MIT Distributed Systems 6.824 Capstone Project

Vincent Zheng
CSCI 49900
Section 4

***Abstract*—MIT Course 6.824 of Spring 2014 covers both abstractions and implementation techniques for engineering distributed systems. Major topics include fault tolerance, replication and consistency. This paper will go in depth about the details of Labs 1 to 3.**

I. Background

MIT Course 6.824 is a 12-unit graduate course which included readings along with their respective labs. Before beginning to tackle these labs, it was necessary to learn the basics of the Go programming language. In addition to learning Go, it was also necessary to learn the concepts of distributed systems from a higher level perspective before diving into the more specific details. This was done through the provided readings but also through other text resources and videos.

II. Introduction

This paper will describe the learning process and implementations of the required tasks in the MIT 6.824 labs. The labs required a high level of critical thinking to complete each section and built off of one another. Weaknesses in one part of the lab would be addressed in the next. The labs start off by tasking us with implementation MapReduce which is achieved through the use of multiple workers who either perform the Map or the Reduce functions of the algorithm. Later, we explore the issues of this type of implementation and how to address each one.

III. Experiments

A. Lab 1: MapReduce

Part I: Word Count

The very first section of Lab 1 requires us to implement the MapReduce algorithm. We are already given a template of the MapReduce function included with the lab. We are also given an input file that has been provided for us which is taken in as a parameter to the template MapReduce function.

MapReduce works by first splitting up this input file into multiple pieces and distributing these pieces to multiple ”workers” or servers. These servers will produce key, value pairs where the key will be the words within the piece of the file that they received and the key will first be mapped to the value 1. Note that during this phase, there may be duplicate words as keys. This part of the algorithm is referred to as the ”Map” phase.

After the map phase is finished, these key/value pairs will be sent to the workers tasked with the ”Reduce” phase. This part of the algorithm is the ”shuffle” phase which involves grouping key values by sending the same key value to specific workers. For example, worker 1 may receive the words ”dog” and ”cat” while worker 2 receives the words ”mouse” and ”gorilla”. This allows for us to move on to the ”Reduce” phase involves summing up the total number of occurrences for each word for the file as a whole. This is done by identifying duplicate keys, which have been already grouped up by the ”shuffle” phase and summing up their counts to achieve the total word count for the file.

Part II: Distributing MapReduce Jobs

Going further into the specifics of the distribution of labor, the first thing is what is the optimal number of workers required to complete a single MapReduce task. Within the template code, we are given the number of Map jobs generated and the number of Reduce jobs generated stored in MapReduce struct under the variables nMap and nReduce. The approach taken was to generate as many Map workers are there are jobs and to do the same for reduce workers. Map workers and Reduce workers would be generated and stored into two different channels. Whenever a new Map or Reduce job was available, the workers would be assigned to a specific job. The channels are constantly polled to check if there is a worker available to be assigned a job.

It is important to note that since we are assigning tasks across servers, we must utilize Remote Procedure Calls(RPCs) to do so. In simple terms, an RPC is a procedure which allows us to request a service from a program located on a different server. Our code assigns one single master, which generates workers and then, through the use of RPCs assigns these workers either a Map or Reduce job. Once all jobs are completed, the workers are then killed.

Part III: Handling Worker Failures

Handling worker failure at this stage is quite simple because there is no need for workers to keep a persistent state. If a worker fails any RPCs given to it by the master server, then the task just needs to be reassigned to a different worker. However, an RPC failure does not necessarily mean that a worker failed. It could simply mean that the worker is unreachable at this point in time.

An RPC failure is detected in our code through the return output of our RPC. If the RPC returns a 0, then this is considered an RPC failure, and if it returns a 1, this is a success. Another important thing to note is that if the same task is given to another worker while a worker is already working on this task, it does not matter because both should return the same output in the end. Thus, we do not need to worry about this. At this point, one might ask themselves what would happen if the master server failed. In this particular stage, the whole system would fall apart. This part will be address in Lab 2 and will be resolved with the implementation of a viewservice.

B. Lab 2: Primary/Backup Key/Value Service

Part I: The Viewservice

During this lab, we will be implementing a viewservice that keeps track the primary, backup, and idle master servers. These servers will provide a key/value service and will constantly be updated to remain consistent with one another about which keys and values are the correct ones after jobs are completed. The primary, backup, and idle servers are of course, subject to change due to master failure. Therefore, the view service must keep track of numbered ”views”. These ”views” will keep track of the current primary, backup, and idle at the time. If this happens to change, the view will be updated to reflect the new servers and their roles. The primary in a view can only be the current primary or the backup of the previous view. This helps ensure that the key/value service state is preserved. The only exception is when the viewservice is first started up. In this instance any server can be accepted as the primary. The backups for the viewservice can be any server as long as it is not the primary.

