapReduce operates over a different set of assumptions [16] than DHTs. MapReduce platforms are highly centralized and tend to have single points of failure[17] as a result. A centralized

design assumes that the network is relatively unchanging and does not usually have mechanisms to handle node failure during execution or, conversely, cannot speed up the execution of a job by adding additional workers on the fly. Finally deploying these systems and developing programs for them has an extremely steep learning curve.

If we make MapReduce operate under the same assumptions as a DHT, we have effectively further abstracted the MapReduce paradigm and created a system that can operate both in a traditional large datacenter or as part of a P2P network. The system would be highly resistant to failures at any point, scalable, and automatically load-balance. The administrator can add any number of heterogeneous nodes to the system to get it operate.

4.1.1 Current MapReduce DHT/P2P combos

There have been a few implementations combining MapReduce with a P2P framework, in varying capacities. I will present two here, as well as my own implementation, ChordReduce.

P2P-MapReduce

Marozzo et al. [18] investigated the issue of fault tolerance in centralized MapReduce architectures such as Hadoop. They focused on creating a new P2P based MapReduce architecture built on JXTA called P2P-MapReduce. P2P-MapReduce is designed to be more robust at handling node and job failures during execution.

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

responsible for it.

Experimental results were gathered via simulation and compared P2P-MapReduce to a centralized framework. Their results showed that while P2P-MapReduce generated an order of magnitude more messages than a centralized approach, the difference rapidly began to shrink at higher rates of churn. When looking at actual amounts of data being passed around the network, the bandwidth required by the centralized approach greatly increased as a function of churn, while the distributed approach again remained relatively static in terms of increased bandwidth usage. They concluded that P2P-MapReduce would, in general, use more network resources than a centralized approach. However, this was an acceptable cost as the P2P-MapReduce would lose less time from node and job failures [18].

Parralel Processing Framework on a P2P System

Lee et al.'s work [19] draws attention to the fact that a P2P network can be much more than a way to distribute files and demonstrates how to accomplish different tasks using Map and Reduce functions over a P2P network. Rather than using Chord, Lee et al. used Symphony [20], another DHT protocol with a ring topology. To run a MapReduce job over the Symphony ring, a node is selected by the user to effectively act as the master. This ad-hoc master then performs a bounded broadcast over a subsection the ring. Each node repeats this broadcast over a subsection of that subsection, resulting in a tree with the first node at the top.

Map tasks are disseminated evenly throughout the tree and their results are reduced on the way back up to the ad-hoc master node. This allows the ring to disseminate Map and Reduce tasks without the need for a coordinator responsible for distributing these tasks and keeping track of them, unlike Hadoop. Their experimental results showed that the latency experienced by a centralized configuration is similar to the latency experienced in a completely distributed

framework.

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. ChordReduce was designed 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.

4.1.2 Experiment Description: Comparison of MapReduce paradigm on different DHTs

In order to test MapReduce over a DHT, I will do the following:

- Implement CAN [12], Pastry [10], Chord [11], Kademlia [3], VHash, and ZHT [6] /similar
  - This covers different geometries with different base parameters.
  - This also necessitates the creation of an extensible DHT framework.
  - The DHT should be extended with more powerful search functionality (see distributed database below), and built-in policies for virtual nodes.
- Compare results with each other and a traditional MapReduce platform, such as Hadoop.
- Certain DHTs may be better suited to different problem formulation

4.2 High End Computing

PonD?

4.2.1 Metadata Management

4.2.2 Robustness

4.2.3 Experiment Description:

4.3 Graph Processing on a DHT

Lookup Graphlab

4.3.1 Embedding

4.3.2 Experiment Description:

4.3.3 Distribute the work for solving a graph on a DHT

4.3.4 Comparison to well established or state of the art methods

4.4 Machine Learning Problems on A DHT

Bayesian Learning

4.4.1 Experiment Description:

Take MapReduce machine learning algorithm

4.5 Distributed Databases

Want to find all files that match the criteria?

Simple: Find all files with "author = John Smith". Idiot solution, assign "author = John Smith" a hash key, it's value is a file with all the files with the (that doesn't scale)

Complex: Processing database queries. Find all files with age ¡ 20 and niceness ¿12

4.6 Semiautomagic Load Balancing

4.7 Resources

4.7.1 Planetlab

4.7.2 Local Cluster

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] 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.

[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] 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.

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

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

[14] 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.

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

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

[17] 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.

[18] 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.

[19] 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.

[20] 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.