BIG DATA EUROPE

Support Action

Big Data Europe – Empowering Communities with Data Technologies

Project Number: 644564 Start Date of Project: 01/01/2015 Duration: 36 months

Deliverable 3.3: Big Data Integrator Deployment and Component Interface Specification

Dissemination Level	Public
Due Date of Deliverable	M8, 31/08/2015 (officially shifted to M10)
Actual Submission Date	M11, 04/11/2015
Work Package	WP3, Big Data Generic Enabling Technologies and Architecture
Task	T3.3
Type	Report
Approval Status	Final
Version	v1.0
Number of Pages	47
Filename	D3.3-Big_Data_Integrator_Deployment_and_Component_Interface_Specification.pdf

Abstract: Describes a flexible and reusable Big Data platform, and how it can be used. Both a high-level overview is supplied, as well as practical HOWTOs. Unmet challenges are described together with their potential solutions.

Project funded by the European Union's Horizon 2020 Research and Innovation Programme (2014 – 2020)

History

Version	Date	Reason	Revised by
0.0	24/07/2015	Initial version	Erika Pauwels (TenForce)
0.1	07/10/2015	First version	Hajira Jabeen (InfAI) Erika Pauwels (TenForce)
0.2	23/10/2015	Intermediate version	Erika Pauwels (TenForce) Aad Versteden (TenForce)
0.3	30/10/2015	Peer review	Paul Massey (TenForce) Hajira Jabeen (InfAI) Yiannis Mouchakis (NCSR-D) George Papadakis (UoA)
0.4	03/11/2015	Address peer review comments	Erika Pauwels (TenForce)
1.0	04/11/2015	Final version	Erika Pauwels (TenForce)

Author List

Organisation	Name	Contact Information
TenForce	Erika Pauwels	erika.pauwels@tenforce.com
TenForce	Aad Versteden	aad.versteden@tenforce.com
TenForce	Paul Massey	paul.massey@tenforce.com
InfAI	Hajira Jabeen	jabeen@informatik.uni-leipzig.de
NCSR-D	Yiannis Mouchakis	gmouchakis@iit.demokritos.gr
UoA	George Papadakis	gpapadis@di.uoa.gr

Executive Summary

The BDE platform aims to evolve into a very flexible and easy to use platform for Big Data pipelines. The platform is operational in its current state, but we anticipate a much simplified user experience as some core components mature. The ideas behind the platform, the direction in which we intend it to evolve, as well as the current way in which the platform can be used, have been described. We have considered the evolution of the platform should the core components not evolve rapidly enough.

Together with the rest of the project, the BDE platform evolves on the uncertainty of the key data needs by the societal challenges. This brings forth a high focus on platform flexibility. The flexibility offered by abstracting the deployment platform using Mesos, and by packaging the components of a pipeline in Docker. This solution gives implementers near-complete flexibility in the choice of base technologies, like Flink, whilst ensuring a unified manner of deploying the resulting components.

The BDE platform uses cutting edge technologies, making it ready for the future. The challenge is that, even with their thriving community, some core components are not fully developed at the time of writing. Docker Swarm is most notable in this regard. The BDE platform is functional without these components and (given some extra development time) alternative solutions are possible in case the core components evolve slower than expected. We don't expect major issues in this regard, but it is important to note that the platform has not yet reached the level of maturity we intend to reach. Future releases of the platform should allow whole pipelines to be deployed in one go, without extra configuration, likely using Docker Compose.

A high focus has been placed on supporting the users of the platform. The use of Docker makes it straightforward to add new components. The Vagrant installation allows developers to get a quick glimpse of the platform without a complex installation. Various HOWTO sections help developers and administrators understand how to use and deploy the platform. The accompanying contextual documentation provides a background to understand the bigger picture.

Due to the constraints of the BDE project, the BDE platform has become more flexible and future-proof. We are optimistic in the positive evolution of the platform.

Abbreviations and Acronyms

API	Application Programming Interface
AWS	Amazon Web Services
BDE	Big Data Europe
CSV	Comma-separated Values
HDFS	Hadoop Distributed File System
JSON	Javascript Object Notation
OS	Operating System
REST	Representational State Transfer
RPM	Redhat Package Manager
TCP	Transmission Control Protocol
UI	User Interface

Table of Contents

List of Figures

1. Introduction

We can derive from [1] and [2] that a single Big Data solution would not suffice for BDE's goals. We envision a reusable Big Data platform which can be used across domains. This document describes the basic infrastructure for the BDE platform, including the necessary setup required to deploy the platform.

Each of the societal challenges seemingly contains different data challenges. There is no clear problem vector which sufficiently covers the challenges of each domain. We also notice that the Big Data space moves fast, with promising components becoming production-ready over the coming years – Flink [3] being a prominent example. For this reason we aim to move towards a reusable Big Data platform. An ambitious platform which copes with different problem vectors and is able to support newly introduced components, whilst still offering support for reusable components.

With these challenges in mind, we are working towards a platform that can be used in a broad context. Coping with various users of the platform, the barrier to entry should be kept comparatively low. The platform aims to provide an easy to use deployment infrastructure and a clear path for incorporating new components. The platform should also offer an easy way to setup and configure the solution to a specific problem. In other words, it should offer a way to setup a complete pipeline.

In this deliverable we describe the components that make up the basis of our architecture. We highlight the current challenges and how we aim to tackle them. We also provide hands-on tutorials for setting up and using the environment.

2. A Reusable Big Data Platform

The goal of the reusable Big Data platform is to support widely varying ways of solving Big Data problems without causing excessive cognitive overhead. This section gives an overview of the design-choices made on the BDE platform architecture and their repercussions without going into too many technical details.

Each of the architectural choices are made in light of a property the Big Data architecture should possess. The properties have been derived from the necessary flexibility of the system, as well as the experience in constructing long-running applications. There is a clear need to be flexible, hence we have chosen those components which allow for just that without these choices having too much of an impact on the overall platform performance. The components are also picked in such a way that they provide minimal cognitive overhead for deploying and implementing a complete pipeline.

This reusable Big Data platform may seem ambitious, and it is, but the necessary components are coming into place to make it possible. The key idea is to add a constraint only if it is necessary to ease the deployment and to pick that choice which leaves most freedom.

In our interpretation, a pipeline which solves a Big Data problem consists of multiple computational components. These components commonly use the MapReduce paradigm [4] to compute their results. The pipeline provides an answer to a specific Big Data problem. It should be easy to develop components and to deploy the pipelines constructed from them.

With these general remarks in place, we consider the following challenges: hardware assumptions, freedom of implementation, selection of technologies and pipeline deployment and (re)configuration. Each of these challenges is elaborated on in the following sections.

2.1 Minimize Hardware Assumptions

Manually maintaining machines is a chore. For big systems it becomes a big problem. Platforms can run on many locations. A cluster can be available in-house, or one could be rented from a cloud provider. The choice of having hardware in-house, or renting it externally should be a business decision, not a technical one. A flexible architecture should therefore abstract from the hardware.

The BDE platform relies on Mesos [5] to provide an abstraction of the hardware. The Mesos platform offers an abstraction of the available machines, making them to appear similar to one big machine. Regardless of where the machines run. With this approach it is possible to address your local cluster in the exact same way as an AWS based cluster, or a combination thereof. Mesos offers abstractions to schedule tasks on the available nodes, keeping in mind the high-level constraints supplied by the user. Frameworks which run on Mesos cluster oftentimes offer support for easily scaling up the nodes which perform a computation.

Reliance on the specific hardware is minimized by using Mesos, thus allowing the BDE platform to reach a wider audience. The technical details of the Mesos platform are described in section 3.1.

2.2 Freedom of Implementation

As described earlier, it is not obvious to see which Big Data technologies will provide the best solutions in 2020. It is important for the BDE platform to easily cope with new components, without deprecating the old ones. The highest level of support which we could offer, would be to run each component on its own set of physical machines. Maintaining these machines would likely take a long time, and the setup would be very custom. A virtual machine would be easier to maintain and setup, but may incur a large performance overhead. The solution combines the best of both: a component which offers a high level of abstraction, without an overly large performance overhead, whilst being easy to install.

Docker [6] offers a good solution in this regard. Docker containers, as these virtual machines are named, offer an abstraction very similar to that of an isolated machine. The necessary binary can automatically be downloaded from a central source. The Docker container abstraction allows the components to incorporate virtually any technology. One container could run Spark, another could run Storm, yet another container can use a complete custom implementation. All of these installations take almost the same amount of work to deploy.

The high-level abstraction for being able to deploy technologies is an important one. It allows us to use varying technologies when solving a particular problem. Section 5.4 describes how to build a Docker image.

2.3 Encourage Common Technologies

Variety is a good thing, chaos is not. Allowing implementers of pipeline components to use virtually any technology is a good trait. However, it provides a challenge in creating a single community. Understanding a complete pipeline may be impossible if all components of the pipeline use a unique technology.

As stated earlier, it is seemingly impossible to know what the optimal components will be in 2020. We should not focus on supporting a single technology. We can minimize chaos by offering more support for some technologies than for others. Technologies with the most support will therefore receive a bit of an extra benefit, making sure they are used more often.

In order to make it simple to construct a pipeline with a technology that is already supported, we construct base Docker images. The base Docker image is similar to a template in a microservice-based architecture. It offers a template implementation which can be extended for the particular problem that is being solved. This solution makes it a lot easier to implement a new pipeline component without limiting the options of the implementer. Should some base technology not be supported yet, a new base image can be built.

We could use a generic logging mechanism, like the ELK stack [7], for shifting through the logs of various components. However we foresee logging to be easier to support as certain components become used more often on the BDE platform. Selecting and supporting a logging framework will therefore become easier when common technologies are embraced.

From [1] we can derive that there are many competing technologies. We currently don't explicit a preference of one technology over another. Choices will appear naturally. Section 5.5 describes how to extend a base Docker with your specific implementation.

2.4 Multiple Problems, one Cluster

We do not expect a cluster to run only a single pipeline. It seems common for a cluster to be used for a limited set of pipelines, as the problem-space stays consistent. However, given the many societal challenges, we expect a cluster to be used for multiple Big Data solutions. The time it takes to reconfigure the server for a new pipeline should therefore be minimized. Switching the cluster from solving one challenge to another would preferably not take an overly long time. In an ideal solution, reconfiguration should be automatic.

For a cluster running multiple pipelines, this would boil down to many hard to manage requirements on the components. It seems suboptimal for a service to lock up specific system resources (like a fixed TCP port). Ideally, an abstraction of the infrastructure would be in place that circumvents these issues.

