Novel heterogeneous deployment of distributed applications across the compute continuum with COMPSs

Author: Unai Perez Mendizabal
Supervisor: Dr. Eduardo Quiñones
Dr. María A. Serrano

A thesis submitted in fulfillment of the requirements for the Master in Innovation and Research in Informatics, Data Science specialization
in the
Facultat d’Informàtica de Barcelona (FIB)
at Universitat Politècnica de Catalunya (UPC) - BarcelonaTech

June 30th, 2020
Declaration of Authorship

I, Unai Perez Mendizabal, declare that this thesis titled, “Novel heterogeneous deployment of distributed applications across the compute continuum with COMPSs” and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a master degree at the Universitat Politècnica de Catalunya (UPC) - BarcelonaTech.

- Where any part of this thesis has previously been submitted for a degree or any other qualification at Universitat Politècnica de Catalunya (UPC) - BarcelonaTech or any other institution, this has been clearly stated.

- Where I have consulted the published work of others, this is always clearly attributed.

- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.

- I have acknowledged all main sources of help.

- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: __________________________________________

Date: __________________________________________
Abstract

Industry 4.0 has caused a revolution in how data is created and analyzed. Great amounts of information are retrieved and generated non-stop by a myriad of sensors and small devices, which form the Internet of Things. For analytic workloads to keep up with the incoming stream of data, applications need to run in a distributed manner, and, as such, many frameworks for distributed applications have flourished in the recent years. These frameworks normally require the use of specific APIs that harm and prevent the code usability and portability. Additionally, those are usually aimed at cloud environments and demand powerful resources to run. The edge and fog computing paradigms have proven to be useful due to their closeness to the data, which can provide real-time guarantees that the cloud is not able to. None of these distributed application frameworks aim at providing a transparent deployment across the compute continuum, thus losing the benefits of a truly distributed workload. COMP Superscalar (COMPSs) is a distributed application framework that uses a very simple API that allows the execution of sequential programs in a distributed manner. Although one of the key benefits of COMPSs is its deployment transparency, it lacks the features necessary for a heterogeneous deployment across the compute continuum. In this thesis, new features for COMPSs are introduced, which extend its transparent deployment and code portability capabilities to edge and fog through containerization, by means of Docker and Linux Containers (LXC). The setup process is reduced to the possible minimum thanks to the new lightweight images. A new integration of cloud platforms has also been designed and implemented to support the most modern, container-oriented cloud platforms such as Docker Swarm or Kubernetes. This has allowed to transparently run distributed applications in heterogeneous continuum environments, facilitating the setup while not introducing any kind of latency to the execution.
# Contents

Declaration of Authorship i

Abstract ii

Contents iii

List of Figures v

List of Tables vi

## 1 Introduction

1.1 Motivation ........................................... 1

1.2 Objectives and goals ................................... 2

1.3 Structure of the thesis ............................... 3

## 2 State of the Art

2.1 Distributed computing platforms ......................... 4

2.1.1 Apache Hadoop .................................... 4

2.1.2 Apache Spark ...................................... 7

2.1.3 Other platforms ..................................... 8

2.2 Cloud computing ....................................... 8

2.2.1 Containerization: Docker and LXC .................. 9

2.2.2 Container orchestration: Kubernetes and Docker Swarm ... 12

2.3 Fog and edge computing ............................... 13

## 3 Background and Methodology

3.1 COMP Superscalar (COMPs) .............................. 16

3.1.1 Overview .......................................... 16

3.1.2 Deployment and resource management .................. 17

Computing Node workers ..................................... 20

Cloud Provider workers ....................................... 21

3.1.3 Docker Swarm support .............................. 22

3.2 Advancing the deployment capabilities of COMPSs .......... 24

## 4 New deployment features in COMPSs

4.1 New lightweight container images ....................... 26

4.1.1 Improved Docker images ........................... 27
5 Evaluation 39
5.1 Experimental setup: Applications 39
5.2 Heterogeneous and automatic deployment 40
5.2.1 Applications 40
5.2.2 Compute continuum 41
5.2.3 Summary of deployment improvements 43
5.3 Performance 44
5.3.1 Deployment on the edge 44
5.3.2 Deployment on the compute continuum (edge and cloud) 45

6 Conclusions 47
6.1 Impact 48
6.2 Future work 49

Bibliography 51
List of Figures

2.1   Graphical representation of the Apache YARN architecture.  . . . . . . .  5
2.2   Representation of the containerized infrastructure  . . . . . . . . . . .  10
2.3   Representation of the regular VM infrastructure  . . . . . . . . . . .  10
2.4   Diagram of the LoRaWAN architecture  . . . . . . . . . . . . . . . . .  15

3.1   Example of DAG  . . . . . . . . . . . . . . . . . . . . . . . . . . . .  17
3.2   Diagram representing regular ComputeNode initialization  . . . . . . .  20
3.3   Diagram for the CloudProvider deployment process  . . . . . . . . . .  21
3.4   Representation of current integration of COMPSs and Docker Swarm  .  24

5.1   Graphical representation of the experimental setup  . . . . . . . . . .  42
5.2   Performance on the edge of the containerized and monolithic execution 44
5.3   Performance on the compute continuum  . . . . . . . . . . . . . . . . .  45
List of Tables

5.1 Table comparing the sizes of images and traditional installations . . . 40
5.2 Table comparing the setup times of images and traditional monolithic installation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43
Chapter 1

Introduction

The growing popularization of the different new computing paradigms, especially the cloud and edge environments, has spawned a plethora of frameworks and platforms to facilitate the parallelization and distribution of workloads, and the correct exploitation of resources. The different computing resource pools are usually used and managed separately. However, current techniques are evolving with the appearance of the concept of compute continuum, which envisions all the available computing resources and pools as just one big accessible resource.

1.1 Motivation

The world is now in the height of the fourth industrial revolution, which is most widely known as Industry 4.0. Smart devices and environments have taken over the world and are producing data massively. The concept of Big Data is not new anymore. Along with Industry 4.0, different computing paradigms showed up, each of them offering diverse benefits and drawbacks to the applications and analytic workloads that run on top of them. These are the cloud, fog and edge computing paradigms. The cloud is composed of powerful computers, while small embedded devices and sensors populate the edge. However, the cloud resources are usually far from where the data, with a clear impact on the connection latency.

Containerization technologies have taken over the cloud and are the main means of application and service deployment. Containers are a kind of virtualization over the kernel that does not require of a guest operating system. They offer closed and isolated environments, along with networking options that can be customized. These technologies have great potential, as they facilitate the setup and deployment of many frameworks and applications, such as Hadoop or Spark, which will be discussed later on in this document.

Distributed analytic frameworks, that are able to take profit of data locality and distributed resources, are needed more than ever. Gaining a clear speed-up by distributing analytic workloads in either the cloud or in the edge has never been easier,
thanks to the distributed application frameworks that offer the infrastructure and API to make it happen. The main problem with distributed application frameworks is that they are, precisely, aimed towards either cloud or edge. Many research is focused on how to implement the efficient communication between both paradigms to connect analytic applications. However, no solution takes into consideration the distribution of applications across the whole spectrum of the compute continuum. On top of that, the portability of the code is non-existent. The applications are on purpose written for a given framework and the distribution relies completely on the provided API.

COMP Superscalar (COMPSs) [1] is a distributed application framework developed by the Barcelona Supercomputing Center (BSC) with the simplest of the APIs. Sequential Python/Java/C++ code can be converted into a distributed COMPSs application by annotating the functions on the code, which will be executed as distributed asynchronous tasks, guaranteeing uttermost code portability. The COMPSs runtime is in charge of executing the annotated code, by scheduling the tasks to the available resources, which also have the COMPSs runtime installed. Nevertheless, the current deployment options of COMPSs are rather limited, as it does not aim towards heterogeneous environments such as the edge and fog, and its support of cloud technologies is limited, not supporting containerization whatsoever.

1.2 Objectives and goals

This thesis tackles the problem of facilitating the distribution of applications across the compute continuum, i.e., transparently on cloud, edge and fog environments. In particular, the contributions of this thesis are as follows:

1. To enhance the deployment options of COMPSs through containerization technologies.

2. To facilitate the transparent deployment and distribution of sequential applications to edge and fog devices by minimizing the setup and making it dynamic and adaptive.

3. To facilitate the transparent deployment and distribution of sequential applications to cloud platforms by providing a common interface for it.

Overall, these three objectives translate into one big goal: To allow the transparent deployment of COMPSs applications across the compute continuum, permitting the seamless interoperability between its components and making the most out of each computing paradigm by using the most suitable containerization techniques for each environment.
Chapter 1. Introduction

1.3 Structure of the thesis

Chapter 2 will address the state of the art regarding frameworks for distributed applications and the latest technologies for the deployment of these across the different components of the compute continuum, this is, the cloud, fog, and edge environments. This chapter is followed by chapter 3, methodology and background, which will establish the context of the project. Specifically, the current state and features of COMPSs will be discussed, and how these can be improved thanks to the benefits that the technologies discussed throughout the state of the art can provide. Right afterwards, chapter 4 will explain the development process and address how these enhancements were made possible by discussing specific concepts of the implementation. Finally, chapters 5 and 6 will discuss the evaluation, which puts to the test the features implemented as part of this project, and discusses the conclusions drawn from it.
Chapter 2

State of the Art

The main points of study for the state of the art have been regarding the history and current condition of distributed applications and frameworks, giving special significance to the setup and deployment of the distributed environment and the reusability, simplicity and portability of the code throughout.

2.1 Distributed computing platforms

Nowadays, due to the rise of resource-heavy data analytics, the necessity and popularity of distributed software frameworks and languages have raised. Many applications move towards a service-oriented, distributed execution model. With websites, applications and services that move such great amounts of data as giants Google and Facebook do, distributing the workload is the only option. However, there have been multiple approaches to application distribution.

2.1.1 Apache Hadoop

The Apache Hadoop project [2] was the first general purpose tool used to distribute analytic workloads over resources connected through a shared network. To do so, Hadoop uses different components, like Hadoop YARN, its resource manager and scheduler, or Hadoop MapReduce [3], to process large data sets in parallel.

YARN [4] is the component that handles the resources and decides what jobs are executed in which resources. It follows a master/slave architecture. All the scheduling decisions are made by the Resource Manager, which acts as the master. Client applications make resource requests to the Resource Manager to run their workloads. Meanwhile, a Node Manager must be running on all slave nodes. This component takes care of monitoring the usage of local resources and lets the Resource Manager know if enough resources are available for new jobs. Figure 2.1 shows a graphical representation of the architecture. This framework requires a per-node setup, including extensive configuration tweaking to optimize the results, and the final deployment.
Chapter 2. State of the Art

Figure 2.1: Graphical representation of the Apache YARN architecture.

