Universidade Federal do Paraná

Departamento de Informática

João Eugenio Marynowski and Andrey Ricardo Pimentel

HadoopTest:
A Dependability Testing Framework For Hadoop

Relatório Técnico
RT_DINF 002/2013

Curitiba, PR
2013

HadoopTest: A Dependability Testing Framework For Hadoop

(A Practical Experience Report)

João Eugenio Marynowski and Andrey Ricardo Pimentel
Department of Informatics
Federal University of Paraná
Curitiba, Brazil 81531-980
{jeugenio,andrey}@inf.ufpr.br

***Abstract*—Hadoop is a popular MapReduce implementation used to analyze large data sets distributed over a huge number of machines. Fault tolerance is a key feature of Hadoop, and it is essential to execute representative fault cases for testing the Hadoop's dependability. The fault case execution is a hard task that must be automated since it involves controlling and monitoring all the Hadoop's distributed components, and injecting faults according to the components processing steps. To address this problem, we present HadoopTest, a dependability testing framework freely available. HadoopTest automates the execution of fault cases in a real deployment scenario and validates the Hadoop's behavior. We show the adequacy and efficacy of HadoopTest by executing representative fault cases and identifying Hadoop bugs.**

Keywords*—*Hadoop; MapReduce; Dependability; Test; Fault Injection

I. Introduction

Hadoop is an open-source implementation of MapReduce, that is a simplified programming model and the associated implementation for processing and analyzing large-scale data [1]. A growing number of companies uses it to analyse the data generated by several applications, such as social networks, data mining, machine learning, log processing, and business intelligence [2]. Hadoop allows one to use a huge number of machines easily for the distributed data processing. As any other large-scale system, Hadoop faces faults frequently due to several different conditions, e.g., outages, hardware problems, and bugs [3]. It is designed to detect and handle those faults; however, it is necessary to test Hadoop to ensure its dependability.

Dependability testing aims at validating the behavior of fault tolerant systems, i.e., it aims at finding errors in the implementation or specification of fault tolerant mechanisms [4], [5]. For this purpose, the system is executed on a controlled testing environment with the injection of artificial faults. A fault case is the set of requirements for a complete execution and validation of a system under test while faults are injected[6], [7]. Two main issues concerning the dependability testing are: (1) generate representative elements from the potentially infinite and partially unknown set of possible fault cases and (2) automate their executions.

In [8], [9] we show how to generate representative fault cases from a Petri Net model of the Hadoop's fault tolerance mechanism. The generated fault cases involve the controlling and monitoring each distributed Hadoop's component, the fault injection according to the components processing steps, and the validation of the Hadoop's behavior.

In this work, we describe HadoopTest, a dependability testing framework that automates the execution of representative fault cases in a real deployment scenario. We show how HadoopTest controls and monitors all Hadoop's distributed components individually. It injects faults according to their processing steps and validates the system behavior. We show the HadoopTest adequacy by executing some representative fault cases and identifying some Hadoop bugs.

The remainder of this paper is organized as follows. The next session introduces basic concepts, presenting a description of Hadoop and defining a fault case. Section II-C presents the representative fault cases. Section III presents our framework for the dependability testing. Section IV describes the results through implementation and experimentation. Section V surveys related work. Section VI concludes.

II. Basic Concepts

Hadoop offers a programming environment based on two high-level functions, *map* and *reduce*, and a runtime environment to execute them on a cluster. The Hadoop architecture includes several *TaskTracker* components, and one *JobTracker* that schedules *map* and *reduce* tasks to run at the *TaskTrackers*.

The Hadoop fault tolerance mechanism identifies faulty *TaskTracker* by timeout, and reschedules their tasks to another healthy *TaskTracker*. The fault handling differs between tasks and their processing steps, e.g., if a *TaskTracker* fails when it is executing a *map* task, the *JobTracker* only reschedules its task for another *TaskTracker*; but if a component fails after executing a *map* task, the *JobTracker* reschedules the task for another *TaskTracker* and informs all *TaskTrackers* executing *reduce* tasks that they must read the *map* result from the new *TaskTracker*.

Testing the dependability of Hadoop requires to validate its behavior stimulating its tolerated faults, which by turn requires explicit control over all its components and their processing steps. A fault case is a set of components required for a complete execution and dependability validating of a system.

