Method for Testing the Fault Tolerance of MapReduce Frameworks

João Eugenio Marynowski[a,b], Altair Olivo Santin[a], Andrey Ricardo Pimentel[b]

[a]*Pontifical Catholic University of Parana*
[b]*Federal University of Parana*
Curitiba, Parana, Brazil.

Abstract

A MapReduce framework abstracts distributed system issues, integrating a distributed file system with an application's needs. However, the lack of determinism in distributed system components and reliability in the network may cause applications errors that are difficult to identify, find, and correct. This paper presents a method to create a set of fault cases, derived from a Petri net (PN), and a framework to automate the execution of these fault cases in a distributed system. The framework controls each MapReduce component and injects faults according to the component's state. Experimental results showed the fault cases are representative for testing Hadoop, a MapReduce implementation. We tested three versions of Hadoop and identified bugs and elementary behavioral differences between the versions. The method provides network reliability enhancements as a byproduct because it identifies errors caused by a service or system bug instead of simply assigning them to the network.

Keywords:
Testing MapReduce, Fault Tolerance, Petri net Modeling, Hadoop Reliability, Fault Injection

1. Introduction

Massive data processing (e.g., big data) requires an efficient and reliable network service and connection in order to use a large number of machines. Hence,

☆Corresponding author, tel.: +55 41 32711669, fax: +55 41 32712121, e-mail: jeugenio@ppgia.pucpr.br (J.E. Marynowski). A preliminary version of this work appeared in the Proceedings of the 16th International Conference on Enterprise Information Systems (ICEIS) [1].

some computational systems become fundamental network services. For instance, MapReduce [2] is a service that enables the storage and processing of large amount of data used by various applications, such as social networks or research and business applications. MapReduce offers a programming environment based on two high-level functions, *map* and *reduce*. It also offers a runtime environment to execute these functions on a computer cluster and addresses abstracting issues such as processing distribution, data partition, replication, and fault tolerance.

The MapReduce architecture includes several *worker* components and one *master* component that schedules the *map* and *reduce* tasks to run on the *workers*. As with other network services that use thousands of machines, a MapReduce system often faces failures caused by various conditions, e.g., network connection delays, power outages, hardware problems, and updates or software defects. The MapReduce fault tolerance mechanism identifies faulty *workers* by timeout, and reschedules their tasks to a healthy *worker*. The fault handling differs between tasks and their status. For example, if a *worker* fails when it is executing a *map* task, the *master* only reassigns its task to another *worker*. However, if a *worker*$_x$ fails after executing a *map* task, the *master* reassigns the task to another *worker* (worker$_y$) and informs all *workers* executing *reduce* tasks that they must read the *map* result from the new *worker*$_y$.

It is essential to ensure that failures do not interfere or interrupt the execution of a MapReduce system. Fault tolerance testing aims to find errors in the implementation or specification of fault tolerant mechanisms [3, 4]. For this purpose, the system is executed in a controlled testing environment with the injection of known faults. A fault case is a set of requirements for the complete execution and validation of the system under test when faults are injected [5, 6]. There are two main issues concerning the testing of fault tolerance on network services: how to choose representative elements from the potentially infinite and partially unknown set of possible fault cases, and the automation of fault case executions.

Representative fault cases can be generated (derived) from the model of the system's fault tolerance mechanism. However, modeling a distributed fault tolerance mechanism requires a formal model that represents the concurrent and distributed behavior of the system. The model must represent components as dynamic items, to be easily inserted or removed, without substantial changes in the model. Furthermore, the model should represent the system components without specifying their actions and states, allowing an action to be performed by any enabled component.

An important issue regarding the provisioning of network services occurs when a user faces some performance issues. Usually, users do not have enough information to disambiguate whether the problem occurs in the network connection or in the provided service. When the network service is tested, it is easier identify in each circumstance whether the problem is in the network service, or connection. In this paper, we work with the hypothesis that is possible to enhance network reliability by disambiguating the situation using service tests.

We present an approach for testing MapReduce fault tolerance based on the generation and execution of representative fault cases. Representative fault cases are derived from the reachability graph of a Petri net (PN) that models the fault tolerance mechanism. A reachability graph consists of all possible sequences of transition firings from a PN. The PN model represents MapReduce components as dynamic items, enabling them to be easily removed or inserted without substantial model changes and without specifying their actions, allowing an action to be executed by any enabled component. HadoopTest is a testing framework that automates the execution of representative fault cases in a distributed environment, controlling and monitoring each system component and injecting faults according to their status.

We apply our approach to test three versions of an open-source MapReduce implementation called Hadoop [7]. Experimental results show that the fault cases derived from the PN model are representative for testing Hadoop by injecting faults according to the fault tolerance mechanism and identifying some errors. HadoopTest automates the execution of representative fault cases in a distributed environment and has the required properties of a test framework: controllability, time measurement, non-intrusiveness, repeatability, and efficacy.

We organized the remainder of the paper as follows. Section 2 describes related work. Section 3 presents the modeling of the MapReduce fault tolerance mechanism and the process used to generate fault cases. Section 4 presents the framework to automate the execution of fault cases. Section 5 describes the experimental results of Hadoop testing. Section 6 concludes the paper.

2. Related Work

Dean and Ghemawat [2], *Abouzeid et al.* [8], and *Sangroya et al.* [9] addressed MapReduce fault tolerance testing, but they assigned the generation of fault cases to the test engineer. In general, the generated fault cases disregard the internals of the MapReduce fault tolerance mechanism, e.g., they inject faults in some nodes for a defined period (e.g., failing 3 of 10 nodes for 10 s, 30 s after the execution beginning). Thus, the fault cases used do not consider the various fault handling behaviors of the MapReduce fault tolerance mechanism.

Bernardi et al. [10], *Jacques-Silva et al.* [11], and *Lefever et al.* [12] tested other distributed systems also using fault cases provided by the test engineer. Leaving the fault case generation to the test engineer results in a test that is limited by their knowledge. *Benso et al.* [13], *Chandra et al.* [14], and Henry [15] generate fault cases randomly in order to fulfill this deficiency. However, even random fault cases cannot find the errors that appear only when faults are injected into a specific sequence of failures that are not executed randomly. *Joshi et al.* [16], and *Fu et al.* [17] systematically generated fault cases from the source code. However, the approach is too costly and limits the fault cases to a few concurrent faults.

Echtle et al. [5], *Ambrosio et al.* [6], and *Bernardi et al.* [10] generated fault cases from an abstraction of the fault tolerance mechanism. However, these existing proposals are not applicable to MapReduce systems because they are specific and limited to single-machine systems.

Pan et al. [18], *Butnaru et al.* [19], *Zhou et al.* [20], and *Almeida et al.* [21] presented testing frameworks for distributed systems to control distributed components and validate the system behavior. However, these frameworks do not control the system-specific circumstance for injecting a fault according to the component processing. *Jacques-Silva et al.* [11], *Pham et al.* [22], *Lefever et al.* [12], and *Hoarau et al.* [23] presented fault injection frameworks, but with the same problem. They support the execution of fault cases with the injection of various multiple faults, but they do not consider the system processing stages.

Herriot [24] is a testing framework that provides a set of interfaces to validate small system parts, e.g., a method or function. *Csallner et al.* [25] presented an approach to systematically search for MapReduce application faults based on badly defined map and reduce functions. *Pan et al.* [18], *Tan et al.* [26], and *Huang et al.* [27] evaluated MapReduce execution logs to detect MapReduce performance problems. Although these approaches evaluate system functionality and performance, they do not automatically execute fault cases and validate the system considering its fault tolerance.

3. Generating Representative Fault Cases

This section presents Petri Net-based Fault Cases Generation (PbGen), our approach to model the system fault tolerance mechanism and create representative fault cases from a PN model.

A fault case is a set of requirements for the complete execution, fault injection, and validation of a system under test. In a distributed system, a fault case is a set of actions that must be executed by a set of testers. A coordinator controls a fault case execution, and each tester controls a system component while executing fault case actions.