Algorithm 1 Word count implemented on the map-reduce paradigm

1: function MAP(word)
2:     return word, 1
3: end function
4: function REDUCE(key, values)
5:     return word, sum(values)
6: end function

The MapReduce framework is the core interface of the Hadoop ecosystem, with which the developer will be able to use the distributed computing capabilities of their deployed YARN cluster. Developing a MapReduce application consists of implementing the map and reduce functions in one of the supported languages, Java being the most commonly used one. The map function, as the name implies, must map instances of data to a key. The reduce function will receive a key and all elements mapped to it, and will have to return a single value for that key. Algorithm 1 showcases how to implement a word count following the map-reduce paradigm.

In the example, the map function receives a word as input, but it could be a collection of words or a text. The function must return the word as the key, and the amount of times it shows up in the input. In this case, as the input is just the word, a tuple with the word and the number 1 is returned. The reduce function receives a word as the key and all values that have been corresponded to it, which represent the amount of times it shows up. The sum of the values should be the total amount of appearances, which is then returned. Even though the theory behind the paradigm is really simple, the implementation of the library adds complexity into the already verbose Java syntax.

\[1\text{https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html}\]
Listing 1 Java MapReduce example implementing a word count

```java
public class WordCount {

    public static class Map extends MapReduceBase implements Mapper<LongWritable, Text, Text, IntWritable> {
        private final static IntWritable one = new IntWritable(1);

        @Override
        public void map(LongWritable key, Text value, OutputCollector<Text, IntWritable> outputCollector, Reporter reporter) throws IOException {
            StringTokenizer tokenizer = new StringTokenizer(value.toString());
            while (tokenizer.hasMoreTokens()) {
                outputCollector.collect(new Text(tokenizer.nextToken()), one);
            }
        }
    }

    public static class Reduce extends MapReduceBase implements Reducer<Text, IntWritable, Text, IntWritable> {

        @Override
        public void reduce(Text key, Iterator<IntWritable> values, OutputCollector<Text, IntWritable> outputCollector, Reporter reporter) throws IOException {
            int sum = 0;
            while (values.hasNext()) {
                sum += values.next().get();
            }
            outputCollector.collect(key, new IntWritable(sum));
        }
    }

    public static void main(String[] args) throws Exception {
        JobConf conf = new JobConf(WordCount.class);
        conf.setJobName("wordcount");
        conf.setOutputKeyClass(Text.class);
        conf.setOutputValueClass(IntWritable.class);
        conf.setMapperClass(Map.class);
        conf.setCombinerClass(Reduce.class);
        conf.setReducerClass(Reduce.class);
        conf.setInputFormat(TextInputFormat.class);
        conf.setOutputFormat(TextOutputFormat.class);
        FileInputFormat.setInputPaths(conf, new Path(args[0]));
        FileOutputFormat.setOutputPath(conf, new Path(args[1]));

        JobClient.runJob(conf);
    }
}
```

With the word count example, the specific problem is very appropriately resolved using a MapReduce technique. However, not all processing pipelines can be easily solved with the this paradigm. Indeed, the Hadoop ecosystem was populated with tools that allowed development of data analytic workloads without the need to adhere to this paradigm, while still using the MapReduce engine underneath. Such is the case for Apache Pig [5], which uses its own programming language, Pig Latin.
2.1.2 Apache Spark

Performing data analytics has gotten more and more complex with the time. This has caused the appearance of a new kind of data processing frameworks, the so called streaming processing frameworks, which process the data as it reaches the cluster. Frameworks that allow the processing of pre-loaded data are called batch processing frameworks. Apache Spark [6] has become the de-facto solution for distributed data analytics. It supports both stream and batch data processing: the former processes the data as soon as it reaches the cluster; the latter, the most popular of Spark uses, allows the processing of pre-loaded data.

The basic functioning of Apache Spark is similar to Hadoop, as it also follows a master/slave architecture. More specifically, the driver is the node in which the application is launched. The master is responsible for scheduling how the application will be distributed along the cluster, which is composed of executors. The master communicates through messages over the network with the executors, which are managed by cluster management platforms such as YARN.

The main contribution of Spark comes in how the data is handled. The concept of Resilient Distributed Dataset or RDD [7] is an efficient fault-tolerant abstraction for an element collection that can be operated in parallel. It is based on logging the transformations used to build a dataset, provided by means of coarse-grained functions, instead of logging the actual transformed data, which aims to avoid the expensive cost of copying great amounts of data over the network, potentially in a replicated manner.

The example source code in listing 2 showcases the usage of these coarse-grained functions from Spark’s Java API. The code implements a word count in Java, just like the code in listing 1. It is not using the streaming library, as it is reading the file directly from the file system. It is a visibly shorter code that takes profit of the function pipelining pattern and lambda functions, while also giving support to HDFS as the underlying distributed file system from which the data bulk is read. As opposed to MapReduce, Spark officially supports Python and Scala for application development.

```
Listing 2 Java Spark example implementing a word count
JavaRDD<String> textFile = sc.textFile("hdfs://...");
JavaPairRDD<String, Integer> counts = textFile
    .flatMap(s -> Arrays.asList(s.split(" ")).iterator())
    .mapToPair(word -> new Tuple2<>(word, 1))
    .reduceByKey((a, b) -> a + b);
```


2.1.3 Other platforms

Many projects have spawned that fill the same data processing market as Spark, some of which have fallen into Apache’s umbrella. Such is the case for Apache Storm [8] and Flink [9], two low-latency streaming processing frameworks. On paper, both fulfill the same function, but offer different programming APIs that might be more suitable for specific applications. One key difference between these processing frameworks and Spark Streaming (presented in section 2.1.2), is that the former are actual event-driven streaming processing framework and, thus, process data as it reaches the application. On the contrary, Spark Streaming receives data and packs it in micro-batches that are processed using the same underlying engine than that of the regular Spark applications. This achieves a greater throughput, gaining some latency on the way.

However, all of the aforementioned frameworks require the user to write framework-specific code, meaning the code is not reusable straight away. Apache Storm requires the user to write spouts that ingest the data and bolts that processing, forming an application called topology, whereas Apache Flink uses an API reminiscent to that of Spark. The exception to this rule is Apache Beam [10], which acts as an intermediary between the code and the underlying analytic runtime. This permits the code written by the user to transparently run on any of the different supported frameworks, among which Flink and Spark stand. Although an improvement, the code written for Beam is directly compatible only with Beam.

2.2 Cloud computing

The processing frameworks presented throughout section 2.1 oftentimes function as an ever-running service to which the workload can be submitted. That is exactly how Spark works: A cluster is previously setup, defining the roles of the different nodes, to choose from master and slave. In other words, these frameworks are deployed as multi-node infrastructures, where each node fulfills a specific role, aimed at guaranteeing safety against network, server or application failures. The nodes in these infrastructures need to be configured prior to usage. The difficulty of the setup and configuration phases will depend on the environment and the framework of choice.

Moreover, these frameworks not only require, but expect, multiple physical nodes and powerful resources, which might not be affordable for all users or companies. Having on-premise clusters usually incurs on high maintenance and electricity expenses, and having dedicated personnel. Cloud computing has served as a solution to this issue. This computing paradigm moves the heavy computation of those analytic workloads to the cloud, where computing power is lent by providers as a service in different ways, depending on the scope and requirements, and typically follows a pay-per-use model.
These *as-a-service* models, and are classified by the level of abstraction provided over the physical resource[11]:

- **Infrastructure-as-a-service (IaaS)**, provides a virtual machine with the requested operating system, while the user remains oblivious of the underlying technologies, such as the hypervisor or the orchestrator that guarantees the availability of the resource.

- **Platform-as-a-service (Paas)**, provides the basic infrastructure and dependencies to run the user’s application or service.

- **Software-as-a-service (SaaS)**, allows the user to directly use a software that is already hosted somewhere in the cloud.

- **Function-as-a-service (FaaS)**, allows to directly run functions on demand or on an event-driven fashion.

These models, in the order of mention, are noticeably layers of abstraction over the previous models, which makes easy to understand their success: They are able to provide the exact level of abstraction the user needs, while also being run in the cloud and saving the user the typical on-premise expenses.

The cloud used to utilize virtual machines as the base to offer their services; they could be created with a specific amount of resources bound to them, depending, again, on the requirements and the workload, or the budget. This is, it functioned in a similar way as IaaS, using orchestration software like Open Stack[12] to guarantee the availability of the resource, and specialized hypervisors to ensure stability. However, the container was the one concept that would bring the diversity and flexibility that characterizes the cloud computing nowadays.

### 2.2.1 Containerization: Docker and LXC

The containers are a form of virtualization at OS-level, which means they are ran just on top of the kernel and are able to use resources directly. In the contrary, regular virtual machines are not able to take as much profit from the underlying resources, because they require a guest operating system and an hypervisor, this is, the software that manage resource virtualization and virtual machines, which introduces considerable latency. The contrast between these two technologies is represented in the diagrams in figures 2.2 and 2.3. Both of these images were taken from Docker’s homepage.²

Containers are environments that are completely isolated from any external influence, unless explicitly specified. Containers achieved their current status of popularity halfway through the past decade, mainly due to the release and popularity

²[https://www.docker.com/resources/what-container](https://www.docker.com/resources/what-container)
of Docker [13]. Docker is the de-facto container engine and runtime that first implemented what is now officially considered the containerization standard, containerd. The most fundamental component of the engine is a container daemon, named containerd, with the function of abstracting system calls and OS specific functionality to run containers. containerd has since been detached off of Docker, and has been adopted by Kubernetes [14] and Kata Containers [15], among other modern container solutions. It serves as an industry-standard container runtime, aiming on simplicity, robustness, and portability.

One of the main contributions introduced by Docker is the concept of images, which work in a similar manner as the snapshots of regular virtual machines. Images behave like stills of the directory tree and environment, from which the container will be launched. Internally, images are comprehended by an ordered union of multiple layers. Each of these layers represent changes made to the said file system. Images can be created by writing a Dockerfile, a file describing the set of instructions to create the image from a base image, an example of which is shown in listing 3, or by committing an existing container. The ability of sharing images has permitted the most prominent of its uses; to encapsulate an application within a closed environment, ensuring its correct operation and stability. This way, developers are able to create and share the images of their software that can be deployed within seconds, with no additional setup and guarantee of a proper functioning. Images are shared through the use of private or public registries, being the DockerHub[16], the official public image repository hosted by Docker, the most popular one.

LXC (short for Linux Containers) is one of the main alternatives to Docker, developed by Canonical, the people behind Ubuntu. The main goal of LXC is to work as a lightweight and more efficient replacement to traditional virtual machines. Thus, LXC aims at providing a clean-slate environment for developers to work in, as opposed to Docker, which is oriented to aid in the deployment of applications. As a consequence, LXC images are not optimized for transmission and do not follow the
Chapter 2. State of the Art

Listing 3 Dockerfile example, which builds the image of a Java application

```bash
FROM ubuntu:18.04

ENV JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64

RUN apt update && \
    apt install -y openjdk-8-jre

ADD my_app.jar /root/

CMD ["java", ",-jar"]

ARG ["/root/my_app.jar"]
```

A layered pattern of Docker images. Finally, LXC allows the creation of unprivileged containers that can run without `sudo` privileges, which is not the case for Docker, making it unreliable for certain environments such as supercomputers.

