Process Mining Monitoring for Map Reduce Applications in the Cloud

Federico Chesani, Anna Ciampolini, Daniela Loreti and Paola Mello
DISI - Department of Computer Science and Engineering, University of Bologna, Viale del Risorgimento 2, Bologna, Italy

Keywords: Business Process Management, Map Reduce, Monitoring, Cloud Computing, Autonomic System.

Abstract: The adoption of mobile devices and sensors, and the Internet of Things trend, are making available a huge quantity of information that needs to be analyzed. Distributed architectures, such as Map Reduce, are indeed providing technical answers to the challenge of processing these big data. Due to the distributed nature of these solutions, it can be difficult to guarantee the Quality of Service: e.g., it might be not possible to ensure that processing tasks are performed within a temporal deadline, due to specificities of the infrastructure or processed data itself. However, relaying on cloud infrastructures, distributed applications for data processing can easily be provided with additional resources, such as the dynamic provisioning of computational nodes. In this paper, we focus on the step of monitoring Map Reduce applications, to detect situations where resources are needed to meet the deadlines. To this end, we exploit some techniques and tools developed in the research field of Business Process Management: in particular, we focus on declarative languages and tools for monitoring the execution of business process. We introduce a distributed architecture where a logic-based monitor is able to detect possible delays, and trigger recovery actions such as the dynamic provisioning of further resources.

1 INTRODUCTION

The exponential increase in the use of mobile devices, the wide-spread employment of sensors across various domains and, in general, the trending evolution towards an "Internet of everything", is constantly creating large volumes of data that must be processed to extract knowledge. This pressing need for fast analysis of large amount of data calls the attention of the research community and fosters new challenges in the big data research area (Chen et al., 2014b). Since data-intensive applications are usually costly in terms of CPU and memory utilization, a lot of work has been done to simplify the distribution of computational load among several physical or virtual nodes and take advantage of parallelism.

Map Reduce programming model (Dean and Ghemawat, 2008) has gained significant attraction for this purpose. The programs implemented according to this model can be automatically split into smaller tasks, parallelized and easily executed on a distributed infrastructure. Furthermore, data-intensive applications requires a high degree of elasticity in resource provisioning, especially if we deal with deadline constrained applications. Therefore, most of the current platforms for Map Reduce and distributed computation in general (Apache Hadoop, 2015; Apache Spark, 2015) allow to scale the infrastructure at execution time.

If we assume that the performance of the overall computing architecture is stable and a minimum Quality of Service (QoS) is guaranteed, Map Reduce parallelization model makes relatively simple to estimate a job execution time by on-line checking the execution time of each task in which the application has been split – as suggested in the work (Mattess et al., 2013). This estimation can be compared to the deadline and used to predict the need for scaling the architecture.

Nevertheless, the initial assumptions are not always satisfied and the execution time can differ from what is expected depending on either architectural factors (e.g., the variability in the performance of the machines involved in the computation or the fluctuation of the bandwidth between the nodes), or domain-specific factors (e.g., a task is slowed down due to the input data content or location). This unpredictable behavior could be run-time corrected if the execution relayed on an elastic set of computational resources as that provided by cloud computing systems. Offering "the illusion of infinite computing resources available on demand" (Armbrust et al., 2009), cloud computing is the ideal enabler for tasks characterized by a large and variable need for computational power.

Cloud computing is indeed knowing a wide success in a plethora of different applicative domains, thanks to the maturity of standards and implementations. Usually, the cloud is the preferred choice for applications that must comply to a set of contract terms and functional and non-functional requirements specified by a service level agreement (SLA). The complexity of the resulting overall system, as well as the dynamism and flexibility of the involved processes, often require an on-line operational support checking compliance. Such monitor should detect when the overall system deviates from the expected behavior, and raise an alert notification immediately, possibly suggesting/executing specific recovery actions. This run-time monitoring/verification aspect – i.e., the capability of determining during the execution if the system exhibits some particular behavior, possibly compliance with the process model we have in mind – is still matter of an intense research effort in the emergent Process Mining area. As pointed out in (Van Der Aalst et al., 2012), applying Process Mining techniques in such an online setting creates additional challenges in terms of computing power and data quality.

Starting point for Process Mining is an event log. We assume that in the architecture going to be analyzed it is possible to sequentially record events. Each event refers to an activity (i.e., a well-defined step in some process/task) and it is related to a particular process instance. Note that, in case of a distributed computation, we also need extra information such as, for instance, the resource/node executing, initiating and finishing the process/task, the timestamp of the event, or other data elements.

While, in an cloud architecture, several tools exist for performing a generic, low-level monitoring task (Ceilometer, 2015; Amazon Cloud Watch, 2015), we also advocate the use of an application-/process-oriented monitoring tool in the context of Process Mining in order to run-time check the conformance of the overall system. Essentially, the goal of this work is to apply the well-known Process Mining techniques to the monitoring of complex distributed applications, such as Map Reduce in a cloud environment.

Since Map Reduce applications typically operate in dynamic, complex and interconnected environments demanding high flexibility, a detailed and complete description of their behavior seems to be very difficult, while the elicitation of the (minimal) set of behavioral constraints/properties that must be respected to correctly execute the process (and that cannot be directly incorporated at design time into the system) can be more realistic and useful. Therefore, in this context, we will adopt a verification framework based on constraints, called MOBUCON EC (Monitoring business constraints with Event Calculus (Montali et al., 2013b)), able to dynamically monitor streams of events characterizing the process executions (i.e., running cases) and check whether the constraints of interest are currently satisfied or not. MOBUCON is an extension of the constraint-based Declare language (Pesic and van der Aalst, 2006) and is data aware. This allows us to specify properties of the system to be monitored involving time constraints and task data. The Event Calculus (EC) formalization has been proven a successful choice for dealing with runtime verification and monitoring, thanks to its high expressiveness and the existence of reactive, incremental reasoners (Montali et al., 2013b).

