On the Feasibility of Byzantine Fault-Tolerant MapReduce in Clouds-of-Clouds

Miguel Correia[1], Pedro Costa[2], Marcelo Pasin[2], Alysson Bessani[2], Fernando Ramos[2], Paulo Verissimo[2]
[1]INESC-ID/IST [2]FCUL/LaSIGE – Lisboa, Portugal
miguel.p.correia@ist.utl.pt, pcosta@lasige.di.fc.ul.pt, {pasin, bessani, fvramos, pjv}@di.fc.ul.pt

***Abstract*—MapReduce is a framework for processing large data sets largely used in cloud computing. MapReduce implementations like Hadoop can tolerate crashes and file corruptions, but there is evidence that general arbitrary faults do occur and can affect the correctness of job executions. Furthermore, many individual cloud outages have been reported, raising concerns about depending on a single cloud.**

We present a MapReduce runtime that tolerates arbitrary faults and runs in a set of clouds at a reasonable cost in terms of computation and execution time. The main challenge is to avoid sending through the internet the huge amount of data that would normally be exchanged between map and reduce tasks.

I. Introduction

MapReduce is a framework for processing large data sets composed of a programming model and a runtime environment [1]. Since first presented in 2004, MapReduce was the theme of a lot of research and become widely used in cloud computing environments. The original implementation by Google is not available, but a few open versions appeared, among which the very popular Hadoop [2]. Commercial versions have also appeared, e.g., Amazon's Elastic MapReduce.

Both the original MapReduce and Hadoop include mechanisms to recover from common faults, such as node crashes and file corruption in hard disks. Specifically, data processing tasks are monitored and restarted if they stop, and files contain checksums that are used to detect their corruption [2], [3].

However, evidence of two other failure modes suggests the need for further research on MapReduce fault tolerance.

Firstly, arbitrary faults that may affect the *correctness of the results of MapReduce* have been known to happen. A study in a large number of servers in Google datacenters for 2.5 years concluded that DRAM errors are more prevalent than previously believed, with more than 8% DIMMs affected by errors yearly, even if protected by error correcting codes (ECC) [4]. A Microsoft study of 1 million consumer PCs shown that CPU and core chipset faults are also frequent [5]. The original MapReduce fault tolerance mechanisms can not deal with such faults, often called *arbitrary or Byzantine faults* [6]. Note that, although the term 'Byzantine' often relates today to malicious faults, we only consider *arbitrary faults of accidental origin* in this work, actually remaining in the original scope of the paper that coined the term, defining the foundations for Byzantine Fault Tolerance (BFT) [7].

Secondly, cloud outages can *interrupt the execution of MapReduce jobs*. Several cases have been recently reported, including the disruption of the Amazon EC2 US East Region datacenter during almost one week in April 2011, and the disruption of the Windows Azure service for a few hours in February 2012.

To tolerate either kind of fault, we need to add some form of redundancy to the computations. Killing two birds with one stone, we plan to leverage on previous work that developed a BFT MapReduce [8] running on single clouds (see related work in Section V), extending it to a multi-cloud environment, thus tolerating both kinds of faults mentioned above.

This paper is thus about a Byzantine fault-tolerant (BFT) MapReduce runtime system for *clouds-of-clouds*. A cloud-of-clouds is an abstraction of a cloud environment that is formed by a set of clouds or datacenters connected by the internet. Unlike federated clouds, the operation of the multi-cloud environment is under control of the user, not the providers [9].

In order to give some insight on our approach, note that an application-agnostic way of adding redundancy consists in executing tasks more than once and compare their outputs. This is essentially what is done in the state machine approach [10]. Several Byzantine fault-tolerant replication algorithms based on this approach have been proposed [11], [12], [13]. By running in a cloud-of-clouds, such a BFT system can tolerate outages of some clouds and continue to run. In consequence, it becomes possible to simultaneously ensure the *integrity* and *availability* of MapReduce executions.

