UNIVERSITY OF
FORWARD
THINKING
WESTMINSTER

WestminsterResearch
http://www.westminster.ac.uk/westminsterresearch

A Cloud-agnostic Queuing System to Support the Implementation of Deadline-based Application Execution Policies

Kiss, T., Deslauriers, J., Gesmier, G., Terstyanszky, G., Pierantoni, G., Abu Oun, O., Taylor, S.J.E., Anagnostou, A. and Kovacs, J.

NOTICE: this is the authors' version of a work that was accepted for publication in Future Generation Computer Systems. Changes resulting from the publishing process, such as peer review, editing, corrections, structural formatting, and other quality control mechanisms may not be reflected in this document. Changes may have been made to this work since it was submitted for publication. A definitive version was subsequently published in Future Generation Computer Systems.

The final definitive version in Future Generation Computer Systems is available online.

© 2019. This manuscript version is made available under the CC-BY-NC-ND 4.0 license https://creativecommons.org/licenses/by-nc-nd/4.0/

The WestminsterResearch online digital archive at the University of Westminster aims to make the research output of the University available to a wider audience. Copyright and Moral Rights remain with the authors and/or copyright owners.

Whilst further distribution of specific materials from within this archive is forbidden, you may freely distribute the URL of WestminsterResearch: ((http://westminsterresearch.wmin.ac.uk/).

In case of abuse or copyright appearing without permission e-mail repository@westminster.ac.uk

Available online at www.sciencedirect.com

SciVerse ScienceDirect

A Cloud-agnostic Queuing System to Support the Implementation of Deadline-based Application Execution Policies

Tamas Kiss[1], James DesLauriers[1], Gregoire Gesmier[1], Gabor Terstyanszky[1], Gabriele Pierantoni[1], Osama Abu Oun[2], Simon J E Taylor[3], Anastasia Anagnostou[3], Jozsef Kovacs[4],

[1]Centre for Parallel Computing, University of Westminster, 115 New Cavemdish Street, London W1W 6UW, United Kingdom
[2]Cyber Security Group, School of Computing, University of Kent, Canterbury, United Kingdom
[3]Modelling & Simulation Group, Brunel University London, Kingston Lane, Uxbridge UB8 3PH, United Kingdom
[4]MTA SZTAKI, Budapest, Hungary

Abstract

There are many scientific and commercial applications that require the execution of a large number of independent jobs resulting in significant overall execution time. Therefore, such applications typically require distributed computing infrastructures and science gateways to run efficiently and to be easily accessible for end-users. Optimising the execution of such applications in a cloud computing environment by keeping resource utilisation at minimum but still completing the experiment by a set deadline has paramount importance. As container-based technologies are becoming more widespread, support for job-queuing and auto-scaling in such environments is becoming important. Current container management technologies, such as Docker Swarm or Kubernetes, while provide auto-scaling based on resource consumption, do not support job queuing and deadline-based execution policies directly. This paper presents JQueuer, a cloud-agnostic queuing system that supports the scheduling of a large number of jobs in containerised cloud environments. The paper also demonstrates how JQueuer, when integrated with a cloud application-level orchestrator and auto-scaling framework, called MiCADO, can be used to implement deadline-based execution policies. This novel technical solution provides an important step towards the cost-optimisation of batch processing and job submission applications. In order to test and prove the effectiveness of the solution, the paper presents experimental results when executing an agent-based simulation application using the open source REPAST simulation framework.

Keywords: cloud computing, container technologies, deadline-based auto-scaling, JQueuer, MiCADO, agent-based simulation

1. Introduction

Cloud computing offers scalable and on-demand access to large amounts of computational and data resources. Operating System (OS) level virtualization, also known as container-based virtualization has recently attracted much attention due to its near-native performance and low virtualization overhead [1]. In science gateways, containers simplify packaging applications so as to run on any cloud independently of the cloud's configuration. A container orchestration engine takes multiple resources in the cloud, combines them into a single pool, and provides an abstracted layer between the cloud resources and the application container that run on these resources. Most applications can be containerised along with all their libraries so as to run in

any cloud without the need to install any prerequisites. Containers are stateless which makes them suitable for services that perform single functions and do not need to store data in the containers. A Web service is an example of these stateless services where it is possible to create/clone several containers so as to process HTTP requests concurrently.

On the other hand, several types of batch processing applications in science and business gateways, for example discrete-event and agent-based simulation or image/video processing, require a mechanism to launch containers and provide each one of them with the jobs or data which should be processed. These applications typically consist of hundreds or thousands of scenarios which need to be executed, usually independently from each other. Some scenarios are lightweight allowing several of them to be executed at the same time on the same machine in different containers, while others consume large amount of resources and need longer time to be accomplished. These applications are all dealing with the submission of jobs where the results need to be kept after execution.

The need to provide the containers with jobs or data and collect the output after execution is not the only difference between stateless services and batch processing applications. Typical policies applied when scaling containers are also different. Scaling up/down the containers of a stateless service may depend on the load on its containers, CPU and memory consumption, number of requests, average response time, etc. However, in batch processing applications, we might need to take scaling decisions depending on completely different types of policies such as the time required to process a job, the deadline to finish all jobs in the queue, or the length of the queue. In some cases, we do not need to scale up the containers if there are no more jobs in the queue, even if all containers are consuming 100% of their resources. The time required to finish a job (job duration) is one of the most important factors to be taken into consideration in order to decide the number of containers needed to process jobs in the queue. Job duration may differ completely from one job to another making this estimation even more complex. For example, a queue of video files to be processed might contain videos of different lengths, from one minute to one hour. If the overall execution of the experiment needs to be finished by a given deadline, then this results in the need to periodically auto-scale up/down the containers based on user-defined scaling policies.

While Stateless Services are widely supported by current technologies, there is very limited or no support for job-queuing, execution and related policy-based auto-scaling mechanisms which are required by batch processing applications. The lack of these components forces application developers to spend time and money to develop proprietary tools or to customize open source libraries to work in container-based environments. These components should also be available to run on public and private clouds based on various technologies. Therefore, the solutions should be agnostic to the underlying cloud middleware.

In this paper we propose JQueuer, a cloud-agnostic queuing system that supports the scheduling of a large number of jobs in containerised cloud environments. We also demonstrate that JQueuer can be integrated with a managed container platform, such as MiCADO, Microservices-based Cloud Application-level Dynamic Orchestrator that was presented in [2]. MiCADO originally supported the scaling of stateless services only based on performance based metrics, such as memory or CPU utilisation. By extending MiCADO with JQueuer as an external service, it is possible to realise deadline-based execution policies required by job submission or batch processing applications. The job queuing and scheduling is provided by JQueuer, while MiCADO is responsible for the implementation and enforcement of suitable auto-scaling policies. The applicability of the implemented solution is demonstrated by designing a deadline-based execution policy for an agent-based simulation application using the open source REPAST framework [3] and executing several experiments to assess its efficiency. While these experiments demonstrate that JQueuer and MiCADO can be efficiently utilised to execute such experiments, it also has to be noted that the focus of this paper is to demonstrate the technical concepts and their usability, and not to design and optimise deadline based execution policies. This latter was out of our scope and will only be covered in future work.

Based on the above, the major contribution of this paper is a novel cloud agnostic queuing system for containerised environments. JQueuer is a stand-alone component that can be easily reused by other

researchers or developers seeking a suitable job-queue for container-based execution. Additionally, the paper also provides an example for the utilisation of this component when implementing deadline-based execution policies.

The major motivation behind our work was inspired by the requirements of real-life industry, public sector and research focused use-cases currently being investigated in two European research projects. In the COLA (Cloud Orchestration at the Level of Application) project [5], Saker Solutions Ltd. [37], a UK-based simulation consultancy company is developing a discrete-event based simulation application for evacuation modelling. In the CloudiFacturing (Cloudification of Production Engineering for Predictive Digital Manufacturing) [39] project DSS Consulting [38], a Hungarian technology company is working together with a truck component manufacturer to optimise its production processes, also using discrete-event simulation. In both scenarios large scale simulation experimentation is required that needs to be significantly speeded up using cloud computing resources. These simulation runs need to finish by given deadlines otherwise timely, sometimes critical decisions cannot be made. On the other hand, the number of jobs to execute in these simulations and their duration can significantly change between scenarios. Therefore, it is not possible to reliably predict the volume of resources required. In current practice, both companies use fixed internal resources that are limited, not scalable and also expensive to maintain. When migrating their applications to the cloud, they both would like to utilise its elastic nature fully, leaving it to the underlying layers to define the optimal volume of resources to be utilised, and optimise the execution for reaching the given deadline while minimising costs. The agent-based simulation scenario presented in this paper, although it comes from academic research, is very similar in nature to these industry use-cases and therefore it provides a good basis for experimentation.

