Autonomous Load Balancing in Distributed Hash Tables Using Churn and The Sybil Attack

Andrew Rosen Brendan Benshoof Robert W. Harrison Anu G. Bourgeois
Department of Computer Science
Georgia State University
Atlanta, Georgia
rosen@cs.gsu.edu bbenshoof@cs.gsu.edu rharrison@cs.gsu.edu anu@cs.gsu.edu

***Abstract*—Foobar**

***Index Terms*—Distributed Computing; Decentralized Systems; Peer-to-Peer Networks; Distributed Hash Tables; Self-Organizing Networks; Load-Balancing; Sybil Attack;**

I. Introduction

Distributed Hash Tables (DHTs) are a class of well researched decentralized key-value storage systems. DHTs are most frequently used to construct the overlays of P2P file-sharing systems, such as the incredibly popular BitTorrent [1]. DHTs have also been used in many unorthodox applications, such as machine learning [2] and, most relevant for our discussion, distributed computing [3].

Our previous research [3] examined using a DHT as the organizing mechanism for a distributed platform. This enabled us to create an exceptionally fault tolerant distributed computing platform that was easy to setup and could be run in a completely decentralized P2P environment.

One of key components of a Distributed Hash Table is a cryptographic hash function, most commonly SHA1 [4]. Distributed Hash Tables rely on this cryptographic hash function to generate identifiers for data stored on the network. The cryptographic hash of the filename or other identifier for the data is used as the location of the file or data in the network. It can also be used to generate the identifiers for nodes in the network.

Ideally, inputing random numbers into a cryptographic hash function should produce a uniformly distributed output. However, this is impossible in practice [5] [6].

In practice, that means given any DHT with files and nodes, there will be an inherent imbalance in the network. Some nodes will end up with a lion's share of the keys, while other will have few responsibilities (Table I).

This makes it especially disheartening to try and ensure as even a load as possible. We cannot rely on a centralized strategy to fix this imbalance, since that would violate the principles and objects behind creating a fully decentralized and distributed system.

Therefore, if we want to create strategies to act against the inequity of the load distribution, we need a strategy that individual nodes can act upon autonomously. These strategies need to make decisions that a node can make at a local level, using only information about their own load and the topology they can see.

A. Motivation

The primary motivation for us is creating a new viable type of platform for distributed computing. Most distributed computing paradigms, such as Hadoop [7], assume that the computations occur in a centralized environments. One enormous benefit is a centralized system has much greater control in ensuring load-balancing.

However, in an increasingly global world where computation is king and the Internet is increasingly an integral part of everyday life, single points of failure failures quickly become more and more risky. Users expect their apps to work regardless of any power outage affecting an entire region. Customers expect their services to still be provided regardless of any. The next big quake affecting the the San Andreas fault line is a matter of when, not if. Thus, centralized systems with single points of failure become a riskier option and decentralized, distributed systems the safer choice.

Previous research has demostrated the feasibility of a DHT-based distributed computational system. [8] demonstrated ABC awesomely, but however had XYZ issue Similarly [9] showed ABC awesomely, had XYZ as issue Our own research [3] was awesome, but did not load balance properly.

Our previous work in ChordReduce [3] focused on creating a decentralized distributed computing framework based off of the Chord Distributed Hash Table (DHT) and MapReduce. ChordReduce can be thought of a more generalized implementation of the concepts of MapReduce. One of the advantages of ChordReduce can be used in either a traditional datacenter or P2P environment.[1] Chord (and all DHTs) have the qualities we desire for distributed computing: scalability, fault tolerance, and load-balancing.

Fault tolerance is of particular importance to DHTs, since their primary use case is P2P file-sharing, such as BitTorrent [1]. These systems experience high levels of churn– disruptions to the network topology as a result of nodes entering and leaving the network. ChordReduce had to have the same level of robustness against churn that Chord did, if not better.

During our experiments with ChordReduce, we found that high levels of churn actually made our computations run *faster*. We hypothesized that the churn was effectively load-balancing the network.

B. Objectives

This paper serves to prove our hypothesis that churn can load balance a Distributed Hash Table. We also set out to show that we can use this in a highly controlled manner to greater effect.