A. Representative Fault Cases

The representativeness of a fault case is how important it is to identify bugs on a system under test [10], [11]. We consider the representative fault cases for the dependability testing of Hadoop as the generated through an abstraction of its fault tolerance mechanism. This approach is successfully used by other systems [6], [7], [12], and guides the generation to a finite set of actually tolerated fault cases, and that must be tested to ensure the system dependability.

In [8], we show how to model the Hadoop fault tolerance mechanism using Petri Nets (PN) [13]. We modeled the Hadoop components as dynamic items, to be easily inserted and removed, and to model the independence of these components with their actions and states, i.e., an action can be executed by any enabled component.

The representative fault cases obtained were composed by a set of actions that must be executed in parallel, by a set of components that are instantiated dynamically on run-time. Therefore, the fault case execution involve the controlling and monitoring of each distributed Hadoop's component. The fault injection according to the components processing steps, and the validation of the Hadoop's behavior.

B. Definitions

Definition 2.1 (Fault Case): A fault case is a 4-tuple $\mathcal{F} = (C^{\mathcal{F}}, A^{\mathcal{F}}, R^{\mathcal{F}}, \mathcal{O})$ where: $C^{\mathcal{F}} = \{c_0, c_1, \ldots, c_n\}$, and it is a list of system components; $A^{\mathcal{F}} = \{a_0, a_1, \ldots, a_m\}$, and it is a list of actions that can involve fault injections; $R^{\mathcal{F}} = \{r_{a_0}, \ldots, r_{a_m}\}$, and it is a list of action results; and $\mathcal{O}$ is an oracle.

The oracle is a mechanism responsible for verifying the system behavior during a fault case execution, and associating its result, i.e., a verdict *pass*, *fail* or *inconclusive*. Each action execution can get the result: *success*, *failure*, or *timeout* (without response during a time limit). If all action executions get *success*, the $\mathcal{F}$ verdict is *pass*. If any action execution gets *failure*, the $\mathcal{F}$ verdict is *fail*. But if at least one action execution gets *timeout*, the $\mathcal{F}$ verdict is *inconclusive*, making the test inaccurate for assigning some of the earlier statements and, moreover, it is necessary to re-run the fault case.

Definition 2.2 (Fault Case Action): A fault case action is a 7-tuple $a_i = (h, D, n, C', I, W, t)$ where:

- $h \in \mathbb{N} | h \leqslant |A|$, and it is an hierarchical order in which action a_i must execute - actions with same h execute in parallel;
- $D \subseteq A | \forall a_j \in D : r_{a_j} = success$, and it is a set of actions that must be successfully executed before a_i, otherwise the action result r_{a_i} is *failure*;
- $n \in \mathbb{N} | n \leqslant |C'|$, and it is the number of success action results, i.e., the number of action executions to result *success* for a_i;
- $C' \subseteq C$, and it is a set of components that execute a_i;
- I is a set of instructions or commands executed by the components;
- W is an optional instruction or command that is a trigger required to execute a_i;
- t is a time to execute a_i.

C. A Representative Fault Case Example

Table I shows an example of a representative fault case. The goal is to validate the MapReduce execution while two components fail, one when executing a *map* task and another when executing a *reduce* task. This fault case involves four components $C^{\mathcal{F}} = \{c_0, \ldots, c_3\}$ and eight actions $A^{\mathcal{F}} = \{a_0, \ldots, a_7\}$. The component c_0 executes the action a_0 to start the *JobTracker*. If action a_0 succeeds, the components $\{c_1, c_2, c_3\}$ execute the action a_1 to start the *TaskTrackers*. Otherwise, the action a_1 finishes and receives the *failure* result. This occurs with all actions that have a dependency relation with a failed action, recursively. Without failed actions, the process continues, and the component c_0 executes a_2 that submits a job. During the job execution: (1) only the first component ($n_{a_3} = 1$) of $\{c_1, c_2, c_3\}$ fails when it executes the *map* task ($W_{a_3} = runningMap()$), and (2) only the first component ($n_{a_4} = 1$) of $\{c_1, c_2, c_3\}$ fails when it executes the *reduce* task ($W_{a_4} = runningReduce()$). At action a_5, c_0 validates the job result, comparing the expected with the obtained. The next actions, a_6 and a_7, stop the Hadoop execution.

