MRBS: Towards Dependability Benchmarking for Hadoop MapReduce

Amit Sangroya, Damián Serrano and Sara Bouchenak

University of Grenoble - LIG - INRIA, Grenoble, France
{Amit.Sangroya,Damian.Serrano,Sara.Bouchenak}@inria.fr

Abstract. MapReduce is a popular programming model for distributed data processing. Extensive research has been conducted on the reliability of MapReduce, ranging from adaptive and on-demand fault-tolerance to new fault-tolerance models. However, realistic benchmarks are still missing to analyze and compare the effectiveness of these proposals. To date, most MapReduce fault-tolerance solutions have been evaluated using microbenchmarks in an ad-hoc and overly simplified setting, which may not be representative of real-world applications. This paper presents MRBS, a comprehensive benchmark suite for evaluating the dependability of MapReduce systems. MRBS includes five benchmarks covering several application domains and a wide range of execution scenarios such as data-intensive vs. compute-intensive applications, or batch applications vs. online interactive applications. MRBS allows to inject various types of faults at different rates and produces extensive reliability, availability and performance statistics. The paper illustrates the use of MRBS with Hadoop clusters.

Keywords: Benchmark; Dependability; MapReduce; Hadoop

1 Introduction

MapReduce has become a popular programming model and runtime environment for developing and executing distributed data-intensive and compute-intensive applications [1]. It offers developers a means to transparently handle data partitioning, replication, task scheduling and fault-tolerance on a cluster of commodity computers. Hadoop [2], one of the most popular MapReduce frameworks, provides key fault-tolerance features.

There has been a large amount of work towards improving fault-tolerance solutions in MapReduce. Several efforts have explored on-demand fault-tolerance [3], replication and partitioning policies [4], [5], adaptive fault-tolerance [6], [7], and extending MapReduce with other fault-tolerance models [8], [9]. However, there has been very little in the way of empirical evaluation of MapReduce dependability. Evaluations have often been conducted in an ad-hoc manner, such as turning off a node in the MapReduce cluster or killing a task process. Recent tools, like Hadoop fault injection framework offer the ability to emulate non-deterministic exceptions in the distributed filesystem. Although they provide a means to program unit tests for HDFS, such low-level tools are meant to be used by developers who are familiar with the internals of HDFS, and are unlikely to be used by end-users of MapReduce systems. MapReduce fault injection must therefore be generalized and automated for higher-level and easier use. Not only it is necessary to automate the injection of faults, but also the definition and generation of MapReduce faultloads. A faultload describes *what* fault to inject (e.g. a node crash), *where* to inject it (e.g. which node of the MapReduce cluster), and *when* to inject it (e.g. five minutes after the application started).

This paper presents MRBS (MapReduce Benchmark Suite), for evaluating the dependability of MapReduce systems. MRBS enables automatic faultload generation and injection in MapReduce. This covers different fault types, injected at different rates, which will provide a means to analyze the effectiveness of fault-tolerance in a variety of scenarios. Three different aspects of load are considered: dataload, workload and faultload. MRBS allows to quantify dependability levels provided by MapReduce fault-tolerance systems, through an empiric evaluation of the availability and reliability of such systems, in addition to performance and cost metrics. Moreover, MRBS covers five application domains: recommendation systems, business intelligence, bioinformatics, text processing, and data mining. It supports a variety of workload and dataload characteristics, ranging from compute-oriented to data-oriented applications, batch applications to online interactive applications. Indeed, while MapReduce frameworks were originally limited to offline batch applications, recent works are exploring the extension of MapReduce beyond batch processing [10].

MRBS shows that when running the Bioinformatics workload and injecting a faultload that consists of a hundred map software faults and three node faults, Hadoop MapReduce handles these failures with high reliability (94% of successful requests) and high availability (96% of the time). We wish to make dependability benchmarking easy to adopt by end-users of MapReduce and developers of MapReduce fault-tolerance systems. MRBS allows automatic deployment of experiments on private or public clouds being independent of any particular infrastructure. MRBS is available as a software framework to help researchers and practitioners to better analyze and evaluate the dependability and performance of MapReduce systems. It can be downloaded from *http://sardes.inrialpes.fr/research/mrbs.*

2 Background on Hadoop Fault Tolerance

MapReduce is a programming model and a software framework introduced by Google in 2004 to support distributed computing and large data processing on clusters of commodity machines [1]. MapReduce supports a wide range of applications such as image analytics, next-generation sequencing, recommendation systems, search engines, social networks, business intelligence, and log analysis.

