Abstract—MapReduce (MR) based systems are the most popular solution for large-scale data processing, where developers can easily use a large number of machines to implement large-scale data intensive applications. These systems are often deployed on large-cluster of commodity machines, where failures happen constantly due to bugs, hardware problems, outages, etc. In order to ensure their trustworthy, MR based systems must be tested extensively under abnormal conditions. Moreover, some features should be considered when testing these systems, such as: system evaluation during execution, distributed testing control, real-world condition (e.g. volatility control and fault injections), and MR components control and monitoring. In this article we present a framework for testing MR based systems. This framework is based on the individual control and monitoring of MR components and combines fault injection with functional tests. We used this framework to test two applications deployed with Hadoop, the Apache open source MapReduce implementation.

I. INTRODUCTION

Over the last decade the amount of data generated by several applications, like customer reports, survey data and social networks, have reached several petabytes. This amount of data tends to increase dramatically along the next few years up to iotabytes [2]. To analyze such large amount of data, one needs a great effort of data processing. Large-scale data processing has attracted widespread attention both in the research community and in the industry. New types of frameworks for data analysis scaling up to this level are required. MapReduce (MR) [2] appears as the most popular large-scale data processing framework.

MR systems decompose a job into smaller tasks, and a massive data set into smaller partitions, such that each task processes a different partition in parallel. The input data set is stored in a collection of partitions in a Distributed File System (DFS) deployed across machine nodes in a cluster. A MR program consists of two functions called Map and Reduce. The Map function is applied in the partition of input data that exists in the cluster node and outputs a set of intermediate key/value pairs. These set of pairs are grouped by each intermediate key and are written to files in the Map node’s local disk. Each Reduce reads and processes the files assigned by the Master node, and writes the results to an output file in the DFS. The Master is a single node responsible for coordinating Maps and Reduces daemons distributed across the nodes in the cluster. MR systems are also designed to be fault-tolerant. If a Map or Reduce fails, then the Master restarts the task on an alternate node.

Testing MR based systems is hard. First, testing needs to follow the MR execution workflow (i.e., from deploying the input up to the output production through the Map and Reduce functions). Second, distributed testing can be performed by node stimulation along execution. Stimulus are divided into actions, developed within a MR based systems, and faults injected on nodes to reproduce real-world conditions (eg., outages, bugs, etc). Generally, this is done by testing instrumentation through a component called Tester hung with each node. Third, instrumentation can not exert influence on the final result. By influence we mean the instrumentation overhead. This overhead can be even worse while testing at large-scale. MR claims to run at large-scale machine clusters, then the testing facility shall scale up as well.

In this work, we present a test harness facility, called HadoopTest, to validate MR based systems. HadoopTest is capable of to deploy test cases across multiple nodes and manage their execution with low instrumentation overhead. The original contribution is to properly stimulate MR based systems following its workflow. This is achieved by instrumenting each node of Maps and Reduces, and the master node with different distributed Testers. These Testers are responsible for sending stimulus to each instance and to coordinate the overall test case execution through a distributed synchronization algorithm. HadoopTest was used to test two applications bundled into Hadoop, the Apache open source MapReduce implementation. The experimental evaluation shows that the framework is scalable, does not influence the results (i.e., low overhead on instrumentation) and identifies fails through the mutant based fault injections.

The rest of the paper is organized as follows. The next section discusses related work. Section ?? introduces the basic concepts on testing. Section ?? presents our framework for testing MR systems. Section ?? describes the initial results through implementation and experimentation. Section ?? concludes.

II. RELATED WORK

An open issue for testing MR based systems is to follow its workflow along testing. HadoopTest tackles this issue due to a synchronization algorithm used to control the execution of distributed Testers. The algorithm paces the dispatch of stimulus to Testers in a precise order (i.e., the MR workflow). Basically, a stimuli is an action of a test case, such as: the
MR functions, or the faults injected on nodes to reproduce real conditions.

The MRUnit [?] is a testing library designed to help bridging the MR applications and the JUnit [?] testing facility. It provides a set of interfaces that must be used during the development of a MR based system. Different from our approach, MRUnit neither instruments different nodes at the same time nor stimulates them in parallel which restricts to follow the MR workflow. Furthermore, the instrumentation is based on the inclusion of code within the system under test (SUT) source code (ie., contamination). This intrusive approach may spoil the source code generating new bugs.

Herriot [?] is also a library for testing MR in a distributed way. It differs from MRUnit in its external component, called Test Node. This component is responsible for the overall test execution (e.g., start, stop in a controlled manner). However, this control is only possible whether implementing the Herriot’s interface, which also leads to the SUT contamination. Moreover, it does not inject faults that are likely while testing large-scale distributed systems such as MR.

Ganesha [?] is a black-box diagnosis technique that examines OS-level metrics to detect and diagnose faults in MR systems. It does not require any modifications on the source code, since it only does off-line analysis of nodes. An issue of off-line validation is to determine a global clock. This is required to build the correct execution trace. Indeed, this approach can be complementary to HadoopTest, since we validate while executing test cases.

PeerUnit [?], [?] is a framework for testing peer-to-peer systems. It ensures the sequencing of test case actions synchronizing their execution through all system nodes. PeerUnit allows to instrument any SUT component also leveraging the approach of distributed testers. Moreover, It also injects some faults in the SUT. It differs from HadoopTest on the test case execution workflow. First, PeerUnit does not execute different stimulus in parallel. Second, it does not provide a hierarchy of stimulus that can be useful to follow the precise MR workflow.