This work presents an on-line monitoring system to check the compliance of each node of a distributed infrastructure for data processing running on a cloud environment. The resulting information is used for taking scaling decisions and dynamically recovering from critical situations with a best effort approach (by means of an underlying previously implemented infrastructure layer). This could be considered as a first step towards a Map Reduce engine with autonomic features either in run-time detecting undesired task behaviors, or in handling such events with dynamic provisioning of computational resources in a cloud scenario.

The paper is organized as follows. In Section 2, after introducing the applicative scenario based on the Map Reduce model, we present the overall architecture, describing the main components and their relationships. A special emphasis is given to the monitoring block, based on declarative constraints. Section 3 presents the use case scenario, based on the execution of a well-known benchmark over the popular Map Reduce platform Hadoop. This section also includes the experimental results demonstrating the potential of our approach. Related work and Conclusions follow.

2 SYSTEM CONTEXT AND SPECIFICATIONS

In this section, we propose a framework architecture to online detect user-defined critical situations in a Map Reduce environment and autonomously react by providing or removing resources according to high-level rules definable in declarative language.

2.1 Applicative Scenario

Map Reduce is a programming model able to simplify the complexity of parallelization. Following this approach, the input data-set is partitioned into an arbitrary number of parts, each exclusively processed by a different computing task, the *mapper*. Each *mapper* produces intermediate results (in the form of *key/value* pairs) that are collected and processed by other tasks, called *reducers*, in charge of calculating the final results by merging the *values* associated to the same *key*. The most important feature of MapReduce is that programs implemented according to this model are intrinsically parallelizable.

In this scenario, the estimation of the execution time can be crucial to check deadline or detect bottlenecks but the time to execute each *mapper* or *reducer* task can vary depending on different factors – e.g., the content of the block of input data analyzed, the performance of the machine on which the task is executed, the location of the input data (local to the task or on another machine), the bandwidth between the physical nodes of the distributed infrastructure. For this reason, the prediction of the execution time for Map Reduce applications is not a trivial task.

Since elasticity is so crucial in the data-intensive scenario, all the main platforms that implement the Map Reduce model offer application scale-up/-down as a feature, making relatively simple to add (or remove) computational nodes to the distributed infrastructure while performing a data-intensive analysis.

2.2 Framework Architecture

The main component of the proposed architecture is Map Reduce Auto-scaling Engine. This application-level software consists of three main subcomponents (grey blocks in Figure 1): the Monitoring, Recovery and Platform Interface. These elements interacts with the Map Reduce platform to detect and react to anomalous sequences of events in the execution flow.

The Monitoring component takes as input a high-level specification of the system properties describing the expected behavior of a Map Reduce workflow and the on-line sequence of events from the Map Reduce platform's log. Given these input data, the Monitoring component is able to rise alerts whenever the execution flow violates user-defined constraints. The alarms are evaluated by the Recovery component in order to estimate how many computational nodes must be provisioned (or de-provisioned) to face the critical condition according to user-defined rules taken as input.

Finally, the Platform Interface is in charge of translating the requests for new Map Reduce nodes into virtual machine (VM) provisioning requests to the infrastructure manager. The Platform Interface is also responsible for the installation of Map Reduce-

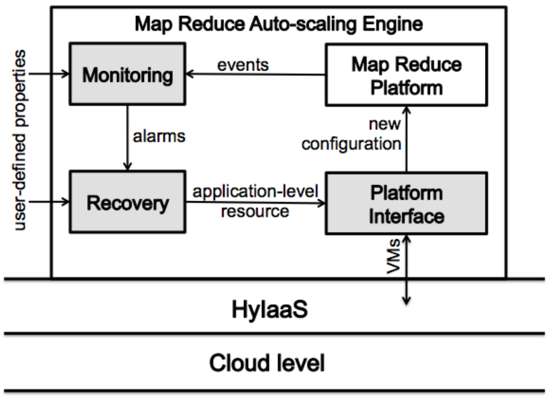


Figure 1: Framework architecture.

specific software on the newly provided virtual machines (VMs). The output of this subcomponent is a new configuration of the computing cluster with a different number of working nodes.

As shown in Figure 1, Map Reduce Auto-scaling Engine relays on a lower level component called Hybrid Infrastructure as a Service (HyIaaS) for the provisioning of VMs (Loreti and Ciampolini, 2015). This layer encapsulates the cloud functionality and interacts with different infrastructures to realize a hybrid cloud: if the resources of a private (company-owned) on-premise cloud are no longer enough, HyIaaS redirects the scale-up request to an off-premise public cloud. Therefore, thanks to HyIaaS, the resulting cluster of VMs for Map Reduce computation can be composed by VMs physically deployed on different clouds. Further details about HyIaaS can be found in (Loreti and Ciampolini, 2015).

