Map Reduce on a Chord Distributed Hash Table

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—**Large problems in Computer Science are often able to be split into many smaller, functionally identical tasks, the results of which can be reduced into the desired solution. MapReduce is a framework for performing distributed computations of these types of problems, but most of these frameworks are strongly dependent on a heirarchical structure. Chord is a distributed hash table that provides $\log_2 n$ connectivity among processors. Our experiments show that a Chord ring has number of desirable advantages for implementing MapReduce, such as a lack of a single point of failure, fault-tolerance, an even distribution of work, scalability, and minimal overhead.**

I. Introduction

Ever since Google published a paper on the subject [**?**], MapReduce has rapidly become an integral part in the world of data processing. Using MapReduce, a user can take a large problem, split it into small, equivalent parts and send those parts off to be worked on by other processors. These results of these computations are then sent back to the user and combined until one large answer results. Numerous programming tasks - word counts, reverse indexing, sorting, Monte-Carlo approximations - can be efficiently distributed using MapReduce [**?**].

The most popular platform for MapReduce is Hadoop [**?**]. Hadoop is an open-source Java implementation deveolped by Apache and Yahoo! [**?**]. Haddop has two components, the Hadoop Distributed File System (HDFS) and the Hapdoop MapReduce Framework [**?**]. Under HFDS, nodes are arranged in a heirarchical tree, with a master node, called the NameNode, at the top. The NameNode is responsible for keeping track of whcih DataNodes posess which files, as well as coordinating the work done under a MapReduce job **double check the latter**.

However, what if we desire a less heirarchical structure among our nodes? A single node in charge is a single point of failure. We need a system that can scale, is fault tolerant, has a minimal amount of latency, and distributes files evenly.

Chord [**?**] is a distributed hash table (DHT) that possesses these qualities. Chord guarantees a worst-case $\log n$ lookup time for a particular node or file in the network. It is highly fault-tolerant to node failures and churn. It scales extremely well and there is little maintenance required by the network as a whole to handle individual nodes. Files in the network are distributed evenly among its members.

However, rather that viewing Chord solely as a means for sharing files, we see it as a means for distributing work. We have developed a system, called CHRONUS, that builds off the concepts that make Chord a powerful means of evenly distributing files and applies it toward distributing work among member nodes[1]. We built a framework on CHRONUS for performing Map and Reduce tasks on Chord ring. CHRONUS leverages the underlying protocol to distribute map and reduce tasks to nodes evenly, provide greater data redundancy, and guarantee a greater amount of fault tolerance. The network has no single point of failure.

At the same time we avoid the architectural and file system constraints of systems like Hadoop. Nodes in CHRONUS can be setup in a cluster for high performance or they can be deployed the Internet, for volunteer computing tasks. Our experiments demonstrate that our framework is highly scalable, solving problems significantly faster when distributed. The larger the problem is, the greater the speedup gained by incorperating more nodes into the problem. Our framework also provides a high level of robustness during execution; we can lose any node to churn and still process jobs successfully[2].

Section X covers the specifics of the Chord Protocol and the design choices we made for implementation. We describe MapReduce and Hadoop in more detail in Section QA4162. Related Work is discussed in Section 42. Details of CHRONUS's implementation and code is described in Section Q, while our experiments and their results are covered in Section Z. Lastly, Section L discusses fruitful avenues of future research.

II. 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 and port, 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 the overlay is a circle, this assignment is computed 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

[1]repetitive
[2]Brendan, check

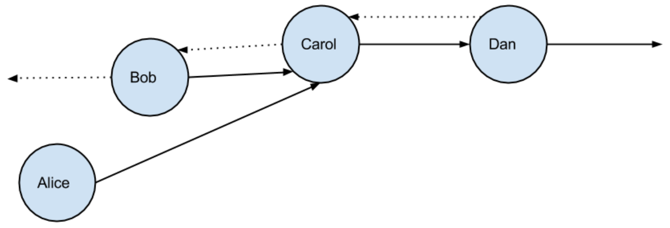

Fig. 1: Alice has incorrectly determined that Carol is her appropriate successor. When Alice stabilizes, Carol will let her know about Bob.