However, these algorithms typically require $3f + 1$ replicas to tolerate up to f faulty ones (e.g., 4 to tolerate one faulty) and are often inefficient if not executed in a local network. Moreover, and specifically in MapReduce executions, map and reduce tasks exchange data of the order of magnitude of the input, e.g., of gigabytes. Exchanging data of this size through the internet would make running MapReduce in a cloud-of-clouds entirely impractical.

The main contribution of the paper is to show that it is possible to solve this problem, essentially by exchanging only hashes of outputs between the clouds. More generically, the challenge of the work is achieving efficiency in terms of four parameters: computation (the number of replicas executed), communication (amount of data exchanged between clouds), latency (the time to run the job), and monetary cost (of the service to the user). Besides the above mentioned use of hashes, this efficiency is mainly obtained: (1) by executing one job tracker in each cloud; (2) by executing only $f + 1$ replicas of the tasks when there are no faults, and one additional

replica per faulty one; and (3) by using an overlay network to circumvent slow/faulty routes between clouds.

Note that there is a baseline cost that must be incurred when achieving the quality of protection we propose, which is significantly higher than that of native MapReduce and Hadoop. For instance, a configuration with $f = 1$ will require each task to be executed twice and three clouds, instead of one execution in one cloud. We believe this is a more than acceptable cost for applications with critical requirements with regard to integrity *and* availability. Our perspective is that the number of such applications is increasing.

II. Hadoop MapReduce

In this section we briefly present the operation of Hadoop MapReduce [2], which is similar to Google's MapReduce [1].

The MapReduce programming model computes a job in two phases (see Figure 1). Programmers specify two functions, *map* and *reduce*. The input is typically large (e.g., gigabytes) and divided in files called *splits*. In the first phase, each split is processed by the map function that generates key-value pairs. Then, these outputs are shuffled according to their keys and passed to the reduce tasks (each reduce typically gets input from all maps) that process them again. This simple idea has been shown to be useful for many different applications [1].

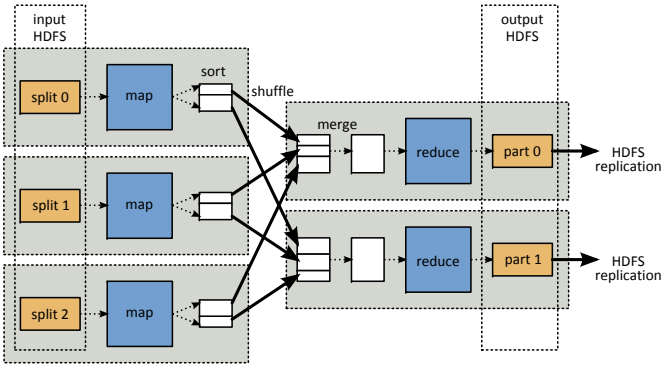

Figure 1: Computation of a job in MapReduce [2].

Both the splits and the outputs of the reduce tasks are stored in a file system. Due to the typical large size of these files, Hadoop has a specific file system for this purpose, HDFS, similar to Google's GFS [3]. HDFS stores files in blocks of 64 MB by default. HDFS contains a *name node* that manages data storage and many *data nodes*, typically one per server, which store the blocks. Blocks are usually replicated in a few data nodes for fault tolerance.

Users submit a job by providing the map and reduce functions, and the location of the splits in the HDFS (see Figure 2). The processing of a job is controlled by the *job tracker*, which is centralized. The map and reduce tasks are executed by *task trackers*, which are executed in servers (e.g., one per core). Whenever possible, a map task is executed in the server storing the split it must process (locality).

Task trackers periodically send heartbeat messages to the job tracker. The missing of heartbeat messages allows the job tracker to figure out that a task stalled or failed. Using different nodes, the job tracker runs extra, speculative, tasks for those lagging behind and restarts the failed ones. Nevertheless, this model only supports crashes, not arbitrary faults.