There are many implementations of MapReduce among which the popular open-source Hadoop framework, which is also available in public clouds such as Amazon EC2 or Open Cirrus. A Hadoop cluster consists of a *master node* and *slave nodes*. Users (i.e. clients) of a Hadoop cluster submit MapReduce jobs to the master node which hosts the *JobTracker* daemon that is responsible of scheduling the jobs. Each slave node hosts a *TaskTracker* daemon that periodically communicates with the master node to indicate whether the slave is ready to run new tasks. If it is, the master schedules appropriate tasks on the slave.

Hadoop framework also provides a distributed filesystem (HDFS) that stores data across cluster nodes. HDFS architecture consists of a *NameNode* and *DataNodes*. The *NameNode* daemon runs on the master node and is responsible of managing the filesystem namespace and regulating access to files. A *DataNode* daemon runs on a slave node and is responsible of managing storage attached to that node. HDFS is thus a means to store input, intermediate and output data of Hadoop MapReduce jobs. Furthermore, for fault tolerance purposes, HDFS replicates data on different nodes.

One of the major features of Hadoop MapReduce is its ability to tolerate failures of different types, as described in the following.

Node Crash: In case of a slave node failure, the JobTracker on the master node stops receiving heartbeats from the TaskTracker on the slave for an interval of time. When it notices the failure of a slave node, the master removes the node from its pool and reschedules ongoing tasks on other nodes.

Task Process Crash: A task may also fail because a map or reduce task process suddenly crashes, e.g., due to a transient bug in the underlying (virtual) machine. Here again, the parent TaskTracker notices that a task process has exited and notifies the JobTracker for possible task retries.

Task Software Fault: A task may fail due to errors and runtime exceptions in *map* or *reduce* functions written by the programmer. When a TaskTracker on a slave node notices that a task it hosts has failed, it notifies the JobTracker which reschedules another execution of the task, up to a maximum number of retries.

Hanging Tasks: A map or reduce task is marked as failed if it stops sending progress updates to its parent TaskTracker for a period of time (indicated by *mapred.task.timeout* Hadoop property). If that occurs, the task process is killed, and the JobTracker is notified for possible task retries.

3 Dependability Benchmarking for Hadoop MapReduce

To use MRBS, three main steps are needed: (i) build a faultload (i.e. fault scenario) to describe the set of faults to be injected, (ii) conduct fault injection experiments based on the faultload, and (iii) collect statistics about dependability levels of the MapReduce system under test. This is presented in Figure 1.

The evaluator of the dependability of a MapReduce system chooses an application from MRBS' set of benchmarks, depending on the desired application domain and whether he/she targets compute-oriented or data-oriented applications. MRBS injects (possibly default) workload and dataload in the system under test. MRBS also allows the evaluator to choose specific dataload and workload, to stress the scalability of the MapReduce system (see Sections 3.3 and 3.4 for more details).

