Dependability Testing of MapReduce Systems

João Eugenio Marynowski¹, Andrey Ricardo Pimentel¹, Taisy Silva Weber² and Antonio Junior Mattos

¹Department of Informatics, Federal University of Paraná, UFPR, Curitiba, Brazil
²Department of Informatics, Federal University of Rio Grande do Sul, UFRGS, Porto Alegre, Brazil

Keywords: MapReduce, Hadoop, Dependability, Test, Fault Injection, Fault Tolerance.

Abstract: MapReduce systems have been widely used by several applications, from search engines to financial and commercial systems. There is considerable enthusiasm around MapReduce systems due to their simplicity and scalability. However, they lack a testing approach and framework ensuring their dependability. In this work, we propose a complete dependability testing solution for MapReduce systems. This solution is a model-based approach to generate representative fault cases, and a testing framework to automate their execution. Moreover, we introduce a new way to model distributed components using Petri Nets, and we show the promising results of the proposed testing framework, HadoopTest, on identifying faulty systems in real deployment scenarios.

1 INTRODUCTION

The amount of data stored by various applications, such as social networks, commercial applications, and research, have grown to over petabytes. There are many frameworks to facilitate the analysis of large data sets; MapReduce is one of them, with broad adoption. It abstracts parallel and distributed issues such as data partition, replication, distributed processing, and fault tolerance (Dean and Ghemawat, 2004).

Despite a considerable number of MapReduce applications may present partial results, such as large-scale web indexing and pattern-based searching, several applications must present full results, such as applications in domains of business, financial, and research. To make use of MapReduce in such domains, it is essential to test its dependability (Abouzeid et al., 2009; Teradata Corporation, 2012; Hadoop, 2012).

Dependability test 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 (Avizienis et al., 2004; Ammann and Offutt, 2008). For this purpose, the system is executed on a controlled testing environment with the injection of artificial faults. Two main issues concerning this approach are: generating representative elements from the potentially infinite and partially unknown set of fault cases, and automating their executions.

Testing the dependability of MapReduce systems requires to execute fault cases capable of stimulating all of its tolerated faults, which, by turn requires explicit control over its processing steps.

In this work, we present a solution for dependability testing of MapReduce systems through the generation and execution of representative fault cases. We use a Petri Net model of the fault tolerance mechanism to generate these fault cases, and a framework to automate their execution in real deployment scenarios. Additionally, we introduce a new approach to interpret the model components, modeling the MapReduce components as dynamic items, and modeling the independence of these components with their actions and states.

This paper is organized as follows. The next session introduces the basic concepts, presenting a description of MapReduce and defining fault cases. Section 3 presents our approach to model MapReduce fault tolerance mechanism. Section 4 shows how we generate the representative fault cases. Section 5 presents our framework for dependability testing. Section 6 describes the initial results through implementation and experimentation. Section 7 surveys related work. Section 8 concludes the paper.
2 BASIC CONCEPTS

2.1 MapReduce

MapReduce is a simplified programming model and the associated implementation for processing and analyzing large scale data. It offers a programming environment based on two high-level functions, map and reduce, and a runtime environment to execute them on a cluster. The MapReduce architecture includes several worker components, and one master that schedules map and reduce tasks to run at the workers.

Figure 1 shows a MapReduce execution instance. The master receives a Job and coordinates five worker components, identified by {worker0, ..., worker4}. It assigns the map function to {worker0, worker1, worker2, worker3}, and each one reads the input data from the files splitted in a Distributed File System (DFS), applies the user-defined map function on each split, and creates several outputs locally. The master assigns the reduce function to {worker4, worker1, worker0}, and each one reads the map outputs locally or remotely, applies user-defined reduce function, and writes the results to the DFS.

![MapReduce Execution](image)

Figure 1: A MapReduce execution overview.

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 processing steps, e.g., if a worker fails when it is executing a map task, the master only reschedules its task for another worker; but if a component fails after executing a map task, the master reschedules the task for another worker and informs all workers executing reduce tasks that they must read the map result from the new worker.

2.2 Fault Case

A fault case is a distributed test case extension involving the components required for a complete execution and validation of a system under test while faults are injected (Echtle and Leu, 1994; Ambrosio et al., 2005; de Almeida et al., 2010b).

Definition 2.1 (Fault Case). A fault case is a 4-tuple $F = (C^F, A^F, R^F, O)$ where:

- $C^F = \{c_0, c_1, ..., c_n\}$, and it is a finite set of system components;
- $A^F = \{a_0, a_1, ..., a_m\}$, and it is a finite set of actions that can involve fault injections;
- $R^F = \{r_{00}, r_{01}, ..., r_{mn}\}$, and it is a finite set of action results;
- $O$ 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 ($a_i$) can get its result ($r_{ij}$): success, failure, or timeout (without response during a time limit). If all action results ($R^F$) get success, the $F$ verdict is pass. If any action result is failure, the $F$ verdict is fail. But if at least one action execution gets timeout, the $F$ verdict is inconclusive, making the test inaccurate for assigning some of the earlier statements and, moreover, it is necessary to rerun the fault case.