Definition 3.1 (Fault case). *A fault case is a 4-tuple $\mathcal{F} = (T^{\mathcal{F}}, A^{\mathcal{F}}, R^{\mathcal{F}}, \mathcal{O})$ where: $T^{\mathcal{F}} = \{t_0, t_1, \ldots, t_n\}$ is a list of testers that control the system components, $A^{\mathcal{F}} = \{a_0, a_1, \ldots, a_m\}$ is a list of actions that can involve fault injections, $R^{\mathcal{F}} = \{r_{a_0}, \ldots, r_{a_m}\}$ is a list of action results, and $\mathcal{O}$ is an oracle.*

The oracle is the mechanism responsible for verifying the system behavior during a fault case execution, and associating a test result, called a verdict with it. A fault case verdict can be: *pass*, *fail*, or *inconclusive*. Further, each action execution can have the result: *success*, *failure*, or *timeout* (no response over a period of time). The $\mathcal{F}$ verdict is *pass* if all action executions are a *success*. The $\mathcal{F}$ verdict is *fail* if any action execution is a *failure*. The $\mathcal{F}$ verdict is *inconclusive* if at least one action execution is a *timeout*, meaning that the test is inaccurate and the fault case must be rerun.

Definition 3.2 (Fault case action). *A fault case action of $A^{\mathcal{F}}$ is a 7-tuple $a_i = (h, D, n, T', W, \theta, I)$, where:*

- *$h \in \mathbb{N} \mid h \leqslant |A|$ is a hierarchical order in which action a_i must be executed (actions with the same h execute in parallel);*
- *$D \subseteq A \mid \forall a_j \in D : r_{a_j} = success$ is a set of actions that must be successfully executed before a_i, otherwise the action result r_{a_i} is failure;*
- *$n \in \mathbb{N} \mid n \leqslant |C'|$ is the number of success action results, i.e., the number of testers that must execute a_i and get success as result;*
- *$T' \subseteq T^{\mathcal{F}}$ is a set of testers that can execute a_i;*
- *W is a trigger, i.e., an optional instruction or command that is executable by testers and is required to execute a_i;*
- *θ is the time to execute a_i;*
- *I is a set of instructions or commands that are executable by testers.*

The representative fault cases for testing fault tolerant network services are the fault cases generated from the model of the system fault tolerance mechanism. In our use case, the model must represent the MapReduce components, a *master* and various *workers*, as dynamic items to be easily inserted or removed without substantial changes in the model. Furthermore, the model should represent the components without specifying their actions and states, allowing an action to be performed by any enabled component.

3.1. PN Modeling for the MapReduce Fault Tolerance Mechanism

A PN is a graphical and mathematical tool that is able to represent a system considering its distributed properties, such as parallelism, distribution, asynchronism, and non-determinism [28, 29]. A PN graph consists of circles representing places, bars representing transitions, and arcs connecting places and transitions. An arc may have an associated number that represents its weight (1 when absent). Inside the places, there are small black circles, called tokens, that are removed from one place and added to another place after a transition has been fired. A marking is any distribution of tokens and represents a system state. A transition fires only when there are enough tokens in the places of the input arcs that are connected by the transition.

Traditionally, a token represents a message or datum for modeling distributed system using PN. A token assumes several states (places) that are altered by actions (transitions). This approach represents distinct and parallel actions, however, it does not model the independence of the states and actions with respect to the system components.

Marynowski et al. [1, 30] used tokens to represent MapReduce components, places to represent system states or components status, and transitions to represent fault case actions. MapReduce components are easily added and removed from the system by simply changing the number of tokens in the PN, and any available component (worker or nodes) can execute an action, i.e., any token can be used to fire a transition. Here, we present an extended version of the MapReduce fault tolerant model to represent the complete process of an application execution and to test and validate the *master* behavior where there are no available *workers*.

Figure 1 shows a PN that models a fragment of the MapReduce fault tolerance mechanism. The model represents the process to start the *master* and *workers*, start an application, and stop the components afterward. Additionally, the model represents the handling of faults that occur while a *worker* executes a *map* or *reduce* task, or when all *workers* fail, causing the application to fail.

Places *begin*, *nodes*, and *end* represent the initial state with some nodes available (the number of tokens in place *nodes*) and the final state with the system offline. Places *master.online* and *worker.online* represent the number of *master* and *worker* components that were started. Places *worker.runningMap* and *worker.runningReduce* represent a *worker* running a *map* and a *reduce* task. Place *master.endApp* represents the end of the application that can be a successful application, where the expected result is correct, even if some *workers* fail, or a failed application, when all *workers* fail. Place *lastWorker* represents the case when there is only one *worker* on-line, and place *noWorker* represents the case when all *workers* fail.

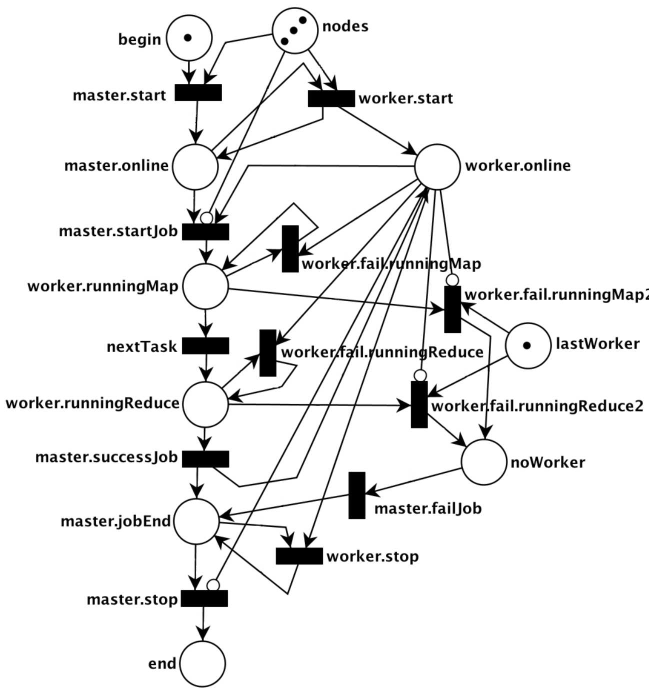


Figure 1: PN of the MapReduce fault tolerance mechanism.

Transitions *master.start* and *worker.start* represent the *master* and *worker* initializations, respectively. Transition *master.startJob* represents the start of a MapReduce application by the *master*. Transitions *worker.fail.runningMap* and *worker.fail.runningReduce* represent a fault injection in a *worker* while it executes a *map* or *reduce* task, respectively. Transition *nextTask* represents the assignment of a *reduce* task to a *worker* by the *master*. Transition *master.successJob* represents the successful end of the application and that the result is available to be validated. Transitions *worker.fail.runningMap2* and *worker.fail.runningReduce2* are mutually exclusive because of the place *lastWorker* – only one of these transitions can be executed. Transition *master.successJob* represents the end of the application and that the result is available. Transition *master.failJob* represents the end of the application without an available result, since all *workers* failed. Finally, transitions *worker.stop* and *master.stop* represent the stopping of the online *workers* and *master*, respectively.

PN initial marking consists of one token in the place *begin*, one token in *lastWorker*, and, for the sake of simplicity, we considered only three tokens in *nodes*.

This marking enables only the firing of transition *master.start*. When *master.start* fires, it removes one token from *begin* and one from *nodes*, and adds one token to *master.online*. Transition *worker.start* is then enabled, and at every firing, it removes one token from *nodes* and one from *master.online*. It then returns one token to *master.online* and adds one to *worker.online*. Transition *master.startJob* is enabled only when there are no tokens in *nodes* because there is an inhibitor arc between them. When transition *master.startJob* fires, it removes one token from *master.online*, removes one token from *worker.online*, and adds one token to *worker.runningMap*.