The hybrid nature of the resulting cluster is often very useful (especially if the on-premise cloud has limited capacity) but can also further exacerbate the problem of Map Reduce performance prediction. If part of the computing nodes is available through a higher latency, the execution time can be substantially afflicted by the allocation of the tasks and the amount of information they trade with each other. Despite the complexity of the scenario, we want the monitoring system to offer a simple interface for the elicitation of the properties to be respected. Nonetheless, it should be able to rapidly identify critical situations. To this end, we apply the MOBUCON framework to the monitoring component and benefit from the application of well-known Process Mining techniques to our environment.

2.3 Monitoring the System Execution w.r.t. Declarative Constraints

Monitoring complex processes such as Map Reduce approaches in dynamic and hybrid clouds has two fundamental requirements: on one hand, there is the need of a language expressive enough to capture the complexity of the process and to represent the key properties that should be monitored. Of course, for practical applications, such language should come already equipped with sound algorithms and reasoning tools. On the other hand, any monitor must produce results in a timely fashion, being the analysis carried out on the fly, typically during the system execution.

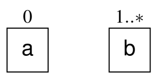
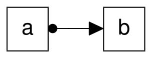
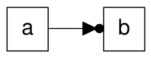
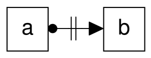
Declarative languages are one of the solutions proposed in the field of Business Process Management to answer the above requirements. In particular, they have been adopted to model business rules and loosely-structured processes, mediating between support and flexibility.

Among the many proposals, we focused on the Declare language (Pesic and van der Aalst, 2006), a graphical, declarative language for the specification of activities and constraints. The Declare language has been extended with temporal deadlines and data-aware constructs in (Montali et al., 2013b; Montali et al., 2013a), where also the MOBUCON tool has been presented, together with some figures about its performances in a run-time context.

Declare is a graphical language focused on *activities* (representing atomic units of work), and *constraints*, which model expectations about the (non) execution of activities. Constraints range from classical sequence patterns to loose relations, prohibitions and cardinality constraints. They are grouped into four families: *(i) existence* constraints, used to constrain the number of times an activity must/can be executed; *(ii) choice* constraints, requiring the execution of some activities selecting them among a set of available alternatives; *(iii) relation* constraints, expecting the execution of some activity when some other activity has been executed; *(iv) negation* constraints, forbidding the execution of some activity when some other activity has been executed. Tab. 1 shows few simple Declare constraints.

The Declare language provides a number of advantages: being inherently declarative and open, it supports the modeler in the elicitation of the (minimal) set of behavioral *constraints* that must be respected by the process execution. Acceptable execution courses are not explicitly enumerated, but rather, they are implicitly defined by the execution traces that comply with all the constraints. In this sense, Declare is indeed a notable example of *flexibility by design*.

Table 1: Some Declare constraints.

	**Absence** The target activity a cannot be executed **Existence** Activity b must be executed at least once
	**Response** Every time the source activity a is executed, the target activity b must be executed after a
	**Precedence** Every time the source activity b is executed, a must have been executed before
	**Negation response** Every time the source activity a is executed, b cannot be executed afterwards

Moreover, Declare (and its extensions) supports temporal deadlines and data-aware constraints, thus making it a powerful modeling tool. The MOBUCON tool fully supports the Declare language; moreover, being based on a Java implementation of the EC formalism (Kowalski and Sergot, 1986), it provides a further level of adaptability: the system modeler can directly exploit the EC – as in (Bragaglia et al., 2012) – or the Java layer underneath for a fully customizable monitoring. Finally, MOBUCON and the extended Declare support both atomic and non-atomic activities.

3 USE CASE SCENARIO

The architecture shown in Figure 1 has been implemented and analyzed using a testbed framework. In particular, a simulation approach has been adopted to create specific situations, and to verify the run-time behavior of the whole architecture. To this end, synthetic data has been generated, with the aim of stressing the Map Reduce implementation.

3.1 Testbed Architecture and Data

The Map Reduce model is implemented and supported by several platforms. In this work we opted for Apache Hadoop (Apache Hadoop, 2015), one of the most used and popular frameworks for distributed computing. Hadoop is an open source implementation consisting of two components: Hadoop Distributed File System (HDFS) and Map Reduce Runtime. The input files for Map Reduce jobs are split into fixed size blocks (default is 64 MB) and stored in HDFS. Map Reduce runtime follows a master-worker architecture. The master (Job-Tracker) assigns tasks to the worker nodes. Each worker node runs a Task-Tracker daemon that manages the currently assigned task. Each worker node can have up to a predefined number of *mappers* and *reducers* simultaneously running. This concurrent execution is controlled through the concept of *slot*: a virtual container that can host a running task. The user can specify the number s_w of *slot*s for each worker w. This number should reflect the maximum number of processes that the worker can concurrently run (e.g., on a dual core with hyperthreading s_w is suggested to be 4). The Job-Tracker will assign to each worker a number n_w of tasks to be concurrently executed, such that the relation $n_w \leq s_w$ is always guaranteed.

We define S as the total number of *slot*s in the MapReduce platform:

$$S = \sum_w s_w \tag{1}$$

The value in Eq. 1 also addresses the total number of tasks that the platform can concurrently execute.

For the sake of simplicity, we start focusing only on map phase deadlines because all the map tasks usually operates on similar volumes of data and we can assume that in a normal execution they will require similar amount of time – as also suggested by (Mattess et al., 2013). The deadline t_M for each *mapper* can be evaluated as:

$$t_M = \frac{D_M \cdot S}{M} \tag{2}$$

Where D_M is the deadline for the execution of the map phase and M is the total number of mapper to be launched. Conversely, the amount of data processed by the reduce phase is unknown until all the *mappers* have completed, thus complicating the estimation of a deadline for each *reducer*.