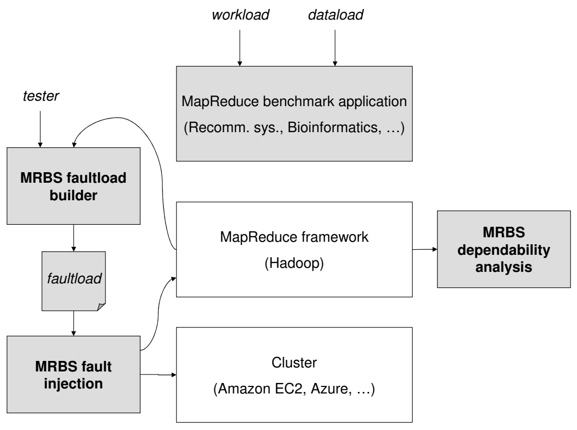


Fig. 1. Overview of MRBS dependability benchmarking

3.1 Faultload Builder

A faultload in MRBS is described in a file, either by extension, or by intention. In the former case, each line of the faultload file consists of the following elements: the time at which a fault occurs , the type of fault that occurs and, optionally, where the fault occurs. A fault belongs to one of the fault types handled by Hadoop MapReduce[1]. Another way to define a more concise faultload is to describe it by intention. Here, each line of the faultload file consists of: a fault type, and the mean time between failures (MTBF) of that type. Thus, testers can explicitly build synthetic faultloads representing various fault scenarios.

A faultload description may also be automatically obtained, either randomly or based on previous application runs' traces. The random faultload builder produces a faultload description where, with each fault type, is associated a random MTBF between 0 and the length of the experiment. Similarly, the random faultload builder may produce a faultload by extension, where it generates the i-th line of the faultload file as follows: $< time\_stamp_i,\ fault\_type_i,\ fault\_location_i >$, with $time\_stamp_i$ being a random value between $time\_stamp_{i-1}$ (or 0 if $i = 1$) and the length of the experiment, $fault\_type_i$ and $fault\_location_i$ random values in the set of possible values. A faultload description may also be automatically generated based on traces of previous runs of MapReduce applications.

MRBS faultload builder is relatively portable: its two first variants – explicit faultload builder and random faultload builder – are general enough and do not rely on any specific platform. The trace-based faultload builder is independent from the internals of the MapReduce framework, and produces a faultload description based on the structure of the MapReduce framework's logs; it currently works on Hadoop MapReduce framework.

[1] Other types of faults, such as network disconnection, may be emulated by MRBS, although we do not detail them in this paper.

3.2 Fault Injection

The output of the MRBS faultload builder is passed to the MRBS fault injector. The MRBS fault injector divides the input faultload into subsets of faultloads: one crash faultload groups all crash faults that will occur in all nodes of the MapReduce cluster (i.e. node crash, task process crash), and one per-node faultload groups all occurrences of other types of faults that will occur in one node (i.e. task software faults, hanging tasks).

The MRBS fault injector runs a daemon that is responsible of injecting the crash faultload. In the following, we present how the daemon injects these faults, in case of a faultload described by extension, although this can be easily generalized to a faultload described by intention. Thus, for the i-th fault in the crash faultload, the daemon waits until $time\_stamp_i$ is reached, then calls the fault injector of $fault\_type_i$ (see below), on the MapReduce cluster node corresponding to $fault\_location_i$. This fault injector is called as many times as there are occurrences of the same fault at the same time. The fault injection daemon repeats these operations for the following crash faults, until the end of the faultload file is reached or the experiment finishes.

The MRBS fault injector handles the per-node faultloads differently. A per-node faultload includes faults that occur inside tasks. MRBS intercepts task creation to check whether a fault must be injected in that task, in which case the fault injector corresponding to the fault type is called (see below). MRBS does not require the modification of the source code of the MapReduce framework. Instead, it synthesizes a new version of the MapReduce framework library using aspect-oriented techniques. The synthetic MapReduce library has the same API as the original one, but underneath this new library includes task creation interceptors that encode the fault injection logic.

Node Crash Injection: A node crash is simply implemented by shutting down a node. This fault injector uses the API of the underlying cloud infrastructure to implement such a fault. For example, in case of a public cloud such as Amazon EC2, a node crash consists in a premature termination of an Amazon EC2 instance. However, if a tester wants to conduct multiple runs of the same dependability experiment, and if faults are implemented by shutting down machines, new machines must be acquired from the cloud at the beginning of each run, which may induce a delay. For efficiency purposes, we propose an implementation of MapReduce node fault which kills all MapReduce daemons running on that node. Specifically, in the case of Hadoop these include the *TaskTracker* and *DataNode* daemons running in a slave node[2]. The timeout to detect a MapReduce node failure is set to 30 seconds, a value set in *mapred.task.tracker.expiry.interval* Hadoop property.

Task Process Crash Injection: This type of fault is implemented by killing the process running a task on a MapReduce node.