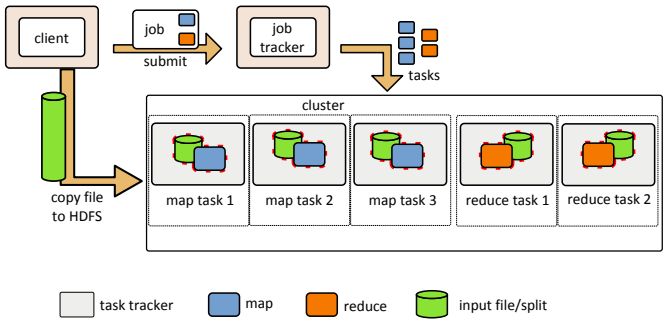

Figure 2: Submission of a job to Hadoop MapReduce.

III. BFT MapReduce for Clouds-of-Clouds

The basic idea of our BFT MapReduce is to replicate every task in a few clouds or datacenters. The challenge is doing it efficiently.

A. System Model

The system runs in a set of *clouds* or cloud-of-clouds and is composed by a set of processes. The *clients* request job executions and the *task trackers* execute map/reduce tasks. There is one *job tracker* in every cloud. There is an HDFS instance in every cloud but we do not enter in details as it is used only to store the job's initial inputs and final outputs.

We consider a hybrid failure model, assuming that: (i) there is a bound on the maximum number of *clouds* (and consequently of job trackers) that can fail arbitrarily, t out of $2t + 1$ clouds; (ii) there is no limit on the number of tasks that fail arbitrarily, with the exception of a bound f on the number of faulty task replicas that *return the same output*. Processes/components that fail are said to be *faulty* and those that do not, are said *correct*.

Note the reasonable weakness of our arbitrary failure assumptions, i.e., we just impose that no more than f replicas of a faulty task T can return the same wrong output o. Logically, f and t obey the relation $f \geq t$ as one faulty cloud can corrupt several map/reduce replicas.

Furthermore, we assume that the *client* that submits the job does not fail, as it is not part of MapReduce. We also do not care about HDFS failures, since a Byzantine fault-tolerant HDFS exists [12].

The system is asynchronous, i.e., we make no assumptions about bounds on processing and communication delays for our mechanisms, although the original Hadoop mechanisms make assumptions about such times for termination.

B. Basic approach

In the beginning, the splits are stored in HDFS. The difference in relation to the original MapReduce is that we assume that there is a copy of each split in every cloud. Moving data

to a different cloud is expensive, but MapReduce is often used to process data in the servers where it is already stored (e.g., indices of web sites are computed in the servers where the data returned by the web crawler is stored). The difference here is that the data would be stored in a few clouds, instead of a single one, but this is an obvious requirement when the objective is availability despite cloud outages.

The basic approach consists in creating $2f + 1$ replicas of each map and reduce task, spread through the $2t + 1$ clouds (i.e., between $\left\lfloor \frac{2f+1}{2t+1} \right\rfloor$ and $\left\lceil \frac{2f+1}{2t+1} \right\rceil$ replicas per cloud). The job tracker schedules the execution of tasks in task trackers in all the clouds.

In the original MapReduce, each reduce task fetches the output of every map task. In the basic BFT MapReduce [8], each reduce task *replica* fetches the output of every map task *replica*. Whenever a reduce replica fetches the outputs $o_1, ..., o_{2f+1}$ of replicas of a map task $r_1^i, ..., r_{2f+1}^i$, it compares these outputs and picks the output provided by $f + 1$ replicas (no more than f faulty replicas produce the same output). If no $f + 1$ outputs match, more replicas of the map task r^i are executed ($r_{2f+2}^i, ...$). Each reduce replica stores its output in HDFS and it is up to the client to compare the outputs and pick the correct outputs. These two votes – of map and reduce task outputs – prevent faulty replicas or faulty clouds from corrupting the result of the job.

This basic approach has the following problems, when directly applied to a cloud-of-clouds:

- communication cost: each reduce task replica fetches the output of every map task replica, most of these fetches being made between different clouds through the internet;
- computation cost: every task is executed $2f + 1$ times, a minimum of 3 times, even if there are no faults;
- centralized job execution control: the job tracker is in one of the clouds so it has to control task trackers in other clouds remotely and use large timeouts in the failure detection mechanism. It is also a single point of failure.