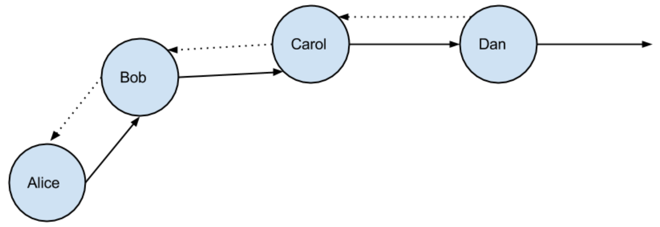

Fig. 2: After completing stabilize, Alice makes Bob her successor and notifies him. Bob then made Alice as his predecessor.

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 INSERT FIG HERE.

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 $successor(n + 2^{i-1})$ $mod\ 2^m$. Because hash values won't be perfectly distributed, it is perfectly acceptable to have duplicate entries in the *finger table*.

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. Node n uses the information to set his successor, but the other nodes in the ring will not acknowledge n's presence yet. Node n relies on the stabilize routine to fully integrate into the ring.

The stabilize routine helps the network integrate new nodes and route around nodes who have left the networls. 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 **??**). 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 **??**).

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, 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.

When a node n changes his successor, n asks if the successor is holding any data n should be responsible for. The successor looks at all the data n is better suited to hold onto, packages it up, and sends it along to n. The successor does not have to delete this data. If fact, keeping this data as a backup is beneficial to the network as a whole, as n could decide to leave the network at any point.

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 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 fill the gap that would be left over. 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. Fingers that pointed to that node would be corrected during the finger maintenence period. This allows for the network to adjust to the change with a minimum of fuss.

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. If the backup leaves, he send his stuff to his successor, since the backup's successor would be the one responsible for the info now.

A. Extensions of Chord

The Cooperative File System (CFS) is an anonymous, distributed file sharing system built on top of Chord [**?**]. In CFS, rather than storing an entire file at a single node, the file is split up into multiple chunks around 10 kilbytes in size. These chunks are each assigned a hash and stored in nodes corresponding to their hash in the same way that whole files are. The node that would normally store the whole file instead stores a *key block*, which holds the hash address of the chunks of the file.

The chunking allows for numerous advantages. First, it promotes load balancing. Each piece of the overall file would (ideally) be stored in a different node, each with a different backup or backups. This would prevent any single node from becoming overwhelmed from fulfilling multiple requests for a large file. It would also prevent retrieval from being bottlenecked by a node with a relatively low bandwidth. Finally, when Chord uses some sort of caching scheme like that described in CFS [**?**], caching chunks as opposed to the entire file resulted in about 1000 times less storage overhead.

Chunking also opens up the options for implementing additional redundancy such as erasure codes[**?**]. With erasure codes, redundant chunks are created but any combination of a particular number of chunks is sufficient to recreate the file. For example, a file that would normally be split into 10 chunks might be split into 15 encoded chunks. The retreival of any 10 of those 15 chunks is enough to recreate the file. Implementing erasure codes would presumably make the network more fault tolerant, but that is an exercise left for future work.

III. MapReduce and Hadoop

1) What is it: MapReduce became a major topic of interest in 2004 when Google published a paper detailing their implementation, called MapReduce [**?**].

At its core, MapReduce [**?**] is a system for division of labor, providing a layer of speration between the programmer and the nastier parts of parallel processing. The programmer sends a large task 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 produces a result for each map operation. This intermediate data can then be reduced, combining these sets of intermediate data into a set, which is further combined with other sets. This process continues until one set of data remains.

The classic example given for MapReduce is counting the occurrence of each word in a collection of documents. The master node splits up the documents into multiple chunks and sends them off to workers. Each worker then goes through each chunk and creates a small word frequency list. These lists are then used by other workers, who combine them into larger and larger lists, until the master node is left with a word frequency list of all the words in the documents.

One very popular implementation of MapReduce is Apache's Hadoop. [**?**]. Hadoop is an Apache open-source project and backed by Yahoo!.

2) Requirements for a viable MapReduce platform: scalability and fault-tolerence.

Chord is highly scalable and fault tolerent. Chord guarantees a latency of $\log(n)$, making it extremely scalable. Chord's ability to automatically assign nodes responibilty as the ring adds and loses nodes makes it extremely fault tolerant.