III. HadoopTest

HadoopTest is a test framework to help researchers and practitioners to execute fault cases automatically. It extends the PeerUnit testing framework [14]. HadoopTest controls and monitors all Hadoop's distributed components. It injects faults according to their processing steps and validates the system behavior.

The HadoopTest architecture consists of one *coordinator* and several *testers*. The *coordinator* controls the execution of distributed testers. It coordinates the actions of fault cases and generates the verdict from tester results. Each *tester* receives coordination messages, executes fault case actions in MapReduce components, and returns their results.

Figure 1 shows HadoopTest running a fault case example for the dependability testing of Hadoop. The *coordinator* individually controls the execution of four testers, identified by $\{t0, ..., t3\}$. Tester $t0$ controls the *JobTracker* component, and each other tester $\{t1, t2, t3\}$, controls a *TaskTracker* instance. Tester $t0$ submits a *job* to the *jobtracker* that coordinates the *TaskTracker* components, identified by {*tasktracker0,tasktracker1,tasktracker2*}. It assigns the *map* function to {*tasktracker0,tasktracker1,tasktracker2*}, and each one reads the input data from the files splitted in the Hadoop Distributed File System (HDFS)[15], applies the user-defined *map* function on each split, and creates the outputs locally. However, tester $t2$ injects a fault on *tasktracker1* while it executes a *map* function. The *jobtracker* assigns the *reduce* function to *tasktracker0* and *tasktracker2*, but tester $t1$ injects a fault on *tasktracker0*. Then, only *tasktracker2* reads the map outputs locally or remotely, applies user-defined *reduce* function, and then, writes the results to an HDFS file.

TABLE I. A HADOOP FAULT CASE EXAMPLE

	h	D	n	C'	I	W	t
a_0	1	$\emptyset$	1	$\{c_0\}$	$startJobTracker()$		9000
a_1	2	$\{a_0\}$	3	$\{c_1, c_2, c_3\}$	$startTaskTracker()$		1000
a_2	3	$\{a_1\}$	1	$\{c_0\}$	$sendJob()$		900000
a_3	3	$\{a_1\}$	1	$\{c_1, c_2, c_3\}$	$failTaskTracker()$	$runningMap$	1000
a_4	3	$\{a_1\}$	1	$\{c_1, c_2, c_3\}$	$failTaskTracker()$	$runningReduce$	1000
a_5	4	$\{a_2\}$	1	$\{c_0\}$	$assertResult()$		10000
a_6	5	$\{a_1\}$	1	$\{c_1, c_2, c_3\}$	$stopTaskTracker()$		1000
a_7	6	$\{a_0\}$	1	$\{c_0\}$	$stopJobTracker()$		1000

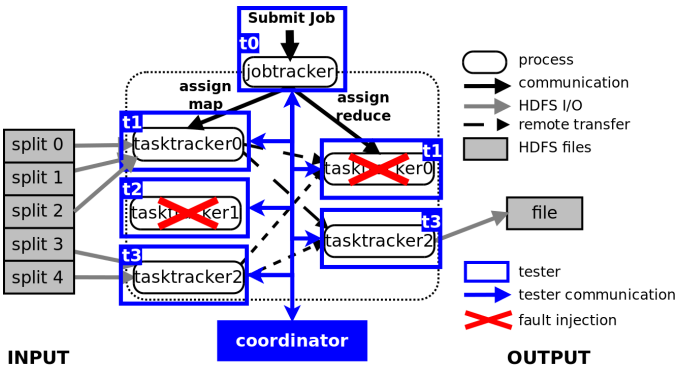

Fig. 1. HadoopTest running a fault case example

A. The Fault Case Coordination

The fault case execution consists of coordinating and controlling testers to execute actions in a distributed, parallel and synchronized way. Algorithm 1 shows the main steps to coordinate testers to execute a fault case $\mathcal{F}$. For each hierarchical level h, existing in $A^{\mathcal{F}}$, *coordinator* sends messages to testers for executing actions in parallel, receives the local results, and processes them to set the action results, $R^{\mathcal{F}}$. After executing all actions, the oracle $\mathcal{O}$ analyzes $R^{\mathcal{F}}$ and assigns a fault case verdict.

