View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by ZENODO

Seamless Task Offloading on Multi-Clouds and Edge Resources: an Experiment

Andreas Tsagkaropoulos[1], Giannis Verginadis[1], Dimitris Apostolou[1,2], Gregoris Mentzas[1]
[1]*Information Management Unit (IMU), Institute of Communication and Computer Systems (ICCS), National Technical University of Athens (NTUA), Athens, Greece*
[2]*Department of Informatics, University of Piraeus, Piraeus, Greece*
atsagaropoulos@yahoo.gr, jvergi@mai.ntua.gr, dapost@unipi.gr, gmentzas@mail.ntua.gr

***Abstract*— Cloud computing has been growing at an increasing rate over the last few years. Commercial and scientific applications have come to rely on it as a development tool due to its exceptional characteristics in processing power and storage. The trend has been augmented with the emergence of the Internet of Things and smart processing devices at the edge. Contrary to the line of thought commonly adopted, we present in this work an alternative platform that considers edge devices as possible processing nodes, and provide a two-level task scheduling deployment that can handle not only binary modules, but also code-level fragmentations. We also go through a simple implementation of the platform, using production-ready solutions, while validating it on public and private clouds, and physically-separated edge devices.**

Keywords—Big Data Processing, Dynamic Resources Management, Edge Computing.

I. INTRODUCTION

The area of Cloud Computing, the practice of using a network of remote servers hosted on the Internet to store, manage, and process data, rather than a local server or a personal computer, has been growing increasingly over the last few years. Most research & industrial efforts have focused on the optimization of the deployment of centralized cloud environments. The results of these efforts have been the various implementations of Cloud Computing services by a number of Cloud Computing vendors (Amazon [1], Google[2], Microsoft [3] and others). All of these services emphasize the use of a strong storage and processing core to deliver processed data to the less capable devices of the users. However, the increasing need for supporting interaction between IoT and cloud computing systems has also led to the creation of the edge computing model, which aims to provide processing and storage capacity as an extension of available IoT devices, without needing to move data/processing to a central cloud data-center [4].

The ever increasing challenge for efficiently deploying and managing the big data applications, necessitates the extension of the “traditional” single provider cloud model by considering the exploitation of resources established either by several providers (i.e. multi-clouds) and/or resources situated at the extreme edge of the network. The active incorporation of edge devices into big-data processing topologies, along with the exploitation of multi-cloud resources, effectively delivers a joint and powerful processing platform, appropriate for coping with the increased needs of big data. The off/onloading of processing tasks over edge devices (e.g. mobile phones, routers, cameras, UAVs etc.) may either intelligently filter out the incoming (to cloud resources) high velocity / high volume data stream or efficiently process it by using closer to the edge cloud infrastructures without any restrictions on who may provide them (i.e. single/multiple providers, private/public resources). In our previous work [5], we discussed an envisioned dynamic, distributed, self-adaptive and proactively configurable architecture for processing Big Data streams. In particular, we aimed to combine real-time Big Data, mobile processing and cloud computing research in a unique way that entails multi-cloud resources use and extension of the fog computing paradigm to the extreme edge of the network. In this paper and based on the already introduced vision [5], we address the need for refactoring big data intensive applications so that their tasks can be distributed and processed over a joint processing platform composed of cloud nodes and edge devices, thus materializing an important aspect of our initial vision.

This paper is organized as follows: Chapter 1 contains a general introduction to the subject addressed; Chapter 2 presents available technologies for multi-clouds and edge processing that could be used as a joint processing platform while it sketches the approach for implementing it. Chapter 3 describes the details of the deployment and a small experiment conducted based on the concept. Chapter 4 highlights some of the challenges addressed in our concept that have not been solved by research until now, and Chapter 5 concludes, mentioning general directions for future research.

II. MULTI-CLOUDS AND EDGE AS A JOINT PROCESSING PLATFORM

In order to extend the traditional central cloud model with the active incorporation of the edge devices, we consider that Virtual Machines, emulations of computer systems, can be situated both within the premises of a cloud computing provider and on edge devices, for example, mobile phones, effectively delivering a joint processing platform. Data intensiveness is the characteristic of numerous research and commercial applications [6] that would be greatly benefited from such an extension of the traditional cloud approach. Applications such as social networks, scientific simulations, genome data analysis and fintech solutions are typically executed on powerful but costly distributed infrastructural resources.

We focus on technologies and on their potential integration as means to alleviate boundaries between available cloud provider offerings but also among resources at the extreme edge of the network.

