Networks and Distributed Systems
Programming Project

ID: 1155107
Yuan Huang

School of Computer Science
University of Birmingham

March 20, 2013

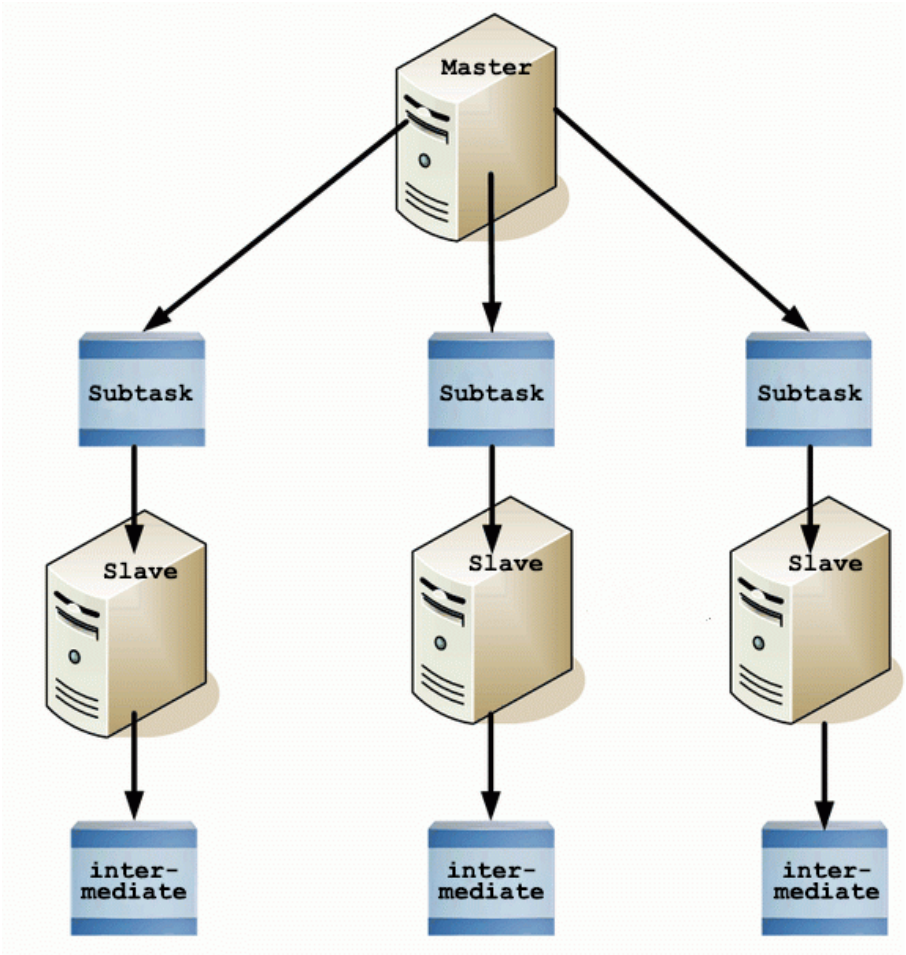
Content

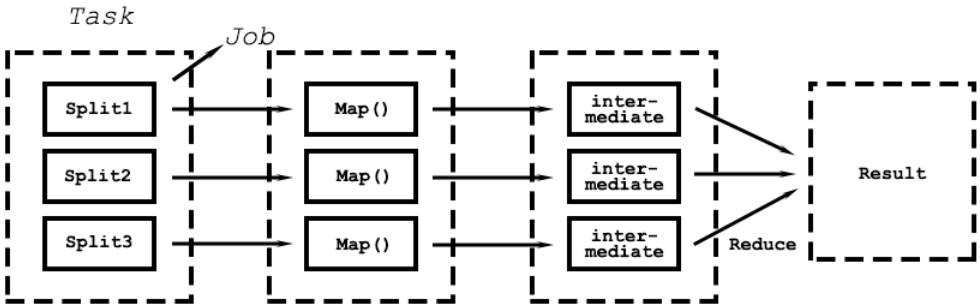
1 Introduction

Recently several organizations are building data center scale of computer systems to meet the increasing demands of high data storage and processing requirements of data-intensive and compute-intensive applications. On the industry front, companies such as Facebook and its competitors have constructed large-scale data centers using MapReduce which is a programming model for processing large data set for distributed computing to analyze user behavior and the effectiveness of ads on the site in order to optimize the user experience. This report explores a MapReduce-like distributed system implemented in Java for counting the number of occurrences of words in a given text file and programed in an effective and user-friendly way with fault tolerance design.