C. BFT MapReduce scheme

Our proposal for BFT MapReduce for clouds-of-clouds can be thought of as the basic approach enhanced with four mechanisms: distributed job tracker, deferred execution, digest communication, and overlaid communication. Next we explain each of these mechanisms.

a) Distributed job tracker: In the original MapReduce, there is a single job tracker that controls the execution of tasks in the available task trackers, whereas in our BFT MapReduce there is one job tracker per cloud that controls the execution of tasks in that cloud, but not in the other clouds (namely, heartbeats are exchanged only inside clouds, not between clouds).

Reduce tasks obtain information about the map tasks that finished processing, from the job tracker. In the BFT version, each job tracker periodically sends the other tracker replicas, information about finished map tasks. Therefore, reduce tasks can obtain that information from their local job tracker. Faulty job trackers or faulty clouds can stop collaborating in the execution of the job or return wrong information about the status of the task replicas they execute. Nevertheless, the existence of redundancy and the voting scheme allows job execution to progress and finish, in the presence of up to t faulty clouds.

b) Deferred execution: This mechanism consists in executing only $f+1$ replicas of every task in $t+1$ clouds. In fact, arbitrary faults tend to be rare so there is no need to execute $2f + 1$ replicas of every task. The job trackers in the other t clouds still collaborate in the execution of the job but run no tasks. We call these t clouds *standby clouds* because they do not perform task computations in the absence of faults.

Typically there will be $f + 1$ matching outputs for every task so this degree of replication is enough. If there are less than $f + 1$ matching outputs, more replicas of the same task (map or reduce) are executed until a match is found. First the standby clouds are used to run these extra tasks, but if more are needed then replicas start to be executed in all clouds until a match is found.

c) Digest communication: In MapReduce, each reduce task fetches the output of each map task. In our BFT MapReduce tasks are replicated $f + 1$ times (with deferred execution and no faults), therefore the number of fetches is multiplied by $(f+1)^2$, most of them done between different clouds. The digest communication mechanism plays the extremely important role of reducing the overhead of this communication. The mechanism consists in dividing fetches in two cases:

- intra-cloud fetch: a fetch of the output of a map task by a reduce task of the same cloud is done normally, i.e., the output is moved from the map to the reduce task;
- inter-cloud fetch: if the map and reduce tasks are in different clouds, only a cryptographic hash of the output is moved (e.g., an SHA-1 hash).

This mechanism replaces the transmission of a possibly-large output, by the transmission of a small, fixed-size, hash (e.g., 20 bytes in the case of SHA-1), thus drastically reducing the communication through the internet. Instead of comparing the outputs of the map replicas, a reduce task calculates the hash of the local outputs and compares all the hashes. Again, since arbitrary faults are rare, the values obtained from local replicas are usually correct, making this a viable mechanism.

d) Overlaid communication: The communication between clouds can be delayed by temporary failures of links, router ports, switches, etc. Layer-3 routing is able to recover from these faults but can take some time, depending on the extension of the fault. A solution to this problem consists in doing application-layer routing, i.e., in constructing an overlay network composed by a set of *gateways* [14]. In our case, each cloud (or datacenter) contains a gateway. A reduce task sends a request for fetching an output outside its cloud by sending it to its local gateway (i.e., the gateway in its cloud). The latter sends it to the other gateways, which deliver it in turn to their local map tasks. The map tasks reply, also by sending to their local gateway.

This mechanism allows circumventing slow or faulty routes between gateways by sending the message through an alter-

native application-layer route. The gateways monitor the state of the communication with other gateways. If, for instance, gateway g_1 is unable to communicate with g_2, it sends the message to g_3 requesting it to be delivered to g_2 (an application-layer route). Several routing schemes can be implemented by the gateways, exploiting redundancy in time (i.e., doing retransmissions) and in space (sending by more than one overlay channel). Research about the best approach is left as future work.

IV. Evaluation

This section evaluates our BFT MapReduce for clouds-of-clouds. We aim to answer two questions:

- What is the computation cost in terms of number of tasks executed and how does it compare to alternative approaches?
- What is the time needed to run a given job (makespan) and how does it compare to alternative approaches?