At this point, two transitions can fire, *nextTask* and *worker.fail.runningMap*. If *nextTask* fires, it removes one token from *worker.runningMap* and adds one token to *worker.runningReduce*. If transition *worker.fail.runningMap* fires, it removes one token from *worker.runningMap*, removes one token from *worker.online*, and returns one token to *worker.runningMap*. The marking after *worker.fail.running-Map* fires presents only one token in *worker.runningMap* and one token in *last-Worker*, enabling *worker.fail.runningMap2* to fire.

When *worker.fail.runningMap2* fires, it removes one token from *worker.run-ningReduce*, removes one token from *lastWorker*, and adds one token to *noWorker*. Now, only *master.failJob* is enabled, and when it fires, it removes one token from *noWorker* and adds one token to *master.endApp*. This behavior can occur both with *worker.fail.runningReduce* and *worker.fail.runningReduce2*. However, if *master.successJob* fires, it removes one token from *worker.runningReduce* and adds one token each to *worker.online* and *master.endApp*. Transition *worker.stop* fires sometimes, removing tokens from *worker.online* if it has any. Finally, transition *master.stop* fires to remove one token from *master.endApp* and add one token to *end*, representing the end of the MapReduce execution.

PN extensions also enable us to model behaviors that are implicitly specified in the fault tolerance mechanism, such as the expected *master* behavior when it is impossible to complete a job. Moreover, we intend to use colored and temporal PN extensions to represent other kinds of faults for distributed systems, such as omission, timing, and response faults [31].

3.2. Generating Representative Fault Cases

Representative fault cases are derived from a reachability graph of the PN that models the system fault tolerance mechanism. A reachability graph consists of the set of all possible sequence of transitions firings from a PN. Vertices represent PN markings, and edges represent executed transitions.

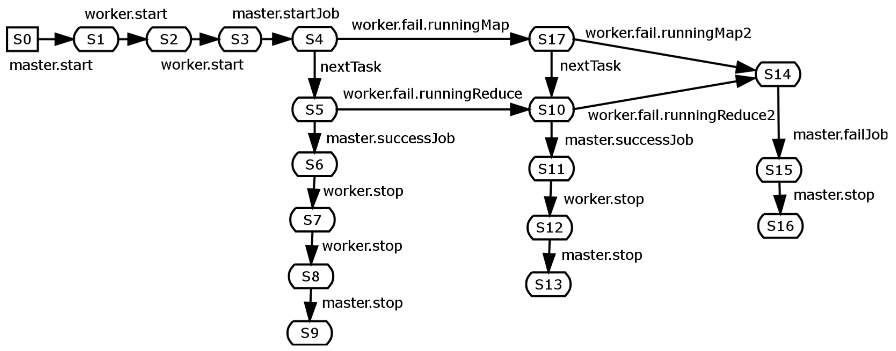


Figure 2: Reachability graph for the PN of Figure 1 with three tokens in the *nodes* place.

Figure 2 shows a reachability graph derived from the PN of the MapReduce fault tolerance mechanism (Figure 1). The graph initial vertex *S0* represents the initial PN marking, with one token in *begin*, three tokens in *nodes*, and one token in *lastWorker*. The *master.start* firing creates the *S1* vertex, the second possible marking of the PN, i.e., one token in *master.online*, two tokens in *nodes*, and one token in *lastWorker*. The *worker.start* firing creates *S2* vertex that enables the *worker.start* to fire, reaching *S3*. The *S3* vertex represents the marking with one token in *master.online* and two token in *worker.online*. The *master.startJob* firing creates the *S4* vertex that represents one token in *worker.runningMap*, one in *worker.online*, and one in *lastWorker*.

From vertex *S4*, two transitions can fire and, consequently, at least two paths are possible. The *nextTask* transition firing creates the *S5* vertex and *worker.fail.runningMap* creates *S17*. Following the *nextTask* path, two transitions can also fire. The *master.successJob* creates *S6*, following through *worker.stop* that creates *S7*, *worker.stop* that creates *S8*, and *master.stop* that creates *S9*. Returning to *S5* vertex, *worker.fail.runningReduce* creates *S10*, following through *master.successJob* that creates *S11*, *worker.stop* that creates *S12*, and *master.stop* that creates *S13*. Other path from the *S10* vertex is the *worker.fail.runningReduce2* transition that creates *S14*, *master.failApp* that creates *S15*, and *master.stop* that creates *S16*. Returning to the *S4* vertex, *worker.fail.runningMap* firing creates *S17* that can follow to *S10* with a *nextTask* firing, or to *S14* with a *worker.fail.running-Map2* firing.

Any change in the PN model requires the creation of a new reachability graph for mapping all new possible transition firings. For example, changing only the number of tokens in the *nodes* place from three to four at the PN in Figure 1, the resulting reachability graph has twenty three vertices instead of eighteen, as shown in Figure 2.

The process to generate fault cases from a PN model consists of traversing all

Table 1: Action list of a fault case example generated from the model.

	h	D	n	T'	W	θ	I
a_0	1	$\emptyset$	1	$\{t_0\}$		9000	$startMaster$
a_1	2	$\{a_0\}$	2	$\{t_1, t_2\}$		1000	$startWorker$
a_2	3	$\{a_1\}$	1	$\{t_0\}$		900000	$startJob$
a_3	3	$\{a_1\}$	1	$\{t_1, t_2\}$	$runningMap$	1000	$failWorker$
a_4	3	$\{a_1\}$	1	$\{t_1, t_2\}$	$runningReduce$	1000	$failWorker$
a_5	4	$\{a_2\}$	1	$\{t_0\}$		10000	$assertResult$
a_6	5	$\{a_0\}$	1	$\{t_0\}$		1000	$stopMaster$

possible paths in a reachability graph, and then mapping each path as the action list $A^{\mathcal{F}}$ of fault case $\mathcal{F}$. The action list $A^{\mathcal{F}}$ is obtained from the edges that connect the graph vertices and from the number of tokens in the places represented by the vertices. The set of testers $T^{\mathcal{F}}$ is obtained from the number of tokens in the *nodes* place from vertex *S0*. The other fault case components ($R^{\mathcal{F}}$ and $\mathcal{O}$) are provided by the testing framework, as described in Section 4.

Returning to the reachability graph shown in Figure 2, we have six possible paths from vertex *S0*. We map the action list of a fault case from the set of transitions of each path, i.e., the path *master.start*, *worker.start*, *worker.start*, *master.startJob*, *worker.fail.runningMap*, *nextTask*, *master.successJob*, *worker.stop*, and *master.stop*. Similarly, as explained in the modeling section, there are transitions that are represented in a reachability graph but are not used to generate an action list. These transitions are not mapped as fault case actions because they represent system events that do not need to be validated. Transitions that compose a fault case are represented by arcs that are initially labeled with *master* and *worker*.

Each fault case action $a_i = (h, D, n, T', W, \theta, I)$ of the action list $A^{\mathcal{F}}$ is mapped as follows. Hierarchical order h is obtained from the numerical sequence of transitions on each path, whereas the actions to start a job and inject a fault have the same h value. Required action set D is obtained from the sequence of transitions. The number of successful action results n is obtained from the marking, i.e., the number of tokens in the places. The tester set T' is obtained from the transition identification and the marking. Instruction I and trigger W also are obtained from the transition label. Time limit θ is set manually, since we do not model temporal characteristics at this time.

Table 1 shows the action list $A^{\mathcal{F}}$ of a generated fault case $\mathcal{F}$ that aims to validate the MapReduce execution if two components fail: one when it executes a *map* task and another when it executes a *reduce* task. Actions $\{a_0, \ldots, a_6\}$ are executed in this sequence, starting with action a_0 and following their hierarchical level (attribute h). However, actions $\{a_2, a_3, a_4\}$ are executed in parallel because they have the same hierarchical level, $h = 3$. Action a_0 that starts the *master* component is only executed by tester t_0. If action a_0 has the result *success* and $D_{a_1} = a_0$, the other testers $\{t_1, t_2, t_3\}$ execute action a_1 to start the *worker* components. Otherwise, if action a_0 has the result *failure*, action a_1 finishes and is assigned the result *failure*. This behavior occurs with all actions that have a dependency relation with an action that has the result *failure*, recursively.

