On the Design of Resilient Multicloud MapReduce

Pedro A. R. S. Costa*, Fernando M. V. Ramos*, Miguel Correia†
LaSIGE, Faculdade de Ciências, Universidade de Lisboa – Portugal*
INESC-ID, Instituto Superior Técnico, Universidade de Lisboa – Portugal†
palcosta@fc.ul.pt, fvramos@ciencias.ulisboa.pt, miguel.p.correia@tecnico.ulisboa.pt

***Abstract*—MapReduce is a popular distributed data-processing system for analyzing big data in cloud environments. This platform is often used for critical data processing, e.g., in the context of scientific or financial simulation. Unfortunately, there is accumulating evidence of severe problems – including arbitrary faults and cloud outages – affecting the services that run atop cloud services.**

Faced with this challenge, we have recently explored *multicloud* solutions to increase the resilience and availability of MapReduce. Based on this experience, we present system design guidelines that allow to scale out MapReduce computation to multiple clouds in order to tolerate arbitrary and malicious faults, as well as cloud outages. Crucially, the techniques we introduce have reasonable cost and do not require changes to MapReduce or to the users' code, enabling immediate deployment.

***Index Terms*—Hadoop, MapReduce, Fault-tolerance, Multicloud**

I. Introduction

Cloud computing has emerged as the paradigm for outsourcing computation. Cloud service providers have been building massive data centers that are distributed over several geographical regions to efficiently meet the demand for this service. These data centers typically contain tens of thousands of commodity servers and use virtualization technology to do provisioning of computing resources. Clouds are starting to be used together [1] forming *multiclouds*. When the combination of clouds is created by the users inconspicuously to the cloud providers, such multiclouds can be called *clouds-of-clouds* [2]. The purposes of using several clouds vary, but common goals are increasing performance and reducing costs.

Dependability problems in cloud services can cause great losses to its users and are becoming increasingly common. Hardware components are prone to soft and hard failures that reduce their reliability and the availability of the cloud service, with impact on the software running atop. Studies made at Google and Microsoft concluded that errors in the DRAM, chipset, and CPU of commodity servers are more prevalent than previously believed [3], [4]. Therefore, *fault tolerance* in cloud computing platforms and applications is a crucial issue to the users, not to mention the cloud providers themselves.

Cloud computing has enabled computation of massive volumes of data that traditional database and software techniques had difficulty in processing in acceptable time [5]. One of the most popular distributed data-processing systems for analyzing *big data* in cloud environments is Hadoop MapReduce [6], an open-source platform based on Google's MapReduce paradigm [7]. The popularity of this framework made the MapReduce model prevalently used for critical applications such as medical research and finance, where outputting wrong results and service unavailability may be unacceptable. Unfortunately, Hadoop does not deal with arbitrary and malicious faults and does not scale the computation out to multiple clouds to deal with availability issues.

In this article, we give an overview of our recent research on scaling out Hadoop to multiple clouds for tolerating arbitrary faults, malicious faults, and cloud outages (unavailability of entire data centers). This is in contrast to previous work on multi-cloud MapReduce (e.g., G-Hadoop [8]) that has not considered resilience, having focused exclusively on scalability of computation. The design guidelines we propose include two additional goals to foster adoption: the overhead should be acceptable, and no changes to Hadoop nor to the user's code should be required.

To address these challenges, the design we propose for resilient MapReduce systems is based on three core ideas. The first consists in performing replication of the processing in a set of clouds. Using replication may be considered expensive, but cloud outages are becoming so common [9], [10] that even cloud providers are exploring this approach (Amazon recently launched the Cross-Region Replication service [11]). Importantly, our solution *minimizes the replication overhead* (Section III). The second idea is to leverage the diversity provided by a multicloud environment in the design of *context-based scheduling schemes* that distribute the processing across clouds in such a way that performance is improved (Section IV). Thirdly, the solution should include fine-grained replication (at the task level) to achieve quick recovery in case of a fault. This can be achieved without modifying the Hadoop source code by means of a new abstraction we propose: the *logical job* (Section V).