We present three additional strategies that nodes can use to redistribute the workload in the network. Each of these three strategies relies on nodes creating virtual nodes, and so that they are represented multiple times in the network. The idea behind this is that a node with low work can create virtual nodes to seek out and acquire work from other nodes.

This tactic is the same one used in the Sybil [10] attack, but limited in scope and performed for altruistic and beneficent reasons, rather than malicious ones. Consequently, we'll call our virtual nodes in this scenario Sybils for clarity and brevity. None of the strategies require a centralized organizer, merely a way for a node to check the amount of files or tasks it and its Sybils are responsible for.

We also want to show how distributed computing can be performed in a heterogeneous environment. We want to see if our strategies result in similar speedups in both homogeneous and heterogeneous environments.

[1]The other one being that new nodes could join during runtime and receive work from nodes doing computations.

TABLE I: The median distribution of tasks (or files) among nodes. We can see the standard deviation is fairly close to the expected mean workload ($\frac{tasks}{nodes}$). Each row is the average of 100 trials.

Nodes	Tasks	Median Workload	σ
1000	100000	69.410	137.27
1000	500000	346.570	499.169
1000	1000000	692.300	996.982
5000	100000	13.810	20.477
5000	500000	69.280	100.344
5000	1000000	138.360	200.564
10000	100000	7.000	10.492
10000	500000	34.550	50.366
10000	1000000	69.180	100.319

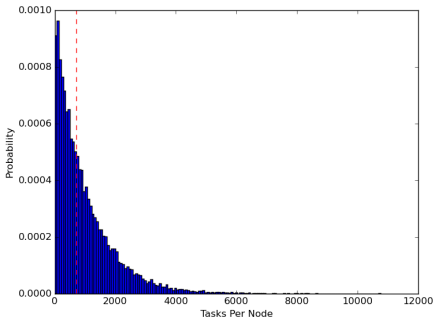

Fig. 1: The probability distribution of workload in a DHT with 1000 nodes and 1,000,000 tasks or files. The vertical dashed line designates the median. Keys were generated by feeding random numbers into the SHA1 hash function [4], a favorite for many distributed hash tables.

II. How Work Is Distributed in DHTs: A Visual Guide

As we have previously mentioned, many DHTs use a cryptographic hash function to choose keys for nodes and data. However, no cryptographic hash function can uniformly distribute it's outputs across its range.

III. Simulation

We will be simulating our strategies on a Chord distributed hash table [11], although it is fairly straightforward to implement our strategies for other, more complex DHTs. Nodes in a Chord ring are given an ID, drawn from a cryptographic hash function, typically SHA1 [4]. Any data in the network is given a key in a similar manner. Nodes are responsible for all the keys between their predecessor's ID and their own.

We assume that the network starts our experiments stable and the data necessary already present on the nodes and backed-up. The following analysis and simulation relies on an important assumption about DHT behavior often assumed but not necessarily implemented.

We assume that nodes are active and aggressive in creating and monitoring the backups and the data they are responsible for. We have demonstrated the effectiveness and viability of implementing an active backup strategy in other work [3] [12]. As previously mentioned, we also assume nodes have the ability to examine the amount of work they have and know how many tasks or pieces of data exist for a particular job.

Another assumption is that nodes do not have control in choosing their IDs from the range of hash values. This means that nodes cannot automatically change their spacing to ensure they are all evenly spread out across the network. Likewise, nodes cannot create necessarily create Sybils exactly where they would like, but would have to search for an appropriate ID in between two other nodes. We discussed this process in previous work and showed that it is extremely quick for a node to do so [13].

A. The Parameters

Our simulations relied on a great number of parameters and variables. We present each of them below.

1) Constants and Terminology: First, we'll define informally define the vocabulary we use to discuss our simulations.