Without failed actions, the process continues and three actions are executed in parallel, $\{a_2, a_3, a_4\}$. Action a_2 is executed by tester t_0, who submits a MapReduce application to the *master*. Concurrently, actions a_3 and a_4 are executed by all testers from $\{t_1, t_2\}$. Both actions a_3 and a_4 inject an interruption fault (*crash*) in the *worker* of each tester. However, only the first tester that executes a *map* task fails, since $n_{a_3} = 1$ and W_{a_3}=*runningMap*. Similarly, the first tester that executes a *reduce* task fails, since $n_{a_4} = 1$ and W_{a_4}=*runningReduce*. Action a_4 is executed by tester t_0, where the system behavior is validated using the instruction *assertResult()*. The *assertResult()* instruction verifies whether the application result is the expected result. If it is, the action result is *success*, otherwise, the action result is *failure*. However, if t_0 is not able to retrieve any results from the system within the time limit $\theta_{a_4} = 1000$, the action result is *timeout*. The *timeout* result also is attributed to all actions that do not finish within their time limit. Finally, action a_6 is executed by tester t_0 to stop the *master* component, since there are no *workers* to stop because both failed during the fault case execution.

If the system behavior while testing is as expected, i.e., all actions results are *success*, the fault case verdict is *pass*. The fault case verdict is *fail* if any action result is *failure*, and is *inconclusive* if neither of the foregoing statements is true.

4. Automating the Execution of Representative Fault Cases

HadoopTest is a testing framework that helps researchers and practitioners automating the execution of representative fault cases. HadoopTest controls the execution of different MapReduce components on several machines, ensures faults are injected in the correct components and circumstances, monitors the system behavior, and validates it by observing whether it behaved as expected. HadoopTest is based on PeerUnit [21], a testing framework for P2P systems [32],that controls the execution of distributed test cases (sequential actions executed by different components). HadoopTest extends PeerUnit to (i) control different MapReduce components, (ii) allow the parallel execution of actions, and (iii) dynamically identify testers that must execute actions according to the status of each MapReduce component.

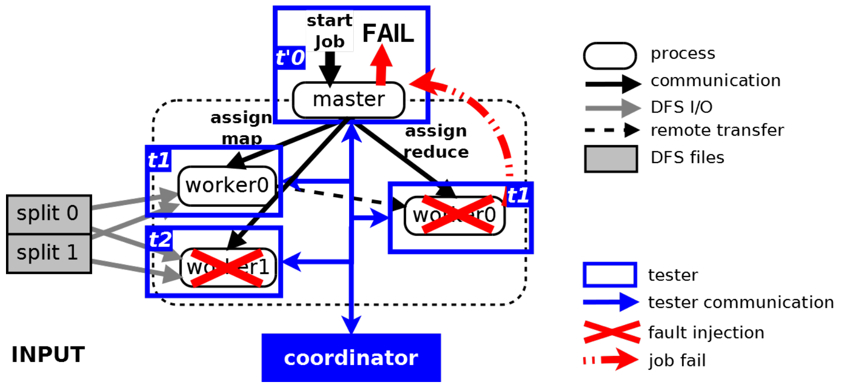


Figure 3: HadoopTest controlling the execution of a fault case.

The HadoopTest architecture consists of two components: a *coordinator* and several *testers*. The *coordinator* controls the execution of distributed testers, coordinates the fault case actions, and generates the fault case verdict based on the testers' results. Each *tester* controls a MapReduce component: it receives coordination messages, executes fault case actions, and returns the results.

Figure 3 represents HadoopTest executing a fault case for which the action list is described in Table 1. This execution involves the control of three MapReduce components: one *master* and two *workers* (*worker0* and *worker1*). The coordinator individually controls the execution of three testers, identified by $\{t'0, t1, t2\}$. Tester $t'0$ controls the *master*, and each tester $\{t1, t2\}$ controls a *worker*. Tester $t'0$ submits a MapReduce job to the *master* that coordinates the execution and assigns *map* and *reduce* tasks to the *workers*. Tester $t2$ injects a fault on *worker1* while it executes a *map* task. Tester $t1$ also injects a fault, but on *worker0* while it executes a *reduce* task. The job cannot be processed and the *master* must return a job fail.

4.1. Action Executions by Testers

A tester follows four steps to execute a fault case action. First, the tester receives a message to execute an action. Second, it waits for the right circumstances to execute the action. Third, it verifies whether it can execute the action. Fourth, the tester executes the instructions or commands that are specified by the action.

Algorithm 1 shows the detail of these related four steps. The process starts when a message is received to execute action a_i. If trigger W_{a_i} is defined, the tester waits for its execution. After the trigger execution or if the trigger is not defined, the tester verifies if the number of required action results n_{a_i} is greater than zero. If it is, the tester executes instruction I_{a_i} and returns the results. Otherwise, the tester returns *failure*, informing the coordinator that it cannot execute the action.

```
Algorithm 1: Fault case action execution
Data: $a_i$, a fault case action.
Output: an action result.
begin
    $a_i \leftarrow$ Receive message to execute an action
    if $W_{a_i} \neq NULL$ then
        Run $W_{a_i}$
    if $n_{a_i} > 0$ then
        return Run $I_{a_i}$
    return $failure$
```

4.2. Fault Case Coordination

The fault case execution consists of coordinating and controlling testers to execute actions in a distributed, parallel, and synchronized way. Algorithm 2 presents the main steps to coordinate testers to execute actions of a fault case $\mathcal{F}$. For each hierarchical level h that exists in action list $A^{\mathcal{F}}$, the coordinator executes three steps: (1) verifies failures in the action dependencies and sends messages to testers to execute the actions in parallel; (2) ensures the action execution time and receives the local action results from the testers; (3) sets action results $R^{\mathcal{F}}$ based on the local results obtained from the testers. After executing the actions at all levels, oracle $\mathcal{O}$ assigns a fault case verdict based on the action results.

```
Algorithm 2: Fault case coordination
Input: $\mathcal{F}$, a fault case; $\mathcal{M}$, a map function between $A^{\mathcal{F}}$ and the hierarchical level of its
        actions.
Data: $n_r$, a number of actions results with success; $R_l$, a local results list for each tester
        after executing its actions.
Output: a verdict.
begin
    foreach $h \in \mathcal{M}(A^{\mathcal{F}})$ do
        $n_r \leftarrow$SendMessages($\mathcal{M}^{-1}(h), R^{\mathcal{F}}$)
        $R_l \leftarrow$ReceiveAnswers($\mathcal{M}^{-1}(h), n_r$)
        $R^{\mathcal{F}} \leftarrow$ProcessResults($\mathcal{M}^{-1}(h), R_l$)
    return $\mathcal{O}(R^{\mathcal{F}})$
```

Algorithm 3 shows the *SendMessages* operation that sends messages to the testers with the actions that each one needs to execute. This operation receives an action list with the same hierarchical level, verifies its relations dependencies, and, if there are no related failure action results, sends messages to the testers to execute the related actions. At the end of the process, *SendMessages* returns the number of action results with success.

The $SendMessages$ input is an action list that must be executed in parallel,

Algorithm 3: SendMessages

```
Input: $A'$, a list of actions at the same hierarchical level; $R^{\mathcal{F}}$, a list of action results.
Output: $n_r$, the number of successful action results.
begin
    $n_r \leftarrow 0$
    foreach $a_i \in A'$ do
        if $R^{\mathcal{F}}[a_j] = success, \forall a_j \in D_{a_i}$ then
            Send message to execute $a_i$ for all $t \in T'_{a_i}$
            $n_r \leftarrow n_r + n_{a_i}$
        else
            $R^{\mathcal{F}}[a_i] \leftarrow failure$
    return $n_r$
```

A', and an actions results list, $R^{\mathcal{F}}$. For each a_i action of A', it is verified if all actions in its dependency relation list, D_{a_i}, have the result success. In such a case, the messages to execute a_i are sent to all testers in T'_{a_i}, and the number of required results, n_r, is updated. Otherwise, i.e., a_i cannot be executed because at least one action in its dependency relation list has the result failure, then the a_i result is *failure*. This process ends by returning the number of required results for the actions in execution by the testers.