III. BASIC CONCEPTS

A. MapReduce Based Systems

A MapReduce based system is composed by three subsystems: Distributed File System (DFS), MR Implementation and MR Application. DFS creates multiple replicas of data blocks and distributes them on compute nodes throughout a cluster (e.g. HDFS [2]). The MR Implementation, e.g. Hadoop [2], enables automatic parallelization and distribution of a MR Application on the cluster compute nodes. A MR Application is developed by the user and implements the Map and Reduce functions (e.g. the map and reduce functions necessary to solve a word count problem).

Figure ?? [2, ] shows the data and workflow of the MR Implementation. Initially the job is submitted by the user to the master which coordinates the MR execution. It decides how many map and reduce tasks to run and how to allocate them to available workers. Worker is a client processes of MR Implementation and normally it is executed one by machine. The master picks idle workers and assigns each one a map or a reduce task. For instance, worker 0 and worker 1 do map task and reduce task after. A worker who is assigned a map task reads the contents of the corresponding input split from DFS, it applies user-defined map function and it writes the result to an output DFS file. A reduce worker reads these files remotely from every map workers, it applies them the user’s reduce function and writes the result to an output DFS file.

B. Testing Background

Testing is a process to validate system to ensure its quality. Testing may focus on different aspects of software as functionality, robustness, performance, timing constraints etc. We can to employ testing at different levels correlating them with the systems development phases [2]. The test lowest level is called unit testing and it is related to source code validation, checking basic programming commands, e.g. attributions, conditionals, repetitions, etc. The system testing is the test highest level and it is related to the project phase and it verifies the system integration with the environment.

Different kinds of testing are generally user depending on what kind of behavior one wants to check. In this work, we
are focusing on dynamic testing, i.e., evaluating its behavior during the execution of a test case considering real problems and inputs. The test case objective is to verify if a System Under Test (SUT) feature is correctly working according to certain quality criteria: robustness, correctness, completeness, performance, security, etc. We simplistically defined a test case as composed of a name, an intent, a sequence of actions, the test configuration (e.g. number and organization of nodes), a sequence of input test data and the expected output (what the test engineer expects to have by the end of a test).

Along testing, an input helps to exercise the functional aspects of a SUT and aims to generate an output. By the end of the test execution, the exercised systems generates some results in form of an output value (e.g., changes to data, reports, communication messages sent out, and output to screens). The result of the test case and the expected output are used to assign a verdict, or a validation label to the test case. A verdict is assigned among three values: pass, fail and inconclusive.

The oracle is the mechanism responsible for assigns verdicts. If the result and the expected output are the same, the verdict is pass. Otherwise, the verdict is fail. The verdict may also be inconclusive, meaning that the test case result is not precise enough to satisfy the test intent and must be done again. There are different sorts of oracles: assertions, value comparison, log file analysis, manual analysis, etc. An entire testing technique must then includes: test criteria, test cases generation techniques and an oracle.

We focus on conformance testing that is essentially a type of functional testing of a black box nature. The source code of the SUT is unknown and your behavior is checked with respect to a specification. The principle of testing is to apply inputs to SUT and to compare the observed outputs to the expected results. For testing distributed systems, basic concepts must support distributed characteristics, such as: the environment in which test are executed, the design of test cases, the execution of test cases, the observation that can be made during the test execution, the interpretation of the observed results, and the purposes related to the test cases.

Solutions of testing distributed systems are commonly based on Conformance Testing Methodology and Framework (CTMF) defined in ISO/IEC 9646 [?]. The CTMF defines conceptual architectures to support test execution. A test architecture is a description of how a Implementation Under Test (IUT) is to be tested, in particular, which inputs can be controlled and which outputs can be observed. Yet, the CTMF defines a System Under Test (SUT) as a set of IUTs. The CTMF defines four test architectures: local test, distributed test, coordinated test, and remote test. CTMF also defined two testing contexts: the single-party testing context and the multi-party testing context.

Figure ?? shows the CTMF testing framework of multi-party testing context using distributed test architecture [?]. This configuration possibles an IUT to have several nodes running at the same time. The testing framework is made up of Upper Testers (UTs), Lower Testers (LTs), a Lower Tester Control Function (LTCF) and Coordination Points (CPs). The UTs emulate the action of the IUT user while the LTs act on lower services simulating stubs invoked by IUT through Protocol Data Unit (PDU) messages. The LTCF coordinates the actions of multiple LTs and generates test verdict of the test case gathering information from all the LTs after they stopped the test case execution. UTs, LTs and LTCF communicate with each other by CPs, while UTs and LTs communicate with SUT through Points of Control and Observation (PCOs).

C. Testing MapReduce Based Systems

The goal of this research challenge is to evaluate the MR implementation in extreme conditions, for instance, with node failures during its execution or only with the failure of one determined map task. To reproduce this behavior it is necessary an individual and complete control of each MR Implementation component, i.e. master and workers (see Figure ??). Moreover, if one wants to drop only a determined map task for example, it is necessary to identify which worker is executing this task. In fact, we state five testing properties required to effectively test MR based systems: (1) dynamic testing; (2) distributed testing; (3) to reproduce a real-world environment; (4) inspect ability; and (5) MR source-code isolation.

First, dynamic testing is necessary because only during its execution is possible to evaluate the system considering real problems, such as nodes failures, network traffic, high latency, performance variations, etc. These failures and properties may interfere on the test case result and we only can measure them through the MR system execution.