One of the most widely used technologies providing cluster resource management is Mesos [7]. Mesos is a distributed systems kernel providing applications with API's for resource management and scheduling for clusters stretching across entire datacenter and cloud environments. This functionality is based on the Master-Slave model, or – using the terminology of Mesos – Master-Agent. Thus, an Agent instance is executed on every processing node, and communicates with the Master node(s) which are responsible for making resource offers to frameworks. A Mesos framework is analogous to an application over a Linux kernel; It leverages the functionality of the kernel to deliver higher-level applications to the user. Similarly, frameworks in Mesos use the resource offers produced by the master nodes, to execute applications. In order to leverage the resources management provided by Mesos, and bring task scheduling to the edge, we investigated the Aurora [8] Mesos framework. Aurora runs applications across a shared pool of machines and keeps them running. Its architectural elements are Worker nodes which run over Mesos Agents and carry out the actual processing, Coordinator Nodes which run over Mesos Master nodes and coordinate processing, and Client Nodes, which are not dependent on Mesos modules and submit jobs for execution. The result of the execution can be monitored through a comprehensive web interface.

There have been multiple approaches for task offloading from mobile devices to the cloud. Jade [9] offloads classes from an Android device for execution in other Android devices or the cloud. It considers the state of the application as well as the device to make informed decisions on offloading, thus optimizing energy usage.

Another approach is the Java Parallel Processing Framework, (JPPF) [10]. JPPF is a software solution enabling the use of grid computing in applications, demanding minimum code modification. It is used in our platform, because it can allocate tasks not only on any Java-capable machine, but also on Android edge nodes. Its architecture – similarly to Aurora – also follows the Client-Server and Master-Slave models (JPPF slaves are called 'nodes') [11]. A JPPF topology includes at least one driver, one node and one client. Each module can be run independently from the others, but all of them can also run on the same machine. The Client is responsible for submitting jobs for execution, which comprise of one or more tasks. The tasks of a job are scheduled for execution by the JPPF driver, to one or more JPPF nodes that execute them and return the results to the driver which delivers them back to the client. We present below an architecture that can be used to deploy cloud applications consisting of processing elements that can be executed in parallel. First, the main application structure is deployed, and then smaller processing tasks are created for every method contained in the source code that will be offloaded, and are grouped in a relevant job. These processing tasks can then be distributed not only on cloud nodes but also on edge devices. This can have a very positive impact on the type of data processed by the cloud nodes, as edge devices – being nearer to the source of information – can perform advanced filtering on the information to be processed. Besides, edge devices can manipulate or enhance raw data, before sending it for main processing tasks to cloud nodes.

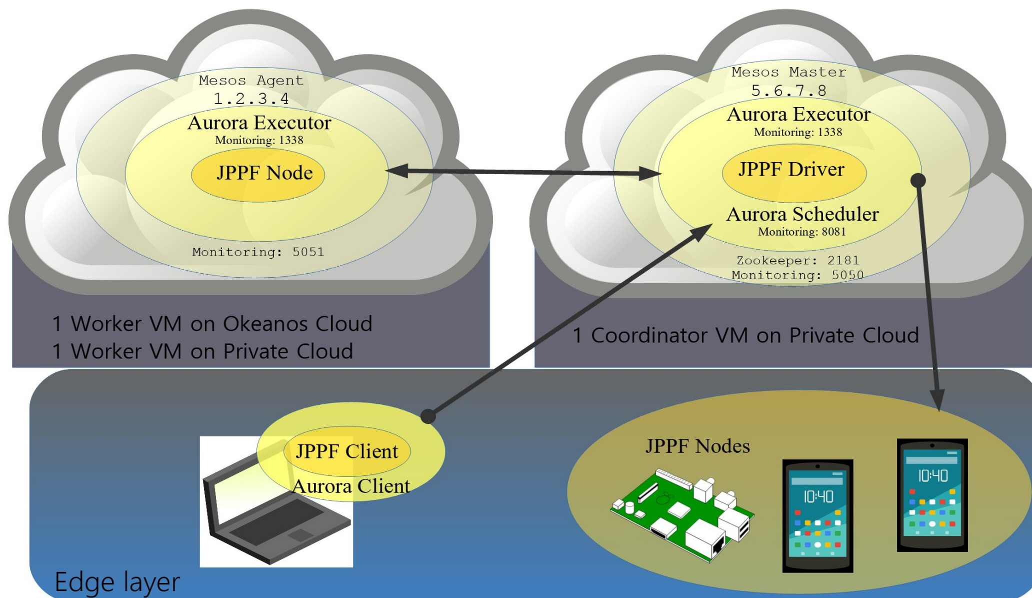

Fig 1. Experiment Architecture

Our approach presents a methodology for the distribution of tasks involving largely independent data streams, which do not need strict synchronization. This is realized by either rephrasing the standard processing flow when access to the source code of the application is available, or correctly tagging the binary modules of the application. Given an arbitrary topology of processing nodes, the application should be fragmented based on its needs and its structure for semantically-correct execution on remote nodes.