As mentioned, containers are isolated environments. This means that the processes running within a container can not see nor affect the processes running in other containers or in the host\(^3\). As a direct consequence of this isolation, the file systems of host and containers can not directly interact. And on top of it, containers can not be reached over the network by default, unless explicitly specified. Docker’s networks allow containers to interact between them without being exposed to the outside world, whereas port forwarding binds a port inside the container to one outside the container, in a specific network interface of the host, so that other nodes over the network or the host itself can interact with the software within the containers.

Complex services and stacks of applications, like Wordpress or the aforementioned Hadoop ecosystem, can be set up using the Docker CLI\(^1\), and manually configuring each and all containers in place. However, Docker also introduced Docker Compose \(^1\), a tool that permits the deployment of a multi-container application or service through a YAML configuration file. This file can be used to extensively define the container deployment, which makes it useful for both simple and complex applications. Listing 4 shows an example of a Docker Compose YAML to deploy Wordpress.

Containers are stateless by definition, as they are created as blank slates from the image they are based on. If a container crashes, everything inside it is lost, potentially incurring in loss of important data. Volumes put a solution to this problem in local environments, by allowing to bind a specific directory from the host, or a arbitrary directory managed by the Docker daemon, to a directory in the container. However, platforms such as Kubernetes or Docker Swarm market themselves as frameworks for the management of stateful services such as databases, the kind of application suffering more from the stateless nature of containers, and put a solution to the mentioned issue through deployment automation and orchestration.

\(^3\)https://docs.docker.com/engine/security/security/
Chapter 2. State of the Art

Listing 4 Docker Compose file example for Wordpress

```yaml
version: '3.3'

services:
  db:
    image: mysql:5.7
    volumes:
      - db_data:/var/lib/mysql
    restart: always
    environment:
      MYSQL_ROOT_PASSWORD: wordpress
      MYSQL_DATABASE: wordpress
      MYSQL_USER: wordpress
      MYSQL_PASSWORD: wordpress

  wordpress:
    depends_on:
      - db
    image: wordpress:latest
    ports:
      - "8000:80"
    restart: always
    environment:
      WORDPRESS_DB_HOST: db:3306
      WORDPRESS_DB_USER: wordpress
      WORDPRESS_DB_PASSWORD: wordpress
      WORDPRESS_DB_NAME: wordpress
    volumes:
      db_data: {}
```

2.2.2 Container orchestration: Kubernetes and Docker Swarm

Even though Kubernetes [14] is the most popular container orchestration platform, Docker Swarm [19] is one of its main alternatives, marketed as the natural next step from Docker, with the same basic features as the former. Meanwhile, OpenShift [12] and Rancher [20] are two of the main commercial distributions of Kubernetes, meaning that, while the latter is a free, open source framework, the former are treated as products and usually offer more features and IT support behind a paywall. This section will only focus on the open source projects.

Docker Swarm, as mentioned earlier, is the next logical step if using Docker. It is publicized more as an additional mode of running Docker than a platform or framework of its own, and implements the master/slave architecture, dubbing the different components manager nodes and worker nodes.

Kubernetes follows a master/slave architecture, which can provide highly available master nodes. The master runs node monitoring and scheduling components, the API server through which the nodes and the `kubectl` CLI communicate with the master components, and a database to store configuration specifics. Meanwhile, the slave nodes run a proxy and the `kubelet`, the node agent that the master utilizes to manage the node and the containers it runs. Each node and master must be separately set up.
Chapter 2. State of the Art

One key difference in comparison with Kubernetes is the setup simplicity. Transitioning from many individual nodes with Docker installed to a whole cluster of Docker Swarm is a matter of running a command in each of the nodes, which will launch the Swarm mode, starting from the manager. Having a highly available master is not only supported, but also straight-away to deploy.

Both these platforms have in common that they permit to define, in a simple, human-readable way, the specification for how an application should be deployed, including the amount of replicas that each application needs, this is, the number of container copies. This specification can also be used to define how the platform should react to different events (see, a container crashing). All these operations can be done through an API, which can be communicated with in some markup language or text format, like YAML or JSON. Simpler configurations, though, are probably deemed to not use the features of these platforms to their full extent.

These platforms have traditionally been deployed in on-premise hardware. As a matter of fact, Hadoop was originally designed to run on commodity hardware, this is, in any available, cheap hardware. But as the use of cloud evolved into the as-a-service paradigms explained earlier, different ways of deploying and using these container orchestration platforms started showing up.

Services like Amazon Web Services (AWS) [21] or Microsoft’s Azure [22] offer a myriad of resources and utilities to make the most of their cloud. For example, Amazon’s Elastic Compute Cloud (EC2) provides users with virtual machines, i.e., a infrastructure as a service that can be used to deploy a user managed Kubernetes cluster. Or, on the contrary, Amazon’s Elastic Kubernetes Service (EKS) acts as a Kubernetes-as-a-service. At the same time, Kubernetes can be configured to be used as a FaaS (function-as-a-service) using frameworks such as OpenWhisk [23] or Knative [24], or Amazon’s Lambda [25]. This kind of service permits the user to create event-triggered functions, in a pay-per-use fashion and completely abstracted from the underlying infrastructure, from the user’s point of view.

2.3 Fog and edge computing

Cloud infrastructures certainly have evolved with the appearance of these platforms and service models, and, nowadays, they are the best solution for the deployment of specific application and workloads. However, along with Industry 4.0, new ways of monitoring and interacting with machines, devices and infrastructures have also been developed. Examples of these are smart cities or smart factories, which create vast amounts of data all the time that, in order to be analyzed, needs to be transmitted to the cloud.

Moreover, the analysis might be time-critical, for example, to foresee (and be able to prevent) a grave malfunctioning in a machine that could harm employees.
Chapter 2. State of the Art

The latency of sending streaming data to the cloud is high. Thus, the edge and fog computing paradigms offer a compromise between the high performance of the cloud and low latency, by running part of the analysis on site, close to where the data is actually generated. The phenomenon of creating smart environments through the deployment of small devices and sensor is widely called Internet of Things (IoT).

The edge and fog computing paradigms are primarily composed by devices and sensors of a smaller scale, such as small computers like the Raspberry Pi or embedded computing boards like the Nvidia Jetson series. These are usually equipped with not as powerful, but more energy-efficient ARM processors. These devices usually use batteries and are potentially deployed in remote locations, meaning the trickiest part of these computing paradigms tends to be the maintenance, for both hardware and software. The devices must be installed such that remotely accessing them is possible, via protocols such as SSH or VNC. Additionally, the software must be specifically compiled for the processor of the device, making the development more difficult. Finally, these devices need to be monitored, to make sure that no unexpected issues happen and that the connectivity, which is a crucial for the usage of these devices, is not lost.

Due to the growing popularity of these paradigms and the devices used for this purpose, some communication protocols and frameworks build on top of them have also gotten popular. LoRa (Long Range) is a physical layer protocol, focused on energy efficiency and, as its name implies, long range connections. Thus, it is a perfect fit for IoT networks, populated with less powerful, remote devices, to communicate efficiently and from great distances. Then again, LoRaWAN is an architecture specification, oriented towards IoT environments that facilitates the deployment and execution of analytic applications, pictured in diagram 2.4, based on the use of LoRa for data retrieval. The Things Stack and ChirpStack are the two most popular commercial implementations of this architecture.

Due to the growing popularity of these paradigms and the devices used for this purpose, containers and container management platforms have also found their way into the IoT market. Although Docker currently supports ARM processors, container images, at least in Docker’s case, still need to be created specifically for each target processor architecture. But, once this issue is dealt with, the advantages that containers and management systems can provide to edge and fog computing, in terms of deployment, service safety and monitoring, are significant.

As such, new distributions of Kubernetes, aimed at the IoT fog and edge environments, have showed up with the time [26]. Such is the case for K3S, KubeEdge or MicroK8S. All of them market themselves as lightweight, IoT oriented Kubernetes implementations, that offer a subset of the basic features as the original platform, meaning these platforms enable the deployment and orchestration of containerized

4https://medium.com/pruebas-de-laboratorio-de-la-modulaci%C3%B3n-lora/lorawan-d00f48384160
services and applications as if it was the cloud. This way, all the as-a-service offerings that the Kubernetes-like platforms allow are made possible on the edge and fog layers.
Chapter 3

Background and Methodology

3.1 COMP Superscalar (COMPs)

COMPSs is a framework for distributed applications that heavily aims on the programmability and code portability of the applications it runs. Considering that one of the main goals of this project is to facilitate a transparent deployment and usage of resources over the compute continuum, using a framework that already guarantees resource utilization transparency to some degree as the base is only a natural step. The following sections will introduce COMPSs and its basic functioning.

3.1.1 Overview

COMPSs is a framework with the objective of easing the execution of sequential Java, Python or C/C++ applications, potentially parallelizable, in a distributed manner. It achieves this by offering a simple task-based programming model through means of its API. With this model, the user is only responsible for identifying the functions that will be executed in a distributed manner by annotating them, using Java annotations or Python decorators.

COMPSs implements a master/worker paradigm. The role of the COMPSs master is played by the node where the user launches the application. The master takes care of running the COMPSs runtime, which is in charge of remotely deploying the COMPSs workers in the available resources and handling the execution of the application.

When annotated code is executed using COMPSs, the runtime reads the code to find calls to those functions annotated as tasks. These functions are, afterwards, run as asynchronous parallel tasks, in the different available resources. The accesses to the data from the tasks within the main code are also read. This way, the runtime is able to find out what tasks depend on the output of which tasks. These are called data dependencies. When tasks that depend on one another are scheduled and run in different resources, a data transfer must occur, in which the depending worker will directly receive the needed data from the worker that has generated it. The runtime
is responsible for all the scheduling, management and coordination of the resources
and their dependencies on data.

The code in listing 5 shows an example of an annotated Python program. Using
the mentioned information about tasks and data dependency, the COMPSs runtime
builds a Directed Acyclic Graph or DAG, with which it represents the task executions
and data transfers between them. The example in figure 3.1 represents the DAG of
the code in listing 5. Nodes represent tasks, whereas edges are the representation of
the data dependencies between tasks. If any given task or node is run in a different
resource than the one that is pointing at it, there must be a data transfer.