Second, with distributed testing, the test cases shall be decomposed in actions to be executed individually and in-
dependently by each MR Implementation component. This property is very important for MR based systems because each component has a different action to do, e.g. while one worker is applying a map function another is applying a reduce function. Moreover, we can apply a test case where only one node has your response time decrease (by fault injection) to verify if the MR Implementation is really fault tolerant.

Third, the goal of reproducing a real-world environment is to enable test cases to be deployed considering: volatility control, fault injections, and MR systems scalability. Volatility control is the ability of reproducing the addition and removal of nodes in a controlled manner. Fault injections are necessary to include system perturbation that may lead to a system wrong behavior (e.g., high latency, noise, etc). And the framework must be as scalable as MR based systems.

Fourth, inspect ability is the capacity of control and monitor the MR components individually and completely. This allows to put the components in any state (i.e., running, idle, or stopped) and also monitors their activity at any time. For instance, to query a specific state of a Map, Reduce or Master task.

Finally, the goal of SUT source-code isolation is to avoid that the MR source code is altered by the test because this could add new bugs. If the MR source code is altered and it is detected a failure while testing, it can be difficult to know if the failure was caused by the alteration or by the code instrumentation. Although code instrumentation for testing may not easily introduce new bugs, we agree that it is an important property to be reached.

IV. HADOOPTEST: MAPREDUCE TESTING FRAMEWORK

In this section, we present HadoopTest, an integrated solution for testing MR based systems under abnormal conditions. This solution is based on a framework that controls individually each MR components, and combines fault injection with functional tests. Our solution makes possible to create and deploy test case to evaluate the MR based systems considering real problems, such as nodes failures, network traffic, high latency, performance variations, etc. For instance, it enables to create and deploy a simple test case where one determined map task must be interrupted during the MR execution for evaluating the system behavior. This test case is adequate to dispatch the actions of a test case through the testers and to maintain a list of unavailable components. The role of the tester is to execute test case actions and to control the MR component. In practice, each tester receives the description of the overall test sequence and is thus able to know when to apply a local execution sequence on MR component. One tester must controls the MR master due to centralized coordination of MR jobs execution. The others testers are responsible to control each MR worker.

Each worker can be assigned to execute a map task or a reduce task. Then, in addition to distinguish the master and worker components, we need to identify which worker is executing a map or a reduce task, e.g. if we want to interrupt only one determined single map task.

Figure ?? shows the MR testing architecture applied to a MR implementation instance. There are one coordinator and six testers, identified by t0...t5. The coordinator is not associated directly by MR Implementation, it controls the tester actions that use MR component interfaces. The t0 controls the MR master and, consequently, it is responsible for the MR execution and assigning the map and reduce task for each worker. The other components, t1...t5, control each worker instance through MR worker interfaces.

Distributed testing can be done since each tester can be
individually and explicitly controlled by test case, for instance, a test case can indicate that one worker is dropped out. It is possible to deploy fault injections applying lower testers functions on testers. This also enables to put the MR components in any state (i.e. running, idle, or stopped) and monitors their activity at any time. All this deployment can be done without the MR source code alteration by the test.

B. MR Available Interfaces/PCOs

The first thing to do for testing MR based systems it is to identify the Points the Control and Observation (PCO) available. Figure ?? shows the available PCOs in MR based systems identified by the steps of MR execution. The five possible PCOs between these steps are: (1) at the begging, before initiate all MR components; (2) between initiate all MR components and load input; (3) between load input and submit job; (4) between submit job and extract output; (5) after extract output finalizing the MR execution.

![Fig. 6. MR Execution PCOs](image)

These PCOs make possible to deploy only simple test cases submitting a job and evaluating your final result. Our objective is to evaluate MR based systems influencing them during job execution, e.g. to drop a determined map task while running. To reproduce this behavior it is necessary PCOs at each MR Implementation component, i.e. master and workers (see Figure ??). Originally, the first PCO available enables to initialize and to identify them, but during the execution it is not possible to control and monitor them. Moreover, if one wants to drop only a map or reduce task, it is necessary to identify which worker is executing this task.

C. Definitions

Let us denote by $C$ the set of MR Implementation components that must be tested. It is necessary to identify these components because they are responsible for different activities and they have different interfaces.

**Definition 4.1 (MR Components):** A set of MR components, noted $C$, is a collection of a MR master, noted $M$, and a set of MR workers, noted $W = \{W_0, ..., W_n\}$. For simplicity, $\forall W_i \in W$, $W_i$ keeps a DFS slave instance, since we aim at testing only MR implementation.

We denote by $UT$ and $LT$ the set of Upper Tester and Lower Tester, respectively. Each pair of $UT$ and $LT$ are responsible for controlling and observing each MR component. $UT$ acts as SUT users while $LT$ acts as lower services or stubs.

**Definition 4.2 (Upper and Lower Testers):** A set of Upper and Lower testers is defined by $UT$ and $LT$, respectively;

$$\exists (UT_i \land LT_i) \forall C_i \in C$$

We denote by $UT$ and $LT$ the set of Upper Tester and Lower Tester, respectively, where $|UT| = |C|$ and $|LT| = |C|$. Each pair of $UT$ and $LT$ are responsible for controlling and observing each MR component. $UT$ acts as SUT users while $LT$ acts as lower services or stubs.

$GTC$ is the global test case that verifies the SUT, and $A$ is the set of actions executed by $GTC$ on $C$. With the global test case must be possible to test MR based system respecting its work-flow.