3) Why are we an interesting and viable alternative to Hadoop: Here is a list.

- Much easier to program on ours? Citation of complaints.
- Hadoop has a specific architecture. Ours is really easy to setup. Step 1: add node to ring. Step 2: there is no step 2.
- Hadoop is bound to it's filesystem. For our system to use a database, make the query during stage(). Or if part of the job requires making queries to the database, just write those calls in.
- If I'm reading this correctly, Hadoop keeps track of where the data is stored and asks the node where the data is stored to do the map tasks. In CHRONUS we don't make any assumption about the data, including where it's coming from. This means we can lose time feeding in our data
- Hadoop has master node bottle neck issue [CITATION]. Ours can have that happen but it's much less of an issue because we can collate on the way back. This would take extra $waittime \log(n)$ time for the reduce step.
- Hadoop is more latency tolerant. There are ways to work around that. BRENDAN EXPOUND HERE.
- I seem to get the implication from [**?**] that Hadoop must wait for all maps to finish before reducing. CHRONUS doesn't.
- CHRONUS doesn't rely in a scheduler; we just send messages.
- There is no single point of failure for CHRONUS. Our master node can go down and we don't care. Hadoop's name node is a single point of failure [**?**].
- DataNodes register with the NameNode, so that the

DataNode's identity and responsibilities persist even if the node is restarted under different address or port.
- no need for heartbeats. HFDS has nodes send a heartbeat to the NameNode every 3 seconds [?].

IV. Related Work

We have identified two extant papers that have done work on combining P2P concepts with MapReduce. Both papers are similar to our research, but differ in crucial ways

A. P2P-MapReduce

Marozzo et al. [?] looked at the issue of fault tolerance in centralized MapReduce architectures such as Hadoop. They focused on creating a new, P2P based MapReduce architecture called P2P-MapReduce. P2P-MapReduce is designed to be more robust, able to better handle node and job failures during execution.

Rather than use a single master node, P2P-MapReduce would have 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. Failures of the slave nodes are handled by the master node responsible for it.

Experimental results were gathered via simulation. Their results showed that while P2P-MapReduce could generated an order of magnitude more messages than a centralized approach, the difference rapidly began to shrink at higher rates of churn. However, when looking at actual amounts of *data* being passed around the network, rather than the number of messages, the bandwidth[3] required by the centralized approach greatly increases as a function of churn, while the distrubuted approach again remains relatively static in terms of increased bandwidth usage and messages sent.

RUNTIME RESULTS?? 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[4] [?].

B. Parallel Processing Framework on a P2P System Using Map and Reduce Primitives

[?].

- Their work, like ours, 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.
- based off of an implementation of Symphony [?], DHT protocol very similar to Chord.
- Their churn is handled by the underlying protocol and no special arrangements have been made for handling churn during a job.
- If their ad-hoc master node goes down during the execution, there is no recovery.

Both these papers have promising results and confirm the viablility of our own effort. The experimental results in P2P-MapReduce demonstrate that while there is additional overhead resulting from the fault-tolerant aspects of the system, it's more than made up for by job failures [?].

Both systems have their masters take a very hands on approach, whereas CHRONUS leverages the neighbors of the nodes to handle any problems, like recovery.

V. CHRONUS

CHRONUS, at its core, is a fully functional Chord implementation in Python[5]. Our installation was designed to be as simple as possible. It consists of downloading our code[?] and running chord.py. You can specify the port you plan on using and the IP and port of a node in the ring you want to join. Once that is done, the node will automatically integrate into the ring. Other than a hashing library, we wrote the protocol and networking code from scratch. Once we could create a stable ring, we created various services to run on top of our C, such as a file system. Our file system is capable of storing whole files or splitting the file up among the ring. Our MapReduce module is a service that runs on top of our Chord implementation, just like our file service.

In our implementation of a distributed map reduce, each node takes on responsibilities of both a worker 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. To start a job, the user contacts a node a specified hash address. The node he contacts to be the stager may be his own computer, and this is preferable when the job involves dividing up a large pile of data.