3.1.2 Deployment and resource management

COMPSs supports a handful of different worker resource types, ranging from
regular computing nodes such as laptops, to clusters and supercomputers. The most
used ones, and also most relevant for this project, are the compute node and the cloud
provider. On the one hand, the former represents a regular bare-metal or virtual node.
These nodes need to already have both COMPSs and the application files installed on
them, and they are accessed by the runtime on execution start-up using passwordless
SSH. On the other hand, the latter represents a request for virtual machines to a
supported cloud provider through its API, as the name implies. Examples of supported
Chapter 3. Background and Methodology

Listing 6 Example for resources.xml showing dependency declaration

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ResourcesList>
  <ComputeNode Name="host_with_opencv">
    <Processor Name="P1">
      <ComputingUnits>4</ComputingUnits>
    </Processor>
    <Adaptors>
      <Adaptor Name="es.bsc.compss.nio.master.NIOAdaptor">
        <SubmissionSystem>
          <Interactive/>
        </SubmissionSystem>
        <Ports>
          <MinPort>43001</MinPort>
          <MaxPort>43002</MaxPort>
        </Ports>
      </Adaptor>
    </Adaptors>
    <Software>
      <Application>opencv</Application>
    </Software>
  </ComputeNode>
</ResourcesList>
```

Listing 7 Example of PyCOMPSs task annotated with a constraint

```python
@constraint(AppSoftware="opencv")
def execute_trackers(*args):
    import cv2 as cv
    pass
```

cloud providers are Google Compute Engine or Slurm. These will be discussed more in depth later on in this section.

Internally, COMPSs uses a class by the name of WorkerStarter, instantiated once per ComputeNode, that takes care of establishing a passwordless SSH connection with the resource, based on the provided settings, and remotely executes a script. This script will launch a COMPSs worker in the remote resource and return the port to which the worker will listen. After this process, the COMPSs runtime is able to associate an IP and port to each ComputeNode, and start communicating with it, using the communication adaptor of choice. COMPSs adaptors are the implementations of communication protocols that COMPSs uses for the communications between master and worker, and among workers. The default and most used adaptor is the NIO adaptor, which communicates using TCP through the NIO (non-blocking input/output) libraries bundled with Java, from version 8 on.

The resources are listed and configured through two XML configuration files, the resources.xml and project.xml files. They serve similar but different purposes. The resources.xml file serves as a list of all configured and available workers in the environment, whereas the project.xml represents the subset of resources to be used for one specific application. The resources are represented by a tag, corresponding to its type and a unique name, i.e., `<ComputeNode Name="NodeName">`, `<CloudProvider Name="CloudName">`. The name is just an identifier in the case of CloudProvider
resources, whereas it must be the hostname or IP of the ComputeNode resource.

Within this tag, sub-tags are used to specify the configuration of the corresponding resource. In the resources.xml file, for example, the description of the hardware specification of the resource and the available software for execution customization can be specified. In the case of the project.xml file, this includes the username to be used for SSH connection establishment or the customization of application paths (ClassPath for Java, Pythonpath for Python).

These files also allow the user to define which software or libraries are available in each of the resources specified in the resources.xml file. Listing 6 shows an example on how to fill these fields. With this setting, the corresponding tasks of both the Java and the Python application can be annotated with the @Constraint annotation or decorator, respectively. An example of this is shown in listing 7. Upon reading this task, COMPSs will only run it on resources specified to fulfill the constraint, like the host_with_opencv from listing 6.

COMPSs applications are launched using the runcompss command, bundled with the COMPSs runtime when it is installed. When the application is launched, both configuration files are read and, specifically, workers are deployed and started in the resources according to the project.xml file.

At the end, both resource types work in the same basic way. A resource exists or will be deployed, in which a COMPSs worker will be launched. The initialization phase differs but, after initializing, the worker will need to be listening to a port, in the same specific IP or hostname that was originally defined in both XML configuration files. The COMPSs runtime will use this IP and port to communicate with the remote worker with the communication adaptor of choice. If the communication initiated by the adaptor does not succeed, the worker initialization will be considered failed,
and the worker discarded for the execution. Thus, the application will execute with whichever workers are confirmed to have been initialized correctly.

**Computing Node workers**

The runtime uses the IP or hostname of the XML files to establish the SSH connection with the ComputeNode workers. Regarding the username, it follows the regular functioning of SSH, where, if no user is specified through the project.xml file, the current username will be used. In any case, the used username must be able to establish a passwordless SSH to the resource, by having its public key added to the remote user’s list of authorized keys. Once the connection is correctly established, the COMPSs worker is launched, according to the configuration of the XML files. An example of ComputeNode worker configuration is shown in 8 and 9, and figure 3.2 shows a graphical representation of the environment and the process.

The minimal resources.xml example shows how to express the Name of the resource, the amount of ComputingUnits or task executor threads to use, or the communication adaptor of choice, including the range of ports to use for this communications. The minimal project.xml is yet smaller, where only the COMPSs installation directory and a temporary working directory need to be specified.
Listing 9 Default ComputeNode project.xml used by COMPSs

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Project>
  <MasterNode/>
  <ComputeNode Name="localhost">
    <InstallDir>/opt/COMPSs</InstallDir>
    <WorkingDir>/tmp/COMPSsWorker/</WorkingDir>
  </ComputeNode>
</Project>
```

Figure 3.3: Diagram for the CloudProvider deployment process

Cloud Provider workers

In the case of CloudProvider workers, the configuration needs to provide information about the number of virtual machines (VMs) to be deployed. Then, the runtime decides how many VMs are deployed in each of the configured cloud providers. To do so, a Cloud tag needs to be added, where provider-specific settings is specified, along with some information that is shared with the ComputeNode. The basic configuration of a CloudProvider resource includes the <Images> settings, which defines the image to be used when creating the VM, as well as the <InstanceType>, which defines the hardware specification of the virtual machine to launch. As with regular virtual machines, these images define the operating system to be deployed. If many images and/or instance types are defined, COMPSs uses more detailed information regarding cloud pricing and hardware specification, provided in the resources.xml file.

In the beginning of the execution, as many virtual machines as the already introduced InitialVMs tag specifies are requested by the COMPSs runtime, one by
one. Each virtual machine hosts one COMPSs worker. While the application runs, if any of the virtual machines fails or crashes, the runtime will also take care of making sure the total amount of machines stays over the minimum, as specified by MinimumVMs. If the workload exceeds the capacity of the deployed virtual machines at any given time, the COMPSs runtime will request more resources, never surpassing the maximum amount set by MaximumVMs, and always optimizing the cost, according to the information provided in the resources.xml configuration file. Nowadays, COMPSs provides connectors for Apache jcloud, Slurm, Docker Swarm, and Apache Mesos out of the box. Listings 10 and 11 show an example of how a cloud provider is configured in the project.xml and resources.xml files respectively.

### 3.1.3 Docker Swarm support

COMPSs implements support for a containerized, distributed execution with Docker Swarm. To do so, it provides two scripts: compss_docker_gen_image and runcompss-docker. The former assists in the building of the image, using the official COMPSs Docker image as the base, compss/compss. From it, the needed application files are added, and a new image constructed. This image can then be run using the aforementioned runcompss-docker command, which will launch a COMPSs master and as many COMPSs workers as requested with the command inside containers. The command must be run in a Docker Swarm manager, so that the containers can be launched in the Swarm worker nodes that it manages, allowing direct and easy network connectivity between all workers. Figure 3.4 represents this execution process and environment graphically.
Chapter 3. Background and Methodology

Listing 11 Cloud provider example for resources.xml

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ResourcesList>
  <CloudProvider Name="BSC_full">
    <Endpoint>
      <ConnectorJar>example-conn.jar</ConnectorJar>
      <ConnectorClass>es.bsc.conn.example.Example</ConnectorClass>
    </Endpoint>
    <Images>
      <Image Name="ExampleImage">
        <Adaptors> <!-- Adaptor configuration --> </Adaptors>
        <Price>
          <TimeUnit>100</TimeUnit>
          <PricePerUnit>36.0</PricePerUnit>
        </Price>
      </Image>
      <InstanceTypes>
        <InstanceType Name="Instance1">
          <Processor Name="P1">
            <ComputingUnits>4</ComputingUnits>
            <Architecture>x86_64</Architecture>
            <Speed>3.0</Speed>
          </Processor>
          <Memory>
            <Size>1000.0</Size>
          </Memory>
          <Storage>
            <Size>2000.0</Size>
          </Storage>
        </InstanceType>
      </InstanceTypes>
    </Images>
  </CloudProvider>
</ResourcesList>
```

Internally, this integration works by creating a Docker Compose YAML file like the one shown in listing 4. In this file, it creates a master service of one replica and a worker service with as many replicas as the user specified to the runcompss-docker command, using the recently created image. The worker containers are launched as server operating systems, with an awaiting SSH server, whereas the master container is launched with a traditional runcompss process. For this purpose, both XML configuration files are automatically generated, which will refer to the worker containers as regular ContainerNodes. In addition to this, the Docker Swarm CloudProvider is also added, so that more containers can be launched within the swarm if the workload is too large for the available containers. This will be explain more in-depth in section 4.3.

Due to how the integration is designed, the COMPSs master and all COMPSs workers run inside containers. These are isolated environments by design, which means that containers can not be accessed from the outside unless strictly specified. In addition to this, the provided base image, compss/compss, is not available for ARM architectures. All these issues, added up to the inability of using a custom pair of XML configuration files, prevents the Docker integration from working in combination...
Chapter 3. Background and Methodology

3.2 Advancing the deployment capabilities of COMPSs

COMPSs is a practical and efficient framework for distributed applications, and it certainly has found its place in a market with many alternatives, being its API the strongest asset. However, COMPSs does present some limitations in its design and has not adapted to or embraced more innovative and advanced technologies. This is the case for the compute continuum, an implicitly heterogeneous environment comprised of edge, fog and cloud devices.

For traditional COMPSs workers of the ComputeNode type to work, the whole COMPSs runtime has to be installed in the nodes, in addition to application files and their dependencies. The installation process also requires of many dependencies and the installation guide. Using COMPSs for many applications can cause conflicts on the long run, if applications rely on different versions of same dependencies, which could affect maintainability.

That is how containerization comes into play. Containers are created from images, which behave like stills of a file system. It can put an end to the infamous sentence "it worked on my laptop" that is usually attributed to developers. If the container from an image works in one environment, it is capable of running everywhere. It represents a self-contained working application.

This means that COMPSs can be extended to deploy and use containers as its workers. If the application could run within the container sans COMPSs, then...
installing the necessary files to run a COMPSs worker inside the container should be enough to make this container work as a COMPSs worker, no matter the underlying operating system or environment.

Furthermore, both Docker and Linux Containers (LXC) implement their own image transmission feature, meaning the application transmission is actually addressed, and it would no longer be required to manually install the application’s dependencies and files in the nodes, which could be an issue in the edge and fog environments. However, for this to work, the COMPSs Docker image needs to be completely redesigned to achieve a lightweight solution. This way, the image transmission costs will be minimized.

Finally, the cloud has taken a significant turn from virtual machines towards container-based environments, mainly driven by the most low-level as-a-service models such as Software-as-a-service (SaaS) or Container-as-a-service (CaaS) with platforms such as Kubernetes. Integrating these with COMPSs CloudProviders seems not only appropriate, but the next logical step towards embracing the newest containerization technologies, this time, aimed at the cloud.
Chapter 4

New deployment features in COMPSs

As explained in section 3.1.3, COMPSs already provides a Docker support. However, it does not take full benefit of a containerized compute continuum environment, due to the following reasons:

- Current Docker image size is too large (2.5 GB). The whole COMPSs runtime, with all its master and worker features, is installed within the image.
- A COMPSs execution using Docker is configured and launched in a different way with respect to the rest of executions, because the process is automated using two scripts that are dedicated to this feature.
- A direct consequence of previous item is that current Docker support is incompatible with other existing resources, i.e., the user must either run a Docker-based execution using the `CloudProvider`, or use the previously configured `ComputeNodes`.

Based on these drawbacks and in order to adapt COMPSs to new compute continuum technologies, this thesis proposes a new containerized support in COMPSs that overcome current features.

4.1 New lightweight container images

New Docker images must be designed to be as lightweight as possible. To achieve so, the minimal components of the COMPSs runtime that will guarantee the correct functioning of the COMPSs worker must be identified, getting rid of the remaining components and features. Also, building the base image on top of a lighter parent OS image, using Alpine instead of Ubuntu, drastically reduces the weight of the image. These findings can be applied to build the base LXC image as well and, thus, implement an integration that acts as an alternative to Docker.
Chapter 4. New deployment features in COMPSs

4.1.1 Improved Docker images

Docker images can be built using two methods. The first one is by writing a Dockerfile, like the one shown in listing 3. The other one is by creating a container from an already existing image, then adding the necessary files and dependencies, and storing it by running the `docker commit` command. Both these methods work in the same basic way. However, the Dockerfile method is encouraged, as it allows the building process to be reproducible, shareable, and automatic. If the building process can not be automatized (needs manual input at some point), then the manual approach must be used.

Images are made up of a layers in a specific order, each of these layers being an image. All images are uniquely identified by a 256 byte hexadecimal hash code. Additionally, images can be identified by a name and a tag. An image can only have one hash code, but many name and tag combinations can refer to the same actual image. Each line of the Dockerfile adds a new layer to the image that is being created. Thus, the key to create a lightweight image is to base it on lightweight base images and apply as little changes as possible, either by installing almost no libraries or by adding a few small files.

The images that are most commonly used for building new images are those of operating systems. The smallest operating system for this regard is Alpine Linux. The size of its images average 5 megabytes, so it seems like the most reasonable choice for building smaller images. However, using Alpine comes with some major inconveniences.

Alpine is built using a different implementation of the standard C library libc, called musl, than most Linux systems do. This library is an interface between the user and the kernel and implements important functions and system calls like `fork`. Most Linux distributions use glibc, making the binaries and libraries compiled in these systems incompatible with musl-based operating systems. For this reason, Alpine makes use of its own package management software, `apk`, which is capable of installing hundreds of libraries and other dependencies. These have been specifically compiled and shared by the Alpine community. Packages that are not available in this repository must be manually compiled.

The main drawback of `apk` is that it does not behave in a deterministic manner. One `apk` installation command can install a different version of the requested software depending on the moment it is run, because `apk` does not allow specifying the version to install. This can break over time a working automatized image building process that makes use of a Dockerfile, if the software for which the image is being built is incompatible with different versions of its dependencies. This is a real issue addressed by Alpine users, and with no solution as for now.
Chapter 4. New deployment features in COMPSs

Listing 12 Minimal installation of COMPSs worker for Java applications

