Job-queuing and Auto-scaling in Container-based Cloud Environments

Osama ABU OUN
University of Westminster, London, UK
Email: o.abuoun@westminster.ac.uk

Tamas Kiss
University of Westminster, London, UK
Email: t.kiss@westminster.ac.uk

***Abstract*—Many applications process large quantities of data that takes significant time and requires big amount of computational resources. Optimising the execution of such applications in a cloud computing environment by keeping costs at minimum but still completing the task 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 technologies, such as Docker or Kubernetes provide limited support in this area. This paper presents *JQueuer* and *CAutoScaler*, a couple of cloud-independent solutions that offer job-queuing and automated scalability at the level of containers. Applying these solutions leads to more cloud-aware applications providing transparent auto-scaling for end-users and optimising execution time and costs. Business and science gateways will benefit from using an orchestrator combined with *JQueuer* and *CAutoScaler* since it will provide the layers needed to auto-scale the containers and to batch/sweep the jobs from a queue depending on a user-defined policy.**

***Keywords*—cloud computing, container technologies, Docker Swarm, JQueuer, autoscaler, MiCADO**

I. Introduction

Cloud computing offers scalable and on-demand access to large amount 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 containerized 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 function and do not need to store data in the containers. Web Service is an example of these *Stateless Services* where it is possible to create/clone several containers so as to process HTTP requests. Several types of *Batch Processing* applications in science and business gateways, for example discrete event simulation and 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 long 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 to collect the output after execution are not the only main differences between Stateless Services and Batch Processing applications. The policy used in scaling containers is another big difference. Scaling up/down the containers of stateless service may depend on load on these containers, CPU and memory consumption, number of requests, etc. In *Batch Processing* applications, we might need to take scaling decisions depending on different type of policies such as: the time needed to process a job or the deadline to finish all jobs in 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 of them are consuming 100% of their resources. The duration needed 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* differs completely from one job to another. For example, a queue of video files to be processed might contain one minute video lengths and one hour video lengths. This results in the need to periodically auto-scale up/down the containers after calculating the number of containers needed using a user-defined policy.

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 so as 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. Running applications and experiments on private clouds might be necessary due to confidentiality reasons. In this paper we propose *JQueuer* and *CAutoScaler* to address the above issues. The proposed solutions take a list of jobs and an auto-scaling policy, start the application containers in the cloud, dispatch the jobs to the containers, and auto-scale up or down the number of containers in order to finish the jobs according to the given policy. The

paper is organized as follows. Section II discusses the state of the art and related work. Section III explains the structure of an experiment which will be used as in our design. Sections IV and V provide the detailed design of the *JQueuer* and the *CAutoScaler*, respectively. Implementation, testing and results are described in sections VI and VII. Finally, Section VIII concludes this paper and presents future work and desired improvements.

II. State of the Art

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. We are going to compare job-queuing and auto-scaling in three widely used *Container Orchestration Engines*.

a. *Docker Swarm:* It is an orchestration tool which manages cluster of Docker Engines. 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 run so the service has the capacity it needs, and so on. Scaling a service changes the number of container instances running that piece of software, assigning more computing resources to the service in the process [2]. The containers in a service are stateless. Therefore it is not adequate for running jobs.

b. *Kubernetes:* It groups containers that make up an application into logical units for easy management and discovery. Kubernetes uses job concept and it provides the possibility to define a *parallelism* variable [3] which is equivalent to the number of container instances in Docker Swarm.

c. *Apache Mesos:* Apache Mesos is a project to manage computer clusters [4]. There are several software projects which have been built on Apache Mesos. Jenkins, the leading open-source automation server has a plug-in for Mesos in which it scales up/down the jobs according to the job queue size.

d. *AWS (Amazon Web Services) - EC2:* AWS provides the EC2 Container Service (ECS), which is a cluster manager but it is not possible to use ECS outside of AWS which prevents using these components in private clouds.

We can notice that *Docker Swarm* and *Kubernetes* do not provide job-queuing. Therefore, each application should implement its own queue agent so as to access an external queue in order to fetch the jobs. *Kubernetes* adds the parallelism variable statically in the job definition, while *Docker Swarm* gives the possibility to scale up/down the number of containers manually. *Apache Mesos* provides job-queuing through Jenkins plug-in but auto-scaling is achieved according to the job-queue itself and not according to a user-defined policy.