The rest of this paper is organised as follows. Section 2 discusses the state of the art and related work. Section 3 describes the MiCADO framework that will be used as the managed container platform to be integrated with JQueuer and where the deadline-based execution policy is implemented. Section 4 explains the structure of an experiment which will be used when designing JQueuer. Section 5 describes the design and implementation of JQueuer, while Section 6 explains how it has been integrated with MiCADO. The implementation of a REPAST experiment and its performance results are presented in Section 7. Finally, Section 8 concludes this paper and outlines future work.

2. Related work

A number of solutions and studies have been proposed to tackle the problem of auto-scaling jobs and services execution in container-based cloud environments. This section first compares the job-queuing and auto-scaling capabilities of three widely used container orchestration engines, then looks at open-source job scheduling systems, and finally analyses some managed solutions offered by hosted cloud platforms.

Docker Swarm [7] is an orchestration tool which manages a cluster of Docker Engines running Docker containers. Docker provides the service concept in which the services are “containers in production”. A service only runs one image, but it codifies the way that image runs, what ports it should use, how many replicas of the container should exist, and so on. Scaling a service changes the number of container replicas running that piece of software, assigning more computing resources to the service in the process. The containers in a service are stateless. It is widely used and relatively simple to interface with, but, out-of-the-box, Swarm is not suitable for queueing jobs.

Kubernetes [8] is another orchestration tool for Docker, as well as for other container runtimes. Kubernetes is a more complex orchestrator, and as such, offers a larger variety of workloads, including the Kubernetes Job. The Job controller can deploy and scale sets of containers in parallel and ensure that a specific number of containers run to successful completion. This offers more certainty in job completion that does Swarm, and permits batching jobs, but Kubernetes itself offers no built-in system for queuing and no scaling policy for meeting a deadline.

Apache Mesos is a cluster manager that provides efficient resource isolation and sharing across distributed applications [19]. There are several projects which have grown out of Mesos, including container orchestration. Mesos also features a Job framework called Metronome [20]. Less mature than Kubernetes Jobs, Metronome allows for the creation of containers on a schedule, with set resources. However, at the time

of writing, there is not yet any mechanism built-in to Metronome to support a job queue that executes to a set deadline. In another example, Mesos container orchestration is interfaced with Jenkins, an open-source automation server, to deploy and scale jobs in containers according to a fixed scaling policy based on job queue length [21]. Although Jenkins and this Mesos plugin provide a working model for a scalable job queuing system, its scaling policy is fixed and static.

While these container orchestrators offer scalability, they do not provide monitored job-queuing support or complex, deadline-based scaling policies out-of-the-box. In comparison, JQueuer monitors and executes queued jobs in containers, while MiCADO provides flexible deadline-based scaling policies and auto-scaling of the underlying cloud resources.

There are several widely used job scheduling systems in the open-source domain that are starting to support containerised environments. However, these solutions are far too heavy and their Docker support is currently limited. HTCondor is an open-source high throughput computing software framework for coarse-grained distributed parallelization of computationally intensive tasks [22]. HTCondor has support for launching containers but it does not communicate with container orchestrators which prevents the applications from using services offered by these orchestrators, such as networking between containers or Docker Compose. SLURM (Simple Linux Utility for Resource Management) [23] is another open source, fault-tolerant, and highly scalable cluster management and job scheduling system for large and small Linux clusters. While both HTCondor and SLURM support integration with an elastic cluster to auto-scale the underlying compute nodes, they are far from being cloud-agnostic, relying on the proprietary cloud services of only a few specific cloud service providers. Furthermore, the auto-scaling logic of these clusters are generally inflexible and do not support queue monitoring and deadline-based scaling out-of-the-box. In comparison, JQueuer and MiCADO seek to avoid such vendor lock-in and provide flexible scaling rules.

CQueue is a promising job scheduling service, fully supporting Docker containers, which offers job queue submission and the parallel execution of those jobs in containers on any cloud provider [24]. CQueue, similarly to HTCondor and SLURM, takes a stateless approach to the queue. It creates a new container when a job is pulled from the queue and kills that container when the job completes. However, large-scale scientific and industry simulation software may be resource demanding in which case creating a new container for each job can generate large overheads. These overheads lead to an overall decrease in efficiency on completing the queue. The design described in this paper takes a more stateful route to the container, running it as a service which is able to pick up job after job from the queue, and is only killed when a policy enforces a scale-down, or the experiment ends.

Within the domain of large hosted providers, Amazon Web Services (AWS) offers AWS Batch [25] as a part of the Elastic Container Service (ECS). This is a full featured queue system which schedules, scales and executes jobs in containers across multiple virtual machine nodes and offers optimised scaling based on job resource requirements. AWS Batch does not scale to a deadline by default and is only available inside AWS. Therefore, it is not a solution for private or community clouds, or any other public cloud. Microsoft Azure offers a very similar solution, also called Batch [26]. Just like the AWS solution, it offers container and virtual machine scaling and submission to a queue with job execution in the container, and even has a promising custom metric auto-scaler. However, just like the AWS solution, it ties any user to one specific cloud.

Within academic research, there is ample work which compares scaling algorithms for deadline-constrained workflows in the cloud, done at a theoretical level using a cloud simulator to mimic the behaviour of scalable virtual machines [33] [34]. Other research efforts look at implementing scaling mechanisms which manage auto-scaling virtual machines in order to complete queued jobs before a given deadline [35] [36]. However, each of these papers is focused at the level of virtual machines, and does not delve into container environments. According to our best knowledge, no research to date has been dealing with the queued execution of jobs in containers which scale automatically along with the underlying cluster in order to finish the queue of jobs before a set deadline.

When compared to the above detailed solutions, JQueuer has been designed to implement containers as services, support flexible scaling policies based on custom metrics, and be platform and cloud-independent. In addition, it can work with any container orchestration engine.

3. MiCADO – Microservices–based Cloud Application-level Dynamic Orchestrator

MiCADO is an application-level multi-cloud orchestration and auto-scaling framework that is currently being developed in the European H2020 COLA project [5]. The concept of MiCADO is described in detail in [2]. In this section only a high-level overview of the framework is provided to explain its architecture, building blocks and implementation.

The generic, technology independent architecture of MiCADO is presented in Figure 1. MiCADO consists of two main logical components: Master Node and Worker Node. Master Node is the head of the cluster performing the collection of information on microservices, the calculation of optimised resource usage, the decision making, and the realization of decisions related to handling resources and to scheduling microservices. Worker Nodes are volatile components representing execution environments for the microservices. These nodes are continuously allocated/released based on the dynamically changing requirements of the running microservices. Once a new Worker Node is allocated and attached to the cluster, the Master Node utilizes its resources by allocating microservices on it.

The MiCADO Master Node (box with dashed line on the left in Figure 1) includes six components. MiCADO Submitter is the primary service endpoint for creating an infrastructure to run an application, and managing this infrastructure and the application itself. The incoming description is interpreted by the MiCADO Submitter and related parts are forwarded to other key components. Creating new MiCADO Worker Nodes and deploying application containers on these Worker Nodes are the responsibility of Cloud Orchestrator and Container Orchestrator components, respectively. The Cloud Orchestrator is responsible for communication with the Cloud API for allocating and releasing resources, and building up and shutting down new MiCADO Worker Nodes when necessary. The Container Orchestrator allocates new microservices (realised by containers) on the Worker Nodes, keeps track of their execution and destroys them if necessary. The Monitoring System collects information on load of the resources and on resource usage of the container services, and provides this information for the other components on the Master Node. It also provides alerting functionality in relation to the measured attributes to detect values that require reaction. The Policy Keeper implements policies and makes decisions related to allocating/releasing cloud resources and scheduling container services among the Worker Nodes. Moreover, this component makes sure that the Cloud and the Container Orchestrator are instructed in a synchronised way during the operation of the entire system. The Execution Optimizer is a background microservice performing long-running calculations on demand for finding optimised setup of both cloud resources and container infrastructures.

MiCADO Worker Nodes (boxes with dashed line on the right in Figure 1) contain the Node/container monitor that is responsible for measuring the load of the resources and the resource usage of the container services. The measured attributes are then provided to the Monitoring System running on the Master Node. The Container Executor starts, executes and destroys containers upon requests from the Container Orchestrator. Container components are realizing the user services defined in the (container) infrastructure description submitted through the MiCADO Submitter on the Master Node.