Algorithm 1: Coordination

Input: $\mathcal{F}$, a fault case; $\mathcal{M}$, a map function between $A^{\mathcal{F}}$ and the hierarchical orders of its actions.
Data: n_r, a number of success action results; and R_l, a set of local tester results.
Output: A verdict.
foreach $h \in \mathcal{M}(A^{\mathcal{F}})$ **do**
  $n_r \leftarrow$ SendMessages$(\mathcal{M}^{-1}(h), R^{\mathcal{F}})$
  $R_l \leftarrow$ ReceiveResults$(\mathcal{M}^{-1}(h), n_r)$
  $R^{\mathcal{F}} \leftarrow$ ProcessResults$(\mathcal{M}^{-1}(h), R_l)$
return $\mathcal{O}(R^{\mathcal{F}})$

Algorithm 2 shows the *SendMessages* function. It receives a set of actions with the same hierarchical order, checks the action dependency relations, and if no problems, sends controlling messages to the specified testers execute them and returns the required number of success action results.

Algorithm 3 shows the *ReceiveResults* function. It receives action results while the required number of success actions results and the time limit are not achieved. It verifies whether the result can be considered, checking the number of success actions results from related action; and returns the set of local tester results.

Algorithm 2: Send Messages

Input: A', a set of same hierarchical order actions; $R^{\mathcal{F}}$, a set of action results
Output: n_r, a number of success action results
$n_r \leftarrow 0$
foreach $a_i \in A'$ **do**
  if $R^{\mathcal{F}}[a_j] = pass, \forall a_j \in D_{a_i}$ **then**
    Send 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

Algorithm 3: Receive Results

Input: A', a set of actions; n_r, a number of success action results
Output: R_l, a set of local tester results
while $(n_r > 0) \wedge (clock < t_{a_i}, \forall a_i \in A')$ **do**
  Receive result r from t and identify its action a_i
  if $(n_{a_i} = |C'_{a_i}|)$ **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 ;

Finally, Algorithm 4 shows the *ProcessResults* function. It processes the local results of actions executed in parallel and assigns a single result for each action, compounding the set of action results that it is returned.

B. The Faul Case Action Execution

Algorithm 5 shows the steps to execute a fault case action by a *tester*. It receives the coordination message to execute a_i. If the trigger W_{a_i} is defined, it waits its execution. After that, or if W_{a_i} is not defined, the *tester* verifies if the number of success action results n_{a_i} is greater than zero, then it executes the set of instructions I_{a_i} and returns its result. Otherwise, it returns *failure*.

Algorithm 4: Process Results

Input: A', a set of actions; R_l, a set of local results
Output: $R^{\mathcal{F}}$, a set of actions results
foreach $a_i \in A'$ **do**
  if $n_{a_i} = 0$ **then**
    $R^{\mathcal{F}}[a_i] \leftarrow success$
  if $R_l[c] = success, \forall c \in C'_{a_i}$ **then**
    $R^{\mathcal{F}}[a_i] \leftarrow success$
  else
    if $\exists r \in R_l[c] : r = failure, \forall c \in C'_{a_i}$ **then**
      $R^{\mathcal{F}}[a_i] \leftarrow failure$
    else
      $R^{\mathcal{F}}[a_i] \leftarrow timeout$
return $R^{\mathcal{F}}$;

Algorithm 5: Fault Case Action Execution

Data: a_i, a fault case action
Output: An action execution result
$a_i \leftarrow$ ReceiveAction()
if $W_{a_i} \neq NULL$ **then**
  Run W_{a_i}
if $n_{a_i} > 0$ **then**
  return Run I_{a_i}
return $failure$

C. Writing Fault Cases

A fault case is composed by a set of system components, a set of fault case actions, a set of action results, and an oracle (Definition 2.1). HadoopTest provides the set of action results and the oracle, while obtains the set of system components from a set of fault case actions. Thus, to deploy a fault case in HadoopTest is necessary to describe a set of fault actions and execute it via HadoopTest.

A set of fault case actions is a Java class where each action is a method marked with the *@TestStep* annotation. This annotation has metadata that represent the attributes of a fault case action. HadoopTest uses reflection to read this metadata and use them during the fault case execution. The *@TestStep* available attributes are:

- *order*, that is the h attribute and it is a natural number;
- *depend*, that is the D attribute and it is an optional string composed by a set of methods separated by comma;
- *answers*, that is the n attribute and it is an optional natural number;
- *range*, that is the C' attribute and it is a string composed by a set of tester identifiers (natural numbers) separated by comma, or a natural number range (e.g., "$1-3$"), or an asterisk to all testers;
- *when*, that is the W attribute and it is a string composed by a command;
- *timeout* is the t attribute and it is a natural number interpreted as milliseconds.