Both of the problems described here, quick pipeline deployment and isolation of computing components, can be solved with technologies in the Docker space. Isolation of system resources can be offered with Docker containers. All ports are available to a Docker container, as an abstraction of the hardware is made. Deploying a complete pipeline could be solved by using Docker Compose [8]. Docker Compose allows a user to specify a set of Docker containers, and how they need to interact with each other. This may allow a complete pipeline to be described in a single file.

A complete solution to this problem is currently not readily available. The future looks bright though. Docker Swarm [9] should offer a solution which allows us to deploy pipelines using Docker Compose. This technology is under very active development, and could therefore not yet be tested. Until this date, we have to resort to manual port-bindings, as described in 6.2.

2.5 Conclusion

The chosen limitations supply many benefits in terms of deployment, without placing too many constraints on the implementation of the components themselves. Not all of the technologies described here are finalized, as was noted above and will be repeated in the relevant sections. With these constraints we can offer much flexibility in the components of a pipeline as well as on the underlying hardware, whilst still offering a uniform method of deployment.

3. The Base Platform

3.1 Components in the Base Platform

3.1.1 Base Platform Architecture

Figure 1 gives an overview of the BDE platform deployment architecture. The lowest level consists of the hardware nodes which may be servers available in a data-center or hosted in the cloud. On top of this hardware layer, Mesos is installed. Apache Mesos is an open-source cluster manager which ensures the separate nodes in the cluster behave as one big machine to the user. It can be thought of as a kernel for a distributed operating system. The cluster manager dispatches work to nodes in a transparent way and gets back work results. It also provides cluster maintenance operations like adding or removing nodes and rebalancing the work.

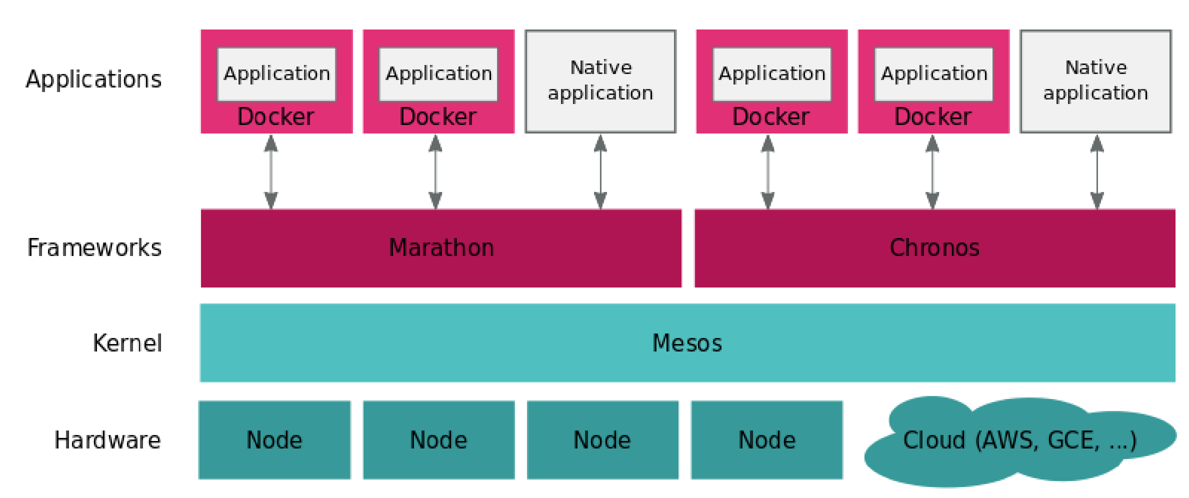


Figure 1: Base platform architecture

On top of the kernel layer there is a service layer providing frameworks that integrate with the Mesos kernel. Mesos provides an API in three languages (Java, Python and C++) so users can develop their own framework. As a number of frameworks are already publicly available, it is not the intent of the BDE project to build custom frameworks from scratch. The two most important frameworks that are already available are Marathon [10] and Chronos [11]:

- Marathon can be thought of as the startup or init daemon for a distributed OS. It automatically handles hardware or software failures and ensures that an application is always running. Marathon provides a web interface and a REST API to start, stop and scale applications.
- Chronos can be thought of as a cron daemon for a distributed OS. It is a fault-tolerant job scheduler that supports complex job topologies, thus being more flexible than a regular crontab. Chronos provides a web interface and a REST API to list, modify and run jobs.

The top layer of the architecture is the application layer. The intent of the BDE platform is to package each application component as a Docker container making components reusable across applications. Docker provides a simple packaging tool which can be used for development as well as for testing and production deployments. Selecting Docker as container also preserves some freedom in the packaging of the software. Docker containers may contain Debian packages, RPM packages, Java services, etc. so developers are not limited by choosing Docker. Moreover, as the Dockerfile, from which a Docker image is built, is just a plain text file, this file (and its different versions) can easily be managed in a version control system like Git.

The BDE platform will provide base images for a selection of Big Data technologies like Spark, Storm, Kafka, etc. The selection will mainly depend on the selected pilot cases and their requirements. The images will be built in such a way that they can easily be reused and/or extended with a custom application. Taking Spark as an example, the BDE platform will provide Docker images for the Spark master and Spark worker(s) which can be reused without modifications in all Spark applications. Next to these images, the BDE platform will also provide a Spark submit Docker image which allows you to submit a custom Spark application on the Spark cluster by extending the Docker image. Hence, the entire Spark application will be a collection of several Dockers working together. Of all these Dockers only one, the Spark submit Docker, needs to be customized per application.

Marathon as well as Chronos support the deployment of isolated Docker containers on top of Mesos. Marathon can run, deploy and scale Docker containers with ease. Chronos supports scheduling Docker jobs which run a finite command in a Docker container. However when using Marathon one cannot fully exploit the power of Docker because Marathon does not support the entire Docker API as Docker Engine does. For example, Docker containers cannot be linked (using the `--link` option) when deployed through Marathon. Since running applications on top of the BDE platform requires the deployment of a collection of Docker containers working together, an additional component might be needed to facilitate deployment. That is where Docker Swarm comes into play.

3.1.2 Docker Swarm Integration

The Docker community is currently working on a solution that responds to the need to deploy Docker containers at scale on a cluster: Docker Swarm. Swarm offers clustering support which turns a pool of Docker hosts into a single virtual host, i.e. all nodes in the cluster behave to the user as one big machine on which Docker containers can be deployed. The swarm is managed by a Swarm Manager. Requests to start, stop and remove containers are sent to the manager which then schedules the container on one of the nodes. Figure 2 shows a swarm consisting of three nodes which is managed by a Swarm manager.

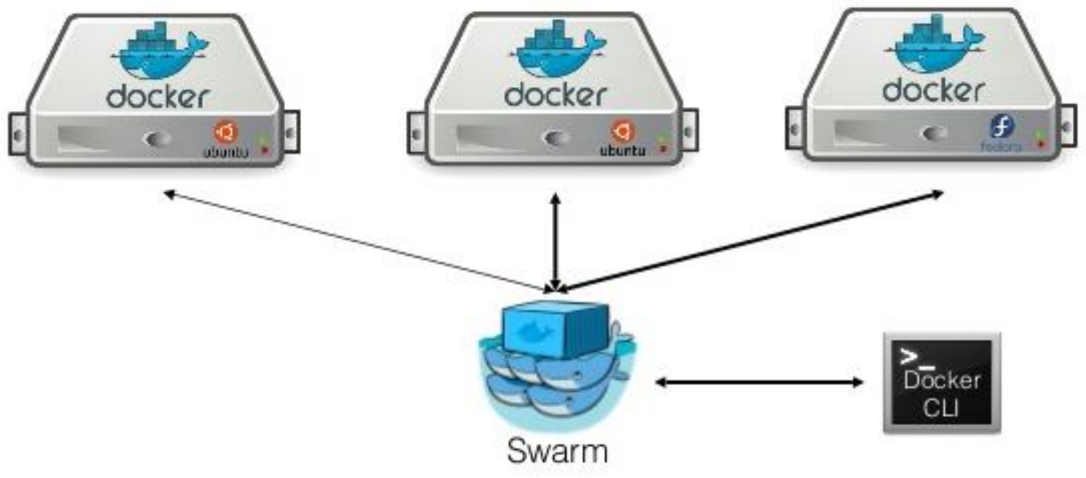


Figure 2: A Docker Swarm consisting of three nodes managed by a Swarm manager

One of the main advantages of Docker Swarm is the support of the standard Docker API: any tool which already communicates with a Docker daemon can use Docker Swarm to transparently scale to multiple hosts. Docker Swarm is currently in beta so it is not yet recommended to be used in a production environment.

By default Swarm comes with a built-in scheduler. A Swarm agent runs on each worker node in the cluster and communicates with the Swarm manager. An agent can run Docker containers when ordered to do so by the Swarm manager. Swarm is developed in such a way that the built-in scheduling backend can easily be replaced by a third-party backend while it still serves the standard Docker API. One of the supported backends is the Mesos scheduler. The Mesos slaves then fulfil the role of agents in the swarm. The Mesos master handles the scheduling of Docker containers.

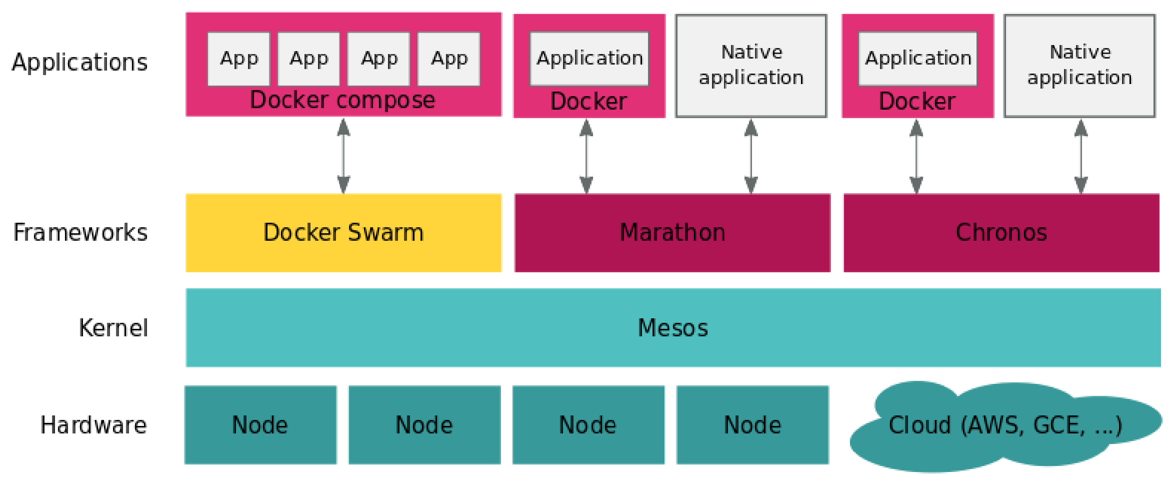


Figure 3: Base platform architecture including Docker Swarm and Docker Compose

As illustrated in Figure 3, Docker Swarm can be integrated in the BDE platform as a framework on top of Mesos next to Chronos and Marathon. The platform will benefit from the integration, for example by the facilitation of the deployment of linked Docker containers. Since the Mesos integration is still in an early stage we cannot come up with definite conclusions here, but the integration and speed of development look promising.