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

- $h \in \mathbb{N}$, $h \leq |A|^F$, and it is an hierarchical order in which action $a_i$ must execute - actions with same $h$ execute in parallel;
- $n \in \mathbb{N}$, $n \leq |C|^F$, and it is the success number of action executions to result success for $a_i$;
- $C' \subseteq C^F$, 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 $a_i$;
- $D \subseteq A^F \forall a_j \in D : j < i, r_{ij} = SUCCESS$, and it is a set of actions that must be successfully executed before $a_i$, otherwise the action result $r_{ai}$ is failure;
- $t$ is a time to execute $a_i$.

3 MODELING MapReduce FAULT TOLERANCE MECHANISM

Modeling the MapReduce fault tolerance mechanism demands a formal model of the concurrent and distributed behavior of its components. The model must represent the components as dynamic items, enabling
them to be easily removed or inserted, without substantial model changes. Moreover, the model should represent the components without specifying their actions, allowing an action to be performed by any enabled component. This feature is essential to model the rescheduling process of faulty map and reduce tasks.

Finite State Machine (FSM) (Bernardi et al., 2012) and Petri Net (PN) (Callou et al., 2012) are the main approaches to model distributed systems involving their dependability properties. FSM abstracts the details of the system behavior and enables to a direct relation to the MapReduce processing steps. Although there are extensions to represent other features (e.g., timing specs), FSM restricts the modeling of several components that have parallel and distinct behaviors. Each component needs a specific set of states and an alphabet to model its behavior.

The PN modeling enables us to take a new approach to interpret their components, modeling the MapReduce components as dynamic items, to be easily inserted or removed. Moreover, it allows to model the independence of these components with their actions and states, i.e., an action can be executed by any enabled component.

Figure 2 shows a Petri Net that models part of the MapReduce fault tolerance mechanism that handles faults while running Map and Reduce functions. Labeled transitions represent the fault case actions, tokens represent MapReduce components, and places represent their states or processing steps. When the transition "master.sendJOB" fires, it consumes one token from "online master" and one from "online workers", and produces a new one in "worker.runningMap". Now, two transitions can fire, "nothing" and "worker.runningMap-FAIL". If "nothing" fires, it consumes one token from "worker.runningMap", and produces one in "worker.runningReduce". If "worker.runningMap-FAIL" transition fires, it consumes one token from "worker.runningMap" and one from "online workers", and produces one again in "worker.runningMap". This behavior occurs similarly if "worker.runningReduce-FAIL" fires when it has a token at "worker.runningReduce", but it is necessary to have a token at "online workers" for firing it.

PN allows a comprehensively modeling of the MapReduce fault tolerance mechanism. Moreover, it allows extensions to model other behaviors implicitly specified in MapReduce fault tolerance mechanism, such as the temporal faults identification and the process interruption when it is impossible complete a job.

4 GENERATING REPRESENTATIVE FAULT CASES

The representativeness of a fault case is how important it is to identify defects on a system under test (Arlat et al., 2003; Natella et al., 2012). We consider the representative fault cases for the dependability testing of MapReduce systems as generated through an abstraction of its fault tolerance mechanism. This approach is successfully used to test other systems (Echtle and Leu, 1994; Ambrosio et al., 2005; Bernardi et al., 2012). It guides the generation to a finite set of fault cases that should be tolerated, and that they must be tested to ensure the system dependability.

We generate representative fault cases from a reachability graph of the Petri Net that models the MapReduce fault tolerance mechanism. A reachability graph consists of all possible sequences of transition firings from a Petri Net. Each possible path starting from the root graph vertex composes one fault case. This approach is applicable only in a Pure Petri Net, i.e., that has no loops.

Figure 3 shows a reachability graph generated from the Petri Net example at Figure 2. There are three fault case possible: (1) without faults, executing "master.sendJOB" and "master.assertRESULT"; (2) with one fault, adding "worker.runningMap-FAIL"; and (3) with the other possible fault, "worker.runningReduce-FAIL".