Our Hadoop testbed is composed of 4 VMs: 1 master and 3 worker nodes. Each VM has 2 VCPUs, 4GB RAM and 20GB disk. At the cloud level we use 5 physical machines, each one with a Intel Core Duo CPU (3.06 GHz), 4GB RAM and 225GB HDD. Since a dual core machine (without hyperthreading) can concurrently execute at most two tasks, we assigned two *slot*s to each worker. Our Map Reduce platform can therefore execute up to six concurrent tasks ($S = 6$).

As for the task type, we opted for a word count job, often used as a benchmark for assessing performances of a Map Reduce implementation. In our scenario we prepared a collection of 20 input files of 5MB each. Consequently, Hadoop Map Reduce Runtime launches $M = 20$ *mappers* to analyze the input data. In this testbed, we would like to complete the map phase in $D_M = 200$ seconds, so every map task should not exceed one minute execution.

According to the default Hadoop configuration, the output of all these *mappers* is analyzed by a single *reducer*. In order to emulate the critical condition of some tasks showing an anomalous behavior, we artificially modified 8 input files, so has to simulate a dramatic increase of the time required to complete the task. The *mappers* analyzing these blocks resulted to be 6 times slower than the normal ones.

Note that, as other MapReduce platforms, Hadoop has a fault tolerance mechanism to detect the slow tasks and relaunch them from the beginning on other – possibly more performing – workers. This solution is useful in case the problem is caused by architectural factors (poor performance or bandwidth saturation on the original worker), but is likely to be counter-productive when the execution slow down is related to the content of the data blocks involved. In that case indeed, the problem will occur again on the new worker. The only way to speed up the computation is by assigning to the newly provided workers other pending tasks in the queue, thus to increase the value of S for the MapReduce platform.

3.2 Properties to be Monitored

In this work we mainly focus on time-constrained data insight: the aim is to identify as soon as possible the critical situation of the Map Reduce execution going to complete after a predefined deadline. Practically speaking, this correspond to situations where the total execution time of the Map Reduce is expected to stay within some (business-related) deadline: e.g., banks and financial bodies require to perform analyses of financial transactions during night hours, and to provide outcomes at the next work shift.

The MOBUCON framework already provides a model of activities execution, where a number of properties to be monitored are already directly supported. In particular, a support for non-atomic activities execution is proposed within the MOBUCON framework, where for each start of execution of a specific ID, a subsequent end of execution (with same ID) is expected. This feature has been particularly useful during the verification of our testbed, to identify a number of exceptions and worker faults due to problems and issues not directly related to the Map Reduce approach. For example, during our experiments we ignored fault events generated by power shortages of some of the PC composing the cloud. The out-of-the-box support offered by MOBUCON was exploited to identify these situations and rule them out.

To detect problematic mappers, we decided to monitor a very simple Declare property between the start and the end of the elaboration of each mapper.

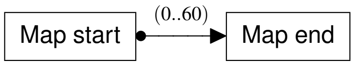


Figure 2: Declare Response constraint, with a metric temporal deadline.

Declare augmented with metric temporal deadlines as in (Montali et al., 2013b) was exploited to this end, and the constraint shown in Figure 2 illustrates the *Response* constraint we specified in MOBUCON. It simply states that after an event *Map start*, a corresponding event *Map end* should be observed, within zero and 60 seconds [1] . Notice that MOBUCON correlates different events on the basis of the *case*: i.e., it requires that every observed event belongs to a specific case, identified by a single case ID. To fulfill such requirement, we fed the MOBUCON monitor with the events logged by the Hadoop stack, and exploited the Map identifier (assigned by Hadoop to each mapper) as a *case ID*. This automatically ensures that each *Map start* event is indeed matched with the corresponding *Map end* event.

The constraint shown in Figure 2 allows us to detect mappers that are taking too much time to compute their task. The deadline set to 60 seconds has been chosen on the basis of the total completion time we want to respect while analyzing the simulation data. Naturally, some knowledge about the application domain is required to properly set such deadline. Mappers that violate the deadline are those that, unfortunately, were assigned a long task. This indeed would not be a problem for a single mapper. However, it could become a problem if a considerable number of mappers gets stuck on long tasks, as this might undermine the completion of the whole bunch of data within a certain deadline. Note that, if the user doesn't have any knowledge of the volume of data to be processed – and consequently, the number of map tasks to be launched is not known *a priori* –, this methodology allow him to still detect anomalies in the data that can require additional resources to speed up the computation. For example, the deadline for each map task can be computed at execution time by taking into account the average completion time for each completed mapper. The same approach can be used for the runtime estimation of the reduce phase deadline compliance.

Besides supporting the monitoring of Declare constraints, MOBUCON supports also the definition of user-specific properties. We exploited this feature and expressed a further property by means of the EC language. The property, that we named *long_execution_maps*, aims to capture all the mappers that have already violated the deadline, and that are still *active* (i.e., a start event has been seen for that mapper, and no end event has yet been observed). Such definition is given in terms of an EC axiom:

$$\begin{array}{l} initiates(\\ \qquad deadline\_expired(A, ID), \\ \qquad status(i(ID, long\_execution\_maps), too\_long), \\ \qquad T \\ ) \quad \leftarrow \\ \qquad holds\_at(status(i(ID, waiting\_task), pend), T), \\ \qquad holds\_at(status(i(ID, A), active), T). \end{array}$$

We do not provide here all the details about the axiom – the interested reader can refer to (Kowalski and Sergot, 1986) for an introduction to EC. Intuitively, the axiom specifies that at any time instant T, the happening of the event $deadline\_expired(A, ID)$ initiates the property *long_execution_maps* with value *too_long* for the mapper *ID*, if that mapper was still active and there was a constraint *waiting_task* still not fulfilled. The *waiting_task* constraint is indeed the response constraint we discussed in Figure 2.