3.2 Cluster Setup per Node

Figure 4 shows the node setup of the BDE base platform listing the installed components per node. Section 5.2 describes the installation of each of these components. Generally speaking, the nodes in the cluster can be divided into three categories: work initiators, work distributors and work executors.

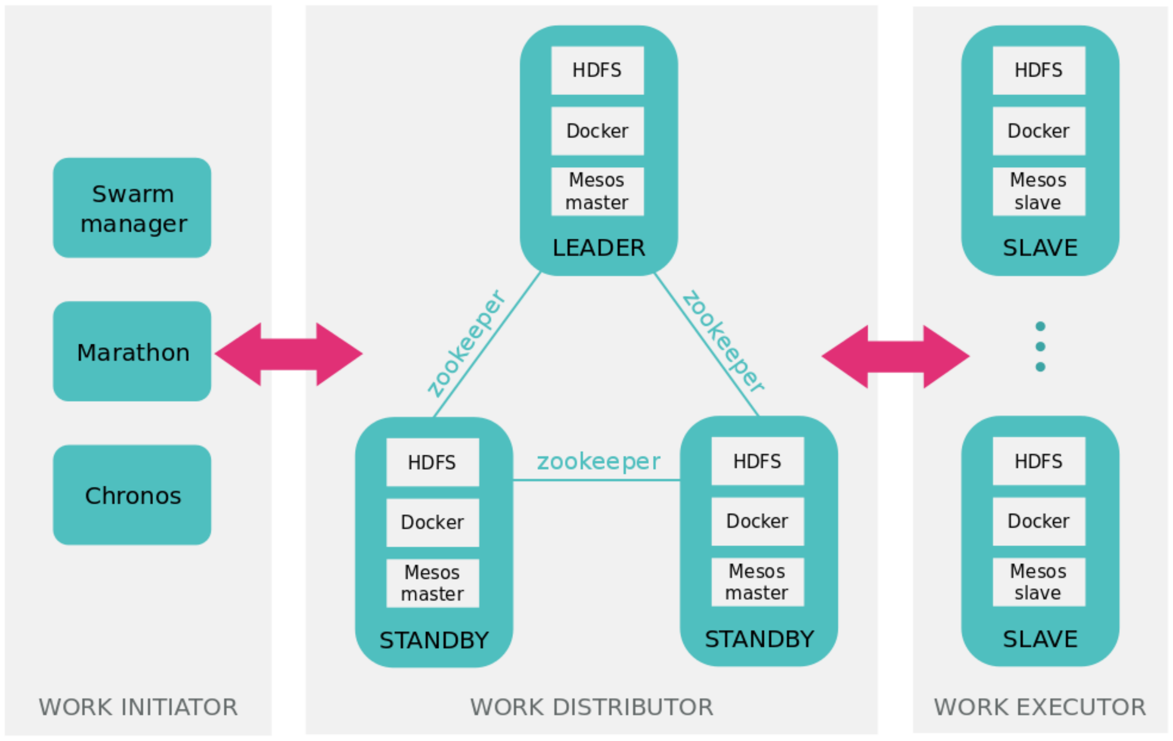


Figure 4: Cluster setup of the base platform

Work initiators are nodes that initiate work on the cluster. In the BDE base platform this role will be fulfilled by Marathon, Chronos and the Swarm manager. Each of these components will launch dockerized applications on the cluster.

The work distributors receive work from the work initiators, dispatch this work to the work executors and get back work results from them. In the BDE platform the work is distributed by the Mesos master. This is a daemon running on at least one node in the cluster.

As illustrated in Figure 4 Apache Mesos can be installed in a high-availability mode using Apache ZooKeeper [12]. The Mesos installation then consists of multiple Mesos masters: one active master and several failovers. Apache ZooKeeper coordinates the leader election and detection by masters, slaves and other schedulers. Although the high-availability mode is not required for Mesos to function correctly, it is highly recommended to enable failovers in a cluster to avoid the Mesos master being a single point of failure.

Work executors, frequently called the slaves or workers, receive work from the work distributors, execute the work, and send work results back to them. In the BDE base platform the work executors are the Mesos slaves. Similar to the Mesos master, a Mesos slave is a daemon running on each worker node in the cluster.

The work in the BDE platform is packaged as dockerized applications. Hence, the work distributor as well as the work executor nodes should be able to handle and run Docker containers and thus should have Docker installed. Work distributors and executors should also have access to the data to run the application on and to a location to write the output to. This requirement can be fulfilled by installing a distributed file system like HDFS [13] on the cluster nodes. The installation manual in section 5.2 describes a native installation of HDFS. However it might be the case that future investigation reveals it is better to install HDFS as a Docker container.

Each of the components of the BDE platform is described in more detail in [1]. Section 5.2 describes step-by-step how to install this base platform on your cluster. Section 5.1 describes how the environment can be installed for initial testing.

4. Constructing Pipelines for a Reusable Platform

An application pipeline consists of multiple computational components, which are wired together in order to solve a specific Big Data problem. As specified earlier, the goals are to allow for widely varying implementation in the computational components, whilst still allowing for a simple way to construct and deploy a complete pipeline. The construction of both the pipeline as well as of the computational components is heavily based on Docker and related technologies. Both the construction of the computational components and the wiring of a pipeline are discussed in this section.

4.1 Computational Components

A computational component solves a specific Big Data problem using a specific technology. It is a unit that can be used to solve a portion of the problem in the pipeline.

The components can best be grouped by their used technology, as you would build a regular pipeline. Constructing the components in Docker containers will not yield an extra performance impact when staying within a particular stack. However, multiple steps of the computation can be performed using different technologies. A batch layer could therefore use a completely different technology than the corresponding speed layer. The same holds for storage backends, and they can be swapped out as the Big Data solution evolves. The components make it easier to maintain the Big Data solution.

A base Docker supports the easy creation of a new component in a supported base technology. If you intend to solve the batch layer problem of your component in Spark, you can use the base Spark Docker. The BDE project will provide such base images for a selection of Big Data technologies. The selection will mainly depend on the pilot cases that need to be implemented.

The development of the Spark algorithm can occur as you normally do. When the component is tested and is working fine, it can be wrapped, so it can easily be deployed in the BDE platform. Wrapping a component consists of little more than adding the JAR file, which contains the Java bytecode to run on the platform, to existing containers. In order to do this, extend the base Docker images and add the JAR file. See section 5.5 for the specific commands. Packaging code in Docker makes it easy to share new components for a pipeline.

This construction does not add a lot of burden on developers when they use a known technology, but it also supports them when using new technologies. Docker images behave like little virtual machines thus dependencies can be added to them as necessary. A dependency could be a particular C header file which needs to be available on the machine, or it could be a completely different Big Data technology. The computational components have their own space which cannot interfere with other components. This solves the problem of requiring incompatible libraries on the host system.

Using Docker as a basis for the components of the pipeline, we supply a lot of freedom in terms of our implementation and ease the implementation of the components. This technology also fits well within the construction and running of pipelines. Although some of the libraries in the Docker ecosystem are still under active development, the finalized product should be a perfect fit.

4.2 Building Pipelines

A pipeline connects multiple computational components together into one smooth-running system. The goal is to ensure that computational components can be wired together easily and that the pipelines can be deployed without too much effort.

In order to construct a pipeline, the components need to be connected to each other. Docker provides abstractions for its containers which help connect the dots. For sharing files, Docker offers shared folders, in which the folder of one Docker container is transparently available in another Docker container. Further support is supplied in terms of links. A container can be accessible from another container under a specific hostname. This abstraction allows us to ignore the IP addresses of the specific machines. Hence, the computational components can be constructed without much information about the environment in which they will appear, minimizing the necessary configuration.

A pipeline can be constructed using Docker Compose. Deploying Docker Compose pipelines is under active development. Development moves forward and we are looking forward to the implementation. A Docker Compose file contains the identifiers of all Dockers which should be ran, and how the components need to be wired together. The Docker Compose file could thus contain a Docker for the Cassandra database and another Docker which contains the Spark application to compute results on top of this. The latter container is a custom component which is constructed by extending a known Spark base Docker image. When specifying the pipeline, the connections are made between the components.

A Docker Compose powered pipeline can scale its components as is expected from a Big Data pipeline. It is a manual effort, but with a single command, the workers of the pipeline can be scaled up. Given enough room on the Mesos cluster, to which nodes can be added on the fly, more computing nodes will be launched with the goal of finishing the computation in a shorter time frame.

Installing computational components for a pipeline should not take too much human effort. With Docker, the necessary content to run can be downloaded on demand using the Docker registry. A Docker registry is a place where many Docker images are stored. When Docker does not yet know how to run a specific component, it can automatically download the necessary image from the registry. This is an added benefit in terms of deployment, which has a big impact on how we can run the components. The installation of new components is therefore semi-transparent. Section 5.4.4 explains how to publish a Docker image on Docker Hub [14], the public Docker registry provided by Docker itself. It may take a while to download the image, but no manual effort is required. We should note that this delay can cause reporting issues, a case which we are keeping an eye on. Terminated Docker containers are also not removed automatically, but that should be an easy problem to solve.

A pipeline can be constructed in an easy format, and it is even easier to run it. Base technologies for running these images on a Big Data platform are under active development. Some of them are not production-ready yet. However, given how well this component fits within the goals of BDE, and how rapidly the ecosystem tackles new challenges, we are looking forward to incorporate the components in the BDE platform. On the whole these characteristics make the BDE platform a simple platform to develop components for, and to run complete pipelines on.