**Definition 4.3 (Global Test Case):** A Global Test Case noted $\tau$ is a tuple $\tau = (A^\tau, T^\tau, L^\tau, S^\tau)$ where:

- $A^\tau$ is an ordered set of actions $\{a_0, ..., a_n\}$, where $A^\tau \subseteq A$;
- $T^\tau$ is a set of testcases, where $T^\tau \subseteq T$;
- $L^\tau$ is a set of local verdicts; and
- $S^\tau$ is a schedule.

If a global test case contains a coordination message from $p$ to $q$ before an input $?i_q$ at $q$ then the tester at $q$ waits to receive this coordination message before sending $?i_q$. trata o problema de quando executar o delete depois do insert, mas isso não ocorre no MR

**Definition 4.4 (Action):** A test case action is a tuple $a^\tau_i = (h, I, \iota, T', S, D)$ where:

- $h$ is a hierarchical level;
- $I$ is a set of instructions;
- $T'$ is a set of testcases that executes the action, where $T' \subseteq T$;
- $D$ is a set of dependent actions;
- $\iota$ is the interval of time in which $I$ must be executed.

$D$ is a set of actions that must be executed before $a^\tau_i$, where $\forall a_j \in D : (D \subseteq A^\tau) \land (j < i), v_{a_j}^\tau \neq error$.

In distributed systems, the autonomy and the heterogeneity of nodes interfere directly in the execution of service requests. While close nodes may answer quickly, distant or overloaded nodes may need a considerable delay to answer. Consequently, clients do not expect to receive a complete result, but the available results that can be retrieved within a given time. Thus, test case actions (Definition ??) must not wait indefinitely for results, but specify a maximum delay (timeout) for an execution. The instructions are typically calls to the system components interface, as well as any statement in the test case programming language. A test sequence is a sequence of messages (e.g. service requests) exchanged among peers along testing.

The Schedule is a map between actions and sets of testers, where each action corresponds to the set of testers that execute it.

**Definition 4.5 (Schedule):**

- $A$ is a set of actions;
- $T'$ is a set of testcases that executes the action, where $T' \subseteq T$;
- $S$ is a necessary stimulus to execute $a^\tau_i$;
\[ D \text{ is a set of actions that must be executed before } a^n_i, \]

where \( \forall a_j \in D : (D \subseteq A^*) \land (j < i), v^n_i \neq \text{error}; \)

A schedule is a map \( S = A \rightarrow \Pi, \) where \( \Pi \) is a collection of
tester sets \( \Pi = \{T_0, \ldots, T_n\}, \) and \( \forall T_i \in \Pi : T_i \subseteq T \)

Definition 4.6 (Test Sequence): A test sequence is a se-
quence of test actions \( TS(A), \) where each action \( a^n_i \) has a
hierarchical level \( h^n_i. \) Actions with lower levels are executed
before actions with higher levels.

Definition 4.7 (Local verdict): A local verdict \( l^n_i \) is given by
comparing the expected result, noted \( E, \) with the output,
noted \( O. \) \( E \) and \( O \) may be a single value or a set of values
from any type. However, these values must be comparable. The
local verdict \( l \) of a tester \( t \) of action \( a \) is defined as follows:

\[
l^n_i = \begin{cases} 
\text{pass,} & \text{if } O = E \\
\text{fail,} & \text{if } O \neq E \\
\text{inconclusive,} & \text{if } O = \emptyset 
\end{cases}
\]

Once all actions of a test case \( \tau \) have finished their execu-
tion, \( \tau \) is able to construct its global verdict \( V^\tau, \) where it is
built based on the local verdicts of all testers \( t \in T^\tau. \)

D. Test Case Example

Table ?? shows a simple MR test case to illustrate the
usability of HadoopTest. The goal of this test case is to detect
errors on a MR implementation. More precisely, it permits
evaluate the operation of handling worker failures, i.e., the
fault tolerant solution. This test case is deploy based on MR
problem configuration also showed by Figure ??.

This test case involves six testers \( T = \{ t_0, \ldots, t_6 \} \) and
seven actions \( A = \{ a_0, \ldots, a_6 \}. \) The tester \( t_0 \) executes the
action \( a_0 \) that initiates the MR master component. After, the
other testers, \( \{ t_1, \ldots, t_5 \}, \) execute the action \( a_1 \) initiates the
MR worker components. The action \( a_2 \) is only executed by
tester \( t_0 \) that submits the job to MR master component. During
MR job execution, the tester \( t_2 \) executes the action \( a_3 \) and
is dropped from the system. Then, the tester \( t_0 \) executes the action \( a_4 \) that evaluates the output. If the output data is
the same as the expected, then the verdict is \text{pass}. Otherwise,
it is \text{fail}. If \( t_0 \) is not able to retrieve any output data, then
the verdict is \text{inconclusive}. Then, the testers \( \{ t_1, t_3, t_4, t_5 \} \)
execute the action \( a_5 \) where they are stopped. Finally,
the tester \( t_0 \) executes the action \( a_6 \) that finishes the test case.