The core techniques introduced in our design emanate from our experience in building two resilient multicloud MapReduce solutions – Medusa [12][1] and Chrysaor [13][2] – and one earlier solution that performs replication in a single cloud [14]. We evaluated our solutions in a real testbed, considering several MapReduce applications, to assess the performance in different scenarios (Section VI). The main result is that, by applying the proposed techniques it is possible to scale out Hadoop MapReduce to multicloud environments in order to tolerate the above-mentioned classes of faults at reasonable costs, while requiring minimal modifications to the users' jobs,

[1]Code available at https://bitbucket.org/pcosta_pt/medusa
[2]Code available at https://bitbucket.org/pcosta_pt/chrysaor

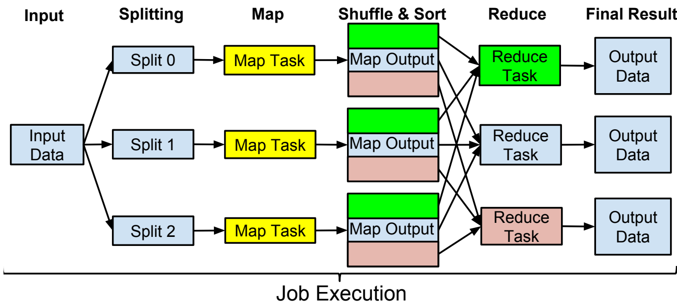

Fig. 1: Execution of a MapReduce job

and in a way that is compatible with any Hadoop MapReduce version.

II. Background

MapReduce was originally designed by Google for calculating web search indexes and running other large-scale data processing jobs [7]. Hadoop MapReduce is an open implementation that appeared a few years later and that is currently the most adopted [6].

The term MapReduce denominates both a programming model and the corresponding runtime environment. As the name indicates, MapReduce involves two functions: *map* and *reduce*. The unit of execution is the *job*, which is typically broken in one phase that executes *map* tasks and another that executes *reduce* tasks (each task runs the map or reduce functions once). Figure 1 shows a generic example of the execution of a job. The input data is split into files called *splits*. When a job starts running, each split is processed by the map function in a map task (map phase). Then, the result of the tasks are partitioned, transferred and sorted (shuffle&sort phase). In the end, the reduce tasks process the partitioned data using the reduce function (reduce phase). This simple model can express many real world applications [7].

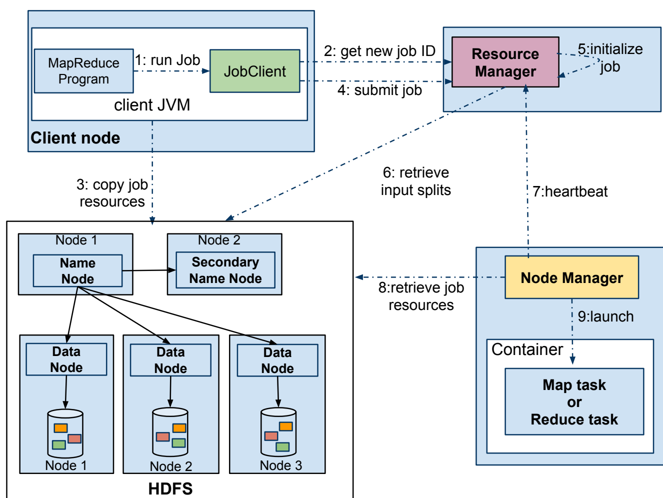

Fig. 2: Architecture of Hadoop MapReduce

The main components of Hadoop are the Hadoop MapReduce and Hadoop Distributed File System (HDFS) (see Figure 2). In Hadoop, MapReduce jobs are submitted to and managed by a central component called *resource manager*. The resource manager assigns map and reduce tasks to *node managers*, monitors these nodes, and tracks the progress of the job execution. The node manager is responsible for managing containers where tasks run. Although the figure shows a single node manager, typically there are many of those, as they are the components that do the (large-scale) data processing. A Hadoop MapReduce runtime works in a single data center. HDFS is the default file system for Hadoop MapReduce. It stores files broken into blocks that are replicated in different servers (*data nodes*) for fault tolerance. HDFS can handle many servers for scalability.

Hadoop was designed to be fault-tolerant as, with thousands of devices (computers, network switches and routers, power units), component failures are necessarily frequent. Hadoop tolerates faults using two techniques: *(i)* monitoring and restarting tasks when servers, node managers or the tasks crash; and *(ii)* adding checksums to the files in HDFS to detect data corruption in disks. However, these mechanisms only work in a single cloud, cannot deal with cloud outages, and only tolerate crash faults – not arbitrary or malicious errors.