5. The Platform, Hands-On

5.1 HOWTO: Get an idea of the platform with Vagrant

Section 5.2 shows the full installation of the platform. The platform's installation is straight-forward but time-consuming. A preview can be installed using Vagrant [15] in order to get a basic idea of the platform, and how it can be used. The installation will run on a single node and it has a limited set of slaves, but it is trivial to install and provides a good overview. We start with the installation of the platform in the following sections, assuming that you have Vagrant installed as described in [16].

5.1.1 Hardware Requirements

- 10Gb RAM
- 30Gb Disk space

5.1.2 Software Requirements:

- VirtualBox v5.0.6 [17]
- Vagrant v1.7.4
- CLI (Like GNU Bash on Unix systems or cygwin on Windows)

5.1.3 Starting the Cluster

```
$ git clone https://github.com/big-data-europe/vagrant-mesos-
multinode
$ cd <clone directory>
$ vagrant up
```

After first boot: restart the Vagrant instances to correct the network configuration.

```
$ vagrant halt
$ vagrant up
```

You can ssh into the machines using the regular Vagrant commands:

```
$ vagrant ssh slave1
```

5.2 HOWTO: Setup the Base Platform

5.2.1 Version details

Following are the details of components used, all open-source:

- Apache Hadoop 2.7.1
- Ubuntu (32 bit) 14.04
- Java 7

5.2.2 Installing Java

The following commands download and install the JDK in `/usr/lib/jvm`. The JRE can be found inside the JDK folder at `/usr/lib/jvm/java*/jre`.

```
$ sudo apt-get install openjdk-7-jdk
$ javac -version
>> java version "1.7.0_25"
```

5.2.3 SSH and Create a New User

Install OpenSSH:

```
$ sudo apt-get install openssh-server
```

Create `hduser` user belonging to the `hadoop` group:

```
$ sudo addgroup hadoop
$ sudo adduser --ingroup hadoop hduser
$ sudo adduser hduser sudo
```

Switch to the `hduser`:

```
$ sudo su hduser
```

Setup an SSH key:

```
$ ssh-keygen -t rsa -P ''
```

Your private key has been saved in `/home/hduser/.ssh/id_rsa`. Your public key has been saved in `/home/hduser/.ssh/id_rsa.pub`.

Add the public key to the authorized keys so you can login without a password.

```
$ cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
$ ssh localhost
$ exit
```

5.2.4 Allow Passwordless Sudo

We have not set a password for our user. Hence, we need passwordless sudo. Open the sudoers configuration file:

```
$ sudo visudo
```

Add the following line to the file:

```
hduser ALL=(ALL) NOPASSWD:ALL
```

Remove the password for your user for extra security:

```
$ sudo passwd hduser -l-t
```

5.2.5 Installing Hadoop

5.2.5.1 Download Hadoop

Download and extract Hadoop.

```
$ cd downloads
$ wget http://mirrors.gigenet.com/apache/hadoop/common/stable/
hadoop-2.7.1.tar.gz
$ sudo tar xvzf hadoop-2.7.1.tar.gz -C /usr/local
$ cd /usr/local
$ sudo mv hadoop-2.7.1 hadoop
$ sudo chown -R hduser:hadoop hadoop
```

5.2.5.2 Configure Hadoop environment variables

Append the following to the `bashrc` file (verify the JVM path exists):

```
#Hadoop variables
export JAVA_HOME=/usr/lib/jvm/java-1.7.0-openjdk-amd64/
export HADOOP_INSTALL=/usr/local/hadoop
export PATH=$PATH:$HADOOP_INSTALL/bin
export PATH=$PATH:$HADOOP_INSTALL/sbin
export HADOOP_MAPRED_HOME=$HADOOP_INSTALL
export HADOOP_COMMON_HOME=$HADOOP_INSTALL
export HADOOP_HDFS_HOME=$HADOOP_INSTALL
export YARN_HOME=$HADOOP_INSTALL
```

Edit `/usr/local/hadoop/etc/hadoop/hadoop-env.sh`. Change JAVA_HOME variable

```
export JAVA_HOME=/usr/lib/jvm/java-1.7.0-openjdk-amd64/
```

Validate the Hadoop installation by requesting the version:

```
$ hadoop version
```

5.2.6 Configure all Properties for a Single Node Hadoop Cluster

Add the following to the `<configuration>` property of `/usr/local/hadoop/etc/hadoop/core-site.xml`:

```
<property>
  <name>fs.default.name</name>
  <value>hdfs://localhost:9000</value>
</property>
```

Add the following to the `<configuration>` property of `/usr/local/hadoop/etc/hadoop/yarn-site.xml`:

```
<property>
  <name>yarn.nodemanager.aux-services</name>
  <value>mapreduce_shuffle</value>
</property>
<property>
  <name>yarn.nodemanager.aux-services.mapreduce.shuffle.class</name>
  <value>org.apache.hadoop.mapred.ShuffleHandler</value>
</property>
```

Add the following to the `<configuration>` property of `/usr/local/hadoop/etc/hadoop/mapred-site.xml`:

```
<property>
  <name>mapreduce.framework.name</name>
  <value>yarn</value>
</property>
```

Add the following to the `<configuration>` property of `/usr/local/hadoop/etc/hadoop/hdfs-site.xml`:

```
<property>
  <name>dfs.replication</name>
  <value>1</value>
</property>
<property>
  <name>dfs.namenode.name.dir</name>
  <value>file:/home/hduser/mydata/hdfs/namenode</value>
</property>
<property>
  <name>dfs.datanode.data.dir</name>
  <value>file:/home/hduser/mydata/hdfs/datanode</value>
</property>
```

5.2.7 Initialise HDFS

Format the namenode:

```
$ hdfs namenode -format
```

Start the Hadoop service:

```
$ start-dfs.sh
$ start-yarn.sh
$ jps
```

If everything is successful, you should see following services running

```
2583 DataNode
2970 ResourceManager
3461 Jps
3177 NodeManager
2361 NameNode
2840 SecondaryNameNode
```

5.2.8 Validate the Setup

If you would like to test the setup, you can run the following example:

```
$ hadoop jar /usr/local/hadoop/share/hadoop/mapreduce/hadoop-
mapreduce-examples-2.7.1.jar pi 2 5
```

You can check the health of the HDFS Namenode on http://localhost:50070 and of the secondary namenode on http://localhost:50090.

5.2.9 Setting up a Hadoop Cluster

Setup the hostname for the Hadoop master (e.g. `hadoopmaster`) and the Hadoop slave (e.g. `hadoopnode`) in `/etc/hostname` on each machine. Add all hostnames with their corresponding IP address to `/etc/hosts` on all machines. Setup SSH and a new user as described in 5.2.3.

Add the hostnames of the Hadoop master nodes to `/usr/local/hadoop/etc/hadoop/masters` and of the slave nodes to `/usr/local/hadoop/etc/hadoop/slaves`.

Add the following to the `<configuration>` property of `/usr/local/hadoop/etc/hadoop/core-site.xml`:

```
<property>
  <name>fs.default.name</name>
  <value>hdfs://HadoopMaster:54310</value>
</property>
```

Add the following to the `<configuration>` property of `/usr/local/hadoop/etc/hadoop/yarn-site.xml`:

```
<property>
  <name>yarn.resourcemanager.resource-tracker.address</name>
  <value>HadoopMaster:8025</value>
</property>
<property>
  <name>yarn.resourcemanager.scheduler.address</name>
  <value>HadoopMaster:8035</value>
</property>
<property>
  <name>yarn.resourcemanager.address</name>
  <value>HadoopMaster:8050</value>
</property>
```

Add the following to the `<configuration>` property of `/usr/local/hadoop/etc/hadoop/mapred-site.xml`:

```
<property>
      <name>mapreduce.job.tracker</name>
      <value>HadoopMaster:54311</value>
</property>
<property>
      <name>mapred.framework.name</name>
      <value>yarn</value>
</property>
```

Add the following to the `<configuration>` property of `/usr/local/hadoop/etc/hadoop/hdfs-site.xml`:

```
<property>
  <name>dfs.replication</name>
  <value>3</value>
</property>
```

5.2.10 Setting up Mesos Cluster with Marathon

5.2.10.1 Install Mesos

```
$ sudo apt-key adv --keyserver keyserver.ubuntu.com \
     --recv E56151BF
$ DISTRO=$(lsb_release -is | tr '[:upper:]' '[:lower:]')
$ CODENAME=$(lsb_release -cs)
$ echo "deb http://repos.mesosphere.io/${DISTRO} ${CODENAME} main" \
    | sudo tee /etc/apt/sources.list.d/mesosphere.list
$ sudo apt-get -y update
$ sudo apt-get -y install mesos marathon
$ sudo service zookeeper restart
```

You can now start the Mesos master, Mesos slave and/or Marathon using the following commands:

```
$ sudo service mesos-master start
$ sudo service mesos-slave start
$ sudo service marathon start
```

5.2.10.2 Zookeeper settings

Choose a unique identifier between 1 and 255 for each of the nodes and set it in `/etc/zookeeper/conf/myid` and `/var/lib/zookeeper/myid` on the respective host.

Add each node with its unique identifier `{ID}` and IP address `{IP}` to the file `/etc/zookeeper/conf/zoo.cfg` as follows:

```
server.{ID}={IP}:2888:3888
```

The file needs to be available on each of the nodes with every node mentioned.

Configure the Mesos ZooKeeper URL in `/etc/mesos/zk` on each node as follows:

```
zk://IP1:2181,IP2:2181,IP3:2181/mesos
```

You can also setup Marathon in high-availability mode using ZooKeeper using a similar setting as above.

5.2.10.3 Quorum

Set `/etc/mesos-master/quorum` on each master node to a number greater than the number of masters divided by 2. For example, the optimal quorum size for a five node master cluster would be three. In this case, there are three masters and the quorum size should be set to two on each node.
Restart the services to ensure everything boots up correctly:

```
$ sudo service mesos-master restart
$ sudo service mesos-slave restart
$ sudo service marathon restart
```

5.2.11 Installing Docker

Install Docker on each node in the cluster by executing the command:

```
$ curl -sSL https://get.docker.com | sh
```

5.3 HOWTO: Setup Docker Swarm

5.3.1 Using the Built-In Scheduler

To create a Docker Swarm using the built-in scheduler you first need to create a swarm. Run the following command on any machine that has Docker installed:

```
$ docker run --rm swarm create
```

You will get a Swarm cluster id as response. Store this id somewhere as you will need it in the following commands to setup the swarm.

