Using Chord as a Platform for Software as a Service for a Generalized, Distributed MapReduce.

Andrew Rosen Brendan Benshoof Matt Erwin Anu Bourgeois
Department of Computer Science, Georgia State University
34 Peachtree St NW
Atlanta, Georgia 30303
rosen@cs.gsu.edu

***Abstract*—DRAFT ABSTRACT: We developed a Peer-to-peer architecture for software as a service. We show viability by implementing MapReduce. MapReduce does not automatically distribute work among slave nodes without configuration. In Chord, nodes are evenly distributed and are responsible for files based on their position in the network. We exploit this property to automatically distribute MapReduce tasks evenly among nodes in the same manner.**

I. Introduction

Peer-to-peer (P2P) networks are of enormous interest to network research, as they provide an efficient way to distribute large files to millions of users without many of the limitations of the traditional client-server model. In the client-server model, the server addresses each client's request individually, creating a single point of failure due to traffic or the loss of the server. In peer-to-peer networks, each node takes on both the role of client and server. If someone wishes to introduce data into the a peer-to-peer network, peers can both request the file and distribute it to others. This takes much of the burden of file distribution off of what would have normally been an extremely busy server [16].

P2P networks have resulted in an unprecedented amount of data being purveyed through the Internet. While recent studies show that P2P accounts for 13.2% of downstream traffic, down from its peak of 40% of all Internet traffic, the actual volume of P2P traffic is at its highest yet [12]. Many companies have embraced legal uses of P2P technology, such as providing ways to stream media or provide large updates [13]. For example, Blizzard uses a modified BitTorrent protocol to distribute updates and patches to millions of users for games such as World of Warcraft and Starcraft II [4].

1) SaaS: In recent years, there has been a trend of crowd-sourcing large and complicated tasks among willing participants.

Folding@home [2].

seti at home that kind f thing

Our thing is totally way better because it we're making it generic, knatch. Why is generic better? becuase it makes it easier for developers.

We create the system and genericized version of map-reduce

2) Current models of distributing work:

II. Background

Not all peer-to-peer networks are equal; there are variety of protocols and methodologies that a networks could implement and this affects what kind of solutions are available to reduce the traffic on the network [8] [16].

The most basic type of network is a structured, centralized network. Peers in this network communicate with a central server to provide their files and to locate other peers that have the files they are searching for. This is structured in the sense that the layout of the overlay network is tightly controlled, in this case by the server(s). While this avoids the problems of routing, it has the same issues of scalability as a client-server layout and is not much of an improvement. An example of this network is the long defunct Napster [8] [16].

On the other side of the spectrum, there are unstructured, decentralized networks. These networks create overlay links between nodes in a random manner. This leads to a very unstructured overlay, but it is one that is very easily constructed. No single node is responsible for the whole of the network; files are located by sending out requests to neighboring peers, which in turn request from their neighbors and so on. Should a file become suddenly popular, this flood of this requests can easily bring some peers to their knees, unable to deal with the high level of traffic [9]. This makes these types of networks also a poor choice for implementation [16].

Modern P2P implementations are hard to classify easily, due to the variety of methods used to create a working network that avoids the weaknesses of the two above network types. Many networks today use a decentralized structured approach to distribute files, where the topology of the overlay is constructed and controlled by the protocol and the information about the network is distributed among the peers. This distribution is typically accomplished by a distributed hash table (DHT). Networks that use a DHT choose specific peers in which to place information about how to find particular files or data. These peers are chosen so that the peer's ID in the network corresponds to the file or data's ID, typically by hashing both ID's and comparing them [8].

In addition, the network topology is distributed among various peers. Each peer has a table consisting of other peers in the network and the means of communicating with them.

The contents of this table are also controlled by the protocol. The table handles the routing of requests from one node to another; when a peer receives a request it cannot fulfill, such as information about where to find a particular file, it directs the the request to the node that is "closest" to the destination of the request. How this works is determined by each protocol [15]. It should be noted that closeness is relative to the algorithm; depending on what identifiers are assigned, a node in New York City might be "close" to a node in Russia, but "far" from a node physically a few miles away. Examples of protocols that use these techniques are trackerless BitTorrent [7], Chord [15], and Kademlia [11]. As our work is implemented using Chord, we provide more detail in the following section.