With the *long_execution_maps* property we can determine within the MOBUCON monitor which are the mappers that got stuck on some task. However, to establish if a problem occur to the overall system, we should aggregate this information, and consider for each time instant *how many* mappers are stuck w.r.t. the total number of available mappers. Exploiting the MOBUCON feature of supporting also a *healthiness function*, we provided the following function:

$$\text{System health} = 1 - \frac{\#long\_execution\_maps}{\#total\_maps\_available} \qquad (3)$$

In other words, the system health is expressed as the fraction of mappers that are not busy with a long task, over the total number of launched mappers. The lower the value, the higher the risk that the overall Hadoop framework gets stuck and violates some business deadline. In order to make the health function more responsive, we can define a window of map task to be considered in the computation of system health.

3.3 The Output from MOBUCON Monitor

In Figure 3, we show what happens when we analyze a word count execution on the Hadoop architecture described in Section 3.1, with respect to the properties discussed in Section 3.2. Note that, as we focus

[1] MOBUCON accepts deadlines at different time units. In this paper we opted for expressing the time unit in terms of seconds, although depending on the application domain minutes or milliseconds might be better choices.

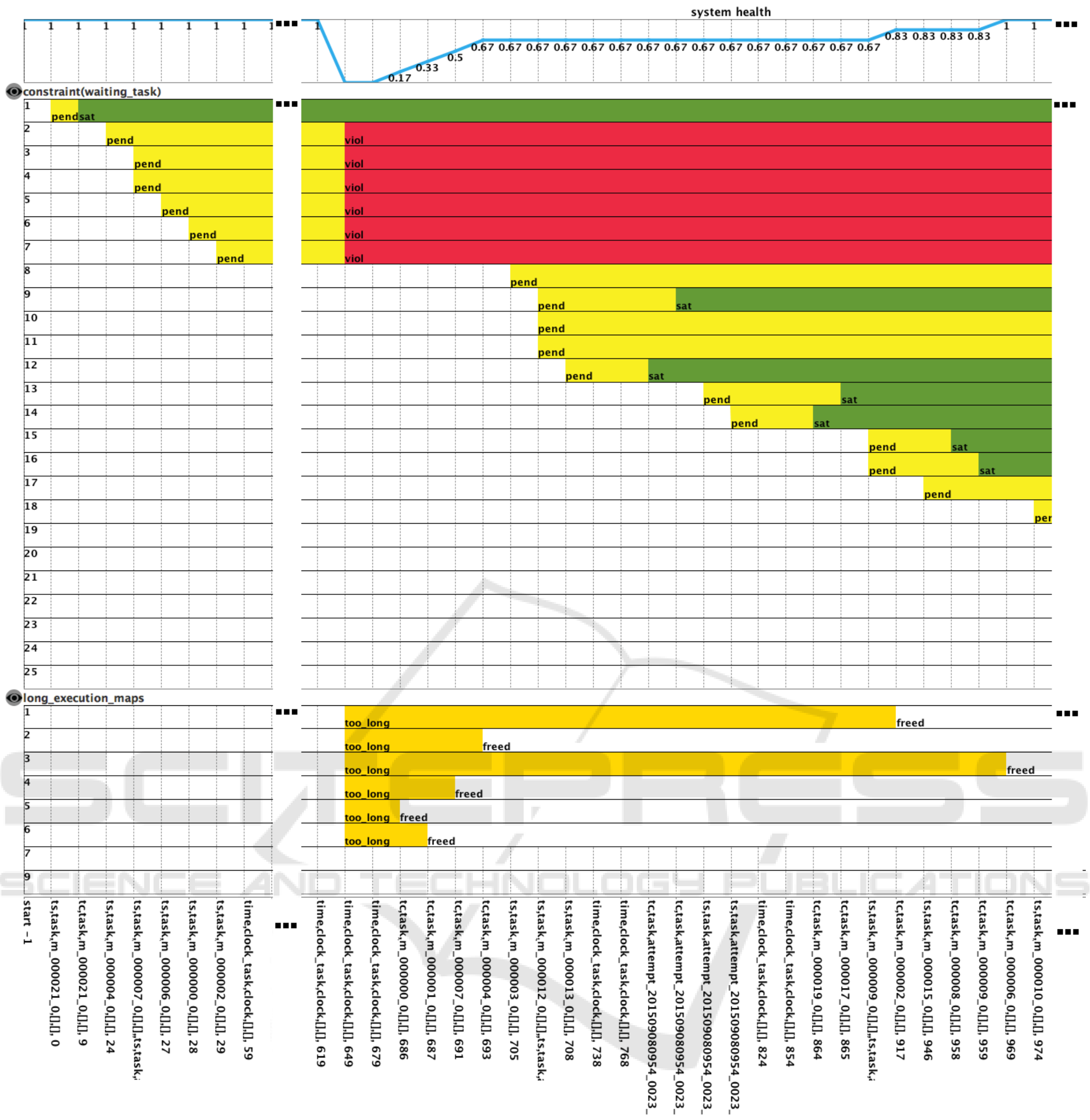


Figure 3: The output of the MOBUCON monitor for the execution of word count job on the given testbed.

on the performance of the system when data-specific factors slow down the computation, the declarative semantic employed and the results of the following evaluation are independent from the specific MapReduce-encoded problem (e.g., word count, terasort, inverted index etc.).

