Making Hadoop MapReduce Byzantine Fault-Tolerant*

Alysson N. Bessani, Vinicius V. Cogo, Miguel Correia, Pedro Costa, Marcelo Pasin, Fabricio Silva
Universidade de Lisboa, Faculdade de Ciências, LASIGE – Lisboa, Portugal
{bessani, mpc, pasin, fabricio}@di.fc.ul.pt, vielmo@lasige.di.fc.ul.pt, psdc1978@gmail.com

Luciana Arantes, Olivier Marin, Pierre Sens, Julien Sopena
LIP6, Université de Paris 6, INRIA Rocquencourt – Paris, France
{luciana.arantes, olivier.marin, pierre.sens, julien.sopena}@lip6.fr

1. Introduction

MapReduce is a programming model and a runtime environment designed by Google for processing large data sets in its *warehouse-scale machines* (WSM) with hundreds to thousands of servers [2, 4]. MapReduce is becoming increasingly popular with the appearance of many WSMs to provide *cloud computing* services, and many applications based on this model. This popularity is also shown by the appearance of open-source implementations of the model, like Hadoop that appeared in the Apache project and is now extensively used by Yahoo and many other companies [7].

At scales of thousands of computers and hundreds of other devices like network switches, routers and power units, component failures become frequent, so fault tolerance is central in the design of the original MapReduce as also in Hadoop. The modes of failure tolerated are reasonably benign, like component crashes, and communication or file corruption. Although the availability of services based on these mechanisms is high, there is anecdotal evidence that more pernicious faults do happen and that they can cause service unavailabilities. Examples are the Google App Engine outage of June 17, 2008 and the Amazon S3 availability event of July 20, 2008.

This combination of the increasing popularity of MapReduce applications with the possibility of fault modes not tolerated by current mechanisms suggests the need to use fault tolerance mechanisms that cover a wider range of faults. A natural choice is Byzantine fault-tolerant replication, which is a current hot topic of research but that has already been shown to be efficient [5, 6]. Furthermore, there are critical applications that are being implemented using MapReduce, as financial forecasting or power system dynamics analysis. The results produced by these applications are used to take critical decisions, so it may be important to increase the certainty that they produce correct outputs. Byzantine fault-tolerant replication would allow MapReduce to produce correct outputs even if some of the nodes were arbitrarily corrupted. The main challenge is doing it at an affordable cost, as BFT replication typically requires more than triplicating the execution of the computation [5].

This abstract presents ongoing work on the design and implementation of a Byzantine fault-tolerant (BFT) Hadoop MapReduce. Hadoop was an obvious choice because it is both available for modification (it is open source) and it is being widely used. This work is being developed in the context of FTH-Grid, a cooperation project between LASIGE/FCUL and LIP6/CNRS.

*This work was partially supported by the FCT and the EGIDE through programme PESSOA (FTH-Grid project), and by the FCT through the Multi-annual and the CMU-Portugal Programmes.

2. Hadoop MapReduce

MapReduce is used for processing large data sets by parallelizing the processing in a large number of computers. Data is broken in splits that are processed in different machines. Processing is done in two phases: *map* and *reduce*. A MapReduce application is implemented in terms of two functions that correspond to these two phases. A map function processes input data expressed in terms of key-value pairs and produces an output also in the form of key-value pairs. A reduce function picks the outputs of the map functions and produces outputs. Both the initial input and the final output of a Hadoop MapReduce application are normally stored in HDFS [7], which is similar to the Google File System [3]. Dean and Ghemawat show that many applications can be implemented in a natural way using this programming model [2].

A MapReduce *job* is a unit of work that a user wants to be executed. It consists of the input data, a map function, a reduce function, and configuration information. Hadoop breaks the input data in *splits*. Each split is processed by a map task, which Hadoop prefers to run in one of the machines where the split is stored (HDFS replicates the splits automatically for fault tolerance). Map tasks write their out-

put to local disk, which is not fault-tolerant. However, if the output is lost, as when the machine crashes, the map task is simply executed again in another computer. The outputs of all map tasks are then merged and sorted, an operation called *shuffle*. After getting inputs from the shuffle, the reduce tasks process them and produce the output of the job.

