Dynamic Execution of Scientific Workflows in Cloud

E. Kail[1], J. Kovács [2], M. Kozlovszky[1,2] and P. Kacsuk[2,3]
[1] Óbuda University, John von Neumann Faculty of Informatics, Biotech Lab
Bécsi str. 96/b., H-1034, Budapest, Hungary
[2] MTA SZTAKI, LPDS, Kende str. 13-17, H-1111, Budapest, Hungary
[3] University of Westminster, 115 New Cavendish Street, London W1W 6UW
{kail.eszter, kozlovszky.miklos}@nik.uni-obuda.hu,
{jozsef.kovacs, kacsuk}@sztaki.mta.hu

Abstract - Scientific workflows have emerged in the past decade as a new solution for representing complex scientific experiments. Generally, they are data and compute intensive applications and may need high performance computing infrastructures (clusters, grids and cloud) to be executed. Recently, cloud services have gained widespread availability and popularity since their rapid elasticity and resource pooling, which is well suited to the nature of scientific applications that may experience variable demand and eventually spikes in resource. In this paper we investigate dynamic execution capabilities, focused on fault tolerance behavior in the Occopus framework which was developed by SZTAKI and was targeted to provide automatic features for configuring and orchestrating distributed applications (so called virtual infrastructures) on single or multi cloud systems.

I. INTRODUCTION

Over the last few years, cloud computing has emerged as a new model of distributed computing by offering hardware and software resources as virtualization-enabled services. Cloud providers give application owners the option to deploy their application over a network with a virtually infinite resource pool with modest operating and practically no investment costs. Today, cloud computing systems follow a service-driven, layered software architecture model, with Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS). In this paper we are primarily focusing on IaaS cloud services. In an IaaS environment the CPU, storage, and network resources are supplied by a collection of data centers installed with hundreds to thousands of physical resources such as cloud servers, storage repositories, and network backbone. It is the task of the cloud orchestrator to select the appropriate resource for an initiated application or service executed in the cloud.

Due to their rapid elasticity and almost infinite resource pooling capabilities cloud services have also gained widespread popularity for enacting scientific experiments. Scientific experiments are widely used in most scientific domains such as bioinformatics, earthquake science, astronomy, etc. In general they consist of multiple computing tasks that can be executed on distributed and parallel infrastructures.

Scientific workflows are used to model these scientific experiments at a high level abstraction. They are graphically represented by Directed Acyclic Graphs (DAGs), where the nodes are the computing tasks and the edges between them represent the data or control flow. Since these experiments mostly require compute and data intensive tasks the execution of scientific workflows may last for even weeks or months, and may manipulate even terabytes of data. Thus scientific workflows should be executed in a dynamic manner in order to save energy and time.

Dynamic execution has three main aspects: fault tolerance, intervention and optimization techniques. Fault tolerance means to continue the execution with the required SLA even in the presence of failures, or to adapt to new situations and actual needs during runtime. Since scientific workflows are mainly explorative by nature scientists often need to monitor the execution, to get feedback about the status of the execution and to interfere with it. Intervention by the scientist, workflow developer or the administrator may also be needed in a planned or in an ad-hoc manner. The third aspect of dynamic execution concerns with optimization mechanisms, such as performance, budget, time or power optimization techniques.

In this paper we are investigating the possibilities of executing scientific workflows dynamically in Occopus, we examine the required extensions to provide a reliable service for workflow orchestration and propose a fault tolerant mechanism which is based on the workflow structure and replication technique.

Occopus [7] is a newly introduced framework, developed by the Hungarian SZTAKI and was targeted to provide automatic features for configuring and orchestrating distributed applications on single or multi cloud systems.

Our paper is structured as follows: in the next section we give a brief overview about the related work on communication middleware used in distributed systems and on fault tolerance in cloud. In section III we introduce Occopus framework in a more detailed fashion. In section

IV we analyze the possibility of executing workflows in Occopus and section V introduces our solution in detail. Finally in the last section we give a brief insight into our fault tolerant proposal and the conclusion closes our work.

II. RELATED WORK

A. Communication middleware in the cloud

As distributed applications transcending geographical and organizational boundaries the demands placed upon their communication infrastructures will increase exponentially. Modern systems operate in complex environments with multiple programming languages, hardware platforms, operating systems and the requirement for dynamic deployments, and reliability while maintaining a high Quality-of-Service (QoS).