The job of this stager is to take the work and divide it into *data atoms*, which are the smallest individual units that work can be done on. This might a line of text in a document, the result of a summation for a particular intermediate value, or a subset of items to be sorted. The specifics of how to divide the work are defined by the user in a *stage* function. These *data atoms* are then given a random hash and sent to that hash address, guaranteeing an even distribution of the data atoms throughout the network. The *data atoms* also contain the map function and reduce function, as defined by the user.

Nodes which receive data atoms apply the map function to the data to create intermediate data atoms, which are sent to other BLANK. DESCRIBE HOW TWO INTERMEDIATE ATOMS GET TO THE SAME PLACE. PULSED REDUCE EVERY .1 seconds. Nodes that receive at least two intermediate values merge them into one data atom using the reduce function. The atoms are continually merged until only one remains at the hash address of the stager.

Once the reductions are finished, the user gets his results from the node at the stager's address. This may not be be the stager himself, as once the stager has sent all the data atoms,

[3]ANU, is this the correct term?

[4]"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."[?]

[5]The original implementation is in C++

his job is done. The stager does not need to collect the results work, since the work is sent to the stager's hash address, rather than the stager itself. Thus, the stager could quit the network after staging, and both the user and the network would be unaffected by the change.

We avoided any complicated additions to the Chord architecture; instead we used the protocol's properties to achieve what we want. The Stager process randomly assigns a destination hash to each job it sends out, knowing the even distribution of the network will ensure that each node that ends up getting a Map task will be doing an equal amount of work.

If node n was responsible for collecting all the data and suddenly goes down, his successor will automatically receive all the incoming data, which he can add to his backup of the data n had. Here, we are leverging two features. First, we use the automatic assignment of responsibility to automatically route the data to the sucessor. Second the same process Chord uses to backup files is used to backup the intermediate data.

Similar precautions are taken for nodes working on map and reduce tasks. Those tasks get backedup by a node's successor, who will run the task if the node quits the network before the doing the job (eg the successor loses his predecessor). The only addition to the architecture we've made here is that a node n will send a message to his sucessor when n's map or reduce task is done. This message tells n's successor that the backup is no longer needed. This is done to prevent a job being submitted twice if n goes down after finishing his task and sending it out.

1) Recap the issues of normal MapReduce and why ours is better: We present our implementation of MapReduce not as a direct competitor to MapReduce, but as proof of a more versatile system able to support many complex operations.

The big advantage of our system is the ease of development.

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. If a node joins the ring as the MapReduce process is running, that node can be assigned work automatically. The stager does not need to keep track of the status of the network. In a node goes down while performing an operation, his successor takes over for him.

All a developer needs to do to write three functions: the staging function, map, and reduce. These define how to split up the work into manageable portions, the work to be performed on each portion to get some results, and how to combine these results into a single result.

No single node needs to keep track of all other nodes. Each node is automatically looked after by other nodes via the Chord protocol.

2) Calculating Pi: To stress test the network, we use MapReduce to run a Monte-Carlo approximation of π.

3) Word Count:

4) My computer can do way more work: In which case you boot up more instances of a node, declaring you can do that much more of the network's work.

A. In which we address possible criticism

1) Mirco versus small: We had huge issues with micro instances because of their sporadic cpu power. Less of an issue with Small instances.

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.

4) Disjoint rings: We can assign every node a *ring id*. When a node creates a ring, he randomly chooses a number for the ring id. Nodes joining that ring use that ring's id. If a node finds another node on differing ring, he compares the their ring id's and leaves if

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

6) File system via hash prefixing: First k bits of hash used to describe files

VI. Experiments

A. Setup

1) Execution: Once started, nodes would pull the latest version of the code and run it, automatically joining the network. We could choose any arbitrary node as the stager and tell it to run the MapReduce process. We found that the network was robust enough that we could take a node we wanted to be the stager out of the network, modify it's MapReduce test code, have it rejoin the network, and then run the code new code without any problems. Since only the stager has to know how to create the map tasks and what number, the other nodes don't have to be updated and are happy to perform the jobs they are given..

B. Results

VII. Conclusion and Future Work

Our stuff [**?**] is awesome. CHRONUS can be adapted to handle mutable data, unlike other MapReduce schemes. Tackle security and distrubuted authentication. Really security, We're aware that as our prototype code currently stands, and adversary could make his map and reduce tasks do anything he wants(I direct your attention to the word *protoype*)