Algorithm 4 shows the $ReceiveAnswers$ operation, which consists of the second step to coordinate the testers to execute the actions of a fault case. This operation receives the action results while either the required number of successful executions or the time limit has not yet been reached. When a result is received, the operation checks the required number of successful executions against the related action to verify if the result can be considered, and returns the result list of the testers' local results.

The *ReceiveAnswers* operation receives actions results while either the number of required results or the greater time limit of the actions in execution has not yet been reached. The coordinator identifies which action a_i refers to action result r, and classifies it according to the number of required results. If the number of required results n_{a_i} is smaller than the size of T'_{a_i}, the result is verified. If it is *success*, n_{a_i} is decremented, otherwise the result is neglected. If the number of required results is the same as the size of T'_{a_i}, the local result is stored in the local results list R_t, which is returned for subsequent validation.

Finally, algorithm 5 shows the $ProcessResults$ operation that is the last step for coordinating the execution of actions in parallel. This operation processes the local actions results that were executed in parallel and assigns a single result to each action. All obtained action results comprise the action results list $R^{\mathcal{F}}$ that

Algorithm 4: ReceiveAnswers

```
Input: $A'$, the action list; $n_r$, the number of successful action results.
Output: $R_l$, the local results list of each tester after executing its actions.
begin
    while $(n_r > 0) \wedge (clock < \theta_{a_i}, \forall a_i \in A')$ do
        Receive result $r$ of $t$ and identify its action $a_i$
        if $(n_{a_i} = |T'_{a_i}|)$ then
            $n_r \leftarrow n_r - 1$
            $R_l[t] \leftarrow r$
        else
            if $(n_{a_i} > 0)$ then
                if $(r = success)$ then
                    $n_r \leftarrow n_r - 1$
                    $n_{a_i} \leftarrow n_{a_i} - 1$
                    $R_l[t] \leftarrow r$
    return $R_l$
```

is returned to the fault case execution. The action results list is used to verify the action dependency relations and generate the fault case verdict.

The *ProcessResults* operation starts by checking if the local results of all testers are *success*, i.e., if the number of required results is zero. If so, the action result is also *success*. If any local result is *failure*, the action result is also *failure*. However, if the operation has not received any result by the time limit, the action result is *timeout*.

4.3. HadoopTest Implementation

HadoopTest provides an interface for testing three Hadoop versions, but can be adapted to test other MapReduce systems or other distributed systems. HadoopTest is open-source and available by request from the authors. HadoopTest is implemented in Java and makes intensive use of dynamic reflection and annotations. It uses these characteristics to select and execute the fault case actions.

A fault case $\mathcal{F}$ is implemented as a Java class, where methods marked with the notation @*TestStep* compose the action list $A^{\mathcal{F}}$. The notation @*TestStep* has the following attributes: *Order* is the element h_{a_i}, a positive integer, and defines the a_i hierarchical level. *Depend* is the element D_{a_i}, an optional string, composed of an identifier's action list (methods with notation @*TestStep*) that must be successfully executed before a_i. *Answers* is the element n_{a_i}, an optional positive integer, that defines the number of successful required results; *Range* is the element T'_{a_i}, a string composed of a testers' identifier list (positive integers) separated by comma,

Algorithm 5: ProcessResults

```
Input: $A'$, an action list; $R_l$, a local result list of each tester after executing its actions.
Output: $R^{\mathcal{F}}$, an action results list.
begin
    foreach $a_i \in A'$ do
        if $n_{a_i} = 0$ then
            $R^{\mathcal{F}}[a_i] \leftarrow success$
        if $R_l[t] = success, \forall t \in T'_{a_i}$ then
            $R^{\mathcal{F}}[a_i] \leftarrow success$
        else
            if $\exists r \in R_l[t] : r = failure, \forall t \in T'_{a_i}$ then
                $R^{\mathcal{F}}[a_i] \leftarrow failure$
            else
                $R^{\mathcal{F}}[a_i] \leftarrow timeout$
    return $R^{\mathcal{F}}$
```

or a positive integer range (e.g., "1–3"), or an asterisk, for which all testers execute a_i; *When* is the element W_{a_i}, a string composed of a command that must be executed to enable a_i execution; *Timeout* is the element θ_{a_i}, a positive integer interpreted as milliseconds that is the time limit for a_i execution.

4.4. Writing a Fault Case

HadoopTest simplifies the description of a fault case, only requiring a description of its action list $A^{\mathcal{F}}$ that is a Java class with methods with the annotation *@TestStep*. Listing 1 shows a part of the Java class *FaultCase1* that describes the action list of the representative fault case described in Table 1.

Class *FaultCase1* is a subclass of *AbstractMR* that implements the MapReduce library and provides the methods that abstract the complexity of the MapReduce manipulation, such as method *startWorker* that starts a *worker* component. The starting and stopping of MapReduce components are implemented using Hadoop *scripts*. Scripts *runningMap.sh* and *runningReduce.sh* were implemented using the Hadoop log file. Each script waits until a specific string occurs. Method *failWorker* injects an interruption fault and is implemented by the *kill* bash command. Method *assertResult* validates the application result by comparing it with the expected result.

The first method is *a0* and describes a fault case action, since it has the notation *@TestStep*. The purpose of the method is to start the component *master*, and the notation attributes indicate that the method *a0* must: be initially executed (*order = 1*), has no dependency relation (*depend* = " "), have one successful result (*answers*

= *1*), be executed by tester t_0 (*range* = *0*), has no trigger (*when* = *“”*), and be executed until time 100 (*timeout* = *100*).

```
public class FaultCase1 extends AbstractMR{
 @TestStep(order=1, depend="", answers=1, range="0", when="", timeout=100)
 public void a0() {     startMaster();   }
 @TestStep(order=2, depend="a0", answers=3, range="1-2", when="", timeout=1000)
 public void a1() {    startWorker();   }
 @TestStep(order=3, depend="a1", answers=1, range="0", when="", timeout=900000)
 public void a2() {    startJob();   }
 @TestStep(order=3, depend="a1", answers=1, range="1-2",
      when="runningMap.sh", timeout=1000)
 public void a3() {    failWorker();   }
 @TestStep(order=3, depend="a1", answers=1, range="1-2",
      when="runningReduce.sh", timeout=1000)
 public void a4() {    failWorker();   }
 @TestStep(order=4, depend="a2", answers=1, range="0", when="", timeout=10000)
 public void a5() {    assertResult();   }
 @TestStep(order=5, depend="a0", answers=1, range="0", when="", timeout=1000)
 public void a6() {    stopMaster();   }
}
```

Listing 1: First part of class *FaultCase1*.

The second method of class *FaultCase1* is *a1*. The purpose of this method *a1* is to start a *worker* component, and the attributes of its annotation indicate that method *a1* must: be executed after the previous method (*order* = *2*), be executed only if action *a0* is executed successfully (*depend* = *“a0”*), have three success results (*answers* = *3*), be executed by testers t_1, t_2, t_3 (*range* = *“1–3”*), has no trigger (*when* = *“”*), and be executed up to a time limit of 1000 (*timeout* = *1000*).

The following methods, *a2*, *a3*, and *a4*, must be executed in parallel because they have the same hierarchical level (*order* = *3*) and after the successful execution of method *a1* (*depend* = *a1*). Method *a2* sends an application, and must be executed by tester t_0, who controls the *master*. Method *a3* injects a fault on the *worker*, and must be executed by the first tester that executes trigger *runningMap.sh*. Method *a4* injects a fault on the *worker* and must be executed by the first tester that executes trigger *runningReduce.sh*.

Method *a5* validates the application result, and must be executed by tester t_0, who controls the *master*, and after the successful execution of method *a2* (*depend* = *a2*) . Method *a6* stops the component *master* and must be executed by tester t_0.