A. Number of tasks executed

This section evaluates the computation cost analytically, by calculating the number of tasks executed. Besides the BFT MapReduce, we consider three other approaches. The first is the original MapReduce, which does not tolerate Byzantine faults, but executes every task only once when there are no faults (*Original MR*). The second consists in using state machine replication (*SMR MR*) to run MapReduce in a cloud-of-clouds. Byzantine fault-tolerant SMR requires $3f+1$ replicas, although only $2f+1$ have to do the computation [15]. The third approach consists in using a simple result comparison scheme (*Result-cmp MR*): $f+1$ full executions of the job are done; the results are compared; if they differ, more job executions are done until there are $f+1$ equal results, which give the correct result. This scheme executes every job $f+1$ times when there are no faults and one more time per faulty result. Finally, our own solution executes every task $f+1$ times when there are no faults, and one task more per fault.

Figure 3 compares the number of map replicas executed when there are no faults and $t = f = 1$ for different numbers of input splits. We consider only map tasks because the number of reduce tasks is constant, whereas the number of map tasks is equal to the number of splits in the original MapReduce. Figure 3 shows that our approach and Result-cmp MR have twice the cost of the original MapReduce, whereas SMR MR is three times more expensive.

The graph would change if there were faults. For each faulty task out, BFT MapReduce would execute an additional task replica. This would be unnoticeable in the graph if the number of faults would be small. On the contrary, another job would be executed for each fault that affected a job execution in Result-cmp MR. If there was a single fault, the line for Result-cmp MR in the graph would fall above the SMR MR line. The latter would not be affected by a fault, but it can only tolerate f faulty replicas, not an unlimited number as BFT MR and Result-cmp MR.

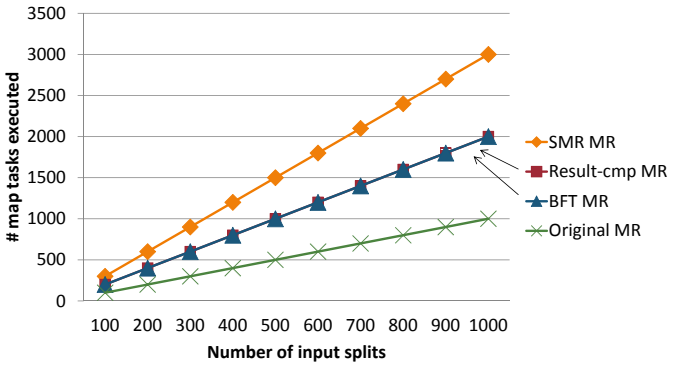

Figure 3: Number of map tasks executed vs. number of splits (no faults, $t = f = 1$).

B. Makespan

We do not have an operational prototype of our BFT MapReduce yet, so we evaluate the makespan analytically. However, we have produced some experimental data to validate the estimated results.

We derived a formula to assess the time to execute both the original MapReduce and the BFT MapReduce analytically. Then, we validated this formula by comparing the time values it produced, with executions of the applications of Hadoop's Gridmix2 benchmark.[1] We obtained a deviation below 15% between the real values measured and those given by the formula. These experiments were made both with the original Hadoop MapReduce and with a previous BFT Hadoop MapReduce [8] , which, albeit with some differences from the BFT MapReduce presented in this paper (see Section V) is close enough to provide useful data from a controlled experiment.

The rationale for the formula is that the time to execute a MapReduce job has three main components: the time to execute the map tasks, which depends on the number of map tasks that can be executed in parallel; the time to do the shuffle, i.e., to send the map task outputs to the reduce tasks (Ts); and the time to execute the reduce tasks, which depends on the number of reduce tasks executed in parallel. The formula that gives the makespan for a job (Tj) when there are no faults is the following:

$$Tj = \alpha \left\lceil \frac{Nm}{Pm \times N \times Nc} \right\rceil Tm + Ts + \alpha \left\lceil \frac{Nr}{Pr \times N \times Nc} \right\rceil Tr \quad (1)$$

where N is the number of task trackers per cloud, Nc the number of clouds not in standby (1 for the original MapReduce), Pm the number of map tasks that can be executed in parallel per task tracker, Pr the number of reduce tasks that can be executed in parallel per task tracker, Tm is the average time to execute a map task (which depends on the job), Tr is the average time to execute a reduce task (also dependent of the job), Nm is the number of map tasks (or input splits) not counting replicas, Nr is the number of reduce tasks not