E. Execution

The algorithm has three steps: registration, action execu-
tion and verdict construction. Before the execution of a \( \tau, \)
each \( t \in T \) registers its actions with the \text{coordinator}. For
instance, in Example ??, tester \( t_2 \) may register the actions
\( A' = \{ a_1, a_2, a_6 \}. \) Once the registration is finished, the
\text{coordinator} builds the schedule, mapping the actions with their
related subset of testers. In our example, action \( a^3 \) is mapped
to \( \{ t_3, t_4 \}. \)

Once \( S \) is built, the coordinator traverses all test cases \( \tau \in
DTS \) and then the actions of each \( \tau. \) For each action \( a^n_i, \) it uses
\( S(a^n_i) \) to find the set of testers that are related to it and sends
the asynchronous message \text{execute}(a^n_i)\forall t \in S^\tau(a^n_i). \) Then, the
coordinator waits for the available testers to inform the end
of their execution. The set of available testers corresponds to
\( S^\tau(a) - T_u, \) where \( T_u \) is the set of unavailable testers. In our
example, once \( a_1 \) is finished, testers \( \{ t_0, t_1, t_2 \} \) inform the
\text{coordinator} of the end of the execution.

\begin{verbatim}
Algorithm 1: A dependent test case execution
Input: \( A^\tau, \) an ordered set of actions; \( T^\tau, \) a set of
    testers; \( S^\tau, \) a set of dependent
    actions; \( L^\tau, \) a set of local verdicts
Output: a global verdict
\[ S^\tau \leftarrow \emptyset; \]
\[ \varepsilon^\tau \leftarrow \emptyset; \]
\text{foreach} \( t \in T \) \text{do}
\[ S^\tau \leftarrow \text{register}(t, A^\tau); \]
\text{foreach} \( a \in A^\tau \) \text{do}
\[ \text{error} \leftarrow \text{false}; \]
\text{foreach} \( d \in D(a) \) \text{do}
\text{if} \( d \in \varepsilon^\tau \) \text{then}
\[ \text{error} \leftarrow \text{true}; \]
\text{break;}
\text{if} \( \text{error} \) \text{then}
\text{foreach} \( t \in S^\tau(a) \) \text{do}
\[ L^\tau(t, a) \leftarrow \text{execute}(a); \]
\text{foreach} \( l \in L^\tau(a) \) \text{do}
\text{if} \( l \neq \text{pass} \) \text{then}
\[ \varepsilon^\tau \text{.add}(a); \]
\text{break;}
\text{else}
\[ a.\text{setResult}(	ext{ERROR}); \]
\[ \varepsilon^\tau \text{.add}(a); \]
\text{return oracle} \( \varepsilon^\tau \); \end{verbatim}

Thus, the \text{coordinator} knows that \( a_1 \) is completed and
the next action can start. When a tester \( t \in T^\tau \) receives
the message \text{execute}(a^n_1), \) it executes the suitable action.
If the execution succeeds, then a message \text{ok} is sent to the
\text{coordinator}. Otherwise, if the action timeout is reached, then
a message \text{error} is sent. Once the execution of \( \tau \) finishes, the
\text{coordinator} asks all testers for a local verdict. In the example,
if \( t_3 \) gets the correct string "fourteen" in \( a_5, \) then its local
verdict is \text{pass}. Otherwise, it is \text{fail}.

After receiving all local verdicts, the \text{coordinator} is able to
assign a verdict \( L^\tau. \) If any local verdict is \text{fail}, then \( L^\tau \)
is also \text{fail}, otherwise the \text{coordinator} continues grouping
each \( l^n_\tau \) into \( L^\tau. \) When \( L^\tau \) is completed, it is analyzed to
decide between verdicts \text{pass} and \text{inconclusive} as described in
Algorithm ??. This algorithm has two inputs, a set of local
verdicts \( L \) and an index of relaxation \( \varphi, \) representing the
level of acceptable \text{inconclusive} verdicts. If the ratio between
the number of \text{pass} and the number of local verdicts is greater
than \( \varphi, \) then the verdict is \text{pass}. Otherwise, the verdict is
TABLE I
MAPREDUCE SIMPLE TEST CASE

<table>
<thead>
<tr>
<th>Action</th>
<th>Hierarchy</th>
<th>Tester</th>
<th>Instructions</th>
<th>Dependency</th>
<th>Timeout</th>
</tr>
</thead>
<tbody>
<tr>
<td>a₀</td>
<td>0</td>
<td>t₀</td>
<td>startMaster()</td>
<td>∅</td>
<td>100</td>
</tr>
<tr>
<td>a₁</td>
<td>1</td>
<td>t₁₋₅</td>
<td>startWorkers()</td>
<td>{a₀}</td>
<td>1000</td>
</tr>
<tr>
<td>a₂</td>
<td>2</td>
<td>t₂</td>
<td>sendJob()</td>
<td>{a₀}</td>
<td>1000000</td>
</tr>
<tr>
<td>a₃</td>
<td>2</td>
<td>t₂</td>
<td>dropWorker()</td>
<td>{a₀}</td>
<td>1000</td>
</tr>
<tr>
<td>a₄</td>
<td>3</td>
<td>t₀</td>
<td>assert(output, expected)</td>
<td>{a₂}</td>
<td>10000</td>
</tr>
<tr>
<td>a₅</td>
<td>4</td>
<td>t₁₋₅</td>
<td>stopWorkers()</td>
<td>{a₀,a₃}</td>
<td>1000</td>
</tr>
<tr>
<td>a₆</td>
<td>5</td>
<td>t₀</td>
<td>stopMaster()</td>
<td>{a₀}</td>
<td>1000</td>
</tr>
</tbody>
</table>

Algorithm 2: A hierarchical test case execution

Input: \( A^\tau \), a set of actions; \( T^\tau \), a set of all testers; \( TA^\tau \), a set of testers to each action; \( S^\tau \), a schedule; \( L^\tau \), a set of local verdicts; \( O^\tau \), a incremental action number sequence; \( M^\tau \), a mapping order by actions.