The current implementation of MiCADO utilizes Occopus [6], an open source multi-cloud orchestration solution as Cloud Orchestrator that is capable of launching virtual machines on various private (e.g. OpenStack or OpenNebula-based) or public (e.g. Amazon Web Services or CloudSigma[31]) cloud infrastructures, and also via the CloudBroker Platform [30] (please note that in the current implementation of MiCADO all virtual machines must be mapped to the same cloud, therefore multi-cloud applications where virtual machines of the same application are running in different clouds, are not supported). For Container Orchestration the MiCADO prototype applied in this paper uses Docker-Swarm [7]. However, it is worth mentioning that the latest version of MiCADO also supports Kubernetes [8] as this component. The monitoring component is based on Prometheus [9], a lightweight, low resource consuming, but powerful monitoring tool. The MiCADO Submitter and Policy Keeper components were custom implemented during the COLA Project. The current MiCADO prototype does not include the Optimiser component, its design and

development has just started at the time of writing this paper. Infrastructure and policy descriptions (left hand side of Figure 1) are provided in the form of a TOSCA-based (Topology and Orchestration Specification for Cloud Applications) [10] Application Description Template (ADT) (for further details regarding MiCADO ADTs please see [11]). Finally, application logic is deployed in Docker containers on the MiCADO worker nodes.

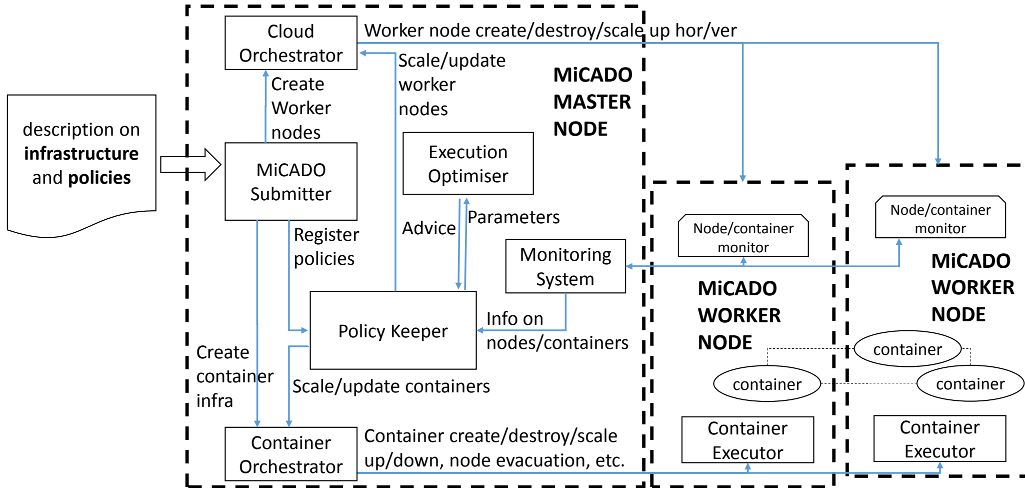

Figure 1 Generic, technology independent architecture of MiCADO

4. Experiment Structure

In previously mentioned batch processing or job submission applications, for example simulations or image/video processing, there are always numerous scenarios that need to be completed on large computational resources. However, as these application areas evolved independently, the vocabulary used to identify the different units of execution are rather varied. In order to avoid any confusion or misunderstanding, in this section we define and present these units of execution as experiment, job and task, as they are illustrated in Figure 2. The figure illustrates the structure of the JSON (JavaScript Object Notation) [12] file that is required to define and submit an experiment in JQueuer, and therefore it indicates the purpose and role of these various untis of execution.

4.1 Experiment

An experiment consists of two parts. The first part is a set of global parameters (upper part of Figure 2), and the second part consists of a list of jobs (lower part of Figure 2). Global parameters define the desired cloud infrastructure and scaling properties. For launching the cloud infrastructure, these include the endpoint of the container management platform, the application's Docker image and the resources of the virtual machine workers to be provisioned. For scalability, the necessary parameters include the deadline by which the experiment should be finished, and an estimated average execution time for each job. The full list of global parameters can be seen in the upper part of Figure 2.

The second part contains the experiment jobs and tasks - the data necessary to run the intended workloads on the infrastructure defined by the global parameters. An experiment (including jobs and tasks) might be stored using various formats. In case of JQueuer this format is JSON, but other possible options include XML (Extensible Markup Language) or YAML (Yet Another Markup Language). The next sections describe the definition of jobs and tasks within an experiment.

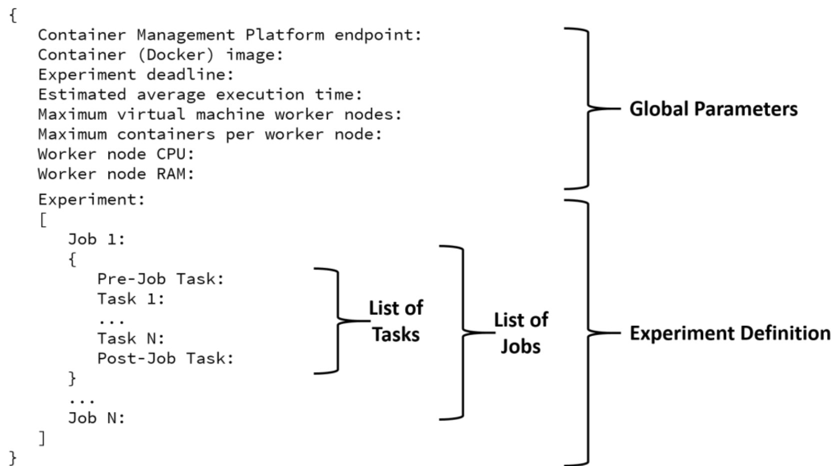

Figure 2 Definition of global parameters and list of jobs and tasks for an experiment (experiment.json)

4.2 Job

A job consists of three parts: Pre-job Command (1), Tasks (2) and Post-job Command (3). While the first and third parts are optional, the second part is required. Pre-Job, Post-Job and task commands will be invoked within the container so as to launch an application or execute a system call, for example.

1. Pre-Job Command (Optional): It is the command that should be invoked in the container at the beginning of each job and before running the tasks. The command might be used to initialize the parameters or to reserve the resources which are needed to execute a task.
2. Tasks (Required): It is a list of tasks that should all be executed sequentially in the same container. If any task fails for any reason, the whole job will be considered as "failed" and it will be re-queued or cancelled, depending on the configuration of the system. Quite often, each job consists of one task only. However, in some experiments tasks are depending on each other and need to be executed in a certain order inside a job (e.g. the first task would parse the argument and download files from a server, the second task would run the application, while the third task will upload the results to a server). Another motivation to put multiple tasks inside one job is to enhance network utilization and reduce overhead by fetching and executing multiple inputs in a batch (e.g. fetching of multiple images at once in order to be analysed sequentially instead of fetching one image at a time).
3. Post-Job Command (Optional): This command should be executed after finishing all the tasks of the job and before getting a new job from the job queue. It might be used to free the resources, reset the parameters, etc. A job is considered "accomplished" when all its tasks have executed successfully.

4.3 Task

A task is the smallest unit in this structure. It contains the command line that should be called in the container and the parameters (arguments) which should be passed along with this command.

An example of the above structure is a simulation experiment. The experiment has global parameters including the container's image. Let there be a thousand jobs in this experiment and let each job consists of one task. The task in this case will contain the command line of the simulation application that needs to be executed inside a container and the different sets of parameters that this command requires.

5. JQueuer Design and Implementation

JQueuer is a queuing system that can be used in conjunction with container technologies to support the execution of a large number of jobs.. JQueuer is a distributed system that is composed of two independent

components: JQueuer Manager and JQueuer Agent (Figure 3). In the following, we are going to discuss the structure and functionality of each of these.

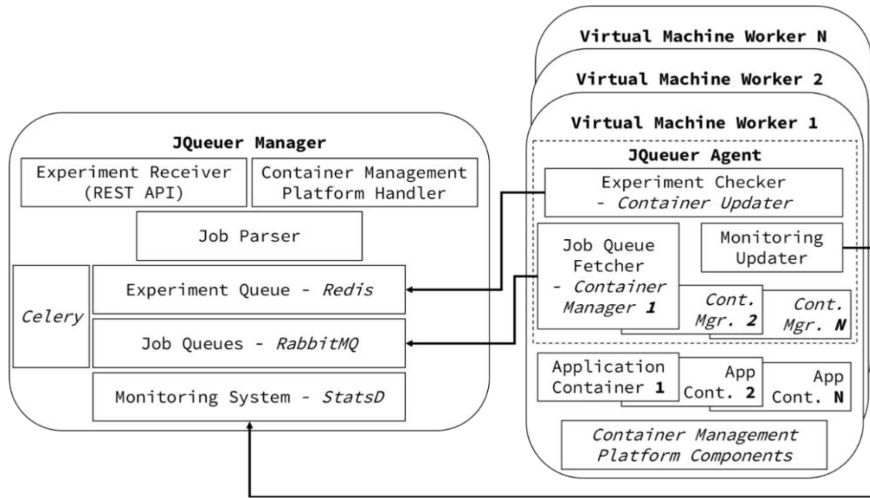

Figure 3 JQueuer Manager and Agent - design and implementation with a generic container management platform