In this paper we consider that MapReduce tasks, both map and reduce, can suffer arbitrary faults, often called *Byzantine faults*. These tasks may for instance stop or produce wrong results. To deal with these faults we execute two or more replicas of each task. We assume that there are limits on the number of faulty replicas and clouds (including resource managers), and that there is a proxy that does not fail (details next).

In the following sections, we describe the three key techniques we propose as guidelines for the design of multicloud resilient Hadoop MapReduce frameworks.

III. Just enough replication

Replication is a common strategy to ensure the integrity and availability of distributed services in which individual components may fail due to crash faults or arbitrary faults.

A. Replication for MapReduce

There are many algorithms to tolerate Byzantine faults in the literature, but only one that does so in the context of MapReduce [14]. That framework is a modified Hadoop MapReduce that essentially replicates map and reduce tasks (i.e., runs several copies of each), then compares the results obtained by replicas to detect Byzantine faults. The main challenge of this solution is the efficiency of replication. This is achieved by requiring only $f+1$ replicated tasks to tolerate f faults in case there are no faults, instead of the $3f+1$ replicas the typical Byzantine fault-tolerant state machine replication algorithms would require [15]. However, this solution works only in a single data center and as such does not tolerate cloud outages.

Figure 3 presents generically our approach of replicating MapReduce in a set of clouds, i.e., in a multicloud. The proxy is the central component to which the client submits a MapReduce job (note that this is transparent to the client – she does not need to be aware of the presence of a proxy).

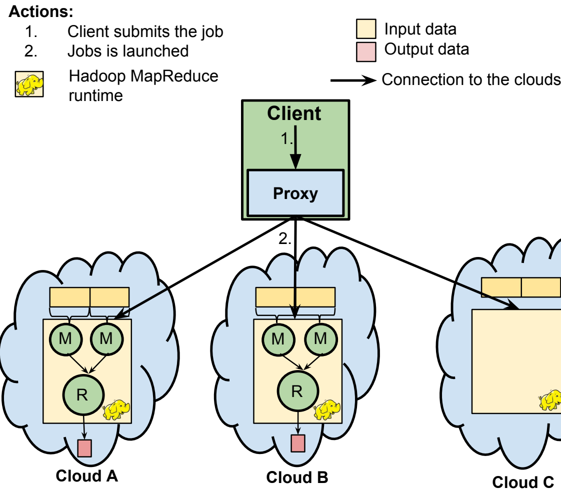

Fig. 3: Proxy executing a job in two clouds without faults ($f = 1$)

The proxy is an intermediate component between the user and the Hadoop resource managers installed in different clouds. In this example, each cloud contains one Hadoop runtime, and the proxy interacts with the several runtimes by sending them Hadoop commands (e.g., instructing to start a job) over a secure channel (SSH).

Consider that f is the maximum number of task replicas and clouds that may fail (we normally consider these two thresholds separately, but we are simplifying here). When a job is submitted by the client, our solution involves executing $f+1$ replicas, one per cloud. In the figure, we consider $f = 1$, so the proxy selects two clouds – A and B – to execute the replicas. During the execution, each task replica will produce an output and the respective digest (a collision-resistant hash, calculated, e.g., using SHA-256). The digests will be compared to check if the result is correct: if there are $f+1$ equal outputs then that output is correct as at most f replicas may produce wrong results. Otherwise, a new replica is executed. If a cloud stops responding a new one is selected (Cloud C in the figure).

We have developed two solutions that follow the generic scheme of Figure 3: Medusa and Chrysaor. Their main difference is that they work at different levels of granularity: Medusa deals with faults at the job level, whilst Chrysaor deals with faults at the task level. Sections III-B and III-C describe these solutions.

B. Replication in Medusa

Medusa replicates *full jobs* and compares only their final outputs, more precisely, their digests. If it obtains $f + 1$ identical digests, then the outputs are equal, as digests are collision-resistant (no two different outputs can produce the same hash). Otherwise, the proxy cannot identify which of the replicas is faulty and it will react accordingly; it only knows that there is a disagreement on the result.