The four basic components of Hadoop are: the *client*, which submits the MapReduce job; the *job tracker*, which coordinates the execution of jobs; the *task trackers*, which control the execution of map and reduce tasks in the machines that do the processing; HDFS, which stores files.

3. BFT Hadoop MapReduce

We assume that clients are always correct. The rationale is that if the client is faulty there is no point in worrying about the correctness of the job's output. Currently we also assume that the job tracker is never faulty, which is the same assumption done by Hadoop [7]. However, we are considering removing this restriction in the future by replicating also the job tracker using BFT replication. In relation to HDFS, we do not discuss here the problems due to faults that may happen in some of its components. We assume that there is a BFT HDFS, which in fact has already been presented elsewhere [1]. Task trackers are present in all computers that process data, so there are hundreds or thousands of them and we assume that they can be Byzantine, which means that they can fail in a non-fail-silent way.

The key idea of BFT Hadoop's task processing algorithm is to do majority voting for each map and reduce task. Considering that f is a higher bound on the number of faulty task trackers, the basic scheme is the following:

1. start $2f + 1$ replicas of each map task; write the output of these tasks to HDFS;
2. start $2f + 1$ replicas of each reduce task; processing in a reduce starts when it reads $f + 1$ copies of the same data produced by different map replicas for each of map task; the output of these tasks is written to HDFS.

This basic scheme is straightforward but is also inefficient because it multiplies the processing done by the system. Therefore, we use a set of improvements:

Reduction to $f+1$ replicas. The job tracker starts only $f+1$ replicas of the same task and the reduce tasks check if all of them return the same result. If a timeout elapses or some of the returned results do not match, more replicas (at most f) are started, until there are $f + 1$ matching replies.

Tentative execution. Waiting for $f+1$ matching map results before starting a reduce task can put a burden on end-to-end latency for the job completion. A better way to deal with the problem is to start executing the reduce tasks just after receiving the first copies of the required map outputs, and then, while the reduce is still running, validate the input used as the map replicas outputs are produced. If at some point it is detected that the input used is not correct, the reduce task can be restarted with the correct input.

Digest replies. We need to receive at least $f + 1$ matching outputs of maps or reduces to consider them correct. These outputs tend to be large, so it is useful to fetch the first output from some task replica and get just a digest (hash) from the others. This way, it is still possible to validate the output without generating much additional network traffic.

Reducing storage overhead. We can write the output of both map and reduce tasks to HDFS with a replication factor of 1, instead of 3 (the default value). We are already replicating the tasks, and their outputs will be written on different locations, so we do not need to replicate these outputs even more. In the normal case Byzantine faults do not occur, so these mechanisms greatly reduce the overhead introduced by the basic scheme. Specifically, without Byzantine faults, only $f + 1$ replicas are executed in task trackers, the latency is similar to the one without replication, the overhead in terms of communication is small, and the storage overhead is minimal.

4. Conclusion and Future Work

This abstract briefly presents a solution to make Hadoop MapReduce tolerant to Byzantine faults. Although most BFT algorithms in the literature require $3f + 1$ replicas of the processing, our solution needs only $f + 1$ in the normal case, in which there are no Byzantine faults.

Currently we are implementing a prototype of the system, which we will evaluate it in a realistic system to see if the actual costs match our expectations.

References

[1] A. Clement, E. Wong, L. Alvisi, M. Dahlin, and M. Marchetti. UpRight cluster services. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, Oct. 2009.

[2] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating Systems Design & Implementation*, Dec. 2004.

[3] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. *SIGOPS Operating Systems Review*, 37(5), 2003.

[4] U. Hoelzle and L. A. Barroso. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan and Claypool Publishers, 2009.

[5] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative Byzantine fault tolerance. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, Oct. 2007.

[6] G. S. Veronese, M. Correia, A. N. Bessani, and L. C. Lung. Spin one's wheels? Byzantine fault tolerance with a spinning primary. In *Proceedings of the 28th IEEE Symposium on Reliable Distributed Systems*, Sept. 2009.

[7] T. White. *Hadoop: The Definitive Guide*. O'Reilly, 2009.