III. IMPLEMENTATION

To implement the fragmentation mechanism facilitating the distribution of processing tasks based on the Aurora framework, we need a number of bare metal or Virtual Machines with Linux-based OS's installed, capable of installing deb [12] or rpm [13] packages, as at the time of writing Mesos is unavailable on Windows [14]. Before the installation of Mesos on Ubuntu, a design decision concerning the number of Master nodes available to the topology must be taken. While one Zookeeper Server node [15] may be enough for testing purposes (and our small-level experiment), three or five nodes could be needed when hundreds of thousands of requests need to be serviced [16].

The installation of Mesos on Ubuntu is carried out following the instructions provided by the Aurora team [17]. A number of deb packages are downloaded depending on the role of each node in the processing topology, and then installed. Then the relative configuration files in the nodes and the Master are edited to reflect the current topology implementation.

Once the topology has been set up, the Client can start submitting processing jobs to the Scheduler for execution. The Scheduler deploys the incoming jobs to the Executor(s) satisfying the deployment constraints and the resource requirements described in the specification file. For example, if a node has a tag that designates it as 'low-network', the scheduler will avoid sending network tasks on that node and instead will send other types of tasks. Once a task is successfully initiated, it will continue to run indefinitely, until stopped from the Client or completing its processing [18]. While tasks are running, Aurora provides a web-browser accessible interface that allows monitoring the processing state, the resources it consumes, the standard output (stdout), and standard error (stderr) streams per Worker. Additional information concerning the internal state of the Masters and Agents is also provided by Mesos, using the relevant JSON endpoints.

To enable the offloading of processing to edge resources, and permit application developers to fine-tune the performance of the system at source code level, we used JPPF. The tasks submitted for execution on JPPF can be either a binary application of the OS, a script, or even a Java method / runnable [19]. This depicts the versatility of the framework to handle a wide range of programming components, which identifies it as a prominent candidate for Cloud Applications.

Thus, the JPPF node and driver modules should be downloaded in the processing nodes using Desktop operating systems or their derivatives (e.g Raspbian), and on Android devices (OS version 4.4 or higher). The details of the deployment (e.g. the location of the driver, the connection mode etc.) for cloud processing nodes are controlled by configuration files pertaining to each of the JPPF modules.

A. *Experiment Walkthrough*

The first step for deploying the implementation described above is to install the official packages mentioned in the installation instructions [17] or to re-compile them. We carried out the experiment on Ubuntu 14.04 and Ubuntu 16.04 machines running on a private and a public cloud, as well as on a number of Android devices.

We used 1 VM on the public Cloud Okeanos (used by the Greek Research and academic community)[20] and 2 VM's on our private cloud, running Openstack (Figure 1). Two Android Marshmallow devices used the JPPF Android node application, setting the number of processing threads equal to the CPU cores, from the application menu, and a Raspberry Pi 2 was also used.

Once Mesos and Aurora have been installed on each node, some changes must be performed on the initial configuration files so as to establish communications between Mesos nodes.

First, we changed the address of the Master (Mesos) node on all machines that will host Agent nodes by editing the domain name inside /etc/mesos/zk to match that of the Master. Then we created a file named 'attributes' inside the /etc/mesos-slave folder, containing a jppf-role:node or jppf-role:driver tag. This not only permits us to tag nodes so that only the appropriate tasks reach them, but also upgrades Mesos Agents to system services. Finally, the processing modules are started in the manner described in the Aurora installation guide.

B. *Aurora Application Deployment*

All platforms using Aurora, manage the execution of applications using configuration files, which describe the jobs and the tasks that will be executed in the topology. The Aurora Process object which will start a JPPF-node is the following:

```
start_driver = Process(
name = 'start_driver',
cmdline = 'cd /home/ubuntu/JPPF-5.2.3-driver/ && /home/ubuntu/JPPF-5.2.3-driver/startDriver.sh')
```

As the Process will be initiated on Linux JPPF nodes, special attention should be paid so that the execution path matches the deployment path of the JPPF modules.