Output: \( V^\tau \), a global verdict
\( S^\tau \leftarrow \emptyset \);
\( \varepsilon^\tau \leftarrow \emptyset \);
\( \mathcal{L}^\tau \leftarrow \emptyset \);

foreach \( t \in T \) do
    \( S^\tau \leftarrow \text{register}(t, A^\tau) \);
    foreach \( a \in S^\tau(t) \) do
        \( S^\tau(h) \leftarrow \text{register}(H^\tau(a), A^\tau) \);

foreach \( h \in S^\tau(A^\tau) \) do
    foreach \( a \in S^\tau(h) \) do
        foreach \( t \in S^\tau(a) \) do
            send execute(a) to t;
        foreach \( l \in L^\tau(a) \) do
            if \( l \neq \text{PASS} \) then
                \( \varepsilon \text{.add}(a) \);
                break;
    return oracle(\( L^\tau \));

Algorithm 3: A global test case execution

Input: \( A^\tau \), an ordered set of actions; \( T^\tau \), a set of testers; \( D^\tau \), a set of dependent actions; \( H \), an hierarchical level

Output: a global verdict
\( S^\tau \leftarrow \emptyset \);
\( \varepsilon^\tau \leftarrow \emptyset \);
\( \mathcal{L}^\tau \leftarrow \emptyset \);

foreach \( t \in T \) do
    \( S^\tau \leftarrow \text{register}(t, A^\tau) \);
    foreach \( a \in S^\tau(t) \) do
        \( S^\tau(h) \leftarrow \text{register}(H^\tau(a), A^\tau) \);

foreach \( h \in S^\tau(A^\tau) \) do
    foreach \( a \in S^\tau(h) \) do
        foreach \( d \in D^\tau(a) \) do
            if \( d \in \varepsilon^\tau \) then
                error \leftarrow \text{FALSE} ;
                a.setResult(\text{ERROR}) ;
                \( \varepsilon \text{.add}(a) \);
                break;
        if \( \{ \text{error} \} \) then
            foreach \( t \in S^\tau(a) \) do
                send execute(a) to t;
        foreach \( l \in L^\tau(a) \) do
            if \( l \neq \text{PASS} \) then
                \( \varepsilon \text{.add}(a) \);
                break;

return oracle(\( L^\tau \));

inconclusive.

V. EXPERIMENTAL VALIDATION

In this section, we present the results of three experiments used for validating HadoopTest implementation effectiveness. Initially, we present two MR applications and the test cases that were considered during the experiments. Then, the HadoopTest integrity and scalability are empirically proved through the comparison with Hadoop, on section ?? . Finally, on section ?? , we present the HadoopTest effectiveness on failure identification from the evaluation of mutant systems.

All of our experiments were executed on the Grid5000 platform\(^1\) using clusters running Debian GNU/Linux. The cluster nodes were connected by a 10 Gbps network and they had a similar configuration: 2 Intel Xeon 2.33GHz dual-core processors, 4 GB RAM memory and 80 GB SATA HD.

A. Case Study

Our experiments were implemented testing the Hadoop, a open-source MapReduce implementation maintained by Apache Foundation [7]. We validate the HadoopTest implementation testing two popular applications distributed by

\(^1\)Grid 5000 Platform: http://www.grid5000.fr
Hadoop, PiEstimator and WordCount. These applications are simple and can have your executions evaluate in details for ensuring the implementation integrity and the tests realized.

The PiEstimator application calculated the $\pi$ value using the Monte Carlo method, that is based on considering a circle exactly inscribed inside a square with side length 1. The map function randomly creates points inside the square and identifies the points placed inside the circle and the points placed outside it. The reduce function accumulates the points placed inside ($I$) and outside ($O$) identified by the map function. The fraction $I/T$, let $T = I + O$, is obtained the approximation of the circle area by the square area, where area of the inscribed circle is $\pi/4$ and the area of unit squid is 1. So, the estimated $\pi$ value is obtained by $4 \times (I/T)$ from the reduce results.

The WordCount application get the total occurrences of each word into one or several text files. The Map function get a portion of the input file, divides it on words and for each word it emits a pair composed by the word and the number 1. The Reduce function sums the counts for each word and emits a pair with the word and sum.

Table ?? shows the actions identified to compose a common test case for testing these two applications. A common test case is possible since the difference between then is only the submission and result validation process, action $a_0$. Initially, the action $a_1$ is executed only at tester $t_0$, which initializes the Distributed File System (DFS) master component. Next, the action $a_2$ is executed everywhere initializing the DFS slave components. The action $a_3$ is executed at tester $t_0$ initializing the Hadoop master component, called JobTracker. Then, the action $a_4$ is executed everywhere initializing the Hadoop worker components, that can execute both a Map or Reduce task, and are called TaskTrackers. After these actions the Hadoop is ready to execute tasks.

<table>
<thead>
<tr>
<th>TABLE II</th>
<th>TEST CASE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action Seq.</td>
<td>Tester</td>
</tr>
<tr>
<td>$a_0$</td>
<td>$t_0$</td>
</tr>
<tr>
<td>$a_1$</td>
<td>$t_0$</td>
</tr>
<tr>
<td>$a_2$</td>
<td>$t_0$</td>
</tr>
<tr>
<td>$a_3$</td>
<td>$t_0$</td>
</tr>
<tr>
<td>$a_4$</td>
<td>$t_0$</td>
</tr>
<tr>
<td>$a_5$</td>
<td>$t_0$</td>
</tr>
<tr>
<td>$a_7$</td>
<td>$t_0$</td>
</tr>
</tbody>
</table>

So, the tester $t_0$ execute the action $a_5$ that submits a job to the Hadoop master component and after the job execution, your result is validated with the expected result. Finally, the action $a_6$ and $a_7$, are executed to stop the master and slave components, respectively. The action $a_6$ is executed at all testers and the action $a_7$ is only executed by the tester $t_0$.