Tick
: In a simulation, normal measurements of time such as a second are completely arbitrary. We will be using an abstract *tick* to measure time. We consider the tick the amount of time it takes a node to complete one task (or more depending on our variables, see below) and perform the appropriate maintenance to keep the network consistent and healthy.[2]

Maintenance
: We assume nodes use the active, aggressive strategy from ChordReduce and UrDHT [3] [12]. Every maintenance cycle, each node checks and updates its list of neighbors (successors and predecessors in Chord) and responds appropriately. We assume that a tick is enough time to accomplish at least one maintenance cycle.

Task
: We measure the size of a distributed computing job in tasks. Each task has a key that corresponds to the key of a file or chunk of a file stored on the network. We assume that it takes a tick for a node to consume at least one task.

IDs and Keys
: We will be using SHA-1 [4], a 160-bit hash function, to generate node IDs and keys for tasks. We assume each task's key corresponds to some piece of data with the same key, a scheme we used in our previous work on ChordReduce [3].

2) Experimental Variables: We tested a number of strategies and variables that could affect each strategy. While we believed the overall strategy would result in largest differences in runtime, we wanted to see what effects, if any, each of the variables would have on a particular strategy.

Strategy
: We use several different strategies, each discussed in Section IV: churn, random injection, neighbor injection, and invitation. Each of these strategies differs in how nodes attempt to autonomous balance the workload of the network. None of the strategies require centralized organization.

Homogeneity
: This variable controls whether the network is homogeneous or not. In a homogeneous network, each node has the same strength, which dictates how much work is consumed per a tick and how many Sybils it can create. In a heterogeneous network, each node has a strength chosen uniformly at random from 1 to `maxSybils`.

Work Measurement
: This variable dictates how much work is consumed in a tick. Each node can either consume a single task per a tick or their strength's worth of tasks per a tick.

Network Size
: How many nodes start in the network. We assume that this can grow during the experiment, either via churn or by creating Sybils. We discuss how this works in Sections IV-A and IV-B.

Number of Tasks
: We measure the size of of a job in tasks. This number is typically a few orders of magnitude greater than the network size.

Churn Rate
: The churn rate is a float between 0 and 1. This represents both the chance each individual node has of leaving (or joining) the network each tick. It also corresponded to the fraction of nodes we expect to leave the network each tick. Churn can be self induced or a result of actual turbulence in the network. Like most analyses of churn [8], we assume churn is constant throughout the experiment and that the joining and leaving rate are equal.

[2] If we need to be more concrete, define a tick as a New York Second, which is "the period of time between the traffic lights turning green and the cab behind you honking."
– Terry Pratchett

Max Sybils `maxSybils` is the maximum number of Sybils a node can have at once.

Sybil Threshold `sybilThreshold` dictates the number of tasks a node must have before it can create a Sybil.

Successors The number of successors each node keeps track of. Nodes also keep track of the same number of predecessors.

We also considered using a variable noted as the `AdaptationRate`, which was the interval at which nodes decided whether or not to create a Sybil. Preliminary experiments showed `AdaptationRate` to have a minimal effect on the runtime, so it was removed.

3) Outputs: The most important output was the number of ticks it took for the experiment to complete. We compared this runtime to the what we call the "ideal runtime," which is our expected runtime if the every node in the network was given an equal number of tasks and performed them without any churn or creating any Sybils. For example, consider a network with 1,000 nodes and 100,000 tasks, where each node consumes a single task each tick. This network has an ideal runtime of 100 ticks ($\frac{100000}{1000} = 100$).[3]

We used these to calculate a "runtime factor," the ratio of the experimental runtime compared to the ideal runtime. For example, in the network from our previous example took 852 ticks to finish, its factor is 8.52. We prefer to use this runtime factor for comparisons since it allows us to compare networks of different compositions, task assignments, and other variables.

We also collected data on the average work per tick and statistical information about how the tasks are distributed throughout the network. We additionally performed detailed observations of how the workload is distributed and redistributed throughout the network during the first 50 ticks.

IV. Strategies

For our analysis, we examined four different strategies for nodes to perform autonomous load balancing. We first show the effects of churn on the speed of a distributed computation. We then look a three different strategies for in which nodes take a more tactical approach for creating churn and affecting the runtime.

Specifically, nodes perform a limited and controlled Sybil attack [10] on their own network in an effort to acquire work with their virtual nodes. Our strategies dictate when and where these Sybil nodes are created.