Listing 1 shows the *FaultCaseExample* sliced class. It is a subclass of *AbstractHadoop* and implements two main methods of the representative fault case example described in Table I. The *AbstractHadoop* class implements the Hadoop library and provides access to the methods that abstract the Hadoop programming complexity, e.g., the method *failTaskTracker()*, that injects a fault in the *TaskTracker* component.

Listing 1. Fault Case Class Example

```
public class FaultCase extends AbstractMR{
  ...
  @TestStep(order=3, depend="a1", answers=1,
    range="0", when="", timeout=900000)
  public void a2() {
    sendJob();
  }
  @TestStep(order = 3, depend = "a1",
    answers = 1, range = "1-3",
    when = "waitMapRun", timeout = 1000)
  public void a3() {
    failTaskTracker();
  }
  ...
}
```

IV. Experimental Validation

This section presents an evaluation and validation of HadoopTest through the automatic execution of fault cases for the Hadoop's dependability testing. We follow the well-known properties presented by *Arlat et al.* [10] to characterize the fault injection techniques. We generated fault cases from a model of the Hadoop fault tolerance mechanism, and we use them to show the HadoopTest adequacy.

The experiments were executed on the Grid5000 platform [16] using up to 200 cluster machines running Debian GNU/Linux. The cluster machines were connected by a 1 Gbps network and they had a similar configuration: 2 Intel Xeon 2.6GHz dual-core processors, 8 GB RAM memory and 250 GB SATA HD.

A. Controllability

The controllability property denotes the ability to control *where* and *when* faults are injected. We executed the fault case example presented in Table I to show the high HadoopTest controllability property. We executed this fault case considering two setups. In the first, *tester 1* failed its *TaskTracker* because it was the first to execute a *map* task, and *tester 2* failed its *TaskTracker* because it was the first to execute a *reduce* task. The fault case verdict was PASS once the Hadoop behavior was according the specified, and the result was the expected.

In the second setup, we executed the same fault case with other set of machines. Although the fault case verdict also was PASS, *tester 0* failed its *TaskTracker* first, instead of *tester 1* as before occurred. Hadoop distributed tasks differently in this setup, but HadoopTest successfully and individually controlled

each Hadoop component, it injected faults according to its processing steps and validated its behaviors.

HadoopTest enables an apprimorate control of Hadoop when executing a fault case. We executed two similar fault cases to show this. Both fault cases have four testers and two testers fault their *TaskTracker* when they are executing a *map* task. However, in one, the *TaskTrackers* failed have the input data, in the other not, i.e., the input data remain at the *TaskTracker* online. HadoopTest returned the verdict PASS in the first fault case, once Hadoop stopped and returned an error informing that the input data were no longer available. In the second fault case, the verdict was FAIL because Hadoop interrupted the execution when the second *TaskTracker* failed, although the input data remained in the other active *TaskTracker*. The correct behavior would be to schedule the tasks to the active *TaskTracker*, but Hadoop did not do it due to a corruption of a control file.

B. Time Measurement

HadoopTest enables to acquire and use timing information associated to the monitored events, e.g., measurement of error detection latency. We detected that Hadoop does not use some configuration attributes that would be to detect component faults. We set the *mapreduce.task.timeout* attribute to one minute, but Hadoop spent about thirteen minutes to forward a failed map task. The same occurred with other *timeout* attributes to detect fault components, except *mapreduce.JobTracker.expire.trackers.interval* that could be successfully used to set the fault component detection latency.

We executed another fault case to validate the Hadoop behavior when the input data were no longer available, i.e., all components that store data fail. This fault case has one *JobTracker* and two *TaskTracker* that store the input data. Both *TaskTrackers* fail when executing a *map* task, and as there no other *TaskTracker*, *JobTracker* should interrupt the execution and return an error, but it does not occur. Hadoop did not interrupt the job execution for until thirteen hours, when HadoopTest interrupted the *sendJob* execution by timeout and the fault case verdict was FAIL.