There are other job scheduling systems such as: SLURM and HTCondor. SLURM (Simple Linux Utility for Resource Management) is an open source, fault-tolerant, and highly scalable cluster management and job scheduling system for large and small Linux clusters. SLURM can be used as cluster manager and job scheduler but it is not compatible with windows operating system. Any job queuing solution should be platform-independent since Docker started to support windows operating system in order to run legacy scientific applications and tools. HTCondor is an open-source high-throughput computing software framework for coarse-grained distributed parallelization of computationally intensive tasks [5]. HTCondor has support for launching containers but it does not communicate with container orchestrators which prevents the applications from using services offered by them such as: networking between containers, docker compose, etc.

JQueuer and *CAutoScaler* are designed to be platform-independent and cloud-independent, in addition that they can work with any *Container Orchestration Engines*.

III. Experiment Structure

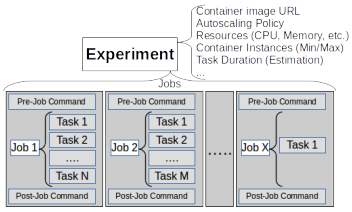

Fig. 1. Experiment Structure

In previously mentioned job submission type 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 it is illustrated in Fig. 1. These terms are described as follows:

A. Experiment

An experiment consists of two parts. The first part is a set of global parameters (upper part of Fig. 1), and the second part consists of a list of jobs (lower part of Fig. 1). Global parameters are, for example, the application container's image, the auto-scaling policy (e.g. the deadline by which the experiment should be finished), the minimum and maximum resources for each container, and an estimated length of each task. An experiment is considered to be "accomplished" when all the jobs are executed successfully.

Experiment's Description Language: An experiment (including jobs and tasks) might be stored using various formats, such as XML (Extensible Markup Language), JSON (JavaScript Object Notation), or YAML (Yet Another Markup Language).

There are two options so as to describe an auto-scaling policy. In the first option, the auto-scaling policy can be described using the same format that describes the experiment. In this case, the implementation of the auto-scaling policy is part of

the main system. The other option is to consider the auto-scaling policy as an external module/service (module-based policy) which could be called (along with the arguments) using a RESTful (Representational State Transfer) API. In this case the auto-scaler calculates the number of containers and sends it back to the system to take the appropriate action. In this second option, the experiment should have a field that contains the URL of the auto-scaling policy service. This option could be used in science gateways since it separates the development of the auto-scaling policies from the main system and gives the possibility to application developers so as to add their own policies.

B. 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, execute a system call, etc.

(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 failed for any reason, the whole job will be considered as "failed" and it will be re-queued or canceled, depending on the configuration of the system. Most of the time, 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 analyzed 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 to be "accomplished" when all its tasks are executed successfully.

C. 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 different sets of parameters that this command line requires.

IV. JQueuer Design

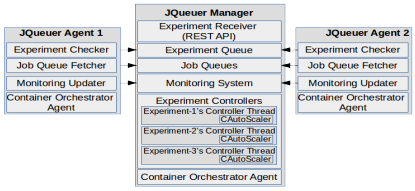

Fig. 2. JQueuer Structure

JQueuer is a queuing system that can be used in conjunction with container technologies to support the execution of a large number of jobs and the enforcement of certain up or down scaling policies. *JQueuer* is a distributed system that is composed of two independent components: *JQueuer Manager* and *JQueuer Agent* (Fig. 2). In the following, we are going to discuss the structure and the functionality of each of these. Communication methods between the *JQueuer Manager* and the *JQueuer Agent* were left out so as to be defined in the implementation according to the technologies used.

A. JQueuer Manager

JQueuer Manager is the main component of the *JQueuer* system which should be running on a *Container Orchestration Manager* (e.g. Docker Swarm) where it can control the containers and the services. The *JQueuer Manager* consists of several sub-components. Each sub-component has different sets of tasks. The sub-components and their tasks are described as follows:

1) Experiment Receiver: It is 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.

2) Experiment Queue: It is 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.

3) Experiment Controllers: It is an array of threads in which each thread controls a single experiment. A controller will be instantiated directly after receiving the experiment and it will keep running as long as the experiment is not accomplished. A controller is in charge of the following tasks:

a. Job Parsing: It parses the jobs from the experiment file/object and adds these jobs in a job queue dedicated to this experiment.
b. Monitoring (Data Analysis): It analyses the experiment execution data that was received from the *Monitoring*

Database. The resulted information will be used for deciding whether the system should scale up, scale down or continue with the current number of containers.

c. Decision of auto-scaling: This is done using the *CAutoScaler* component which calculates the number of containers needed to accomplish the experiment according to the auto-scaling policy. The *CAutoScaler* is discussed in detail in Section V.

