DRACO PaaS: a Distributed Resilient Adaptable Cloud Oriented Platform

Antonio Celesti, Nicola Peditto, Fabio Verboso, Massimo Villari and Antonio Puliafito
DICIEAMA, Faculty of Engineering, University of Messina
Contrada di Dio, S. Agata, 98166 Messina, Italy.
e-mail: {acelesti, mvillari, npeditto, fverboso, apuliafito}@unime.it

Abstract—This paper focuses on Platform as a Service (PaaS). From the advent of Cloud computing, the latest trend has been to design ad hoc platforms specialized to address given operational scenarios. In this context, the need of an adaptable, fault tolerant and secure cloud platform for the development of specific PaaS is becoming more and more compelling. In this work, we present the Distributed Resilient Adaptable Cloud Oriented (DRACO) PaaS. DRACO PaaS is more than a cloud platform. In fact, it introduces a new development model for PaaS. More specifically, we will focus on how DRACO PaaS can be adopted for the development of any kind of specialized PaaS.

Keywords—Cloud Computing; Distributed System; PaaS; Service Composition.

I. INTRODUCTION

Nowadays, Cloud computing is challenging the traditional management methods, proposing a new way to conceive the provisioning, configuration, portability, and interoperability of services. Different Cloud architectures have been proposed until now considering the three main service levels, i.e. Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS). Currently, it is not simple to give a precise definition of requirements that are common to different PaaS(s).

The lack of guidelines and computing models makes the designing of PaaS rather difficult. In fact, a tenant (e.g., society, organization, government, etc) is free to conceive its own PaaS. At the same time, software architects and developers have to solve several issues, including fault-tolerance, configuration, accounting, scalability, performance, security, and the interoperability with other PaaS and with the underlying IaaS service layer. In this context, the need of an adaptable cloud platform is becoming more and more compelling.

In this paper, we propose a Distributed Resilient Adaptable Cloud Oriented (DRACO) PaaS that aims to provide both a generic adaptable cloud platform and a new computing model for the development of specific tenant PaaS. DRACO PaaS represents a middleware that on one hand provides a bind with the underlying IaaS layer, and on the other hand an environment for the development, deployment, and execution of a specific tenant PaaS.

The design of DRACO PaaS as been inspired by FCAPS (Fault-tolerance, Configuration, Accounting, Performance and Security), that is, the ISO Telecommunications Management Network model. According to the FCAPS model, each DRACO Element (DRE) consists of five different Signaling Components (SCs), that are respectively “F”, “C”, “A”, “P”, “S”, and one single Computing Component (CC). Moreover, DREs are interconnected each other using two overlay networks: the Signaling Overlay Network (SON) and the Computing Overlay Network (CON). These two overlay networks allow parallelism between control and computing tasks. Each CC is connected to the SON using an I/O full-duplex channel. In this way, DRACO PaaS can have an autonomic control of each DRE and, at the same time, of the whole environment. Instead, each CC is connected on the CON by means of another I/O full-duplex channel. The CON allows to compose services interconnecting different DREs. In this way it is possible to build any kind of PaaS as in a Service Oriented Architecture (SOA) organizing the interconnected DREs, in a workflow suitably linking the CC’s I/O channels. Thus, it is possible to deal with different CONs, hence different workflows, each one linking different DREs as well as in a Directed Acyclic Graph (DAG).

The paper is organized as follows. Related works are discussed in Section II. In Section III, we motivate the paper highlighting the need of both an adaptable cloud platform and of a new design pattern for the development of specific PaaS. Section IV, introduces the DRAGO computing model. Section V presents the DRACO architecture. Section VI describes how can be possible to compose different DREs in order to build a specific PaaS through a workflow. In Section VII, it is discussed an implementation of the DRACO architecture combining Python and XMPP. Experiments are discussed in Section VIII, considering a video transcoding case of study. Section IX concludes the paper.