```
compass
|-- adaptors
| `-- nio
|   `-- worker
|       `-- compss-adaptors-nio-worker.jar
|-- configuration
| `-- log
|   |-- COMPSsWorker-log4j.debug
|   |-- COMPSsWorker-log4j.off
|   `-- it-log4j.instrument
|-- scripts
| `-- system
| `-- adaptors
|   `-- nio
|       |-- persistent_worker_clean.sh
|       |-- persistent_worker_starter.sh
|       `-- setup.sh
```

Listing 13 Dockerfile to build Java worker base image

```
FROM alpine:3.8
LABEL maintainer="unai.perez@bsc.es"
COPY compss /opt/COMPSs/Runtime
RUN apk add --update --no-cache util-linux openjdk8-jre-base bash
```

The `ubuntu:18.04` image weighs 85 megabytes and includes the universally known `apt` package manager. COMPSs works best with Ubuntu, as explained earlier, so constructing a COMPSs image based on Ubuntu is guaranteed to work. The dependency installation would add, at the very least, megabytes of weight to the image, no matter which is the base operating system image. Thus, it must be decided if the lightweightness of Alpine outdo the benefits of using a robust and consolidated, but heavier operating system base like Ubuntu, which would also help in guaranteeing the consistency of a potential automated image building. If the latter is the case, once the benefits of lightweightness have been given up, the door opens to try new operating system base images, such as CentOS or Debian.

The minimal files from the COMPSs runtime that are needed in the worker will depend on the features that want to be used. Within the scope of this project, only two options have been considered: A base image for COMPSs Python applications and another for COMPSs Java applications. The COMPSs runtime is written in Java, so running a Java worker requires less dependencies. The tree of the minimal COMPSs installation for the Java worker container is shown in listing 12. To build the image, the contents of the `Runtime` directory are put aside in another directory named `compass`, then built using the Dockerfile in listing 4. The base image for Java applications could indeed be built using Alpine as the base image, as all dependencies are available through `apk`, and no `libc` originated compilation conflict happened. This Dockerfile creates an image of only four layers, which add up to around 97 megabytes in total.
Chapter 4. New deployment features in COMPSs

Listing 14 Dockerfile to build Java worker base image

```bash
FROM bscppc/conn-base as builder

ENV JAVA_HOME=/usr/lib/jvm/default-jvm
ENV LD_LIBRARY_PATH=$(JAVA_HOME)/jre/lib/amd64/server

ADD /root/framework

RUN apk add --no-cache --update bash git openjdk8 util-linux vget maven
  → alpine-sdk autoconf automake libtool boost boost-dev libxml2-dev tcsh python3
  ↪ python3-dev python2-dev py3-pip
  ↪ pip3 install setuptools wheel numpy dill guppy3
  ↪ sed -i 's/#define __NEED_time_t/#define __NEED_time_t
  → #define __CPU_SETSIZE
  ↪ 1024/ ' /usr/include/sched.h
  ↪ cd /root/framework/builders
  ↪ ./buildlocal -N -b -p --skip-tests
  ↪ apk del --no-cache --update git openjdk8 wget maven autoconf automake libtool
  ↪ boost boost-dev libxml2-dev tcsh python python2-dev
  ↪ apk add --no-cache --update openjdk8-jre-base

FROM alpine:3.10

ENV JAVA_HOME=/usr/lib/jvm/default-jvm
ENV LD_LIBRARY_PATH=$(JAVA_HOME)/jre/lib/amd64/server

COPY --from=builder /opt/COMPSs/Bindings /opt/COMPSs/Bindings
COPY --from=builder /opt/COMPSs/Runtime/adaptors/nio/worker
COPY --from=builder /opt/COMPSs/Runtime/adaptors/nio/worker
COPY --from=builder /opt/COMPSs/Runtime/configuration/log
COPY --from=builder /opt/COMPSs/Runtime/configuration/log
COPY --from=builder /opt/COMPSs/Runtime/scripts/system/adaptors/nio
COPY --from=builder /opt/COMPSs/Runtime/scripts/system/adaptors/nio

RUN apk add --no-cache --update python3 bash util-linux openjdk8-jre-base py3-pip
  ↪ python3-dev alpine-sdk libtool
  ↪ pip3 install wheel numpy dill guppy3
  ↪ apk del --no-cache --update python3-dev alpine-sdk libtool
  ↪ ln -s /usr/bin/python3 /usr/bin/python
  ↪ rm -rf /opt/COMPSs/Bindings/c
  ↪ /opt/COMPSs/Runtime/scripts/system/adaptors/nio/docker
  ↪ /opt/COMPSs/Runtime/scripts/system/adaptors/nio/lxc
  ↪ rm -f /opt/COMPSs/Bindings/python/3/pycompss
  ↪ cp -r /opt/COMPSs/Bindings/python/2/pycompss
  ↪ /opt/COMPSs/Bindings/python/3/pycompss
  ↪ rm -rf /opt/COMPSs/Bindings/python/2
```

In the case of COMPSs applications written in Python, more files from the runtime and more dependencies are necessary. However, using for this use case Alpine has been proven complex. Two methods were tested for the creation of the image with Alpine. The first was setting up the directory tree along with a Dockerfile and copying the precompiled files directly into the container. This did not work, due to incompatibilities between the two libc implementations. The other method was to copy the complete COMPSs runtime, compile it in-site, and then trim down the images final size by removing unnecessary files.

Many issues and incompatibilities were found and solved through trial and error throughout the image creation phase, but an Alpine-based Python application base
image could be created, with a size of 236 megabytes. The construction was then written into a Dockerfile to be automatized in a similar manner as the Java base image. The Dockerfile, shown in listing 14, is proof of how many adjustments were needed for the compilation to work in an Alpine environment. As opposed to the Java Dockerfile, this one is designed to be run from the root directory of the COMPSs runtime code. All the contents are copied into a builder container for its compilation, and all dependencies are installed. Finally, the compiled elements are copied to a fresh Alpine container, where only the necessary files and libraries are installed. Last but not least, the builder container is based off of a custom Alpine image with internal Maven dependencies installed which would not be obtainable otherwise, due to them not being publicly available.

4.1.2 New LXC support

Linux Containers, as explained, have an environment-oriented design, contrary to the application-oriented design of Docker. Thus, LXC does not use an analogue to Dockerfiles, which means that, from the explained approaches for image creation, only the second is available: A new container must be created, files and dependencies manually installed, then committed. So, using the CLI, and manually following the steps defined in the Dockerfile on an Alpine container was enough to create the base images. Listing 15 shows this process through the commands, from a directory with all needed files, just like with the Dockerfile in listing 13.

The result shows that the Java base image has a weight of 67 megabytes and the Python one is 119 megabytes, less than the Docker image for both cases. Thus, LXC images are generally smaller but more convoluted to construct from scratch. Also, images are not a sequence of ordered layers as with Docker, so any time an image needs to transferred, the whole 67 megabytes need to be sent over to the destination. On the other hand, no repository is needed, as any LXC instance can be used for image transmission.

It also important to note that images are not compatible across architectures, which means images must be created in the target architecture. However, Docker is building an experiment feature that will allow cross-creation of images.

To create an application image from the Docker base image, a Dockerfile needs to be written, the first statement being `FROM ${image-name}`, where image name refers
Chapter 4. New deployment features in COMPSs

to the base image. Then, the Dockerfile must be built using `docker build`. For LXC, an instance of the image must be instantiated by running `lxc launch image-name app`, where `app` is the name of the created container. Then, after manipulating the container, it can be stored as a new image with `lxc publish app -force -alias app-image`.

4.2 Automatic containerized deployment: container starters

To allow for a transparent and homogeneous deployment and usage of containers across the compute continuum, the current container integration needs a redesign. The same way `ComputeNodes` can be configured through the XML configuration files to natively run the COMPSs worker, these resources should also be able to deploy a COMPSs worker over different containerization engines. Once the initial deployment is done, the COMPSs runtime only cares about there being a COMPSs worker attached to a known port, through the IP or hostname specified in the XML files, to which to establish a connection.

To implement a containerized deployment for the COMPSs workers, a new abstract class `Starter` has been created. It contains part of the code of the old `WorkerStarter` class, discussed in section 3.1.2, which is now a child class of `Starter`. Additionally, another abstract child, named `ContainerStarter`, has been created, which contains generic boilerplate code for the COMPSs worker initialization and declares an abstract method that all its children must implement.

The implementations of this class will need to add the specifics of the communication between container engine. Their objective is to launch a container with the correct configuration and the default behavior of executing the worker as soon as it is ready. For this, the `Starter`s receive the command that needs to be executed to correctly launch a worker. Two child classes have already been implemented for the two container engines that will be supported, one `DockerStarter` and one `LXCStarter`.

Both Docker and LXC offer multiple communication interfaces for the interoperability with other software. The main interface in both cases is the command line interface (CLI), usually just called the `docker` or `lxc` commands respectively. If the user establishes an SSH connection to the engine’s host, they should be able to directly manage the local engine installation. However, the user must be able to run the `docker` command without `sudo`, which is not the default behavior. This is not necessary for LXC, which is able to separate containers per user. All in all, interacting with a remote containerization engine through SSH comes with a list of assumptions, specially in the Docker’s case.

The main and encouraged alternative to remotely interact with Docker is to use the REST API through HTTP/S. The user can manually configure to which interfaces and ports the Docker daemon of a specific resource must listen to, and
what protocols each of these bindings utilize. This configuration can be added to the /etc/docker/daemon.json file or, as it is usually done, to the systemctl service configuration file (or system equivalent) as flags of the dockerd binary. A common use case for this is to configure the daemon so that it uses HTTPS when contacted through a wireless interface, while using unsecured HTTP or straight-up TCP when contacted through the wired Ethernet interface.

So, thanks to the transparency that the new abstract classes provide, two different DockerStarters have been implemented. The first one uses insecure HTTP to communicate to the API of the remote Docker. This means the daemon must be correctly configured to accept HTTP requests on a reachable IP and port. Support for HTTPS has not been added. The second one, however, continues on the legacy of the original COMPSs WorkerStarter and uses passwordless SSH to establish a connection with the ComputeNode. Through the connection, it runs a script that takes care of creating a container from the configuration that is given to it. This script is run remotely through SSH, meaning it need not be installed in the resource. As mentioned, the SSH must be passwordless and the utilized user must be able to do interact with Docker without sudo. This can be achieved by using the root user or by adding the custom user to the docker user group. The LXCStarter also works with the help of a script that is remotely executed too, just as the DockerSSHStarter.

The SSH based Starter comes with some extra optional configuration. When Docker’s REST API is requested to create a container, the image is automatically downloaded if it is not already locally available. This will put the creation of the worker on hold and delay the rest of the execution. However, the scripts are prepared to check different scenarios. One of these is that if image is not available, the script will pull it in the background, but fail the deployment immediately. This way, COMPSs
Chapter 4. New deployment features in COMPSs

Listing 17 Example of resources.xml for containerized execution

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ResourcesList>
  <ComputeNode Name="localhost">
    <Processor Name="MainProcessor">
      <ComputingUnits>4</ComputingUnits>
    </Processor>
    <Adaptors>
      <Adaptor Name="es.bsc.compss.nio.master.NIOAdaptor">
        <SubmissionSystem>
          <Interactive/>
        </SubmissionSystem>
        <Ports>
          <MinPort>43001</MinPort>
          <MaxPort>43010</MaxPort>
        </Ports>
        <Properties>
          <Property>
            <Name>Engine</Name>
            <Value>${engine}</Value>
          </Property>
        </Properties>
      </Adaptor>
    </Adaptors>
  </ComputeNode>
</ResourcesList>
```