Medusa can deal with three faulty scenarios: *(i)* with accidental faults, *(ii)* with malicious faults, and *(iii)* with cloud outages. Initially, Medusa will launch $f+1$ replicas of the job in distinct MapReduce runtimes, running in different clouds. When these jobs finish executing, Medusa will validate the computation by comparing the digests of their outputs. Medusa deals with accidental faults by re-executing the same faulty job in the same clouds until it obtains equal results. The rationale for re-executing in the same clouds is that accidental faults are inherently intermittent, so it is to expect they will eventually no longer affect the same job. This is in contrast with malicious faults or cloud outages, that require re-executions in an extra cloud. In the former case, because one of the clouds cannot be trusted. In the latter, due to one of the clouds being no longer available. In any of these cases the framework re-executes the faulty job in another cloud until it obtains $f + 1$ equal results. In the malicious case, if the re-execution ends correctly it is possible not only to validate the results but also to find which cloud is compromised. The execution aborts if no final result is obtained and no more clouds are available.

C. Replication in Chrysaor

Unlike Medusa, Chrysaor replicates *tasks*, not jobs. All map and reduce tasks produce a digest of their output and the proxy compares the digests of every set of replicas. This allows to identify which tasks have produced different results and re-execute them immediately after they finish, instead of having to wait for the end of the job and to re-execute it fully (as in Medusa). Faults in the map phase have to be dealt slightly differently from faults in the reduce phase (this will be made more clear in Section V).

Similarly to Medusa, when Chrysaor is in the presence of a fault, it cannot identify which of the replica(s) is (or are) faulty. When dealing with accidental faults, Chrysaor has the ability to re-execute the *task* for which there was no $f + 1$ identical digests in the same clouds, until it obtains $f + 1$ equal results.

When dealing with malicious faults or cloud outages, it is necessary to execute the tasks in an extra cloud, for the same reasons as above. If the system is dealing with a fault in a map task, Chrysaor executes the faulty tasks in another cloud until it obtains $f + 1$ equal results. If the re-execution of the map tasks has ended correctly, the solution has the capability to validate the results and find which cloud is compromised and exclude it from the rest of the execution. If a malicious fault or cloud outages have happened during the execution of the reduce tasks, it is necessary to run a new full job in a new cloud, and then validate the output. The execution aborts if no correct result is obtained and no more clouds are available to re-execute the tasks.

Note that we explained the difference between handling accidental faults and malicious faults as if the system was able to distinguish them, which is not the case. In practice, the system is configured with a threshold on the number of times it tries to handle faults as if they were accidental, then considers them malicious (i.e., starts using a new cloud).

IV. Context-based scheduling

When we are dealing with several cloud providers, we are facing *heterogeneity* in the server machines and the network. Choosing the best clouds is critical to gain in performance.

Notice that we do not mean the "best cloud" per se, but the cloud with more resources available to the user at the moment of submitting a MapReduce job. Naturally, if a job runs in a particular cloud with high computational power and is connected by high-bandwidth links, it ought to take relatively shorter time for the job to finish. On the contrary, if a cloud has low bandwidth links and low computational power, or if it is overloaded, it might take longer for the job to complete.

Devising a context-based scheduler that distributes replicated tasks across different clouds based on network throughput and computational power requires predicting which clouds (and which connections) will be the fastest. This prediction needs to consider both the historical performance as well as the current status of each cloud, allowing us to incorporate the heterogeneity of the clouds into the scheduling decision. As a consequence, our scheduler is split into two parts: one for *estimating data transmission time* and the other for *estimating data processing time*. We detail each metric in the following.

Estimating data transmission time. The data transmission time between two clouds depends on *(i)* its geographical distance, *(ii)* the network throughput, and *(iii)* the size of the data to transfer. Considering that the throughput varies with respect to the traffic load to other clouds (among other variables), the framework needs to periodically monitor the throughput for each pair of clouds in the system.