II. RELATED WORKS

The Cloud Computing paradigm is moving forward from the definition of its operation and capabilities towards the implementation of middleware able to solve real problems. In the area of PaaS, many works are appearing in the literature for managing sensors, government services, business processes, medical images, and smart things (e.g. homes, cities, cars, tv, phones etc.). All of these use cases have strong requirements in terms of computation needs, storage of data (i.e., Big Data) and interactions among heterogeneous devices. In [1], the authors propose a PaaS architecture for the rapid development of wireless sensor network (WSN). In particular
they report an interesting *Dynamic Stack* matching the sensors capabilities with a real cloud stack in order to provide the SaaS, PaaS, and IaaS layers inside a sensor node. A first example of a PaaS architecture for Mobile Applications is described in [2]. The authors looked at a framework able to manage high scalability in data collection and processing. The work is interesting for its description, but it lacks of a real implementation. Another design of a PaaS is presented in [3]. The work shows how Smart Homes of the future should interact with Clouds. The authors state that *Peer-To-Peer* networks may help to setup a scalable infrastructure. Another type of theoretical PaaS is shown in [4]. It tries to address the issues related to Government Services, where several stakeholders are considered. The use case is rather complex and includes several components: federated identities, user interfaces (i.e., Web, Mobile and Managed Applications), computing elements (where different roles are identified), Database and Networking equipments. The overall architecture shows how several different Application Patterns can be easily applied on top of this framework, combining different block diagrams. In [5] a Platform for treating Medical Images has been proposed. It is an hybrid architecture where Grid resources are also used. In particular, the authors introduced a new PaaS framework, called *Acigna-G*, strongly based on Grid computing mechanisms. The use case may benefit from using PaaS, but the way the authors developed the framework suffers from several constraints related to the use of Grid. An interesting work is reported in [6] where the authors introduce and implement a PaaS platform adaptation. In particular they adapted MPI to MapReduce PaaS Clouds. A plethora of commercial PaaS cloud are operating in the market, and for this reason the authors decided to create an adapter that reconcile some inconsistencies and incompatibilities of the MPI applications.

All the works presented above consider PaaS as platforms that utilize Hadoop, Grid and other consolidate commercial frameworks such as Microsoft Azure and Google App Engine.

### III. Motivations

The motivations that have led to DRACO PaaS are basically three:

1. Provide a platform for the development of complex algorithms in the Cloud.
2. Design a middleware for the development of PaaS(s) able to bind the IaaS and the PaaS service layers.
3. Conceive a new computing model for the development of PaaS in the Cloud.

In the following we better explain such motivations.

#### A. Development of Complex Algorithms in the Cloud

The computation power of Cloud computing can be utilized for the development of complex distributed algorithms. Often such algorithms are orchestrated as workflows in which the computing process is divided among different cooperating functional units in order to achieve a common goal. Such algorithms usually involve parallel processing and require a considerable amount of computational and storage resources. Examples includes:

- ordering algorithms
- mathematical algorithms
- image processing
- search algorithms
- optimization algorithms
- ...

In literature there are many algorithms that require a considerable amount of resources to be processed. An example is represented by the Video Transcoding [7]. It is a direct digital-to-digital data conversion algorithm from one encoding to another, such as for video and audio files. This is usually done in cases where a target device does not support the format or has limited storage capacity that mandates a reduced file size, or to convert incompatible or obsolete data to a better-supported or modern format. An example of workflow of such an algorithm is depicted in Figure 1.

#### B. A Middleware for the Development of PaaS

From an architectural point of view, cloud systems can be designed according to a three-layer stack: from the bottom Virtual Machine Manager (VMM), Virtual Infrastructure Manager (VIM), and Cloud Manager (CM). The VMM layer can be generally associated to the Hypervisor (e.g., Xen, KVM, VMware, Virtual Box, Virtual PC, etc) running on top of the Operating System (OS). The VIM provides an abstraction layer for the VMM. It interacts with the Hypervisor running on top of the OS of each physical server included in the cloud’s datacenter, and enables the capability of deploying Virtual Machines (VMs) on the physical hardware where the VMM is running. Most of the cloud middleware mainly exploit this layer adding some new features. The main purpose of the VIM layer is essentially the ability to setup VMs (preparing disk images, setting up networking, and so on) regardless of the underlying VMM. The goal of the CM layer is to build on top the VIM a platform or an application with cloud-like interfaces. Considering the three cloud service levels, this implies to build PaaS and SaaS deploying them in an IaaS.
However, the development of the CM layer is not trivial at all because software architects have to plan head provisioning, monitoring, performance, Identity Management (IdM) and scalability. The main issue regarding the design of this level is that currently available PaaS and SaaS are different from each others, and a flexible, adaptable cloud oriented PaaS middleware has not been developed yet.

