UrDHT: A Unified Model for Distributed Hash Tables

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

TABLE I: The median distribution of tasks among nodes. We can see the standard deviation is a particularly useless measurement in this scenario. Each row is the average of 100 trials.

Nodes	Tasks	Median Workload	Avg σ of workload.
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

***Abstract*—Foobar**

***Index Terms*—Peer-to-Peer Networks; Distributed Hash Tables; Self-Organizing Networks; Load-Balancing;**

I. Introduction

Distributed Hash Tables rely on cryptographic hash functions to generate identifiers for both nodes and data. Ideally, inputing random numbers into a cryptographic hash function should produce a uniformly distributed output. However, this is impossible in practice [1] [2].

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 [3], 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.

Our previous work in ChordReduce [4] 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 [5]. These systems experience high levels of

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

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 3 strategies nodes can use to redistribute the workload in the network that do no require a centralized organizer.

We also want to show how distributed computing can be performed in a heterogeneous environment.

II. Previous Work

ChordReduce [4] 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.

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

In this section, we display graphs to give a visual representation of how work is distributed in a Chord [6] network

IV. Simulation

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. Specifically, we will assume it takes T_{detect} time for a node to detect a change in their responsibility or to detect a new node to hand a backup to and that this check is performed regularly. We have demonstrated the effectiveness and viability of implementing an active backup strategy in other work [4] [7].

Another assumption is that nodes do not have control in choosing their IDs from the range of hash values.

Smaller chunking results in more files spread throughout the network and a greater chance of the data being evenly spread across the network

The chances of a critical failure happening within a time interval T is the chances of some chain or cluster of nodes responsible for a single record dying within T:

$$r^s$$

Where r is the failure rate over that time interval and s is the number nodes storing that record, either as a primary system, or a backup. Incidentally, this time interval $T = T_{detect} + T_{transfer}$

A. The Parameters

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

1) Constants and Terminology:

Tick In a simulation, normal measurements of time such as a second are arbitrary, so we be using the abstract *tick* to measure time. We consider the tick the amount of time it takes a node to complete one task per sybil and perform the appropriate maintanence.[2]

Maintenance We assume nodes use the active, aggressive strategy from ChordReduce and UrDHT [4] [7]. 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

Task A distributed computing job is defined 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 a task.

Hash Function We will be using SHA-1 [8], a 160-bit hash function. Keys for will be drawn randomly from this hash function.

2) Experimental Variables:

Strategy We use several different strategies (discussed in Section V) for balancing the load of tasks among the nodes. None of the strategies require centralized data.

Homogeneity This variable controls whether the network is homogeneous or not. In a homogeneous network, each node has the same strength, which

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

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. Nodes each 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.

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 Measured in ticks, this can be self induced or a result of actual turbulence in the network. Like most analyses of churn [9], we assume churn is constant throughout the experiment and that the joining and leaving rate are equal.

Max Sybils `maxSybils` is the maximum number of Sybils a node can create.

Sybil Threshold The `sybilThreshold` calculates the amount 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:

V. 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 this strategy in Section VI.

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.[3] 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.

Similarly, we have a pool of waiting nodes the same size as the network. When they generate an appropriate random number, they join the network. We assume that nodes enter and leave the network at the same rate.

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 [11] and often assumed to be the way DHTs operate. We have implemented it in ChordReduce[4] and UrDHT[7] 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.

B. Random Sybil 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 checks its threshold and decides whether or not to make a Sybil 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.[4] If a node has at least one Sybil, but no work, it has its Sybils quit the network. We have the decision to make a Sybil occur every 5 ticks and limit each node

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

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

to creating a single Sybil a check to avoid overwhelming the network.

As we discuss in Section VI, this strategy is surprisingly effective and comes close to ideal runtimes.

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 finding and injection strategy assumes that the node with the largest range of responsibility has. The nodes estimate the number of tasks based on the range instead of querying the nodes as that 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.

We might want to look at changing this strategy to choosing a random id in the neighbor range.

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 the `sybilThreshold` creates a Sybil to acquire work from the node(it is possible for a call for help to go unanswered).

Nodes determine whether or not they are overburdened using the `sybilThreshold`.

In invitation, churn losses can be greatly detrimental.

VI. Results Of Experiments

References

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

[2] 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.

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

[4] 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*.

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

[6] 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

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

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

[9] 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.

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

[11] 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.