Estimating data processing time. The time for completing a given MapReduce job mainly depends on the following variables: *(i)* the capacity of the cloud running this job; and *(ii)* the configuration of the job. For example, a high level of parallelization (*i.e.*, a large number of map and reduce tasks) for the same job in the same cloud, and having tasks accessing mostly local splits implies shorter data processing times. As such, estimating the data processing time involves having a scheduler that takes into consideration three types of features: *job configuration*; *cloud capacity*; and *cloud overhead*. We describe the representative features for each type in the following:

- *Job configuration features.* Several variables in the job configuration need to be considered for the scheduler to predict the duration of the next job execution. These include the size of the input data, the number of map tasks, and the number of reduce tasks. These variables are known before the job starts. Clearly, large input data and a small number of map and reduce tasks imply long job completion times.
- *Cloud capacity features.* Different clouds have different characteristics, as they are composed of diverse hardware infrastructure (number and type of servers, etc.). Features of this type include the clock speed and the number of cores of the CPUs, the total memory capacity, etc. These variables define the cloud capacity, but they do not give evidence of the load of the cloud in a particular moment.
- *Cloud overhead features.* These features assess the load of the cloud at a specific time, in comparison to its base capacity (cloud capacity features). For instance, if there is resource contention in the cloud and there are jobs waiting to be launched, most likely this cloud should not be selected to execute the job. In contrast, if a cloud has sufficient free resources the scheduled job can finish early, even if its capacity is relatively low. The proposed scheduler uses the number of MapReduce jobs that are currently running in the cloud, the percentage of completion of the running jobs, the number of jobs queued to run, and the size of the input data of the running jobs as features to measure the cloud overhead.

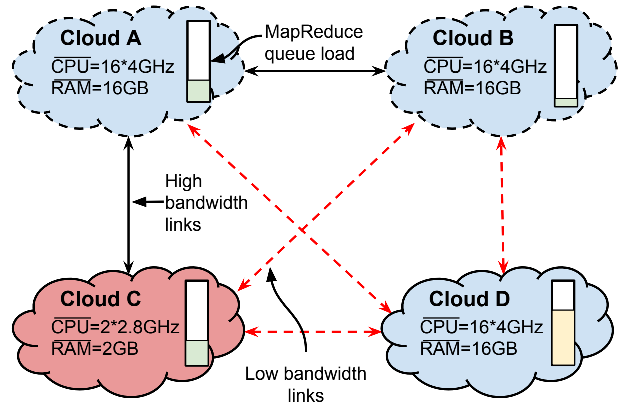

Fig. 4: Scheduling example

Figure 4 shows an example of our context-based scheduler. In the figure, we have three clouds with higher computational power (Cloud A, B, and D), each with 4 machines with 4GHz CPUs and 16 cores, and 16GB of RAM. Cloud C is less computationally powerful with 2.8GHz CPUs with only 2 cores, and 2GB of RAM. Notice that Cloud D has a high load in the MapReduce queue, which means that there are several jobs waiting to execute. In terms of the network connections between clouds, clouds A, B, and C are interconnected with high-throughput links (shown with the plain arrows). The remaining clouds are either connected with low bandwidth links or are overloaded with traffic (dashed arrows). In this example, Cloud A and Cloud B seem the best option to run the next MapReduce job replicas.

The goal of the scheduler is to take into account both an estimation of the data transfer time and the data processing time (considering both the job configuration, the cloud capacity, and the cloud overhead features) to choose the best cloud to run the next job. The scheduler estimates the time to transfer the data between clouds based on the historic throughput measurements and the size of input data to transfer. To estimate the data processing time, the scheduler uses linear regression based on the job and cloud features, in order to obtain the weight factors that are part of the next job prediction.

V. Fine-grained replication

Following our work in Medusa, the previous section mainly considered replication of MapReduce jobs to obtain fault tolerance and availability. In that section, even if a single small task of a large job fails and produces a wrong output, then the whole job will produce a wrong output and will need to be re-executed.

To improve the efficiency of the system, it would be desirable to perform replication and re-execution at the task level.

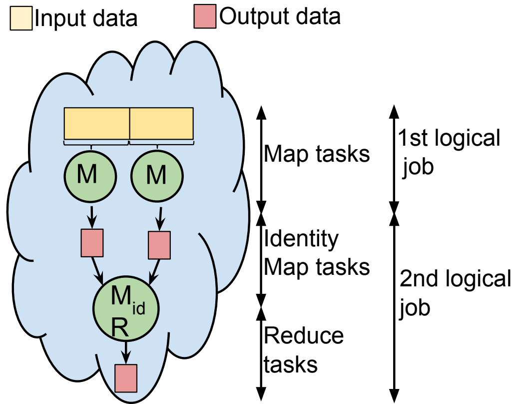

Fig. 5: Logical MapReduce Job

However, this fine-grained form of replication is a challenge, as it would require modifying the Hadoop MapReduce source code. The main reason of the problem is that the execution of a job in Hadoop MapReduce cannot be interrupted "externally". The need to modify Hadoop is, however, problematic as it would require users to use our own version of the platform, hindering adoption (for instance, users could not use publicly available versions, such as Amazon Elastic MapReduce).