Cloud orchestration in general means to build up and manage interconnections and interactions between distributed services on single or multi cloud systems. At first the orchestrator allocates the most appropriate resource for a job from a resource pool, than monitors the functioning of the resource with a so called heartbeat mechanism. However, this mechanism only gives feedback about the physical status of the resources (CPU, memory usage, etc.) it cannot provide reliability in communication and data sharing.

Concerning scientific workflows the main challenge is to provide high availability, reliable communication, fault tolerance and SLA based service. In most scientific workflow management system a special middleware is responsible to maintain the connection, and data movement between the distributed services and to schedule the tasks according to available resources and predefined constraints (data flow model).

To provide a reliable, flexible and scalable communication between the services a suitable communication middleware is needed.

Communication middlewares can be categorized as Remote Procedure Call (RPC) oriented Middleware, Transaction-Oriented Middleware (TOM), Object-Oriented/Component middleware (OOCM) and Message-Oriented Middleware (MOM) [6].

RPC oriented Middleware is based on a client-server architecture and provides remote procedure calls through APIs. This kind of communication is synchronous to the user, since it waits until the server returns a response thus it does not enable a scalable and fault tolerant solution for workflows [5].

A Transaction-Oriented Middleware (TOM) is used to ensure the correctness of transaction operations in a distributed environment. It is primarily used in architectures built around database applications [14]. TOM supports synchronous and asynchronous communication among heterogeneous hosts, but due to its redundancies and control information attached to the pure data for ensuring high reliability, it results in low scalability in both the data volume that can be handled, and in the number of interacting actors.

An Object-Oriented/Component Middleware (OOCM) is based on object-oriented programming models and supports distributed object request. OOCM is an extension of Remote Procedure Calls (RPC), and it adds several features that emerge from object-oriented programming languages, such as object references, inheritance and exceptions. These added features make OOCM flexible, however this solution enables still limited scalability.

A Message-Oriented Middleware (MOM) allows message passing across applications on distributed systems. A MOM provides several features such as:

- asynchronous and synchronous communication mechanisms
- data format transformation (i.e. a MOM can change the format of the data contained in the messages to fit the receiving application [16])
- loose coupling among applications
- parallel processing of messages
- support for several levels of priority.

Message passing is the ancestor of the distributed interactions and one of the realizations of MOM. The producer and the consumer are communicating via sending messages. The producer and the consumer are coupled both in time and space; they must both be active at the same time. The consumer receives messages by listening synchronously on a channel and the recipient of a message is known to the sender.

Message queues are newer solutions for MOM, where messages are concurrently pulled by consumers, as well as a subscription based exchange solution, allowing groups of consumers to subscribe to groups of publishers, resulting in a communication network or platform, or a message bus. Message queues provide an asynchronous communication protocol. Its widespread popularity lies in not only its asynchronous feature but in the fact that it provides persistence, reliability and scalability enabling both time and space decoupling of the so called publishers and consumers.

Advanced Message Queuing Protocol (AMQP) [1] is an open standard application layer protocol for message-oriented middleware. RabbitMQ [3] is an open source message broker software (sometimes called message-oriented middleware) that implements the AMQP and can be easily used on almost all major operating systems.

B. Fault tolerance in cloud

Although cloud computing has been widely adopted by the industry, still there are many research issues to be fully addressed like fault tolerance, workflow scheduling, workflow management, security, etc [8]. Fault tolerance is one of the key issues amongst all. It is a complex challenge to deliver the quality, robustness, and reliability in the cloud that is needed for widespread acceptance as tools for the scientists' community.

To deal with this problem many research has been already done in fault tolerance. Fault tolerance policy can be proactive and reactive. While the aim of proactive

techniques is to avoid situations caused by failures by predicting them and taking the necessary actions, reactive fault tolerance policies reduce the effect of failures on application execution when the failure effectively occurs. Different fault tolerance challenges and techniques (resubmission, checkpointing, self-healing, job migration, preemptive migration) have been implemented using various tools (HAProxy, Hadoop, SGuard,) in the cloud. Also there are a lot of methods created for providing fault tolerant execution of scientific workflows in the cloud. Mostly they heavily rely on sophisticated and complex models of the failure behavior specific to the targeted computing environment. In our investigations we are targeting a solution that is mostly based on the workflow structure and data about the actual execution timings retrieved from provenance database.