4) Job Queues: Each experiment depends on a dedicated job queue which has its own ID. The mechanism used to dispatch jobs from job queues is discussed in Subsection IV-B.

5) Monitoring System: The monitoring system contains the monitoring data related to all experiments. The system will be accessed from the *Experiment Controllers* so as to gather the monitoring data related to their experiments.

6) Container Orchestrator Agent: The agent works as a bridge between the *Experiment Controllers* and the *Container Orchestrator Daemon* that is used in the cloud. The agent receives the commands from the controllers, forwards them to the Daemon and waits for the results. The supported commands by the agent are:

a. Create Service: This command is used to create the *Experiment's Service* which requires two parameters: container's image URL, and initial number of containers in the service. Before sending the command to the Daemon, the agent will form a third parameter (*Service Name*) using the image URL parameter. Service Name will be used as an ID during the execution.

b. Get Service's Status: The controller issues this command along with its *Service Name* in order to get the status of its *Experiment's Service*. The result consists of all data related to the service including the number of containers running under this service and their statuses.

c. Scale Up/Down: The parameters required for this command are: *Service Name* and *number of containers* needed to run under this service which is calculated by the *CAutoScaler*.

d. Destroy Service: This command will be issued by the controller when all its jobs are executed successfully.

B. JQueuer Agent

An instance of *JQueuer Agent* component should be running on each *Container Orchestration Node*. An instance should exist on the *Container Orchestration Manager* if an *Experiment Service* is running one or more of its containers on this Manager. The *JQueuer Agent* is responsible of controlling the services' 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 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*.

4) Container Orchestrator Agent: The agent acts as bridge between the *JQueuer Agent* and the Container Orchestration Daemon on the local machine. The first item is used by *Job Queue Fetcher* so as to recognize the containers that belong to a certain experiment.

V. CAutoScaler Design

CAutoScaler is the second main part in the system. It is responsible of taking and executing the decision of scaling up/down an experiment by calculating the number of containers which should be running according to the scaling policy. The *CAutoScaler* works as a sub-component of the *Experiment Controller*. As there is a controller for each experiment, these will all have their own *CAutoScaler* instances. The current design focuses on one single policy, the deadline policy, in which the experiment should be accomplished before a given deadline by using at least the minimum number of containers and without surpassing the maximum number of containers. Each container will be allocated with minimum number of resources (memory and processing). Please note that the system is not limited to this one policy, and it is very much possible to extend it to process different policies by implementing them in the *CAutoScaler*. The *CAutoScaler* has two main components: *Container Calculator* and *Container AutoScaler*. These two functionalities are explained in detail in the following two subsections.

A. Container Calculator

This functionality focuses on the process of calculating the number of containers needed at any moment. The process starts as soon as the experiment is received and it calculates the number of containers needed to finish the experiment according to the scaling policy. The algorithm applied to calculate the number of containers depends on the phase where the experiment is currently in. The first phase is called the *Initial Phase* which covers the period between receiving the experiment and the successful execution of the first job. The second phase is called the *Monitored Phase* which starts as soon as a job is successfully executed, and it finishes when the experiment is finished which means all jobs are executed. The main difference between the two phases is the way how the average execution time of a task is calculated. Since there is still no monitoring data in the *initial phase*, the calculation is based on the the estimation of duration needed to execute a single task as provided by the user. In the *Monitored Phase*, the calculation is based on the average of duration of all tasks (of one particular experiment) which have been executed so far. The number of containers ("containers count") will be equal to the multiple of the duration needed to execute a single

task, the average number of tasks per job, and by number of jobs in the queue, and the result is divided by remaining time until the deadline. The resulted value will also be checked against the minimum and maximum number of containers. At the beginning, the controller will start the *Experiment Service* using the "containers count" as an initial number of containers. When the *Experiment Service* is started and the number of containers which are in "running" state is greater than zero, the *Container AutoScaler* will process the number resulted from the *Container Calculator* and scale up/down the service as explained in Subsection V-B.