[1] http://hadoop.apache.org/mapreduce/docs/current/gridmix.html

counting replicas, and α is the number of replicas (1 for the original MapReduce, $f+1$ for the BFT MapReduce). Notice that Tr and Tm are independent from the MapReduce version executed, but Ts is higher with the BFT MapReduce than with the original one.

This formula deserves a further comment. When Hadoop starts a job it launches not only Pm map tasks but also Pr reduce tasks. However, these Pr reduces are essentially idle until the map tasks finish. Therefore, the parameter Tr is in fact the average of the reduce task execution time not counting that idle time.

Figure 4 depicts the variation of the makespan (Tj) with the number of maps and reduces executed in parallel ($Pm \times N \times Nc = Pr \times N \times Nc$) for the Gridmix *combiner* application, which essentially counts occurrences of words in the split files. The figure presents both estimated (estim) and real executions (real). The parameters used in the estimations were obtained from the real execution of the original MapReduce: $Tm = 16s$, $Tr = 12s$, $Nm = 400$, $Nr = 100$. The input splits had 64 MB. In both cases we had $f = t = 1$. For the BFT MapReduce, $Nc = t + 1 = 2$. In the estimations we consider that $Ts = 0$ (later we discuss the impact of this parameter).

The experimental values were measured with the original MapReduce and the above-mentioned previously existing BFT MapReduce. The differences of the latter system to our BFT MapReduce were minimized in the experiment: it was executed in a LAN, so we are considering $Ts = 0$ in the estimates, and it also runs $f + 1$ copies of each task with no faults, therefore the values are comparable. The figure shows that they are indeed comparable: the estimated and real makespan for both the original and BFT MapReduce executions are similar.

The figure allows us to conclude that the BFT MapReduce takes approximately twice the time to run in comparison to the original Hadoop with the same degree of parallelism. It also shows that the makespan gets lower with more parallelism and the times become closer.

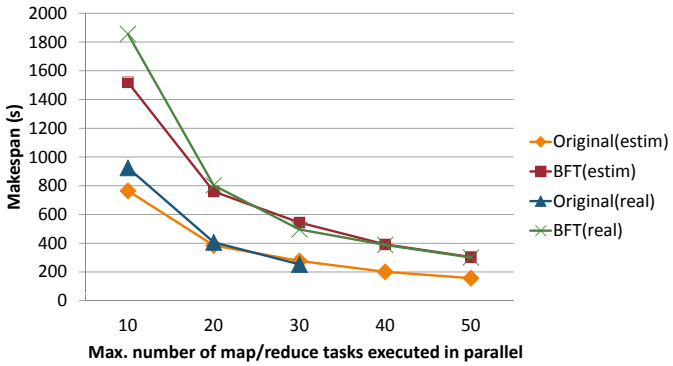

FIGURE 4: Makespan as a function of the degree of parallelism (no faults, $t = f = 1$).

Figure 5 shows only estimates, obtained using Formula 1. It shows the makespan varying the number of splits, thus also Nm, with $Pm \times N = Pr \times N = 20$ and $Nr = 100$. The figure shows that the makespan grows linearly with the number of splits, which was to be expected as the size of the input is proportional to this number (every split had 64 MB).

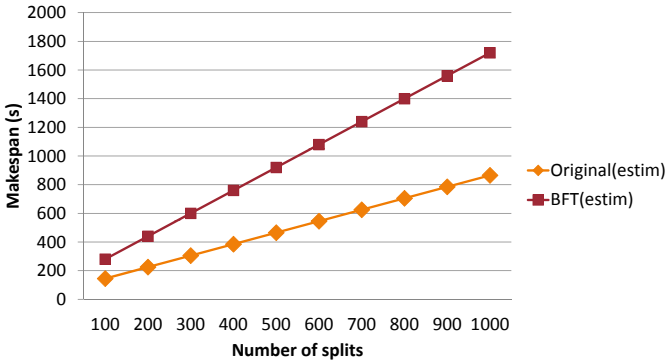

FIGURE 5: Makespan varying the number of splits (no faults, $t = f = 1$).