We introduced a new abstraction in Chrysaor to perform fine-grained replication without changing Hadoop: the notion of a *logical job*. From the Hadoop viewpoint, each logical job is a complete MapReduce job, but from the Chrysaor viewpoint, there is one logical job to execute the map tasks and another one to execute the reduce tasks (see Figure 5). Moreover, if the replicas of a task produce different outputs, a new logical job is created to re-execute only that task. The use of logical jobs is transparent to the clients' applications, which request the execution of jobs as usual.

During the first logical job execution, each map task creates a digest of the map output. In the figure, the output data is represented as the squares exiting each map task. The digests will be fetched and compared by Chrysaor to check if all map task replicas produced equal results. This is the case in the example (we are considering no faults), and so the second logical job is launched. The second logical job cannot start from the reduce tasks (a MapReduce job always starts from a Map function). To solve this issue, we start this job with an identity map task, a simple task that outputs the input without modification. The second logical job will then read the data that was stored previously using identity map tasks, and perform the shuffle&sort phase before the reduce tasks start. Each reduce task will produce the final output and the system will compare the results. In the example, as there are no faults, the results are equal and the job execution terminates successfully. If that was not the case, a new logical job would be created to re-execute only that task.

With the use of logical jobs, it is thus possible to have a finer-grained control of the map or reduce tasks.

VI. Evaluation

We evaluated the two systems that form the basis of our multicloud resilient MapReduce design – Medusa and Chrysaor – experimentally, by running the two prototypes using several nodes in different regions of the Amazon EC2 service. We ran several real-world applications (available with Hadoop). In this article we focus on the comparison of the two replication approaches – job replication (Medusa) and task replication (Chrysaor) – between themselves and with the original Hadoop. We consider two applications: one communication-bound (CB) and the other computationally-intensive (CI). We present the results in Table I. We invite the interested reader to obtain further information on the evaluation, including a detailed analysis on its several results, in the paper that proposed Chrysaor [13].

CB application	No faults	Arbitrary faults		Malicious faults	
		Map	Reduce	Map	Reduce
Original Hadoop	379	not tolerated			
Job replication	438	823		1008	
Task replication	516	547	1053	547	1054

CI application	No faults	Arbitrary faults		Malicious faults	
		Map	Reduce	Map	Reduce
Original Hadoop	557	not tolerated			
Job replication	926	1831		1831	
Task replication	777	816	1438	816	1438

TABLE I: Performance of job replication vs task replication (in seconds, considering a 4GB input)

Comparison with the original Hadoop. One of our goals was to have an acceptable performance overhead, so the table shows average times for the execution of the original Hadoop in the clouds considered. Our job and task replication solutions have an overhead, as seen in the table, but it is reasonably low (between 16% and 39%). An overhead was unavoidable, as we are doing more computation: this is the price to pay for the benefit of tolerating severe faults. The overhead is limited mainly due to the principle of just enough replication explained in Section III.

Performance without faults. The approach followed by Medusa of replicating jobs achieves slightly better performance when compared with task replication for the CB application. The reason is the main overhead introduced by the required logical job abstraction: the identity map tasks require additional computational time. This additional computational time came from the fact that the output produced by the map tasks are larger than the input data. Overall, the characteristics of the CB application have brought a penalty to the new abstraction. Interestingly, the results are inverse with the CI application. As the application is computationally intensive, the relative cost of the identity maps is less pronounced. In addition, one optimization allowed by the Chrysaor design (namely, the generation of digests while the output is being produced) overcomes the logical job overhead.

Performance with arbitrary faults. We tested both solutions introducing arbitrary faults. The fine-grained replication at the task level allows the system to react immediately when a fault happens in the map tasks, which explains why task replication was always the fastest solution in this case. When a fault happens in the reduce tasks the result is different. As in the case without faults, in the CB application, the re-execution of identity tasks makes the overall solution slower than job

replication. In the CI application task replication was always faster than job replication, for the same reasons as before.