will keep working with the remaining workers, while the image is guaranteed to be available in the failed worker for subsequent executions.

To run a containerized execution of COMPSs, an image needs to be created, following the steps explained earlier. Then, the XML configuration files need to be adapted, as shown in the listings 16 and 17, in which, container-based resources will also be referred to as ComputeNodes. The encouraged setup is to have one resources.xml file with global settings, then one project.xml file per application. This way, all nodes will be configured globally to be used as either native or containerized workers, by respectively omitting or using the case-sensitive property Engine, which can take the values lxc or docker. Then, in each project.xml file, the name of the image must be specified for the application, using the property ImageName.

Due to the networking isolation feature of the containers, software running on their inside is not accessible by default from the outside. This means that to make the COMPSs worker accessible from the outside, a port forwarding needs to be configured. While working on this implementation, an issue was found in which the COMPSs worker could not reach back to the COMPSs master outside the container. This was solved by adding a special flag in the execution, master_name, which, as the name implies, lets the user define which name or IP the master should use when communicating. Therefore an example COMPSs application can be run using container engines by running runcompss [-project ${project.xml}] [-resources ${resources.xml}] -master_name ${ip} ${app} ${args}, where the optional flags project and resources point to the corresponding configuration files, app is the

33
Chapter 4. New deployment features in COMPSs

Listing 18 Abstract Connector class, base for all cloud connectors

```java
public abstract class Connector {
    public abstract Object create(String requestName, HardwareDescription hd, SoftwareDescription sd, Map<String, String> prop) throws ConnException;

    public abstract VirtualResource waitUntilCreation(Object id) throws ConnException;

    public abstract void destroy(Object id);

    public abstract float getPriceSlot(VirtualResource virtualResource);

    public abstract void close();
}
```

canonical name of the Java class or the Python file name, followed by the application’s arguments.

4.3 Container-oriented cloud providers

This container engine support, implemented through the Starter classes, gives way to other possibilities. Docker Swarm, as mentioned earlier, permits to define the services to deploy in a YAML file, and how many replicas of each service must be deployed. These replicas are launched all across the Swarm, in different resources, transparent to the user. As mentioned throughout section 2.2.2, there exist other frameworks that offer similar features, the main alternative being Kubernetes. COMPSs can take profit off of these, by implementing an integration that deploys many container replicas to be used as workers of one execution.

However, a limitation was found on the internal design of the Starter classes: they are only able to deploy one worker per resource. Changing the desing of the internal handling of ComputeNodes is not viable. This means that it would not be possible to write a custom Starter for Docker Swarm or Kubernetes that would be run using the ComputeNode name as the API endpoint. But, this is also for the good.

As a matter of fact, these frameworks are traditionally cloud-oriented, even though edge-oriented alternatives have recently spawned, as explained in section 2.3. Even in these cases, the frameworks use very similar communication interfaces and protocols, usually YAML or JSON over HTTP or HTTPS. So, implementing customized, Starter-like classes that support the deployment of multiple replicas per resource is the next natural step. And, considering the cloud affinity of these platforms, the CloudProviders of COMPSs are a perfect conceptual fit.

As of today, the traditional COMPSs CloudProviders used to work by requesting virtual machines to cloud provisioners such as the Apache Mesos. When an application is launched with a CloudProvider configured as a usable resource, the
runtime requests virtual machines through the corresponding cloud connector, one by one, until the specified minimum amount of virtual machines are available. As explained earlier, the COMPSs runtime will take care of increasing or reducing the resource amount, always between the specified limits, in the way that fits the execution the most.

The communication between the COMPSs runtime and the cloud providers is implemented through the stateful `Connector` classes. For each `CloudProvider` tag in the XML configuration files, one cloud connector is instantiated, hence the need for it to be stateful and keep track of the work done throughout the execution. Listing 18 shows the methods to implement to develop a custom connector.

However, when shifting the conversation from virtual machines to containers running on a container orchestration framework, some issues arise. First off, the notion of instance types, as explained in section 3.1.2, loses weight. Containers can be enforced hardware limitations, which is not the same as dedicating hardware to a virtual machine, but is not a common practice.

The concept that does gain relevance in a containerized context is the image. The `CloudProviders` already incorporate an `Image` tag that was used to specify the operating system to run in the virtual machine, much like regular, local virtual machines. The `Image` tag allows the configuration of the communication adaptors and some constraint-related declarations, among others. In other words, the `Image` tag comprises the configuration that would normally be part of the `ComputeNode`. Many `Image` elements can be created per `CloudProvider`, when, in reality, only one image should be used per application. So, all in all, despite the fact that the `Image` design that was implemented does not completely fit the new concept of container images, many of these configurations can be reused when implementing a container-oriented integration.

The main discrepancy between the original `CloudProviders` and the container approach is how the resources are requested and created. With virtual machines, this process is carried out with one resource at a time. This is, for each time the `create` method of the corresponding `Connector` class is called, one virtual machine gets created. Nevertheless, when requesting the creation of a deployment to any of the container orchestration platforms mentioned, the amount of containers must be specified in the only request done to the platform’s API. Neither the `Connector` class nor the COMPSs runtime are designed to be able to deploy multiple container replicas.

This means that the current cloud provider integration needs to be redesigned so that it supported cloud-oriented containerization platforms without losing support for virtual machine provisioning services. Additionally, the user must be provided a way to specify the amount of container replicas they want deployed. For this, it is clear that the XML configuration files need to be modified to accommodate the new design. The tags `MinimumVM`, `MaximumVM`, and `InitialVMs` refer to the amount
Chapter 4. New deployment features in COMPSs

Listing 19 Example of new project.xml for cloud providers

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Project>
  <MasterNode/>
  <Cloud>
    <CloudProvider Name="local-swarm">
      <InitialVRs>4</InitialVRs>
      <MaximumVRs>4</MaximumVRs>
      <Images>
        <Image Name="bscppc/cholesky">
          <InstallDir>/opt/COMPSs/</InstallDir>
          <WorkingDir>/tmp/COMPSsWorker</WorkingDir>
        </Image>
        </Images>
    <InstanceTypes>
      <InstanceType Name="InstanceType1" />
    </InstanceTypes>
    </CloudProvider>
  </Cloud>
</Project>
```

of virtual machines that all running cloud connectors will have to deploy, instead of
keeping the count in a per-connector fashion.

Listings 19 and 20 show an example, aimed at a local Docker Swarm installation,
of the new proposed configuration XML files for cloud providers. The previously
mentioned tags, which referred to virtual machines, are renamed to refer to the more
general concept of virtual resources. These tags are also rearranged, so that each spec-
ified CloudProvider element can set its own InitialVRs and MaximumVRs amounts.
The MinimalVRs tag is not created, and is instead represented by the InitialVRs tag.
The remaining configuration stays the same as before.

Internally, the values of these new tags are read, and the InitialVRs value is
passed to the corresponding cloud Connector class. The runtime will keep and use the
MaximumVRs configuration so that more replicas can be deployed in the future, if the
current workers can not keep up with the workload. This feature already existed in
COMPSs prior to this implementation, but now it will update an existing deployment
instead of deploying new virtual machines one by one. This feature, though, is not
implemented as of the writing of this document. Thus, it is encouraged that the
InitialVRs and MaximumVRs values are the same, to avoid unwanted conflicts and
latency.

Traditionally, when the COMPSs runtime deployed virtual machines through
these cloud connectors, it would just create the resource, and then access it through
SSH to launch the worker. However, containers must be started with a running
worker, executed using a worker command. Otherwise, the default container behavior
is to just stop, causing the whole process to halt. In the case of Docker, this is
done with a single request, setting the worker command as the container's startup
command. Cloud connectors did not have access to the worker command on older
versions of COMPSs. This was solved with the creation of a StarterCommand class,
Listing 20 Example of new \texttt{resources.xml} for cloud providers

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ResourcesList>
  <CloudProvider Name="local-swarm">
    <Endpoint>
      <Server>http://192.168.0.101:2376</Server>
      <ConnectorJar>docker-swarm-conn.jar</ConnectorJar>
      <ConnectorClass>es.bsc.conn.swarm.DockerSwarm</ConnectorClass>
    </Endpoint>
    <Images>
      <Image Name="bscppc/cholesky">
        <CreationTime>120</CreationTime>
        <Adaptors>
          <Adaptor Name="es.bsc.compss.nio.master.NIOAdaptor">
            <SubmissionSystem>
              <Interactive/>
            </SubmissionSystem>
            <Ports>
              <MinPort>43100</MinPort>
              <MaxPort>43110</MaxPort>
            </Ports>
          </Adaptor>
        </Adaptors>
      </Image>
    </Images>
    <InstanceTypes>
      <InstanceType Name="InstanceType1">
        <Processor Name="Processor1">
          <ComputingUnits>4</ComputingUnits>
        </Processor>
        <Price>
          <TimeUnit>1</TimeUnit>
          <PricePerUnit>0.085</PricePerUnit>
        </Price>
      </InstanceType>
    </InstanceTypes>
  </CloudProvider>
</ResourcesList>
```

which encapsulates all needed information and generates the worker command for the execution.