5.1 JQueuer Manager Design

JQueuer Manager is the main component of the JQueuer system. It runs externally of any container management platform (running Docker Swarm/Kubernetes) as a standalone component. JQueuer Manager consists of several sub-components, named in non-italicised font in Figure 3. Each sub-component has a different set of tasks. The sub-components and their tasks are described as follows:

1. Experiment Receiver: A RESTful web service which provides a standard API to submit the experiment file/object to the JQueuer system via HTTP Request. When an experiment is received, the "Experiment Receiver" will generate an "Experiment ID" which will be used to identify this experiment in the system. The experiment sender will receive this ID as a HTTP response. Experiments to JQueuer are described in JSON format as illustrated in Figure 2 and explained in Section 4.
2. *Container Management Platform* Handler: This component offers communication with a managed container platform cluster (Swarm/Kubernetes/Mesos cluster). The Handler is responsible for ensuring that the container and virtual machine infrastructure is built, and for defining a set of scaling rules for that infrastructure. For the current set of experiments, JQueuer integrates with an external MiCADO platform, so the Handler accomplishes building the infrastructure by generating and submitting an ADT (see Section 3) through the MiCADO Submitter API.
3. Experiment Queue: A list of the experiment IDs which have been submitted. Each experiment has two important items in this queue: the Experiment Service Name and the Job Queue ID. JQueuer Agents use this list to recognize whether the containers running (on their virtual machines) should be controlled or not.
4. Job Parser: The Job Parser is responsible for extracting the job and task data from the *experiment.json* file and adding the jobs, and their tasks to a job queue dedicated to this experiment.
5. Job Queues: Each experiment depends on a dedicated Job Queue that has its own ID. The mechanism used to dispatch jobs from Job Queues is discussed in the next section.
6. Monitoring System: The Monitoring System contains the monitoring data related to all experiments. The monitoring data will be exposed as Prometheus metrics via a Prometheus exporter.

5.2 JQueuer Agent Design

An instance of JQueuer Agent component should be running on each virtual machine node in the managed container platform cluster. The JQueuer Agent is responsible for controlling the service containers of the experiments, fetching jobs from the Job Queues, monitoring the execution and sending data to the JQueuer Manager. From functional point of view, this component can be divided into sub-components, also shown in Figure 3, as follows:

1. Experiment Checker: This sub-component monitors the Experiment's Queue in the JQueuer Manager. When a new experiment is added, the Experiment Checker will fetch the Experiment Service ID and the Job Queue ID items.
2. Job Queue Fetcher: It uses the Job Queue ID which has been obtained from the Experiment Checker so as to fetch the jobs from an experiment job queue and execute them on the containers of the corresponding Experiment Service.
3. Monitoring Updater: It monitors job execution on local containers and sends data and statistics to the Monitoring System in the JQueuer Manager.

5.3 JQueuer Implementation

In this section, the technologies and tools that have been used in the first implementation of JQueuer are described. They are visualised in italics in Figure 3, next to their respective components discussed in sections 5.1 and 5.2 above. The aim was to reuse existing open source products as components, and as a result minimise development time and effort. The choice of technologies was a result of thorough investigation and comparison. However, due to limitations in length, this selection process is not detailed in this paper.

The two main components of the designed architecture, JQueuer Manager and JQueuer Agent have been developed using Python 3 and were prepared as Docker images. These components include Celery [13], an asynchronous distributed task/job queuing system that was used together with Rabbitmq [14], a message broker for job queuing, and Redis [15], an in-memory database for capturing results. Redis was also applied for experiment queuing and simplifying data exchange between the manager and the agents. Statsd [16] was selected for monitoring and exporting statistics and events of the JQueuer Agents as Prometheus metrics. We used the official Docker images of each of these components together with Docker compose, a tool for defining and running multi-container Docker applications in order to group all containers and simplify the deployment and communication among them. As input to JQueuer Manager, experiments are described in JSON format, as it was presented in Figure 2.

JQueuer Agent has two main components: Container Updater and Container Manager.

1. Container Updater (Experiment Checker): The main function of this subcomponent is to monitor the containers on the local virtual machine node to distinguish which containers belong to a particular experiment. Each Docker container shows in its information the name of its Docker Swarm Service. The Container Updater will check the container services against the list of experiments on the Redis server. If the container is new and it belongs to one of the experiments, a new Container Manager will be forked to manage this container and it will be added to the list of containers that this agent is responsible for.
2. Container Manager (Job Queue Fetcher): This component is responsible for managing and controlling an experiment container. The life cycle of a Container Manager starts when fetching a job from the Job Queue that corresponds to its container. It then executes any pre-job script in the container and goes through the list of tasks. Tasks are executed sequentially, and after finishing them successfully, the Container Manager will run any post-job script. It sends statistics to the StatsD server such as: job starting/finishing time and task starting/finishing time. If the job fails for any reason, it informs StatsD of the time spent before the job has failed, and it signals this failure to the Celery server. After finishing the job, it fetches another job and starts executing it. Container Manager continues working until the Job Queue of its experiment becomes empty.

Containers from different experiments can coexist on the same machine. JQueuer Agent will provide each Container Manager with a Container ID and a Job Queue ID. The Container List contains only those containers that belong to an experiment, and have been assigned to managers.

6. JQueuer integration with MiCADO

This section describes JQueuer's integration with a managed container platform, in our case with MiCADO. As a standalone queue, JQueuer relies on an external service to provision the infrastructure of virtual machines and containers where the experiment jobs can be executed. This service should also allow for the definition of policies or rules that control the scaling of the infrastructure at both virtual machine and container levels. MiCADO was selected as the managed container platform for this integration because it supports Docker Swarm and Kubernetes orchestration, offers automated cloud orchestration through Occopus, and provides a flexible approach to defining scaling rules. Integrating JQueuer with MiCADO enables the realisation of deadline based execution policies, and the managed execution of large number of jobs in various container-based cloud computing environments.

It should be noted that JQueuer does not depend on any cloud or container technology, and it is also agnostic to the cloud middleware and resources where the applications are executed. Therefore, it depends entirely on the managed container platform for which container technologies and which cloud resources can be used for executing the jobs queued by JQueuer. In the implemented solution, as MiCADO was applied as the managed container platform, containers are managed by Docker Swarm, and virtual machines can be instantiated on clouds supported by Occopus (i.e. CloudSigma, Amazon, OpenStack, OpenNebula or CloudBroker).

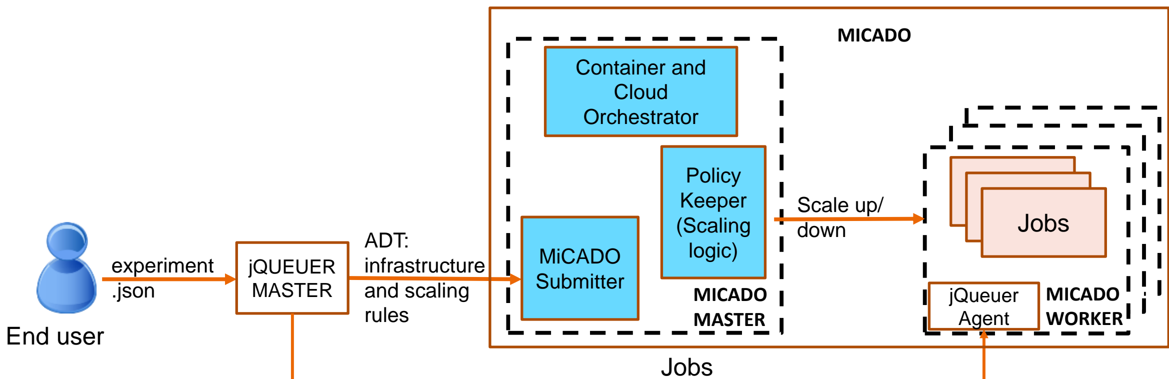

Figure 4 High-level architecture of integrating JQueuer as an external component to MiCADO

The high-level architecture of the integrated JQueuer/MiCADO solution is illustrated in Figure 4. As it can be seen in the figure, JQueuer is an external component to MiCADO that receives the Experiment.json file as input and (specifically for this integrated solution) generates the necessary ADT as input for MiCADO. This ADT describes both the necessary virtual machine and container infrastructures (including the dockerised version of the job executable, together with the JQueuer Agent), and also contains the auto-scaling rules. Based on the ADT, MiCADO deploys the worker nodes and containers with the JQueuer Agent deployed on them. This JQueuer Agent communicates with the JQueuer Master to receive the next job from the queue. Once the job is completed, JQueuer Agent asks for the next one until all jobs in the queue are finished. The number of MiCADO worker nodes and containers are managed by the MiCADO Master component based on the scaling rules and policies. The Policy Keeper of MiCADO, based on the scaling policy received from JQueuer, is responsible for deploying new workers or destroying existing ones (i.e. scaling up or down). Although this integration, especially the utilisation of the ADT and the MiCADO Policy Keeper is specific to MiCADO, it also has to be noted that similar managed container platforms can also be used in relation to JQueuer in order to achieve containerised execution and auto-scaling.