B. Container AutoScaler

The *Container AutoScaler* starts processing the number of containers needed (received from the *Container Calculator*) as soon as there is at least one container (of the Experiment Service) is in "running" state. The process uses *coherence index* so as to delay the scaling up/down in order to make sure that the number of containers needed is coherent with the last two results. The *coherence index* is increased by one every time the function receives a new calculation's suggestion. For example, if the last calculation suggested a scaling up while the current one is suggesting scaling down, the function will reset the counters and starts over without performing any scaling. If the last calculation and the current one are both suggesting the same thing (scaling up or scaling down), the function will take the lesser number of containers between the two calculation in the case of scaling up and the greater number in case of scaling down. The idea behind selecting the lesser and greater number is to scale up and down gradually as much as possible which gives the system the possibility to reevaluate the number of containers needed to finish the experiment.

In all cases, the function will not scale up or down until the *coherence index is at least three* which means the last three calculations are suggesting the same action.

VI. System Implementation

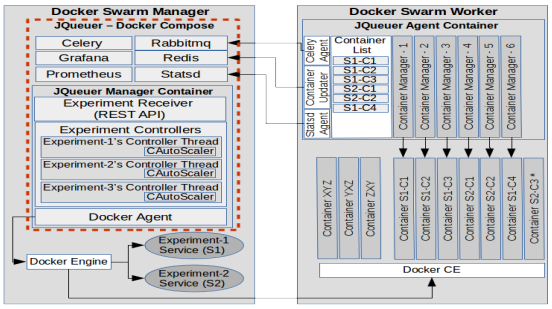

Fig. 3. System Implementation

In this section, technologies and tools which have been used in the first implementation of *JQueuer* and *CAutoScaler* (Fig. 3) are described. The aim was to reuse existing open source products as components, and as a result minimise development time and effort. Choice of technologies was a result of thorough investigation and comparison which due to limitations in length is not detailed here.

As Container Orchestrator, *Docker Swarm* [6] the native clustering tool for Docker containers was selected. 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. The Docker Swarm Manager node (left hand side of Fig. 3) hosts *JQueuer Manager*, with additional components for queuing, scheduling and data storage. These components include Celery [7], an asynchronous distributed task/job queuing system that was used together with Rabbitmq [8] (message broker) and Redis [9] (an in-memory database) for capturing results. Redis was also applied for experiment queuing, simplifying data exchange between the manager and the agents. For monitoring, Prometheus [10] was applied as core monitoring system, statsd for saving statistics and events of the *JQueuer Agents* so that they can be accessed from Prometheus, and Grafana as data visualization tool. We used the official docker images of each of these components together with Docker compose, a tool for defining and running multi-container docker application in order to group all containers and simplify the deployment and communication among them. The *Experiment* is described in JSON format. *JQueuer Agent* (right hand side of Fig. 3) has two main components: *Container Updater* and *Container Manager*.

a. *Container Updater:* The main function of this subcomponent is to monitor the containers on the local machine so as to distinguish containers which 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 experiment list in the Redis server. If the container is new and it belongs to one of the experiments, a new *Container Manager* will be forked so as to manage this container and it will be added to its Manager List.
b. *Container Manager:* It is responsible of managing and controlling an experiment container. The life cycle of a *Container Manager* starts by fetching a job from the job queue that corresponds to its container. It then executes the 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 the post-job script. It sends the statistics to the statsd server including: job starting/finishing time and task starting/finishing time. If the job failed for any reason, it sends the time spent before the job has failed to statsd, and it signals this failure to the Celery server. After finishing the job, it fetches another job and starts executing it. *Container Manager* keeps working until the job queue of this experiment becomes empty.

Containers from different experiments can coexist on the same machine, as it is shown in (Fig. 3). *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. That is the reason why we do not see

Container S2-C3 on the list as this should be added only when the *Container Updater* checks the containers in its next round. The containers *JQueuer Agent*, *XYZ*, *YXZ* and *ZXY* have been ignored since they do not belong to any experiment.

VII. Testing & Results

JQueuer was tested with Repast Simphony (RepastS) [12], an open source agent-based modeling and simulation system using a simplified infectious disease simulation model. Modeling and simulations are often based on scientific work and involve interdisciplinary research teams which can be supported using science gateways. Jobs were added to the experiment's JSON file along with the following parameters: RepastS Docker image URL, estimated execution time of a single task, minimum memory and CPU required by each RepastS container, minimum and maximum number of containers, and deadline. RepastS was deployed in a Docker container, also including a script that takes the HTTP URL of a simulation scenario and the FTP URL of the results server. The script fetches the simulation scenario from the HTTP server, processes it, and then transfers the output file to the FTP server.