4.5. Running a Fault Case

To run a fault case in HadoopTest, is necessary for Hadoop to already be configured on the set of machines to be used during the test. In addition, the HadoopTest configuration files (*peerunit.properties* and *hadoop.properties*) must be adequately updated. The next step is to run the HadoopTest *coordinator* and one *tester* in the machine in which the Hadoop master component was configured. The test machines then run the HadoopTest tester on each of the other machines and wait for the completion of the fault case execution. The fault case verdict is presented by the coordinator at the standard output and in the HadoopTest log file.

5. Experimental Results

This section presents the experimental results for testing Hadoop[7] using our two main contributions: PbGen, an approach to generate representative fault cases, and HadoopTest, a testing framework to automate the execution of fault cases. The results are described and grouped according to six testing framework requirements: controllability, time measurement, non-intrusiveness, repeatability, reproducibility, and efficacy.

5.1. Cluster Setup and Workload

In a previous study [1], initial experiments were presented using two hundred machines on the Grid'5000 [1], demonstrating the scalability and time non-intrusiveness of HadoopTest. In this study, we conducted experiments on a blade with five nodes (virtual machines), to remove potential interference or network connectivity problems. Each node has two virtual cores, 2 GB memory, and a 50 GB disk. All nodes ran Ubuntu 12.04.4 LTS (Precise Pangolin), Java 1.6.0_45 64 bit, Hadoop-0.22.0 (*hadoop0*), Hadoop-1.2.1 (*hadoop1*), and Hadoop-2.2.0 (*hadoop2*) with default configuration parameters.

Hadoop-2 splits into separated entities the two major facilities of the MapReduce *master* component, (i) a *resource manager* to handle the use of resources across the cluster, and (ii) an *application master* to deal with the life-cycle of jobs running on the cluster. Therefore, despite of improving scalability and resource management, Hadoop-2 demands at least three components (*i*, *ii* and a *worker*) to execute a job – little different from a *master* and a *worker* of Hadoop-1.

The workload was composed of a 5 MB file with network traffic logs and the *WordCount* MapReduce application, which calculates the number of distinct words in files. *WordCount* is bundled with all Hadoop versions and has been broadly adopted to analyze network traffic and logs [33, 34, 35]. The *map* function receives as input the filename and its content, and for each word, the function outputs a pair composed of the word and the numeral one (1). The *reduce* function

[1]https://www.grid5000.fr

Table 2: Fault distribution by execution and Hadoop version.

execution	**hadoop0**†	**hadoop1**†	**hadoop2**†
KillMap1	*worker3*	*worker3*	*worker2*
KillMap2	*worker2*	*worker2*	*worker2*
KillMap3	*worker1*	*worker2*	*worker3*
KillReduce1	*worker1*	*worker2*	*worker2*
KillReduce2	*worker3*	*worker3*	*worker2*
KillReduce3	*worker1*	*worker2*	*worker2*

†All executions received a *PASS* verdict.

sums the values found for each word and outputs a pair composed of the word and its number of occurrences.

5.2. Controllability

Testing a MapReduce system requires a framework that individually controls all distributed components to determine in which of them (where) and in which state (when) the component is. The where and when data are only available at running time because Hadoop schedules *map* and *reduce* tasks differently at each execution.

Table 2 shows the Hadoop components in which HadoopTest injected faults at each fault case execution, considering two fault cases and three Hadoop versions. None of the executions have errors, consequently, they received a *PASS* verdict, showing that the three Hadoop versions tolerated the injected faults. The *KillMap1* refers to the first execution of the *KillMap* fault case that aims to validate the Hadoop execution with three *workers* and while a *worker* fails when it executes a *map* task. The *worker1* failed on *hadoop0*, and *worker2* failed on *hadoop1* and *hadoop2*. The behavior is similar for the other executions of *KillMap*, except for *hadoop2* on *KillMap2* where *worker2* failed. A similar behavior occurs when executing the *KillReduce* fault case. In the first execution, *worker2* failed on all hadoop versions. The *worker2* also failed for all executions on *hadoop2*. However, *worker1* failed in the second and third *KillReduce* executions on *hadoop0*, and *worker1* failed in the third execution on *hadoop1*.

Table 3 also shows the Hadoop components in which HadoopTest injected faults for each fault case execution. However, in this case, the fault cases involved three *workers* and a fault injection on two of them. The *KillMapReduce* fault case aims to validate the Hadoop execution with three *workers*. In this case, two *workers* fail, one when it executes a *map* task and another when it executes a *reduce* task. The *2KillMap* fault case aims to validate the Hadoop execution with two *workers* that fail when they execute a *map* task. In addition to the different

Table 3: Task distribution of different executions and Hadoop versions.

execution	**hadoop0**†	**hadoop1**†	**hadoop2**‡
KillMapReduce1	*worker1, worker3*	*worker1, worker2*	*worker1, worker2*
KillMapReduce2	*worker1, worker2*	*worker1, worker3*	*worker1, worker2*
KillMapReduce3	*worker1, worker3*	*worker2, worker3*	*worker2, worker3*
2KillMap1	*worker1, worker2*	*worker2, worker3*	*worker3, worker2*
2KillMap2	*worker1, worker3*	*worker2, worker3*	*worker1, worker3*
2KillMap3	*worker2, worker3*	*worker1, worker3*	*worker2, worker3*

†All executions received a *PASS* verdict. ‡All executions received a *FAIL* verdict.

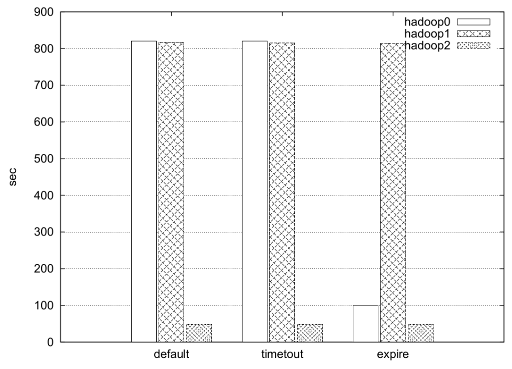
distribution of task and faults that occurred on the executions, the Hadoop versions presented different job results. The *hadoop0* and *hadoop1* executions did not presented errors, achieving a *PASS* verdict and showing that they tolerated the injected faults. However, the *hadoop2* executions achieved a *FAIL* verdict, because it stayed without *workers* to execute the job after HadoopTest failed its two *workers*.

Although the *hadoop2* executions received a *FAIL* verdict for the *KillMapReduce* and *2KillMap* fault cases, the behavior was expected, requiring us to execute a different fault case. We executed the *2KillMapWaitJobFail* fault case that fails two *workers* when they execute a *map* task, and validates if the *master* assigns a fail job when there is no *worker* available. We executed *2KillMapWaitJobFail* on four nodes to test *hadoop2*, and this received a *PASS* verdict on all fault case executions. The *hadoop0* and *hadoop1* were tested considering the *2KillMapWaitJobFail* fault case, but with three nodes (keeping only the *master* running after *workers* failed). Both *hadoop0* and *hadoop1* executions received a *FAIL* verdict because they did not interrupt the job execution for ten hours, at which point the HadoopTest interrupted the fault case by timeout.

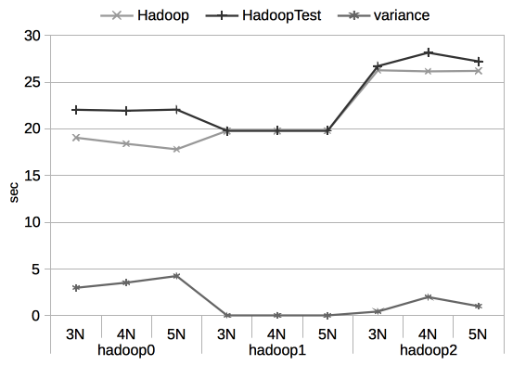
The Hadoop versions scheduled tasks differently, considering the different job executions, and HadoopTest followed this behavior, controlling the fault case execution accordingly. Moreover, HadoopTest injected faults according the status of each component and validated the Hadoop behavior correctly.