The Aurora Task concerning starting the driver is the following:

```
start_driver_task = SequentialTask(
  processes = [start_driver],
  resources = Resources(cpu = 0.1, ram = 100*MB, disk=1*MB))
```

Finally the Aurora Job object containing the task is the following:

```
jobs = [
   Service(
       cluster = 'example',
       environment = 'devel',
       role = 'www-data',
       name = 'start_driver',
       task = start_driver_task,
       constraints = {
       'jppf-role': 'driver'
       },
```

```
    )
]
```

C. Creation and Deployment of JPPF tasks

To demonstrate the capability of our platform to fragment an application and schedule Java tasks for execution, a simple Java class with three time-consuming methods was created. Our workloads include a naive recursive Fibonacci number calculator [21], a sleeping process, and a math routine processing random numbers. The program, was tested on a local setup, and then was restructured to enable cloud execution, with the conversion of simple method calls to one job with the method calls as its tasks.

In JPPF, a new job (provided the required libraries are included) can be added in two lines of code:

```
JPPFJob job = new JPPFJob();
job.setName(“jobname”);
```

Similarly, a task can be added with the following code:

```
Task<?> task = job.add("name_of_method",new Java_Class_of_method(),method_parameter);
sleep_task.setId("My method 1");
```

The above snippet highlights the significant capability of JPPF to run unaltered source code, simply by using the name of the method to be run. The only pre-requisite to achieve distributed processing in this manner, is the writing of the job and task objects, and the management of the results. A very simple extensible application which can be used as a basis for more complex programs can be found at [22].

In order for tasks to be executed on Android nodes, it is necessary to include a dex file in the code containing all classes that have at least one task to be executed in the manner shown above, as Android nodes process their data offline [23]. Assuming that the required libraries are packaged in a file named ‘library.jar’, and that the Android SDK has been installed and a version of its build-tools (we used 23.0.0) is in the system path, executing the following command: dx --dex --output library-dex.jar library.jar it should provide us with the file necessary to include (library-dex.jar).

Then, For Linux/Windows processing nodes, the configuration file in the config/jppf-node.properties directory of the JPPF node should be adjusted with a text editor to turn off auto-discovery of the JPPF driver, and assign the IP address of the driver node. Android nodes on the other hand need to define the IP address of the JPPF driver in the relevant entry of the settings menu of the application.

Now, the execution of the Aurora job can be started in a terminal:

```
aurora job create example/www-data/devel/start_node jppf_simple.aurora
```

The result of following the presented course of action is that we should be able to see in the Aurora Scheduler dashboard and the Aurora Thermos dashboard the execution state of our driver and node jobs, along with the standard output and standard error streams. Now, the JPPF Application can be compiled and started. In Android, a mint JPPF Node application will show the number of tasks the device has processed and whether it is processing any tasks now.

Finally, depending on the code we have written to handle the processing of the results, we can get insights on how the tasks were processed..

IV. Related Work

The area of Application Refactoring for cloud deployment has been active, and a reasonable number of results have been published on the subject.

Vasconcelos et al. [24] presented a novel approach to support organizations in automatically adapting their existing software applications to the cloud. The approach is based on the loosely-coupled implementation of non-intrusive code transformations, called cloud detours, which enable the seamless replacement of local services used by an application with counterpart services in the cloud. A similar approach was followed by Kwon and Tilevich [25]. Our approach can immediately complement this strategy as it permits the migration of the complete application to the cloud (removing the need to maintain a client and a server for the application), while also granting the flexibility to execute certain methods of the source code at select nodes.

Akherfi et al. [26], reviewed Mobile Cloud Computing (MCC) offloading frameworks. Among the solutions reviewed, there was not one which could demonstrate on-demand source-code level task scheduling in multi-cloud deployments. The solutions reviewed, either operated on a higher abstraction level, or continuously used server resources to deliver the service. Although the platform proposed cannot offload mobile device tasks to the cloud (at the moment), it can consider edge resources in addition to multiple cloud environments for task allocation. Thus, it improves service integration in the cloud ecosystem, and shifts the emphasis from the mobile device to the enabling architecture.

Hilton et al. [27] developed Cloudifyer, a touchdevelop IDE plugin which refactors touchdevelop scripts in place. The emphasis of their approach was on enabling multi-user support through the usage of cloud types. Our approach encourages the deployment of an application as-is on the cloud (with Aurora) while also providing the means to optimize tasks placement and execution – if access to source code is available – with JPPF.

V. Conclusions

We presented an approach for implementing a cloud computing architecture benefiting from the processing power of edge devices. Our Mechanism has the advantages of the aggregation of resources by the Aurora platform, as well as the fine-grained adaptability offered by JPPF. Applications can either utilize the Aurora framework for quick execution, or can fine-tune their execution with method-level code refactoring.