The SSH connection into a cloud resource is responsibility of the \textit{runtime connector} classes. The default is called \texttt{DefaultSSHConnector}. To avoid using SSH, the alternative \texttt{DefaultNoSSHConnector} must be used. Its use can be requested by means of a flag when running the application. However, it is clear that certain cloud connectors (those oriented towards virtual machines) will require the \texttt{DefaultSSHConnector}, while the new container-oriented cloud connectors must use \texttt{DefaultNoSSHConnector}. The use of the flag means only one runtime connector will be used for the whole execution, meaning the simultaneous use of a container cloud provider and a virtual machine is not possible. This has been solved by adding a default runtime connector per cloud connector, which will load automatically unless the runtime connector flag is used. As a consequence, the same COMPSs application images to be used with \texttt{Starters} will be compatible for their use with \texttt{CloudProviders}, as they do not
Listing 21 New abstract Connector class specification

```java
public abstract class Connector {
    public abstract Object create(String requestName, HardwareDescription hd, 
                                  SoftwareDescription sd, Map<String, String> prop, StarterCommand starterCMD, 
                                  int replicas) throws ConnException;

    public abstract List<VirtualResource> waitUntilCreation(Object... ids) throws 
                                  ConnException;

    public abstract void destroy(Object id);

    public abstract float getPriceSlot(VirtualResource virtualResource);

    public abstract void close();

    public static RuntimeConnector getRuntimeConnector();
}
```

include SSH servers and such a connection to inside the resource is no longer required.

Finally, traditional cloud connectors returned a list of IPs where virtual machines had been deployed. This IP was accessed through SSH by the DefaultSSHConnector. This had to be changed for container-oriented cloud connectors, which now return the IP and port of the communication, just like the Starter classes. The new abstract Connector class specification is shown in listing 21.
Chapter 5

Evaluation

In this chapter the effect of the proposed features will be measured for both the setup of the environment and the ultimate performance of the applications.

5.1 Experimental setup: Applications

This section presents the applications used for the evaluation of the proposed deployment technologies, consisting of 3 well-know HPC applications and a real use-case to be deployed as part of the European H2020 CLASS project [27].

• The matrix multiplication or \texttt{matmul} application, as the name implies, generates two random matrices of the size specified by the arguments and multiplies them.

• The K-Means application is a simple iterative clustering algorithm that takes the amount of clusters to find, $k$, as an input. Then, $k$ centroids that represent the center of a cluster are randomly generated. The iterative process consists of assigning each element to its closest centroid and recalculating the new position of the centroid as the average of all the elements assigned to it, until convergence or the maximum amount of iterations is reached.

• The Cholesky decomposition, or just \texttt{cholesky}, is a matrix decomposition algorithm, this is, an algebraic operation to transform a matrix into a product of matrices.

• The trajectory prediction application reads a video and detects cars and vulnerable road users, such as pedestrians, through the use of a YOLO neural network [28]. The application also tracks the objects detected. All this information is sent to the cloud, where the trajectory prediction of each object is computed. This trajectory can be used for collision avoidance, among other potential functions. The YOLO process is running on the edge side, independently from COMPSs, simulating the cameras installed in the city by using a video instead of real live footage. The COMPSs application is written in Python
Chapter 5. Evaluation

<table>
<thead>
<tr>
<th></th>
<th>Java</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monolithic installation</td>
<td>1.16 GB</td>
<td>1.87 GB</td>
</tr>
<tr>
<td>Official base image</td>
<td>2.65 GB</td>
<td>2.65 GB</td>
</tr>
<tr>
<td>New worker base image</td>
<td>97 MB</td>
<td>236 MB</td>
</tr>
<tr>
<td>matmul</td>
<td>97 MB (+20 KB)</td>
<td>-</td>
</tr>
<tr>
<td>kmeans</td>
<td>97 MB (+36 KB)</td>
<td>-</td>
</tr>
<tr>
<td>cholesky</td>
<td>-</td>
<td>394 MB (+158 MB)</td>
</tr>
<tr>
<td>Trajectory prediction</td>
<td>-</td>
<td>933 MB</td>
</tr>
</tbody>
</table>

Table 5.1: Table comparing the sizes of images and traditional installations

and takes care of interacting with the YOLO process through sockets, and then performing the calculations needed for the prediction.

5.2 Heterogeneous and automatic deployment

5.2.1 Applications

To evaluate the benefits of the newly introduced deployment options, the applications presented in section 5.1 have been containerized to allow their execution in diverse environments, all within the compute continuum.

The creation of the application images was done through the usage of customized Dockerfiles. For this purpose, two base Docker images, which include the minimal COMPSs files, are publicly available in the DockerHub, by the name of bscppc/compss-worker\(^1\) for Java applications and bscppc/pycompss-worker\(^2\) for Python applications, for both \(x86_64\) and \(aarch64\) architectures. Table 5.1 lists the sizes of the traditional installation and the images used for this evaluation, while also including the original base image, compss/compss, used for the traditional integration of COMPSs with Docker Swarm.

Building the application images from the new COMPSs base images is a matter of installing the dependencies of the application, if applies, and adding the corresponding application files in the \(/\text{compss}\) directory: a .py script in the case of Python, or a JAR file for Java applications.

\texttt{matmul} and \texttt{kmeans} are implemented in Java and, as such, the compilation is configured so that both JAR files contain all necessary dependencies for the application to work. The respective images have been built using the Dockerfiles shown in listings 22 and 23. As a result, both images weigh just 97 megabytes, only some kilobytes over the weight of the base image.

\(^1\)https://hub.docker.com/repository/docker/bscppc/compss-worker
\(^2\)https://hub.docker.com/repository/docker/bscppc/pycompss-worker
Chapter 5. Evaluation

Listing 22 Dockerfile used to build the matmul app image
FROM bscppc/compss-worker
ADD matmul.jar /compss/

Listing 23 Dockerfile used to build the kmeans app image
FROM bscppc/compss-worker
ADD kmeans-frag-app.jar /compss/

Listing 24 Dockerfile used to build the cholesky app image
FROM bscppc/pycompss-worker
ADD cholesky.py /compss/
RUN apk add --no-cache --update py3-scipy

The Cholesky decomposition is implemented in Python and uses both numpy and scipy. The former is already installed as part of the base Docker image, due to it being a COMPSs runtime dependency. The latter instead, needs to be installed manually in the Dockerfile. Thus, listing 24 shows the Dockerfile used for this application. As a consequence of this extra dependency, the image weighs 394 megabytes, 132 more than the base image.

The goal of the trajectory prediction application is to show the interactivity between devices in a heterogeneous environment by running parts of the application exclusively in specific parts, this is, to show that the user can choose where each task is run, while being oblivious of the deployment and the parallelization. The mentioned application parts refer to the object detection in the edge, closer to where the data is generated, and the trajectory prediction in the cloud, where more powerful resources are available. This has been achieved by using the constraint annotation of the COMPSs API. Part of the application requires CUDA libraries for compilation, which are not available for Alpine. As a consequence, the image of this application had to be built from scratch, with no Dockerfile whatsoever, and using the Ubuntu base image, resulting on 933 megabytes of application image.

The application images, as well as the base images, must be and have been built in their target architectures. Docker’s experimental manifest [29] feature has allowed to define a mask image that automatically forwards the requesting client to the image of its architecture. This permits to define all images of different architectures under the same name, which eases the deployment and configuration of the execution.

5.2.2 Compute continuum

The setup is represented in figure 5.1. The experimental setup for the evaluation represents the edge with one Nvidia AGX Xavier, represented as the green processor chip in the figure, and two Nvidia Jetson TX2 devices, represented as the black chips.
Chapter 5. Evaluation

The Nvidia Jetson AGX Xavier includes a 8-core ARMv8 host processor, a 512-Core Volta GPU and a 4-core i7-4600U processor; whereas the Nvidia Jetson TX2 devices feature a 4-core ARMv8 host processor and a Pascal GPU with 256 NVIDIA CUDA cores. A Dell Latitude laptop with a x86_64 i7-4600U CPU will be used, in different executions, as a regular ComputeNode, both monolithic (with a traditional complete COMPSs installation) and containerized, or as a single-node Docker Swarm CloudProvider, so that a complete heterogeneous environment can be defined.

The Nvidia Xavier device will act as the COMPSs master. This means that the COMPSs runtime has been installed using the traditional installation method. On top of this, the preparation of the overall setup has required of a user in the master device that can perform passwordless SSH to all resources, including itself.

Two experiments will be run as part of this evaluation, using multiple different scenarios:

- The first experiment will implement two scenarios. The first scenario will use all four devices as monolithic COMPSs workers, whereas the second one will use the resources for containerized workers.

- The second experiment will use the computer as a CloudProvider for two of the scenarios. In the other two, it will be part of the ComputeNodes. The edge nodes will be used for deploying containerized workers for two of those scenarios, one coinciding with the CloudProvider and one without it; and the same will go for monolithic workers.

The specific setup for the monolithic executions has involved the installation of the complete COMPSs runtime and the dependencies of each application in all
Chapter 5. Evaluation

<table>
<thead>
<tr>
<th></th>
<th>Java</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monolithic COMPSs installation</td>
<td>20 minutes</td>
<td>30 minutes</td>
</tr>
<tr>
<td>New worker base image</td>
<td>8 seconds</td>
<td>15 seconds</td>
</tr>
<tr>
<td>matmul</td>
<td>+0 seconds</td>
<td>-</td>
</tr>
<tr>
<td>kmeans</td>
<td>+0 seconds</td>
<td>-</td>
</tr>
<tr>
<td>cholesky</td>
<td>-</td>
<td>+9 seconds</td>
</tr>
<tr>
<td>Trajectory prediction</td>
<td>-</td>
<td>+30 seconds</td>
</tr>
</tbody>
</table>

Table 5.2: Table comparing the setup times of images and traditional monolithic installation

the resources, the cost of which is at 1.87 gigabytes of space disk and more than 30 minutes of installation.