We discuss the effectiveness of each of the strategies and present the results of our simulations in Section V.

A. Induced Churn

Our first strategy, *Induced Churn*, relies solely on churn to perform load balancing. This churn can either be a product of normal network activity or self-induced.[4] By self-induced churn, we mean that each node generates a random floating point number between 0 and 1. If the number is $\leq churnRate$, the node shuts down and leaves the network and gets added to the pool of nodes waiting to join.

Similarly, we have a pool of waiting nodes that begins the same initial size as the network. When they generate an appropriate random number, they join the network and leave the waiting pool. We assume that nodes enter and leave the network at the same rate. Because the initial size of the network and the pool of waiting nodes is the same and nodes move between one another at the same random rate, the size of either does not fluctuate wildly.

As we have previously discussed, nodes in our network model actively back up their data and tasks to the a number of successors in case of failure. In addition, when a node joins, it acquires all the work it is responsible for. While this model is rarely implemented for DHTs, it is discussed [14] and often assumed to be the way DHTs operate. We have implemented it in ChordReduce[3] and UrDHT[12] and demonstrated that the network is capable from recovering from quite catastrophic failures and handling ludicrous amounts of churn.

The consequences of this are that a node suddenly dying is of minimal impact to the network. This is because a node's successor will quickly detect the loss of the node and know to be responsible for the node's work. Conversely, a node joining in this model can be a potential boon to the network by joining a portion of the network with a lot of tasks and immediately acquiring work.

This strategy acts as a baseline with which to compare the other strategies, as it is no more than a overcomplicated way of turning machines off and on again. All strategies below are attempts to do better than random chance. However, this strategy also serves to confirm the speedup phenomenon we observed in our previous work on a distributed, decentralized MapReduce framework [3].

[3]The ideal runtime also happens to be the average number of tasks per node. An interesting, but mathematically unsurprising, coincidence with a few consequences for our data collection.

[4]All distributed systems experience churn, even if only as hardware failures.

B. Random Injection

Our second strategy we dubbed *Random Injection*. In this strategy, once a node's workload was at or below the `sybilThreshold`, the node would attempt to acquire more work by creating a Sybil node at a random address.

A node compares its workload to the `sybilThreshold` and decides whether or not to make a Sybil. This check occurs every 5 ticks. A node cam also have multiple Sybils, up to `maxSybils` in a homogeneous network or the node's `strength` in a heterogeneous network.[5] If a node has at least one Sybil, but no work, it has its Sybils quit the network. We limit each node to creating a single Sybil during this decision to avoid overwhelming the network.

C. Neighbor Injection

Neighbor Injection also creates Sybils, but in this case, nodes act on a more restricted range in an attempt to limit the network traffic. Nodes with `sybilThreshold` or less tasks attempt to acquire more work by placing a Sybil among it's successors. Specifically, it looks for the biggest range among it's successors and creates a Sybil in that range.

This range strategy of finding the largest range and injecting assumes that the node with the largest range of responsibility will have been allocated the most work. This is a sensible assumption since the larger the range of a node's responsibility, the more *potential* tasks it can receive. We compare this estimation strategy to actually querying the neighbor and asking how many tasks they have. An estimation, if accurate, would be preferable to querying the nodes as an estimation can be done without any communication with the successor nodes.

To avoid constant spamming of a range, once a node creates a Sybil, but does not acquire work, it may be advisable to mark that range as invalid for Sybil nodes so nodes don't keep trying the same range repeatedly.

D. Invitation

The *Invitation* strategy is the reverse of the Sybil injection strategies. In this strategy, nodes with a higher than desired level of work announces it needs help to its predecessors. The predecessor with least amount of tasks less than or equal to the `sybilThreshold` creates a Sybil to acquire work from the node. It is possible for an invitation to get more work to be refused by anyone if no predecessor is at the `sybilThreshold` or each predecessor has too many Sybils. Nodes determine whether or not they are overburdened using the `sybilThreshold`.

[5]No benefit was shown by increasing `maxSybils` beyond 10.