The more detailed JQueuer/MiCADO interactions are visualised in Figure 5. These interactions can be broadly divided into two separate tasks: (1) JQueuer submitting the experiment to MiCADO, and (2) MiCADO analysing JQueuer metrics to make scaling decisions.

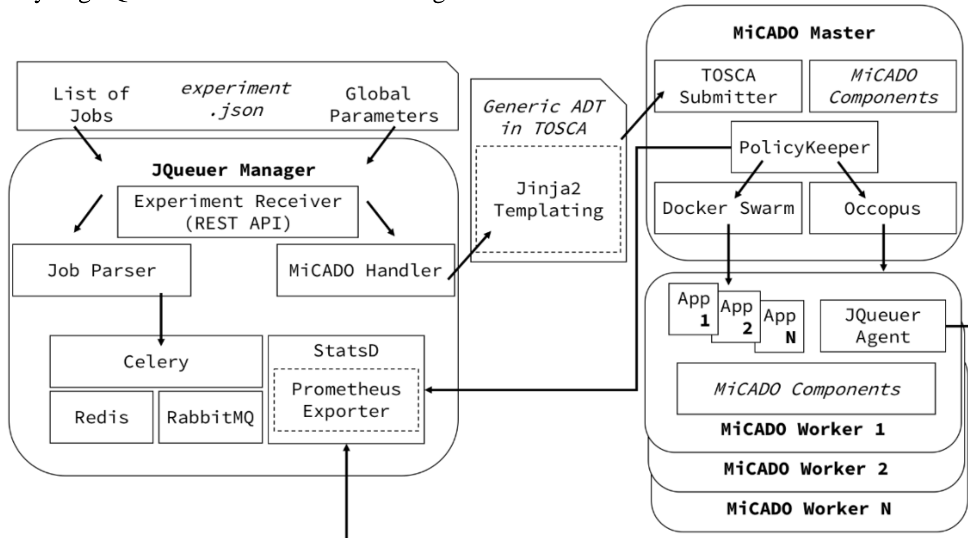

Figure 5 JQueuer integrating with MiCADO via the Container Management Platform (MiCADO) Handler

6.1 JQueuer Submitting the Experiment to MiCADO

The first task, where JQueuer submits an experiment to an external service – in this case MiCADO – begins with the JSON configuration file (*experiment.json* in the top left corner of Figure 5). This file is responsible for defining two subsets of information which are described in detail in Section 4:

a. the experiment jobs and tasks
b. the properties of the experiment to be launched, which include: container image, virtual machine image, scaling thresholds, and experiment deadlines.

This second subset contains the information required by the managed container platform to build the infrastructure and define the scaling rules for the experiment. As shown in Figure 5, in order to pass this data to MiCADO, it must first be converted into the TOSCA-compliant Application Description Template (ADT) format supported by the framework. To accomplish this, a generic ADT is written following the Jinja2 [17] templating language. The Jinja2 engine then generates an ADT specific to a given experiment by automatically filling key placeholders in the ADT with the corresponding information from JQueuer's *experiment.json* configuration file. The generated and now complete ADT is then submitted to the MiCADO framework via the TOSCASubmitter API, which then manages the creation of container and virtual machine infrastructure and attaches the unique scaling rules.

The actions of translation to the ADT format and submission to the MiCADO API are supported by the MiCADOHandler class on the JQueuer Master. The handler contains the generic template to be filled, instantiates the Jinja2 engine to generate the final ADT, and makes a POST request to the MiCADO API endpoint to submit the infrastructure and rules. The design supports agnosticism in JQueuer, as a handler can be written for any managed container service, as long as it can translate the *experiment.json* configuration into a compliant format, and send the generated data to the service in order to launch an infrastructure.

6.2 MiCADO analysing JQueuer metrics to make scaling decisions

The second task sees the Policy Keeper component in MiCADO managing virtual machine and container scaling within the infrastructure in order to complete the experiment by a set deadline. It does this based on a pre-defined set of scaling rules included in the generic ADT used in the previous task. These scaling rules, which define expressions and queries based on JQueuer-specific metrics, are combined with the scaling

thresholds, limits and deadlines defined by the user in *experiment.json* to build a complete scaling policy for the experiment.

Generally, the Policy Keeper of MiCADO provides a simplistic approach to building policies based on Prometheus metrics. Given that Prometheus can be extended with community-developed exporters to extract metrics from nearly any piece of software, it is an ideal candidate for a pluggable policy enforcer such as Policy Keeper. Policy Keeper can natively attach to any Prometheus exporter, either internal or external to the cluster it runs on. Extremely generic policies can then be abstracted to suit a wide variety of needs and use-cases. Creating the policies for JQueuer involves little more than taking a generic deadline-based, job-completion policy from Policy Keeper, and extending it with specific queries and expressions related to JQueuer.

Policy Keeper supports querying Prometheus to build expressions, alerts and constants that can then be used to express scaling logic using the Python scripting language. Conditional statements based on the queried metrics determine when the containers and nodes should scale up or down. The Python logic is written into the generic ADT, which, after Jinja2 templating, resolves as a complete and working scaling policy for the infrastructure of the current experiment.

The JQueuer-specific metrics are stored in the statsd server on the JQueuer Manager and are exposed via a Prometheus exporter built into the statsd container. The endpoint for this statsd server is fed into the ADT during Jinja2 templating, and on submission to MiCADO, Policy Keeper instructs Prometheus to connect to it. Once connected, the JQueuer-specific metrics become available to Policy Keeper and the expressions and queries defined in the ADT are resolved so that metric-based scaling can take place.

The generic scaling policy for a deadline-based job-completion is created thusly:

First, a set of constants are defined for the user-supplied *estimated average execution time, experiment deadline, maximum virtual machine workers, and maximum containers per worker*. Then, queries are fetched via Prometheus for *remaining time, jobs in queue, jobs completed, and calculated average execution time*.

Python is then used to express the following logic:

The user-supplied *estimated average execution time* is used to calculate the initial number of containers required to complete the total number of jobs in queue before the user-supplied *experiment deadline*. After having completed 5% of the jobs in queue, the queried *calculated average execution time* is used to calculate the new number of containers required to complete the experiment in time. This average is updated as new jobs are completed. Up-scaling of containers and nodes can occur at any time, however, to prevent constant scaling, down-scaling will only occur when the change in containers is greater than three, or the change in worker nodes is greater than one. The user-defined maximums ensure that the infrastructure does not scale out of bounds.

Although the above expressed logic implements a suitable deadline-based policy, a particular problem with down-scaling can happen if Docker Swarm, the container orchestrator component of MiCADO decides to kill a container that currently executes a job. In this case, significant execution time can be lost as the job will be killed and rescheduled to a different container. In order to prevent this, outside of the ADT, the following measures are taken to prevent a job-in-execution being killed during down-scaling:

The container image itself is edited so that PID 1 inside the container points to a shell script. This script points to the normal entry point of the container, but adds protection by using Linux *trap* [18] to catch any interrupt signal forwarded to the container. When Policy Keeper instructs Swarm to scale down the number of containers, it sends such a signal. On catching the interrupt signal, the shell script ensures that the job runs to completion before the container is killed. In order to avoid holding these resources infinitely, such as in a situation where a job hangs, container orchestrators offer a grace-period setting – the time to wait after having sent an interrupt signal (SIGINT), before sending a kill signal (SIGKILL). We override this value using the user-supplied estimated average execution time and, since a SIGKILL cannot be caught by trap, if the grace-period elapses, the container will be killed regardless of whether the job has completed, and consequently the killed job will be rescheduled.

7 Deadline-based execution of an agent-based simulation

Simulation is a technique commonly used in science and industry to study a variety of problems across a wide range of domains by building and experimenting with models under difficult conditions [27]. Agent-based Simulation (ABS) is a widely used type of simulation [28]. It has roots in complex systems, complex adaptive systems and artificial life. ABS allows modellers to represent loosely structured systems in terms of actors (or agents) and their interactions with each other and their environment. For example, ABS has been used to study social networks, healthcare, supply chains, economic growth, climate change, power distribution systems and physical activity. An ABS typically consists of a set of autonomous agents (with attributes that describe the state of the agent), a set of agent relationships (how each agent interacts with other agents and its environment) and the environment (the "world" in which the agents exist). The use of a single computer restricts these to being executed in sequence. As with other forms of simulation (e.g. discrete-event simulation), empirical studies also often require many experiments to be run (e.g. the simulation of a model with different sets of parameters). Further, as some models can be stochastic, replications need to be run to build confidence intervals. This can lead to extremely lengthy or prohibitive experimentation time, especially if the run time of a single simulation is large.

