Towards Dependability Testing of MapReduce Systems

João Eugenio Marynowski
Department of Informatics
Federal University of Paraná, UFPR
Curitiba, Brazil
jeugenio@inf.ufpr.br

Abstract—MapReduce systems have been widely used by several applications, from search tools to financial and commercial systems. There is considerable enthusiasm around these systems due to their simplicity and scalability. However, there is a lack of a testing approach, and a framework to ensure they are dependable. The goal of this PhD is to propose a complete dependability testing solution for MapReduce systems. This solution is a model-based approach for generating representative fault cases, and a testing framework to control their execution automatically. Initial experiments demonstrate promising results with HadoopTest framework coordinating their execution automatically. Initial experiments demonstrate promising results with HadoopTest framework coordinating fault cases across distributed MapReduce components and identifying faulty systems.

João Eugenio Marynowski is a 4th year PhD student, working with advisor Prof. Dr. Andrey Ricardo Pimentel.

I. INTRODUCTION

The amount of data stored by various applications, such as social networks, business applications, and research, grown to over petabytes. There are many frameworks to ease analyze on large data sets; MapReduce is one of them, with broad adoption. It abstracts issues such as processing distribution, data partition, replication, and fault tolerance [1].

Most MapReduce applications may lack completeness on its results, e.g., indexing large-scale web data. However, MapReduce applications in domains of business, financial, and research require completeness [2]. Thus, it is essential to test MapReduce dependability.

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 [3], [4]. For this purpose, the system is executed on a controlled testing environment with the injection of artificial faults. An issue when injecting faults is to find representative elements in the potentially infinite and partially unknown set of fault cases. Additionally, the testing environment must: (1) control and monitor all MapReduce distributed components, (2) inject faults according to their processing steps, and (3) validate the system behavior.

In this work, I present a solution for dependability testing of MapReduce systems through the generation and execution of representative fault cases. I use a Petri Net model of the fault tolerance mechanism to generate representative fault cases, and an optimized framework to automate the execution of them in a distributed environment. Additionally, I 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.

II. FUNDAMENTALS AND RELATED WORK

MapReduce is a simplified programming model and the associated implementation for processing and analyzing large scale data [1]. 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.

The MapReduce fault tolerance mechanism identifies faulty workers by timeout, and reschedules their tasks to another 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. So, to test the MapReduce dependability is essential to validate the system behavior injecting faults in all processing steps, according to the fault tolerance mechanism.

Related work to MapReduce dependability generate fault cases randomly or by the Test Engineer [1], [2], [5]. These approaches are insensible for testing because they inject faults disregarding the 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 [6], [7]. Some work generate the fault cases through an abstraction of the fault tolerance mechanism [8], [9], [10]. I believe the last is the most suited approach for generating representative fault cases, because it guides to the fault cases really tolerated by the system.
III. Modeling MapReduce Fault Tolerance Mechanism

Modeling the MapReduce fault tolerance mechanism demands a formal model of the concurrent and distributed behavior of their 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 component enabled. This feature is essential to model the process of scheduling map and reduce fault tasks that are dynamically rescheduled to other components.

Finite State Machine (FSM) [10] and Petri Net (PN) [11] are the main approaches applied for modeling distributed systems involving their dependability properties. FSM modeling represents the system behavior abstracting its details, and has 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.

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

Figure 1 shows a sliced example of the MapReduce fault tolerance mechanism. Labeled transitions are the fault case actions, tokens are MapReduce components, and places are their states or processing steps. When "master.sendJOB" fires, it consumes one token from "online master" and produces a new one in "worker.runningMap". Now, two transitions can fire, "T1" and "worker.runningMap-FAIL". If "T1" fires, it consumes one token from "worker.runningMap", and produces one again 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". Similarly occurs 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 possibles to comprehensively model 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.

IV. Generating Representative Fault Cases

A fault case is a test case extension involving the components required for a complete execution and validation of a system under test while faults are injected.

Definition 4.1 (Fault Case): A fault case is a 4-tuple \( F = (C, A, R, O) \) where:

- \( C \) is a finite set of system components;
- \( A \) is a finite set of actions that can involve fault injections;
- \( R \) is a finite set of action results;
- \( O \) is an oracle.

The oracle is the 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 \( F \) verdict is pass. If any action execution gets 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 rerun the fault case.

Definition 4.2 (Action): A fault case action is a 7-tuple \( a_i = (h, n, C', I, W, D, t) \) where:

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

Figure 2 shows a reachability graph generated from the Petri Net example at Figure 1. 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”.

![Petri Net modeling example.](image)

Table I shows a fault case example generated from a Petri Net. The goal is to validate the MapReduce execution while one component fails when executing a map task. This fault case involves four components \( C = \{ c_0, \ldots, c_3 \} \) and seven actions \( A = \{ a_0, \ldots, a_6 \} \). The component \( c_0 \) executes the action \( a_0 \) to start the master. Whether action \( a_0 \) succeeds, the components \( \{ c_1, \ldots, c_3 \} \) execute the action \( a_1 \) to start the workers. Otherwise, the action \( a_1 \) finishes and receives the failure result. This occurs with all action 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 \( n_{a_3} = 1 \) of \( \{ c_1, \ldots, c_3 \} \) fails when it executes the map task \( W_{a_3} = \text{runningMap}() \). At action \( a_4 \), the \( c_0 \) validates the job result, comparing the expected with the obtained. The next actions stop the MapReduce execution.