III. Occopus Architecture

Occopus [7] (Fig. 1) has five main components: enactor issues virtual machine management requests towards the infrastructure processor; infrastructure processor, which is the internal representation of a virtual infrastructure (enabling the grouping of VMs serving a common aim); cloud handler enables federated and interoperable cloud use by abstracting basic IaaS functionalities like VM creation service composer, which ensures that VMs meet their expected functionalities by utilizing configuration management tools and finally the information broker that decouples the information producer and consumer roles with a unified interface throughout the architecture.

After receiving the required infrastructure description the enactor immediately compiles it to an internal representation. It is the role of the enactor to forward and upgrade the node requests to the infrastructure processor and to monitor the state of the infrastructure continuously during the setup and the existence of the infrastructure. This monitoring function is achieved by the help of the info broker. Among others this component is responsible for tracking the information flow between the nodes. If it notices a failure of a node or a connection it notifies the enactor. The enactor then upgrades the infrastructure description and forwards it to the infrastructure processor.

The infrastructure processor receives node creation and node destruction requests from the enactor. During creation infrastructure processor sends a contextualized VM requests to the cloud handler. Within the contextualization information the processor places a reference to some of the previously created attributes of VMs.Node destruction requests are directly forwarded to the cloud handler component.

The cloud handler as its basic functionality, provides an abstraction over IaaS functionalities and allow the creation, monitoring and destruction of virtual machines. For these functionalities, it offers a plugin architecture that can be implemented with several IaaS interfaces (currently Occopus supports EC2, nova, cloudbroker, docker and OCCI interfaces).

The main functionality of the Service Composer is the management of deployed software and its configuration on the node level. This functionality is well developed and even commercial tools are available to the public. Occopus Service Composer component therefore offers interfaces to these widely available tools (e.g., Chef, Puppet, Docker, Ansible).

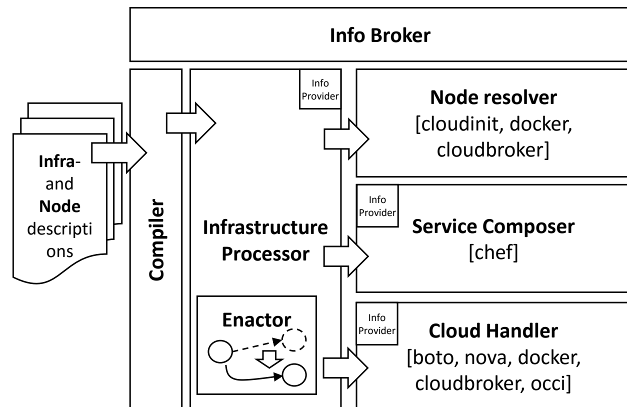

Figure 1. Occopus architecture

IV. Scientific workflows in Occopus

A. *Scientific workflows*

In Occopus framework a virtual infrastructure can be built upon a directed acyclic graph representing some complex scientific experiment which consists of numerous computational steps. The connection between these computational steps represent the data dependency, in other words the dataflow during the experiment. With Occopus the infrastructure descriptor would contain the needed resource requirements for each task and also the SLA for the tasks or for the whole workflow. An SLA requirement could be time or budget constraint or a need for green execution. In such a scenario the execution could be seen as data flowing across the VMs starting from the entry task executed on the first VM, terminating by the exit task executed on the last VM. In a scenario like this every computational step is mapped to an individual VM. After submitting the virtual infrastructure descriptor based on the workflow model Occopus would support the creation of an infrastructure like this. This type of workflow creation and execution is called Service Choreography in related works.

1) Advantages

Concerning the execution of scientific workflows in Occopus has several advantages:

The resources are available continuously, it means that task execution are not forced to wait for free resource. The infrastructure is built up easily without expertise knowledge of the individual cloud providers. The monitoring is also provided by the Occopus framework. There is no need for scheduling and resource allocation is done by Occopus based on the virtual infrastructure descriptors.

2) Problems with scientific workflows executed in Occopus

Concerning scientific workflows there might arise a lot of issues:

Scientific experiments are being data and compute intensive which may last for even weeks or month and may use or produce even terabytes of data. Due to the long execution time many failure could arise. Types of faults that can arise during execution and need to be handled in order to provide a fault tolerant execution can mainly be categorized to the following categories: VM faults, programming faults, network issues, authentication problems, file staging errors, and data movement issues. In order not to lose the already calculated work, fault tolerance must be provided.