To investigate the performance of the deadline based execution approach presented earlier, we have used an agent-based simulation of an infection network [29]. The model illustrates how simulation can be used to study the changes in individual behaviour and the impact on the spread of a disease. It is written using REPAST, a widely used agent-based simulation tool [3]. The simulation consists of three types of agents that move in an environment and interact with each other, representing the susceptible, infected and recovered population.

The overall architecture diagram of the experiment is illustrated in Figure 6. As preparation for the experiment with MiCADO and JQueuer, the REPAST model was first compressed into a *model.tar* file and each set of parameters were saved in separate files. These input files were placed into an external file server (right hand side of Figure 6) that is also used to store the generated outputs. Each job in this experiment consists of a single task, with each task is a simulation run that does not require any pre-job or post-job tasks.

In the next step, the experiment's global and experiment definition related parameters were prepared and placed into the *experiment.json* file (left hand side of Figure 6). Some parameters, such as MiCADO endpoint, Docker image, specification of worker nodes (CPU and RAM), and the estimated duration of a simulation run (from previous experience) were fixed for all experiments. As REPAST is computationally demanding, it was only efficient to run a single container on a VM. Therefore, the number of containers per VM was also fixed and set to one. Some other parameters, including deadline and maximum number of VMs were variables that were changed in the different experiments to illustrate and test the auto-scaling capabilities of MiCADO. The input data to an experiment consists of the URL to the compressed model, the input parameters XML file, the URL to an external file (FTP) server where the input/output files are stored, the credentials to access the server, and the command for the execution of the application.

To compare our deadline-based scheduling approach to a fixed resource approach, we used an implementation of REPAST running on the CloudBroker (CB) Platform with fixed resource allocation [30]. For both experiments (MiCADO and CB) resources of the CloudSigma cloud [31] were used (2.2GHz CPU/2GB RAM instances). A test scenario of 200 REPAST infection model runs was executed for comparison. Generally, our approach was to run the test scenario using a fixed number of VMs via CloudBroker and then to use information from these runs to set the deadline to test MiCADO's performance with different upper limits of VMs. In the first set of experiments (Experiments 1-4 in Figure 7) we used two variants of job submission via CloudBroker (manual allocation and round robin scheduling) to reflect two general forms of fixed resource systems and to give context to MiCADO's performance (Experiments 1-2). The best of the average execution times was used to set the deadline for MiCADO. Two MiCADO experiments were then performed (Experiments 3-4) to study deadline behaviour against different upper limits of VMs. In the second set of experiments (experiments 5-6) we compared the fixed approach (this time just with round robin scheduling) on 5 instances with MiCADO under more relaxed resource constraints. Finally, we put MiCADO under real pressure. In Experiment 7 we set the maximum number of virtual machines in MiCADO to 10 but restricted the available resources from the cloud service provider's side in a way that

MiCADO could not launch more than 6 worker nodes (the cloud account was restricted in such way). In Experiment 8 we significantly increased the number of overall jobs to execute from 200 to 1,000 and set the deadline beyond 5 hours.

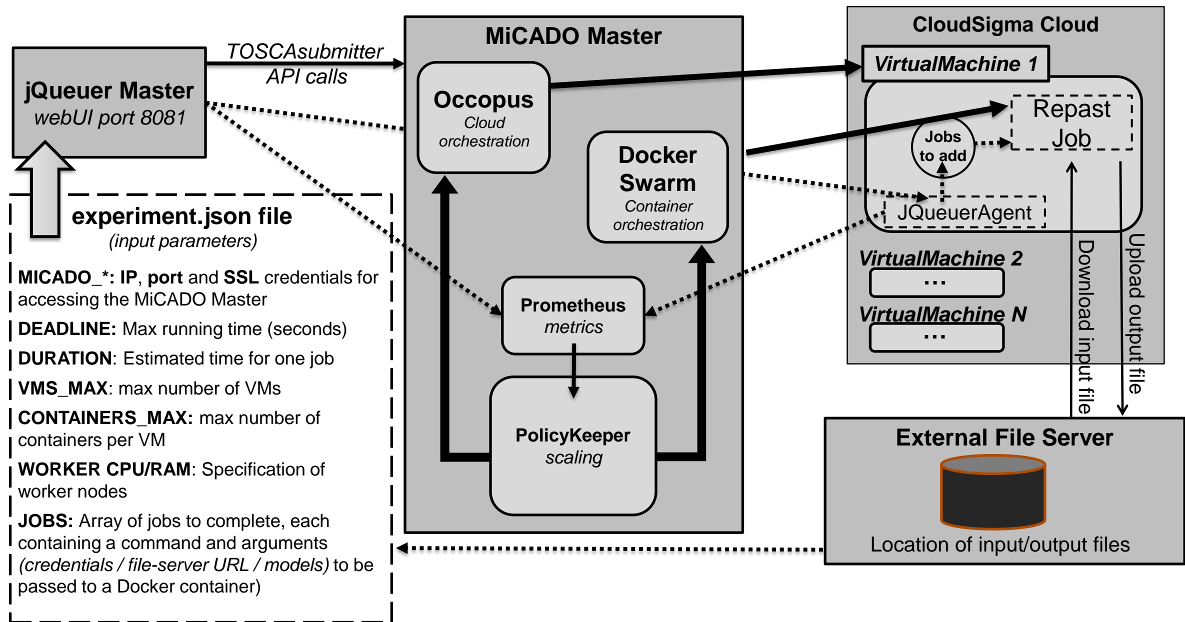

Figure 6 – Executing REPAST simulation jobs with MiCADO

Experiment	System	Description
1	CloudBroker	10 VMs allocate experiments to VMs manually (20 simulations per VM)
2	CloudBroker	10 VMs Round robin scheduling on demand
3	MiCADO	10 VMs max, deadline set by Experiment 1
4	MiCADO	15 VMs max, deadline set by Experiment 1
5	CloudBroker	5 VMs Round robin scheduling on demand
6	MiCADO	10 VMs max, deadline set by Experiment 5
7	MiCADO	10VMs max but account restricted to 6VMs, deadline set by Experiment 1
8	MiCADO	10VMs max, 1,000 jobs, deadline set by Experiment 5 * 5

Figure 7 – Description of experiments comparing fixed resource allocation with deadline-based scaling

When executing the experiments, five runs of both manual and round robin scheduling were performed with CloudBroker on 10 VMs first (Experiments 1-2). The best average performance (with rounding) was set as the deadline for MiCADO which was the result of the manual allocation and a time of 31minutes. In the next step, the same experiments were run on MiCADO (Experiments 3-4) with the above defined deadline, and setting the maximum number of instances to 10 and 15, respectively. Figure 8 illustrates the performance of MiCADO for each of the VM maximums (blue area) and compares these to the direct CB execution scenarios (red rectangular area). Please note that each experiment was repeated three times to check the relative consistency of the results. However, for illustration, only one of these runs is represented in the figures.

The figures demonstrate how MiCADO adjusted the number of VMs depending on the progress being made against the deadline. Both figures show how the number of VMs processing jobs changes. At the beginning of the experiments there is some notable and unavoidable overhead required to set up the necessary

infrastructure and start deploying VMs and containers. At the end of the experiments, once all jobs have finished, the virtual machines are shut down automatically and at the same time by MiCADO. The length of the experiment is measured until this automated shut-down. (Please note that in case of manual scheduling and execution, VMs needed to be shut down manually, putting significant burden on the operator to avoid unnecessary resource usage.)