Docker needs to be installed on any machine that will be part of the Docker Swarm. A TCP port (e.g. 2375) needs to be opened on each node for communication with the Swarm manager. Start the Docker daemon as follows:

```
$ docker daemon -H tcp://0.0.0.0:2375
```

Next, run the following command on each of the nodes that needs to be part of the swarm using the cluster id created in the first step:

```
$ docker run -d swarm join --addr=<node_ip>:2375 \
  token://<cluster_id>
```

`<node_ip>` is the IP via which the node is accessible for the Swarm manager.

Finally, run a Swarm manager to manage the swarm by executing the following command on a machine that has Docker installed and has access to all the nodes in the swarm:

```
$ docker run -p <manager_port>:2375 -d swarm manage \
  token://<cluster_id>
```

`<manager_port>` is the port on which the Swarm manager will start listening (e.g. 2222).

To get information on the available agents in the swarm, execute the following command:

```
$ docker -H tcp://0.0.0.0:2375 info
```

An example response looks like:

```
Containers: 2
Images: 6
Role: primary
Strategy: spread
Filters: affinity, health, constraint, port, dependency
Nodes: 2
 ip-172-31-15-52: 52.29.9.203:2375
  └ Containers: 2
  └ Reserved CPUs: 0 / 2
  └ Reserved Memory: 0 B / 4.052 GiB
  └ Labels: executiondriver=native-0.2, kernelversion=3.13.0-48-
generic, operatingsystem=Ubuntu 14.04.2 LTS, storagedriver=aufs
```

```
 ip-172-31-15-55: 52.29.46.154:2375
  └ Containers: 0
  └ Reserved CPUs: 0 / 2
  └ Reserved Memory: 0 B / 4.052 GiB
  └ Labels: executiondriver=native-0.2, kernelversion=3.13.0-48-
generic, operatingsystem=Ubuntu 14.04.2 LTS, storagedriver=aufs
CPUs: 4
Total Memory: 8.105 GiB
Name: 7e83dc5875c5
```

How to issue commands to the Swarm manager is explained in more detail in section 5.6.2. More details on the installation of Docker Swarm with the built-in scheduler can be found at [18].

5.3.2 Using the Mesos Scheduler

To create a Docker Swarm using the Mesos scheduler in the backend you first need to setup the Mesos master(s) and slaves as described in section 5.2.10. Docker must be installed on all of the slaves. A TCP port (e.g. 2375) needs to be opened on each node for communication with the Swarm manager. Then, we can start the Docker daemon as follows:

```
$ docker daemon -H tcp://0.0.0.0:2375 -H unix:///var/run/docker.sock
```

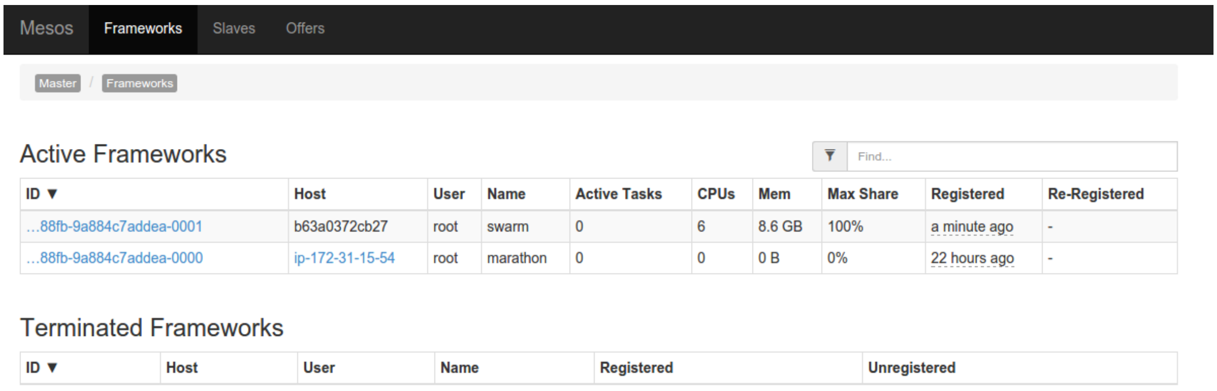
Make sure that the Mesos slaves are capable of starting tasks in a Docker container by setting the containerizer option to `--containerizers=docker` on slave startup and have a TCP port to listen to the Mesos master (5051 by default).

Next, start the Swarm manager. This is typically done on the same node as the Mesos master.

```
$ docker run --name swarm-manager \
     -p <manager_port>:2375 -p 5051:5051 \
     -e SWARM_MESOS_USER=root \
     -d swarm manage \
       -c mesos-experimental <mesos_master_ip>:<mesos_master_port>
```

`<manager_port>` is the port on which the Swarm manager will listen on (e.g. 2222). `<mesos_master_ip>` and `<mesos_master_port>` are the IP address and port of the Mesos master. The default port is 5050.

Swarm will be listed as an active framework on the Mesos dashboard if the Swarm manager started successfully. This is shown in Figure 5.



Active Frameworks

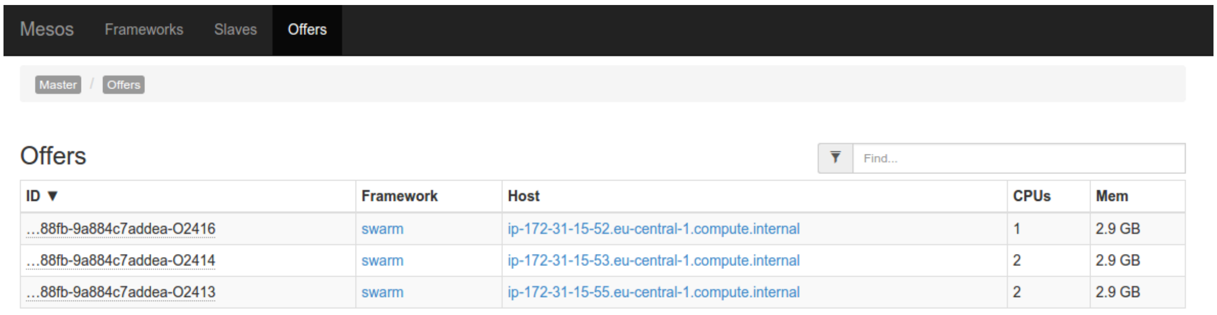
ID ▼	Host	User	Name	Active Tasks	CPUs	Mem	Max Share	Registered	Re-Registered
…88fb-9a884c7addea-0001	b63a0372cb27	root	swarm	0	6	8.6 GB	100%	a minute ago	-
…88fb-9a884c7addea-0000	ip-172-31-15-54	root	marathon	0	0	0 B	0%	22 hours ago	-

Terminated Frameworks

ID ▼	Host	User	Name	Registered	Unregistered

Figure 5: Mesos Frameworks dashboard

The Mesos slaves will make offers with their available resources to the Swarm framework as shown in Figure 6.



Offers

ID ▼	Framework	Host	CPUs	Mem
…88fb-9a884c7addea-O2416	swarm	ip-172-31-15-52.eu-central-1.compute.internal	1	2.9 GB
…88fb-9a884c7addea-O2414	swarm	ip-172-31-15-53.eu-central-1.compute.internal	2	2.9 GB
…88fb-9a884c7addea-O2413	swarm	ip-172-31-15-55.eu-central-1.compute.internal	2	2.9 GB

Figure 6: Mesos Framework Offers dashboard

The information on the offers can also be obtained through the Swarm manager by executing the following command:

```
$ docker -H tcp://0.0.0.0:2375 info
```

An example response looks like:

```
Containers: 3
Images: 12
Role: primary
Strategy: spread
Filters: affinity, health, constraint, port, dependency
Offers: 3
  Offer: 2c150bbf-d044-488e-88fb-9a884c7addea-O2278
   └ ports: 6000-31000
   └ cpus: 2
   └ mem: 2.858 GiB
   └ disk: 3.869 GiB
  Offer: 2c150bbf-d044-488e-88fb-9a884c7addea-O2279
   └ ports: 6000-31000
   └ cpus: 2
   └ mem: 2.858 GiB
   └ disk: 3.869 GiB
  Offer: 2c150bbf-d044-488e-88fb-9a884c7addea-O2280
   └ ports: 6000-31000
   └ cpus: 2
   └ mem: 2.858 GiB
   └ disk: 3.869 GiB
CPUs: 6
Total Memory: 8.575 GiB
Name: 2856087d51cf
```

Issuing commands to the Swarm manager is explained in more detail in section 5.6.2. More details on the installation of Docker Swarm using the Mesos scheduler can be found at [19].

5.4 HOWTO: Build a Docker Image

5.4.1 Creating a Dockerfile

A Docker image forms the basis of a Docker container. It is a read-only template built by executing the instructions from a Dockerfile successively. Each instruction has a format like:

```
INSTRUCTION statement
```

To build a Docker image, create a new directory containing a file named `Dockerfile`. This file will contain the instructions to build the Docker image. Besides the Dockerfile the directory could contain other files or subdirectories that are required to build the image.

The first instruction in a Dockerfile is the `FROM` instruction telling Docker which image the new image will be based on. Commonly used base images are the BusyBox, CentOS and Debian/Ubuntu images [20]. Any Docker image can serve as a base for another image. For example, if we want to base our image on Ubuntu 14.04, we would use the `ubuntu` image with version tag `14.04`. The first instruction would be:

```
FROM ubuntu:14.04
```

The next instruction in the Dockerfile usually is the `MAINTAINER` instruction which specifies who maintains the image. E.g.:

```
MAINTAINER Erika Pauwels <erika.pauwels@tenforce.com>
```

Although this instruction is not required it is advised to set a maintainer on the image so people know who to contact when they have questions.

Following the `MAINTAINER` instruction are the instructions to actually build the Docker image. Most commonly used instructions are:

- RUN: run a command in a shell. For example if you want to install Tomcat in your image, you will have to add the following lines to your Dockerfile:

```
RUN apt-get update && apt-get install -y tomcat7
```

- ADD: add files or directories to the file system of the container.
- ENV: set an environment variable in the container. Note that these environment variables can be overwritten by the user when running the Docker container.
- CMD: default instruction for an executing container. This command will be executed each time the Docker container is started. There can only be one such instruction in a Dockerfile. If you want to run multiple commands at container startup it is good practice to bundle these commands in a bash script, e.g. `startup.sh`, add the bash script to the container and use this instruction:

```
CMD ["/bin/bash", "/startup.sh"]
```

To get a list of all available instructions check the Docker documentation page [21].