IV. THE DRACO PaaS COMPUTING MODEL

The DRACO PaaS computing model extends the Von Neumann stored program control (SPC) computing model to create self-configuring, self-monitoring, self-healing, self-protecting and self-optimizing (self-managing or self-*) distributed software systems. As opposed to self-organizing systems that evolve based on probabilistic considerations, this approach focuses on the encapsulation, replication, and execution of distributed and managed tasks that are precisely specified.

DRACO PaaS, consists of a Signaling Overlay Network (SON) over Computing Overlay Network (CON) that allows parallelism between the control (e.g., setup, monitoring, analysis and reconfiguration based on PaaS variations, business priorities, security, and QoS constraints) and the computing functions of the distributed software components. As depicted in Figure 2, a DRACO Element (DRE) is a computing entity which allows fault, configuration, accounting, performance and security through five different Signaling Components (SCa), i.e., Fault-tolerance “F”, Configuration “C”, Accounting “A”, Performance “P”, and Security “S” according to the FCAPS model. Each different SC is connected over the SON through full-duplex channels. Inside each DRE, the Computing Component (CC) receives the service or application to execute through the “C” SC over the SON. The fault-tolerance, accounting, performance, and security of each DRE are continuously monitored in real time through messages respectively exchanged by the “F”, “A”, “P”, and “S” SCs over the SON. The specialization of a PaaS is accomplished by setting up the FCAPS requirements and by programming individual CCs to perform a specific task and connecting them in a workflow over the CON as in a Directed Acyclic Graph (DAG).

According to the DRACO PaaS, the users of the system are “tenants” who develop their own specific PaaS setting up workflows. A tenant can be a company, a government, an organization, or simple human end-user. For example, a tenant can be a company specialized in the delivering of video streaming services. As it wants to provide a new video transcoding SaaS via web to its clients, it builds a video transcoding PaaS by means of the DRACO PaaS. Each DRE CC receives/delivers input/output from/to other DRE CCs. The task run by each CC depends on the tenant requirements and it is programmed and executed as a loadable module in each DRE. The distributed software components along with associated profiles defining their use and management constraints are executed by DRACO PaaS with self-management by means of a signaling-enabled control using the SON. A Workflow can be used as blueprint to setup, execute and control the down-stream DAG at each DRE based on local/global policies and business priorities.

A DRACO PaaS can be deployed on IaaS service layer, using VMs, each one running a different DRE. The use of VMs allows to implement different DRACO PaaS independently from any underlying Hardware/Software configuration. Moreover, the FCAPS model makes the workflows self-managed or more simply self-* according to the autonomic system theory.

Considering the three-layer cloud stack discussed in Section III-B, the CM layer is the hardest one to be designed. In fact, cloud architects when designing a platform working at the CM layer have to plan fault tolerance, configuration, accounting, performance, and security management. Depending on the PaaS business logic complexity that cloud architects want to achieve, the design of PaaS is not so trivial at all. In this context the DRACO PaaS computing model can greatly facilitate the designing, implementation, and maintenance of a seamless CM layer, where complex PaaS can be developed, deployed, and executed. Figure 3 shows how DRACO PaaS can help the designing of a CM layer. More specifically, DRACO PaaS uses IaaS provided by the VIM layer in order to design a PaaS at the CM layer.

V. COMPOUNDING ELEMENTS OF DRACO PaaS

The actors of the DRACO PaaS computing model are based on elementary units called DRACO Elements (DREs). Each DRE can play one of three different roles:

- **Portal.** It is the node that allows to access the platform. It manages the initialization of workflow, providing the
required Jobnodes and setting up the environment in which they operate. It can also manage the requests for replacement of Jobnodes falling during the execution in a workflow.