Performance with malicious faults. The conclusions of these experiments are similar to the previous case. As before, replication at the task level was always the fastest when tolerating faults in the map tasks. However, task replication was slower when dealing with faults at the reduce side in comparison with job replication when the job was not computationally-intensive, again due to the need to execute identity map tasks in the second logical job.

In summary, the main conclusions that can be drawn from our experiments are that task replication is favorable for workloads that are (i) more computationally-intensive and (ii) centered in map tasks. For (i), in this sort of applications, the relative overhead of the logical job abstraction is low. For (ii), and independently of the nature of the application, a fault in a map is always handled more efficiently with Chrysaor fine-grained replication scheme. Importantly, in most MapReduce jobs the number (and size) of map tasks is much larger than the number (and size) of reduce tasks, which means that in the common case the benefits of a fine-grained solution will outweigh the overhead introduced to guarantee transparency.

VII. Conclusion

In the era of cloud computing, Hadoop MapReduce has emerged as a popular tool for processing big data in a distributed way. The MapReduce framework is prepared to tolerate crash faults by re-executing tasks, but other faults that can affect the correctness of results are known to happen and will probably happen more regularly in the future. Moreover, the design of MapReduce is targeted to a single data center (a single cloud), which makes this framework vulnerable to cloud outages, which are also common.

Based on our recent experience in building such systems, in this article, we present three techniques to assist in the design of multicloud resilient MapReduce systems. Namely, minimizing the required replication; applying context-based job scheduling, based on cloud and network conditions; and performing fine-grained replication. Put together, these techniques offer resilience at reasonable cost, and they are immediately deployable using existing, unmodified Hadoop MapReduce solutions.

References

[1] M. Lacoste, M. Miettinen, N. Neves, F. M. Ramos, M. Vukolic, F. Charmet, R. Yaich, K. Oborzynski, G. Vernekar, and P. Sousa, "User-centric security and dependability in clouds-of-clouds," *IEEE Cloud Computing*, Sep. 2016.

[2] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa, "DepSky: Dependable and secure storage in a cloud-of-clouds," in *Proceedings of the 6th Conference on Computer Systems (EuroSys)*, 2011, pp. 31–46.

[3] 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.

[4] E. B. Nightingale, J. R. Douceur, and V. Orgovan, "Cycles, cells and platters: an empirical analysis of hardware failures on a million consumer PCs," in *Proceedings of the 6th Conference on Computer Systems (EuroSys)*, 2011, pp. 343–356.

[5] C. Snijders, U. Matzat, and U.-D. Reips, "Big data: Big gaps of knowledge in the field of internet science," *International Journal of Internet Science*, vol. 7, no. 1, pp. 1–5, 2012.

[6] T. White, *Hadoop: The Definitive Guide*, 1st ed. O'Reilly Media, Inc., 2009.

[7] 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.

[8] L. Wang, J. Tao, R. Ranjan, H. Marten, A. Streit, J. Chen, and D. Chen, "G-hadoop: Mapreduce across distributed data centers for data-intensive computing," *Future Gener. Comput. Syst.*, vol. 29, no. 3, pp. 739–750, Mar. 2013. [Online]. Available: http://dx.doi.org/10.1016/j.future.2012.09.001

[9] C. Cerin et al., "Downtime statistics of current cloud solutions," Jun. 2013, the International Working Group on Cloud Computing Resiliency.

[10] G. Clarke, "Microsoft Azure was most fail-filled cloud of 2014," http://www.theregister.co.uk/2015/01/16/microsoft_worst_cloud_uptime_2014/, January 2015.

[11] Amazon Web Services Inc, "Amazon S3 introduces cross-region replication," https://aws.amazon.com/pt/about-aws/whats-new/2015/03/amazon-s3-introduces-cross-region-replication/, Mar 2015.

[12] P. A. R. S. Costa, X. Bai, F. M. V. Ramos, and M. Correia, "Medusa: An efficient cloud fault-tolerant mapreduce," *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pp. 443–452, 2016.

[13] P. A. R. S. Costa, F. M. V. Ramos, and M. Correia, "Chrysaor: Fine-grained, fault-tolerant cloud-of-clouds mapreduce," *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2017.

[14] P. Costa, M. Pasin, A. N. Bessani, and M. Correia, "On the performance of Byzantine fault-tolerant mapreduce," *IEEE Transactions on Dependable and Secure Computing*, vol. 10, no. 5, pp. 301–313, 2013.

[15] 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.