Every key/value server(master, backup, idle) should send a ping to the viewservice from time to time. This interval can be specified by the programmer. The viewservice will acknowledge this ping by returning a description of the current view(which server is currently primary, backup, and idle). The purpose of the ping is to let the viewservice know that the server is still alive. The acknowledgment lets the key/value servers know about the current view and which role the server plays in this view.

A server is considered dead by the viewservice if it does not receive a ping from a server for the specified interval. If a server dies, and is restarted, it will notify the viewservice that this occurred. If the primary dies, the backup will take over. If there is no backup, the idle server will take over. Whenever any of these cases occur, the view must be changed. However, the view must not change until the viewservice receives an acknowledgement from the current view’s primary. This rule prevents the viewservice from getting more than one view ahead of the key/value servers. The downside is if the primary dies before it ever sends an acknowledgement, the viewservice cannot change views and spins forever. If the backup dies, the idle will take over for the backup.

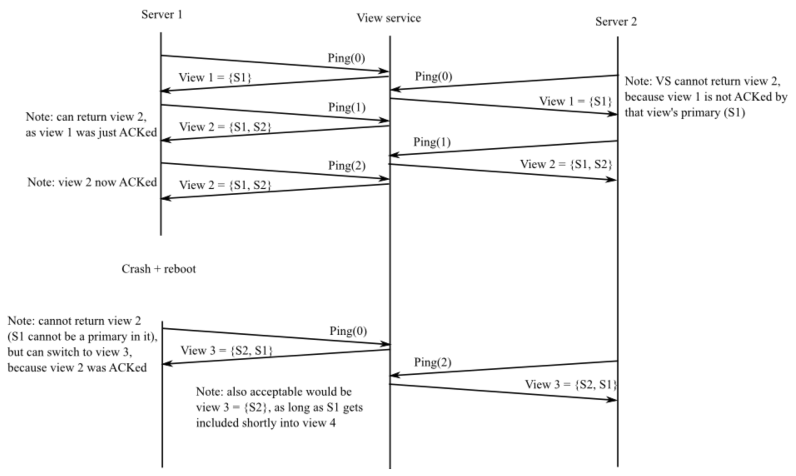

Server 1
View service
Server 2
Ping(0)
View 1 = {S1}
Ping(0)
Note: VS cannot return view 2, because view 1 is not ACKed by that view's primary (S1)
View 1 = {S1}
Ping(1)
Note: can return view 2, as view 1 was just ACKed
View 2 = {S1, S2}
Ping(1)
Ping(2)
View 2 = {S1, S2}
Note: view 2 now ACKed
View 2 = {S1, S2}
Crash + reboot
Note: cannot return view 2 (S1 cannot be a primary in it), but can switch to view 3, because view 2 was ACKed
Ping(0)
View 3 = {S2, S1}
Ping(2)
View 3 = {S2, S1}
Note: also acceptable would be view 3 = {S2}, as long as S1 gets included shortly into view 4

Fig. 1. Sequence of view changes

Part II: The Primary/Backup Key/Value Service

The primary/backup key/value service is responsible for keeping track of the keys and values outputted from the MapReduce jobs. This was the job of the single master server before the implementation of the viewservice. Currently, we do have a primary master, but we have a backup server and an idle server ready to take over the role of master if necessary. Both the backup and the idle servers are also keeping track of which key/value pairs are the correct ones, not just the primary server.