- **Bridge.** It is the node that allows to bind the IaaS with the PaaS service level, requesting the VMs in which the DREs have to be deployed. The request for instantiation of VMs is made to IaaS Providers according to given policies. Different plugins allow to interact with different IaaS Providers such as Amazon EC2, OpenNebula, CLEVER. The Bridge node communicates with them through plugins containing the API for the communication with the underlying VIM layer. In DRACO the VMs are classified according to their virtual hardware characteristics (e.g., CPU, RAM, Storage, etc); this allows to define in DRACO PaaS different vmtypes.

- **Jobnode.** It is the type of node that allows to build workflows. Each Jobnode can play, in turn, three different roles: Master, Backup, and Worker. At the beginning each Jobnode is a neutral node, called Agoranoode, that doesn’t have specialization. When the Portal node assigns a workflow to a set of Jobnodes, each one polymorphically assumes one of the three roles:
  - **Master.** It is one per workflow. It is responsible to build and monitor the workflow. It tracks the status and manages the event of the single Worker Jobnodes and of the whole workflow.
  - **Backup.** It is a passive copy of the Master Jobnode that becomes active in case of a fault occurs.
  - **Worker.** It is the Jobnode that runs the single task or service of a workflow.

In simple term, a workflow is a combination of one Master Jobnode, a Backup Jobnode, and one or more Worker Jobnodes.

As the DRE is polymorphic, it has a common modular structure independently from the role played at run time. Figure 4 shows the basic modules of each DRE.

- **Brain.** It is the main component of the DRE because it manages the events detected by the SCs. This module loads the appropriate plugin to manage each specific event. Each type of DRE has the same Brain’s structure. However, Portal, Bridge, and Jobnode are specialized to have different behavior when an event happens.

- **Signaling Component (SC).** It acts as FCAPS manager. There are five SCs in a DRE, one for each FCAPS element. Each one waits for a target XMPP message in the SON. When a SC detects a new message, it decodes the event associated with the message and forwards it to the Brain module. In particular, according to the type of SON message, there are five different SCs:
  - **SC F.** It decodes all XMPP presence messages related to the events of login and logout of a DRE in a chat room.
  - **SC C.** It manages messages regarding the system configuration events, as the request for a new DRE, its initialization and the management of the policies that have to be applied in relation to a specific event.
  - **SC A.** It decodes the accounting events regarding the use of the platform performed by a tenant, as the monitoring of reserved resources, with the aim of producing a billing as precise as possible.
  - **SC P.** It manages the messages related to the performance of the DREs in order to trigger the load balancing strategies, through the major algorithms available in literature.
  - **SC S.** It decodes all messages related to security and privacy.

- **Communicator.** It manages the communication with the XMPP server and waits for messages coming from it and from the other DREs. This component filters the messages and forwards them to one of the five SCs.

- **Jobnode’s specific modules:**
  - **Computing Component (CC).** It is the component that manages the task execution of a Worker Jobnode.

- **Portal’s specific modules:**
  - **Tenant Listener.** Through this module, a Portal accepts requests for tenant’s workflow submission.
  - **Job Manager.** It manages single workflow submission requests received by the Portal. In particular, it prepares the environment for the workflow execution.

- **Bridge’s specific modules:**
  - **Getter (Jobnode, Portal, and Bridge).** They are three independent modules creating specific DREs. This middleware has the peculiarity to ensure the highest level of fault-tolerance, according to the principles of autonomic systems (i.e., self-*). Moreover, it is totally transparent for the tenant.