A. Related Work

The Cooperative File System (CFS) is a file sharing system built on top of Chord [5] which extended the idea of storing files to storing chunks of files.

P2P-MapReduce [10], is similar to our work, but looks only at MapReduce; MapRecude is only one of the services CHRONUS provides. It consumed more network resources than the traditional centralized implementation, but was much more tolerant to churn and lost less time when nodes?jobs failed[1]. P2P-MapReduce was not implemented on a large scale; the test results for larger networks were derived from simulations

1) Folding at home?: Folding@home [2]

III. CHORD

The Chord protocol [15] takes in some key and returns the identity (ID) of the node responsible for that key. These keys are generated by hashing a value of the node, such as the IP address, or by hashing the filename of a file. The hashing process creates a m-bit hash identifier.

The nodes are then arranged in a ring from the lowest hash-value to highest. Chord then takes the hashed files and places each in the node that has the same hashed identifier as it. If no such node exists, the node with the first identifier that follows this value. This node responsible for the key κ is called the $successor$ of κ, or $successor(\kappa)$. Since we are dealing with a circle, this assignment is done in module 2^m space. For example, if there were some portion of the network with nodes 20, 25, and 27, node 25 could be responsible for the files with the keys (21,22,23,24,25). If node 25 were to decide to leave the network, it would inform node 27, who would then be responsible for all the keys node 25 was covering. An example Chord network is drawn in in Figure **??**.

With this scheme, we can reliably find the node responsible for some key by asking the next node in the circle for the information, who would then pass the request through the circle until the successor was found. We can then proceed to directly connect with the successor to retrieve the file. This naive approach is largely inefficient, and is a simplification of the lookup process, but it is the basis of how Chord theoretically works.

To speed up the lookup time, each node builds and maintains a *finger table*. The *finger table* contains the locations of up to m other nodes in the ring. The ith entry of node n's *finger table* corresponds to the node that is the $successor(n + 2^{i-1}) \ mod \ 2^m$ [2].

A. Stabilize

When a node n is told to find some key, n looks to see if the key is between n and $successor(n)$ and return $successor(n)$'s information to the requester. If not, it looks for the entry in the finger table for the closest preceding node n' it knows and asks n' to find the successor. This allows each step in the to skip up to half the nodes in the network, giving a $\log_2(n)$ lookup time. Because nodes can constantly join and leave the network, each entry in the table is periodically checked and updated.

To join the network, node n first asks n' to find $successor(n)$ for it. ,but those nodes won't acknowledge n's presence in the ring yet[3]. Node n relies on the stabilize routine to fully integrate into the ring.

The stabilize routine helps the network adjust to churn. Each node periodically checks to see who their successor's predecessor is. In the case of a static network, this would be the checking node. However, if the checking node gets back a different node, it looks at that returned node's hash value and changes their successor if needed. Regardless of whether the checking node changes its successor, that node then notifies the (possibly) new successor, essentially telling the successor "based on the information I have, I'm your predecessor. Check to see if you need to update your predecessor information," to which the successor obliges. A more concrete example:

Suppose Alice, Bob, Carol, and Dan are members of the ring and everyone happens to be ordered alphabetically (Figure 1). Alice is quite sure that Carol is her successor. Alice asks Carol who her predecessor is and Carol says Bob is. Since Bob is closer than Carol, Alice changes her successor to Bob and notifies him.

When Bob sees that notification, he can see Alice is closer than whoever his previous predecessor is and sets Alice to be his predecessor. During the next stabilization cycle, Alice will see that she is still Bob's predecessor and notify him that she's still there (Figure 2).

1) Responsibility: One of the major design choices for Chord implementation is not figuring which node is responsible for a given key, but figuring out who decides which node is responsible for a given key.

In our implementation, a node n is responsible for the keys ($predecessor(n)$, n]. In other words, when n gets a message,

[1]Fix this, actual quote was "In summary, the experimental results show that even if the P2P-MapReduce system consumes in most cases more network resources than a centralized implementation of MapReduce, it is far more efficient in job management since it minimizes the lost computing time due to jobs failures."

[2]Because hash values won't be perfectly distributed, it is perfectly acceptable to have duplicate entries in the *finger table*.

[3]This is fine. You don't have to be *in* the ring to interact with it. You just have to be able to talk to a node in the ring.