• When a node fails, the computation which was done by this node is lost. As described in the previous section Occopus monitors the nodes of the virtual infrastructure and when the enactor notices a failed node, it is deleted from the virtual infrastructure list and a new one is created. The execution could be restarted on it. But what should happen with the data that was consumed by this failed node?

• Let us focus on only one aspect of SLA-s (Service Level Agreement), namely the time constraints. Scientific workflows are often constrained by soft or hard deadlines. While soft deadline means that the proposed deadline should be met with a probability p, hard deadline means that the results are useless after the deadline. When there is a failure upon recovery the makespan of the whole workflow is increased and maybe deadlines cannot be met. How can it be ensured that SLAs are met?

• Fault tolerance technique should also be concerned when executing scientific workflows in the cloud. The most frequent fault tolerant techniques in the cloud are using resubmission and replicas. How can it be ensured that more than one successors of the same type (replica) is able to receive the results of the predecessor(s) and how can it be provided that the number of replicas can change dynamically in time?

In the next section we are looking for the best solution that address the issues described above.

V. SOLUTIONS

There are two main widespread used alternatives that can give solutions for the above mentioned problems and are supported by open source softwares. One of them is based on service discovery feature, while the other uses a message queueing system.

A. Service Registry

Service discovery is a key component of most distributed systems and service oriented architectures deploying more services. Service locations can change quite frequently due to host failure or replacement, similarly to a scientific workflow execution. A node must discover somehow the IP address and the port number of the peer application. One solution is to use a dedicated, centralized service registry node.

A service registry, is a database of services, their instances and their locations. The main task of it is to register hosts, ports and authentication credentials, etc. Service instances are registered with the service registry on startup and deregistered on shutdown. Clients of the service query the service registry to find the available instances of a service. If a node fails the service registry database is upgraded with the new node.

Concerning workflow execution if a node fails then the service registry updates its database with the new client, but the computation that was already done is lost. Also the consumed data is lost with the failed node. If a computation has successfully terminated on a VM then the results of this computation task can be (should be) stored in provenance database, but because of the nature and size of the provenance database, it must be located on a permanent storage. To retrieve these data from this storage may have high latency due to geographic location which can be far from the cloud provider.

The flexibility of the solution is also not so good. In this case the nodes know where to send data so they use the synchronous remote procedure call or the asynchronous message passing middleware. As it was mentioned already in the related work the RPC model does not support large volume of data and does not support reliable transport of data. Also with message passing middleware the communication abstract is the channel and a connection must be set up between producer and consumer and consumer listens for the channel synchronously.

Using this solution a special agent would be needed to orchestrate the execution of the workflow itself. Without this agent this solution would work only for small workflows that use does not move high volume of data does not need long time to be executed and the reliability of the resources are high.

B. Message Queuing

Using message queues would simplify almost all of the above mentioned problems. Advanced Message Queuing (AMQ) Protocol is an open standard message oriented middleware. In this approach the message producer does not send the message directly to a specific consumer instead characterize messages into classes without knowledge of which consumer there may be. Similarly, consumers only receive messages that are of interest, without knowledge of existing producers. AMQP operates over an underlying reliable transport layer protocol such as Transmission Control Protocol (TCP). The basic idea is that consumers and producers use a special node to accomplish message passing, which serves as a rendezvous point between senders and receivers of messages. They are the queues which are buffers that temporary or permanently store the messages. The middleware server has two main functionalities: one of them is buffering the messages in memory or on disk when the recipient cannot accept it fast enough and the other one is to route the messages to the appropriate

queue. When a message arrives in a queue the server attempts to deliver it to the consumer immediately. If this is not possible the message is stored until the consumer is ready. If it is possible the message is deleted from the buffer immediately or after the consumer has acknowledged it. The reliability lies in this feature. The acknowledgments could be sent only when the node had successfully processed the data. The scalability and fault tolerance can be realized by clustering the same type of nodes. In this solution the consumers and producers are not known to each other and there can be more than one consumer belonging to a single queue. The power of AMQP comes from the ability to create queues, to route messages to queues and even to create routing rules dynamically at runtime based on the actual environmental conditions. This feature would enable to realize an SLA based, fault tolerant execution for workflows.