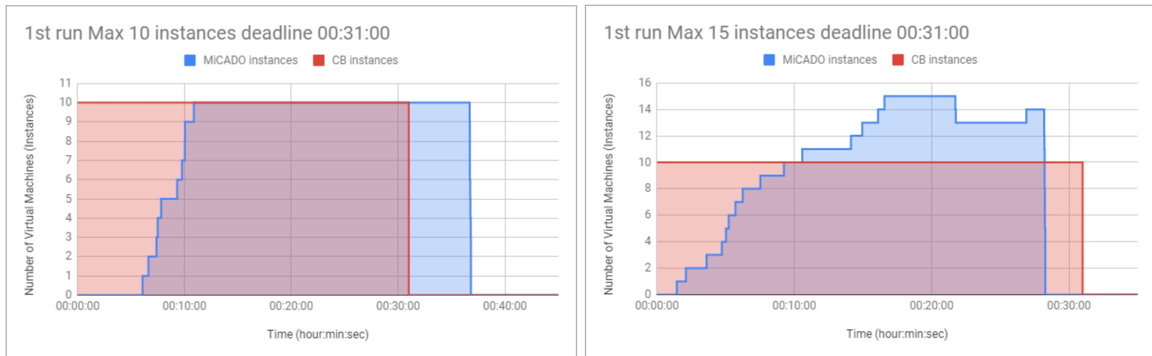

Figure 8 – Performance of automated scaling with maximum instances set to 10 and 15 and with deadline from batched job distribution (Experiments 3 and 4)

The adjustment of VMs in Experiment 3 simply means scaling up to the maximum number of VMs as MiCADO is stretched and using the maximum number of instances most of the time. In this scenario MiCADO was given the impossible task to match (or better) the performance of the optimised manual job distribution, using a priori knowledge about the jobs, that is not always available. In doing so, MiCADO scaled up the number of resources continuously until reaching the maximum, and then kept processing jobs using this maximum number of VMs. As it was expected, the deadline could not be reached as MiCADO needed some time to react and realise the need to scale up (causing some delay at the beginning of the experiment). On the other hand, MiCADO managed to almost match the best fixed resource performance and to process the same work in approximately the same time (with only 2-3 minutes delay). Additionally, although the deadline was missed, the utilisation of cloud resources is still slightly better than in the manual scenario, providing close to optimal resource utilisation. It was also noted, that MiCADO performed much better than the round robin scheduling that picked up and executed jobs individually on CloudBroker, and took almost twice as long as the optimised manual job distribution.

In Experiment 4 the scaling up and down is better visible and illustrates that MiCADO finished the jobs by the set deadline comfortably (as it can scale up to 15 VMs maximum). It can also be observed that in Experiment 4 MiCADO makes a pre-mature scaling down decision which it needs to adjust later on. Such, seemingly unnecessary decisions are justified by the fact that the actual job execution time is unpredictable and therefore shorter or longer jobs can turn up randomly any time. MiCADO's decision is based on the latest monitoring information and therefore on data that happened in the past. This also means that in case of "extreme" job distribution (e.g. if a very long job is turning up at the end of an experiment), the current MiCADO implementation can miss the deadline as it was not expecting this "out of character" behaviour based on previous monitoring information. The machine learning-based optimiser component of MiCADO that is currently under development is expected to improve this situation.

In all scenarios and runs, we have also calculated the average number of VMs that were used by MiCADO. This number varied between 7.87 to 9.44 in Experiment 3, and 8.48 and 9.24 in Experiment 4 which demonstrates that MiCADO used slightly lower number of instances on average than the fix allocation. This is due to the fact that not all jobs have the same execution time and this execution time cannot be predicted beforehand. In the manual allocation we simply allocated the same number of jobs per VM. However, when leaving the scheduling and auto-scaling to MiCADO, it has the ability to better optimise variable length jobs and allow the execution of variable number of jobs by workers that can result in a better overall execution time. As conclusion, it can be observed that although MiCADO has been put under real

pressure in this scenario as it was compared to an ideal, specifically generated and pre-batched job distribution (which in most real-life scenarios is not possible due to various length of simulation jobs), it was preforming reasonably well by using less resources than the manual allocation and completing the tasks by the deadline or with less than 10% extra time used.

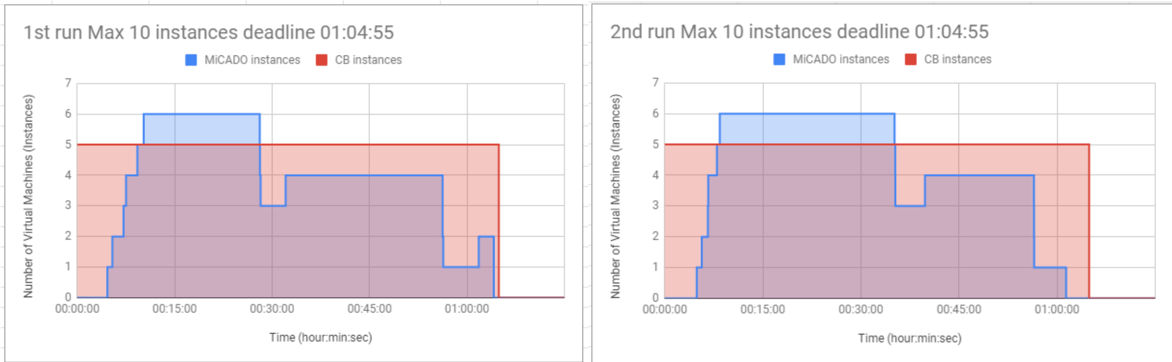

Figure 9: Performance of automated scaling with maximum instances set to 10 with relaxed deadline (Experiment 6)

In the next set of experiments (Experiments 5 and 6) a more relaxed scenario was prepared for MiCADO to better illustrate its auto-scaling capability. Five runs of round robin scheduling were performed with CloudBroker on 5 VMs. The average execution time from these experiments was set as the approximate deadline for MiCADO (1hour 4 minutes 55 seconds). In the MiCADO experiments the maximum number of VMs were set to 10 and the experiment was repeated three times. Figure 9 shows two of these runs as examples of the performance of MiCADO in this context.

In both examples, MiCADO ran faster than the best fixed resource runtime and finished the jobs by the deadline. In all runs MiCADO consumed a maximum of 6 VMs and, as shown in the figure, this again was "peak" consumption that only existed for a short time. Overall an average of 3.86 VMs were used, less than the 5 VMs of the fixed resource implementation (this again was possible due to the unpredictable and variable execution times of the individual jobs). Overall, MiCADO used less VMs to "beat" the deadline by a small margin. Additionally the figure also demonstrates that MiCADO scaled up differently in different runs based on the actual performance of the VMs that differ slightly in time when using CloudSigma. This unpredictable behaviour coupled with variable job execution time leads to some scaling up/down adjustments performed by MiCADO, as it is visible on both graphs.

Experiment 7 was designed to test how MiCADO behaves in an impossible scenario when the resources are restricted by the cloud service provider and not available in the expected volume. In this experiment the maximum number of resources to be utilised by MiCADO were set to 10. However, the user account on the CloudSigma cloud was restricted in a way that MiCADO could not launch more than 6 worker nodes at a time. The results of this experiment are shown in the left hand pane of Figure 10. As it is evident, MiCADO scales-up to the maximum of 6 workers. However, as it was expected, this amount of resources is not enough to complete the tasks by the set deadline which is missed by a large margin due to these physical constraints. Please note that MiCADO does not provide error message or warning to the user in such scenarios (a feature which can be implemented in the future). However, the scaling of the virtual machines can be observed on the MiCADO dashboard where the user can see the restricted scaling behaviour.

In Experiment 8 the aim was to execute a larger use-case scenario. Therefore, we increased the number of jobs to 1,000 and we primarily tested the scalability and robustness of our solution. As it is evidenced in the right hand pane of Figure 10, MiCADO coped well with the pressure and completed all 1,000 jobs by the deadline. It is important to note that in this case we have not actually performed the fixed resource experiment, only used the outcomes of Experiment 5 to estimate and set a reasonable deadline. As in the previous cases, multiple runs were performed, with the graph showing some of these as examples only.

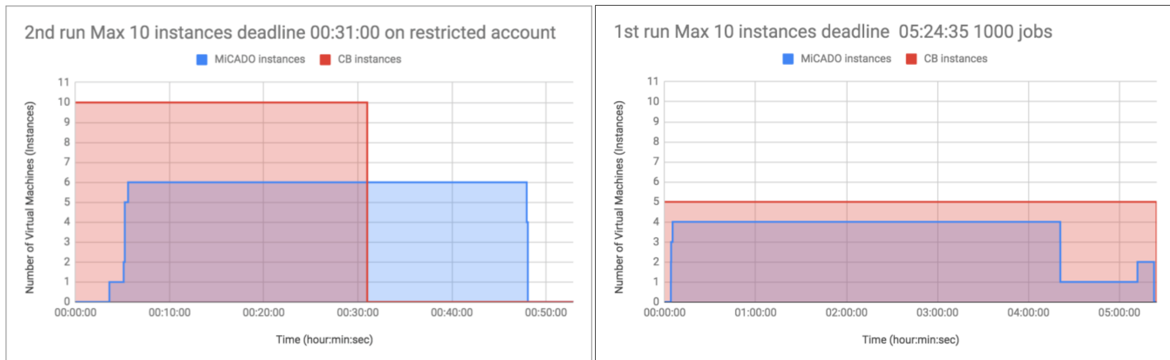

Figure 10: Automated scaling in resource constrained scenario (Experiment 7) and with 1,000 jobs (Experiment 8)

The above experiments demonstrate the performance of MiCADO for a relatively small set of experiments. Performance was comparable with a fixed resource implementation, even under challenging deadlines. Significantly, in this limited demonstration, the experiments show that less VM time was used in the MiCADO implementation than in the fixed resource ones. Arguably this shows that, if an appropriate deadline is set, simulation users can merely set the upper limit of VM expenditure and allow MiCADO to make efficient use of available cloud resources. However, further experimentation with a range of simulation models will be required to generalise this observation.