Let us now discuss the shuffle time, Ts. An important component of this time is the latency in the internet. In 2010 we measured the average communication delay between 20 pairs of nodes in different countries in PlanetLab, both in North America and Europe. We obtained average delays (half round-trip times) between 27.3s (France-Italy) and 94.57s (Vancouver-Pennsylvania) [13]. These measurements were made with small messages. One of the key aspects of our BFT MapReduce is that the messages exchanged between clouds contain only hashes, not complete outputs, therefore they are also small and we expect delays in this range. These values are almost insignificant when compared with the time to execute a complete job as observed, e.g., in Figure 4.

A second factor that impacts Ts is contention: in the original Hadoop there are $Nm \times Nr$ map output fetches, but in the BFT Hadoop there are $(f+1) \times Nm \times (f+1) \times Nr$. However, these fetches are not simultaneous. Consider for simplicity that all reduces start fetching map outputs whenever the maps finish. If there are many more map tasks than task trackers to execute them (which is the normal case) and all map tasks take the same time to run, the fetches will be performed $\left\lceil \frac{Nm}{Pm \times N} \right\rceil$ times, thus dividing the contention along time.

This discussion does not allow us to estimate the value of Ts when the BFT MapReduce runs in a cloud-of-clouds, with clouds interconnected by a wide-area network. However, it leads us to the conclusion that the three mechanisms used have a beneficial impact: deferred execution reduces the contention (a factor of $(f+1)^2$ messages instead of $(2f+1)^2$), digest communication reduces the data sent, and overlaid communication reduces the data sent and allows quicker recovery from network faults in the internet.

V. Related Work

There has been much research on MapReduce since the paper that originally presented it in 2004 [1]. For instance, there has been significant work on running MapReduce efficiently in different environments: multi-core/multiprocessor

systems [16], heterogeneous environments [17], high-latency eventual-consistent environments [18], and others. These works show the popularity of MapReduce, yet from the fault tolerance point of view, they do not advance the original platform.

Algorithms to tolerate Byzantine or arbitrary faults were first introduced some 30 years ago [7]. State machine replication is a generic solution to make a service crash or Byzantine fault-tolerant [10]. Since the practicality of implementing efficient Byzantine fault-tolerant replication was demonstrated in [11], many other algorithms appeared [12], [13], [15], [19]. Some of these algorithms were specifically designed with wide-area networks in mind [13], [19]. However, as already argued, state machine replication is not an efficient solution to implement BFT MapReduce.

The basic idea of executing each task more than once to tolerate arbitrary faults has been proposed in the context of volunteer computing to tolerate malicious volunteers, that return false results of the tasks they were supposed to execute [20]. That work, however, considered bag-of-tasks applications, which are simpler than MapReduce jobs. Furthermore that work focused mostly on scheduling the workers in a way that no more than a number of false results are obtained.

The work presented in this paper builds on previous work of the same authors on BFT MapReduce [8]. That first BFT MapReduce runtime was, however, designed to run in a single cloud / datacenter, not in clouds-of-clouds. Therefore, it was not able to tolerate cloud outages. In theory it would be possible to execute it in several clouds, but it would be extremely inefficient because it would exchange huge amounts of data between maps and reduces over the internet. Furthermore, that first system used a single job tracker, which was a single point of failure and, again, would cause a poor performance in a cloud-of-clouds.

VI. Conclusion

We presented a Byzantine fault-tolerant MapReduce for clouds-of-clouds. The motivation is twofold: to tolerate arbitrary faults that may corrupt the result of MapReduce jobs, and to tolerate outages and other severe faults in clouds. The main challenge of our design is achieving efficiency in terms of computation, communication, latency, and monetary cost, vis-a-vis the nature of the MapReduce programming model, the need to replicate tasks and the limited bandwidth and high latency of the wide-area network that interconnects the clouds. Our solution achieves an interesting level of efficiency by exploiting four mechanisms: distributed job tracker, deferred execution, digest communication, and overlaid communication.