  Figure 5 shows an example of the DRACO PaaS environment involving the aforementioned DREs. DREs communicate each other by two independent networks, such as it happens in telephone networks:

- **SON (Signaling Overlay Network).** It is dedicated to report system events. It is based on the XMPP instant messaging (IM) protocol, by means of each DRE communicates each other as well as the users of a chat room.
Each message XMPP represents an event occurring in the system that will be managed by a specific SC, according to the most appropriate FCAPS component. In particular, there are three specific chat rooms:

- **CloudManagement**. The chat room where the Portal and Bridge nodes interact each other.
- **Agorà**. The chat room including Agoranode (i.e., Jobnodes without specialization). Portals select Jobnodes from this pool to satisfy the requirements of a given workflow (e.g., the number of Worker Jobnode for parallel processing).
- **JobRoom**. The chat room assigned to a workflow, including the Jobnodes belonging to a given workflow (i.e., the Master, Backup, and Worker Jobnodes) and the Portal node to which the workflow has been submitted.

- **CON (Computing Overlay Network)**. It is dedicated to data transfer between the DREs. It is based on the RESTful/CDMI approach for access resources in external file system (e.g., workflow data recovery). Instead, regarding the workflow data processing, the DRACO PaaS has been equipped with a Distributed Hadoop File System (HDFS) by means of the Worker Jobnode of a workflow can access files produced by each other during the execution of the workflow.

VI. WORKFLOW COMPOSITION

A workflow consists of Jobnodes and it is described by an xml file called “jobdescriptor”. This file is created by the tenant and contains the informations regarding:

- Number of Jobnodes (Master, Backup, and Worker) necessary for executing the workflow.
- Type of requested VMs.
- Worker Jobnode I/O.
- Tasks and files to be processed by any Jobnode Worker.

Each workflow runs in a different JobRoom and includes at least one Master Jobnode, one Backup, and several Worker Jobnode interconnected as a DAG. When a Portal receives a request by a tenant, it creates a xmpp JobRoom for the new workflow, then it gets the necessary Jobnodes according to the jobdescriptor. The Master Jobnode is responsible to supervise the events in the JobRoom.

As previously described, any DRE has a vmtype that describes the VMs features on which the DRACO PaaS workflow has to be deployed. The Portal, that receives the request, starts to look at the Agoranodes in the Agora room that have the same vmtype reported in the jobdescriptor. If one or more Jobnodes have been found, it moves them from the Agora chat room to the specific workflow JobRoom. If the Portal cannot find one or more Jobnodes, it sends a request of new DRE VMs to the Bridge, specifying the vmtype and the name of the workflow JobRoom. The new VMs can be pre-configured with the DRE code or can be contextualized at run time [8]. When the new VMs are created, they are joined to the workflow JobRoom.

VMs are set up by Portals when they are started. In fact, every time the administrator wants to start a DRE, he has to specify how many Agoranodes must be provided and their corresponding vmtype. When all the Jobnodes have joined the JobRoom, a Master Jobnode is elected and it assigns the rules to the Worker Jobnode according to the vmtype indicated in the jobdescriptor. During this phase, the Master Jobnode also communicates to Worker Jobnodes the tasks and files to be processed. Commonly tasks and files that have to be processed by any Jobnode are defined by the tenant during the phase of workflow definition. Tasks are the software program of services that have to be run by the different JobNodes. Files represent the input of the workflow (e.g., video to be transcoded). In the workflow the output of a Jobnode can become the input of other Jobnodes. The Master also arranges the dependencies of the Workers, i.e., how they are interconnected each others according to the jobdescriptor. When a Worker Jobnode finishes its task, it sends a message on the XMPP JobRoom alerting the other Workers. At this point, the other depending Workers can start their processing using the just produced output as a new input. When the Master collects all the messages of the correct elaboration of the Workers, it sends a message of shutdown on the JobRoom notifying the Portal that the workflow has been correctly executed. Thus, the Portal destroys the workflow, freeing the resources.

If a Worker Jobnode fails during the execution of the workflow, the Master sends a message to a Portal asking for a new Jobnode with the same vmtype. Thus, the failed Jobnode can be restored and can continue its processing. Figures 6 shows the activity diagrams when a Worker Jobnode failure occurs.