[2] A node crash is not injected to the MapReduce master node since this node is not fault-tolerant.

Task Software Fault Injection: A task software fault is implemented as a runtime exception thrown by a map task or a reduce task. This fault injector is called by the interceptors injected into the MapReduce framework library by MRBS.

Provoking Hanging Tasks: A task is marked as hanging if it stops sending progress updates for a period of time. This type of fault is injected into a map task or a reduce task through the interceptors that make the task sleep a longer time than the maximum period of time for sending progress updates (*mapred.task.timeout* Hadoop property).

The MRBS faultload injector is relatively portable: it is independent from the internals of the MapReduce framework and the per-node faultload injectors are automatically integrated within the framework based upon its API. The current version of the MRBS faultload injector works for Hadoop MapReduce; porting to new platforms is straightforward.

3.3 Benchmark Suite

Conceptually, a benchmark in MRBS implements a service that provides different types of operations, which are requested by clients. The benchmark service is implemented as a set of MapReduce programs running on a cluster, and clients are implemented as external entities that remotely request the service. Depending on the complexity of a client request, the request may consist of one or multiple successive MapReduce jobs. A benchmark has two execution modes: interactive mode or batch mode. In interactive mode, concurrent clients share the MapReduce cluster at the same time (i.e. have their requests executed concurrently). On the other hand, requests from different clients are executed in FIFO order (one after another) in batch mode.

A benchmark run has three successive phases: a warm-up phase, a run-time phase, and a slow-down phase, which length may be chosen by the end-user of the benchmark. The end-user may also choose the number of times a benchmark is run, to produce average statistics. MRBS benchmark suite consists of five benchmarks covering various application domains such as recommendation systems, business intelligence, bioinformatics, text processing, and data mining. The user can choose the actual benchmark.

Recommendation System: Recommendation systems are widely used in e-commerce sites. MRBS implements an online movie recommender system. It builds upon a set of movies, a set of users, and a set of ratings and reviews users give for movies to indicate whether and how much they liked or disliked the movies.

Business Intelligence: The Business Intelligence benchmark represents a decision support system for a wholesale supplier. It implements business-oriented queries that examine large volumes of data, execute queries with a high degree of complexity, and give answers to critical business questions. It uses Apache

Hive on top of Hadoop, a data warehouse that facilitates ad-hoc queries using a SQL-like language called HiveQL.

Bioinformatics: The benchmark includes a MapReduce-based implementation of DNA sequencing. The data used in the benchmark are publicly available genomes. Currently, the benchmark allows to analyze several genomes of organisms such as the pathogenic organisms *Salmonella typhi, Rhodococcus equi, and Streptococcus suis.*

Text Processing: Text processing is a classical application of MapReduce. MRBS provides a MapReduce text processing-oriented benchmark, with three types of operations allowing clients to search words or word patterns in text documents, to know how often words occur in text documents, or to sort the contents of documents. The benchmark uses synthetic input data that consist of randomly generated text files of different sizes.

Data Mining: This benchmark provides two types of data mining operations: clustering and classification. MRBS benchmark considers the case of classifying newsgroup documents into categories. Furthermore, the benchmark provides canopy clustering operations.

3.4 Using MRBS

MRBS comes with a configuration file that involves several parameters among which the following: the actual benchmark to use, the length of the benchmark warm-up phase, runtime phase, and slow-down phase, the size of the benchmark input data set, the size of the MapReduce cluster, the cloud infrastructure that will host the cluster, in addition to workload and faultload characteristics described in the previous section. To keep the use of MRBS simple, these parameters have default values that may be adjusted by MRBS user.

MRBS produces various runtime statistics related to performance and dependability. These include client request response time, throughput, financial cost, failed client requests vs. successful requests, availability, and reliability. MRBS also provides low-level MapReduce statistics related to the number, length and status (i.e. success or failure) of MapReduce jobs; tasks; the size of data read from or written to the distributed file system, etc. These low-level statistics are built using Hadoop counters. Optionally, MRBS can generate charts plotting continuous-time results. More details on MRBS can be found in [11].

