Final Project Report : Locating Criminals Using MapReduce

Cangji Wu , Jiaran Hao, Kangyi Zhang

April 13, 2014

Abstract

This is our final project for the course Distributed System. To understand the design principles of distributed system better, Our group decide to implement an application using MapReduce programming model. Our project emulates the situation where police intend to locate a criminal using the footage from transportation cameras. As the input data set could be huge and response time is valued, MapReduce is applied to solves the task distributedly to improve efficiency. In our implementation of MapReduce system, the mechanism to maintain scalability and fault tolerance is emphasized .

1 Introduction

Our project try to help the police to build a Criminal Positioning System. Imagine the police want to find where the certain criminal is, what he would do? The first step would be find a comparitively clear photo of the face of the criminal, and then using the video streams or pictures from the transpotation camera or other devices of the whole town or city to find whether the criminal has shown in the specific area, if has, he might show again in the area where he has shown most of times. Then the police can send policemen to this area to wait to catch this criminal until he shows again.

We know that in computer vision technology, it is not very hard to recognize a person from the video stream or the picutres, however, this might take 1 second each picture for normal computer. To catch the criminal, we need to position the criminal as far as possible, if the police do not own a

super computer, it would take several days to get the location. However, with map reduce, they can distribute the jobs to several computer to search the criminal at the same time, this would save a lot of time!

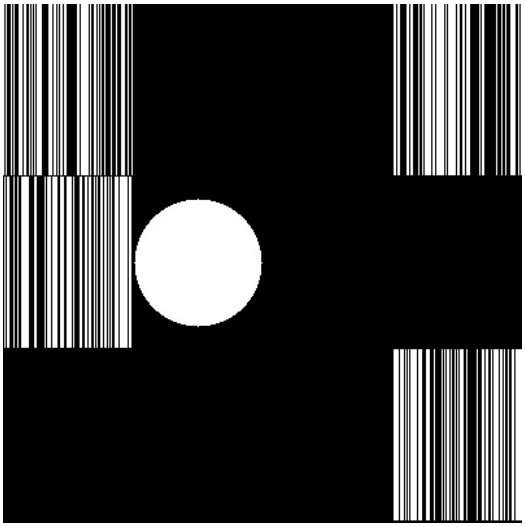

Figure 1: A picture of a gull.

2 Basic Concepts

3 Implementation

3.1 Components

In our implementation, we have five main components. Map function and Reduce function are separated from Worker such that any Map function and Reduce funtion which follows the interface can be integrated into our system. Master and Worker are designed to manage the whole structure. In other words, they manage the communication that is essential no matter what specific Map and Reduce function is running on the infrastructure. File system... Following this design, we separate our application and infrasture so that our design is flexible enough for any job that fits into MapReduce model.

3.2 Locating Algorithm

3.3 File System

3.4 Master

Master manages the whole MapReduce system. It has 4 main functions:
First, master keeps record on who and where the workers are in a table. It holds the IP address and port number of each worker who participates in completing the job. The workers send "JOIN" message to the master then master will assign a unique worker ID for the newly join worker. Then the worker will keep sending "UPDATE " message to the master so that the master would know if a worker has left or crashed.

Second, with the all the information of its workers known by the master, it will assign jobs. It will first divides the whole input as equally as possible then assign a job ID for each job. Then master will informs the workers and tell them what job they account for. The jobs fall into two categories, Map and Reduce. For map worker, the master tells them where to look for the input data on the global file system and its job ID. For Reduce worker, the master tells them from which map workers they should fetch the intermediate results from. After assigning the jobs, the worker table is also updated to keep track all the information so that when node fails, it can find another node to take place of the failed one.

Third, master also plays an important role in identifying failed workers and recovery from fault. The concrete mechanism is dicussed in the fault tolerance section.

At last, master will monitor the working state of its workers such that it will know who has completed its assigned jobs. When all the reduce workers complete jobs, the master will know that it is ready to output the result to the user and will notice all the map and reduce worker to delete their temporoary results and free the resources that they are holding. In addition, by keeping the status of workers, the system can improve efficiency by transferring the jobs from heavy-loaded workers to the ones who already finish their job such that no node will become the bottle-neck. Keeping the work load even is essential to the scalability of the system.

3.5 Worker

In MapReduce model, the worker's job seems simple : if one is assigned as a map worker, it should fetch the input in the global file system and call the Map fuction. Then it stores the intermediate result on local disk, waiting for the reduce worker to ask for data. The reduce worker is even simpler. Just fecth the imtermediate result from map workers, call Reduce function and write the final result to the global disk.