C. Nonintrusiveness

The nonintrusiveness property relates to the level of avoiding or minimizing any undesired impact on the SUT behavior. HadoopTest presents a high nonintrusiveness since it does not need to alter the Hadoop source code to deploy any fault case. HadoopTest uses the Hadoop scripts to start and stop their components, uses local logs as a trigger to activate a fault injection, and uses the *kill* bash command to inject a fault. The *AbstractHadoop* class provides these cited functions and can be implemented by intention, e.g., use another tool to inject faults or Aspect-Oriented Programming [17] to activate a fault injection.

PiEstimator is an application bundled into Hadoop and calculates the π value. We evaluated the HadoopTest impact by executing this application by two ways. In the first, we executed Hadoop alone to know the raw execution time, and in the second, we executed Hadoop by HadoopTest to evaluate the overhead produced during testing. We used 10, 50, 100 and 200 machine-nodes on the Grid5000 to realize this experiment, and varied the *map* tasks number in each execution. Figure 2 shows the average execution time of PiEstimator running on Hadoop and HadoopTest. HadoopTest confirms its high nonintrusiviness once presents a minimal impact controlling Hadoop during the fault cases executions.

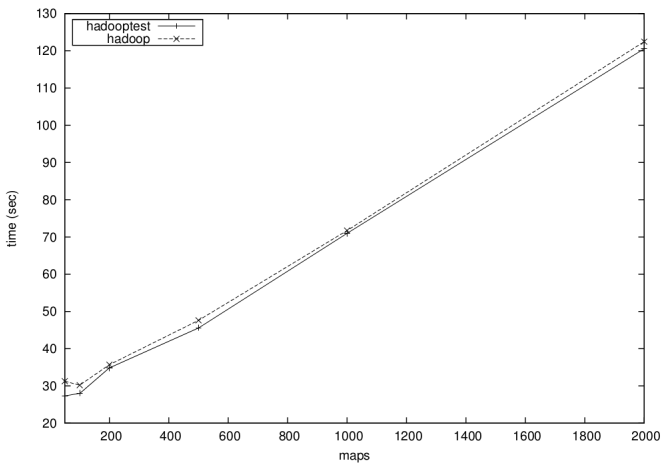

Fig. 2. Execution time variance between Hadoop and HadoopTest

D. Repeatability and Reproducibility

The repeatability is the ability to accurately reproduce fault cases, and reproducibility is the ability to reproduce results statistically for a given set-up. HadoopTest presents high repeatability and reproducibility since all fault cases are fully repeatable and reproducible. We execute at least three times all fault cases presented here and we obtained the same results in all executions[1].

E. Efficacy

The efficacy property is the ability to produce significant fault cases. Our approach achieves a high efficacy since it generates representative fault cases from the model of the Hadoop fault tolerance mechanism, ensuring to exercised it entirely. Moreover, HadoopTest executed fault cases in real deployment scenarios automatically and identified bugs in Hadoop by executing only some fault cases.

V. Related Work

Hadoop related testing frameworks are not applicable to the dependability testing. MRUnit [18] and Herriot [19] provide a set of interfaces that validates small system parts, e.g., a method or a function. Herriot was proposed to be a large-scale automated test framework but its *Csallner et al.* [20] systematically search the bad-defined map and reduce functions, possibly identified by component faults. Others, evaluate Hadoop execution by log analysis to detect Hadoop performance problems [21], [22], [23], [24]. Although, these approaches evaluate Hadop functionality and performance, they do not automatically execute fault cases and validate the system dependability.

Related work to Hadoop dependability generate fault cases randomly or by the Test Engineer [25], [26], [27]. These

[1]Execution logs are available at http://goo.gl/mfKYH

approaches are inadequate for the dependability testing of Hadoop systems because they disregard the internals of the fault tolerance mechanism, i.e., they ignore the behavior of fault recovery protocols regarding the different processing steps, e.g., they inject faults in some machines (fails 3 of 10) for some period (from 30 to 40 sec). They can evaluate system behavior but they cannot test system dependability.

The dependability of other distributed systems is evaluated by fault cases systematically generated from source code, but even applying pruning techniques they are too costly and they limit the fault cases to few concurrent faults [28], [29], [30].

VI. Conclusion

We exposed and analyzed the problem of testing the Hadoop dependability. We presented HadoopTest, a dependability testing framework that executes fault cases in real deployment scenarios automatically. We showed how HadoopTest controls and monitors all Hadoop distributed components individually, injects faults according to their processing steps, and validates the system behavior. We showed the adequacy of HadoopTest by executing some representative fault cases and identifying some Hadoop bugs. We believe our framework is promising for testing other distributed systems, and we plan primarily to test other Hadoop-based systems, such as HadoopDB and Hive.