On the other hand, the setup for the containerized executions has required the installation of Docker and the manual download of the images. COMPSs is able to automatize the image pulling, while Docker Swarm does it by default. Nevertheless, it is encouraged to manually pull the images to avoid putting the burden into the execution. The size of the images was already addressed throughout the previous section.

Installing Docker and downloading the layers corresponding to the base images is a process that only needs to be done once per node. Installing Docker can take up to two minutes. The time it takes to install the complete COMPSs runtime is compared to the time it takes to pull the new images in table 5.2. The download times of the application images are relative and must be added up to the download time of the base app.

5.2.3 Summary of deployment improvements

Thus, before running the experiments, tables 5.1 and 5.2 have made clear the non-functional benefits of using the new containerized deployment methods. This is specially the case considering the target kind of device. Edge and fog devices are shorter in both computing and energetic resources, so optimizing the image sizes and download times can bring huge profits to the environment.

This means that the whole containerized setup, applications and their dependencies included, will take significantly less than five minutes to install per node assuming a clean environment. Installing the COMPSs runtime for Java applications takes 20 minutes, whereas installing Docker and downloading the base image for Java applications takes around 128 seconds, which is almost 10 times faster. For Python applications, using the images makes the setup process 13 times faster. This heavily relies in the lightweightness of the images, which constitute around 8% of the weight of a monolithic installation in the case of Java, and around 12% in the case of Python.
5.3 Performance

Even though the main focus of the integrations discussed and implemented throughout this thesis is to facilitate the transparent and heterogeneous deployment of applications across the compute continuum, it is also crucial that the performance is affected as little as possible. Otherwise, the benefits would not outdo the drawbacks. Therefore, this section evaluates the performance of the applications on the different setups and deployment scenarios considered.

5.3.1 Deployment on the edge

This section evaluates the performance of the applications on the edge scenarios described in section 5.2.2. The proposed containerized deployment is compared against the monolithic deployment already available in COMPSs.

The plot in figure 5.3 compares the execution time of running the trajectory in different environments. Two of those combine traditional ComputeNodes with a Docker Swarm CloudProvider, represented by a laptop, while the other two only use ComputeNodes, as in the previous plots. The plot evidences that the results do not vary too much from one environment to another. This is to be expected, as the
Figure 5.3: Performance on the compute continuum.

CloudProvider is not built over an actual, resourceful cloud provider. This, nevertheless, proves the correct integration of the new container-oriented CloudProviders.

Figure 5.2 shows four graphs that correspond to the four applications introduced in the previous section. The X axis represents the amount of tasks that have been run for the specific execution, which depends on the parameters passed to the application. However, the X axis of the plot corresponding to kmeans references the amount of elements. This is because the parameters do not have any relation to the number of tasks, affecting, instead, to the granularity of the tasks.

As those plots clearly show, there is no noticeable difference between the two execution models, even though the monolithic executions show slightly better results. The key in these plots is that both lines stay very close to each other, and do not show a clear sign of distancing later on. This means that no latency is introduced when using a containerized worker. In the contrary, the reason behind the small gap between the lines could be due to the initialization phase, which seems to take longer than the traditional worker initialization. Despite the extra latency that this initialization requires, the monolithic execution is only up to 7%, 13%, 2%, and 4% faster than the containerized execution, for the matmul, kmeans, cholesky and trajectory prediction applications, respectively.

5.3.2 Deployment on the compute continuum (edge and cloud)

This section evaluates the performance of the trajectory prediction application on the compute continuum scenarios described in section 5.2.2. The work presented
throughout this thesis allows to have two different scenarios on the compute continuum. In both of them, the computation is distributed across the cloud and the edge devices, being the latter used either as monolithic or containerized deployment. Both deployments are compared against the monolithic deployment already available in COMPSs.

The plot in figure 5.3 shows the execution time of the trajectory prediction application. As described for figure 5.2, the X axis represents the number of COMPSs tasks. The plot evidences that the results do not vary too much from one environment to another. This application does not take fully benefit of the cloud execution since the CloudProvider is not built over an actual, resourceful cloud provider. This, nevertheless, proves the correct integration of the new container-oriented CloudProviders.
Chapter 6

Conclusions

The work proposed in this thesis addresses the need of a transparent, centralized and unified method to deploy sequential applications in a distributed manner on the compute continuum. The main challenge of this is the inherent heterogeneity of such an environment. This novel deployment method has been implemented into COMPSs, which will give it the capabilities to deploy its workers all across the continuum with minimal setup effort.

Concretely, the first objective was to use containerization technologies to enhance the deployment options of COMPSs. The newly developed ContainerStarter class, introduced in section 4.2, defines an abstract interface that can be used to implement the integration of COMPSs with container engines. The goal of these Starter classes is to deploy a container with a running COMPSs worker in a specific remote resource. The developer is given complete freedom as to how this feature will be actually implemented.

As part of this project, the Starters for both Docker and Linux Containers have been implemented. These run a bash script that takes care of the setup of the container and its environment, q.v. checking for available ports for forwarding or downloading the image if necessary. These scripts are designed to run remotely, putting emphasis in the fact that no COMPSs component is required to be installed in the resource at all. This translates into COMPSs being able to transparently deploy containerized COMPSs workers on two containerization engines, while also giving the option to implement more integrations in the future.

The second objective was to allow the transparent deployment and distributions of applications to edge and fog devices, minimizing the requirements and setup. The COMPSs Docker image has been redesigned from the ground up to minimize its weight and take benefit from the layered design of Docker images, so as to optimize the image transmission. Section 4.1.1 explains how in depth. Additionally, a LXC base image has also been designed and built. Traditionally, COMPSs requires the application to be in a known location of the remote resource, making the installation of the applications and their dependencies part of the setup. Thus, the new image design
directly addresses this issue, by already including those dependencies and application files in a known location, saving time on the setup.

The **Starter** classes mentioned earlier not only are compatible with the edge and fog devices, but they are also the encouraged way of deploying COMPSs workers in such environments. The only requirement is to set up the passwordless SSH connection between the devices and configuring the user to be able to use both `docker` and `lxc` commands without `sudo`. It is encouraged to pull the images manually to avoid putting a burden on the execution, but, if properly configured, COMPSs will consider the deployment in a resource failed if the image is not there, but will start the download so that the next time the resource is used the image is available.

Finally, the third listed objective was to facilitate this same deployment in cloud platforms. The COMPSs cloud connectors already provided a good interface for the configuration and deployment of virtual cloud resources. These have been redesigned internally so that the very similar configuration files can be applied for a container-oriented cloud deployment, in which the user has control over how many replicas are actually being launched. The integration of these cloud providers is to be implemented in its own cloud connector, which should extend the **Connector** class and implement its very simple abstract methods. But, from a general point of view, this new feature will allow users to deploy their sequential code to the cloud, and execute it in a distributed manner.

All in all, the main objective is to permit the transparent deployment across the compute continuum. The new redesign of the **CloudProviders** allows the interoperation of containerized workers in the cloud with **ComputeNodes** in the local network, for both container-based and monolithic COMPSs workers, as long as all workers have interconnectivity. This allows the developers to set up and work with heterogeneous environments, such as the compute continuum itself, with the minimum effort. In addition to this, the same images used for the containerized **ComputeNodes** can be used for the containerized COMPSs workers in the cloud.

The results of the evaluation, as shown in chapter 5, show that running containerized workers comes with a slight delay which could correspond to a longer initialization phase. However, these containerized workers do not seem to work slower nor faster than traditional workers. The ease of deployment, as just argued, is the main contribution of this project. So, in conclusion, all objectives are considered fulfilled.

## 6.1 Impact

The contributions of this thesis are being utilized for the development and use-cases of three different projects, funded by European Union’s Horizon 2020 research and innovation programme. These are CLASS, Elastic and DeepHealth.
CLASS [27] aims to develop a novel software architecture framework to help big data developers to efficiently distribute data analytics workloads along the compute continuum in a complete and transparent way, while providing sound real-time guarantees. Thus, a large part of the features that this project aims to provide relies on the newly developed ability of COMPSs to transparently deploy across the compute continuum, feature that is contributed by the work presented in this thesis.

ELASTIC [30] has the goal of developing a novel software architecture to help system designers to address the challenges of processing vast amount of information retrieved from geographically distributed data sources. The role of the contributions presented in this thesis is the very similar to that of CLASS. COMPSs will be the collarbone of the software architecture, by taking care of transparently distributing the applications to the edge, where the information is generated, or to the cloud, for faster processing.

DeepHealth aims to offer a unified framework completely adapted to exploit underlying heterogeneous HPC and Big Data architectures; and assembled with state-of-the-art techniques in Deep Learning and Computer Vision. The contributions presented in this thesis allow the deployment of learning workloads in diverse cloud environments.

6.2 Future work

First off, some improvable points of the introduced features will be discussed. Then, a discussion on future adoptable technologies will follow.

Creating a minimalist COMPSs worker image has been one of the focuses of this project, and building an application image has been proven to be a simple process, as Dockerfiles in listings 22 and 24 prove. However, this claims assume that the user knows how to write Dockerfiles and use the Alpine operating system, which behaves different to other mainstream Linux distributions such as Ubuntu. Ideally, a script of some kind could be introduced which would assist in the creation of the application images for those inexperienced in these matters.

Speaking of Docker images; as of the writing of this document, Docker can only run images built in an environment with the same processor architecture. This issue has been addressed in this project by creating an image for each of the used architectures, then creating a Docker manifest that would act as an invisible intermediary between the client and the image of the corresponding architecture. However, Docker is also introducing a new experimental image cross-building feature, buildx [31], (analogue to cross-compilation) that would allow to build images for different architectures from only one device, regardless of its architecture. This feature would be of tremendous help in environments as the heterogeneous one introduced in this
project, as there would be no need to remotely build the images of the edge and fog nodes.

Regarding the cloud, containerization technologies keep evolving and finding new uses. As novel containerization engines or models keep showing up, all of these can also be integrated within COMPSs. The most promising of these would be the serverless model \[32\], often referred to as Function-as-a-Service or just FaaS. This model executes a function when a given predefined event or trigger is registered. Amazon Lambda has been mentioned in chapter 2, which is hosted by Amazon Web Services and uses a pay-per-use model. Nevertheless, FaaS platforms such as KNative and OpenWhisk are designed to be deployed over Kubernetes and Docker respectively. Both these FaaS models can be integrated into COMPSs in the future, implementing a event-driven model into the runtime and providing strict control over what tasks are executed in which platforms, optimizing the price of the execution, considering the pay-per-use rate.
Bibliography