V. HADOOPTEST: FRAMEWORK FOR TESTING MAPREDUCE SYSTEMS

HadoopTest is a test framework to help researchers and practitioners to execute fault cases automatically. In [12], we presented the first version of HadoopTest extending PeerUnit [13] to control actions in parallel and all MapReduce components. Now, HadoopTest can inject faults according to MapReduce processing steps, and it validates the system behavior reporting inconsistencies.

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.

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 \( A \) and the hierarchical orders of its actions

**Data:** \( R_t \), a set of local tester results

**Output:** A verdict

```plaintext
foreach \( h \in M(A) \) do
    SendMessages(\( M(h) \))
    \( R_t \leftarrow \text{ReceiveResults}(\( M(h) \)) \)
    \( R \leftarrow \text{ProcessResults}(R_t, M(h)) \)
return \( O(R) \)
```

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 \( a_i \) 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 \( a_i \).

**Algorithm 2: Action Execution Algorithm**

**Data:** \( a_i \), a fault case action

**Output:** An action execution result

```plaintext
\( a_i \leftarrow \text{ReceiveAction()} \)
if \( W_{a_i} \neq \text{NULL} \) then
    Wait \( W_{a_i} \)
if \( n_{a_i} > 0 \) then
    return Run \( I_{a_i} \)
return FAILURE
```

VI. EXPERIMENTAL VALIDATION

In [12], we presented experiments to validate the HadoopTest initial implementation for testing the Hadoop\(^1\), an open-source MR implementation. We performed the experiments on the Grid5000 plataform\(^2\) using up to 200 cluster machines. We evaluated the produced overhead to

\(^1\)http://hadoop.apache.org/
\(^2\)http://www.grid5000.fr
coordinate the execution of distributed test cases, and validated the HadoopTest ability to identify artificial bugs in two available applications with Hadoop, PiEstimator and WordCount. HadoopTest showed no overhead to control Hadoop, comparing their execution times, and was effective for identifying faulty systems.

I manually executed some experiments involving representative fault cases. 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 due to a corruption of a control file.

In addition, I executed fault cases involving temporal parameters. I identified that Hadoop does not consider 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 thirteen hours.

### VII. Conclusion

I exposed and analyzed the issue of testing MapReduce system dependability. I presented a solution based on the generation and execution of representative fault cases. I generated them from a formal model of the MapReduce fault tolerance mechanism. I evaluated two modeling approaches and adopted the Petri Net because its adequacy. I presented a new approach to interpret the Petri Net components, that models the MapReduce components as dynamic items and the independence of them with their actions and states. Moreover, I showed the HadoopTest framework, that executes fault cases in a distributed environment without an overhead and identifies faulty systems.

The presented modeling of the fault tolerance mechanism enables to generate representative fault cases that are more effective than those generated by current practices. I identified Hadoop bugs by executing some of them manually, but I intend to automate their execution with HadoopTest. I also want to generate representative fault cases automatically from a Petri Net model, and I plan to test other MapReduce systems, such as HadoopDB and Hive.

### References


---

Table I

<table>
<thead>
<tr>
<th>$h$</th>
<th>$n$</th>
<th>$C^a$</th>
<th>$I$</th>
<th>$W$</th>
<th>$D$</th>
<th>$t$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_0$</td>
<td>1</td>
<td>${c_0}$</td>
<td>startMaster()</td>
<td></td>
<td></td>
<td>100</td>
</tr>
<tr>
<td>$a_1$</td>
<td>2</td>
<td>${c_1, \ldots, c_3}$</td>
<td>startWorker()</td>
<td>${a_0}$</td>
<td></td>
<td>1000</td>
</tr>
<tr>
<td>$a_2$</td>
<td>3</td>
<td>${c_0}$</td>
<td>sendJOB()</td>
<td>${a_1}$</td>
<td>1000000</td>
<td></td>
</tr>
<tr>
<td>$a_3$</td>
<td>3</td>
<td>${c_1, \ldots, c_3}$</td>
<td>FAIL()</td>
<td>runningMap()</td>
<td>${a_1}$</td>
<td>1000</td>
</tr>
<tr>
<td>$a_4$</td>
<td>4</td>
<td>${c_0}$</td>
<td>assertRESULT()</td>
<td></td>
<td>${a_2}$</td>
<td>10000</td>
</tr>
<tr>
<td>$a_5$</td>
<td>5</td>
<td>${c_1, \ldots, c_3}$</td>
<td>stopWorker()</td>
<td></td>
<td>${a_1}$</td>
<td>1000</td>
</tr>
<tr>
<td>$a_6$</td>
<td>6</td>
<td>${c_0}$</td>
<td>stopMaster()</td>
<td></td>
<td>${a_0}$</td>
<td>1000</td>
</tr>
</tbody>
</table>

---

3Execution logs are available at http://goo.gl/mfKYH