However, a more efficient and fault-tolerant system requires more consideration. A worker should be designed to take more than one job. In this way, we can transfer jobs from heavy-loaded workers to less loaded ones or reassign jobs if one node fails. In Google thesis on MapReduce, they just assume that there would always be enough workers left. While our implementation wants to make sure the system will give the correct input no matter how many nodes go down. If only one node is left, it will take all the jobs itself. This design makes sense because it first ensures correctness then allows improvement on efficiency by adding more workers. In addition, when the reduce workers fetch imtermediate data, the map workers they try to connect may already be down. If so, the reduce workers should have the mechanism to wait for the master to inform them that they should inquire another node instead of the crashed one.

Therefore, when master assigns jobs, each job has a job ID. The reduce worker will keep a map whose key is a unique job ID and value is the contact information of relevant map node. Similarly, map workers also keep a map using job ID as the key as they may hold intermediate result of several different jobs.

3.6 Fault Tolerance

Most princinple in designing distributed system is atucally helping people to fight node fault. In our design, we combine three most common techniques used in the distributed systems to tolerate faults.

First, replication. Several copies of the input data are stored on different node suchs that if one goes down, the substitute will always have access to required input. Actually, the distributed file system itself can be a very complex component instead of our simple stimulation. The file system is a key part of the Hadoop platform.

Second, heart beat. In our system, every node keeps update message

to the master, the master has a timer to count the interval of each update message. If one node fails to send the update message for some reason, the master goes into a process to reassign jobs. It will first find the least loaded node and read the worker table to get all the jobs that are assigned to the failed node then transfer those jobs to the least loaded node. If a job is a Map job, it will also find which reduce worker is responsible for this map job and tell this reduce worker to go to the new node. Here, our job ID can be very convenient. We can just use to job ID to access the map in the reduce worker and change the value(address).

Note that this reassign process must reassign the reduce job first and then the map jobs. Because in our implementation a worker can have both map job and reduce job at the same time and the reduce job may wants to fecth data from the same node. If the reduce job are reassigned first, the master will inform the new node when reassigning the map job. For example, at first there are 3 nodes A,B,C and 3 jobs 1,2,3. A,B are map workers and C is reduce worker. Frist C goes down then master reassigns C's job(job 3) to B. Now B has one map work(job 2) and one reduce work(job 3). The reduce job(job3) says it should ask node A and node B for data. Sadly, B goes down too. At this very point, we should reassign B's job3 first thus we reassign it to A. Then we reassign B's map job(job 2), master will find A is responsible for reduce B's map job. Eventually, A has 3 jobs and finish the whole operation. Otherwise, we will fail to find who is responsible for job 2 because B is already down.

Third, logging to fight master fault. This part is not implemented in our code for time limit but is still essential to the system. When master assigns job and update each worker's status. It should keep a log in the global file system. When master goes down, we find another available worker and converts it to master. It then reads the log in the global file system then initializes according to contents in the log and keep doing the master's work.

These three techniques will defend most of the fault cases as they did in other distributed systems.

3.7 Scalability and Iterator Invalidation Principle

Certain factors can affect the scalability of a distributed system.

First, the master makes sure the work is divided into pieces as evenly as possible. When dealing large data input, the cost will be amortized by workers. No node would become the bottle-neck to slow down the whole

system.

Second, with more workers participating, the possibility of worker fault increases. However, no relationship is established between workers which means that MapReduce system can deal with massive worker fault. For example, in Chord, if one sub-network goes down, the rings would be broken if the successor buffer fails to hold enough successors. But in MapReduce model, the workers all depend on the master instead of each other. Thus, increasing number of workers will not be a problem.

Third, as the file system is also distributed, bandwith usage when workers inquring input data can be minimized by smart arrangement of jobs. Most of the workers will just have to read data from itself or clos neighbours. This also will not be a problem.

However, the system has a special node. The work load of master may increase linearly as more workers coming in. Though master only handles management messages, lots of threads would be created in master as concurrency is heavily used in our program to increase performance. These threads may modify the worker table and job list etc. We have to use locks to maintain thread-safe. However, locks serialize concurrent programs thus the performance is compromised. Our method to solve this is to only lock when necessary. Iterator invalidation principle would help us in this process.

1) For data structure such as list or vector, add is invalidation free if no resizing happens. Deletion will not affect the elements before the deleted one.

2) For associative data structure such as map or set, only the reference of the deleted element is invalid.

These two principles will determine how to avoid unecessary locks. For example, if one thread adds a new node to worker table, we should lock against other threads which want to join or delete the worker table map. But it is not necessary to prevent a thread from modifying the existing element. For the thread removing nodes that fails to update , however, we should lock against modification.

4 Experiment and Performance

We create a program to generate random pictures as input. The program takes an integer a as input and generate a pictures with format number.placename.png. Each pictures contains random shapes generate and their sizes are different.

Then we make several copies of these pictures and store them in the nodes. We run our MapReduce application and get the output which contains location and spot frequency pairs to indicate how many times the specified shape appears in the location. The application successfully generates the output in a timely manner. Then we try to crash several workers and find that the application will still obtain the expected result but with delay. The time for the master to detect fault and reassign jobs attributes to this delay.

5 Conclusion

Reference