2 Description of the system architecture

In general, the distributed word counting system divides the computational task into smaller jobs and sends each job to a specific server, and then collects all results from distributed computing. All data processed by this so-called distributed word counting system are in the form of key/value pairs. The execution happens in two phases. In the first phase, a map function is invoked once for each input key/value pair and it can generate output key/value pairs as intermediate results in all server sides. In the second one, all the intermediate results are merged and grouped by keys in the client side. The reduce function is called once for each key with associated values and produces output values as final results.





2.1 Client

At first, the client (master) will add all the available servers (slaves) into slave list. The input task file is then read by the client, which starts distributed counting of the words in the text by first splitting the text into tasks of a given number of lines each and sending these preliminary jobs to the servers by iterating the server list. When a server has received a sub-task and hasn't responded yet, the thread corresponding to that server would wait until the server finished its job. After all the sub-tasks have been distributed and completed, the client will collect and reduce all the results into a single final result and write the output to the disk. All the data transfer is based upon TCP/IP sockets to ensure reliability using string format.

2.2 Servers

The server will wait for accepting new connections and startup a new thread corresponding to each connection. After establishing connection, server will begin to obtain data from client and transfer strings into HashMap and then send back the HashMap to the client using string format.

2.3 Failure Handling

For every sub-task, if the client sends it to the server and won't get the response before timeout, that server will be marked as error, and the client will try to re-connect and re-send the task again until several times which is usually called rollback procedure. If there is still no response, then the server will be recognized as broken status so that it will be removed from the list. It is important that if failure occurred, that specific thread won't accept new sub-task and be unavailable until it completes the job or be deleted from the list.

3 Implementation

The distributed word counting system is implemented in Java with Java Development Kit 6. The complete software consists of two packages: the master (client) and the slave (server), which are described in the following.

3.1 Master

The client starts with class Master that includes the program entrance and setup method. Inside the main method, it sets up the server list including all the slaves used for distributed word counting and the path of the task file, and then it will invoke the method

processCounting() in class Processor to create a thread pool which is an instance of Java built-in class ThreadPoolExecutor. After thread pool executor initialized, it will go through into the method invokeThreadExecutor() which is the core function of this program. All the task will be read and divided into sub-tasks using method getInputList() with a fixed line number depending on the number of slaves and then be sent to its corresponding work server. All the thread will hold a list of strings as a sub-task which is the same as the return value of method getInputList(). And then, the system will use join method of class StringUtils which is belongs to an apache library Commons Lang providing a lot of string manipulation methods to combine the string list as a single string and send it to the server in a straightforward way using TCP/IP socket by Java built-in class PrintWriter. When it is done, the client will wait for server's response and keep the result into a HashMap inside the class Status. The thread pool will arrange the same number thread as the number of slaves and after all the threads obtain the results from servers. The system will continue to read the file and spilt into sub-tasks again and loop the actions like before until the entire input file has been read. At last, the system will call the method reduce() of class Reducer to combine all the HashMap into a single one which is the final word counting result whose keys are the distinct words and values are the occurrences of each word.

3.2 Slaves

The server consists of three classes. Class Slave only has a main method, which is the entrance of the slave program. It opens a server socket with a specific user-defined port number to accept client connections. It will start a new thread class ServeOne for each connection so that all the different distributed sub-tasks won't interfere to each other. The class ServeOne inherits the Java built-in class Runnable. When the server socket accepts a new connection, the run method of class ServeOne will be automatically invoked by Java. Inside the run method, it will read the sub-task string sent by client and invoke other class Mapper to run the map and combine process and then transfer the result HashMap to an one-line string using method mapToString() and send it back to the client.

3.3 Failure Handling

The failure handling is implemented inside call() method in the class Distribute which is belong to the client package. It uses an infinite while loop to try sending the sub-task to the slave. If the sending process is finished, it will jump out of the loop and continue execution. If not, then it will try to re-connect three times until it works and gets the response result back from that slave. If it still doesn't work, the system will change that slave into a broken status so that it will automatically remove that server from the slave list and re-distribute the corresponding sub-task to another slave from the current slave list.

3.4 Log system

The log system is used inside the client package to track any possible errors and important interaction information by the slf4j library, which is a very popular and common logging framework, allows you specify logging levels (e.g. log only critical messages, log only debug messages etc.) created by Ceki Gülcü. For convenience, slf4j simplelogger has been chosen as implementation backend.