4 Evaluation

4.1 Experimental Setup

We conducted several experiments with MRBS on Hadoop clusters running in Amazon EC2 and Grid'5000, but due to space limitation we only present one case study in this paper. The experiments presented in this section were conducted in a cluster running in Grid'5000, a French geographically distributed infrastructure used to study large-scale parallel and distributed systems. The hardware configuration consists of 4-core 2-CPU, 2.5 GHz Intel Xeon E5420 QC CPU, 8 GB memory, 160 GB SATA storage (per node) and 1 GB Ethernet network. The cluster consists of one node hosting MRBS and emulating concurrent clients, and a set of nodes hosting the MapReduce cluster.

Financial cost is \$0.34 per instance·hour. In the following, each experiment is run three times to report average and standard deviation results. The operating system of the nodes is Debian Linux 6 with kernel v2.6.32. The MapReduce framework is Apache Hadoop v0.20.2, and Hive v0.7, on Java 6.

4.2 Experimental Results

In this section, we illustrate the use of MRBS to evaluate the fault-tolerance of Hadoop MapReduce. Here, a ten-node Hadoop cluster runs the Bioinformatics benchmark, used by 20 concurrent clients. The experiment is conducted during a run-time phase of 60 minutes, after a warm-up phase of 15 minutes. We consider a synthetic faultload that consists of software faults and hardware faults as follows: first, 100 map task software faults are injected 5 minutes after the beginning of the run-time phase, and then, 3 node crashes are injected 25 minutes later.

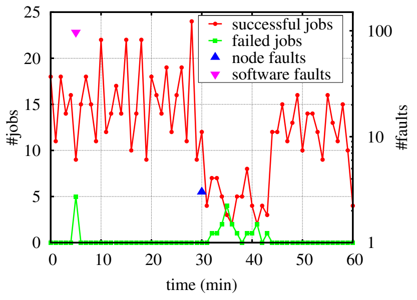
Table 1. Reliability, availability, and cost

Reliability	**Availability**	**Cost (dollars/request)**
94%	96%	0.008 (+14%)

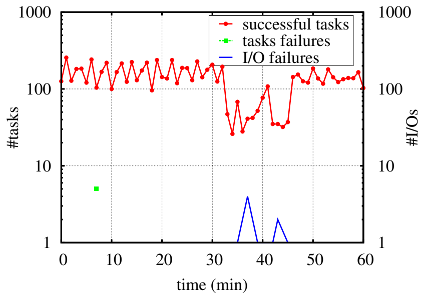
Although the injected faultload is aggressive, the Hadoop cluster remains available 96% of the time, and is able to successfully handle 94% of client requests (see Table 1). This has an impact on the request cost which is 14% higher than the cost obtained with the baseline (non-faulty) system.

To better explain the behavior of the MapReduce cluster, we will analyze MapReduce statistics, as presented in Figures 2(a) and 2(b). Figure 2(a) presents successful MapReduce jobs and failed MapReduce jobs over time. Note the logarithmic scale of the right side y-axis. When software faults occur, few jobs actually fail. On the contrary, node crashes are more damaging and induce a higher number of job failures, with a drop of the throughput of successful jobs from 16 jobs/minute before node failures to 5 jobs/minute after node failures.

Figure 2(b) shows the number of successful MapReduce tasks and the number of failed tasks over time, differentiating between tasks that fail because they are unable to access data from the underlying filesystem (i.e. I/O failures in the Figure), and tasks that fail because of runtime errors in all task retries (i.e. task failures in the Figure). We notice that software faults induce task failures that appear at the time the software faults occur, whereas node crashes