VII. IMPLEMENTATION

DRACO Paas is implemented in python language that makes it cross-platform. Python is an interpreted programming language that provides at the same time an Object Oriented and a scripting environment. Therefore, the choice of Python was due to its flexibility, because it makes the DRE easy to be programmed by tenant developers. As for all the emerging
Cloud solutions, interactivity is a very important factor. In fact, events that occur in the whole Cloud platform have to be notified in real time. For this reason, we chose XMPP as the communication protocol to allow DREs to interact in a real time fashion. Each DRE catches events through XMPP messages and creates new ones, producing the chain event - > message - > event that regulates all the operations of the platform.

Each DRE is univocally identified inside its XMPP chat room by a nickname that contains an UUID, generated when the Bridge gets the VMs. In order to make the process automatic, each DRE is registered using the XMPP in-band registration mechanism. The plug-in approach allows to easily extend DRACO PaaS by the definition of new messages and events. The structure of a standard XMPP message has been personalized with new attributes in order to support the DREs. In particular, 3 attributes were added:

- **thrdes**: It can be F, C, A, P and S according to the SC that will manage the specific FCAPS message;
- **cmtype**: It defines the nature of the message: shout, request data or send data;
- **cmsgctype**: It is a further specification of the cmtype.

Each DRE was implemented with 10 queues, for each of the five types of FCAPS messages (i.e., F, C, A, P, S) and five for the corresponding events. The message is moved through such queues and once it is processed, it generates a specific event.

Each message is caught by the “Communicator” module that executes a first filtering on all the received messages. According to the attribute thrdes, the received messages are inserted in the correct queue. The specific SC thread (i.e., F, C, A, P, S) pops the first message in the queue and, according to the info carried, it creates a new event object that is stored in a corresponding event queue. The five queues are popped by the Brain module that uses a polling mechanism to start the plugin that will manage the event.

Some DREs share the same events, while others manage exclusive events (e.g., the events for recovering Portal and Bridge). If the event consists of simple and fast operations, for instance an update of data in a structure, it is the Brain thread who executes directly the operation, otherwise new threads for a parallel working are started. The task is constituted of one or more .py files, where the operations that need to be executed from the Worker Jobnode are listed. A task is programmed by extending an abstract class defining the methods that have to be implemented for the correct execution in DRACO PaaS. These methods represent APIs for the tenant developers in order to program their workflow. For example, they allow to define the creation and the recovery of checkpoints. The checkpoint is a structure used for preserving data when a failure of a Worker occurs. A class called Container is used, and all the data are stored into it as attributes, in the form “name = value”. These data are defined by the programmer. When a commit of the checkpoint is started, the object Container is serialized and it is stored on the distributed file system of DRACO PaaS. Vice versa when the method Restore is called the Container is restored from the string recovered from the distributed file system. In our implementation, we use the Hadoop File System (HDFS - [9]), using replicas that allows low level fault tolerance and fast data recovery.

### VIII. Experimental Results

In this Section, we discuss several experiments we conducted on a real DRACO PaaS testbed. More specifically, we evaluated the performance of the “video transcoding” workflow using FFmpeg [10]. The objective of the workflow is to process an input video with resolution $r_1$ producing an output video with resolution $r_2$, where $r_2 < r_1$, $r_i \in \{160x120, 160x200, 240x160, 320x200, 320x240, 480x272, 640x200, 640x350, 640x480, 720x348, 720x350, 720x400, 800x600, 1024x768, 1152x864, 1280x720, 1280x800, 1280x1024, 1440x900, 1400x1050, 1680x1050, 1600x1200, 1920x1080, ...\}$, $i \in \mathbb{N}$.