4 Evaluations

The performance of this distributed word counting system is carefully assessed by sequentially increasing the size of the input text file and testing the time consumed by the system to distribute tasks and compute the results with different file size (15 megabytes to 500 megabytes) and different numbers of slaves (from 3 to 5) to evaluate the effects of the level of distribution upon the system's performance. The test environment used is my laptop with i7 processor, 8GB memory, and 256GB SSD, and all the slaves are simulated using virtual machines created by Vmware with Centos 6 operating system binding different ports. The benchmark text files are duplicated texts from Gossip Girl written by Cecily von Ziegesar. All the recorded results are displayed in the table below.

No. of Slave	File Size	Time	File Size	Time	File Size	Time
3	15mb	3.1s	150mb	27s	500mb	84s
4	15mb	2.9s	150mb	19s	500mb	74s
5	15mb	2.7s	150mb	19s	500mb	72s

As can be seen from the result table, the execution time decreased when the number of slaves increased and the execution time grew when the size of file raised.

5 Discussions

This distributed word counting system is an implementation of counting word occurrence through a large-scale input data. As can be seen from the evaluation above, it provides a great performance by dividing a large task into subtasks, which is very similar with Google's MapReduce system. MapReduce is a library that lets you adopt a particular, stylized way of programming that's easy to split among a bunch of clusters. The basic idea is that you divide the job into two parts: Map and Reduce. Map basically takes the problem, splits it into sub-parts, and sends the sub-jobs to the different machines – so all the pieces run at the same time. Reduce takes the results from the slaves and combines them back together to get a single final answer.

However, several differences are also obviously seen between distributed word counting system and MapReduce which list as follows:

- MapReduce could split the input data with fixed size (always 16 - 64 megabytes) even if there is more than one file. However, the input data should be always a single file in the distributed word counting system.
- MapReduce allows users to write their own map and reduce functions so it could accept almost every distributed computing tasks while the distributed word counting system could only be used in word occurrence counting with its own built-in method.
- The distributed word counting system splits the input file during the process by reading text file line by line and sending a fixed number of lines to the servers each time depending on the number of available servers. MapReduce, however, preprocesses the data file by splitting it into blocks of fixed size.
- The distributed word counting system never writes intermediate results to the server local disk and always send them back to the client while MapReduce writes map results

locally and transfer all results or intermediate results using its own system called Hadoop Distributed File System (HDFS) when its server received remote procedure call (RPC).

To solve the problems and differences above, the distributed word counting system could be improved and optimized in three ways. First, the input data shouldn't be restricted as a single file and could be replaced by using a directory so that the system could read each file one by one. Next, it is necessary to make the reduce part in a distributed way as well as the map part implemented just like MapReduce in order to save master's workload and execution time. Finally, MapDB library may be a better way for this software to write all the HashMap into disk to decline the memory usage and provide crash recovery functionality.

6 References

- J. Dean and S. Ghemawat. Map-reduce: simplified data processing on large clusters. Communications of the ACM, 51:1, 107-113.

7 Appendixes

To run the program, all the machines should install Java Runtime Environment 6 or higher and add four external libraries including commons-io-2.4, commons-lang3-3.1, slf4j-api-1.7.2, and slf4j-simple-1.7.2 on the build path. To startup each server (slave), just type **java com/slave/Slave <port>** in the command line to start a socket listening connections at the port you typed. Noticed that all servers should be run first before you start the client. For the client (master), you need use the command **java com/master/Master <input> <output>** to run the program after you modified the file master.java to satisfy your needs. Make sure that the input file **<input>** exists, is a file and readable.

```
8    public static void main(String[] args) {
9        if(args.length == 2) {
10           File input = new File(args[0]);
11           if(input.exists()) {
12               long startTime = System.currentTimeMillis();
13               Processor p = new Processor();
14               p.addSlave("127.0.0.1", 6801);
15               p.addSlave("localhost", 6802);
16               p.addSlave("localhost", 6803);
17               Map<String, Integer> w = p.processCounting(input, args[1]);
18               // p.displayWordCounts(w);
19               p.displayByValue(w);
20               // p.displayByKey(w);
21               long estimatedTime = System.currentTimeMillis() - startTime;
22               System.out.print("\nThe total time for running this job is "
23                       + estimatedTime + "ms.");
24           } else {
25               System.err.println("Input file doesn't exist.");
26               System.err.println("Usage: java com/master/Master <input> <outp
27           }
```

The graph above gives a part of codes inside the source code file master.java, when you need run a task, you should modify this file first. From line 14-16, it shows a list of slaves with corresponding ip address and port number, you can add or modify slaves to fit your own slaves' setup.

The output file will be generated every time after the program execution and be placed and named as the user defined argument **<output>** when you run this software in the command line.