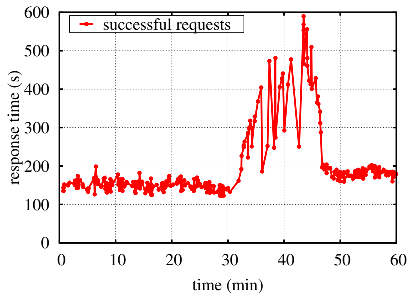
(a) jobs



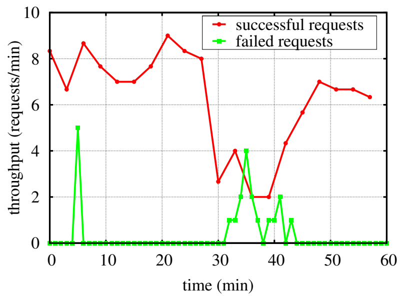
(b) tasks

Fig. 2. Successful vs. failed MapReduce jobs and tasks

induce I/O failures that last fifteen minutes after the occurrence of node faults. Actually, when some cluster nodes fail, Hadoop must reconstruct the state of the filesystem, by re-replicating the data blocks that were on the failed nodes from replicas in other nodes of the cluster. This explains the delay during which I/O failures are observed.



(a) response time



(b) throughput

Fig. 3. Client request response time and throughput

Figure 3(a) shows the response time of successful client requests. With software faults, there is no noticeable impact on response times. Conversely, response time sharply increases when there are node faults, and while Hadoop is rebuilding missing data replicas. Similarly, Figure 3(b) presents the impact of failures on client request throughput. Interestingly, when the Hadoop cluster looses 3 nodes, it is able to fail-over, however, at the expense of a higher response time (+30%) and a lower throughput (-12%).

5 Conclusions and Perspectives

To evaluate the dependability of MapReduce systems, MRBS allows to characterize a faultload, generate it, and inject it in an online Hadoop MapReduce cluster. MRBS performs an empirical evaluation of the availability and reliability of such systems, to quantify their dependability levels. MRBS is available as a software framework to help researchers and practitioners to better analyze and evaluate the fault-tolerance of MapReduce systems. Important perspectives of this work is the addition of security evaluation that supports data security attacks injection scenarios.

6 Acknowledgments

This work was partly supported by the Agence Nationale de la Recherche, the French National Agency for Research, under the MyCloud ANR project, and the University of Grenoble. Part of the experiments were conducted on the Grid'5000 experimental testbed, developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies.

References

1. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. In: OSDI. (2004)
2. Apache Hadoop. (`http://hadoop.apache.org`)
3. Fadika, Z., Govindaraju, M.: LEMO-MR: Low Overhead and Elastic MapReduce Implementation Optimized for Memory and CPU-Intensive Applications. In: IEEE CloudCom. (2010)
4. Ananthanarayanan, G., Agarwal, S., Kandula, S., Greenberg, A., Stoica, I., Harlan, D., Harris, E.: Scarlett: Coping with Skewed Content Popularity in MapReduce Clusters. In: European Conf. on Computer Systems (EuroSys). (2011)
5. Eltabakh, M., Tian, Y., Ozcan, F., Gemulla, R., Krettek, A., McPherson, J.: CoHadoop: Flexible Data Placement and Its Exploitation in Hadoop. In: VLDB. (2011)
6. Jin, H., Yang, X., Sun, X.H., Raicu, I.: ADAPT: Availability-Aware MapReduce Data Placement in Non-Dedicated Distributed Computing Environment. In: ICDCS. (2012)
7. Lin, H., Ma, X., Archuleta, J., Feng, W.c., Gardner, M., Zhang, Z.: MOON: MapReduce On Opportunistic eNvironments. In: HPDC. (2010)
8. Bessani, A.N., Cogo, V.V., Correia, M., Costa, P., Pasin, M., Silva, F., Arantes, L., Marin, O., Sens, P., Sopena, J.: Making Hadoop MapReduce Byzantine Fault-Tolerant. In: DSN, Fast abstract. (2010)
9. Ko, S.Y., Hoque, I., Cho, B., Gupta, I.: Making Cloud Intermediate Data Fault-Tolerant. In: ACM Symp. on Cloud computing (SoCC). (2010)
10. Liu, H., Orban, D.: Cloud MapReduce: A MapReduce Implementation on Top of a Cloud Operating System. In: CCGRID. (2011)
11. Sangroya, A., Serrano, D., Bouchenak, S.: MRBS: A Comprehensive MapReduce Benchmark Suite. Research Report RR-LIG-024, LIG, Grenoble, France (Feb 2012)