5.4.2 Building an Image from a Dockerfile

To build a Docker image from a Dockerfile open a command shell and change the directory to the directory containing your Dockerfile. Next execute the command

```
$ docker build -t my-image:my-tag .
```

A new image will be built from the current directory with the name `my-image` and tagged with `my-tag`. If you do not provide a tag, `latest` will be used by default.

5.4.3 Executing a Docker Image

If the Docker build command finishes successfully the new image will be listed when executing:

```
$ docker images
```

You can now run the container via the command:

```
$ docker run --name my-container {options} my-image:my-tag
```

To get a complete overview of all available options on the Docker build command, run:

```
$ docker help build
```

5.4.4 Publishing a Docker Image

A Docker registry is a public or private service to store and distribute Docker images centrally. Images can be pulled from a repository or pushed to it. It gives team members the ability to share images. For example, one can push an image to a Docker registry which can then be pulled by other team members to deploy the image to testing, staging and production environments.

Docker provides a public Docker registry called Docker Hub which allows to publish Docker images publicly for free. Moreover Docker Hub integrates with GitHub providing an automated workflow of building and publishing a Docker image. The only things a user has to do are:

1. Setup a GitHub repository containing at least the Dockerfile. The repository might contain other files or directories required by the build process.
2. Configure an automated build on Docker Hub as described on [22].

5.5 HOWTO: Extend a General Docker Image

By the choice of Docker, the Big Data platform wants to lower the barrier to build application pipelines for the platform. Building a component for a pipeline should be as easy as extending a Docker image with your application code. The intent of the BDE project is to provide such base images for a selection of Big Data technologies. The selection will mainly depend on the pilot cases that need to be implemented.

To create a Docker image for your application, add a Dockerfile in the root folder of your application code. As explained in section 5.4 one Docker image can extend another by starting your Dockerfile with the statement:

```
FROM base-image-name
```

Next, adding your application to the Dockerfile is as simple as adding the statement:

```
ADD . /application
```

This statement will add the content of the current directory to the `/application` directory inside the Docker container. The exact location and the expected code packaging might differ per base image (e.g. Java based technologies typically expect an executable JAR file), but the documentation of the base Docker image should clearly document how the Docker image can be extended with a custom application.

5.6 HOWTO: Deploy a Single Docker Container

5.6.1 Through Marathon

Marathon provides a framework to deploy long-running containers, including Docker containers, at scale on top of Mesos. It provides a REST API for starting, stopping, and scaling applications. To create a new dockerized application send a POST REST call to `http://{marathon-uri}:{port}/v2/apps` containing a JSON body describing the container that needs to be created. Marathon runs on port 8080 by default.

```
curl -X POST http://{marathon-uri}:{port}/v2/apps \
     -d @container-description.json \
     -H "Content-type: application/json"
```

An example JSON file to launch a Virtuoso Docker container from the 'tenforce/virtuoso' image is:

```
{
  "id": "/db/virtuoso",
  "cpus": 0.5,
  "mem": 1000.0,
  "container": {
    "type": "DOCKER",
    "docker": {
      "image": "tenforce/virtuoso:virtuoso-v7.2.0-latest",
      "network": "BRIDGE",
      "parameters": [
        { "key": "env", "value": "DBA_PASSWORD=foobar" },
        { "key": "hostname", "value": "virtuoso.bde-europe.com" }
      ],
      "portMappings": [
        { "containerPort": 8890, "hostPort": 0 },
        { "containerPort": 1111, "hostPort": 0 }
      ]
    }
  }
}
```

The most important fields in this JSON are:

- id: a unique identifier for the application
- cpus: the number of CPUs the application needs per instance
- mem: the amount of memory in MB that is needed for the application per instance

A full description of the available fields in the JSON body is available at [23].

When deploying a Docker container the following fields are important in the JSON file:

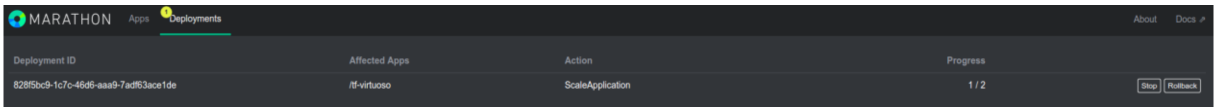
- container.type: should be 'DOCKER' when deploying a Docker container
- container.docker.image: image to use as template to create the Docker container

- container.docker.parameters: parameters to pass to the Docker container. In theory all options available on the Docker run command can be used here, however not all parameters might be supported by Mesos.
- container.docker.portMappings: exposing container ports on the host. If `hostPort` is 0 Mesos will select a random port to bind to.

When the request succeeds, Marathon will return a 200 OK containing a JSON response indicating that the application deployment request has been put in the deployment queue. E.g.:

```
{
  "id": "/tf-virtuoso",
  "container": {
    "type": "DOCKER",
    "volumes": [],
    "docker": {
      "image": "tenforce/virtuoso:virtuoso-v7.2.0-latest",
      ...
   }
  },
  ...
  "deployments": [
    {
      "id": "828f5bc9-1c7c-46d6-aaa9-7adf63ace1de"
    }
  ]
}
```

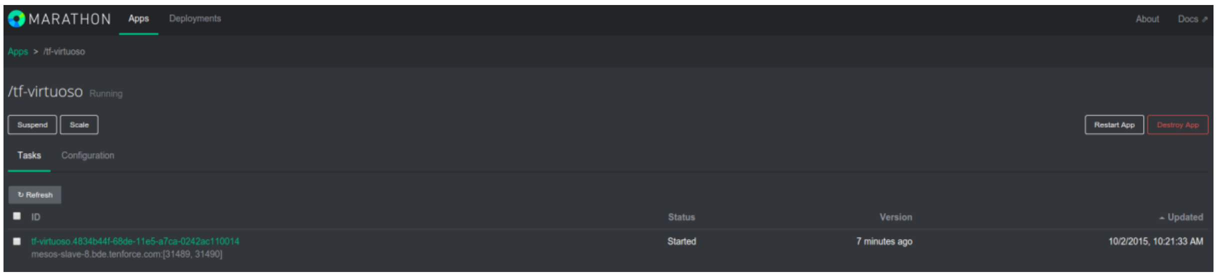
The deployment task will also appear in the Marathon UI at `http://{marathon-uri}:{port}/#/deployments` as shown in Figure 7.



Deployment ID	Affected Apps	Action	Progress
828f5bc9-1c7c-46d6-aaa9-7adf63ace1de	/tf-virtuoso	ScaleApplication	1 / 2

Figure 7: Marathon Deployments dashboard

Once the deployment succeeds, the new application will appear in the application list in the Marathon UI at `http://{marathon-uri}:{port}/#/apps` as shown in Figure 8. By clicking the application name one can see the details of the application: the slave on which the application has been deployed, the ports bound on the host, etc.



/tf-virtuoso Running

ID	Status	Version	Updated
tf-virtuoso.4834b44f-68de-11e5-a7ca-0242ac110014 mesos-slave-8.bde.tenforce.com:[31489, 31490]	Started	7 minutes ago	10/2/2015, 10:21:33 AM

Figure 8: Marathon Applications dashboard

Next to the call to deploy a dockerized application, the Marathon REST API provides calls to monitor the status of the application. All available REST calls are described in the Marathon documentation [23].

5.6.2 Through Docker Swarm

As explained in section 3.1.2 Docker Swarm offers a clustering tool which turns a pool of Docker hosts into a single virtual host. Section 5.3 describes how to setup a Docker Swarm and start a Swarm manager which is used to manage the swarm. Because Docker Swarm serves the standard Docker API, any tool that already communicates with a Docker daemon can use Swarm to transparently scale to multiple hosts.

You can now send Docker commands to the Swarm manager by executing:

```
$ docker -H <manager_ip>:<manager_port> <docker_command>
```

The `-H` option tells the Docker client which socket to connect to – in our case, this is the port of the Swarm manager. Supposing the Swarm manager listens on port 2222 and you run the Docker client on the same machine as the Swarm manager the socket to send commands to must be `tcp://0.0.0.0:2222`.

You can send the same Docker commands to the Swarm manager as you would send to the Docker Engine.

- Display system-wide information including the agents available in the Swarm:

```
$ docker -H tcp://0.0.0.0:2222 info
```

- Launch a Virtuoso Docker container on the cluster:

```
$ docker -H tcp://0.0.0.0:2222 run --name my-virtuoso -d
  tenforce/virtuoso:virtuoso-v7.2.0-latest
```

- List all running containers on the cluster:

```
$ docker -H tcp://0.0.0.0:2222 ps
```

- Remove a container from the cluster:

```
$ docker -H tcp://0.0.0.0:2222 rm my-container-name
```

More information on deploying Docker containers through Docker Swarm can be found at [24].

5.7 HOWTO: Connect Applications and Services

Running applications on top of the Big Data platform requires the deployment of a collection of Docker containers working together. For example if you want to setup a Spark cluster consisting of one Spark master and two Spark workers you may have two Docker images: a Spark master image and a Spark worker image. First you start a Spark master container. Next, you start the Spark worker containers. These workers need to register with the Spark master, but how can they establish a connection with the Spark master?

5.7.1 Through Docker Swarm

Docker provides a link option (`--link`) on its run command to wire containers together. The option has a format like `container-name:alias` meaning that the container with the name `container-name` will be known in the new container as `alias`. Docker implements this by adding an entry in the new container's `/etc/hosts` file linking the `alias` to the linked Docker container's IP address. In the example case of Spark, supposing that the Spark master container is named `my-spark-master`, you can start the Spark worker container with the link option set to `my-spark-master:spark-master`. The Spark master container will now be known as `spark-master` in the Spark worker container.

As Docker Swarm supports the standard Docker API it also supports the link option. However, the Swarm manager currently schedules all dependent Docker containers on the same node in the cluster. For the Spark example this implies that the Spark master as well as the Spark workers will all be scheduled on the same cluster node making it a rather limited single node cluster. The Docker Swarm community is currently actively working on the networking features of Swarm so in the (near) future Swarm will be able to deploy dependent Docker containers on multiple nodes in a cluster [25].

Supposing the Swarm manager listens on port 2222, you can deploy a Spark master and a connected Spark worker by running the following commands:

```
$ docker -H tcp://0.0.0.0:2222 run --name my-spark-master \
     -d bde2020/spark-master:1.5.1-hadoop2.6

$ docker -H tcp://0.0.0.0:2222 run --name my-spark-worker-1 \
     --link my-spark-master:spark-master \
     -d bde2020/spark-worker:1.5.1-hadoop2.6
```

5.7.2 Through Marathon

As already mentioned in section 3.1.1, Marathon does not support Docker's link option. As a workaround we can mimic the Docker link behaviour by providing an add-host option which will add an entry in the container's `/etc/hosts` file. The add-host option takes as format `host:ip`, so the drawback of this workaround is that you need to know the other Docker's IP address in order to establish the link. A container's IP address is only assigned at container start and might change on a restart making this workaround a non-manageable deployment strategy for the BDE platform.