Table 1 shows a set of fault case actions of the fault case (2). The goal is to validate the MapReduce execution while one component fails when executing a map task. This fault case involves three components, \( C^f = \{c_0, c_1, c_2\} \), obtained from the Petri Net tokens, and seven actions \( A^f = \{a_0, \ldots, a_6\} \), obtained from the reachability graph (Figure 3) and the start and stop.
actions. The component $c_0$ executes the action $a_0$ to start the master. If action $a_0$ succeeds, the components \{c$_1$, c$_2$\} execute the action $a_1$ to start the workers. Otherwise, the action $a_1$ finishes and receives the failure result. This occurs with all actions that has a dependency relation with a failed action, recursively. Without failed actions, the process continues and the next execution is $a_2$ by the component $c_0$, and it submits a job. During the job execution, only the first component ($n_0 = 1$) of \{c$_1$, c$_2$\} fails when it executes the map task ($W_0 = \text{runningMap}()$). At action $a_2$, the $c_0$ validates the job result, comparing the expected with the obtained. The next actions stop the MapReduce execution.

5 FRAMEWORK FOR TESTING MAPREDUCE SYSTEMS

HadoopTest is a test framework to automatically execute fault cases. It extends the PeerUnit testing framework (de Almeida et al., 2010a). HadoopTest adds the controlling and monitoring of all MapReduce components, the injecting of faults according to its processing steps, and the validation of its behavior.

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

Figure 4 shows the application of HadoopTest to the MapReduce instance presented in Figure 1, and with a fault injection while a worker executes a Reduce function. The coordinator individually controls the execution of six testers, identified by $t0..t5$, following the fault case. Tester $t0$ controls the master component and each other tester, $t1..t5$, controls a worker instance. This architecture enables the deployment of fault cases applying lower service functions on testers. For instance, tester $t2$ injects a fault on worker1, removing it from the system while it executes the reduce function. This enables to put MapReduce components in any state (i.e., running, idle, or stopped) and monitors their activity at any time.

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 for executing a fault case $F$. For each hierarchical level $h$, existing in $A$, the coordinator sends messages to the testers for executing actions in parallel, receives the local results, and processes them to set action results, $R$. After executing all actions, the oracle $O$ analyzes $R$ and assigns the fault case verdict.

Algorithm 1: Coordination Algorithm.

Input: $F$, a fault case; $M$, a map function between $A^\pi$ and the hierarchical orders of its actions
Data: $R$, a set of local tester results
Output: A verdict

$\text{foreach } h \in M(A^\pi) \text{ do}$
  $\text{SendMessages}(M^{-1}(h), R^F)$
  $R_t \leftarrow \text{ReceiveResults}(M^{-1}(h))$
  $R^F \leftarrow \text{ProcessResults}(R_t, M^{-1}(h))$

return $O(R^F)$ ;

Algorithm 2 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 his execution. After that, or if $W_{a_i}$ is not defined, the tester verifies if the number of success action executions $n_{a_i}$ is greater than zero, then it executes the set of instructions $I_{a_i}$ and returns the execution result. Otherwise, it returns failure, informing to the coordinator that it cannot execute $a_i$.

6 EXPERIMENTAL VALIDATION

This section presents an evaluation of our proposed
solution through the automatic and manual executions of fault cases for testing Hadoop (Hadoop, 2012), an open-source MapReduce implementation. First, we present the results obtained by the manual execution of the representative fault cases. Second, we evaluate the overhead produced by HadoopTest to coordinate the execution of fault cases. Finally, we validate the HadoopTest effectiveness for identifying faulty systems by testing the PiEstimator, an application bundled into Hadoop.

6.1 Manual Execution of Representative Fault Cases

We manually executed some fault cases generated from the modeling of the MapReduce fault tolerance mechanism to confirm their representativeness in the defect identification. One fault case consisted of four components that execute the WordCount, while two components failed by crash when they executed the map task. Hadoop interrupted the execution when the second component failed, although the data remained in the other active component. The correct behavior would be to schedule the tasks to the active component, but Hadoop did not do it due to a corruption of a control file.

In addition, we executed fault cases involving temporal parameters. We identified that Hadoop does not consider \textit{timeout} parameters to detect fault components, and does not interrupt the execution when the data were no longer available, i.e., all components that stored data failed, but Hadoop continued without iteration by two hours.

6.2 The HadoopTest Overhead

We evaluated the HadoopTest overhead by executing PiEstimator in two ways. In the first one, Hadoop is executed alone, to evaluate the raw execution time. In the second one, Hadoop is executed along with HadoopTest, to evaluate the overhead produced during testing. We use 10, 50, 100 and 200 machine-nodes on the Grid'5000 platform to realize this experiment. Figure 5 shows the average execution time of PiEstimator running on Hadoop and HadoopTest. We vary the number of map instances in each execution.

6.3 Identifying Faulty Systems

We used Mutation Testing (Offutt, 1994) to evaluate whether HadoopTest is able to identify faulty systems. We create a set of faulty versions (i.e., mutants) of the PiEstimator. Mutations are changes of arithmetic and logic operators into the original source code to generate incorrect results. The goal is to identify the largest possible number of incorrect results.