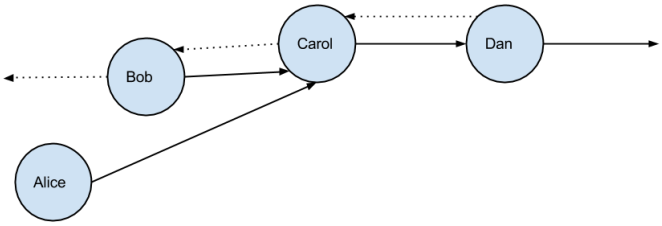

Fig. 1: Alice has incorrectly determined that Carol is her appropriate successor. When Alice stabilizes, Carol will let her know about Bob.

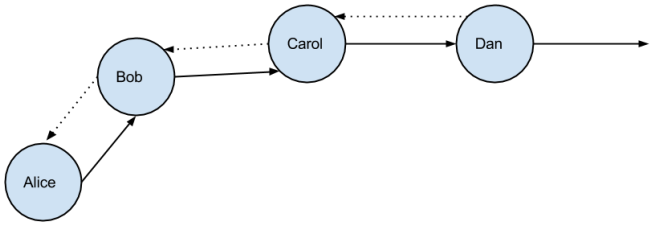

Fig. 2: After completing stabilize, Alice makes Bob her successor and notifies him. Bob then made Alice as his predecessor.

it considers itself the intended destination for the message if the message's destination hash is between $predecessor(n)$ and n. A node that does not have a predecessor assigns itself as its own predecessor and considers itself responsible for all messages it receives. COUNTER WHY THIS IS A BAD IDEA. THIS IS A VERY DEFINITE CRITICISM

2) reassigning responsibility: When a node discovers that he is no longer responsible for certain things, this is what happens

3) hanlding churn and politely quitting: Due to the potentially volatile nature of a peer-to-peer network, Chord has to be able to handle (or at the very least, tolerate) an arbitrary amount of churn. We've already detailed how Chord gradually guides nodes into their correct locations after they join the network. The same is true for when a node leaves the network; the stabilize procedure will guide nodes to their correct successors and predecessors. However, we can exert more control over how to handle nodes leaving the network

A node can leave the ring in one of two ways. A node can either suddenly drop out of existence, or a node can tell the network he is about to leave, letting his successor and predecessor immediately perform the needed changes.

When a node politely quits, he informs both his successor and predecessor and gives them all the information they need to correct *their* predecessor and successor, respectively[4]. He also sends all of the data he is responsible for to his successor, who would become responsible for that data when that node left. This allows for the network to adjust to the change with a minimum of fuss.(CHECK ISSUE WITH NODES DECIDING THAT THEY ARE RESPONSIBLE FOR MESSAGES THEMSELVES, SEE CFS I THINK .)

4) the general idea of backup: Unfortunately, it is impossible that every time a node leave the network it will do so politely. If a node suddenly quits, the data it had stored goes with it. To prevent data from becoming irretrievable, a node can periodically send backups to its successor. So as not to overwhelm the ring with a cascade of backups of backups, the node only passes along what it considers itself responsible for, which changes as nodes enter and leave the network.

[4]This is not an easy to read explanation

IV. MapReduce

At its core, MapReduce [6] is a system for division of labor. A task gets sent to a master node, who then divides that task among slave nodes, which may further divide the task. This task has two distinct parts: Map and Reduce. Map performs some operation on a set of data and then reduces a result for each map operation. This intermediate data is then reduced by other slave nodes then returned to the user.

A. Summary

Summary of MapReduce [6]

Hadoop is this an open source thingy for MapReduce [3]

B. Map

C. Reduce

V. CHRONUS

A. Implementation

Parapgraph here about the bidding nature of the amazon market. age of the inherent ring structure.

1) Code Details: We implemented the Chord based on the psuedocode in the Stoica's paper [15], using Python instead of C++. We also sent messages instead of performing remote procedure calls.

2) chunking and mechanics : Dabek CFS [5]

Design choice of CFS and replication

Rather than storing the whole file in the appropriate node, we instead split the file into arbitrary-sized chunks of data. Each of the chunks are then hashed and sent to their corresponding node.