5.3. Time Measurement

HadoopTest is able to test Hadoop considering its temporal information, either to evaluate component behaviors or the overall system status. For example, HadoopTest is able to evaluate the detection latency of a failed component or use a time-related parameter. Note that when a time parameter is altered and the system does not follow the specified actions, the user can attribute the problem to a network error. System testing enables us to find, identify, and correct errors, improving the system development and deployment as well as enhancing the network service reliability.



(a)



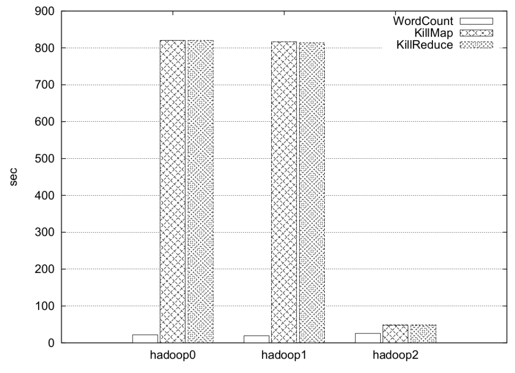
(b)

Figure 4: *KillMap* fault case runtimes for three Hadoop versions with different parameters (a) and Hadoop and HadoopTest runtimes running on three, four, and five nodes (b).

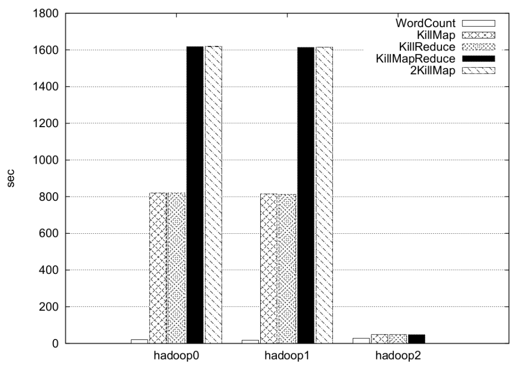
To evaluate whether Hadoop reschedules faulty tasks within the specified fault detection time, we set the *mapreduce.task.timeout* (*timeout*) and *mapreduce.job-tracker.expire.trackers.interval* (*interval*) attributes to 60 s. We then executed a *KillMap* fault case and present the runtimes obtained in Figure 4a. The *timeout* parameter did not change the runtime of any hadoop version. The fault case verdict was *FAIL* for *hadoop0* and *hadoop1*, indicating that these versions did not reschedule a faulty task within the specified detection time. However, the *interval* parameter was considered by *hadoop0* but not by *hadoop1*, which continued to not reschedule a faulty task within the specified detection time.

Figures 5a and 5b show the average runtimes of different fault cases running on three Hadoop versions. The *WordCount* was a job execution controlled by HadoopTest, but without fault injection. Other fault cases are as described in Section 5.2. Figure 5a shows the runtimes considering three nodes, and Figure 5b shows the runtimes considering four nodes. Both figures show a large difference between the executions with and without faults on *hadoop0* and *hadoop1* compared to *hadoop2*. In addition, to demonstrate the effectiveness of injecting faults, the differences enable us to evaluate the latency needed to identify a failed *worker*. On *hadoop0* and *hadoop1*, the latency is about 800 s, while on *hadoop2* the latency is about 20 s.

We also evaluated the latency needed to identify a faulty job, i.e., when the *master* identifies that all *workers* have failed. We set the timeout to 3600 s for the *sendJob* action of the *2KillMapWaitJobFail* fault case. The *hadoop0* and *hadoop1* achieved an *INCONCLUSIVE* verdict, because both executions were interrupted by HadoopTest without a *master* answer.



(a)



(b)

Figure 5: Fault case runtimes for three Hadoop versions and three nodes (a), and four nodes (b).

5.4. Non-intrusiveness

HadoopTest does not require the alteration of the Hadoop source code to execute fault cases. HadoopTest uses Hadoop scripts to start and stop its components, and uses *bash* scripts to analyze *logs* to activate the fault injection. HadoopTest also uses the *kill* bash command to inject interruption faults. This approach, instead of instantiating objects and altering the source-code [36], improves performance and maintainability, and allows us to test other Hadoop versions or Hadoop-based systems without worrying about implementation details.

HadoopTest minimal overhead and scalability were shown in a previous work[1]. HadoopTest increased the Hadoop runtime in less than 2 seconds, controlling a distributed system running up to 200 nodes. Moreover, all results obtained for Hadoop executions were equal to those obtained through HadoopTest.

Here, we present a twofold new evaluation for HadoopTest in a blade environment. First, we executed Hadoop with the default configuration parameters to establish an execution time baseline. Second, we executed Hadoop using HadoopTest to evaluate the overhead produced during a fault case execution. We varied the hadoop versions (*hadoop0*, *hadoop1*, and *hadoop2*), and tested from three to five nodes (*3N* to *5N*). Again, all results obtained using Hadoop were equal to those obtained through HadoopTest, and Figure 4b shows the average execution times of *WordCount* running directly on Hadoop and through HadoopTest. HadoopTest overloaded *hadoop0* between 3 and 4 s, *hadoop2* between 0.5 and 2 s, and *hadoop1* only by 0.01–0.03 s. Moreover, the standard deviation was lower than 2 s, confirming that HadoopTest has minimal runtime intrusiveness that can even be lower for some systems, as was the case for *hadoop1*.

5.5. Repeatability and Reproducibility

HadoopTest presents high repeatability and reproducibility because all fault cases executions that we presented are accurately repeatable and reproducible. HadoopTest is able to repeat the execution that reveals an error, making it possible to identify and remove the defect that generates the error. In spite of the fact that the tests were executed in LAN inside a blade architecture, we repeated them at least three times for all proposed fault cases, and we obtained the same quantitative results for all executions.

5.6. Efficacy

All fault cases generated from the model of the MapReduce fault tolerance mechanism are considered representative for testing because they inject a fault only in a component that is running a MapReduce task. PbGen has high efficacy because it only generates representative fault cases. HadoopTest is able to test Hadoop when executing representative fault cases and evaluate its reliability and availability. Furthermore, the presented approach is efficacious because errors were identified in Hadoop using the HadoopTest execution of the representative fault cases.

6. Conclusion

We exposed and analyzed issues related to testing MapReduce fault tolerance based on generating and executing representative fault cases. We generated (derived) the representative fault cases from the reachability graph of a PN model for the MapReduce fault tolerance mechanism. The fault case executions were automated by HadoopTest, the proposed testing framework that individually controls and monitors each Hadoop distributed component, ensures the fault injection on specified components and their status, and validates the system behavior according to the expected behavior. Experimental results showed that the generated fault cases were representative for testing Hadoop, allowing us to identify some Hadoop errors. We demonstrated that HadoopTest presents the required properties for a testing framework: controllability, time measurement, non-intrusiveness, repeatability, and efficacy. HadoopTest uses a simplified fault case description and is freely available to be used and adapted to test other systems. Furthermore, the testing method obtained, as a byproduct, the network reliability, which will ease the network services development and deployment. We intend to automate the fault tolerance testing as well as the generation and execution of the representative fault cases derived from the PN model. We also intend to enrich the fault tolerance mechanism model and test other fault-tolerant systems by applying the presented method.

7. References

[1] J. E. Marynowski, A. R. Pimentel, T. S. Weber, A. J. Mattos, Dependability Testing of MapReduce Systems, in: Proc. of the ICEIS - Intl. Conf. on Enterprise Information Systems, SciTePress, 2013, pp. 165–172.

[2] J. Dean, S. Ghemawat, MapReduce: Simplified Data Processing on Large Clusters, in: Proc. of the OSDI - Symp. on Operating Systems Design and Implementation, USENIX, 2004, pp. 137–149.