To deploy a Spark worker container on a cluster through Marathon, send the following JSON file to `http://{marathon-uri}:{port}/v2/apps` as explained in section 5.6.1.

```
{
  "id": "spark-worker",
  "cpus": 1,
  "mem": 2800.0,
  "container": {
    "type": "DOCKER",
    "docker": {
      "image": "bde2020/spark-worker:1.5.1-hadoop2.6",
      "network": "BRIDGE",
      "parameters": [
        { "key": "add-host", "value": "spark-master:52.29.46.154" }
      ]
    }
  }
}
```

6. Spark Demo Application: Here I am

A small demo application which we use to evaluate the current state of the BDE platform consists of the 'Here I am' story. In this section we describe the story itself, the way we can deploy it in the current state of the platform and the way we intend it to be deployed in the future. Although this section proves the usability of the current state of the BDE platform, it also shows which improvements we intend to make over the coming months.

6.1 Application Setting

The demo-case 'Here I am' considers the location of trackers at various time intervals. We continuously monitor the location of the tracker and we intend to display a heatmap which shows where the trackers were at various time intervals. Given that this is a Big Data project, we envision having received the location of many thousands of trackers. The output will display a heatmap indicating how busy an area was at a time-interval as shown in Figure 9.

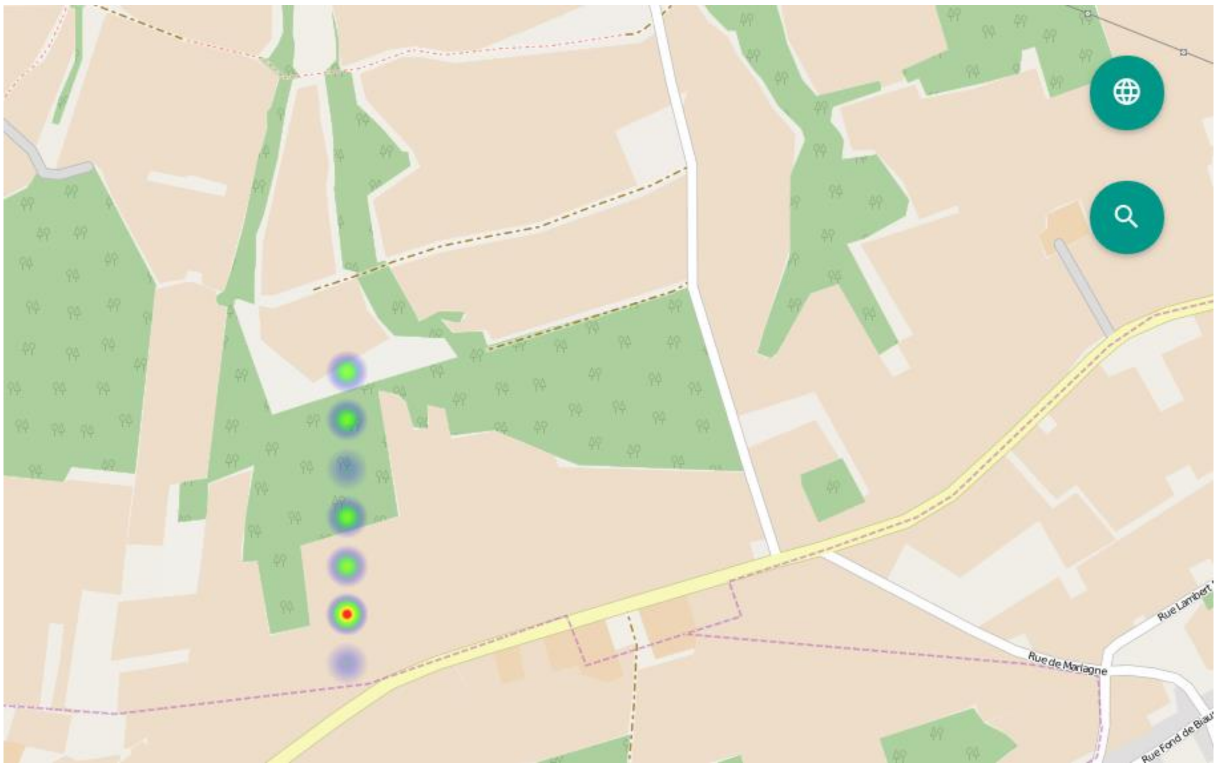


Figure 9: Screenshot: Heatmap displaying tracker locations at a specific time interval

The algorithm assumes that the content of the trackers has been received through an input pipeline and that it is available in CSV format. The algorithm will output JSON content which can be used to render the heatmap on web-capable devices. As this could comprise of billions of events, the calculation will occur offline. The output of our calculation should be small enough so that it can be sent to a web browser for final display without major modifications. This entails minimizing the amount of content in our target output so that it’s both small enough and still usable.

In our setting we receive a large set of data inputs in a simple format and are required to output the contents in a more condensed format. The condensed format will allow us to display a heatmap indicating the activity in various regions.

6.2 HOWTO: Starting the Demo Pipeline in 2015

This section describes the deployment of the Spark demo application 'Here I am', step-by-step, on the BDE platform. The application will be executed on a Spark cluster consisting of one master and two workers. The Spark Docker containers are deployed on the base platform through Marathon. Because of the limitations Marathon currently has, some workarounds are applied to get the application working.

Figure 10 shows the cluster setup of the demo application. The cluster consists of one Mesos master which manages three Mesos slaves. On the Mesos slaves a dockerized Spark cluster is installed consisting of one Spark master and two Spark workers. The Spark Dockers are deployed through Marathon. Finally the demo application is executed on the Spark cluster by extending a Spark submit Docker image with our custom demo application. Spark submit sends the application to the Spark master which on its turn will schedule the work on the Spark workers.

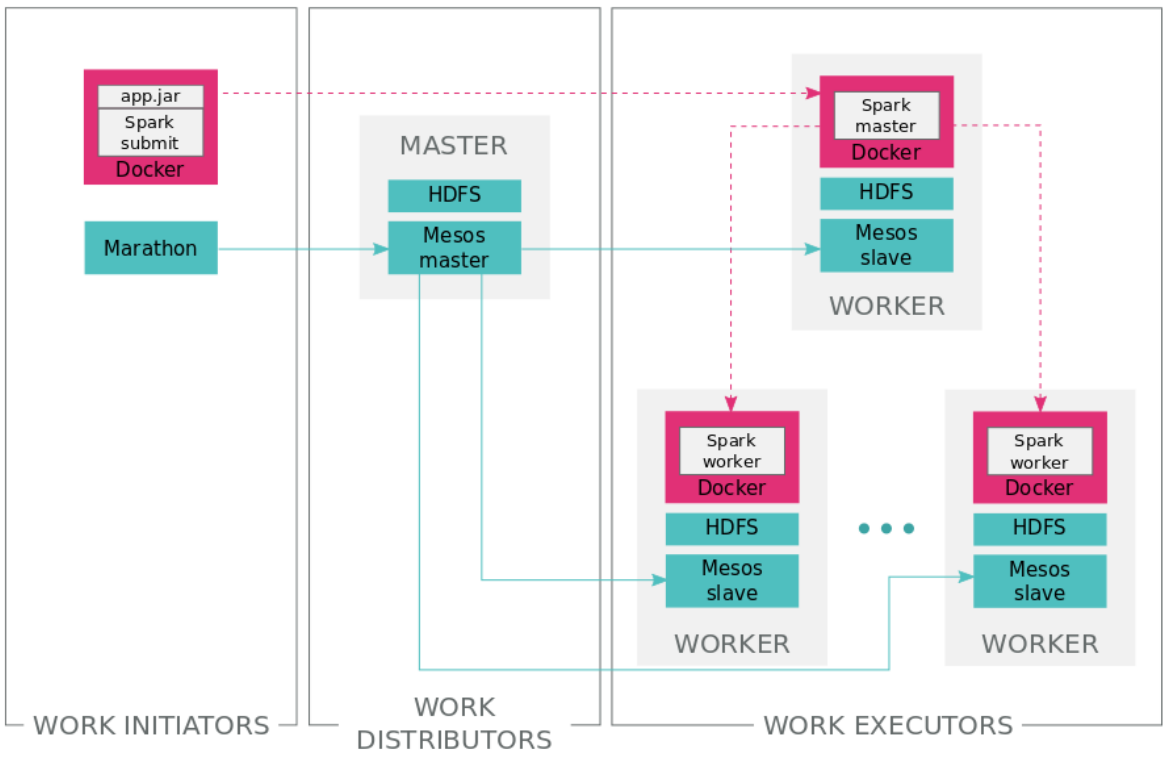


Figure 10: Cluster setup of the Spark demo application

6.2.1 Base Platform Setup

Setup the base platform with Mesos, Marathon and Docker as described in section 5.2. Make sure the Mesos slaves offer a port range of [6000-31000] by starting up the slaves with the option `--resources='ports:[6000-31000]'`. That way Docker container ports can be bound to the same ports on the host machine making the setup less complex.

6.2.2 Spark Master Setup

Next launch a Spark master container on the cluster by sending the following JSON to the Marathon REST API:

```
{
  "id": "spark-master",
  "cpus": 1,
  "mem": 2800.0,
  "container": {
    "type": "DOCKER",
    "docker": {
      "image": "bde2020/spark-master:1.5.1-hadoop2.6",
      "network": "BRIDGE",
      "parameters": [
        { "key": "hostname", "value": "spark-master" }
      ],
      "portMappings": [
        { "containerPort": 8080, "hostPort": 8080 },
        { "containerPort": 7077, "hostPort": 7077 }
      ]
    },
    "volumes": [
        {
            "containerPath": "/data/input",
            "hostPath": "/root/data/input",
            "mode": "RW"
        },
        {
            "containerPath": "/data/output",
            "hostPath": "/root/data/output",
            "mode": "RW"
        }
    ]
  }
}
```

Ports 7077 and 8080 of the Spark master container will be bound to ports 7077 and 8080 on the host machine respectively. The volumes `/data/input` and `/data/output` are mounted in the Spark master container. The `/data/input` volume should contain the input CSV (`localhost.csv`) of the demo application. Ideally this input file should be available on a shared, distributed file system like HDFS which is accessible to each of the Spark workers and master.