We generated 13 mutants of the PiEstimator class.
and Table 2 shows the results. The expected π value returned by the original application was 3.1416 and only the mutants M1, M6, M7, M9 and M12 returned this value. These mutants have the pass verdict on the test case execution while the other mutants M4, M5, M10 and M11 received a fail verdict, since the π computation parameters were modified resulting in a different value than the expected one. In the case of the mutants M0, M2, M3 and M8, the modifications were in the execution parameters which interfered on their correct execution. Hence, they returned NULL as results.

Table 2: Results and verdicts generated by 13 PiEstimator mutants.

<table>
<thead>
<tr>
<th>Mutants</th>
<th>Result</th>
<th>Pass</th>
<th>Fail</th>
</tr>
</thead>
<tbody>
<tr>
<td>M0</td>
<td>NULL</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>M1</td>
<td>3.1416</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>M2</td>
<td>NULL</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>M3</td>
<td>NULL</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>M4</td>
<td>3.0776</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>M5</td>
<td>3.1312</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>M6</td>
<td>3.1416</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>M7</td>
<td>3.1416</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>M8</td>
<td>NULL</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>M9</td>
<td>3.1416</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>M10</td>
<td>3.1408</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>M11</td>
<td>3.1408</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>M12</td>
<td>3.1416</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

We evaluated the HadoopTest effectiveness by the number of detected mutants. When mutation analysis is applied to a system code and generates several mutants, some of them are equivalent to the original source code, due to different reasons, such as the modified part is never executed, and the binary operators used have the same result. We considered the equivalent mutants those that obtained the same output as the original system. The initial implementation of HadoopTest demonstrated promising results by identifying all the non-equivalent mutants of PiEstimator.

We exposed and analyzed the issue of testing MapReduce system dependability. We presented a solution based on the generation and execution of representative fault cases. We generated fault cases from a formal model of the MapReduce fault tolerance mechanism. We evaluated two modeling approaches and adopted the Petri Net because its adequacy. We presented a new way to model distributed components using Petri Nets. We modeled the MapReduce components as dynamic items and the independence of them with their actions and states. Moreover, we showed the HadoopTest framework, that executes i.e., they ignore the behavior of fault recovery protocols regarding the different processing steps, e.g., they inject faults in some machines (fauls 3 of 10) for some period (from 30 to 40 sec). They can evaluate the system behavior, but they cannot test system dependability. Others evaluate the dependability by generating, systematically, fault cases from source code. Such approaches are costly, even after applying pruning techniques, and they limit the fault case generation to few concurrent cases (Joshi et al., 2011; Fu et al., 2004; Marinescu et al., 2010).

Some testing frameworks provide solutions to control distributed components and to validate the system behavior, but they do not inject faults neither consider the components processing steps (Pan et al., 2010; Dragan et al., 2006; Zhou et al., 2006; de Almeida et al., 2010a). Related fault injection frameworks enable to inject multiple and various faults, but they do not control the system dynamically to inject faults according the processing steps (Jacques-Silva et al., 2006; Pham et al., 2011; Stott et al., 2000; Lefever et al., 2004; Hoarau et al., 2007). Moreover, none of the cited frameworks presents results about MapReduce dependability.

MapReduce related testing frameworks are not applicable to the dependability testing. Herriot (Boudnik et al., 2010) provides a set of interfaces that validates small system parts, e.g., a method or a function. Csallner et al. (Csallner et al., 2011) systematically search the bad-defined map and reduce functions, possibly identified by component faults. Others, evaluate MapReduce execution by log analysis to detect MapReduce performance problems (Tan et al., 2008; Pan et al., 2009; Tan et al., 2009; Huang et al., 2010). Although, these approaches evaluate MapReduce functionality and performance, they do not automatically execute fault cases and validate the system dependability.

8 CONCLUSIONS

We exposed and analyzed the issue of testing MapReduce system dependability. We presented a solution based on the generation and execution of representative fault cases. We generated fault cases from a formal model of the MapReduce fault tolerance mechanism. We evaluated two modeling approaches and adopted the Petri Net because its adequacy. We presented a new way to model distributed components using Petri Nets. We modeled the MapReduce components as dynamic items and the independence of them with their actions and states. Moreover, we showed the HadoopTest framework, that executes
fault cases in real deployment scenarios without overhead and identifying faulty systems.

We identified some bugs in Hadoop with the manual execution of representative fault cases, but we intend to automatically execute them with HadoopTest. We plan to automatically generate representative fault cases from a Petri Net model, and test other MapReduce systems, such as HadoopDB and Hive.

ACKNOWLEDGEMENTS

This work is partially sponsored by CAPES and the experiments presented were carried out using the Grid'5000 experimental testbed.

REFERENCES