Figure 3 is composed of four strips, representing the evolution of different properties during the execution. From top to bottom of the figure we have: the health function, graphical representation of the Declare constraint, *long_execution_maps* property and description of the events occurred in each time interval. In the latter in particular (bottom part of Figure 3), the observed events has starting labels `ts` or `tc` to represent the start and the completion of a task, respectively. There are also a number of events starting with the label `time`: these events represent the ticking of a reference clock, used by MOBUCON to establish when deadlines are expired.

The health function on top of Figure 3 is the one defined in Eq. 3: indeed, the system healthiness dramatically decreased when six over seven of the first mappers launched in our testbed got stuck in a long execution task. The *long_execution_maps* strip (third strip from the top in Figure 3) further clarifies the intervals during which the long map tasks exceed their

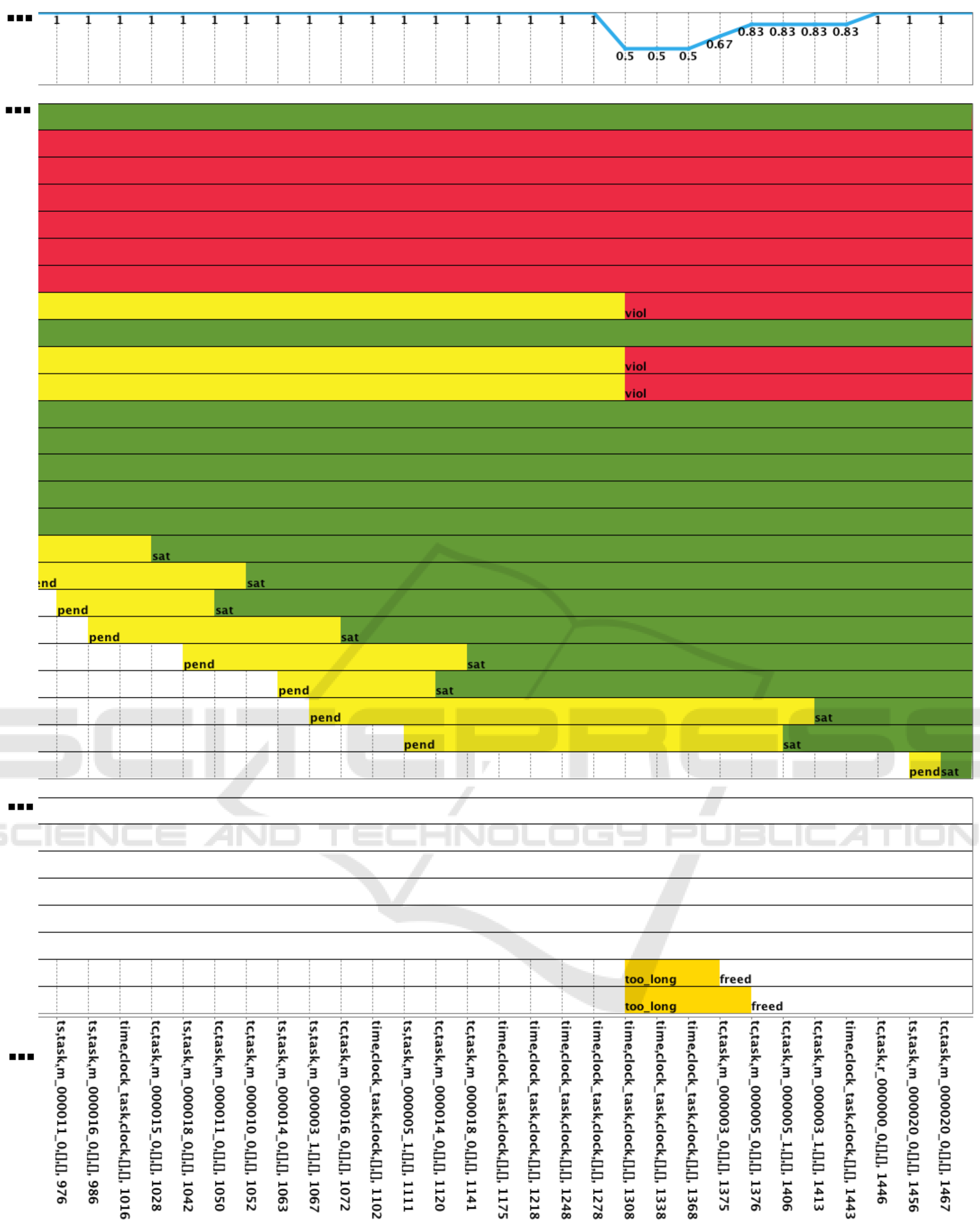


Figure 4: Output of the MOBUCON monitor subsequent to Figure 3.

time deadline.

Finally, the Declare response constraint strip (second strip from the top in Figure 3) shows the status of each mapper: when the mapper is executing, the status is named *pending* and it is indicated with a yellow bar. As soon as there is information about the violation of a deadline (because of a tick event from the reference clock), the horizontal bar representing the status switched from *pending* to *violated*, and the color is changed from bright yellow to red. Notice that once violated (red color), the response constraint remains as such: indeed, this is a consequence of the Declare semantics where no compensation mechanisms are considered.