References

[1] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *OSDI - USENIX Symposium on Operating Systems Design and Implementation*. San Francisco, California: ACM Press, 2004, pp. 137–149. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary;jsessionid=53CA72B524B9A6153BFE89FE26FBB832?doi=10.1.1.163.5292

[2] Hadoop, "The Apache Hadoop," http://hadoop.apache.org/, 2012. [Online]. Available: http://hadoop.apache.org/

[3] Y. Liu, X. Liu, L. Xiao, L. M. Ni, and X. Zhang, "Location-aware topology matching in P2P systems," in *INFOCOM - Joint Conference of the IEEE Computer and Communications Societies*, IEEE, Ed., vol. 4, no. C. IEEE, 2004, pp. 2220–2230. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1354645

[4] A. Avizienis, J.-C. Laprie, B. Randell, and C. E. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, 2004. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1335465

[5] P. Ammann and J. Offutt, *Introduction to Software Testing*. Cambridge University Press, 2008. [Online]. Available: http://books.google.com/books?hl=en&lr=&id=leokXF8pLY0C&oi=fnd&pg=PR5&dq=Introduction+to+Software+Testing&ots=3JsXf2NVb_&sig=ft_wIA_w6mDlqU5dVfNYja6ykYohttp://books.google.com/books?hl=en&lr=&id=leokXF8pLY0C&oi=fnd&pg=PR5&dq=Introduction+to+software+testing&ots=3JsXf2NW7X&sig=KWH4Q0eAg7eP5N9TKeQ0vG9yai8

[6] K. Echtle and M. Leu, "Test of fault tolerant distributed systems by fault injection," in *FTPDS - Workshop on Fault-Tolerant Parallel and Distributed Systems*. IEEE, 1994, pp. 244–251. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=494496http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=494496&contentType=Conference+Publications&searchField=Search_All&queryText=Test+of+fault+tolerant+distributed+systems+by+fault+injection

[7] A. M. Ambrosio, F. Mattiello-Francisco, N. L. Vijaykumar, S. V. de Carvalho, V. Santiago, and E. Martins, "A methodology for designing fault injection experiments as an addition to communication systems conformance testing," in *DSN-W - International Conference on Dependable Systems and Networks Workshops*, Yokohama, Japan, 2005. [Online]. Available: http://pdf.aminer.org/000/220/599/on_the_development_of_fault_tolerant_on_board_control_software.pdf

[8] J. E. Marynowski, "Towards Dependability Testing of MapReduce Systems," in *IPDPS - IEEE International Parallel and Distributed Processing Symposium, PhD Forum*. Boston, MA, USA: To appear, 2013.

[9] E. Marynowski, A. R. Pimentel, T. S. Weber, and A. J. Mattos, "Dependability Testing of MapReduce Systems," in *ICEIS - International Conference on Enterprise Information Systems*, 2013.

[10] J. Arlat, Y. Crouzet, J. Karlsson, P. Folkesson, E. Fuchs, and G. Leber, "Comparison of Physical and Software-Implemented Fault Injection Techniques," *IEEE Transactions on Computers*, vol. 52, no. 9, pp. 1115–1133, Sep. 2003. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.3.5908http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1228509

[11] R. Natella, D. Cotroneo, J. A. Duraes, and H. S. Madeira, "On Fault Representativeness of Software Fault Injection," *TSE - IEEE Transactions on Software Engineering*, 2012. [Online]. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6122035http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6122035

[12] S. Bernardi, J. Merseguer, and D. C. Petriu, "Dependability Modeling and Assessment in UML-Based Software Development," *The Scientific World Journal*, vol. 2012, pp. 1–11, 2012. [Online]. Available: http://www.tswj.com/2012/614635/

[13] G. Callou, P. Maciel, D. Tutsch, and J. Araújo, "A Petri Net-Based Approach to the Quantification of Data Center Dependability," in *Petri Nets - Manufacturing and Computer Science*, P. Pawlewski, Ed. InTech, 2012, p. 492. [Online]. Available: http://cdn.intechopen.com/pdfs/38510/InTech-A_petri_net_based_approach_to_the_quantification_of_data_center_dependability.pdfhttp://www.intechopen.com/books/petri-nets-manufacturing-and-computer-science/a-petri-net-based-approach-to-the-quantification-of-data-center-dependability

[14] E. C. de Almeida, J. E. Marynowski, G. Sunyé, and P. Valduriez, "PeerUnit: a framework for testing peer-to-peer systems," in *ASE - International Conference on Automated Software Engineering*. New York, USA: ACM, 2010, pp. 169–170. [Online]. Available: http://doi.acm.org/10.1145/1858996.1859030

[15] "Hadoop Distributed File System," http://hadoop.apache.org/hdfs/.