3) backup implementation and mechanics: Paragraph about backup

4) Issues of normal MapReduce :

B. Distributed MapReduce

To do a fully distributed map reduce, each node takes on responsibilities of both a slave and master, much in the same way that a node in a p2p file-sharing service will act as both a client and a server.

1) Recap the issues of normal MapReduce and why ours is bettter:

2) Writing your own MapReduce: The big advantage of our system is the ease of development. All a developer needs to do to write three functions: the staging function, map, and reduce. The staging function splits the work into *data atoms*, which are the smallest individual units that work can be done on. This might a line of text in a document, OR OTHER EXAMPLES

The map function works as a user would expect, running a function over a series of values and outputing a list of intermediate *data atoms*. Reduce takes in two *data atoms* and combines them into one.

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 underlying Chord ring handles that automatically.

Sample code can be found in FIGURE SOME NUMBER

3) Running Map Reduce:

4) Redundancy of the "Master" Node: We don't have a master node, we have a stager. If he goes down, everything ends up in his successor

5) Redundancy of the Slave nodes: We backup our jobs and remove the backups as we're going

6) How do we know when we're done:

C. In which we address possible criticism

1) The issue with chunking and IRM: Mutually incomparable

2) If we hash a filename to get the identifier, then you can't have two same file names.:

3) Determining when you're done is expensive: As a whole yes, but merging two sorted lists is $O(n)$ time. A distributed merge sort is $O(\frac{n \cdot \log(n)}{m}$ time????? CITATION NEEDED, where m is

4) Disjoint rings:

5) The security we didn't do: We can have multiple non interacting secure file systems.

VI. Experiments

A set of experiments were run on large groups of Amazon EC2 Micro Instances[1].

VII. Results

VIII. Conclusion

Our stuff [14] is awesome.

References

[1] Amazon.com. Aws documentation: Micro instances. http://goo.gl/1yWuy.

[2] Adam L Beberg, Daniel L Ensign, Guha Jayachandran, Siraj Khaliq, and Vijay S Pande. Folding@ home: Lessons from eight years of volunteer distributed computing. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–8. IEEE, 2009.

[3] Dhruba Borthakur. The hadoop distributed file system: Architecture and design, 2007.

[4] WU Chehai, LU Xianliang, D. Hancong, T. Hui, Z. Xu, and Z. Zhijun. Analysis of content availability optimization in bittorrent. In *Hybrid Information Technology, 2006. ICHIT'06. International Conference on*, volume 1, pages 525–529. IEEE, 2006.

[5] Frank Dabek, M Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with cfs. *ACM SIGOPS Operating Systems Review*, 35(5):202–215, 2001.

[6] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[7] R. Jimenez, F. Osmani, and B. Knutsson. Connectivity properties of mainline bittorrent dht nodes. In *Peer-to-Peer Computing, 2009. P2P '09. IEEE Ninth International Conference on*, pages 262 –270, sept. 2009.

[8] Eng Keong Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim. A survey and comparison of peer-to-peer overlay network schemes. *Communications Surveys Tutorials, IEEE*, 7(2):72 – 93, quarter 2005.

[9] Qin Lv, Pei Cao, Edith Cohen, Kai Li, and Scott Shenker. Search and replication in unstructured peer-to-peer networks. In *Proceedings of the 16th international conference on Supercomputing*, ICS '02, pages 84–95, New York, NY, USA, 2002. ACM.

[10] Fabrizio Marozzo, Domenico Talia, and Paolo Trunfio. P2p-mapreduce: Parallel data processing in dynamic cloud environments. *Journal of Computer and System Sciences*, 78(5):1382–1402, 2012.

[11] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the xor metric, 2002.

[12] S. Ortiz. Is peer-to-peer on the decline? *Computer*, 44(2):11 –13, feb. 2011.

[13] Pablo Rodriguez, See-Mong Tan, and Christos Gkantsidis. On the feasibility of commercial, legal p2p content distribution. *SIGCOMM Comput. Commun. Rev.*, 36:75–78, January 2006.

[14] Andrew Rosen, Brendan Benshoof, and Matt Erwin. Chronus. https://github.com/BrendanBenshoof/CHRONUS.

[15] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *SIGCOMM Comput. Commun. Rev.*, 31:149–160, August 2001.

[16] Chonggang Wang and Bo Li. Peer-to-peer overlay networks: A survey. Technical report, 2003.