For reasons of space, we provide in Figure 4 the evolution of our test (subsequent to what shown in Figure 3). As expected, the total number of mapper violating the deadline constraint is 8, as we provided 8 modified files in the input dataset. MOBUCON is therefore able to suddenly and efficiently identify any anomaly in the Hadoop execution (according to simple user-defined constraints).

The health function values in the output of MOBUCON monitor can be used to determine when a recovery action is needed. The intervention can be dynamically triggered by a simple threshold mechanism over the health function or by a more complex user-defined policy (e.g., implementing an hysteresis cycle), possibly specified with a declarative approach.

Once the number of additional Hadoop workers needed is determined, Map Reduce Auto-scaling Engine relays on HyIaaS for the provisioning of VMs over a single public cloud or federated hybrid environment.

During the evaluation depicted in Figure 3 and 4, 85 events are processed by the MOBUCON monitor in 285 milliseconds (worst case over 10 evaluations). Thanks to the high expressiveness of the adopted declarative language, the user can define complex constraints, thus increasing the computational cost of the runtime monitoring. We are aware that, under this condition, the system can suffer a penalty in the execution time and the described method can show limits when dealing with fast monitored tasks (i.e., the time between task start and task end events is too short for the Monitoring component to evaluate the compliance). Nevertheless, in the envisioned MapReduce scenario, the average duration time is in general higher than the time required by MOBUCON to check the constraints. Furthermore, since the recovery action to provide additional workers is intrinsically time consuming (tens of minutes), the Monitoring component is not requested to be responsive in the order of sub-seconds. Therefore, we can state that the time to detect anomalies shown in Figure 3 and 4 is acceptable for the envisioned scenario.

4 RELATED WORK

Cloud computing is currently used for a wide and heterogeneous range of tasks. It is particularly useful as elastic provider of virtual resources, able to contribute to heavy computing tasks.

Data-intensive applications are an example of resource demanding tasks. A widely adopted programming model for this scenario is MapReduce (Dean and Ghemawat, 2008), whose execution can be supported by platforms such as Hadoop (Apache Hadoop, 2015), possibly in a cloud computing infrastructure. We tested our system with MapReduce applications, choosing Hadoop as execution engine.

Recently, a lot of work has focused on cloud computing for the execution of big data applications: as pointed out in (Collins, 2014), the relationship between big data and the cloud is very tight, because collecting and analyzing huge and variable volumes of data require infrastructures able to dynamically adapt their size and their computing power to the application needs. The work (Chen et al., 2014a) presents an accurate model for optimal resource provisioning useful to operate MapReduce applications in public clouds. Similarly, (Palanisamy et al., 2015) deals with optimizing the allocation of VMs executing MapReduce jobs in order to minimize the infrastructure cost in a cloud datacenter. In the same single-cloud scenario, the work (Rizvandi et al., 2013) focuses on the automatic estimation of MapReduce configuration parameters, while (Verma et al., 2011) proposes a resource allocation algorithm able to estimate the amount of resources required to meet MapReduce-specific performance goals. However, these models were not intended to address the challenges of the hybrid cloud scenario, which is a possible target environment for the provisioning of additional VMs in our system thanks to the underlying HyIaaS layer.

More similarly to our approach, cloud bursting techniques has been adopted for scaling MapReduce applications in the work(Mattess et al., 2013), which presents an online provisioning policy to meet a deadline for the Map phase. Differently from our approach, (Mattess et al., 2013) focuses on the prediction of the execution time for the Map phase with a traditional approach to monitoring, which introduces complexity in the implementation and tuning, whereas our solution can benefit from a simple enunciation of the system properties relaying on Declare language.

Also the work presented in (Kailasam et al., 2014) deals with cloud monitoring/management for big data applications. It proposes an extension of the MapReduce model to avoid the shortcomings of high latencies in inter-cloud data transfer: the computation inside the on-premise cloud follows the batch MapReduce model, while in the public cloud a stream processing platform called Storm is used. The resulting system shows significant benefits. Differently from (Kailasam et al., 2014), we chose to keep complete transparency and uniformity with respect to the allocation of the working nodes and their configuration.

As regards the use of EC for verification and monitoring, several examples can be found in letterature in different application domains but we are not aware of any work applying it to the monitoring of MapReduce jobs in a cloud environment. EC has been used in various fields to verify the compliance of a system to user-defined behavioral properties. For example, (Spanoudakis and Mahbub, 2006), (Farrel et al., 2005) exploit ad-hoc event processing algorithms to manipulate events and fluents, written in JAVA. Differently from MOBUCON they do not have an underlying formal basis, and they cannot take advantage of the expressiveness and computational power of logic programming.

Several authors – (Giannakopoulou and Havelund, 2001), (Bauer et al., 2011) – have investigated the use of temporal logics – Linear Temporal Logic (LTL) in particular – as a declarative language for specifying properties to be verified at runtime. Nevertheless, these approaches lack the support of quantitative time constraints, non-atomic activities with identifier-based correlation, and data-aware conditions. These characteristics – supported by MOBUCON – are instead very important in our application domain.

5 CONCLUSIONS

This work present a framework architecture that encapsulates an application level platform for data-processing. The system lends the Map Reduce infrastructure the ability to autonomously check the execution, detecting bottlenecks and constraint violations through Business Process Management techniques with a *best effort* approach.

Focusing on *activities* and *constraints*, the use of Declare language has shown significant advantages in the monitoring system implementation and customization.