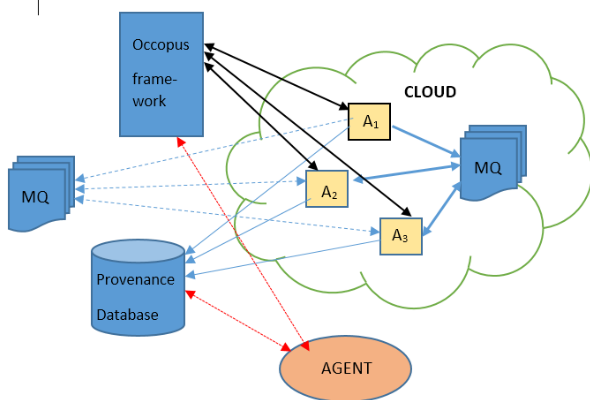

Figure 2. Possible architecture with Message Queueing

In Fig. 2 a possible architecture for executing scientific workflows with Occopus can be seen. The abstract model of the scientific workflow consists of 3 tasks (A_1, A_2 and A_3) for each an individual VM is started in the cloud. These are provided by the Occopus framework. Also Occopus does the monitoring of the resources, as well as the infrastructure upgrading. All of the tasks are communicating with the MQ (message queue), so they are not aware of each other. The MQ can be positioned also in the cloud on a VM or on external storage. It depends on the amount of data that must be shared between the tasks and the geographic location of the VMs. The Agent is only responsible for monitoring the workflow execution according to the predefined constraints (the input, output format of the data, time constraints, etc.) and according to the SLA to request a virtual infrastructure change from the Occopus framework (for example to start more or less replicas of the tasks).

VI. Fault Tolerance Method Based on Workflow Structure

Concerning time critical applications a reliable fault tolerant method should be provided. In this section we lay down the bases of our fault tolerant framework that uses replicas in order to ensure that time critical workflows can be successfully terminated before the soft or hard deadline with a probability of p. In our solution every task in a workflow is assigned certain number of replicas. The number of replicas is determined by the estimated execution time of a task, the structure of the workflow, the failure zone of a task (which is the affected tasks in the case of a failure of a given task), and the estimated failure detection time and resubmission time. Before Occopus starts to build the infrastructure the algorithm is executed to determine the number of replicas of each task and the infrastructure is built. When during execution unexpected situation happens (for example too many failures occurring, or even that there is no error) than the number of replicas can be changed accordingly since Occopus is able to upgrade the infrastructure. Determining the exact details of our algorithm is our future work.

VII. Conlusion

In this paper we have introduced Occopus, a one click cloud orchestrator framework, which supports distributed applications to be executed in single or multi homed clouds. We have investigated the advantages and problems of having scientific workflows being executed with Occopus and gave a proposal for a communication middleware which would provide a reliable, fault tolerant workflow execution environment. We gave a first insight for a fault tolerant method that can be used with Occopus as well and which detailed work out determines our future research direction.

References

[1] AMQP Advanced Message Queuing Protocol, Protocol Specification, Version 0-9-1, 13 November 2008.

[2] E. Curry, Message-Oriented Middleware, in: Q.H. Mahmoud (Ed.), Middleware for Communications, John Wiley and Sons, Chichester, England, 2004, pp. 1–28.

[3] A. Videla, J.W. Williams, RabbitMQ in Action: Distributed Messaging for Everyone, MEAP Edition Manning Early Access Program, 2011.

[4] P.T. Eugster, P. Felber, R. Guerraoui, A.-M. Kermarrec, “The many faces of publish/ subscribe”, ACM Comput. Surv. 35 (2) (2003) 114–131.

[5] K. Geihs, Middleware challenges ahead, IEEE Comput. 34 (6) (June 2001) 24–31.

[6] M. Albano et al. Message-oriented middleware for smart grids, Computer Standards & Interfaces 38 (2015) 133–143.

[7] G. Kecskeméti, M. Gergely, A. Visegrádi, Zs. Németh, J. Kovács, P. Kacsuk, One Click Cloud Orchestrator: bringing Complex Applications Effortlessly to the Clouds, WORKS 2014.

[8] A. Bala, I. Chana, Fault Tolerance- Challenges, Techniques and Implementation in Cloud Computing, IJCSI International Journal of Computer Science Issues, Vol. 9, Issue 1, No 1, January 2012