8 Conclusion and Future Work

This paper presented a set of technologies that enable the efficient deadline-based execution of large number of jobs in containerised cloud environments. As such experiments are typically computationally demanding and require access to distributed computing infrastructures, these building blocks can be efficiently used when implementing science gateways.

The presented components include JQueuer, a cloud-agnostic distributed system designed to support the scheduling of large number of jobs in containers and virtual machines. It was also demonstrated how JQueuer can be integrated with a managed container platform, such as MiCADO, to realise deadline-based scaling policies. Both JQueuer and MiCADO are open source and available at https://github.com/micado-scale. Finally, an agent-based simulation application implemented in REPAST was used to test the behaviour of the combined MiCADO JQueuer solution in various scenarios.

As REPAST is used widely, especially by the academic and research communities, a public gateway, based on the CloudSME AppCenter and its related technologies [4] is currently being set up where such simulations can be executed on cloud computing resources on a pay-as-you-go basis. Similarly to the REPAST solution, an open source discrete-event simulation software package, called JaamSim [32] is also being prototyped with the deadline-based scaling solution and will be offered as public service via the gateway. The results of this work impact both forms of simulation.

In parallel, experimentation for developing more efficient scaling policies and implementing the optimiser component of MiCADO using machine learning techniques is currently ongoing work in the COLA project. This work also incorporates the further investigation of cost models and how cost optimisation can also be considered besides deadline-based execution. Additionally, MiCADO is being extended with several new features and capabilities, for example to support multiple applications/experiments and multiple users under the same MiCADO installation.

Acknowledgements

This work was funded by the COLA Cloud Orchestration at the level of Applications Project No. 731574.

References

[1] S. Wu, C. Niu, J. Rao, H. Jin, and X. Dai, Container-based cloud platform for mobile computation offloading, in 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS), May 2017, pp. 123–132.

[2] T. Kiss, et al., MiCADO - Microservices-based Cloud Application-level Dynamic Orchestrator, Future Generation Computer Systems, 2017. https://doi.org/10.1016/j.future.2017.09.050
[3] M. J. North, et al., Complex adaptive systems modeling with repast simphony, Complex Adaptive Systems Modeling, vol. 1, no. 1, p. 3, Mar 2013. https://doi.org/10.1186/2194-3206-1-3
[4] SJE Taylor, et al., The CloudSME Simulation Platform and its Applications: A Generic Multi-cloud Platform for Developing and Executing Commercial Cloud-based Simulations, in Future Generation Computing Systems, Volume 88, November 2018, pp 524-539, https://doi.org/10.1016/j.future.2018.06.006
[5] COLA – Cloud Orchestration at the Level of Application, https://project-cola.eu/, accessed Online: 2019-01-25
[6] J. Kovacs, P. Kacsuk, Occopus: a Multi-Cloud Orchestrator to Deploy and Manage Complex Scientific Infrastructures, Journal of Grid Computing, vol 16, issue1, pp 19-37, 2018
[7] "Docker swarm," https://docs.docker.com/engine/swarm/, accessed Online: 2018-03-16.
[8] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, J. Wilkes. 2016. Borg, Omega, and Kubernetes. Queue 14, 1, Pages 10 (January 2016), 24 pages. DOI: https://doi.org/10.1145/2898442.2898444
[9] "Prometheus," https://prometheus.io/, accessed Online: 2018-03-16
[10] "OASIS Topology and Orchestration Specification for Cloud Applications Version 1.0" [Online]. Available: http://docs.oasis-open.org/tosca/TOSCA/v1.0/TOSCA-v1.0.html [Accessed: 29-Mar-2018].
[11] G. Pierantoni, T. Kiss, G. Gesmier, J. DesLauriers, G. Terstyanszky, JMM Rapún, Flexible Deployment of Social Media Analysis Tools, International Workshop on Science Gateways, 13-15 June 2018, Edinburgh, UK.
[12] "Introducing JSON", https://www.json.org/, accessed Online: 2019-01-17.
[13] "Celery," http://www.celeryproject.org/, accessed Online: 2018-03-16.
[14] "Rabbitmq," https://www.rabbitmq.com/, accessed Online: 2018-03-16.
[15] "Redis," https://redis.io/, accessed Online: 2018-03-16.
[16] "statsd", https://github.com/etsy/statsd, accessed Online: 2019-01-17.
[17] "Jinja2," http://jinja.pocoo.org/docs/2.10/, accessed Online: 2019-01-15.
[18] "trap man page", http://man7.org/linux/man-pages/man1/trap.1p.html, accessed Online: 2019-01-18
[19] "Apache mesos," http://mesos.apache.org/documentation/latest/frameworks/, accessed Online: 2018-03-16.
[20] "Apache Mesos framework," https://github.com/dcos/metronome, accessed Online: 2019-01-27
[21] D. Abdurachmanov, Optimizing CMS build infrastructure via Apache Mesos. Proceedings, 21st International Conference on Computing in High Energy and Nuclear Physics (CHEP 2015), April 13-17, 2015, Okinawa, Japan
[22] D. Thain, T. Tannenbaum, and M. Livny, Distributed computing in practice: the condor experience, Concurrency and Computation: Practice and Experience, vol. 17, no. 24, pp. 323–356, 2004.
[23] "Slurm Workload Manager, Version 18.08", https://slurm.schedmd.com/, accessed Online: 2019-01-31.
[24] R. Lovas, et al., Orchestrated Platform for Cyber-Physical Systems, *Complexity* 2018:1-16 (2018) https://doi.org/10.1155/2018/8281079.
[25] "What is AWS Batch?", https://docs.aws.amazon.com/batch/latest/userguide/what-is-batch.html, accessed Online: 2019-01-28
[26] "Batch," https://azure.microsoft.com/en-gb/services/batch/, accessed Online: 2019-01-28
[27] A.M. Law, Simulation modeling and analysis, 5th ed., McGraw-Hill, 2015
[28] Macal, C. M. and N. J. North, Tutorial on Agent-Based Modelling and Simulation. *Journal of Simulation* 4(3):151-162.
[29] SJE Taylor, et al., Open Science: Approaches and Benefits for Modeling & Simulation. In *Proceedings of the 2017 Winter Simulation Conference*, edited by W.K.V. Chan et al., 535-549. Piscataway, New Jersey: IEEE.
[30] SJE Taylor et al., Enabling Cloud-based Computational Fluid Dynamics with a Platform as a Service Solution, in IEEE Transactions on Industrial Informatics, DOI 10.1109/TII.2018.2849558, IEEE, Volume 15, Issue 1, pp 85-94, January 2019.
[31] Cloudsigma Holding AG. "Cloud servers & Hosting". [Online]. Available: https://www.cloudsigma.com/. [Accessed: 7 Feb 2019]
[32] DH King, HS Harrison, Open Source Simulation Software "JaamSim", Proceedings of the 2013 Winter Simulation Conference R. Pasupathy, S.-H. Kim, A. Tolk, R. Hill, and M. E. Kuhl, eds.

[33] M. Malawski, G. Juve, E. Deelman, and J. Nabrzyski, Algorithms for cost-and deadline-constrained provisioning for scientific workflow ensembles in IaaS clouds. Future Generation Computer Systems 48 (2015): 1-18.
[34] G. Le, K. Xu, and J. Song. Dynamic resource provisioning and scheduling with deadline constraint in elastic cloud. In 2013 International Conference on Service Sciences (ICSS), 113-117. IEEE, 2013.
[35] M. Mao, J. Li, and M. Humphrey, Cloud auto-scaling with deadline and budget constraints. In 2010 11th IEEE/ACM International Conference on Grid Computing, 41-48. IEEE, 2010.
[36] J. Shi, J. Luo, F. Dong, and J. Zhang. A budget and deadline aware scientific workflow resource provisioning and scheduling mechanism for cloud. In Proceedings of the 2014 IEEE 18th International Conference on Computer Supported Cooperative Work in Design (CSCWD), 672-677. IEEE, 2014.
[37] Saker Solutions, Company Website, [Online]. Available: https://www.sakersolutions.com/ [Accessed: 5 May 2019]
[38] DSS Consulting Company Website, [Online]. Available: http://www.dssconsulting.com/ [Accessed: 5 May 2019]
[39] CloudiFacturing – Cloudification of Production Engineering for Predictive Digital Manufacturing [Online]. Available: https://www.cloudifacturing.eu/ [Accessed: 5 May 2019]