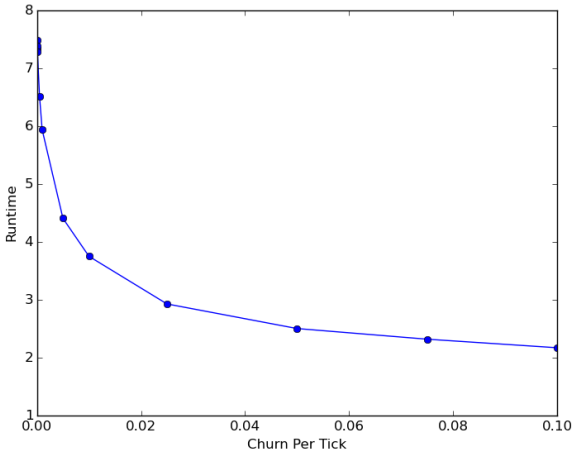

Fig. 2: This graph shows the effect churn has on runtime in a distributed computation. Runtime is measured as how many times slower the computation runs than an ideal computation, where each node receives an equal number of tasks. Neither the homogeneity of the network nor work measurement affected the runtime.

V. Results Of Experiments

We now briefly summarize the results of our simulation before discussing each more in depth below.

All the raw data can be found online [15].

A. Churn

Our results confirmed our hypothesis from Chord-Reduce [3]. Churn has a profound and significant impact on the network's computation, and this effect is more pronounced with higher rates of churn. Figure 2 plots the runtime of the distributed computation against the level of churn in the network. This DHT contained 1000 nodes and 100000 tasks.

We note the significantly diminishing returns that occur after a churn rate of 0.01. One facet not captured by our simulations, but is significant, is the rising maintenance costs after that point.

Also graph average work per tick (Figure 3)

B. Random Injection

The strategy of having under-utilized nodes randomly create Sybil nodes works phenomenally well, approaching very close to the ideal time.

C. Neighbor Injection

D. Invitation

In invitation, churn losses can be greatly detrimental.

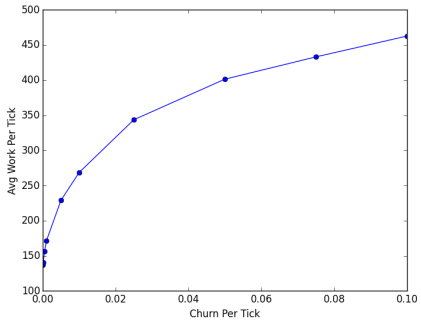

Fig. 3: We can see that this is a mirror image of Figure 2

VI. Future Work

As mentioned in Section III, we made the assumption that nodes cannot choose their own ID and must rely on the strategies described in Chapter **??** [13] for creating a Sybil with the appropriate ID. However, if this assumption was removed, this presents even more strategies for nodes to autonomously load-balance.

References

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

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

[3] A. Rosen, B. Benshoof, R. W. Harrison, and A. G. Bourgeois, "Mapreduce on a chord distributed hash table," in *2nd International IBM Cloud Academy Conference*.

[4] D. Eastlake and P. Jones, "Us secure hash algorithm 1 (sha1)," 2001.

[5] M. Buddingh, "The distribution of hash function outputs," http://michiel.buddingh.eu/distribution-of-hash-values.

[6] S. S. Thomsen and L. R. Knudsen, "Cryptographic hash functions," Ph.D. dissertation, Technical University of DenmarkDanmarks Tekniske Universitet, Department of Applied Mathematics and Computer ScienceInstitut for Matematik og Computer Science, 2005.

[7] "Hadoop," http://hadoop.apache.org/.

[8] 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.

[9] 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*, 2011, pp. 1602–1609.

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

[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. [Online]. Available: http://doi.acm.org/10.1145/964723.383071

[12] A. Rosen, B. Benshoof, R. W. Harrison, and A. G. Bourgeois, "Urdht," https://github.com/UrDHT/.

[13] ——, "The sybil attack on peer-to-peer networks from the attacker's perspective."

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

[15] A. Rosen, "Autonomous load-balancing raw data," https://github.com/abrosen/thesis/tree/master/data/done.