Although this work represents just a first step towards an auto-scaling engine for Map Reduce, its declarative approach to the monitoring issue shows promising results, both regarding the reactivity to critical conditions and the simplification in monitoring constraint definition.

For the future, we plan to employ the defined framework architecture to test various diagnosis and recovery policies and verify the efficacy of the overall auto-scaling engine in a wider scenario (i.e., with a higher number of Map Reduce workers involved).

Finally, particular attention will be given to the hybrid cloud scenario, where the HyIaaS component is employed to transparently perform VM provisioning either on an on-premise internal or an off-premise public cloud. In case of a hybrid deploy, several additional constraints will need to be taken into account (e.g., the limited inter-cloud bandwidth), thus further complicating the implemented monitoring and recovery policies. Nevertheless, we believe that a declarative approach to the problem can contribute to significantly simplify the implementation of the solution.

REFERENCES

Amazon Cloud Watch (2015). Amazon cloud monitor system. https://aws.amazon.com/it/cloudwatch/. Web Page, last visited in Dec. 2015.

Apache Hadoop (2015). Apache software foundation. https://hadoop.apache.org/. Web Page, last visited in Dec. 2015.

Apache Spark (2015). Apache software foundation. http://spark.apache.org. Web Page, last visited in Dec. 2015.

Armbrust, M., Fox, O., and R., G. (2009). Above the clouds: A berkeley view of cloud computing. Technical report, Electrical Engineering and Computer Sciences University of California at Berkeley.

Bauer, A., Leucker, M., and Schallhart, C. (2011). Runtime verification for ltl and tltl. *ACM Trans. Softw. Eng. Methodol.*, 20(4):14:1–14:64.

Bragaglia, S., Chesani, F., Mello, P., Montali, M., and Torroni, P. (2012). Reactive event calculus for monitoring global computing applications. In *Logic Programs, Norms and Action*. Springer.

Ceilometer, O. (2015). the openstack monitoring module. https://wiki.openstack.org/wiki/ceilometer.

Chen, K., Powers, J., Guo, S., and Tian, F. (2014a). Cresp: Towards optimal resource provisioning for mapreduce computing in public clouds. *Parallel and Distributed Systems, IEEE Transactions on*, 25(6):1403–1412.

Chen, M., Mao, S., and Liu, Y. (2014b). Big data: A survey. *Mobile Networks and Applications*, Volume 19(2):171–209.

Collins, E. (2014). Intersection of the cloud and big data. *Cloud Computing, IEEE*, 1(1):84–85.

Dean, J. and Ghemawat, S. (2008). Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113.

Farrel, A., Sergot, M., Sallè, M., and Bartolini, C. (2005). Using the event calculus for tracking the normative state of contracts. *International Journal of Cooperative Information Systems*, 14(02n03):99–129.

Giannakopoulou, D. and Havelund, K. (2001). Automata-based verification of temporal properties on running programs. In *Automated Software Engineering, 2001. (ASE 2001). Proceedings. 16th Annual International Conference on*, pages 412–416.

Kailasam, S., Dhawalia, P., Balaji, S., Iyer, G., and Dharanipragada, J. (2014). Extending mapreduce across clouds with bstream. *Cloud Computing, IEEE Transactions on*, 2(3):362–376.

Kowalski, R. A. and Sergot, M. J. (1986). A Logic-Based Calculus of Events. *New Generation Computing*.

Loreti, D. and Ciampolini, A. (2015). A hybrid cloud infrastructure fo big data applications. In *Proceedings of IEEE International Conferences on High Performance Computing and Communications*.

Mattess, M., Calheiros, R., and Buyya, R. (2013). Scaling mapreduce applications across hybrid clouds to meet soft deadlines. In *Advanced Information Networking and Applications (AINA), 2013 IEEE 27th International Conference on*, pages 629–636.

Montali, M., Chesani, F., Mello, P., and Maggi, F. M. (2013a). Towards data-aware constraints in declare. In Shin, S. Y. and Maldonado, J. C., editors, *Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC '13, Coimbra, Portugal, March 18-22, 2013*, pages 1391–1396. ACM.

Montali, M., Maggi, F. M., Chesani, F., Mello, P., and van der Aalst, W. M. P. (2013b). Monitoring business constraints with the event calculus. *ACM TIST*, 5(1):17.

Palanisamy, B., Singh, A., and Liu, L. (2015). Cost-effective resource provisioning for mapreduce in a cloud. *Parallel and Distributed Systems, IEEE Transactions on*, 26(5):1265–1279.

Pesic, M. and van der Aalst, W. M. P. (2006). A Declarative Approach for Flexible Business Processes Management.

Rizvandi, N. B., Taheri, J., Moraveji, R., and Zomaya, A. Y. (2013). A study on using uncertain time series matching algorithms for mapreduce applications. *Concurrency and Computation: Practice and Experience*, 25(12):1699–1718.

Spanoudakis, G. and Mahbub, K. (2006). Non-intrusive monitoring of service-based systems. *International Journal of Cooperative Information Systems*, 15(03):325–358.

Van Der Aalst, W., Adriansyah, A., de Medeiros, A. K. A., and Arcieri, F. (2012). Process mining manifesto. In *Business Process Management Workshops*. Springer Berlin Heidelberg.

Verma, A., Cherkasova, L., and Campbell, R. H. (2011). *Resource Provisioning Framework for MapReduce Jobs with Performance Goals*, volume 7049 of *Lecture Notes in Computer Science*, pages 165–186. Springer Berlin Heidelberg.