At end of the test case we have a Pass verdict if all actions had success on your executions, otherwise we have Fail, or Inconclusive if it can not do assertions about the results. Normally, the Inconclusive verdict is got when the execution timeout is expired, since the timeout is defined on all actions to ensure its execution finish.

B. Implementation

We conduct the HadoopTest implementation extending of the PeerUnit framework [?]. As seen before (Section ??) this P2P testing framework provides a scalable testing driver that makes possible to deploy test cases in a distributed manner without harming the test results. Then, we extend it to testing MR based systems and here following we show how the test case was written.

PeerUnit is implemented in Java and makes extensive use of dynamic reflection and annotations, using these features to
select and execute distributed test case actions in a controlled manner. The test case is implemented by a class where the test case actions are implemented as annotated methods, i.e., methods adorned by a tag or annotation. The available annotations are: BeforeClass, identifies that the method is first executed in the test case; TestStep, identifies that the method is a test case action; and AfterClass, identifies that the method is last executed in the test case.

We deploy a class named TestCaseExample, which is a subclass of AbstractMR. The AbstractMR class implements the MR and DFS library, and provides access to your interfaces, i.e., the methods that abstract the MR and DFS programing complexity. For instance, the method initMRmaster(), that starts the MR master component; and the method initMRworker(), that starts the MR worker component. This class contains an attribute named expected that is initiated with the expected result.

```java
public class TestCaseExample extends AbstractMR{
    private Object expected = getOutputExpected();

    public void init() {
        readConfiguration();
    }

    The Distributed File System (DFS) server is initiated by the method startDFSSmaster(). The StepClass annotation, and your parameters range and timeout, ensures that this method is executed at first, at all testers and it can runs up to 60000 milliseconds.

    @BeforeClass(range = "*", timeout = 60000)
    public void init() {
        readConfiguration();
    }

    The Distributed File System (DFS) server is initiated by the method startDFSSmaster(). The StepClass annotation, and your parameters range and timeout, ensures that this method is executed at first, at all testers and it can runs up to 60000 milliseconds.

    @BeforeClass(range = "*", timeout = 60000)
    public void init() {
        readConfiguration();
    }

    The Distributed File System (DFS) server is initiated by the method startDFSSmaster(). This method is executed everywhere (range="*").

    @TestStep(order=2, range = "*", timeout = 60000)
    public void startDFSSlave() {
        // Method implementation
    }

    At this moment the DFS is ready and the MR Implementation can be initiate. Then, the method startMRmaster() is executed only at tester 0, where is instantiate the MR master component.

    @TestStep(order=3, range = "0", timeout = 60000)
    public void startMRmaster() {
        initMRmaster();
    }

    After, the method startMRworker is executed everywhere, where are instantiate all MR worker component.

    @TestStep(order=4, range = "+", timeout = 60000)
    public void startMRworker() {
        initMRworker();
    }

    Now, the MR system is ready to execute a job. The method submitJob() is executed only at tester 0, where it submits a job to MR master and verifies the output. If the output object corresponds to the expected, the test case passes, otherwise it fails.

    @TestStep(order=5, range = "0", timeout = 60000)
    public void submitJob() {
        output = execJob(Job);
        assert (expected , result );
    }

    The follow methods, stopSlaves and stopMasters, stimulate the DFS and MR components to exit the system.

    @TestStep(order=6, range = "+", timeout = 60000)
    public void stopSlaves() {
        exitSlaves();
    }

    @TestStep(order=7, range = "0", timeout = 60000)
    public void stopMasters() {
        exitMasters();
    }

C. Integrity and Scalability

Our first experiment objectives to ensure the solution integrity and scalability when executing a test case. Then, we run the PiEstimator directly by Hadoop comparing its execution with running by HadoopTest. The items evaluated were the results and the execution durations. For a better analysis we used 100 machines to perform this experiment. We performed tests with 2, 20, 50 and 100 machines and the results obtained on all of them in the Hadoop implementation was the same as presented in the HadoopTest implementation. This statement shows the HadoopTest integrity where it does not influence the result of the applications tested.

The execution time through HadoopTest had a little increase if compared with the execution time through Hadoop. The figure ?? shows the PiEstimator execution time on Hadoop and on HadoopTest varying the Map functions number on each execution. These executions time were obtained executing the application on 50 machines. Other executions were performed varying the machines number but the influence was less clearly than the presented here.

The initial experiment behavior, manipulating problems until 500 Maps, is justifiable by the overload to the testers maintain the control over the applications execution. As the machines number is fixed and the single processing is very small, this overload is influenced by the network status, because the higher time is spent on message exchange for communication and synchronization. When handled major problems which require more individual processing of nodes, the variance was lower and more constant. Although that influence difficult the performance test, it is acceptable to realize other tests which require greater control of application.
D. Testing Effectiveness

We applied Mutation Testing (or Mutation Analysis) to evaluate our solution effectiveness for identifying system failures. This testing technique is based on injecting faults (defects or bugs) into the SUT. One new system is created for each fault injected, resulting on a set of system mutants. The goal is to detect the difference between the original system and the mutant systems. The result from this operation define if a test tool is able to identify failures or not [?].

We used the Java bytecode engineering library - ASM [?] - to generate the mutant systems based on the alteration of mutants operators existing on original source-code. Each mutant operator represents a class of faults that can be inserted on the source-code, e.g. arithmetic mutant operator can be to replace occurrences of ‘+’ by ‘−’. We deployed the arithmetic and logic operators using the ASM library. Then, the ASM library generates one new mutant system for each operator founded on the main class informed.

