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 [**?**].

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 [**?**]. Many companies have embraced legal uses of P2P technology, such as providing ways to stream media or provide large updates [**?**]. 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 [**?**].

1) SaaS: in recent years, there has been a trend to USE P2P AS A WAY OF DISTRIBUTING WORK FOR SPECIFIC APPLICATIONS. Folding at home 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 [**?**] [**?**].

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 [**?**] [**?**].

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 [**?**]. This makes these types of networks also a poor choice for implementation [**?**].

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 [**?**].

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 [?]. 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 [?], Chord [?], and Kademlia [?]. As our work is implemented using Chord, we provide more detail in the following section.

A. Related Work

CFS is a file sharing system built on top of Chord [?].

P2P-MapReduce [?], 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

III. CHORD

The Chord protocol [?] 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 and Bob are members of the ring and Alice is quite sure that Bob is her successor. Alice asks Bob who his predecessor is and Bob says Carol is. If Carol's hash puts her between Alice and Bob, Alice changes her successor to Carol and notifies Carol. Carol gets the nofitication from Alice. If Carol sees Alice's hash is between Carol's own predecessor and herself, she makes Alice her predecessor.

During the next stabilization cycle, Alice will check to see who Carol's predecessor is and adjust her successor and predecessor based on that.

B. Redundancy

The node $successor(\kappa)$ is responsible for all the values between it's

IV. MAPREDUCE

At its core, MapReduce [?] 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 preduces a result for

---

[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.

each map operation. This intermediate data is then reduced by orther slave nodes then returned to the user.

A. Summary

Summary of MapReduce [**?**]

Hadoop is this an open source thingy for MapReduce [**?**]

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 [**?**], using Python instead of C++. We also sent messages instead of performing remote procedure calls..

2) chunking and mechanics : Dabek CFS [**?**]

Desiugn choice of CFS and replication

3) backup implementation and mechanics: Paragraph about backup

4) Disjoint rings:

5) The security we didn't do:

B. Distributed MapReduce

To do a fully distriuted map reduce, each node takes on responsiblitilies 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.

One of the key issues in

Rather than the traditional tree structure, Chord takes advant

VI. EXPERIMENTS

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

VII. RESULTS

VIII. CONCLUSION

Our stuff [**?**] is awesome.