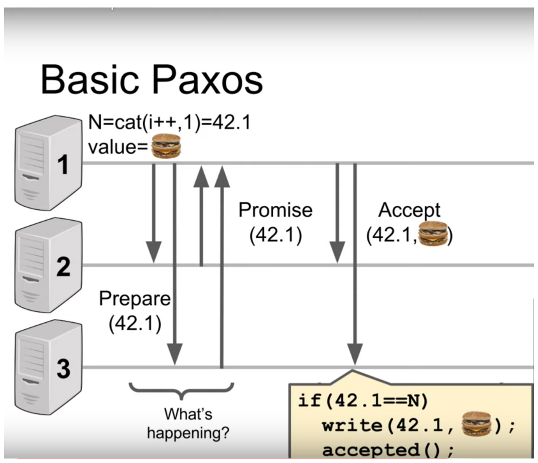
Key’s values are updated with a put() function while the get() function returns the key’s most recently updated value. The primary must send gets as well as puts to the backup, if there is one, and must wait for the backup to respond before serving a request. Puts are followed by gets to ensure that the backup and primary servers have the same values for keys. Clients request keys using their get() functions which the primary can serve and can also request that the primary server updates a value through their put() function. Puts and gets should only return if the operation has been successfully completed. Otherwise, it will keep on trying until it is finished. It is important to note that the servers(master, backup, idle) do not need to constantly communicate with the viewservice for each put() and get(). The viewservice should rely on pings and the servers should rely on the acknowledgement of the viewservice which sends the servers a description of the current view.

C. Lab 3: Paxos-based Key/Value Service

Part I: Word Count

Again, we have run into the issue where there is a single point of failure. We solved the issue of having a single master server keep track of all key/value pairs but since we only have one master view server, if this view server fails then the whole system falls apart once again. This is where the Paxos algorithm comes in. The Paxos algorithm works by having a group of servers(peers) decide on whether or not a certain value is correct for a specific key. A value can only be decided on if the majority of peers accepts that this specific value is the correct one. This means that more than half of the servers must respond that they accept the value as the correct one.

Going into further detail, Paxos assigns three roles to all existing peers: proposer, acceptor, and learner. Each peer can play all three of these roles for a specific round of Paxos. A round begins when a peer decides to take on the role of being a proposer and decides to propose a value. This is known as the prepare phase. This proposal will contain two values, a round-identifier and the value itself. The round-identifier is just a unique number that helps differentiate one round from the other previous rounds. Each new proposal should contain a round-identifier that is greater than all previous proposals. This pair of values is usually in the form (n,v) where n is the round-identifier and v is the value. The proposal is sent out to other peers. These peers will then play the acceptor and learner roles. The accept phase occurs when each individual acceptor receives the proposal and examines this value to check if this round-identifier is the highest round-identifier it has seen so far. If it is not, the acceptor will reject this proposal and reply with a deny message. If the acceptor sees that this round-identifier is higher than that of all of its previously encountered proposals, it will reply with a ”promise” message which is a promise to accept this value. This occurs across all acceptors and if the proposer receives a promise count that is greater than the majority of all peers, the decide phase will begin and the proposer can then ask an acceptor to broadcast that this specific proposal is to be accepted by all peers.



```
if(42.1==N)
  write(42.1, 🍔);
  accepted();
```

Fig. 2. Basic Paxos Algorithm

This algorithm solves the issue and we now no longer have a single point of failure. However, failure can still occur at multiple stages within our algorithm and to any of the peers. If failure happens to an acceptor at the accept phase, as long as a majority of the acceptors respond with a promise to the proposal then the algorithm will just continue. If a majority of acceptors do not respond with a promise then the value is then rejected. Failure can also occur at the prepare phase where it fails before it can send a promise to the proposer. Again, if a majority of acceptors respond with a promise, then the next phase can still occur. If a majority of acceptors end up not responding with a promise, then the proposal is rejected. Proposer failure can occur at the prepare phase, and after sending a prepare message. Failure at the prepare phase and after sending a prepare message means that another peer will take on the role of a proposer and propose a new value with a higher-round identifier. If a proposer fails at the accept stage, a new proposer will take over and try to propose a new value with a higher round-identifier. If an acceptor has already promised to accept the previous proposer’s value, it will notify the proposer that a value has already been accepted. The proposer will then create a higher value round-identifier but with the previously accepted value.

IV. Conclusion

Starting from a very basic implementation of the MapReduce algorithm, the MIT 6.824 course guided the creation of a complex distributed system. Lab 1 helped create a basic distributed system that divided the labor of the different stages of MapReduce amongst a group of workers. Lab 2 solved the issue of master failure by implementing a viewservice that maintained which servers were the primary, backup, and idle servers. However, this created a new single point of failure: the viewservice. To resolve this issue, Paxos was implemented in lab 3. Each lab built on top of the previous labs to address specific issues with the implementation of a clear and concise solution. Although there are still many issues with the current state of the implementation of a distributed system, those issues are out of the scope of this project. However, it is still important to acknowledge that although many points of failure and many issues were resolved and addressed, there are still many more that need to be evaluated and fixed.