The workflow has been arranged in our testbed considering a Portal, a Bridge, Master and $n$ Worker Jobnodes. The Master Jobnode splits the video in $n−1$ blocks assigning each one to a different Worker Jobnode we named “transcoding”. Each one of the $n−1$ “transcoding” Worker Jobnodes performs the video transcoding of its own block in parallel. More specifically each Worker Jobnode splits in turn its own video block in $m$ video chunks, sequentially processing them. The further splitting of each video block in $m$ video chunks allows us to achieve a greater level of granularity in order to define checkpoints in case of Worker fault. When each one of the $n−1$ Worker Jobnodes have finished, another Worker Jobnode we named “Merging” gathers the transcoded video blocks, shuffles and merges them, and produces the whole output transcoded video file. The experiments have been made considering the DRACO PaaS implementation discussed in Section VII.
In our configuration, we adopted the Hadoop Distributed File System (HDFS) and we configured each Jobnode in order to run FFmpeg for the video processing. The experiments were conducted in a testbed with the following hardware specifications:

- 9 blade server IBM X3630M3:
  - CPU: Intel Xeon E5506 @ 2.13 GHz (8 cores).
  - RAM: 36 GB.
  - Storage: 6 x 500 GB (SATA).

In each blade we allocated from 3 to 4 VMs. In our experiments, we evaluated the fault-tolerance of the platform assuming that failures may occur at the Worker Jobnodes and considering input video files of 1 Gigabyte. In particular, we varied three parameters: number of “Transcoding” Worker Jobnodes processing the video blocks in parallel, number of video chunks executed sequentially per “Transcoding” Worker Jobnode, and number of “Transcoding” Worker Jobnodes failed and replaced during the execution of the workflow. A Jobnode failure was generated by stressing the system in order to block several Worker Jobnodes. Our tests aim to investigate the response time related to the recovery of the failed Worker Jobnodes, considering different video chunk granularities. More specifically, in our tests, we divided the input video file in 4, 8, and 16 blocks, hence considering 4, 8, 16 “Transcoding” Worker Jobnodes. In each “Transcoding” Worker Jobnode, we assumed four levels of granularity, further dividing each video block respectively in 2, 4, 8, and 16 video chunks run sequentially in order to save checkpoints of the processed video in HDFS. Checkpoints are required in order to both minimize the amount of data lost and to allow a fast recovery of the processing activities once a failure occurs. Each experiment was repeated 30 times in order to consider mean values and confidence intervals at 95%.

Figure 7, 8, and 9 show the workflow completion time without failure, with 25%, and 50% of failures, considering 2, 4, 8, and 16 chunks sequentially processed by each “Transcoding” Worker Jobnode. On the x-axis we reported the number of “Transcoding” Worker Jobnodes each processing a video block in parallel, whereas, the y-axis shows the time taken for the workflow completion expressed in minutes. In these graphs, we can better observe the performance degradation due to the increment of the video chunks number.

On one hand the performance of the system are partially due to HDFS. In fact, from the experiments, it results that for a highly fault-tolerant system such as DRACO PaaS HDFS represents a bottleneck in writing. In fact, even though the recovery phase is somehow fast, the response time is affected by the latency in writing into the shared directory in HDFS. On the other hand the performance are also affected by the time taken by FFmpeg to read the input video in the shared directory and to detect the chunk to be processed.

For this reasons, depending on the fault-tolerance configuration of each workflow, it is required the right compromise among performance and fault-tolerance to minimize the amount of data lost in case of failure.

IX. Conclusion

Cloud computing is changing the way software systems are organized and managed. In particular, the designing of a PaaS can be very difficult, because software architects have to face several issues concerning fault-tolerance, configuration,
accounting, performance, security and the interaction with the underlying IaaS layer. In this paper, a Distributed Resilient Adaptable Cloud Oriented (DRACO) PaaS is presented. DRACO PaaS introduces a platform for the development of specialized PaaS and also provides a new computing model. The XMPP signaling system allows DRACO PaaS to quickly recover from workflow failures. More specifically, a video transcoding case of study has been analyzed. Our experiments showed how the bottleneck of our configuration is represented by the writing operations on HDFS. For this reason, we can assert that similar applications as the one we analyzed require a distributed file system able to optimize the concurrent writing operations. In future work, we plan to optimize the checkpoint system and secondly to more in deeply analyze the other FCAPS aspects, i.e., configuration, accounting, performance, and security.

Acknowledgements

The research leading to the results presented in this paper has received funding from the European Union’s Seventh Framework Programme (FP7 2007-2013) Project VISION-Cloud under grant agreement number 217019.

References