Acknowledgment

This work was partially supported by the European Commission FP7 through project ICT-257243 (TClouds), and by the Fundação para a Ciência e a Tecnologia through project PTDC/EIA-EIA/115211/2009 (RC-Clouds), the Multi-annual Program (LaSIGE), and project PEst-OE/EEI/LA0021/2011 (INESC-ID).

References

[1] 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.

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

[3] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, 2003, pp. 29–43.

[4] B. Schroeder, E. Pinheiro, and W.-D. Weber, "DRAM errors in the wild: a large-scale field study," in *Proceedings of the 11th International Joint Conference on Measurement and Modeling of Computer Systems*, 2009, pp. 193–204.

[5] E. B. Nightingale, J. R. Douceur, and V. Orgovan, "Cycles, cells and platters: an empirical analysisof hardware failures on a million consumer PCs," in *Proceedings of the EuroSys 2011 Conference*, 2011, pp. 343–356.

[6] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, Mar. 2004.

[7] L. Lamport, R. Shostak, and M. Pease, "The Byzantine generals problem," *ACM Transactions on Programing Languages and Systems*, vol. 4, no. 3, pp. 382–401, Jul. 1982.

[8] P. Costa, M. Pasin, A. Bessani, and M. Correia, "Byzantine fault-tolerant MapReduce: Faults are not just crashes," in *Proceedings of the 3rd IEEE International Conference on Cloud Computing Technology and Science*, 2011, pp. 32–39.

[9] P. Verissimo, A. Bessani, and M. Pasin, "The TCLOUDS architecture: Open and resilient cloud-of-clouds computing," in *Proceedings of the 2nd International Workshop on Dependability of Clouds, Data Centers and Virtual Machine Technology*, Jun. 2012.

[10] F. B. Schneider, "Implementing fault-tolerant service using the state machine aproach: A tutorial," *ACM Computing Surveys*, vol. 22, no. 4, pp. 299–319, Dec. 1990.

[11] M. Castro and B. Liskov, "Practical Byzantine fault-tolerance and proactive recovery," *ACM Transactions Computer Systems*, vol. 20, no. 4, pp. 398–461, Nov. 2002.

[12] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riché, "UpRight cluster services," in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, Oct. 2009.

[13] G. S. Veronese, M. Correia, A. N. Bessani, and L. C. Lung, "EBAWA: Efficient Byzantine agreement for wide-area networks," in *Proceedings of the IEEE 12th International Symposium on High-Assurance Systems Engineering*, Nov. 2010, pp. 10–19.

[14] D. Andersen, H. Balakrishnan, F. Kaashoek, and R. Morris, "Resilient overlay networks," in *Proc. of the 18th ACM Symposium on Operating Systems Principles*, 2001, pp. 131–145.

[15] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin, "Separating agreement form execution for Byzantine fault tolerant services," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, Oct. 2003, pp. 253–267.

[16] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating MapReduce for multi-core and multiprocessor systems," in *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, 2007, pp. 13–24.

[17] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, "Improving MapReduce performance in heterogeneous environments," in *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, 2008, pp. 29–42.

[18] T. Gunarathne, T.-L. Wu, J. Qiu, and G. Fox, "MapReduce in the clouds for science," in *Proceedings of the 2nd IEEE International Conference on Cloud Computing Technology and Science*, 2010, pp. 565–572.

[19] Y. Amir, C. Danilov, D. Dolev, J. Kirsch, J. Lane, C. Nita-Rotaru, J. Olsen, and D. Zage, "Scaling Byzantine fault-tolerant replication to wide area networks," in *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks*, jun 2006.

[20] L. F. G. Sarmenta, "Sabotage-tolerance mechanisms for volunteer computing systems," *Future Generation Computer Systems*, vol. 18, pp. 561–572, Mar. 2002.