Processing-heavy components (e.g. graphics processing, data mining, scientific modelling) can profit from direct

scheduling using the facilities provided by Aurora, while lighter components (light network communication, file access, raw data acquisition) can be scheduled for execution by edge nodes with the use of JPPF. This is not restrictive, but can be used as a guide for application deployment.

A limitation of the experiment conducted above is the manual creation of new tasks by the code developer. A more realistic approach is that tasks are being introduced, e.g. through polling of a relevant database. This could be implemented by the developer annotating the code, so that tasks can be created automatically, saved in a database and be recalled when needed.

We believe that future work in an annotations system, will permit a number of optimizations in the execution policies of Mesos and JPPF. Thus, tasks will be matched with processing nodes based on their processing difficulty, and the efficiency of the platform will be increased.

ACKNOWLEDGMENT

This work is partly funded by the European Commission project H2020 PrestoCloud - Proactive Cloud Resources Management at the Edge for Efficient Real-Time Big Data Processing (732339).

REFERENCES

[1] Amazon Web Services. Available online at: https://aws.amazon.com/ec2/

[2] Google Cloud. Available online at https://cloud.google.com/

[3] https://azure.microsoft.com/en-us/

[4] Villari, Massimo, et al. "Osmotic Computing: A New Paradigm for Edge/Cloud Integration." IEEE Cloud Computing 3.6 (2016): 76-83.

[5] Verginadis, G., Iyad Alshabani, Gregoris Mentzas, Nenad Stojanovic: PrEstoCloud: Proactive Cloud Resources Management at the Edge for Efficient Real-Time Big Data Processing. CLOSER 2017: 583-589.

[6] Chen, CL Philip, and Chun-Yang Zhang. "Data-intensive applications, challenges, techniques and technologies: A survey on Big Data." Information Sciences 275 (2014): 314-347.

[7] Mesos distributed kernel. Available online at: https://mesos.apache.org/

[8] Aurora Mesos framework. Available online at: https://aurora.apache.org/

[9] Qian, Hao, and Daniel Andresen. "Jade: Reducing energy consumption of android app." the International Journal of Networked and Distributed Computing (IJNDC), Atlantis press 3.3 (2015): 150-158.

[10] Java Parallel Processing framework. Available online at: http://jppf.org

[11] JPPF Overview. Available online at: http://jppf.org/doc/5.2/index.php?title=JPPF_Overview

[12] Debian Package extension. Available online at: https://www.debian.org/doc/manuals/debian-faq/ch-pkg_basics.en.html

[13] RPM package manager. Available online at: http://rpm.org/

[14] Mesos on Windows. Available online at: https://issues.apache.org/jira/browse/MESOS-3094

[15] Apache Zookeeper. Available online at: https://zookeeper.apache.org/

[16] Apache Zookeeper Performance. Available online at: https://zookeeper.apache.org/doc/trunk/zookeeperOver.html#Performance

[17] Apache Aurora Installation. Available online at: https://aurora.apache.org/documentation/latest/operations/installation/

[18] Aurora task lifecycle. Available online at: https://aurora.apache.org/documentation/latest/reference/task-lifecycle/

[19] JPPF Task objects. Available online at http://jppf.org/doc/5.2/index.php?title=Task_objects

[20] Okeanos Public Cloud. Available online at: https://okeanos.grnet.gr/home/

[21] An (inefficient) Fibonacci number calculation implementation. Available online at: http://introcs.cs.princeton.edu/java/23recursion/Fibonacci.java.html

[22] JPPF Tutorial. Available online at: http://jppf.org/doc/5.2/index.php?title=A_first_taste_of_JPPF

[23] JPPF Android node. Available online at: http://jppf.org/doc/5.2/index.php?title=Android_Node#Introduction

[24] Vasconcelos, Michel, Nabor C. Mendonça, and Paulo Henrique M. Maia. "Cloud detours: a non-intrusive approach for automatic software adaptation to the cloud." European Conference on Service-Oriented and Cloud Computing. Springer International Publishing, 2015.

[25] Kwon, Young-Woo, and Eli Tilevich. "Cloud refactoring: automated transitioning to cloud-based services." Automated Software Engineering 21.3 (2014): 345-372.

[26] Akherfi, Khadija, Micheal Gerndt, and Hamid Harroud. "Mobile cloud computing for computation offloading: Issues and challenges." Applied Computing and Informatics (2016).

[27] Hilton, Michael, et al. "Refactoring local to cloud data types for mobile apps." Proceedings of the 1st International Conference on Mobile Software Engineering and Systems. ACM, 2014.