[16] "Grid 5000," http://www.grid5000.fr/. [Online]. Available: http://www.grid5000.fr/

[17] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-m. Loingtier, and J. Irwin, "Aspect-Oriented Programming," in *ECOOP - European Conference on Object-Oriented Programming*, no. June. Finland: Springer, 1997.

[18] "MRUnit," http://www.cloudera.com/hadoop-mrunit. [Online]. Available: http://archive.cloudera.com/docs/mrunit/index.htmlhttp://www.cloudera.com/hadoop-mrunit

[19] K. Boudnik, B. Rajagopalan, and A. C. Murthy, "Herriot," https://issues.apache.org/jira/browse/HADOOP-6332, 2010. [Online]. Available: https://issues.apache.org/jira/browse/HADOOP-6332

[20] C. Csallner, L. Fegaras, and C. Li, "New Ideas Track: Testing MapReduce-Style Programs," in *ESEC/FSE 2011*, Szeged, Hungary, 2011.

[21] J. Tan, X. Pan, S. Kavulya, R. Gandhi, and P. Narasimhan, "SALSA: analyzing logs as state machines," in *WASL - Conference on Analysis of System Logs*. CA, USA: USENIX, Dec. 2008, p. 6. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855886.1855892

[22] X. Pan, J. Tan, S. Kalvulya, R. Gandhi, and P. Narasimhan, "Blind Men and the Elephant: Piecing Together Hadoop for Diagnosis," in *ISSRE - International Symposium on Software Reliability Engineering*, 2009.

[23] J. Tan, X. Pan, S. Kavulya, R. Gandhi, and P. Narasimhan, "Mochi: visual log-analysis based tools for debugging hadoop," in *HotCloud - Conference on Hot Topics in Cloud Computing*. USENIX, 2009, p. 18.

[24] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The HiBench benchmark suite: Characterization of the MapReduce-based data

analysis," in *ICDEW - International Conference on Data Engineering Workshops*. IEEE, 2010, pp. 41–51. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5452747

[25] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin, "HadoopDB: An architectural hybrid of MapReduce and DBMS technologies for analytical workloads," in *VLDB - International Conference on Very Large Data Bases*. VLDB Endowment, 2009, pp. 922–933. [Online]. Available: http://dl.acm.org/citation.cfm?id=1687731

[26] A. Sangroya, D. Serrano, and S. Bouchenak, "Benchmarking Dependability of MapReduce Systems," University of Grenoble - LIG - INRIA, Grenoble, France, Tech. Rep., 2012. [Online]. Available: http://rr.liglab.fr/research_report/RR-LIG-027_orig.pdf

[27] F. Dinu and T. E. Ng, "Understanding the Effects and Implications of Compute Node Related Failures in Hadoop," in *HPDC - International Symposium on High-Performance Parallel and Distributed Computing*. New York, USA: ACM, 2012, p. 187. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2287076.2287108

[28] P. Joshi, H. S. Gunawi, and Kou, "PREFAIL: A Programmable Tool for Multiple-Failure Injection," in *OOPSLA - Conference on Object-Oriented Programming*, Portland, Oregon, USA, 2011. [Online]. Available: http://srl.cs.berkeley.edu/$\sim$ksen/papers/prefail.pdf

[29] C. Fu, B. G. Ryder, A. Milanova, and D. Wonnacott, "Testing of java web services for robustness," in *ISSTA - International Symposium on Software Testing and Analysis*, Jul. 2004, pp. 23–33. [Online]. Available: http://dl.acm.org/citation.cfm?id=1013886.1007516

[30] P. D. Marinescu, R. Banabic, and G. Candea, "An extensible technique for high-precision testing of recovery code," in *USENIXATC - Conference on USENIX Annual Technical Conference*. USENIX, Jun. 2010, p. 23. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855840.1855863