Several scenarios have been tested with varying parameters. The number of jobs varied between 250 and 1000 jobs per scenario, the deadline was between one and two hours, and the maximum number of containers were between 10 and 20. The results shown below are for the following scenario: one task per job, 500 jobs, deadline is 90 minutes, estimated task duration is two minutes, the minimum number of containers is one while the maximum is ten, ten Docker Swarm Nodes. The duration statistics were as follows: Job Execution Duration (Min: 52 sec, Max: 80 sec, Avg:59 sec), Failure Duration (Min: 28 sec, Max: 47 sec, Avg:38 sec) and eight jobs have failed (during scaling down as their containers have been terminated), but these have all been resubmitted. Finally, Fig. 4 shows how the *CAutoScaler* was scaling up/down the containers according to the accomplishment of the experiment. The figure clearly indicates that the experiment started with the maximum number of containers based on the estimated execution time provided by the user. However, as the system realised that a smaller number of containers will also be enough to finish the experiment by the set deadline, it started to scale down. Fig. 4 also shows that the experiment finished exactly at the given deadline.

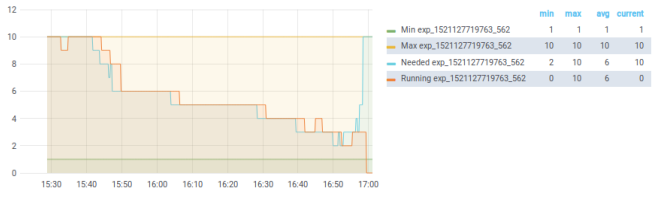

Fig. 4. Repast - Container AutoScaling

VIII. Conclusion and Future Work

This paper presented *JQueuer*, a distributed system designed to execute a large number of jobs in a cloud environment, taking into consideration a predefined set of policies and using restricted number of resources/containers. We explained the design and implementation of its two main components, *JQueuer* and *CAutoScaler*, that represent a unique extension to container-based technologies when handling queues and processing jobs. A beta version of the system was implemented to realise a deadline-based policy, and was tested on an agent-based simulation application in Docker Swarm.

The implementation of *JQueuer* is part of a larger project where an application level cloud orchestrator called MiCADO (Microservices-based Cloud Application Level Dynamic Orchestrator) [13] is being developed within the European COLA (Cloud Orchestration at the Level of Applications) project. The application-level orchestrator will be used as back-end in science or business gateways to provide automated scalability for a wide range of applications, including the submission of jobs. *JQueuer* will also be further developed to enhance scaling down so as to avoid terminating containers while they are executing jobs, and integrate it with the Policy Keeper of MiCADO in order to support a wider range of policies.

Acknowledgment

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

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] "Docker," https://www.docker.com/, accessed Online: 2018-03-16.
[3] "Kubernetes," https://kubernetes.io/docs/tasks/job/fine-parallel-processing-work-queue/, accessed Online: 2018-03-16.
[4] "Apache mesos," http://mesos.apache.org/documentation/latest/frameworks/, accessed Online: 2018-03-16.
[5] 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. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.938
[6] "Docker swarm," https://docs.docker.com/engine/swarm/, accessed Online: 2018-03-16.
[7] "Celery," http://www.celeryproject.org/, accessed Online: 2018-03-16.
[8] "Rabbitmq," https://www.rabbitmq.com/, accessed Online: 2018-03-16.
[9] "Redis," https://redis.io/, accessed Online: 2018-03-16.
[10] "Prometheus," https://prometheus.io/, accessed Online: 2018-03-16.
[11] "Grafana," https://grafana.com/, accessed Online: 2018-03-16.
[12] M. J. North, N. T. Collier, J. Ozik, E. R. Tatara, C. M. Macal, M. Bragen, and P. Sydelko, "Complex adaptive systems modeling with repast simphony," Complex Adaptive Systems Modeling, vol. 1, no. 1, p. 3, Mar 2013. [Online]. Available: https://doi.org/10.1186/2194-3206-1-3
[13] T. Kiss, P. Kacsuk, J. Kovacs, B. Rakoczi, A. Hajnal, A. Farkas, G. Gesmier, and G. Terstyanszky, "Micadomicroservice-based cloud application-level dynamic orchestrator," Future Generation Computer Systems, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X17310506