[3] A. Avizienis, J.-C. Laprie, B. Randell, C. E. Landwehr, Basic Concepts and Taxonomy of Dependable and Secure Computing, IEEE Transactions on Dependable and Secure Computing 1 (1) (2004) 11–33.

[4] P. Ammann, J. Offutt, Introduction to Software Testing, Cambridge University Press, 2008.

[5] K. Echtle, M. Leu, Test of Fault Tolerant Distributed Systems by Fault Injection, in: Proc. of the FTPDS - Workshop on Fault-Tolerant Parallel and Distributed Systems, IEEE, 1994, pp. 244–251.

[6] A. M. Ambrosio, F. Mattiello-Francisco, N. L. Vijaykumar, S. V. de Carvalho, V. Santiago, E. Martins, A Methodology for Designing Fault Injection Experiments as an Addition to Communication Systems Conformance Testing, in: Proc. of the Workshop on Dependable Software - Tools and Methods in the DSN, IEEE, 2005, pp. 1–6.

[7] Apache, The Apache Hadoop, http://hadoop.apache.org/.
URL `http://hadoop.apache.org/`

[8] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, A. Rasin, HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads, Proceedings of the VLDB Endowment 2 (1) (2009) 922–933.

[9] A. Sangroya, D. Serrano, S. Bouchenak, Benchmarking Dependability of MapReduce Systems, in: Proc. of the SRDS - Symp. on Reliable Distributed Systems, IEEE, 2012, pp. 21–30.

[10] S. Bernardi, J. Merseguer, D. C. Petriu, Dependability Modeling and Assessment in UML-Based Software Development, The Scientific World Journal 2012 (1) (2012) 1–11.

[11] G. Jacques-Silva, R. Drebes, J. Gerchman, J. F. Trindade, T. Weber, I. Jansch-Porto, A Network-Level Distributed Fault Injector for Experimental Validation of Dependable Distributed Systems, in: Proc. of the COMPSAC - Intl. Computer Software and Applications Conference, IEEE, 2006, pp. 421–428.

[12] R. Lefever, K. Joshi, M. Cukier, W. Sanders, A Global-State-Triggered Fault Injector for Distributed System Evaluation, IEEE Transactions on Parallel and Distributed Systems 15 (7) (2004) 593–605.

[13] A. Benso, A. Bosio, S. D. Carlo, R. Mariani, A Functional Verification Based Fault Injection Environment, in: Proc. of the DFT - Intl. Symp. on Defect and Fault-Tolerance in VLSI Systems, IEEE, 2007, pp. 114–122.

[14] T. D. Chandra, R. Griesemer, J. Redstone, Paxos Made Live: An Engineering Perspective (2006 Invited Talk), in: Proc. of the PODC - Symp. on Principles of Distributed Computing, ACM, 2007, pp. 398–407.

[15] A. Henry, Cloud Storage FUD: Failure, Uncertainty and Durability (Keynote Address), in: Proc. of the FAST - Symp. on File and Storage Technologies, USENIX, 2009, pp. 1–33.

[16] P. Joshi, H. S. Gunawi, K. Sen, PREFAIL: A Programmable Tool for Multiple-Failure Injection, in: Proc. of the OOPSLA - Conf. on Object-Oriented Programming, ACM, 2011, pp. 171–188.

[17] C. Fu, B. G. Ryder, A. Milanova, D. Wonnacott, Testing of JavaWeb Services for Robustness, in: Proc. of the ISSTA - Intl. Symp. on Soft. Testing and Analysis, ACM, 2004, pp. 23–33.

[18] X. Pan, J. Tan, S. Kavulya, R. Gandhi, P. Narasimhan, Ganesha: Black-Box Diagnosis of

MapReduce Systems, ACM SIGMETRICS Performance Evaluation Review 37 (3) (2010) 8–13.

[19] B. Butnaru, F. Dragan, G. Gardarin, J. Manolescu, B. Nguyen, R. Pop, N. Preda, L. Yeh, P2PTester: A Tool for Measuring P2P Platform Performance, in: Proc. of the ICDE - Intl. Conf. on Data Engineering, IEEE, 2007, pp. 1501–1502.

[20] Z. Zhou, H. Wang, J. Zhou, L. Tang, K. Li., Pigeon: A Framework for Testing Peer-to-Peer Massively Multiplayer Online Games over Heterogeneous Network, in: Proc. of the CCNC - Consumer Communications and Networking Conf., IEEE, 2006, pp. 1028–1032.

[21] E. C. de Almeida, J. E. Marynowski, G. Sunyé, P. Valduriez, PeerUnit: A Framework For Testing Peer-to-Peer Systems, in: Proc. of the ASE - Intl. Conf. on Automated Software Engineering, ACM, 2010, pp. 169–170.

[22] C. Pham, D. Chen, Z. Kalbarczyk, R. K. Iyer, CloudVal: A Framework for Validation of Virtualization Environment in Cloud Infrastructure, in: Proc. of the DSN - Intl. Conf. on Dependable Systems and Networks, IEEE, 2011, pp. 189–196.

[23] W. Hoarau, S. Tixeuil, F. Vauchelles, FAIL-FCI: Versatile Fault Injection, Future Generation Computer Systems 23 (7) (2007) 913–919.

[24] Apache, Large-Scale Automated Test Framework - Herriot, https://issues.apache.org/jira/browse/HADOOP-6332.
URL `https://issues.apache.org/jira/browse/HADOOP-6332`

[25] C. Csallner, L. Fegaras, C. Li, New Ideas Track: Testing MapReduce-Style Programs, in: Proc. of the ESEC/FSE - SIGSOFT Symp. and the European Conf. on Foundations of Software Engineering, ACM, 2011, pp. 504–507.

[26] J. Tan, X. Pan, S. Kavulya, R. Gandhi, P. Narasimhan, Mochi: Visual Log-Analysis Based Tools for Debugging Hadoop, in: Proc. of the HotCloud - Conf. on Hot Topics in Cloud Computing, USENIX, 2009, pp. 1–5.

[27] S. Huang, J. Huang, J. Dai, T. Xie, B. Huang, The HiBench Benchmark Suite: Characterization of the MapReduce-Based Data Analysis, in: Proc. of the ICDEW - Intl. Conf. on Data Engineering Workshops, IEEE, 2010, pp. 41–51.

[28] T. Murata, Petri nets: Properties, Analysis and Applications, Proceedings of the IEEE 77 (4) (1989) 541–580.

[29] G. Callou, P. Maciel, D. Tutsch, J. Araújo, A Petri Net-Based Approach to the Quantification of Data Center Dependability, in: Petri Nets - Manufacturing and Computer Science, InTech, 2012, Ch. 14, pp. 313–336.

[30] J. E. Marynowski, Towards Dependability Testing of MapReduce Systems, in: Proc. of the IPDPS - Intl. Parallel and Distributed Processing Symp., IEEE, 2013, pp. 2282–2285.

[31] F. Cristian, Understanding fault-tolerant distributed systems, Communications of the ACM 34 (2) (1991) 56–78.

[32] S. Androutsellis-Theotokis, D. Spinellis, A Survey of Peer-to-Peer Content Distribution Technologies, ACM Computing Surveys 36 (4) (2004) 335–371.

[33] S. Lee, K. Levanti, H. S. Kim, Network monitoring: Present and future, Computer Networks 65 (June) (2014) 84–98.

[34] R. Fontugne, J. Mazel, K. Fukuda, Hashdoop : A MapReduce Framework for Network Anomaly Detection, in: Proc. of the BigSecurity2014 - Intl. Workshop on Security and Privacy in Big Data, 2014, p. 6.

[35] B. B. Madan, M. Banik, Attack Tolerant Architecture for Big Data File Systems, ACM SIGMETRICS Performance Evaluation Review 41 (4) (2014) 65–69.

[36] J. E. Marynowski, M. Albonico, E. C. de Almeida, G. Sunyé, Testing MapReduce-Based Systems, in: Proc. of the SBBD - Brazilian Symp. on Databases, 2011, pp. 9–16.