As explained in section 5.6.1 send the following call to the Marathon REST API to deploy the Spark master container:

```
$ curl -X POST http://{marathon-uri}:{port}/v2/apps \
     -d @spark-master.json -H "Content-type: application/json"
```

When successfully deployed the Spark master will be available on one of the Mesos slave machines. Go to the Marathon UI to check on which slave the Spark master got deployed. We will need the slave's IP address to connect the Spark worker to the Spark master in the next step.

6.2.3 Spark Worker Setup

Because Marathon cannot link Dockers using the `--link` option on container startup, we have to establish the connection from the Spark worker to the Spark master manually. Therefore, add the following entry to the `/etc/hosts` file of each Spark worker:

```
{spark-master-ip} spark-master
```

E.g.

```
52.29.46.154 spark-master
```

Next send the following JSON to the Marathon REST API to start a Spark worker.

```
{
  "id": "spark-worker",
  "cpus": 1,
  "mem": 2800.0,
  "container": {
    "type": "DOCKER",
    "docker": {
      "image": "bde2020/spark-worker:1.5.1-hadoop2.6",
      "network": "HOST"
    },
    "volumes": [
        {
            "containerPath": "/data/input",
            "hostPath": "/root/data/input",
            "mode": "RW"
        },
        {
            "containerPath": "/data/output",
            "hostPath": "/root/data/output",
            "mode": "RW"
        }
    ]
  }
}
```

By default the Spark worker runs on a randomly selected port. Because we do not know this random port beforehand, we cannot add a port publishing entry to the JSON file to publish the container port on the host. As a consequence, the Spark master will not be able to connect to the Spark worker running in the Docker container. To work around this networking issue the Spark worker container is started with the network option set to `HOST`. In that way Docker does not containerize the container's networking and the Spark worker will be directly available on the host.

The volumes `/data/input` and `/data/output` are mounted in the Spark worker container. The `/data/input` volume should contain the input CSV (`localhost.csv`) of our application. Ideally this file should be made available on a shared, distributed file system like HDFS which is accessible by each worker.

As explained in section 5.6.1 send the following call to the Marathon REST API to deploy the Spark worker container:

```
$ curl -X POST http://{marathon-uri}:{port}/v2/apps \
    -d @spark-worker.json -H "Content-type: application/json"
```

Go to the Marathon UI to follow-up the deployment of the Spark worker on the cluster. Next, scale up the application via the web interface so multiple Spark workers are running on the cluster.

6.2.4 Submit an Application to Spark

Clone the Spark demo application from the BDE GitHub repository [26]. Next build the application using Maven and Java 8.

```
$ git clone
    https://github.com/big-data-europe/demo-spark-sensor-data
$ cd /path/to/demo-spark-sensor-data
$ mvn clean package
```

A JAR will be built including all required dependency JARs in the target folder of the application. Next, submit the application on the Spark cluster by running the following command from the Spark master machine:

```
$ docker run --name spark-demo \
     --net host \
     --volume /path/to/input:/data/input \
     --volume /path/to/output:/data/output \
     --volume /path/to/demo-spark-sensor-data/target/spark-sensor-
demo-1.0-SNAPSHOT-with-dependencies.jar:/app/application.jar \
     -d bde2020/spark-submit:1.5.1-hadoop2.6 \
         --class com.tenforce.bde.spark.demo.sensors.Application \
         --master spark://spark-master:7077 \
         /app/application.jar localhost /data/input /data/output
```

The `/data/input` volume should contain the input CSV (`localhost.csv`) of our application. Ideally this file should be made available on a shared, distributed file system like HDFS which is accessible by the Spark submit container.

If the application submission succeeds, the application appears on the Spark master dashboard which is available at `http://{spark-master-uri}:8080`. Once the application finishes, the resulting JSON files are available in the `/data/output` volume of the Spark submit container.

6.3 HOWTO: Starting the Demo Pipeline in the Future

6.3.1 Base platform setup

Setup the base platform with Mesos and Docker as described in section 5.2. Setup Docker Swarm on top of Mesos and start a Swarm manager listening on port 2222 as described in section 5.3.2. Marathon does not need to be installed.

6.3.2 Spark Master Setup

Next launch a Spark master container on the cluster by sending the following command to the Swarm manager:

```
$ docker -H tcp://0.0.0.0:2222 run \
    -c 1 \
    --name spark-master \
    --hostname=spark-master \
    -p 8080:8080 -p 7077:7077 \
    -d bde2020/spark-master:1.5.1-hadoop2.6
```

6.3.3 Spark Worker Setup

Start two Spark worker containers by executing the following command:

```
$ docker -H tcp://0.0.0.0:2222 run \
    -c 1 \
    --name spark-worker \
    -p 8081:8081 -p 44848:44848 \
    --link spark-master:spark-master \
    -e SPARK_WORKER_PORT=44848 \
    -d bde2020/spark-worker:1.5.1-hadoop2.6
```

The Swarm manager will start a Spark worker which connects to the Spark master and listens on port 44848.

6.3.4 Submit an Application to Spark

Finally submit the demo application on the Spark cluster. Suppose a Docker image 'bde2020/spark-sensor-demo' is available which extends the Spark submit base image with our custom application JAR. Execute the following command to run the algorithm on the Spark cluster:

```
$ docker -H tcp://0.0.0.0:2222 run \
    --name spark-demo \
    --link spark-master:spark-master \
    -d bde2020/spark-sensor-demo:1.0 \
           localhost /data/input /data/output
```

6.3.5 Running a Pipeline through Docker Compose

In a future version of the BDE platform Docker Compose can be used to deploy a full pipeline. The Docker containers to setup the Spark cluster and to submit the application are described in a `docker-compose.yml` file. All the user needs to do to run the pipeline is:

```
$ export DOCKER_HOST=tcp://0.0.0.0:2222
$ cd {path-to-the-docker-compose-folder}
$ docker-compose up
```

7. Conclusion

The BDE platform as it is specified today shows great promise. Within the challenges provided by [2], which may require more flexibility than we originally intended, a platform arose which can be applied to many problem-domains. Some of the base technologies are under active development, but the resulting picture is becoming clearer.

The platform, as specified in this deliverable, is under active development. We have shown that Big Data computations could be made, but that the full power of the system cannot be leveraged yet with the current state of the components on which we depend. Although this case might cause worry at first sight, we place trust in the community supporting the base components.

The BDE platform selects some required base components which minimize the constraints placed on the users of the platform. The effort which was undertaken to tackle this problem, is showing promising results. As the pilot cases start presenting themselves, and as the basic technologies on which we base ourselves become mature, the platform's value will become apparent. We foresee the societal challenges to present their case in a similar pace as the backing technologies so they evolve in harmony. Some aspects of the platform may receive a more in-depth treatment as we learn from the presented use cases.

Through the challenges which unfold as the BDE project evolves, a reusable platform is being constructed. As the selected components are evolving in a harmonious way, the BDE platform steadily evolves into a multi-purpose and easy to use platform for Big Data solutions.

References

[1] Big Data Europe, "WP3 Deliverable 3.1: Assessment on Application of Generic Data Management Technologies I," 2015.

[2] Big Data Europe, "WP3 Deliverable 3.2: Technical Requirements Specification and Big Data Integrator Architectural Design I," Big Data Europe, 2015.

[3] "Apache Flink - Scalable Batch and Stream Data Processing," [Online]. Available: https://flink.apache.org/. [Accessed 03 11 2015].

[4] "MapReduce - Wikipedia," [Online]. Available: https://en.wikipedia.org/wiki/MapReduce. [Accessed 04 11 2015].

[5] "Apache Mesos," [Online]. Available: http://mesos.apache.org/. [Accessed 03 11 2015].

[6] "Docker - Build, Ship, and Run Any App, Anywhere," [Online]. Available: https://www.docker.com.

[7] "Powering Data Search, Log Analysis, Analytics | Elastic," [Online]. Available: https://www.elastic.co/products.

[8] "Docker Compose," [Online]. Available: https://www.docker.com/docker-compose. [Accessed 03 11 2015].

[9] "Docker Swarm," [Online]. Available: https://www.docker.com/docker-swarm. [Accessed 03 11 2015].

[10] "Marathon: A cluster-wide init and control system for services in cgroups or Docker containers," [Online]. Available: https://mesosphere.github.io/marathon. [Accessed 03 11 2015].

[11] "Chronos: Fault tolerant job scheduler for Mesos," [Online]. Available: https://mesos.github.io/chronos/. [Accessed 03 11 2015].

[12] "Apache ZooKeeper," [Online]. Available: https://zookeeper.apache.org/.

[13] "HDFS Architecture Guide," [Online]. Available: https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html. [Accessed 03 11 2015].

[14] "Docker Hub," [Online]. Available: https://www.docker.com/docker-hub. [Accessed 03 11 2015].

[15] "Vagrant," [Online]. Available: https://www.vagrantup.com/.

[16] "Vagrant installation manual," [Online]. Available: https://docs.vagrantup.com/v2/installation/index.html. [Accessed 03 11 2015].

[17] "VirtualBox," [Online]. Available: https://www.virtualbox.org.

[18] "Swarm Installation Manual," [Online]. Available: https://docs.docker.com/swarm/install-manual. [Accessed 23 10 2015].

[19] "GitHub Swarm Mesos cluster integration - README," [Online]. Available: https://github.com/docker/swarm/tree/master/cluster/mesos. [Accessed 18 10 2015].

[20] "Docker Base Images," [Online]. Available: https://docs.docker.com/articles/baseimages. [Accessed 18 09 2015].

[21] "Dockerfile reference," [Online]. Available: https://docs.docker.com/reference/builder. [Accessed 08 10 2015].

[22] "Automated builds on Docker Hub - Setup," [Online]. Available: https://docs.docker.com/docker-hub/builds/#to-set-up-an-automated-build. [Accessed 08 10 2015].

[23] "Marathon REST API documentation," [Online]. Available: https://mesosphere.github.io/marathon/docs/rest-api.html#post-v2-apps.

[24] "Docker Swarm installation manual - Using the Docker CLI," [Online]. Available: https://docs.docker.com/swarm/install-manual/#using-the-docker-cli. [Accessed 23 10 2015].

[25] "Docker Network Driver," [Online]. Available: https://github.com/docker/libnetwork. [Accessed 16 10 2015].

[26] "GitHub - Big Data Europe - Demo Spark Sensor Data," [Online]. Available: https://github.com/big-data-europe/demo-spark-sensor-data.