The original system and the mutants were submitted on HadoopTest to it identify differences between the behavior of original application and mutated ones. This comparison was made by a specific test case executed over the HadoopTest which evaluated the final result of these executions. To evaluate if our solution was efficient to identify the mutants on mutated applications we considered three mutants kinds: dead mutants, alive mutants and equivalent mutants. Dead mutants are the mutations which were identified by a test case. In that case, the test case and the test tool are evaluated as efficient, because they are able to identify that mutants. The alive mutants are mutations that could not be identified by a test case, but can be identified by another one. Therefore, alive mutants do not characterize a test tool failure, they are a consequence of a incomplete test case. Equivalent mutants differ of alive mutants because they never will be identified, save using specific techniques. They usually are semantic errors which does not influence the application result. So, this mutant type need a method to compare original function and mutated one to identify where is the mutation. However, we does not aim to identify equivalent mutants on our solution, we used mutations only to validate our architecture. Therefore, to identify this mutant type we inspected manually the bytecode of mutated applications.

The Table ?? shows the results from test case executions using 13 mutants, nominated M0 to M12, which were generated on PiEstimator application. The approximated π value returned by the original application was 3.1416 and only the mutants M1, M6, M7 e M12 returned the same value. Thus, these mutants had the Pass verdict on test case execution while the other mutants obtained the Fail verdict. We did not obtain Inconclusive verdicts because we configured the timeout on test case higher than the applications execution time. The mutants M0, M2, M3 and M8 modified the applications execution parameters which interfered on their executions. These executions do not returned results, so we considered their results as NULL, generating Fail verdicts. The mutants M4, M5, M10 and M11 changed the π calculation parameters. On applications with these mutants the approximated π value was changed, so their verdicts were Fail.

<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>

To the WordCount application were generated seven mutants from the application main class. Five mutants affected the application result, so they received the Fail verdicts. The other three mutants obtained the Pass verdict because their changes do not influenced the word count operations nor the Hadoop processing. On WordCount application we also had not Inconclusive verdicts.

Figure ?? shows the evaluation of PiEstimator and Wordcount mutants. the test case results from these two applications, considering their mutations. The graph demonstrates that the HadoopTest killed the most of the mutants from these two applications. Our implementation did not only identify the mutants considered equivalents. It is because the test case used is not complex and only evaluated the result from applications. On that case, since the equivalent mutants changed semantic operators, the application result was not affected, thus the HadoopTest test case used did not identify these mutants. If we use a more specific test case it may identify these mutants.
But we only used the mutants to validate the ability from HadoopTest to identify them, so the test cases to identify equivalent mutants is not our goal. On that case, if we do not consider the equivalent mutants, our solution was 100% effective to kill the mutants on mutated applications.

VI. CONCLUSION

In this paper we proposed a integrated solution to test MapReduce systems, which we called HadoopTest. This solution considers five main aspects: dynamic test, distributed test, real-world environment, inspect ability and not System Under Test (SUT) contamination. On HadoopTest we aim to improve tests on MapReduce systems considering these five aspects.

Our solution implements Point of Control and Observation (PCO)s over MapReduce components. These PCOs is made through Upper Testers (UT) which provide the interfaces to execute your test cases on MapReduce components. The MapReduce components were implemented through the Hadoop, a open source MapReduce implementation.

Through the PCOs the HadoopTest is able to execute dynamic tests and execute these tests on a distributed ways. The PCOs also initialize each MapReduce component, controlling and monitoring them, regarding concepts of a real world. For now, to provide the PCOs we only made UTs, they using the Hadoop API, so we do not inserted additional code on the SUT, therefore we do not contaminate it.

To validate our solution we realized some experiments. They validate three dimensions of HadoopTest: the effectiveness to identify defects on applications, integrity and scalability. The WordCount and PiEstimator applications were used on tests. These tests aimed to test the influence of HadoopTest over the MapReduce systems executions. The first one executed the applications over a real MapReduce cluster and after over the HadoopTest implementation and the time of two executions were compared. The execution through HadoopTest increased a bit the time, this increasing was not significant and may not disturb the scalability of the test executions. The result on the two executions is the same, so the HadoopTest did not influence the test result. We also executed some mutations on these applications to test the effectiveness of HadoopTest to detect defects on MapReduce systems. Our solution was able to identify the most part of mutations inserted on tested applications. It does not identify only mutants considered as equivalent which need to specific test cases to be identified.

A new challenge for our solution is the LT implementation. Using only upper testers the HadoopTest is limited on control and monitoring methods, because we only can use the methods provided by the Hadoop API. To provide a better control and monitor over the MR components we will implement LTs. They will control the MR components on the lower layers as Java Virtual Machine (JVM) on Java applications, for example. Other feature to be improved is the action executions. Our test case executions follow a sequence of actions and these actions are executed only on a serial mode. On some situations as the workers being monitored while a job is submitted, for example, we need to execute more than one action in parallel. So, a other new challenge is to restructure the actions synchronization to enables the execution of many actions at same time. On future works we too propose the implementation of a scenario with more than one master component. This is a requirement to MapReduce future versions which will execute with many masters.