Distributed Systems Coursework 2: Simplified MapReduce

1 Overview

The purpose of this practical is to develop a distributed application and improve your understanding of the mechanisms behind MapReduce. You are required to implement and evaluate a simple version of MapReduce, which will operate over lab machines. The intended learning outcomes of this practical are:

- To gain insight into the fundamental problems of designing robust distributed applications
- To gain experience developing distributed applications
- To improve your understanding of MapReduce

2 Problem

Distributed computing frameworks including MapReduce[1], BOINC [5] and Condor [6] are used extensively to support the execution of data-intensive workloads in a distributed manner. You are required to develop a distributed computing application, which is conceptually similar to these frameworks. Like many distributed computing frameworks, your solution must perform a number of functions. It is required to:

- Accept a computational job
- Distribute the work to a set of nodes
- Perform the computation
- Return and verify the results

3 Part 1

In this coursework you will be writing a simple MapReduce framework. You are required to implement four stages, illustrated further in Figure 1:

- Read an input file containing a set of integers from the user. Create input files of different sizes.
- Split this work (input file) between multiple workers, which are distributed between several machines using some form of message passing.
- The nodes must do some simple computation; they must calculate the product of the set of values it has received.
- The nodes must then return the results to the master node which must then calculate the product of the newly returned values to the end user. Most of this functionality is provided by the part-solution.

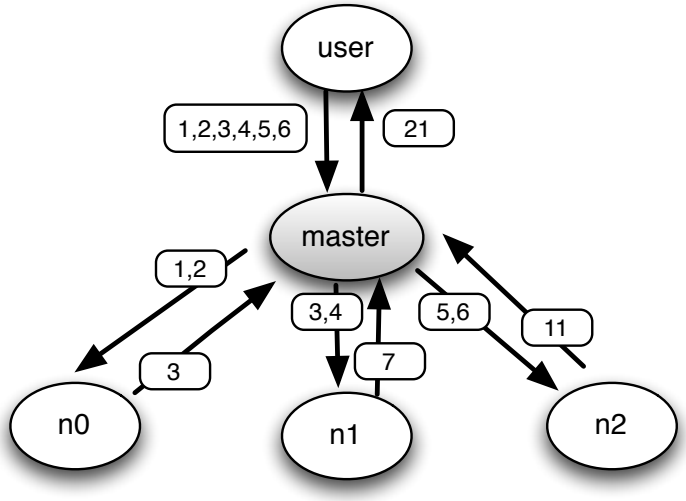

Figure 1: 1) Job is submitted to the master node 2) The master node splits up and distributes the job to the worker nodes 3) The worker nodes perform the computation (sum) and return the values to the master node 4) The master node combines the returned values and then returns the final result to the end user

4 Provided Part-Solution

Provided alongside this specification are two Java programs a Master and a Worker. Both these programs are written in Java and use ZeroMQ [4] and Google Protocol Buffers [2] to encode and transmit messages between components. The following is a description of each component:

4.1 Master

- Master.java contains the main method and creates the appropriate threads
- BootstrapServer.java listens for workers to announce themselves
- Ventilator.java distributes computation between workers
- Sink.java receives results from workers
- Computation.java is an empty class used to represent computation which must be implemented

4.2 Worker

- Worker takes two command line arguments, the local address and the address of the server and instantiates the necessary threads
- BootstrapClient.java sends this workers address to the boostrap server
- ResultsSender.java sends the result of computation to the sink
- WorkReciever.java receives work from the ventilator

Additionally there is a common package which is shared by both applications. This contains the .proto file used to generate the WireMessages.java class. Don't edit this class, rather automatically generate it using the protocol buffer compiler. Protocol buffers are a simple encoding format which can be quickly learned from reading the tutorial [3]. They use a simple domain specific language which compiles to Java. There shouldn't be any need to modify any ZeroMQ related code. The necessary jars and the Protocol Buffer compiler are provided on studres.

The provided files contain the necessary Eclipse metadata to import the project into eclipse. Additionally it is also a Maven project. If you are using Eclipse, simply build and run in the same way you would any Java project. Exporting it as a runnable Jar will be useful for going between multiple lab machines. If you are using Maven the following will build and run the project:

```
DS3$ mvn package
DS3$ cd target
target$ java -cp "*"  uk.ac.st_andrews.cs.cs4301.master.Master
target$ java -cp "*"  uk.ac.st_andrews.cs.cs4301.worker.Worker 127.0.0.1 127.0.0.1
```

5 Part 2

Once you have written a basic MapReduce implementation you are required to improve your solution to deal with errors. In real world systems there are many types of faults. One type of fault is a commission failure (e.g. processing data incorrectly and returning incorrect values). With volunteer computing systems this can be due to malice or due to hardware failure. Irrelevant of the cause, accepting invalid responses as genuine will result in inaccurate results which must be avoided. One of the mechanisms frequently employed is to duplicate the computation such that it is performed multiple times. As it is unlikely for all nodes to return invalid results it is possible to locate which processes are at fault and disregard their results. You are required to implement a mechanism to introduce node failure (e.g. nodes returning garbage) and a mechanism to compensate for this failure. This mechanism must be fully described in your report. Individual research is encouraged, algorithms taken from lectures or from further afield are equally valid.

6 Part 3

In order to obtain a grade greater than 14.5 you must implement at least one of these extensions

6.1 Extension 1: Distributed Reduce

This implementation is not true MapReduce for a number of reasons. One of these reasons is that the reduce stage is done centrally at the master. As an extension, distribute the reduce stage over the worker nodes, using the master only to coordinate where computation is sent and to return the final result.

6.2 Extension 2: Performance Evaluation

Certain types of computation experience a linear speedup when run over additional machines. Such problems are typically referred to as "embarrassingly parallel". In this extension you are to evaluate what, if any, the performance gain of distributing computation over more machines is and if there is a given input size that benefits the most from being distributed.

6.3 Extension 3: Additional Computation

Extend the types of computation that your MapReduce implementation can perform. This can be as simple as implementing additional basic mathematical options or more complex. This additionally requires that the user has a simple means to select the type of computation.

7 Extras

If you have successfully completed Parts 1,2 and 3 and still feel compelled to continue, you are encouraged to add additional features to your own satisfaction. Impressive additions are encouraged!

8 Report

You are required to submit a short (less than 1000 words) report describing your submission. Ensure to include a description and justification for the mechanisms you have implemented as part of your solution.

9 Grading and Deadline

The deadline for the submission is Friday 25 April. Please submit all source code and your corresponding report (as a pdf) to MMS. A grade no higher than 10 will be awarded for the completion of Part 1, and no higher than 14 for Part 2. In order to obtain a grade higher than 14.5 you must complete one of the extensions in Part 3.

References

[1] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[2] Google. Google protocol buffers. https://developers.google.com/protocol-buffers/.

[3] Google. Protocol buffers tutorial. https://developers.google.com/protocol-buffers/docs/javatutorial.

[4] iMatrix. Zmq the guide. http://zguide.zeromq.org/page:all.

[5] The BOINC Project. Boinc. http://boinc.berkeley.edu/.

[6] The HTCondor Project. Htcondor. http://research.cs.wisc.edu/htcondor/.